

# 00. Getting started with TensorFlow: 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/](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 [ ]:

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

2.4.1

## 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 [ ]:

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

Out [ ]:

```
<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 [ ]:

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

Out [ ]:

```
0
```

In [ ]:

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

Out [ ]:

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

In [ ]:

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

Out [ ]:

```
2
```

In [ ]:

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

Out[ ]:

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

In [ ]:

```
matrix.ndim
```

Out[ ]:

```
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 [ ]:

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

Out[ ]:

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

In [ ]:

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

Out[ ]:

```
2
```

In [ ]:

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

Out[ ]:

```
<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]]], dtype=int32)>
```

In [ ]:

```
tensor.ndim
```

Out [ ]:

```
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

$$\begin{bmatrix} 7 \\ 4 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} 7 & 4 \end{bmatrix}$$

## Matrix

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

## Tensor

$$\begin{bmatrix} \begin{bmatrix} 7 & 4 \end{bmatrix} & \begin{bmatrix} 0 & 1 \end{bmatrix} \\ \begin{bmatrix} 1 & 9 \end{bmatrix} & \begin{bmatrix} 2 & 3 \end{bmatrix} \end{bmatrix}$$

```
[ 5   1 ]
```

```
[ 5   6   8   8 ]
```

## 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), where as, tensors created with `tf.Variable()` are mutable (can be changed).

In [ ]:

```
# 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 [ ]:

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

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

In [ ]:

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

```
-----
TypeError                                         Traceback (most recent call last)
<ipython-input-14-daecfbad2415> 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 [ ]:

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

Out [ ]:

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

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

In [ ]:

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

```
-----
AttributeError                                     Traceback (most recent call last)
<ipython-input-16-3947b974feb9> in <module>()
      1 # Will error (can't change tf.constant())
----> 2 unchangeable_tensor[0].assign(7)
```

↳ unchangeable\_tensor

`AttributeError: 'tensorflow.python.framework.ops.EagerTensor' object has no attribute 'as sign'`

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

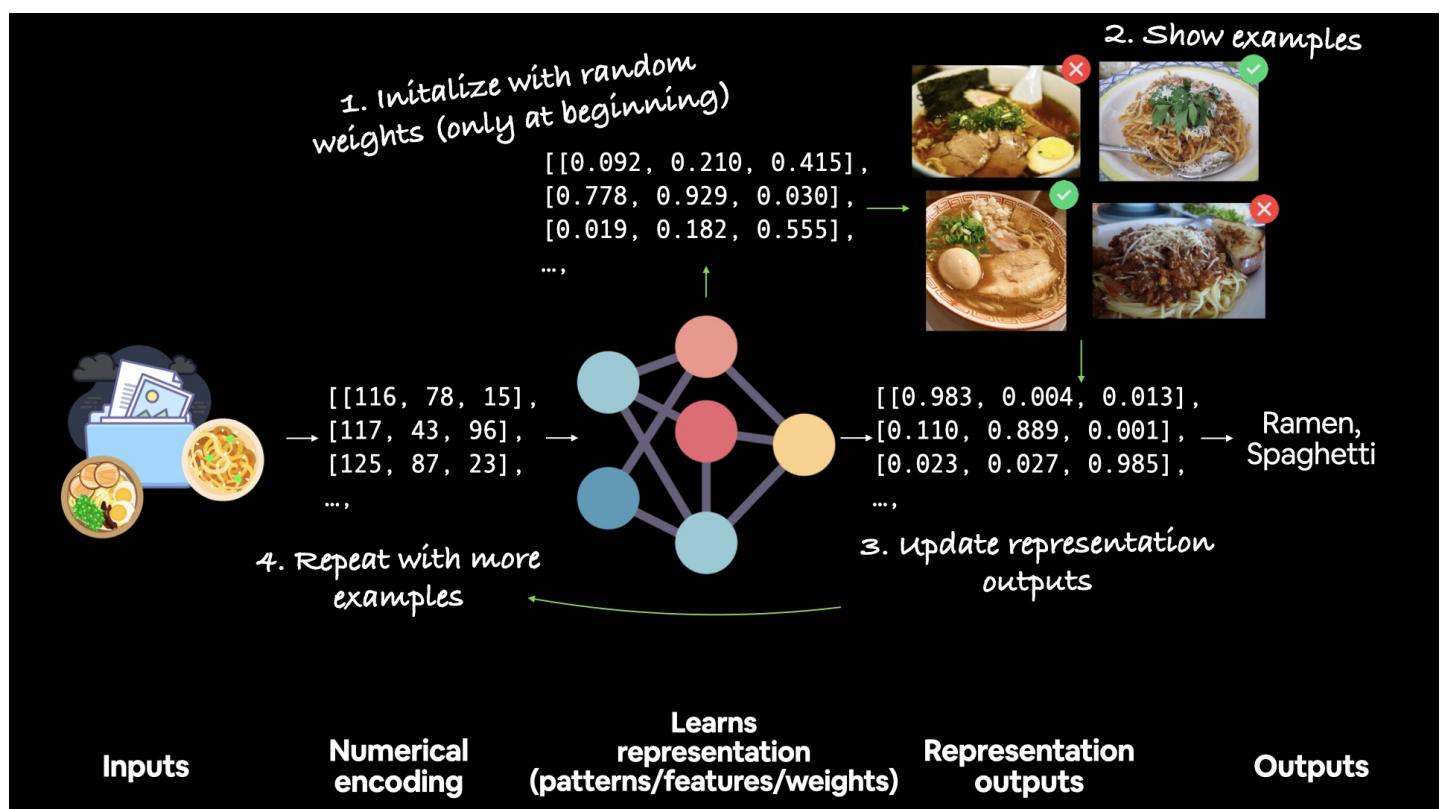
Random tensors are tensors of some arbitrary size which contain random numbers.

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 [ ]:

```
# 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?
random_1, random_2, random_1 == random_2
```

Out[ ]:

```
(<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
 [[[ 0.7565002 -0.060517021
```

```
array([[ 0.7595026, -1.2573844],  
       [-0.23193765, -1.8107855]], dtype=float32)>,  
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=  
array([[-0.7565803, -0.06854702],  
       [ 0.07595026, -1.2573844],  
       [-0.23193765, -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 [ ]:

```
# 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 [ ]:

```
(<tf.Tensor: shape=(3, 2), dtype=float32, numpy=  
array([[-0.7565803, -0.06854702],  
       [ 0.07595026, -1.2573844],  
       [-0.23193765, -1.8107855]], dtype=float32)>,  
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=  
array([[ 0.2730574, -0.29925638],  
       [-0.3652325,  0.61883307],  
       [-1.0130816,  0.2829171]], 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 working with 15,000 images of cats and dogs and the first 10,000 images of were of cats and the next 5,000 were of dogs. This order could effect 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 [ ]:

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

Out [ ]:

```
<tf.Tensor: shape=(3, 2), dtype=int32, numpy=
```

```
array([[10, 7],  
       [2, 5],  
       [3, 4]], dtype=int32)>
```

In [ ]:

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

Out[ ]:

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

**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 [ ]:

```
# Shuffle in the same order every time  
  
# Set the global random seed  
tf.random.set_seed(42)  
  
# Set the operation random seed  
tf.random.shuffle(not_shuffled, seed=42)
```

Out[ ]:

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

In [ ]:

```
# 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[ ]:

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

## 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 [ ]:

```
# Make a tensor of all ones
tf.ones(shape=(3, 2))
```

Out [ ]:

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

In [ ]:

```
# Make a tensor of all zeros
tf.zeros(shape=(3, 2))
```

Out [ ]:

```
<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 [ ]:

```
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,
                shape=[2, 4, 3]) # note: the shape total (2*4*3) has to match the number
                     # of elements in the array
numpy_A, A
```

Out [ ]:

```
(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], dtype=int32),
 <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]]], dtype=int32)>)
```

## 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 [ ]:

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

Out[ ]:

```
<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.]]]], dtype=float32)>
```

In [ ]:

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

Out[ ]:

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

In [ ]:

```
# 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).numpy()) # .numpy() converts to NumPy array
```

```
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): 120
```

You can also index tensors just like Python lists.

In [ ]:

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

In [ ]:

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

In [ ]:

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

Out [ ]:

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

In [ ]:

```
# 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 [ ]:

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

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

In [ ]:

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

Out [ ]:

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

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

In [ ]:

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

Out [ ]:

```
<tf.Tensor: shape=(2, 2, 1), dtype=int32, numpy=
```

```
array([[10],  
       [ 7],  
  
       [[ 3],  
        [ 4]]], dtype=int32)>
```

## 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 [ ]:

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

Out[ ]:

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

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

In [ ]:

```
# Original tensor unchanged  
tensor
```

Out[ ]:

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

Other operators also work.

In [ ]:

```
# Multiplication (known as element-wise multiplication)  
tensor * 10
```

Out[ ]:

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

In [ ]:

```
# Subtraction  
tensor - 10
```

Out[ ]:

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

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

advantage of being sped up later down the line when running as part of a [TensorFlow graph](#).

In [ ]:

```
# Use the tensorflow function equivalent of the '*' (multiply) operator
tf.multiply(tensor, 10)
```

Out[ ]:

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

In [ ]:

```
# The original tensor is still unchanged
tensor
```

Out[ ]:

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

## 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 [ ]:

```
# Matrix multiplication in TensorFlow
print(tensor)
tf.matmul(tensor, tensor)
```

```
tf.Tensor(
[[10 7]
 [3 4]], shape=(2, 2), dtype=int32)
```

Out[ ]:

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

In [ ]:

```
# Matrix multiplication with Python operator '@'
tensor @ tensor
```

Out[ ]:

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

**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 [ ]:

```
# 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[ ]:

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

In [ ]:

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

```
-----
InvalidArgumentError                                     Traceback (most recent call last)
<ipython-input-43-62e1e4702ffd> in <module>()
      1 # Try to matrix multiply them (will error)
----> 2 X @ Y

/usr/local/lib/python3.7/dist-packages/tensorflow/python/ops/math_ops.py in binary_op_wrapper(x, y)
    1162     with ops.name_scope(None, op_name, [x, y]) as name:
    1163         try:
-> 1164             return func(x, y, name=name)
    1165         except (TypeError, ValueError) as e:
    1166             # Even if dispatching the op failed, the RHS may be a tensor aware

/usr/local/lib/python3.7/dist-packages/tensorflow/python/util/dispatch.py in wrapper(*args, **kwargs)
    199     """Call target, and fall back on dispatchers if there is a TypeError."""
   200     try:
--> 201         return target(*args, **kwargs)
   202     except (TypeError, ValueError):
   203         # Note: convert_to_eager_tensor currently raises a ValueError, not a

/usr/local/lib/python3.7/dist-packages/tensorflow/python/ops/math_ops.py in matmul(a, b,
transpose_a, transpose_b, adjoint_a, adjoint_b, a_is_sparse, b_is_sparse, name)
    3313     else:
    3314         return gen_math_ops.mat_mul(
-> 3315             a, b, transpose_a=transpose_a, transpose_b=transpose_b, name=name)
    3316
    3317

/usr/local/lib/python3.7/dist-packages/tensorflow/python/ops/gen_math_ops.py in mat_mul(a, b, transpose_a, transpose_b, name)
    5530     return _result
    5531     except _core._NotOkStatusException as e:
-> 5532         _ops.raise_from_not_ok_status(e, name)
    5533     except _core._FallbackException:
    5534         pass

/usr/local/lib/python3.7/dist-packages/tensorflow/python/framework/ops.py in raise_from_n
```

```

ot_ok_status(e, name)
6860     message = e.message + (" name: " + name if name is not None else "")
6861     # pylint: disable=protected-access
-> 6862     six.raise_from(core._status_to_exception(e.code, message), None)
6863     # pylint: enable=protected-access
6864

```

/usr/local/lib/python3.7/dist-packages/six.py in raise\_from(value, from\_value)

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

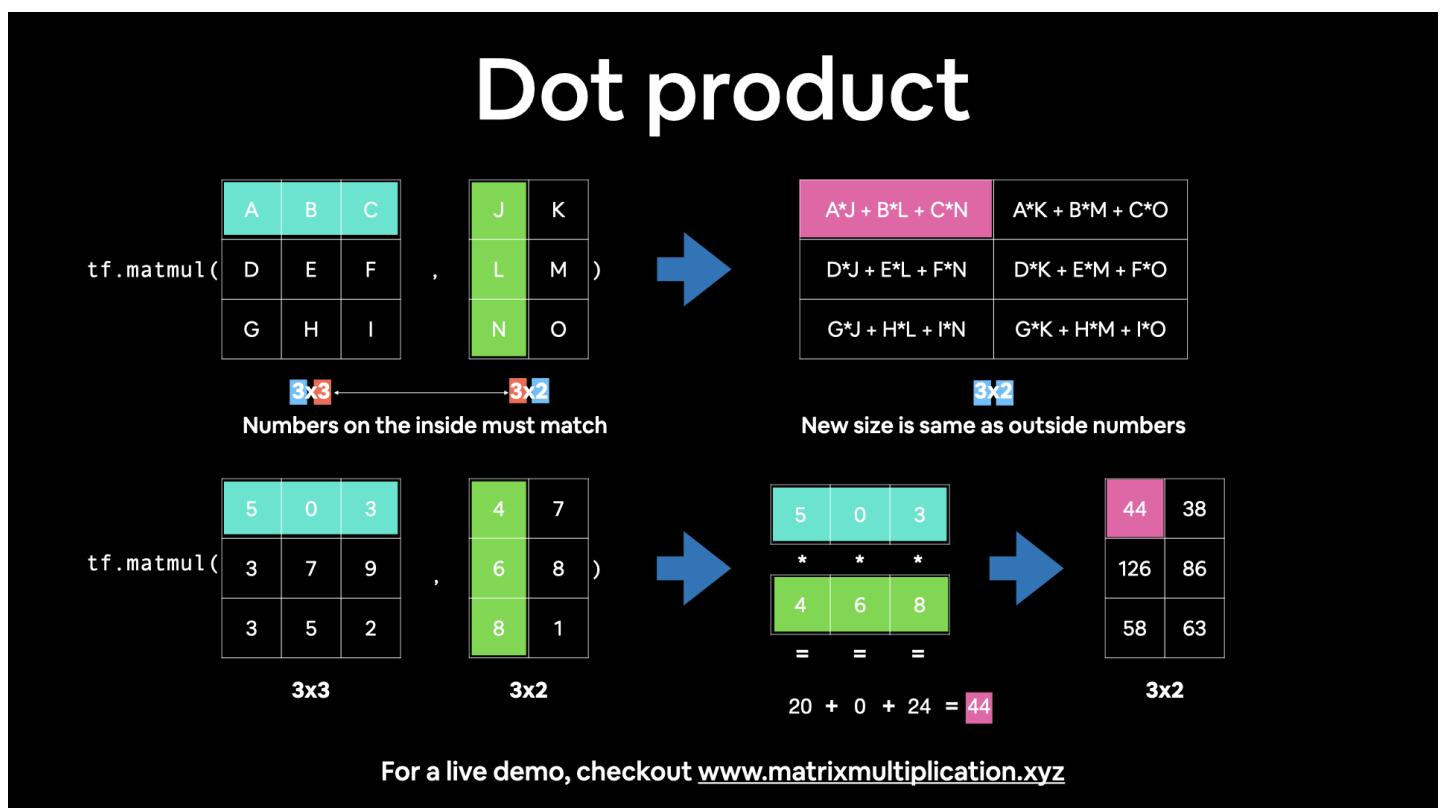
Trying to matrix multiply two tensors with the shape  $(3, 2)$  errors because the inner dimensions don't match.

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 [ ]:

```
# Example of reshape (3, 2) -> (2, 3)
tf.reshape(Y, shape=(2, 3))
```

Out[ ]:

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

In [ ]:

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

Out[ ]:

```
<tf.Tensor: shape=(3, 3), dtype=int32, numpy=
```

```
array([[ 27,  30,  33],  
       [ 61,  68,  75],  
       [ 95, 106, 117]], dtype=int32)>
```

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

In [ ]:

```
# Example of transpose (3, 2) -> (2, 3)  
tf.transpose(X)
```

Out[ ]:

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

In [ ]:

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

Out[ ]:

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

In [ ]:

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

Out[ ]:

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

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

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 [ ]:

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

Out[ ]:

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

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 [ ]:

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

Out[ ]:

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

In [ ]:

```
# Perform matrix multiplication between X and Y (reshaped)
tf.matmul(X, tf.reshape(Y, (2, 3)))
```

Out[ ]:

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

Hmm... they result in different values.

Which is strange because when dealing with `Y` (a `(3x2)` matrix), reshaping to `(2, 3)` and transposing it result in the same shape.

In [ ]:

```
# Check shapes of Y, reshaped Y and tranposed Y
Y.shape, tf.reshape(Y, (2, 3)).shape, tf.transpose(Y).shape
```

Out[ ]:

```
(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 [ ]:

```
# 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  $Y^T$  (note the capital T for tranpose).
- 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

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 & 1 \\ 6 & 7 & 1 \end{bmatrix}$$

## 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 [ ]:

```
# 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
```

Out [ ]:

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

In [ ]:

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

Out [ ]:

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

In [ ]:

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

Out [ ]:

```
<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()`.

In [ ]:

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

Out [ ]:

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

In [ ]:

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

Out [ ]:

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

## 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 [ ]:

```
# 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[ ]:

```
<tf.Tensor: shape=(50,), dtype=int64, numpy=
array([79, 33, 64, 90, 16, 3, 70, 51, 27, 54, 71, 65, 4, 8, 78, 84, 11,
       23, 15, 2, 55, 74, 26, 58, 9, 74, 33, 49, 48, 4, 30, 41, 69, 17,
      39, 37, 3, 63, 13, 37, 72, 80, 93, 29, 61, 82, 85, 40, 79, 14])>
```

In [ ]:

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

Out[ ]:

```
<tf.Tensor: shape=(), dtype=int64, numpy=2>
```

In [ ]:

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

Out[ ]:

```
<tf.Tensor: shape=(), dtype=int64, numpy=93>
```

In [ ]:

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

Out[ ]:

```
<tf.Tensor: shape=(), dtype=int64, numpy=45>
```

In [ ]:

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

Out[ ]:

```
<tf.Tensor: shape=(), dtype=int64, numpy=2262>
```

You can also find the standard deviation ([tf.reduce\\_std\(\)](#)) and variance ([tf.reduce\\_variance\(\)](#)) of elements in a tensor using similar methods.

## 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 [ ]:

```
# Create a tensor with 50 values between 0 and 1
```

```
F = tf.constant(np.random.random(50))
```

```
F
```

```
Out[ ]:
```

```
<tf.Tensor: shape=(50,), dtype=float64, numpy=  
array([0.55725069, 0.66589953, 0.51717871, 0.23663905, 0.03861408,  
       0.25753377, 0.58184723, 0.05496164, 0.97498269, 0.52503693,  
       0.68152575, 0.8151567 , 0.25854921, 0.15479351, 0.43657365,  
       0.34710353, 0.18739747, 0.7859743 , 0.11497802, 0.68242342,  
       0.74991327, 0.16043629, 0.5636553 , 0.4110329 , 0.62245869,  
       0.58129871, 0.44792704, 0.91472587, 0.76207828, 0.36502321,  
       0.96601855, 0.03094982, 0.53464709, 0.73241468, 0.0856144 ,  
       0.22923332, 0.66802859, 0.68604535, 0.53128266, 0.48685057,  
       0.53320669, 0.38020824, 0.81706951, 0.33855374, 0.94007405,  
       0.05419892, 0.10267899, 0.6020753 , 0.78038818, 0.87655176])>
```

```
In [ ]:
```

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

```
Out[ ]:
```

```
<tf.Tensor: shape=(), dtype=int64, numpy=8>
```

```
In [ ]:
```

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

```
Out[ ]:
```

```
<tf.Tensor: shape=(), dtype=int64, numpy=31>
```

```
In [ ]:
```

```
# Find the maximum element position of F  
print(f"The maximum value of F is at position: {tf.argmax(F).numpy() }")  
print(f"The maximum value of F is: {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() == tf.  
reduce_max(F).numpy() }")
```

```
The maximum value of F is at position: 8
```

```
The maximum value of F is: 0.9749826885110175
```

```
Using tf.argmax() to index F, the maximum value of F is: 0.9749826885110175
```

```
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 [ ]:
```

```
# 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[ ]:
```

```
(TensorShape([1, 1, 1, 1, 50]), 5)
```

```
In [ ]:
```

```
# Squeeze tensor G (remove all 1 dimensions)  
G_squeezed = tf.squeeze(G)
```

```
G_squeezed.shape, G_squeezed.ndim
```

Out [ ]:

```
(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 [ ]:

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

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

Out [ ]:

```
<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 [ ]:

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

Out [ ]:

```
<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 [ ]:

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

Out [ ]:

```
<tf.Tensor: shape=(9,), dtype=int64, numpy=array([1, 2, 3, 4, 5, 6, 7, 8, 9])>
```

In [ ]:

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

```
Out[ ]:
```

```
<tf.Tensor: shape=(9,), dtype=int64, numpy=array([ 1,  4,  9, 16, 25, 36, 49, 64, 81])>
```

```
In [ ]:
```

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

```
-----
InvalidArgumentError                                     Traceback (most recent call last)
<ipython-input-74-d7db039da8bb> in <module>()
      1 # Find the squareroot (will error), needs to be non-integer
----> 2 tf.sqrt(H)

/usr/local/lib/python3.7/dist-packages/tensorflow/python/util/dispatch.py in wrapper(*args, **kwargs)
    199     """Call target, and fall back on dispatchers if there is a TypeError."""
   200     try:
--> 201         return target(*args, **kwargs)
   202     except (TypeError, ValueError):
   203         # Note: convert_to_eager_tensor currently raises a ValueError, not a

/usr/local/lib/python3.7/dist-packages/tensorflow/python/ops/math_ops.py in sqrt(x, name)
  4901     A `tf.Tensor` of same size, type and sparsity as `x`.
  4902     """
-> 4903     return gen_math_ops.sqrt(x, name)
  4904
  4905

/usr/local/lib/python3.7/dist-packages/tensorflow/python/ops/gen_math_ops.py in sqrt(x, name)
10035     return _result
10036     except _core._NotOkStatusException as e:
-> 10037         _ops.raise_from_not_ok_status(e, name)
10038     except _core._FallbackException:
10039         pass

/usr/local/lib/python3.7/dist-packages/tensorflow/python/framework/ops.py in raise_from_not_ok_status(e, name)
  6860     message = e.message + (" name: " + name if name is not None else "")
  6861     # pylint: disable=protected-access
-> 6862     six.raise_from(core._status_to_exception(e.code, message), None)
  6863     # pylint: enable=protected-access
  6864

/usr/local/lib/python3.7/dist-packages/six.py in raise_from(value, from_value)
```

```
InvalidArgumentError: Value for attr 'T' of int64 is not in the list of allowed values: b
float16, half, float, double, complex64, complex128
; NodeDef: {{node Sqrt}}; Op<name=Sqrt; signature=x:T -> y:T; attr=T:type,allowed=[DT_BF
LOAT16, DT_HALF, DT_FLOAT, DT_DOUBLE, DT_COMPLEX64, DT_COMPLEX128]> [Op:Sqrt]
```

```
In [ ]:
```

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

```
Out[ ]:
```

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

```
In [ ]:
```

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

```
Out[ ]:
```

```
<tf.Tensor: shape=(9,), dtype=float32, numpy=
```

```
array([1.          , 1.4142135, 1.7320508, 2.          , 2.236068 , 2.4494898,
       2.6457512, 2.828427 , 3.          ], dtype=float32)>
```

In [ ]:

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

Out[ ]:

```
<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 [ ]:

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

Out[ ]:

```
<tf.Variable 'Variable:0' shape=(5,) dtype=int64, numpy=array([0, 1, 2, 3, 4])>
```

In [ ]:

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

Out[ ]:

```
<tf.Variable 'UnreadVariable' shape=(5,) dtype=int64, numpy=array([ 0,  1,  2,  3, 50])>
```

In [ ]:

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

Out[ ]:

```
<tf.Variable 'Variable:0' shape=(5,) dtype=int64, numpy=array([ 0,  1,  2,  3, 50])>
```

In [ ]:

```
# Add 10 to every element in I
I.assign_add([10, 10, 10, 10, 10])
```

Out[ ]:

```
<tf.Variable 'UnreadVariable' shape=(5,) dtype=int64, numpy=array([10, 11, 12, 13, 60])>
```

In [ ]:

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

Out[ ]:

```
<tf.Variable 'Variable:0' shape=(5,) dtype=int64, 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 [ ]:

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

Out[ ]:

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

In [ ]:

```
# Convert tensor J to NumPy with np.array()
np.array(J), type(np.array(J))
```

Out[ ]:

```
(array([ 3.,  7., 10.]), numpy.ndarray)
```

In [ ]:

```
# Convert tensor J to NumPy with .numpy()
J.numpy(), type(J.numpy())
```

Out[ ]:

```
(array([ 3.,  7., 10.]), numpy.ndarray)
```

By default tensors have `dtype=float32`, where as 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 [ ]:

```
# 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
```

Out[ ]:

```
(tf.float64, tf.float32)
```

## 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 [ ]:

```
# 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[ ]:

```
<tf.Tensor: shape=(10,), dtype=int64, numpy=array([ 10,  12,  16,  22,  30,  40,  52,  66,
, 82, 100])>
```

In [ ]:

```
# 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[ ]:

```
<tf.Tensor: shape=(10,), dtype=int64, numpy=array([ 10,  12,  16,  22,  30,  40,  52,  66,
, 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 [ ]:

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

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

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 [ ]:

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

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

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 [ ]:

```
!nvidia-smi
```

Mon Mar 15 22:33:50 2021

```
+-----+  
| NVIDIA-SMI 460.56      Driver Version: 460.32.03    CUDA Version: 11.2      |  
+-----+-----+-----+  
| GPU  Name      Persistence-M| Bus-Id      Disp.A  | Volatile Uncorr. ECC | | | |
| Fan  Temp     Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |  
|          |          |          |          |          |          MIG M. |  
+=====+=====+=====+=====+=====+=====+=====+  
|  0  Tesla T4        Off  | 00000000:00:04.0 Off |                0 | | | |
| N/A   52C     P0    29W /  70W |    224MiB / 15109MiB |      0%     Default |  
|          |          |          |          |          |          N/A |  
+-----+-----+-----+-----+-----+-----+  
  
+-----+  
| Processes:  
| GPU  GI  CI      PID  Type  Process name          GPU Memory |  
|          ID  ID  
+=====+=====+=====+=====+=====+=====+=====+  

```

▀ 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.

## ▀ Extra-curriculum

- 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.

# 01. Neural Network Regression with TensorFlow

There are many definitions for a [regression problem](#) but in our case, we're going to simplify it to be: predicting a number.

For example, you might want to:

- Predict the selling price of houses given information about them (such as number of rooms, size, number of bathrooms).
- Predict the coordinates of a bounding box of an item in an image.
- Predict the cost of medical insurance for an individual given their demographics (age, sex, gender, race).

In this notebook, we're going to set the foundations for how you can take a sample of inputs (this is your data), build a neural network to discover patterns in those inputs and then make a prediction (in the form of a number) based on those inputs.

## What we're going to cover

Specifically, we're going to go through doing the following with TensorFlow:

- Architecture of a regression model
- Input shapes and output shapes
  - $X$  : features/data (inputs)
  - $Y$  : labels (outputs)
- Creating custom data to view and fit
- Steps in modelling
  - Creating a model
  - Compiling a model
    - Defining a loss function
    - Setting up an optimizer
    - Creating evaluation metrics
  - Fitting a model (getting it to find patterns in our data)
- Evaluating a model
  - Visualizing the model ("visualize, visualize, visualize")
  - Looking at training curves
  - Compare predictions to ground truth (using our evaluation metrics)
- Saving a model (so we can use it later)
- Loading a model

Don't worry if none of these make sense now, we're going to go through each.

## How you can use this notebook

You can read through the descriptions and the code (it should all run), but there's a better option.

Write all of the code yourself.

Yes. I'm serious. Create a new notebook, and rewrite each line by yourself. Investigate it, see if you can break it, why does it break?

You don't have to write the text descriptions but writing the code yourself is a great way to get hands-on experience.

Don't worry if you make mistakes, we all do. The way to get better and make less mistakes is to **write more code**.

# Typical architecture of a regression neural network

The word *typical* is on purpose.

Why?

Because there are many different ways (actually, there's almost an infinite number of ways) to write neural networks.

But the following is a generic setup for ingesting a collection of numbers, finding patterns in them and then outputting some kind of target number.

Yes, the previous sentence is vague but we'll see this in action shortly.

Hyperparameter	Typical value
Input layer shape	Same shape as number of features (e.g. 3 for # bedrooms, # bathrooms, # car spaces in housing price prediction)
Hidden layer(s)	Problem specific, minimum = 1, maximum = unlimited
Neurons per hidden layer	Problem specific, generally 10 to 100
Output layer shape	Same shape as desired prediction shape (e.g. 1 for house price)
Hidden activation	Usually <a href="#">ReLU</a> (rectified linear unit)
Output activation	None, ReLU, logistic/tanh
Loss function	<a href="#">MSE</a> (mean square error) or <a href="#">MAE</a> (mean absolute error)/Huber (combination of MAE/MSE) if outliers
Optimizer	<a href="#">SGD</a> (stochastic gradient descent), <a href="#">Adam</a>

**Table 1:** Typical architecture of a regression network. Source: Adapted from page 293 of [Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow Book by Aurélien Géron](#)

Again, if you're new to neural networks and deep learning in general, much of the above table won't make sense. But don't worry, we'll be getting hands-on with all of it soon.

**Note:** A hyperparameter in machine learning is something a data analyst or developer can set themselves, whereas a parameter usually describes something a model learns on its own (a value not explicitly set by an analyst).

Okay, enough talk, let's get started writing code.

To use TensorFlow, we'll import it as the common alias `tf` (short for TensorFlow).

In [ ]:

```
import tensorflow as tf
print(tf.__version__) # check the version (should be 2.x+)
```

2.3.0

## Creating data to view and fit

Since we're working on a **regression problem** (predicting a number) let's create some linear data (a straight line) to model.

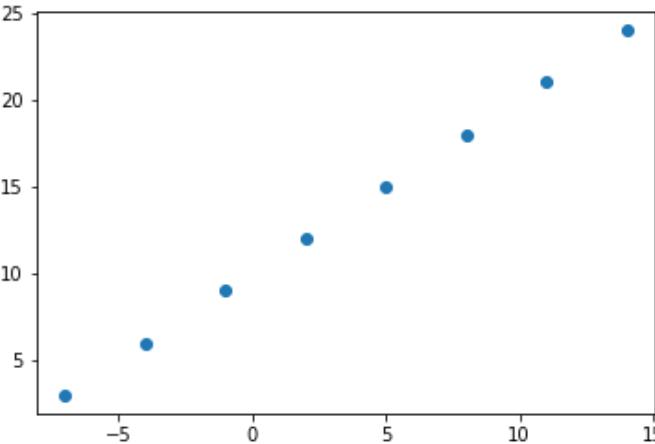
In [ ]:

```
import numpy as np
import matplotlib.pyplot as plt

# Create features
X = np.array([-7.0, -4.0, -1.0, 2.0, 5.0, 8.0, 11.0, 14.0])

# Create labels
y = np.array([3.0, 6.0, 9.0, 12.0, 15.0, 18.0, 21.0, 24.0])
```

```
# Visualize it  
plt.scatter(X, y);
```



Before we do any modelling, can you calculate the pattern between `X` and `Y`?

For example, say I asked you, based on this data what the `Y` value would be if `X` was 17.0?

Or how about if `X` was -10.0?

This kind of pattern discover is the essence of what we'll be building neural networks to do for us.

## Regression input shapes and output shapes

One of the most important concepts when working with neural networks are the input and output shapes.

The **input shape** is the shape of your data that goes into the model.

The **output shape** is the shape of your data you want to come out of your model.

These will differ depending on the problem you're working on.

Neural networks accept numbers and output numbers. These numbers are typically represented as tensors (or arrays).

Before, we created data using NumPy arrays, but we could do the same with tensors.

In [ ]:

```
# Example input and output shapes of a regression model  
house_info = tf.constant(["bedroom", "bathroom", "garage"])  
house_price = tf.constant([939700])  
house_info, house_price
```

Out [ ]:

```
(<tf.Tensor: shape=(3,), dtype=string, numpy=array([b'bedroom', b'bathroom', b'garage']),  
dtype=object)>,  
<tf.Tensor: shape=(1,), dtype=int32, numpy=array([939700], dtype=int32)>)
```

In [ ]:

```
house_info.shape
```

Out [ ]:

```
TensorShape([3])
```

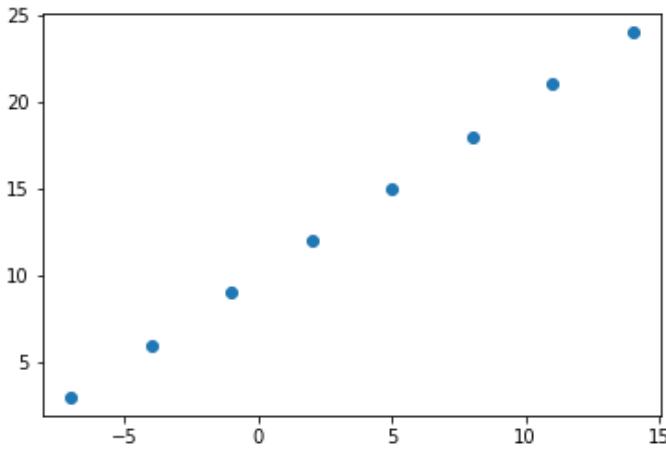
In [ ]:

```
import numpy as np  
import matplotlib.pyplot as plt
```

```
# Create features (using tensors)
X = tf.constant([-7.0, -4.0, -1.0, 2.0, 5.0, 8.0, 11.0, 14.0])

# Create labels (using tensors)
y = tf.constant([3.0, 6.0, 9.0, 12.0, 15.0, 18.0, 21.0, 24.0])

# Visualize it
plt.scatter(X, y);
```



Our goal here will be to use `X` to predict `y`.

So our **input** will be `X` and our **output** will be `y`.

Knowing this, what do you think our input and output shapes will be?

Let's take a look.

In [ ]:

```
# Take a single example of X
input_shape = X[0].shape

# Take a single example of y
output_shape = y[0].shape

input_shape, output_shape # these are both scalars (no shape)
```

Out[ ]:

```
(TensorShape([]), TensorShape([]))
```

Huh?

From this it seems our inputs and outputs have no shape?

How could that be?

It's because no matter what kind of data we pass to our model, it's always going to take as input and return as output some kind of tensor.

But in our case because of our dataset (only 2 small lists of numbers), we're looking at a special kind of tensor, more specifically a rank 0 tensor or a scalar.

In [ ]:

```
# Let's take a look at the single examples individually
X[0], y[0]
```

Out[ ]:

```
(<tf.Tensor: shape=(), dtype=float32, numpy=-7.0>,
 <tf.Tensor: shape=(), dtype=float32, numpy=3.0>)
```

In our case, we're trying to build a model to predict the pattern between `X[0]` equalling `-7.0` and `y[0]`

equalling 3.0.

So now we get our answer, we're trying to use 1  $x$  value to predict 1  $y$  value.

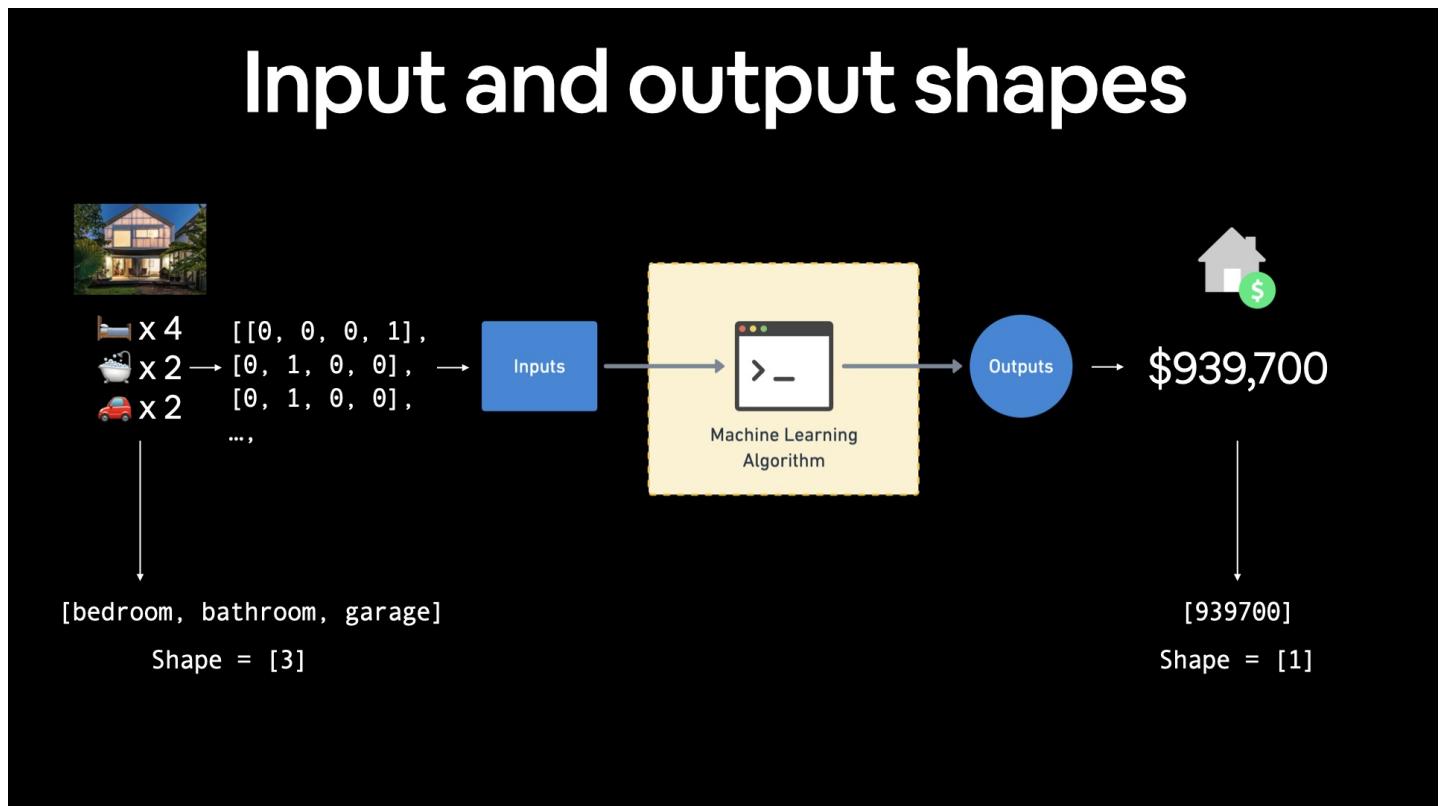
You might be thinking, "this seems pretty complicated for just predicting a straight line...".

And you'd be right.

But the concepts we're covering here, the concepts of input and output shapes to a model are fundamental.

In fact, they're probably two of the things you'll spend the most time on when you work with neural networks: making sure your input and outputs are in the correct shape .

If it doesn't make sense now, we'll see plenty more examples later on (soon you'll notice the input and output shapes can be almost anything you can imagine).



If you were working on building a machine learning algorithm for predicting housing prices, your inputs may be number of bedrooms, number of bathrooms and number of garages, giving you an input shape of 3 (3 different features). And since you're trying to predict the price of the house, your output shape would be 1.

## Steps in modelling with TensorFlow

Now we know what data we have as well as the input and output shapes, let's see how we'd build a neural network to model it.

In TensorFlow, there are typically 3 fundamental steps to creating and training a model.

1. **Creating a model** - piece together the layers of a neural network yourself (using the [Functional](#) or [Sequential API](#)) or import a previously built model (known as transfer learning).
2. **Compiling a model** - defining how a models performance should be measured (loss/metrics) as well as defining how it should improve (optimizer).
3. **Fitting a model** - letting the model try to find patterns in the data (how does  $x$  get to  $y$ ).

Let's see these in action using the [Keras Sequential API](#) to build a model for our regression data. And then we'll step through each.

In [ ]:

```
# Set random seed
tf.random.set_seed(42)
```

```
# Create a model using the Sequential API
model = tf.keras.Sequential([
    tf.keras.layers.Dense(1)
])

# Compile the model
model.compile(loss=tf.keras.losses.mae, # mae is short for mean absolute error
              optimizer=tf.keras.optimizers.SGD(), # SGD is short for stochastic gradient descent
              metrics=["mae"])

# Fit the model
model.fit(X, y, epochs=5)
```

```
Epoch 1/5
1/1 [=====] - 0s 1ms/step - loss: 11.5048 - mae: 11.5048
Epoch 2/5
1/1 [=====] - 0s 1ms/step - loss: 11.3723 - mae: 11.3723
Epoch 3/5
1/1 [=====] - 0s 887us/step - loss: 11.2398 - mae: 11.2398
Epoch 4/5
1/1 [=====] - 0s 3ms/step - loss: 11.1073 - mae: 11.1073
Epoch 5/5
1/1 [=====] - 0s 2ms/step - loss: 10.9748 - mae: 10.9748
```

Out[ ]:

```
<tensorflow.python.keras.callbacks.History at 0x7fa239e64be0>
```

**Boom!**

We've just trained a model to figure out the patterns between `X` and `y`.

How do you think it went?

In [ ]:

```
# Check out X and y
X, y
```

Out[ ]:

```
(<tf.Tensor: shape=(8,), dtype=float32, numpy=array([-7., -4., -1., 2., 5., 8., 11., 14.], dtype=float32)>,
 <tf.Tensor: shape=(8,), dtype=float32, numpy=array([ 3.,  6.,  9., 12., 15., 18., 21., 24.], dtype=float32)>)
```

What do you think the outcome should be if we passed our model an `X` value of 17.0?

In [ ]:

```
# Make a prediction with the model
model.predict([17.0])
```

Out[ ]:

```
array([[12.716021]], dtype=float32)
```

It doesn't go very well... it should've output something close to 27.0.

Question: What's Keras? I thought we were working with TensorFlow but every time we write TensorFlow code, keras comes after `tf` (e.g. `tf.keras.layers.Dense()`)?

Before TensorFlow 2.0+, [Keras](#) was an API designed to be able to build deep learning models with ease. Since TensorFlow 2.0+, its functionality has been tightly integrated within the TensorFlow library.

## Improving a model

How do you think you'd improve upon our current model?

If you guessed by tweaking some of the things we did above, you'd be correct.

To improve our model, we alter almost every part of the 3 steps we went through before.

1. **Creating a model** - here you might want to add more layers, increase the number of hidden units (also called neurons) within each layer, change the activation functions of each layer.
2. **Compiling a model** - you might want to choose optimization function or perhaps change the **learning rate** of the optimization function.
3. **Fitting a model** - perhaps you could fit a model for more **epochs** (leave it training for longer) or on more data (give the model more examples to learn from).

# Improving a model

(from a model's perspective)

```
# 1. Create the model (specified to your problem)
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(4, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])

# 2. Compile the model
model.compile(loss=tf.keras.losses.BinaryCrossentropy(),
              optimizer=tf.keras.optimizers.Adam(lr=0.001),
              metrics=["accuracy"])

# 3. Fit the model
model.fit(X_train_subset, y_train_subset, epochs=5)
```

Smaller model

```
# 1. Create the model (specified to your problem)
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])

# 2. Compile the model
model.compile(loss=tf.keras.losses.BinaryCrossentropy(),
              optimizer=tf.keras.optimizers.Adam(lr=0.0001),
              metrics=["accuracy"])

# 3. Fit the model
model.fit(X_train_full, y_train_full, epochs=100)
```

Larger model

## Common ways to improve a deep model:

- Adding layers
- Increase the number of hidden units
- Change the activation functions
- Change the optimization function
- Change the learning rate (because you can alter each of these, they're hyperparameters)
- Fitting on more data
- Fitting for longer

There are many different ways to potentially improve a neural network. Some of the most common include: increasing the number of layers (making the network deeper), increasing the number of hidden units (making the network wider) and changing the learning rate. Because these values are all human-changeable, they're referred to as hyperparameters and the practice of trying to find the best hyperparameters is referred to as hyperparameter tuning.

Woah. We just introduced a bunch of possible steps. The important thing to remember is how you alter each of these will depend on the problem you're working on.

And the good thing is, over the next few problems, we'll get hands-on with all of them.

For now, let's keep it simple, all we'll do is train our model for longer (everything else will stay the same).

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Create a model (same as above)
model = tf.keras.Sequential([
    tf.keras.layers.Dense(1)
])

# Compile model (same as above)
model.compile(loss=tf.keras.losses.mae,
              optimizer=tf.keras.optimizers.SGD(),
              metrics=["mae"])

# Fit model (this time we'll train for longer)
model.fit(X, y, epochs=100) # train for 100 epochs not 10
```

Epoch 1/100  
1/1 [=====] - 0s 2ms/step - loss: 11.5048 - mae: 11.5048  
Epoch 2/100  
1/1 [=====] - 0s 2ms/step - loss: 11.3723 - mae: 11.3723  
Epoch 3/100  
1/1 [=====] - 0s 2ms/step - loss: 11.2398 - mae: 11.2398  
Epoch 4/100  
1/1 [=====] - 0s 2ms/step - loss: 11.1073 - mae: 11.1073  
Epoch 5/100  
1/1 [=====] - 0s 2ms/step - loss: 10.9748 - mae: 10.9748  
Epoch 6/100  
1/1 [=====] - 0s 2ms/step - loss: 10.8423 - mae: 10.8423  
Epoch 7/100  
1/1 [=====] - 0s 2ms/step - loss: 10.7098 - mae: 10.7098  
Epoch 8/100  
1/1 [=====] - 0s 2ms/step - loss: 10.5773 - mae: 10.5773  
Epoch 9/100  
1/1 [=====] - 0s 2ms/step - loss: 10.4448 - mae: 10.4448  
Epoch 10/100  
1/1 [=====] - 0s 2ms/step - loss: 10.3123 - mae: 10.3123  
Epoch 11/100  
1/1 [=====] - 0s 2ms/step - loss: 10.1798 - mae: 10.1798  
Epoch 12/100  
1/1 [=====] - 0s 2ms/step - loss: 10.0473 - mae: 10.0473  
Epoch 13/100  
1/1 [=====] - 0s 2ms/step - loss: 9.9148 - mae: 9.9148  
Epoch 14/100  
1/1 [=====] - 0s 2ms/step - loss: 9.7823 - mae: 9.7823  
Epoch 15/100  
1/1 [=====] - 0s 2ms/step - loss: 9.6498 - mae: 9.6498  
Epoch 16/100  
1/1 [=====] - 0s 2ms/step - loss: 9.5173 - mae: 9.5173  
Epoch 17/100  
1/1 [=====] - 0s 2ms/step - loss: 9.3848 - mae: 9.3848  
Epoch 18/100  
1/1 [=====] - 0s 2ms/step - loss: 9.2523 - mae: 9.2523  
Epoch 19/100  
1/1 [=====] - 0s 2ms/step - loss: 9.1198 - mae: 9.1198  
Epoch 20/100  
1/1 [=====] - 0s 2ms/step - loss: 8.9873 - mae: 8.9873  
Epoch 21/100  
1/1 [=====] - 0s 2ms/step - loss: 8.8548 - mae: 8.8548  
Epoch 22/100  
1/1 [=====] - 0s 2ms/step - loss: 8.7223 - mae: 8.7223  
Epoch 23/100  
1/1 [=====] - 0s 2ms/step - loss: 8.5898 - mae: 8.5898  
Epoch 24/100  
1/1 [=====] - 0s 2ms/step - loss: 8.4573 - mae: 8.4573  
Epoch 25/100  
1/1 [=====] - 0s 2ms/step - loss: 8.3248 - mae: 8.3248  
Epoch 26/100  
1/1 [=====] - 0s 2ms/step - loss: 8.1923 - mae: 8.1923  
Epoch 27/100  
1/1 [=====] - 0s 2ms/step - loss: 8.0598 - mae: 8.0598  
Epoch 28/100  
1/1 [=====] - 0s 2ms/step - loss: 7.9273 - mae: 7.9273  
Epoch 29/100  
1/1 [=====] - 0s 2ms/step - loss: 7.7948 - mae: 7.7948  
Epoch 30/100  
1/1 [=====] - 0s 2ms/step - loss: 7.6623 - mae: 7.6623  
Epoch 31/100  
1/1 [=====] - 0s 3ms/step - loss: 7.5298 - mae: 7.5298  
Epoch 32/100  
1/1 [=====] - 0s 3ms/step - loss: 7.3973 - mae: 7.3973  
Epoch 33/100  
1/1 [=====] - 0s 2ms/step - loss: 7.2648 - mae: 7.2648  
Epoch 34/100  
1/1 [=====] - 0s 2ms/step - loss: 7.2525 - mae: 7.2525  
Epoch 35/100  
1/1 [=====] - 0s 3ms/step - loss: 7.2469 - mae: 7.2469  
Epoch 36/100  
1/1 [=====] - 0s 2ms/step - loss: 7.2413 - mae: 7.2413

Epoch 37/100  
1/1 [=====] - 0s 2ms/step - loss: 7.2356 - mae: 7.2356  
Epoch 38/100  
1/1 [=====] - 0s 2ms/step - loss: 7.2300 - mae: 7.2300  
Epoch 39/100  
1/1 [=====] - 0s 2ms/step - loss: 7.2244 - mae: 7.2244  
Epoch 40/100  
1/1 [=====] - 0s 3ms/step - loss: 7.2188 - mae: 7.2188  
Epoch 41/100  
1/1 [=====] - 0s 3ms/step - loss: 7.2131 - mae: 7.2131  
Epoch 42/100  
1/1 [=====] - 0s 3ms/step - loss: 7.2075 - mae: 7.2075  
Epoch 43/100  
1/1 [=====] - 0s 5ms/step - loss: 7.2019 - mae: 7.2019  
Epoch 44/100  
1/1 [=====] - 0s 2ms/step - loss: 7.1963 - mae: 7.1963  
Epoch 45/100  
1/1 [=====] - 0s 5ms/step - loss: 7.1906 - mae: 7.1906  
Epoch 46/100  
1/1 [=====] - 0s 3ms/step - loss: 7.1850 - mae: 7.1850  
Epoch 47/100  
1/1 [=====] - 0s 3ms/step - loss: 7.1794 - mae: 7.1794  
Epoch 48/100  
1/1 [=====] - 0s 5ms/step - loss: 7.1738 - mae: 7.1738  
Epoch 49/100  
1/1 [=====] - 0s 2ms/step - loss: 7.1681 - mae: 7.1681  
Epoch 50/100  
1/1 [=====] - 0s 3ms/step - loss: 7.1625 - mae: 7.1625  
Epoch 51/100  
1/1 [=====] - 0s 5ms/step - loss: 7.1569 - mae: 7.1569  
Epoch 52/100  
1/1 [=====] - 0s 2ms/step - loss: 7.1512 - mae: 7.1512  
Epoch 53/100  
1/1 [=====] - 0s 2ms/step - loss: 7.1456 - mae: 7.1456  
Epoch 54/100  
1/1 [=====] - 0s 3ms/step - loss: 7.1400 - mae: 7.1400  
Epoch 55/100  
1/1 [=====] - 0s 8ms/step - loss: 7.1344 - mae: 7.1344  
Epoch 56/100  
1/1 [=====] - 0s 2ms/step - loss: 7.1287 - mae: 7.1287  
Epoch 57/100  
1/1 [=====] - 0s 6ms/step - loss: 7.1231 - mae: 7.1231  
Epoch 58/100  
1/1 [=====] - 0s 1ms/step - loss: 7.1175 - mae: 7.1175  
Epoch 59/100  
1/1 [=====] - 0s 3ms/step - loss: 7.1119 - mae: 7.1119  
Epoch 60/100  
1/1 [=====] - 0s 1ms/step - loss: 7.1063 - mae: 7.1063  
Epoch 61/100  
1/1 [=====] - 0s 1ms/step - loss: 7.1006 - mae: 7.1006  
Epoch 62/100  
1/1 [=====] - 0s 1ms/step - loss: 7.0950 - mae: 7.0950  
Epoch 63/100  
1/1 [=====] - 0s 1ms/step - loss: 7.0894 - mae: 7.0894  
Epoch 64/100  
1/1 [=====] - 0s 4ms/step - loss: 7.0838 - mae: 7.0838  
Epoch 65/100  
1/1 [=====] - 0s 1ms/step - loss: 7.0781 - mae: 7.0781  
Epoch 66/100  
1/1 [=====] - 0s 2ms/step - loss: 7.0725 - mae: 7.0725  
Epoch 67/100  
1/1 [=====] - 0s 4ms/step - loss: 7.0669 - mae: 7.0669  
Epoch 68/100  
1/1 [=====] - 0s 2ms/step - loss: 7.0613 - mae: 7.0613  
Epoch 69/100  
1/1 [=====] - 0s 3ms/step - loss: 7.0556 - mae: 7.0556  
Epoch 70/100  
1/1 [=====] - 0s 1ms/step - loss: 7.0500 - mae: 7.0500  
Epoch 71/100  
1/1 [=====] - 0s 985us/step - loss: 7.0444 - mae: 7.0444  
Epoch 72/100  
1/1 [=====] - 0s 1ms/step - loss: 7.0388 - mae: 7.0388

```
Epoch 73/100
1/1 [=====] - 0s 1ms/step - loss: 7.0331 - mae: 7.0331
Epoch 74/100
1/1 [=====] - 0s 1ms/step - loss: 7.0275 - mae: 7.0275
Epoch 75/100
1/1 [=====] - 0s 4ms/step - loss: 7.0219 - mae: 7.0219
Epoch 76/100
1/1 [=====] - 0s 4ms/step - loss: 7.0163 - mae: 7.0163
Epoch 77/100
1/1 [=====] - 0s 4ms/step - loss: 7.0106 - mae: 7.0106
Epoch 78/100
1/1 [=====] - 0s 3ms/step - loss: 7.0050 - mae: 7.0050
Epoch 79/100
1/1 [=====] - 0s 3ms/step - loss: 6.9994 - mae: 6.9994
Epoch 80/100
1/1 [=====] - 0s 1ms/step - loss: 6.9938 - mae: 6.9938
Epoch 81/100
1/1 [=====] - 0s 2ms/step - loss: 6.9881 - mae: 6.9881
Epoch 82/100
1/1 [=====] - 0s 3ms/step - loss: 6.9825 - mae: 6.9825
Epoch 83/100
1/1 [=====] - 0s 2ms/step - loss: 6.9769 - mae: 6.9769
Epoch 84/100
1/1 [=====] - 0s 9ms/step - loss: 6.9713 - mae: 6.9713
Epoch 85/100
1/1 [=====] - 0s 2ms/step - loss: 6.9656 - mae: 6.9656
Epoch 86/100
1/1 [=====] - 0s 1ms/step - loss: 6.9600 - mae: 6.9600
Epoch 87/100
1/1 [=====] - 0s 2ms/step - loss: 6.9544 - mae: 6.9544
Epoch 88/100
1/1 [=====] - 0s 2ms/step - loss: 6.9488 - mae: 6.9488
Epoch 89/100
1/1 [=====] - 0s 3ms/step - loss: 6.9431 - mae: 6.9431
Epoch 90/100
1/1 [=====] - 0s 3ms/step - loss: 6.9375 - mae: 6.9375
Epoch 91/100
1/1 [=====] - 0s 2ms/step - loss: 6.9319 - mae: 6.9319
Epoch 92/100
1/1 [=====] - 0s 2ms/step - loss: 6.9263 - mae: 6.9263
Epoch 93/100
1/1 [=====] - 0s 2ms/step - loss: 6.9206 - mae: 6.9206
Epoch 94/100
1/1 [=====] - 0s 2ms/step - loss: 6.9150 - mae: 6.9150
Epoch 95/100
1/1 [=====] - 0s 3ms/step - loss: 6.9094 - mae: 6.9094
Epoch 96/100
1/1 [=====] - 0s 3ms/step - loss: 6.9038 - mae: 6.9038
Epoch 97/100
1/1 [=====] - 0s 5ms/step - loss: 6.8981 - mae: 6.8981
Epoch 98/100
1/1 [=====] - 0s 3ms/step - loss: 6.8925 - mae: 6.8925
Epoch 99/100
1/1 [=====] - 0s 2ms/step - loss: 6.8869 - mae: 6.8869
Epoch 100/100
1/1 [=====] - 0s 2ms/step - loss: 6.8813 - mae: 6.8813
```

Out[ ]:

```
<tensorflow.python.keras.callbacks.History at 0x7fa24314c6d8>
```

You might've noticed the loss value decrease from before (and keep decreasing as the number of epochs gets higher).

What do you think this means for when we make a prediction with our model?

How about we try predict on 17.0 again?

In [ ]:

```
# Remind ourselves of what X and y are
```

```
X, y
```

```
Out[ ]:
```

```
(<tf.Tensor: shape=(8,), dtype=float32, numpy=array([-7., -4., -1., 2., 5., 8., 11., 14.], dtype=float32)>,
 <tf.Tensor: shape=(8,), dtype=float32, numpy=array([ 3.,  6.,  9., 12., 15., 18., 21., 24.], dtype=float32)>)
```

```
In [ ]:
```

```
# Try and predict what y would be if X was 17.0
model.predict([17.0]) # the right answer is 27.0 (y = X + 10)
```

```
Out[ ]:
```

```
array([[30.158512]], dtype=float32)
```

**Much better!**

**We got closer this time. But we could still be better.**

**Now we've trained a model, how could we evaluate it?**

## Evaluating a model

**A typical workflow you'll go through when building neural networks is:**

```
Build a model -> evaluate it -> build (tweak) a model -> evaluate it -> build (tweak) a model -> evaluate it...
```

**The tweaking comes from maybe not building a model from scratch but adjusting an existing one.**

### Visualize, visualize, visualize

**When it comes to evaluation, you'll want to remember the words: "visualize, visualize, visualize."**

**This is because you're probably better looking at something (doing) than you are thinking about something.**

**It's a good idea to visualize:**

- **The data** - what data are you working with? What does it look like?
- **The model itself** - what does the architecture look like? What are the different shapes?
- **The training of a model** - how does a model perform while it learns?
- **The predictions of a model** - how do the predictions of a model line up against the ground truth (the original labels)?

**Let's start by visualizing the model.**

**But first, we'll create a little bit of a bigger dataset and a new model we can use (it'll be the same as before, but the more practice the better).**

```
In [ ]:
```

```
# Make a bigger dataset
X = np.arange(-100, 100, 4)
X
```

```
Out[ ]:
```

```
array([-100, -96, -92, -88, -84, -80, -76, -72, -68, -64, -60,
       -56, -52, -48, -44, -40, -36, -32, -28, -24, -20, -16,
       -12, -8, -4,  0,   4,   8,  12,  16,  20,  24,  28,
       32,  36,  40,  44,  48,  52,  56,  60,  64,  68,  72,
       76,  80,  84,  88,  92,  96])
```

```
In [ ]:
```

```
# Make labels for the dataset (adhering to the same pattern as before)
```

```
y = np.arange(-90, 110, 4)
```

```
y
```

Out [ ]:

```
array([-90, -86, -82, -78, -74, -70, -66, -62, -58, -54, -50, -46, -42,
       -38, -34, -30, -26, -22, -18, -14, -10, -6, -2, 2, 6, 10,
       14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62,
       66, 70, 74, 78, 82, 86, 90, 94, 98, 102, 106])
```

Since  $y$  , we could make the labels like so:

$= X$

+ 10

In [ ]:

```
# Same result as above
```

```
y = X + 10
```

```
y
```

Out [ ]:

```
array([-90, -86, -82, -78, -74, -70, -66, -62, -58, -54, -50, -46, -42,
       -38, -34, -30, -26, -22, -18, -14, -10, -6, -2, 2, 6, 10,
       14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62,
       66, 70, 74, 78, 82, 86, 90, 94, 98, 102, 106])
```

## Split data into training/test set

One of the other most common and important steps in a machine learning project is creating a training and test set (and when required, a validation set).

Each set serves a specific purpose:

- **Training set** - the model learns from this data, which is typically 70-80% of the total data available (like the course materials you study during the semester).
- **Validation set** - the model gets tuned on this data, which is typically 10-15% of the total data available (like the practice exam you take before the final exam).
- **Test set** - the model gets evaluated on this data to test what it has learned, it's typically 10-15% of the total data available (like the final exam you take at the end of the semester).

For now, we'll just use a training and test set, this means we'll have a dataset for our model to learn on as well as be evaluated on.

We can create them by splitting our  $X$  and  $y$  arrays.

**Note:** When dealing with real-world data, this step is typically done right at the start of a project (the test set should always be kept separate from all other data). We want our model to learn on training data and then evaluate it on test data to get an indication of how well it generalizes to unseen examples.

In [ ]:

```
# Check how many samples we have
len(X)
```

Out [ ]:

```
50
```

In [ ]:

```
# Split data into train and test sets
X_train = X[:40] # first 40 examples (80% of data)
y_train = y[:40]
```

```
X_test = X[40:] # last 10 examples (20% of data)
y_test = y[40:]

len(X_train), len(X_test)
```

Out [ ]:

(40, 10)

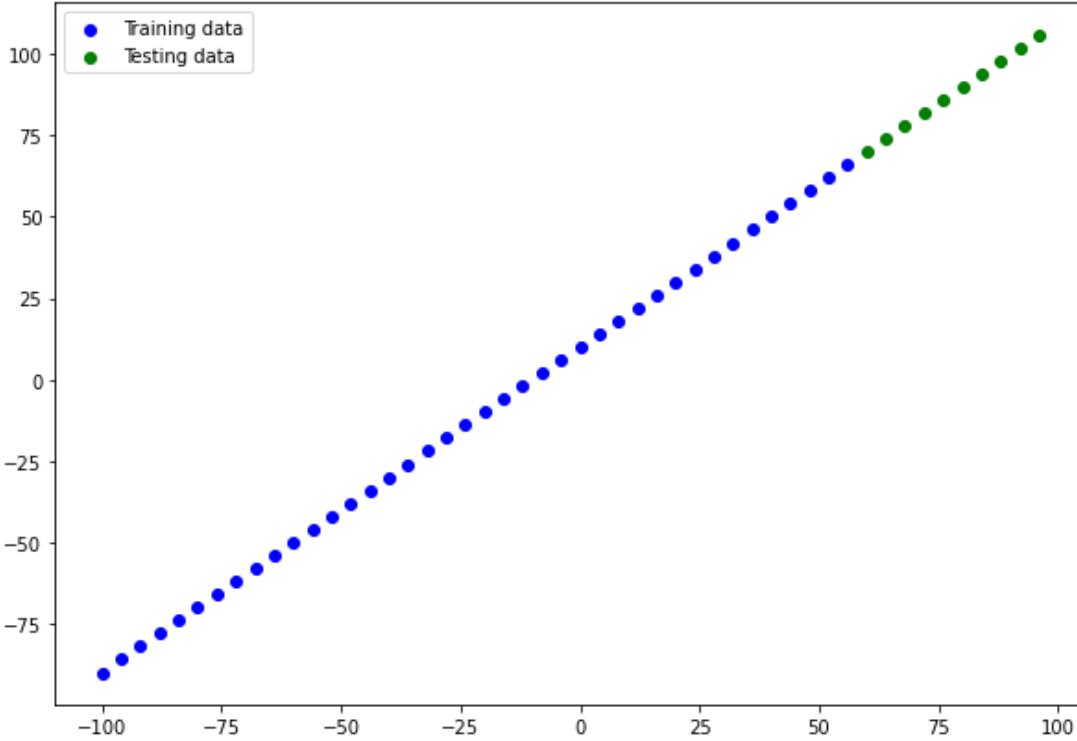
## Visualizing the data

Now we've got our training and test data, it's a good idea to visualize it.

Let's plot it with some nice colours to differentiate what's what.

In [ ]:

```
plt.figure(figsize=(10, 7))
# Plot training data in blue
plt.scatter(X_train, y_train, c='b', label='Training data')
# Plot test data in green
plt.scatter(X_test, y_test, c='g', label='Testing data')
# Show the legend
plt.legend();
```



Beautiful! Any time you can visualize your data, your model, your anything, it's a good idea.

With this graph in mind, what we'll be trying to do is build a model which learns the pattern in the blue dots (`X_train`) to draw the green dots (`X_test`).

Time to build a model. We'll make the exact same one from before (the one we trained for longer).

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Create a model (same as above)
model = tf.keras.Sequential([
    tf.keras.layers.Dense(1)
])

# Compile model (same as above)
model.compile(loss=tf.keras.losses.mae,
```

```
optimizer=tf.keras.optimizers.SGD(),
metrics=["mae"])

# Fit model (same as above)
#model.fit(X_train, y_train, epochs=100) # commented out on purpose (not fitting it just yet)
```

## Visualizing the model

After you've built a model, you might want to take a look at it (especially if you haven't built many before).

You can take a look at the layers and shapes of your model by calling `summary()` on it.

**Note:** Visualizing a model is particularly helpful when you run into input and output shape mismatches.

In [ ]:

```
# Doesn't work (model not fit/built)
model.summary()
```

```
-----
ValueError                                                 Traceback (most recent call last)
<ipython-input-21-7d09d31d4e66> in <module>()
      1 # Doesn't work (model not fit/built)
----> 2 model.summary()

/usr/local/lib/python3.6/dist-packages/tensorflow/python/keras/engine/training.py in summary(self, line_length, positions, print_fn)
 2349     """
 2350     if not self.built:
-> 2351         raise ValueError('This model has not yet been built. '
 2352                           'Build the model first by calling `build()` or calling '
 2353                           '`fit()` with some data, or specify '
```

`ValueError`: This model has not yet been built. Build the model first by calling ``build()`` or calling ``fit()`` with some data, or specify an ``input_shape`` argument in the first layer(s) for automatic build.

Ahh, the cell above errors because we haven't fit our built our model.

We also haven't told it what input shape it should be expecting.

Remember above, how we discussed the input shape was just one number?

We can let our model know the input shape of our data using the `input_shape` parameter to the first layer (usually if `input_shape` isn't defined, Keras tries to figure it out automatically).

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Create a model (same as above)
model = tf.keras.Sequential([
    tf.keras.layers.Dense(1, input_shape=[1]) # define the input_shape to our model
])

# Compile model (same as above)
model.compile(loss=tf.keras.losses.mae,
              optimizer=tf.keras.optimizers.SGD(),
              metrics=["mae"])
```

In [ ]:

```
# This will work after specifying the input shape
```

```
model.summary()
```

```
Model: "sequential_7"
```

Layer (type)	Output Shape	Param #
dense_7 (Dense)	(None, 1)	2
Total params: 2		
Trainable params: 2		
Non-trainable params: 0		

Calling `summary()` on our model shows us the layers it contains, the output shape and the number of parameters.

- **Total params** - total number of parameters in the model.
- **Trainable parameters** - these are the parameters (patterns) the model can update as it trains.
- **Non-trainable parameters** - these parameters aren't updated during training (this is typical when you bring in the already learned patterns from other models during transfer learning).

□ **Resource:** For a more in-depth overview of the trainable parameters within a layer, check out [MIT's introduction to deep learning video](#).

□ **Exercise:** Try playing around with the number of hidden units in the `Dense` layer (e.g. `Dense(2)`, `Dense(3)`). How does this change the Total/Trainable params? Investigate what's causing the change.

For now, all you need to think about these parameters is that their learnable patterns in the data.

Let's fit our model to the training data.

```
In [ ]:
```

```
# Fit the model to the training data
model.fit(X_train, y_train, epochs=100, verbose=0) # verbose controls how much gets output
```

```
Out[ ]:
```

```
<tensorflow.python.keras.callbacks.History at 0x7fa238b42ac8>
```

```
In [ ]:
```

```
# Check the model summary
model.summary()
```

```
Model: "sequential_7"
```

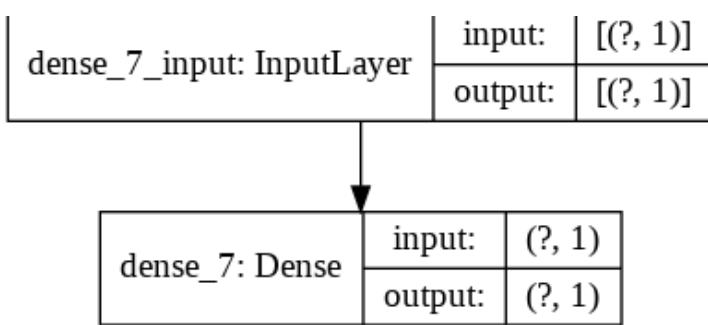
Layer (type)	Output Shape	Param #
dense_7 (Dense)	(None, 1)	2
Total params: 2		
Trainable params: 2		
Non-trainable params: 0		

Alongside summary, you can also view a 2D plot of the model using `plot_model()`.

```
In [ ]:
```

```
from tensorflow.keras.utils import plot_model
plot_model(model, show_shapes=True)
```

```
Out[ ]:
```



In our case, the model we used only has an input and an output but visualizing more complicated models can be very helpful for debugging.

## Visualizing the predictions

Now we've got a trained model, let's visualize some predictions.

To visualize predictions, it's always a good idea to plot them against the ground truth labels.

Often you'll see this in the form of `y_test` vs. `y_pred` (ground truth vs. predictions).

First, we'll make some predictions on the test data (`x_test`), remember the model has never seen the test data.

```
In [ ]:
```

```
# Make predictions
y_preds = model.predict(x_test)
```

```
In [ ]:
```

```
# View the predictions
y_preds
```

```
Out[ ]:
```

```
array([[53.57109 ],
       [57.05633 ],
       [60.541573],
       [64.02681 ],
       [67.512054],
       [70.99729 ],
       [74.48254 ],
       [77.96777 ],
       [81.45301 ],
       [84.938255]], dtype=float32)
```

Okay, we get a list of numbers but how do these compare to the ground truth labels?

Let's build a plotting function to find out.

**Note:** If you think you're going to be visualizing something a lot, it's a good idea to functionize it so you can use it later.

```
In [ ]:
```

```
def plot_predictions(train_data=x_train,
                     train_labels=y_train,
                     test_data=x_test,
                     test_labels=y_test,
                     predictions=y_preds):
    """
    Plots training data, test data and compares predictions.
    """
    plt.figure(figsize=(10, 7))
```

```

# Plot training data in blue
plt.scatter(train_data, train_labels, c="b", label="Training data")
# Plot test data in green
plt.scatter(test_data, test_labels, c="g", label="Testing data")
# Plot the predictions in red (predictions were made on the test data)
plt.scatter(test_data, predictions, c="r", label="Predictions")
# Show the legend
plt.legend();

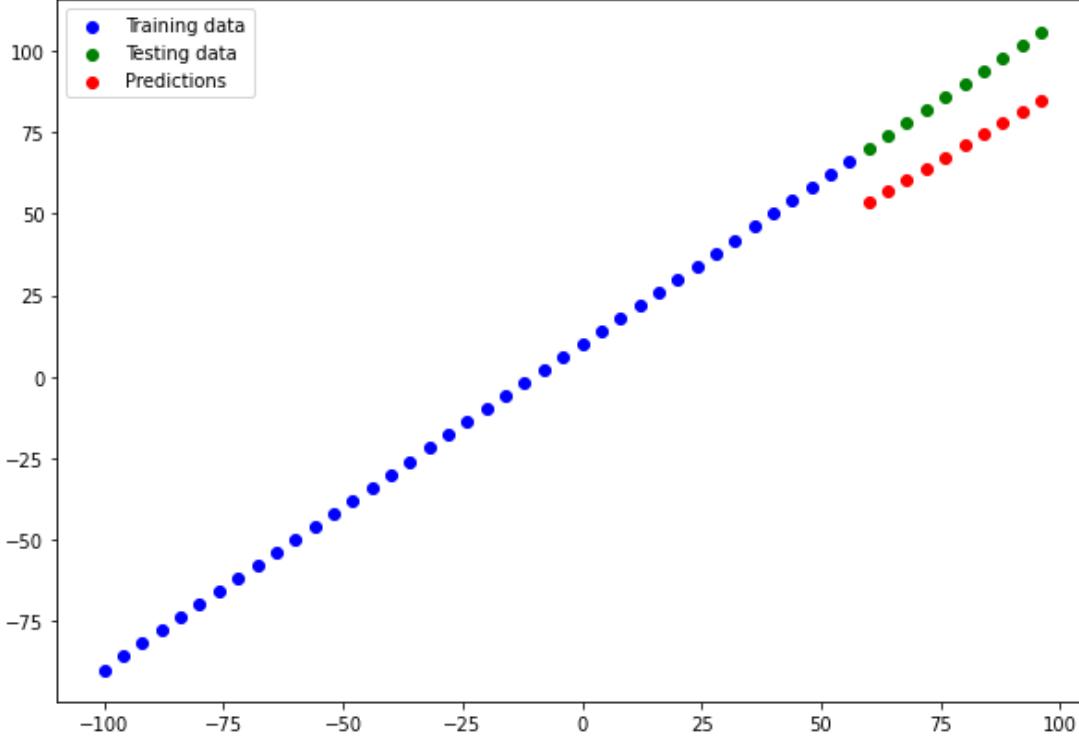
```

In [ ]:

```

plot_predictions(train_data=X_train,
                 train_labels=y_train,
                 test_data=X_test,
                 test_labels=y_test,
                 predictions=y_preds)

```



From the plot we can see our predictions aren't totally outlandish but they definitely aren't anything special either.

## Evaluating predictions

Alongisde visualizations, evalauation metrics are your alternative best option for evaluating your model.

Depending on the problem you're working on, different models have different evaluation metrics.

Two of the main metrics used for regression problems are:

- **Mean absolute error (MAE)** - the mean difference between each of the predictions.
- **Mean squared error (MSE)** - the squared mean difference between of the predictions (use if larger errors are more detrimental than smaller errors).

The lower each of these values, the better.

You can also use `model.evaluate()` which will return the loss of the model as well as any metrics setup during the compile step.

In [ ]:

```

# Evaluate the model on the test set
model.evaluate(X_test, y_test)

```

1/1 [=====] - 0s 1ms/step - loss: 18.7453 - mae: 18.7453

Out [ ]:

```
[18.74532699584961, 18.74532699584961]
```

In our case, since we used MAE for the loss function as well as MAE for the metrics, `model.evaluate()` returns them both.

TensorFlow also has built in functions for MSE and MAE.

For many evaluation functions, the premise is the same: compare predictions to the ground truth labels.

In [ ]:

```
# Calculate the mean absolute error
mae = tf.metrics.mean_absolute_error(y_true=y_test,
                                      y_pred=y_preds)
mae
```

Out [ ]:

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([34.42891 , 30.943668, 27.45843 , 23.97319 , 20.487946, 17.202168,
       14.510478, 12.419336, 11.018796, 10.212349], dtype=float32)>
```

Huh? That's strange, MAE should be a single output.

Instead, we get 10 values.

This is because our `y_test` and `y_preds` tensors are different shapes.

In [ ]:

```
# Check the test label tensor values
y_test
```

Out [ ]:

```
array([ 70,  74,  78,  82,  86,  90,  94,  98, 102, 106])
```

In [ ]:

```
# Check the predictions tensor values (notice the extra square brackets)
y_preds
```

Out [ ]:

```
array([[53.57109 ],
       [57.05633 ],
       [60.541573],
       [64.02681 ],
       [67.512054],
       [70.99729 ],
       [74.48254 ],
       [77.96777 ],
       [81.45301 ],
       [84.938255]], dtype=float32)
```

In [ ]:

```
# Check the tensor shapes
y_test.shape, y_preds.shape
```

Out [ ]:

```
((10,), (10, 1))
```

Remember how we discussed dealing with different input and output shapes is one the most common issues you'll come across, this is one of those times.

But not to worry.

We can fix it using `squeeze()`, it'll remove the the 1 dimension from our `y_preds` tensor, making it the same shape as `y_test`.

□ Note: If you're comparing two tensors, it's important to make sure they're the right shape(s) (you won't always have to manipulate the shapes, but always be on the look out, *many* errors are the result of mismatched tensors, especially mismatched input and output shapes).

In [ ]:

```
# Shape before squeeze()
y_preds.shape
```

Out[ ]:

```
(10, 1)
```

In [ ]:

```
# Shape after squeeze()
y_preds.squeeze().shape
```

Out[ ]:

```
(10,)
```

In [ ]:

```
# What do they look like?
y_test, y_preds.squeeze()
```

Out[ ]:

```
(array([ 70,  74,  78,  82,  86,  90,  94,  98, 102, 106]),
 array([53.57109 , 57.05633 , 60.541573, 64.02681 , 67.512054, 70.99729 ,
        74.48254 , 77.96777 , 81.45301 , 84.938255], dtype=float32))
```

Okay, now we know how to make our `y_test` and `y_preds` tensors the same shape, let's use our evaluation metrics.

In [ ]:

```
# Calculate the MAE
mae = tf.metrics.mean_absolute_error(y_true=y_test,
                                      y_pred=y_preds.squeeze()) # use squeeze() to make same shape
mae
```

Out[ ]:

```
<tf.Tensor: shape=(), dtype=float32, numpy=18.745327>
```

In [ ]:

```
# Calculate the MSE
mse = tf.metrics.mean_squared_error(y_true=y_test,
                                      y_pred=y_preds.squeeze())
mse
```

Out[ ]:

```
<tf.Tensor: shape=(), dtype=float32, numpy=353.57336>
```

We can also calculate the MAE using pure TensorFlow functions.

In [ ]:

```
# Returns the same as tf.metrics.mean_absolute_error()
tf.reduce_mean(tf.abs(y_test-y_preds.squeeze()))
```

Out [ ]:

```
<tf.Tensor: shape=(), dtype=float64, numpy=18.745327377319335>
```

Again, it's a good idea to functionize anything you think you might use over again (or find yourself using over and over again).

Let's make functions for our evaluation metrics.

In [ ]:

```
def mae(y_test, y_pred):  
    """  
    Calculates mean absolute error between y_test and y_preds.  
    """  
    return tf.metrics.mean_absolute_error(y_test,  
                                         y_pred)  
  
def mse(y_test, y_pred):  
    """  
    Calculates mean squared error between y_test and y_preds.  
    """  
    return tf.metrics.mean_squared_error(y_test,  
                                         y_pred)
```

## Running experiments to improve a model

After seeing the evaluation metrics and the predictions your model makes, it's likely you'll want to improve it.

Again, there are many different ways you can do this, but 3 of the main ones are:

1. **Get more data** - get more examples for your model to train on (more opportunities to learn patterns).
2. **Make your model larger (use a more complex model)** - this might come in the form of more layers or more hidden units in each layer.
3. **Train for longer** - give your model more of a chance to find the patterns in the data.

Since we created our dataset, we could easily make more data but this isn't always the case when you're working with real-world datasets.

So let's take a look at how we can improve our model using 2 and 3.

To do so, we'll build 3 models and compare their results:

1. `model_1` - same as original model, 1 layer, trained for 100 epochs.
2. `model_2` - 2 layers, trained for 100 epochs.
3. `model_3` - 2 layers, trained for 500 epochs.

Build `model_1`

In [ ]:

```
# Set random seed  
tf.random.set_seed(42)  
  
# Replicate original model  
model_1 = tf.keras.Sequential([  
    tf.keras.layers.Dense(1)  
)  
  
# Compile the model  
model_1.compile(loss=tf.keras.losses.mae,  
                 optimizer=tf.keras.optimizers.SGD(),  
                 metrics=['mae'])  
  
# Fit the model  
model_1.fit(X_train, y_train, epochs=100)
```

Epoch 1/100

2/2 [=====] - 0s 2ms/step - loss: 15.9024 - mae: 15.9024  
Epoch 2/100  
2/2 [=====] - 0s 2ms/step - loss: 11.2837 - mae: 11.2837  
Epoch 3/100  
2/2 [=====] - 0s 2ms/step - loss: 11.1074 - mae: 11.1074  
Epoch 4/100  
2/2 [=====] - 0s 2ms/step - loss: 9.2991 - mae: 9.2991  
Epoch 5/100  
2/2 [=====] - 0s 2ms/step - loss: 10.1677 - mae: 10.1677  
Epoch 6/100  
2/2 [=====] - 0s 1ms/step - loss: 9.4303 - mae: 9.4303  
Epoch 7/100  
2/2 [=====] - 0s 1ms/step - loss: 8.5704 - mae: 8.5704  
Epoch 8/100  
2/2 [=====] - 0s 2ms/step - loss: 9.0442 - mae: 9.0442  
Epoch 9/100  
2/2 [=====] - 0s 2ms/step - loss: 18.7517 - mae: 18.7517  
Epoch 10/100  
2/2 [=====] - 0s 1ms/step - loss: 10.1142 - mae: 10.1142  
Epoch 11/100  
2/2 [=====] - 0s 2ms/step - loss: 8.3980 - mae: 8.3980  
Epoch 12/100  
2/2 [=====] - 0s 2ms/step - loss: 10.6639 - mae: 10.6639  
Epoch 13/100  
2/2 [=====] - 0s 1ms/step - loss: 9.7977 - mae: 9.7977  
Epoch 14/100  
2/2 [=====] - 0s 2ms/step - loss: 16.0103 - mae: 16.0103  
Epoch 15/100  
2/2 [=====] - 0s 2ms/step - loss: 11.4068 - mae: 11.4068  
Epoch 16/100  
2/2 [=====] - 0s 2ms/step - loss: 8.5393 - mae: 8.5393  
Epoch 17/100  
2/2 [=====] - 0s 2ms/step - loss: 13.6348 - mae: 13.6348  
Epoch 18/100  
2/2 [=====] - 0s 2ms/step - loss: 11.4629 - mae: 11.4629  
Epoch 19/100  
2/2 [=====] - 0s 2ms/step - loss: 17.9148 - mae: 17.9148  
Epoch 20/100  
2/2 [=====] - 0s 2ms/step - loss: 15.0494 - mae: 15.0494  
Epoch 21/100  
2/2 [=====] - 0s 2ms/step - loss: 11.0216 - mae: 11.0216  
Epoch 22/100  
2/2 [=====] - 0s 2ms/step - loss: 8.1558 - mae: 8.1558  
Epoch 23/100  
2/2 [=====] - 0s 2ms/step - loss: 9.5138 - mae: 9.5138  
Epoch 24/100  
2/2 [=====] - 0s 2ms/step - loss: 7.6617 - mae: 7.6617  
Epoch 25/100  
2/2 [=====] - 0s 3ms/step - loss: 13.1859 - mae: 13.1859  
Epoch 26/100  
2/2 [=====] - 0s 3ms/step - loss: 16.4211 - mae: 16.4211  
Epoch 27/100  
2/2 [=====] - 0s 2ms/step - loss: 13.1660 - mae: 13.1660  
Epoch 28/100  
2/2 [=====] - 0s 2ms/step - loss: 14.2559 - mae: 14.2559  
Epoch 29/100  
2/2 [=====] - 0s 2ms/step - loss: 10.0670 - mae: 10.0670  
Epoch 30/100  
2/2 [=====] - 0s 2ms/step - loss: 16.3409 - mae: 16.3409  
Epoch 31/100  
2/2 [=====] - 0s 1ms/step - loss: 23.6444 - mae: 23.6444  
Epoch 32/100  
2/2 [=====] - 0s 2ms/step - loss: 7.6215 - mae: 7.6215  
Epoch 33/100  
2/2 [=====] - 0s 2ms/step - loss: 9.3221 - mae: 9.3221  
Epoch 34/100  
2/2 [=====] - 0s 2ms/step - loss: 13.7313 - mae: 13.7313  
Epoch 35/100  
2/2 [=====] - 0s 2ms/step - loss: 11.1276 - mae: 11.1276  
Epoch 36/100  
2/2 [=====] - 0s 2ms/step - loss: 13.3222 - mae: 13.3222  
Epoch 37/100

2/2 [=====] - 0s 1ms/step - loss: 9.4763 - mae: 9.4763  
Epoch 38/100  
2/2 [=====] - 0s 2ms/step - loss: 10.1381 - mae: 10.1381  
Epoch 39/100  
2/2 [=====] - 0s 2ms/step - loss: 10.1793 - mae: 10.1793  
Epoch 40/100  
2/2 [=====] - 0s 2ms/step - loss: 10.9137 - mae: 10.9137  
Epoch 41/100  
2/2 [=====] - 0s 2ms/step - loss: 7.9063 - mae: 7.9063  
Epoch 42/100  
2/2 [=====] - 0s 2ms/step - loss: 10.0914 - mae: 10.0914  
Epoch 43/100  
2/2 [=====] - 0s 2ms/step - loss: 8.7006 - mae: 8.7006  
Epoch 44/100  
2/2 [=====] - 0s 2ms/step - loss: 12.2047 - mae: 12.2047  
Epoch 45/100  
2/2 [=====] - 0s 2ms/step - loss: 13.7970 - mae: 13.7970  
Epoch 46/100  
2/2 [=====] - 0s 2ms/step - loss: 8.4687 - mae: 8.4687  
Epoch 47/100  
2/2 [=====] - 0s 2ms/step - loss: 9.1330 - mae: 9.1330  
Epoch 48/100  
2/2 [=====] - 0s 2ms/step - loss: 10.6190 - mae: 10.6190  
Epoch 49/100  
2/2 [=====] - 0s 2ms/step - loss: 7.7503 - mae: 7.7503  
Epoch 50/100  
2/2 [=====] - 0s 2ms/step - loss: 9.5407 - mae: 9.5407  
Epoch 51/100  
2/2 [=====] - 0s 2ms/step - loss: 9.1584 - mae: 9.1584  
Epoch 52/100  
2/2 [=====] - 0s 2ms/step - loss: 16.3630 - mae: 16.3630  
Epoch 53/100  
2/2 [=====] - 0s 2ms/step - loss: 14.1299 - mae: 14.1299  
Epoch 54/100  
2/2 [=====] - 0s 2ms/step - loss: 21.1247 - mae: 21.1247  
Epoch 55/100  
2/2 [=====] - 0s 2ms/step - loss: 16.3961 - mae: 16.3961  
Epoch 56/100  
2/2 [=====] - 0s 2ms/step - loss: 9.9806 - mae: 9.9806  
Epoch 57/100  
2/2 [=====] - 0s 2ms/step - loss: 9.9606 - mae: 9.9606  
Epoch 58/100  
2/2 [=====] - 0s 8ms/step - loss: 9.2209 - mae: 9.2209  
Epoch 59/100  
2/2 [=====] - 0s 2ms/step - loss: 8.4239 - mae: 8.4239  
Epoch 60/100  
2/2 [=====] - 0s 2ms/step - loss: 9.4869 - mae: 9.4869  
Epoch 61/100  
2/2 [=====] - 0s 2ms/step - loss: 11.4355 - mae: 11.4355  
Epoch 62/100  
2/2 [=====] - 0s 2ms/step - loss: 11.6887 - mae: 11.6887  
Epoch 63/100  
2/2 [=====] - 0s 2ms/step - loss: 7.0838 - mae: 7.0838  
Epoch 64/100  
2/2 [=====] - 0s 2ms/step - loss: 16.9675 - mae: 16.9675  
Epoch 65/100  
2/2 [=====] - 0s 1ms/step - loss: 12.4599 - mae: 12.4599  
Epoch 66/100  
2/2 [=====] - 0s 2ms/step - loss: 13.0184 - mae: 13.0184  
Epoch 67/100  
2/2 [=====] - 0s 2ms/step - loss: 8.0600 - mae: 8.0600  
Epoch 68/100  
2/2 [=====] - 0s 2ms/step - loss: 10.1888 - mae: 10.1888  
Epoch 69/100  
2/2 [=====] - 0s 2ms/step - loss: 12.3633 - mae: 12.3633  
Epoch 70/100  
2/2 [=====] - 0s 2ms/step - loss: 9.0516 - mae: 9.0516  
Epoch 71/100  
2/2 [=====] - 0s 2ms/step - loss: 10.0378 - mae: 10.0378  
Epoch 72/100  
2/2 [=====] - 0s 1ms/step - loss: 10.0516 - mae: 10.0516  
Epoch 73/100

```
2/2 [=====] - 0s 1ms/step - loss: 12.6151 - mae: 12.6151
Epoch 74/100
2/2 [=====] - 0s 1ms/step - loss: 10.3819 - mae: 10.3819
Epoch 75/100
2/2 [=====] - 0s 1ms/step - loss: 9.7229 - mae: 9.7229
Epoch 76/100
2/2 [=====] - 0s 2ms/step - loss: 11.2252 - mae: 11.2252
Epoch 77/100
2/2 [=====] - 0s 2ms/step - loss: 8.3642 - mae: 8.3642
Epoch 78/100
2/2 [=====] - 0s 2ms/step - loss: 9.1274 - mae: 9.1274
Epoch 79/100
2/2 [=====] - 0s 2ms/step - loss: 19.5039 - mae: 19.5039
Epoch 80/100
2/2 [=====] - 0s 1ms/step - loss: 14.8945 - mae: 14.8945
Epoch 81/100
2/2 [=====] - 0s 1ms/step - loss: 9.0034 - mae: 9.0034
Epoch 82/100
2/2 [=====] - 0s 2ms/step - loss: 13.0206 - mae: 13.0206
Epoch 83/100
2/2 [=====] - 0s 2ms/step - loss: 7.9299 - mae: 7.9299
Epoch 84/100
2/2 [=====] - 0s 2ms/step - loss: 7.6872 - mae: 7.6872
Epoch 85/100
2/2 [=====] - 0s 2ms/step - loss: 10.0328 - mae: 10.0328
Epoch 86/100
2/2 [=====] - 0s 2ms/step - loss: 9.2433 - mae: 9.2433
Epoch 87/100
2/2 [=====] - 0s 2ms/step - loss: 12.0209 - mae: 12.0209
Epoch 88/100
2/2 [=====] - 0s 1ms/step - loss: 10.6389 - mae: 10.6389
Epoch 89/100
2/2 [=====] - 0s 2ms/step - loss: 7.2667 - mae: 7.2667
Epoch 90/100
2/2 [=====] - 0s 2ms/step - loss: 12.7786 - mae: 12.7786
Epoch 91/100
2/2 [=====] - 0s 1ms/step - loss: 7.3481 - mae: 7.3481
Epoch 92/100
2/2 [=====] - 0s 2ms/step - loss: 7.7175 - mae: 7.7175
Epoch 93/100
2/2 [=====] - 0s 1ms/step - loss: 7.1263 - mae: 7.1263
Epoch 94/100
2/2 [=====] - 0s 3ms/step - loss: 12.6190 - mae: 12.6190
Epoch 95/100
2/2 [=====] - 0s 2ms/step - loss: 10.0912 - mae: 10.0912
Epoch 96/100
2/2 [=====] - 0s 1ms/step - loss: 9.3558 - mae: 9.3558
Epoch 97/100
2/2 [=====] - 0s 1ms/step - loss: 12.6834 - mae: 12.6834
Epoch 98/100
2/2 [=====] - 0s 1ms/step - loss: 8.6762 - mae: 8.6762
Epoch 99/100
2/2 [=====] - 0s 2ms/step - loss: 9.4693 - mae: 9.4693
Epoch 100/100
2/2 [=====] - 0s 1ms/step - loss: 8.7067 - mae: 8.7067
```

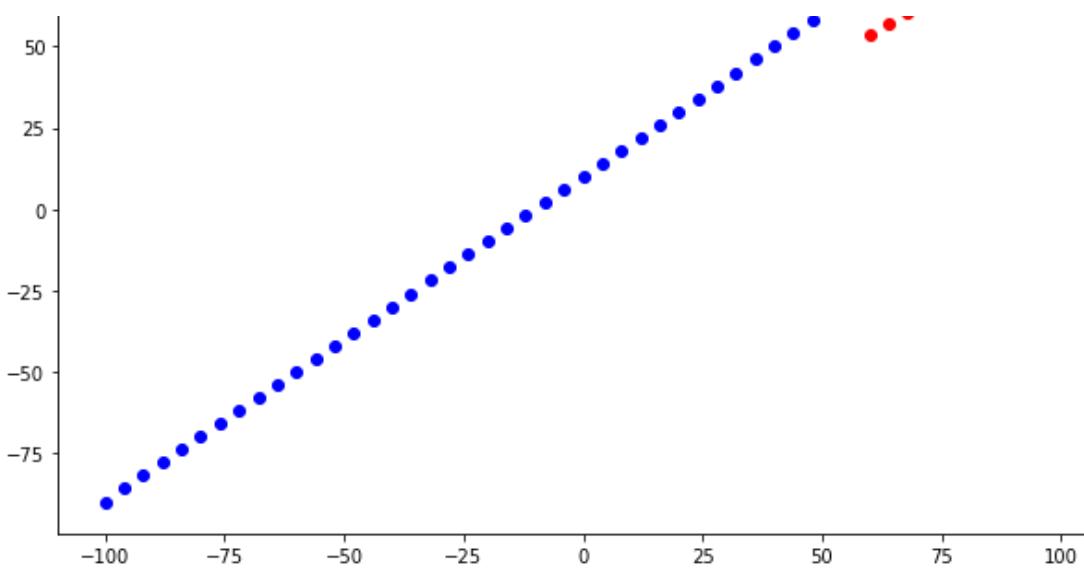
Out[ ]:

```
<tensorflow.python.keras.callbacks.History at 0x7fa2388fec50>
```

In [ ]:

```
# Make and plot predictions for model_1
y_preds_1 = model_1.predict(X_test)
plot_predictions(predictions=y_preds_1)
```





In [ ]:

```
# Calculate model_1 metrics
mae_1 = mae(y_test, y_preds_1.squeeze()).numpy()
mse_1 = mse(y_test, y_preds_1.squeeze()).numpy()
mae_1, mse_1
```

Out [ ]:

(18.745327, 353.57336)

**Build model\_2**

**This time we'll add an extra dense layer (so now our model will have 2 layers) whilst keeping everything else the same.**

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Replicate model_1 and add an extra layer
model_2 = tf.keras.Sequential([
    tf.keras.layers.Dense(1),
    tf.keras.layers.Dense(1) # add a second layer
])

# Compile the model
model_2.compile(loss=tf.keras.losses.mae,
                 optimizer=tf.keras.optimizers.SGD(),
                 metrics=['mae'])

# Fit the model
model_2.fit(X_train, y_train, epochs=100, verbose=0) # set verbose to 0 for less output
```

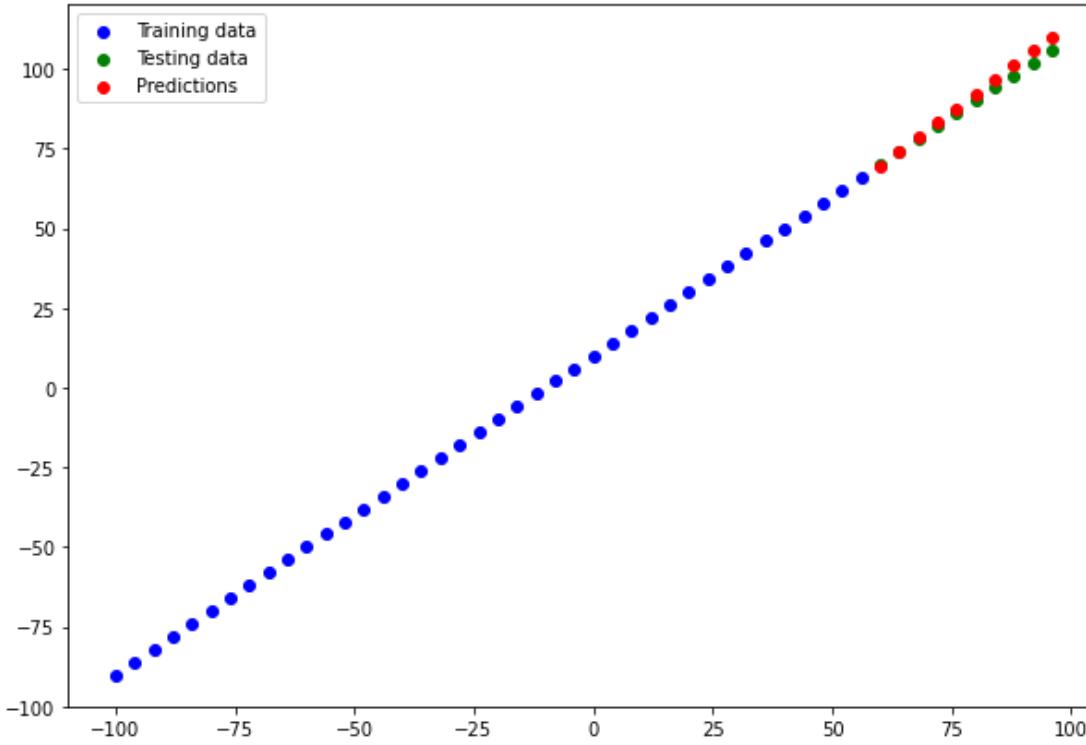
Out [ ]:

<tensorflow.python.keras.callbacks.History at 0x7fa23800bf98>

In [ ]:

```
# Make and plot predictions for model_2
y_preds_2 = model_2.predict(X_test)
plot_predictions(predictions=y_preds_2)
```

WARNING:tensorflow:5 out of the last 5 calls to <function Model.make\_predict\_function.<locals>.predict\_function at 0x7fa239dc71e0> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has experimental\_relax\_shapes=True option that relaxes argument shapes that can avoid unnecessary retracing. For (3), please refer to [https://www.tensorflow.org/tutorials/customization/performance#python\\_and\\_tensorflow\\_and\\_an](https://www.tensorflow.org/tutorials/customization/performance#python_and_tensorflow_and_an)



Woah, that's looking better already! And all it took was an extra layer.

In [ ]:

```
# Calculate model_2 metrics
mae_2 = mae(y_test, y_preds_2.squeeze()).numpy()
mse_2 = mse(y_test, y_preds_2.squeeze()).numpy()
mae_2, mse_2
```

Out[ ]:

```
(1.9098114, 5.459232)
```

Build `model_3`

For our 3rd model, we'll keep everything the same as `model_2` except this time we'll train for longer (500 epochs instead of 100).

This will give our model more of a chance to learn the patterns in the data.

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Replicate model_2
model_3 = tf.keras.Sequential([
    tf.keras.layers.Dense(1),
    tf.keras.layers.Dense(1)
])

# Compile the model
model_3.compile(loss=tf.keras.losses.mae,
                 optimizer=tf.keras.optimizers.SGD(),
                 metrics=['mae'])

# Fit the model (this time for 500 epochs, not 100)
model_3.fit(X_train, y_train, epochs=500, verbose=0) # set verbose to 0 for less output
```

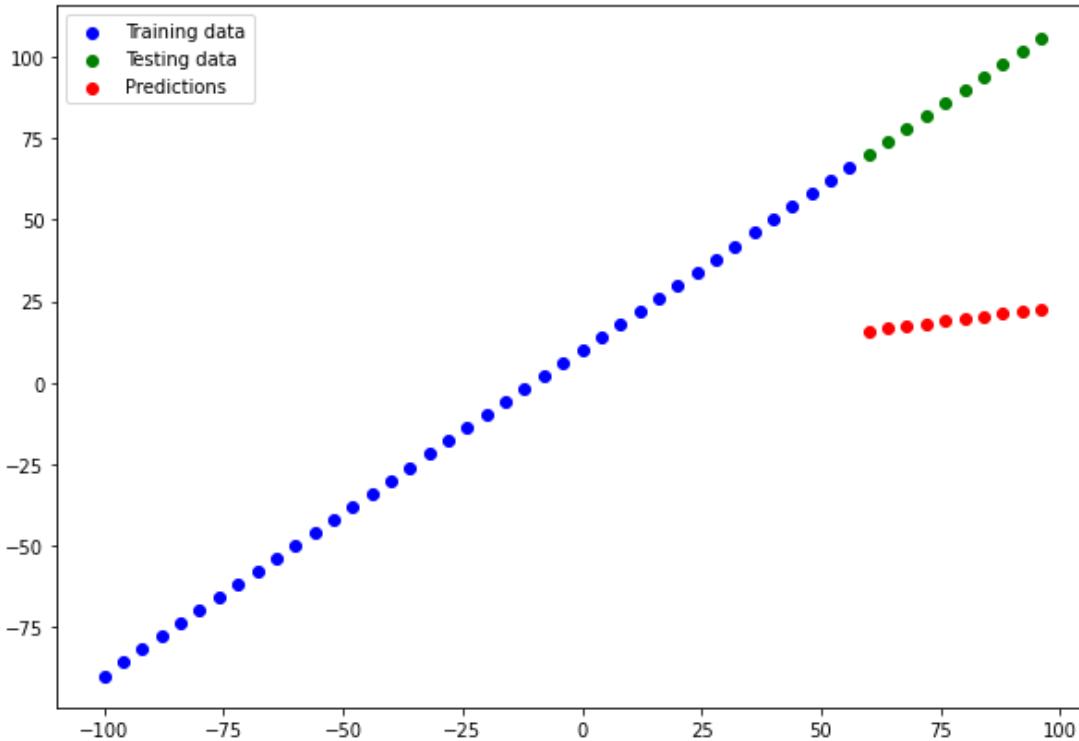
Out[ ]:

```
<tensorflow.python.keras.callbacks.History at 0x7fa239ce6cf8>
```

In [ ]:

```
# Make and plot predictions for model_3
y_preds_3 = model_3.predict(X_test)
plot_predictions(predictions=y_preds_3)
```

WARNING:tensorflow:6 out of the last 6 calls to <function Model.make\_predict\_function.<locals>.predict\_function at 0x7fa239c411e0> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has experimental\_relax\_shapes=True option that relaxes argument shapes that can avoid unnecessary retracing. For (3), please refer to [https://www.tensorflow.org/tutorials/customization/performance#python\\_or\\_tensor\\_args](https://www.tensorflow.org/tutorials/customization/performance#python_or_tensor_args) and [https://www.tensorflow.org/api\\_docs/python/tf/function](https://www.tensorflow.org/api_docs/python/tf/function) for more details.



Strange, we trained for longer but our model performed worse?

As it turns out, our model might've trained too long and has thus resulted in worse results (we'll see ways to prevent training for too long later on).

In [ ]:

```
# Calculate model_3 metrics
mae_3 = mae(y_test, y_preds_3.squeeze()).numpy()
mse_3 = mse(y_test, y_preds_3.squeeze()).numpy()
mae_3, mse_3
```

Out [ ]:

(68.68786, 4804.4717)

## Comparing results

Now we've got results for 3 similar but slightly different results, let's compare them.

In [ ]:

```
model_results = [[ "model_1", mae_1, mse_1],
                  [ "model_2", mae_2, mse_2],
                  [ "model_3", mae_3, mae_3]]
```

In [ ]:

```
import pandas as pd
all_results = pd.DataFrame(model_results, columns=["model", "mae", "mse"])
all_results
```

Out[ ]:

	model	mae	mse
0	model_1	18.745327	353.573364
1	model_2	1.909811	5.459232
2	model_3	68.687859	68.687859

From our experiments, it looks like `model_2` performed the best.

And now, you might be thinking, "wow, comparing models is tedious..." and it definitely can be, we've only compared 3 models here.

But this is part of what machine learning modelling is about, trying many different combinations of models and seeing which performs best.

Each model you build is a small experiment.

**Note:** One of your main goals should be to minimize the time between your experiments. The more experiments you do, the more things you'll figure out which don't work and in turn, get closer to figuring out what does work. Remember the machine learning practitioner's motto: "experiment, experiment, experiment".

Another thing you'll also find is what you thought may work (such as training a model for longer) may not always work and the exact opposite is also often the case.

## Tracking your experiments

One really good habit to get into is tracking your modelling experiments to see which perform better than others.

We've done a simple version of this above (keeping the results in different variables).

**Resource:** But as you build more models, you'll want to look into using tools such as:

- [TensorBoard](#) - a component of the TensorFlow library to help track modelling experiments (we'll see this later).
- [Weights & Biases](#) - a tool for tracking all kinds of machine learning experiments (the good news for Weights & Biases is it plugs into TensorBoard).

## Saving a model

Once you've trained a model and found one which performs to your liking, you'll probably want to save it for use elsewhere (like a web application or mobile device).

You can save a TensorFlow/Keras model using `model.save()`.

There are two ways to save a model in TensorFlow:

1. The [SavedModel format](#) (default).
2. The [HDF5 format](#).

The main difference between the two is the SavedModel is automatically able to save custom objects (such as special layers) without additional modifications when loading the model back in.

Which one should you use?

It depends on your situation but the SavedModel format will suffice most of the time.

**Both methods use the same method call.**

In [ ]:

```
# Save a model using the SavedModel format
model_2.save('best_model_SavedModel_format')
```

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/training/tracking/tracking.py:111: Model.state\_updates (from tensorflow.python.keras.engine.training) is deprecated and will be removed in a future version.

Instructions for updating:

This property should not be used in TensorFlow 2.0, as updates are applied automatically.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/training/tracking/tracking.py:111: Layer.updates (from tensorflow.python.keras.engine.base\_layer) is deprecated and will be removed in a future version.

Instructions for updating:

This property should not be used in TensorFlow 2.0, as updates are applied automatically.

INFO:tensorflow:Assets written to: best\_model\_SavedModel\_format/assets

In [ ]:

```
# Check it out - outputs a protobuf binary file (.pb) as well as other files
!ls best_model_SavedModel_format
```

assets saved\_model.pb variables

**Now let's save the model in the HDF5 format, we'll use the same method but with a different filename.**

In [ ]:

```
# Save a model using the HDF5 format
model_2.save("best_model_HDF5_format.h5") # note the addition of '.h5' on the end
```

In [ ]:

```
# Check it out
!ls best_model_HDF5_format.h5
```

best\_model\_HDF5\_format.h5

## Loading a model

We can load a saved model using the [load\\_model\(\)](#) method.

Loading a model for the different formats (SavedModel and HDF5) is the same (as long as the pathnames to the particular formats are correct).

In [ ]:

```
# Load a model from the SavedModel format
loaded_saved_model = tf.keras.models.load_model("best_model_SavedModel_format")
loaded_saved_model.summary()
```

Model: "sequential\_9"

Layer (type)	Output Shape	Param #
dense_9 (Dense)	(None, 1)	2
dense_10 (Dense)	(None, 1)	2
Total params: 4		
Trainable params: 4		
Non-trainable params: 0		

**Now let's test it out.**

In [ ]:

```
# Compare model_2 with the SavedModel version (should return True)
model_2_preds = model_2.predict(X_test)
loaded_saved_model = tf.saved_model.load('best_model_SavedModel')
loaded_saved_model_preds = loaded_saved_model(X_test)
mae(y_test, loaded_saved_model_preds.squeeze()).numpy() == mae(y_test, model_2_preds.squeeze()).numpy()
```

WARNING:tensorflow:7 out of the last 8 calls to <function Model.make\_predict\_function.<locals>.predict\_function at 0x7fa239a2df28> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has experimental\_relax\_shapes=True option that relaxes argument shapes that can avoid unnecessary retracing. For (3), please refer to [https://www.tensorflow.org/tutorials/customization/performance#python\\_or\\_tensor\\_args](https://www.tensorflow.org/tutorials/customization/performance#python_or_tensor_args) and [https://www.tensorflow.org/api\\_docs/python/tf/function](https://www.tensorflow.org/api_docs/python/tf/function) for more details.

Out [ ]:

True

## Loading in from the HDF5 is much the same.

In [ ]:

```
# Load a model from the HDF5 format
loaded_h5_model = tf.keras.models.load_model("best_model_HDF5_format.h5")
loaded_h5_model.summary()
```

Model: "sequential\_9"

Layer (type)	Output Shape	Param #
dense_9 (Dense)	(None, 1)	2
dense_10 (Dense)	(None, 1)	2
Total params: 4		
Trainable params: 4		
Non-trainable params: 0		

In [ ]:

```
# Compare model_2 with the loaded HDF5 version (should return True)
h5_model_preds = loaded_h5_model.predict(X_test)
mae(y_test, h5_model_preds.squeeze()).numpy() == mae(y_test, model_2_preds.squeeze()).numpy()
```

WARNING:tensorflow:8 out of the last 9 calls to <function Model.make\_predict\_function.<locals>.predict\_function at 0x7fa239732e18> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has experimental\_relax\_shapes=True option that relaxes argument shapes that can avoid unnecessary retracing. For (3), please refer to [https://www.tensorflow.org/tutorials/customization/performance#python\\_or\\_tensor\\_args](https://www.tensorflow.org/tutorials/customization/performance#python_or_tensor_args) and [https://www.tensorflow.org/api\\_docs/python/tf/function](https://www.tensorflow.org/api_docs/python/tf/function) for more details.

Out [ ]:

True

## Downloading a model (from Google Colab)

Say you wanted to get your model from Google Colab to your local machine, you can do one of the following things:

- Right click on the file in the files pane and click 'download'.

- Use the code below.

In [ ]:

```
# Download the model (or any file) from Google Colab
from google.colab import files
files.download("best_model_HDF5_format.h5")
```

## A larger example

Alright, we've seen the fundamentals of building neural network regression models in TensorFlow.

Let's step it up a notch and build a model for a more feature rich dataset.

More specifically we're going to try predict the cost of medical insurance for individuals based on a number of different parameters such as, `age`, `sex`, `bmi`, `children`, `smoking_status` and `residential_region`.

To do, we'll leverage the publicly available [Medical Cost dataset](#) available from Kaggle and [hosted on GitHub](#).

**Note:** When learning machine learning paradigms, you'll often go through a series of foundational techniques and then practice them by working with open-source datasets and examples. Just as we're doing now, learn foundations, put them to work with different problems. Every time you work on something new, it's a good idea to search for something like "problem X example with Python/TensorFlow" where you substitute X for your problem.

In [ ]:

```
# Import required libraries
import tensorflow as tf
import pandas as pd
import matplotlib.pyplot as plt
```

In [ ]:

```
# Read in the insurance dataset
insurance = pd.read_csv("https://raw.githubusercontent.com/stedy/Machine-Learning-with-R-datasets/master/insurance.csv")
```

In [ ]:

```
# Check out the insurance dataset
insurance.head()
```

Out[ ]:

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.900	0	yes	southwest	16884.92400
1	18	male	33.770	1	no	southeast	1725.55230
2	28	male	33.000	3	no	southeast	4449.46200
3	33	male	22.705	0	no	northwest	21984.47061
4	32	male	28.880	0	no	northwest	3866.85520

We're going to have to turn the non-numerical columns into numbers (because a neural network can't handle non-numerical inputs).

To do so, we'll use the `get_dummies()` method in pandas.

It converts categorical variables (like the `sex`, `smoker` and `region` columns) into numerical variables using one-hot encoding.

In [ ]:

```
# Turn all categories into numbers
insurance_one_hot = pd.get_dummies(insurance)
insurance_one_hot.head() # view the converted columns
```

Out[ ]:

age	bmi	children	charges	sex_female	sex_male	smoker_no	smoker_yes	region_northeast	region_northwest	region_southeast	region_southwest
0	19	27.900	0	16884.92400	1	0	0	1	0	0	0
1	18	33.770	1	1725.55230	0	1	1	0	0	0	0
2	28	33.000	3	4449.46200	0	1	1	0	0	0	0
3	33	22.705	0	21984.47061	0	1	1	0	0	0	1
4	32	28.880	0	3866.85520	0	1	1	0	0	0	1

Now we'll split data into features ( `X` ) and labels ( `y` ).

In [ ]:

```
# Create X & y values
X = insurance_one_hot.drop("charges", axis=1)
y = insurance_one_hot["charges"]
```

In [ ]:

```
# View features
X.head()
```

Out[ ]:

age	bmi	children	sex_female	sex_male	smoker_no	smoker_yes	region_northeast	region_northwest	region_southeast	region_southwest
0	19	27.900	0	1	0	0	1	0	0	0
1	18	33.770	1	0	1	1	0	0	0	0
2	28	33.000	3	0	1	1	0	0	0	0
3	33	22.705	0	0	1	1	0	0	0	1
4	32	28.880	0	0	1	1	0	0	0	1

And create training and test sets. We could do this manually, but to make it easier, we'll leverage the already available `train_test_split` function available from Scikit-Learn.

In [ ]:

```
# Create training and test sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.2,
                                                    random_state=42) # set random state
for reproducible splits
```

Now we can build and fit a model (we'll make it the same as `model_2`).

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Create a new model (same as model_2)
```

```

insurance_model = tf.keras.Sequential([
    tf.keras.layers.Dense(1),
    tf.keras.layers.Dense(1)
])

# Compile the model
insurance_model.compile(loss=tf.keras.losses.mae,
                        optimizer=tf.keras.optimizers.SGD(),
                        metrics=['mae'])

# Fit the model
insurance_model.fit(X_train, y_train, epochs=100)

```

In [ ]:

```

# Check the results of the insurance model
insurance_model.evaluate(X_test, y_test)

```

```
9/9 [=====] - 0s 1ms/step - loss: 8628.2363 - mae: 8628.2363
```

Out[ ]:

```
[8628.236328125, 8628.236328125]
```

**Our model didn't perform very well, let's try a bigger model.**

We'll try 3 things:

- Increasing the number of layers (2 -> 3).
- Increasing the number of units in each layer (except for the output layer).
- Changing the optimizer (from SGD to Adam).

Everything else will stay the same.

In [ ]:

```

# Set random seed
tf.random.set_seed(42)

# Add an extra layer and increase number of units
insurance_model_2 = tf.keras.Sequential([
    tf.keras.layers.Dense(100), # 100 units
    tf.keras.layers.Dense(10), # 10 units
    tf.keras.layers.Dense(1) # 1 unit (important for output layer)
])

# Compile the model
insurance_model_2.compile(loss=tf.keras.losses.mae,
                           optimizer=tf.keras.optimizers.Adam(), # Adam works but SGD doesn't
                           metrics=['mae'])

# Fit the model and save the history (we can plot this)
history = insurance_model_2.fit(X_train, y_train, epochs=100, verbose=0)

```

WARNING:tensorflow:Layer dense\_37 is casting an input tensor from dtype float64 to the layer's dtype of float32, which is new behavior in TensorFlow 2. The layer has dtype float 32 because its dtype defaults to floatx.

If you intended to run this layer in float32, you can safely ignore this warning. If in doubt, this warning is likely only an issue if you are porting a TensorFlow 1.X model to TensorFlow 2.

To change all layers to have dtype float64 by default, call `tf.keras.backend.set\_floatx('float64')`. To change just this layer, pass `dtype='float64'` to the layer constructor. If you are the author of this layer, you can disable autocasting by passing `autocast=False` to the base Layer constructor.

In [ ]:

```
# Evaluate our larger model
insurance_model_2.evaluate(X_test, y_test)

9/9 [=====] - 0s 1ms/step - loss: 4924.3477 - mae: 4924.3477

Out[ ]:

[4924.34765625, 4924.34765625]
```

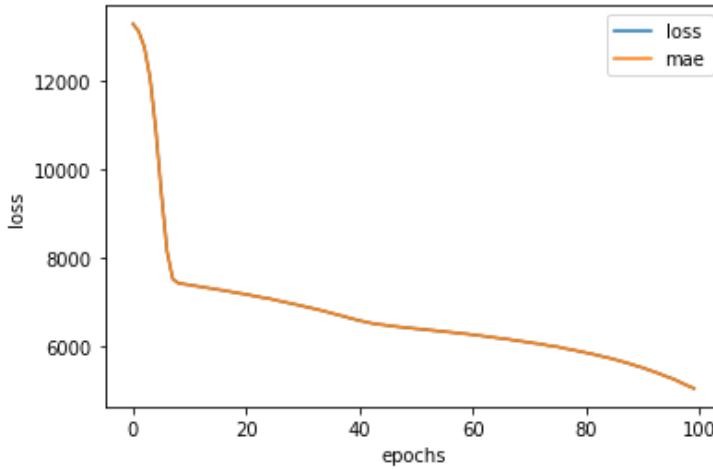
Much better! Using a larger model and the Adam optimizer results in almost half the error as the previous model.

□ Note: For many problems, the [Adam optimizer](#) is a great starting choice. See Andrei Karpathy's "Adam is safe" point from [A Recipe for Training Neural Networks](#) for more.

Let's check out the loss curves of our model, we should see a downward trend.

```
In [ ]:
```

```
# Plot history (also known as a loss curve)
pd.DataFrame(history.history).plot()
plt.ylabel("loss")
plt.xlabel("epochs");
```



From this, it looks like our model's loss (and MAE) were both still decreasing (in our case, MAE and loss are the same, hence the lines in the plot overlap eachother).

What this tells us is the loss might go down if we try training it for longer.

□ Question: How long should you train for?

It depends on what problem you're working on. Sometimes training won't take very long, other times it'll take longer than you expect. A common method is to set your model training for a very long time (e.g. 1000's of epochs) but set it up with an [EarlyStopping callback](#) so it stops automatically when it stops improving. We'll see this in another module.

Let's train the same model as above for a little longer. We can do this by calling fit on it again.

```
In [ ]:
```

```
# Try training for a little longer (100 more epochs)
history_2 = insurance_model_2.fit(X_train, y_train, epochs=100, verbose=0)
```

How did the extra training go?

```
In [ ]:
```

```
# Evaluate the model trained for 200 total epochs
insurance_model_2_loss, insurance_model_2_mae = insurance_model_2.evaluate(X_test, y_test)
```

```
insurance_model_2_loss, insurance_model_2_mae
```

```
9/9 [=====] - 0s 1ms/step - loss: 3494.7285 - mae: 3494.7285
```

Out[ ]:

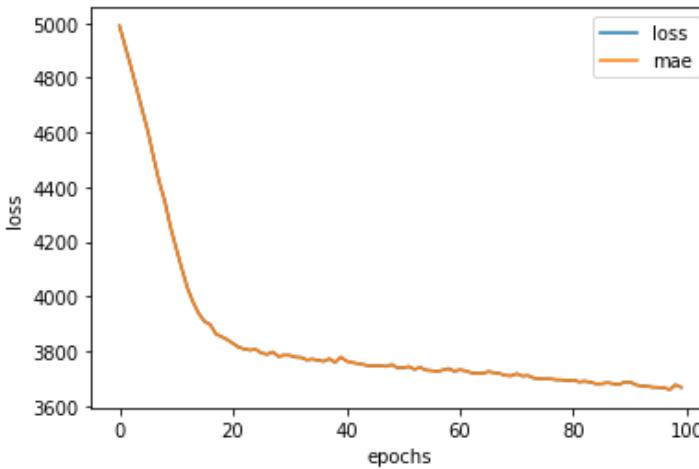
```
(3494.728515625, 3494.728515625)
```

Boom! Training for an extra 100 epochs we see about a 10% decrease in error.

How does the visual look?

In [ ]:

```
# Plot the model trained for 200 total epochs loss curves
pd.DataFrame(history_2.history).plot()
plt.ylabel("loss")
plt.xlabel("epochs"); # note: epochs will only show 100 since we overrid the history variable
```



## Preprocessing data (normalization and standardization)

A common practice when working with neural networks is to make sure all of the data you pass to them is in the range 0 to 1.

This practice is called **normalization** (scaling all values from their original range to, e.g. between 0 and 100,000 to be between 0 and 1).

There is another process call **standardization** which converts all of your data to unit variance and 0 mean.

These two practices are often part of a preprocessing pipeline (a series of functions to prepare your data for use with neural networks).

Knowing this, some of the major steps you'll take to preprocess your data for a neural network include:

- Turning all of your data to numbers (a neural network can't handle strings).
- Making sure your data is in the right shape (verifying input and output shapes).
- **Feature scaling:**
  - Normalizing data (making sure all values are between 0 and 1). This is done by subtracting the minimum value then dividing by the maximum value minus the minimum. This is also referred to as min-max scaling.
  - Standardization (making sure all values have a mean of 0 and a variance of 1). This is done by subtracting the mean value from the target feature and then dividing it by the standard deviation.
  - Which one should you use?
    - With neural networks you'll tend to favour normalization as they tend to prefer values between 0 and 1 (you'll see this especially with image processing), however, you'll often find a neural network can perform pretty well with minimal feature scaling.

Resource: For more on preprocessing data, I'd recommend reading the following resources:

- [Scikit-Learn's documentation on preprocessing data](#)

- [Scale, Standardize or Normalize with Scikit-Learn by Jeff Hale](#).

We've already turned our data into numbers using `get_dummies()`, let's see how we'd normalize it as well.

In [ ]:

```
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf

# Read in the insurance dataset
insurance = pd.read_csv("https://raw.githubusercontent.com/stedy/Machine-Learning-with-R-datasets/master/insurance.csv")
```

In [ ]:

```
# Check out the data
insurance.head()
```

Out[ ]:

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.900	0	yes	southwest	16884.92400
1	18	male	33.770	1	no	southeast	1725.55230
2	28	male	33.000	3	no	southeast	4449.46200
3	33	male	22.705	0	no	northwest	21984.47061
4	32	male	28.880	0	no	northwest	3866.85520

Now, just as before, we need to transform the non-numerical columns into numbers and this time we'll also be normalizing the numerical columns with different ranges (to make sure they're all between 0 and 1).

To do this, we're going to use a few classes from Scikit-Learn:

- [make\\_column\\_transformer](#) - build a multi-step data preprocessing function for the following transformations:
  - [MinMaxScaler](#) - make sure all numerical columns are normalized (between 0 and 1).
  - [OneHotEncoder](#) - one hot encode the non-numerical columns.

Let's see them in action.

In [ ]:

```
from sklearn.compose import make_column_transformer
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder

# Create column transformer (this will help us normalize/preprocess our data)
ct = make_column_transformer(
    (MinMaxScaler(), ["age", "bmi", "children"]), # get all values between 0 and 1
    (OneHotEncoder(handle_unknown="ignore"), ["sex", "smoker", "region"])
)

# Create X & y
X = insurance.drop("charges", axis=1)
y = insurance["charges"]

# Build our train and test sets (use random state to ensure same split as before)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Fit column transformer on the training data only (doing so on test data would result in data leakage)
ct.fit(X_train)

# Transform training and test data with normalization (MinMaxScalar) and one hot encoding
```

```
(OneHotEncoder)
```

```
X_train_normal = ct.transform(X_train)
X_test_normal = ct.transform(X_test)
```

**Now we've normalized it and one-hot encoding it, what does our data look like now?**

In [ ]:

```
# Non-normalized and non-one-hot encoded data example
X_train.loc[0]
```

Out[ ]:

```
age           19
sex      female
bmi        27.9
children       0
smoker     yes
region   southwest
Name: 0, dtype: object
```

In [ ]:

```
# Normalized and one-hot encoded example
X_train_normal[0]
```

Out[ ]:

```
array([0.60869565, 0.10734463, 0.4         , 1.         , 0.         ,
       1.         , 0.         , 0.         , 1.         , 0.         ,
       0.         ])
```

**How about the shapes?**

In [ ]:

```
# Notice the normalized/one-hot encoded shape is larger because of the extra columns
X_train_normal.shape, X_train.shape
```

Out[ ]:

```
((1070, 11), (1070, 6))
```

**Our data is normalized and numerical, let's model it.**

We'll use the same model as `insurance_model_2`.

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Build the model (3 layers, 100, 10, 1 units)
insurance_model_3 = tf.keras.Sequential([
    tf.keras.layers.Dense(100),
    tf.keras.layers.Dense(10),
    tf.keras.layers.Dense(1)
])

# Compile the model
insurance_model_3.compile(loss=tf.keras.losses.mae,
                           optimizer=tf.keras.optimizers.Adam(),
                           metrics=['mae'])

# Fit the model for 200 epochs (same as insurance_model_2)
insurance_model_3.fit(X_train_normal, y_train, epochs=200, verbose=0)
```

Out[ ]:

```
<tensorflow.python.keras.callbacks.History at 0x7fa236315b00>
```

Let's evaluate the model on normalized test set.

In [ ]:

```
# Evaluate 3rd model
insurance_model_3_loss, insurance_model_3_mae = insurance_model_3.evaluate(X_test_normal,
y_test)
```

9/9 [=====] - 0s 1ms/step - loss: 3171.7632 - mae: 3171.7632

And finally, let's compare the results from `insurance_model_2` (trained on non-normalized data) and `insurance_model_3` (trained on normalized data).

In [ ]:

```
# Compare modelling results from non-normalized data and normalized data
insurance_model_2_mae, insurance_model_3_mae
```

Out [ ]:

(3494.728515625, 3171.76318359375)

From this we can see normalizing the data results in 10% less error using the same model than not normalizing the data.

This is one of the main benefits of normalization: faster convergence time (a fancy way of saying, your model gets to better results faster).

`insurance_model_2` may have eventually achieved the same results as `insurance_model_3` if we left it training for longer.

Also, the results may change if we were to alter the architectures of the models, e.g. more hidden units per layer or more layers.

But since our main goal as neural network practitioners is to decrease the time between experiments, anything that helps us get better results sooner is a plus.

## Exercises

We've covered a whole lot pretty quickly.

So now it's time to have a play around with a few things and start to build up your intuition.

I emphasise the words play around because that's very important. Try a few things out, run the code and see what happens.

1. Create your own regression dataset (or make the one we created in "Create data to view and fit" bigger) and build fit a model to it.
2. Try building a neural network with 4 Dense layers and fitting it to your own regression dataset, how does it perform?
3. Try and improve the results we got on the insurance dataset, some things you might want to try include:
  - Building a larger model (how does one with 4 dense layers go?).
  - Increasing the number of units in each layer.
  - Lookup the documentation of [Adam](#) and find out what the first parameter is, what happens if you increase it by 10x?
  - What happens if you train for longer (say 300 epochs instead of 200)?
4. Import the [Boston pricing dataset](#) from TensorFlow `tf.keras.datasets` and model it.

## Extra curriculum

If you're looking for extra materials relating to this notebook, I'd check out the following:

- [MIT introduction deep learning lecture 1](#) - gives a great overview of what's happening behind all of the code

we're running.

- Reading: 1-hour of [Chapter 1 of Neural Networks and Deep Learning](#) by Michael Nielson - a great in-depth and hands-on example of the intuition behind neural networks.

To practice your regression modelling with TensorFlow, I'd also encourage you to look through [Lion Bridge's collection of datasets](#) or [Kaggle's datasets](#), find a regression dataset which sparks your interest and try to model.

## 02. Neural Network Classification with TensorFlow

Okay, we've seen how to deal with a regression problem in TensorFlow, let's look at how we can approach a classification problem.

A [classification problem](#) involves predicting whether something is one thing or another.

For example, you might want to:

- Predict whether or not someone has heart disease based on their health parameters. This is called **binary classification** since there are only two options.
- Decide whether a photo of is of food, a person or a dog. This is called **multi-class classification** since there are more than two options.
- Predict what categories should be assigned to a Wikipedia article. This is called **multi-label classification** since a single article could have more than one category assigned.

In this notebook, we're going to work through a number of different classification problems with TensorFlow. In other words, taking a set of inputs and predicting what class those set of inputs belong to.

### What we're going to cover

Specifically, we're going to go through doing the following with TensorFlow:

- Architecture of a classification model
- Input shapes and output shapes
  - $X$  : features/data (inputs)
  - $Y$  : labels (outputs)
    - "What class do the inputs belong to?"
- Creating custom data to view and fit
- Steps in modelling for binary and multiclass classification
  - Creating a model
  - Compiling a model
    - Defining a loss function
    - Setting up an optimizer
      - Finding the best learning rate
    - Creating evaluation metrics
  - Fitting a model (getting it to find patterns in our data)
  - Improving a model
- The power of non-linearity
- Evaluating classification models
  - Visualizing the model ("visualize, visualize, visualize")
  - Looking at training curves
  - Compare predictions to ground truth (using our evaluation metrics)

### How you can use this notebook

You can read through the descriptions and the code (it should all run, except for the cells which error on purpose), but there's a better option.

Write all of the code yourself.

Yes. I'm serious. Create a new notebook, and rewrite each line by yourself. Investigate it, see if you can break it, why does it break?

You don't have to write the text descriptions but writing the code yourself is a great way to get hands-on experience.

Don't worry if you make mistakes, we all do. The way to get better and make less mistakes is to **write more code**.

## Typical architecture of a classification neural network

The word *typical* is on purpose.

Because the architecture of a classification neural network can widely vary depending on the problem you're working on.

However, there are some fundamentals all deep neural networks contain:

- An input layer.
- Some hidden layers.
- An output layer.

Much of the rest is up to the data analyst creating the model.

The following are some standard values you'll often use in your classification neural networks.

Hyperparameter	Binary Classification	Multiclass classification
Input layer shape	Same as number of features (e.g. 5 for age, sex, height, weight, smoking status in heart disease prediction)	Same as binary classification
Hidden layer(s)	Problem specific, minimum = 1, maximum = unlimited	Same as binary classification
Neurons per hidden layer	Problem specific, generally 10 to 100	Same as binary classification
Output layer shape	1 (one class or the other)	1 per class (e.g. 3 for food, person or dog photo)
Hidden activation	Usually <a href="#">ReLU</a> (rectified linear unit)	Same as binary classification
Output activation	<a href="#">Sigmoid</a>	<a href="#">Softmax</a>
Loss function	<a href="#">Cross entropy</a> ( <code>tf.keras.losses.BinaryCrossentropy</code> in TensorFlow)	<a href="#">Cross entropy</a> ( <code>tf.keras.losses.CategoricalCrossentropy</code> in TensorFlow)
Optimizer	<a href="#">SGD</a> (stochastic gradient descent), <a href="#">Adam</a>	Same as binary classification

**Table 1: Typical architecture of a classification network. Source: Adapted from page 295 of [Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow Book by Aurélien Géron](#)**

Don't worry if not much of the above makes sense right now, we'll get plenty of experience as we go through this notebook.

Let's start by importing TensorFlow as the common alias `tf`. For this notebook, make sure you're using version 2.x+.

In [ ]:

```
import tensorflow as tf
print(tf.__version__)
```

2.3.0

## Creating data to view and fit

We could start by importing a classification dataset but let's practice making some of our own classification data.

**Note:** It's a common practice to get you and model you build working on a toy (or simple) dataset before moving to your actual problem. Treat it as a rehearsal experiment before the actual experiment(s).

Since classification is predicting whether something is one thing or another, let's make some data to reflect that.

To do so, we'll use Scikit-Learn's `make_circles()` function.

In [ ]:

```
from sklearn.datasets import make_circles

# Make 1000 examples
n_samples = 1000

# Create circles
X, y = make_circles(n_samples,
                     noise=0.03,
                     random_state=42)
```

Wonderful, now we've created some data, let's look at the features (`X`) and labels (`y`).

In [ ]:

```
# Check out the features
X
```

Out[ ]:

```
array([[ 0.75424625,  0.23148074],
       [-0.75615888,  0.15325888],
       [-0.81539193,  0.17328203],
       ...,
       [-0.13690036, -0.81001183],
       [ 0.67036156, -0.76750154],
       [ 0.28105665,  0.96382443]])
```

In [ ]:

```
# See the first 10 labels
y[:10]
```

Out[ ]:

```
array([1, 1, 1, 1, 0, 1, 1, 1, 1, 0])
```

Okay, we've seen some of our data and labels, how about we move towards visualizing?

**Note:** One important step of starting any kind of machine learning project is to [become one with the data](#). And one of the best ways to do this is to visualize the data you're working with as much as possible. The data explorer's motto is "visualize, visualize, visualize".

We'll start with a DataFrame.

In [ ]:

```
# Make dataframe of features and labels
import pandas as pd
circles = pd.DataFrame({ "X0": X[:, 0], "X1": X[:, 1], "label": y })
circles.head()
```

Out[ ]:

	X0	X1	label
0	0.754246	0.231481	1
1	-0.756159	0.153259	1
2	-0.815392	0.173282	1
3	-0.393731	0.692883	1

	x0	x1	label
4	0.442208	-0.896723	0

What kind of labels are we dealing with?

In [ ]:

```
# Check out the different labels
circles.label.value_counts()
```

Out[ ]:

```
1    500
0    500
Name: label, dtype: int64
```

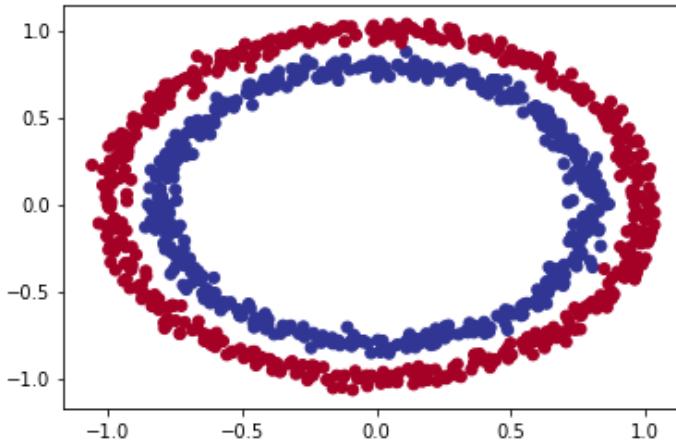
Alright, looks like we're dealing with a **binary classification** problem. It's binary because there are only two labels (0 or 1).

If there were more label options (e.g. 0, 1, 2, 3 or 4), it would be called **multiclass classification**.

Let's take our visualization a step further and plot our data.

In [ ]:

```
# Visualize with a plot
import matplotlib.pyplot as plt
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.RdYlBu);
```



Nice! From the plot, can you guess what kind of model we might want to build?

How about we try and build one to classify blue or red dots? As in, a model which is able to distinguish blue from red dots.

**Practice:** Before pushing forward, you might want to spend 10 minutes playing around with the [TensorFlow Playground](#). Try adjusting the different hyperparameters you see and click play to see a neural network train. I think you'll find the data very similar to what we've just created.

## Input and output shapes

One of the most common issues you'll run into when building neural networks is shape mismatches.

More specifically, the shape of the input data and the shape of the output data.

In our case, we want to input `X` and get our model to predict `y`.

So let's check out the shapes of `X` and `y`.

In [ ]:

```
# Check the shapes of our features and labels  
X.shape, y.shape
```

Out[ ]:

```
((1000, 2), (1000,))
```

Hmm, where do these numbers come from?

In [ ]:

```
# Check how many samples we have  
len(X), len(y)
```

Out[ ]:

```
(1000, 1000)
```

So we've got as many `X` values as we do `y` values, that makes sense.

Let's check out one example of each.

In [ ]:

```
# View the first example of features and labels  
X[0], y[0]
```

Out[ ]:

```
(array([0.75424625, 0.23148074]), 1)
```

Alright, so we've got two `X` features which lead to one `y` value.

This means our neural network input shape will have to accept a tensor with at least one dimension being two and output a tensor with at least one value.

**Note:** `y` having a shape of `(1000,)` can seem confusing. However, this is because all `y` values are actually scalars (single values) and therefore don't have a dimension. For now, think of your output shape as being at least the same value as one example of `y` (in our case, the output from our neural network has to be at least one value).

## Steps in modelling

Now we know what data we have as well as the input and output shapes, let's see how we'd build a neural network to model it.

In TensorFlow, there are typically 3 fundamental steps to creating and training a model.

1. **Creating a model** - piece together the layers of a neural network yourself (using the [functional](#) or [sequential API](#)) or import a previously built model (known as transfer learning).
2. **Compiling a model** - defining how a model's performance should be measured (loss/metrics) as well as defining how it should improve (optimizer).
3. **Fitting a model** - letting the model try to find patterns in the data (how does `X` get to `y`).

Let's see these in action using the Sequential API to build a model for our regression data. And then we'll step through each.

In [ ]:

```
# Set random seed  
tf.random.set_seed(42)  
  
# 1. Create the model using the Sequential API  
model_1 = tf.keras.Sequential([
```

```

    tf.keras.layers.Dense(1)
])

# 2. Compile the model
model_1.compile(loss=tf.keras.losses.BinaryCrossentropy(), # binary since we are working
with 2 classes (0 & 1)
                  optimizer=tf.keras.optimizers.SGD(),
                  metrics=['accuracy'])

# 3. Fit the model
model_1.fit(X, y, epochs=5)

```

```

Epoch 1/5
32/32 [=====] - 0s 970us/step - loss: 2.8544 - accuracy: 0.4600
Epoch 2/5
32/32 [=====] - 0s 1ms/step - loss: 0.7131 - accuracy: 0.5430
Epoch 3/5
32/32 [=====] - 0s 924us/step - loss: 0.6973 - accuracy: 0.5090
Epoch 4/5
32/32 [=====] - 0s 1ms/step - loss: 0.6950 - accuracy: 0.5010
Epoch 5/5
32/32 [=====] - 0s 980us/step - loss: 0.6942 - accuracy: 0.4830

```

Out[ ]:

```
<tensorflow.python.keras.callbacks.History at 0x7f42561c6fd0>
```

**Looking at the accuracy metric, our model performs poorly (50% accuracy on a binary classification problem is the equivalent of guessing), but what if we trained it for longer?**

In [ ]:

```

# Train our model for longer (more chances to look at the data)
model_1.fit(X, y, epochs=200, verbose=0) # set verbose=0 to remove training updates
model_1.evaluate(X, y)

```

```
32/32 [=====] - 0s 850us/step - loss: 0.6935 - accuracy: 0.5000
```

Out[ ]:

```
[0.6934829950332642, 0.5]
```

**Even after 200 passes of the data, it's still performing as if it's guessing.**

**What if we added an extra layer and trained for a little longer?**

In [ ]:

```

# Set random seed
tf.random.set_seed(42)

# 1. Create the model (same as model_1 but with an extra layer)
model_2 = tf.keras.Sequential([
    tf.keras.layers.Dense(1), # add an extra layer
    tf.keras.layers.Dense(1)
])

# 2. Compile the model
model_2.compile(loss=tf.keras.losses.BinaryCrossentropy(),
                 optimizer=tf.keras.optimizers.SGD(),
                 metrics=['accuracy'])

# 3. Fit the model
model_2.fit(X, y, epochs=100, verbose=0) # set verbose=0 to make the output print less

```

Out[ ]:

```
<tensorflow.python.keras.callbacks.History at 0x7f4255995f60>
```

In [ ]:

```
# Evaluate the model
```

```

model_2.evaluate(X, y)

32/32 [=====] - 0s 807us/step - loss: 0.6933 - accuracy: 0.5000
Out[ ]:
[0.6933314800262451, 0.5]

```

Still not even as good as guessing (~50% accuracy)... hmm...?

Let's remind ourselves of a couple more ways we can use to improve our models.

## Improving a model

To improve our model, we can alter almost every part of the 3 steps we went through before.

- Creating a model** - here you might want to add more layers, increase the number of hidden units (also called neurons) within each layer, change the activation functions of each layer.
- Compiling a model** - you might want to choose a different optimization function (such as the [Adam](#) optimizer, which is usually pretty good for many problems) or perhaps change the learning rate of the optimization function.
- Fitting a model** - perhaps you could fit a model for more epochs (leave it training for longer).

The diagram illustrates the process of improving a neural network. It starts with a 'Smaller model' on the left and ends with a 'Larger model' on the right. A horizontal arrow points from left to right, indicating the progression. The code for both models is shown in boxes:

**Smaller model:**

```

# 1. Create the model (specified to your problem)
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(4, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])

# 2. Compile the model
model.compile(loss=tf.keras.losses.BinaryCrossentropy(),
              optimizer=tf.keras.optimizers.Adam(lr=0.001),
              metrics=['accuracy'])

# 3. Fit the model
model.fit(X_train_subset, y_train_subset, epochs=5)

```

**Larger model:**

```

# 1. Create the model (specified to your problem)
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])

# 2. Compile the model
model.compile(loss=tf.keras.losses.BinaryCrossentropy(),
              optimizer=tf.keras.optimizers.Adam(lr=0.0001),
              metrics=['accuracy'])

# 3. Fit the model
model.fit(X_train_full, y_train_full, epochs=100)

```

**Common ways to improve a deep model:**

- Adding layers
- Increase the number of hidden units
- Change the activation functions
- Change the optimization function
- Change the learning rate (because you can alter each of these, they're hyperparameters)
- Fitting on more data
- Fitting for longer

*There are many different ways to potentially improve a neural network. Some of the most common include: increasing the number of layers (making the network deeper), increasing the number of hidden units (making the network wider) and changing the learning rate. Because these values are all human-changeable, they're referred to as [hyperparameters](#)) and the practice of trying to find the best hyperparameters is referred to as [hyperparameter tuning](#).*

How about we try adding more neurons, an extra layer and our friend the Adam optimizer?

Surely doing this will result in predictions better than guessing...

In [ ]:

```

# Set random seed
tf.random.set_seed(42)

# 1. Create the model (this time 3 layers)
model_3 = tf.keras.Sequential([
    tf.keras.layers.Dense(100), # add 100 dense neurons
    tf.keras.layers.Dense(10), # add another layer with 10 neurons
])

```

```

    tf.keras.layers.Dense(1)
])

# 2. Compile the model
model_3.compile(loss=tf.keras.losses.BinaryCrossentropy(),
                 optimizer=tf.keras.optimizers.Adam(), # use Adam instead of SGD
                 metrics=['accuracy'])

# 3. Fit the model
model_3.fit(X, y, epochs=100, verbose=0) # fit for 100 passes of the data

```

Out[ ]:

```
<tensorflow.python.keras.callbacks.History at 0x7f4254f58ef0>
```

Still!

We've pulled out a few tricks but our model isn't even doing better than guessing.

Let's make some visualizations to see what's happening.

**Note:** Whenever your model is performing strangely or there's something going on with your data you're not quite sure of, remember these three words: **visualize, visualize, visualize**. Inspect your data, inspect your model, inspect your model's predictions.

To visualize our model's predictions we're going to create a function `plot_decision_boundary()` which:

- Takes in a trained model, features (`X`) and labels (`y`).
- Creates a meshgrid of the different `X` values.
- Makes predictions across the meshgrid.
- Plots the predictions as well as a line between the different zones (where each unique class falls).

If this sounds confusing, let's see it in code and then see the output.

**Note:** If you're ever unsure of what a function does, try unraveling it and writing it line by line for yourself to see what it does. Break it into small parts and see what each part outputs.

In [ ]:

```

import numpy as np

def plot_decision_boundary(model, X, y):
    """
    Plots the decision boundary created by a model predicting on X.
    This function has been adapted from two phenomenal resources:
    1. CS231n - https://cs231n.github.io/neural-networks-case-study/
    2. Made with ML basics - https://github.com/GokuMohandas/MadeWithML/blob/main/notebooks/08_Neural_Networks.ipynb
    """

    # Define the axis boundaries of the plot and create a meshgrid
    x_min, x_max = X[:, 0].min() - 0.1, X[:, 0].max() + 0.1
    y_min, y_max = X[:, 1].min() - 0.1, X[:, 1].max() + 0.1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100),
                         np.linspace(y_min, y_max, 100))

    # Create X values (we're going to predict on all of these)
    x_in = np.c_[xx.ravel(), yy.ravel()] # stack 2D arrays together: https://numpy.org/devdocs/reference/generated/numpy.c_.html

    # Make predictions using the trained model
    y_pred = model.predict(x_in)

    # Check for multi-class
    if len(y_pred[0]) > 1:
        print("doing multiclass classification...")
        # We have to reshape our predictions to get them ready for plotting
        y_pred = np.argmax(y_pred, axis=1).reshape(xx.shape)
    
```

```

else:
    print("doing binary classification...")
    y_pred = np.round(y_pred).reshape(xx.shape)

# Plot decision boundary
plt.contourf(xx, yy, y_pred, cmap=plt.cm.RdYlBu, alpha=0.7)
plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.RdYlBu)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())

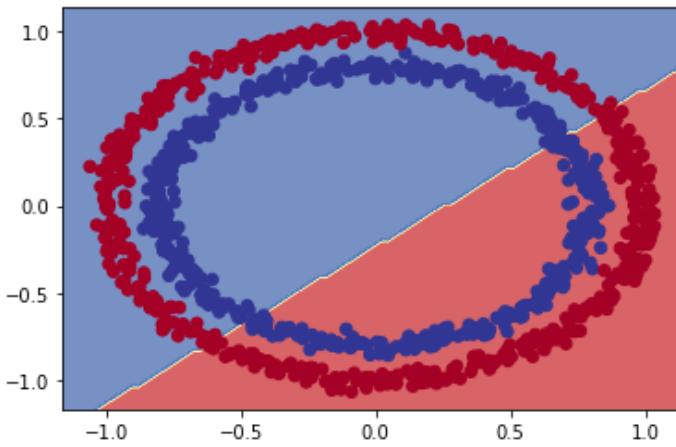
```

**Now we've got a function to plot our model's decision boundary (the cut off point its making between red and blue dots), let's try it out.**

In [ ]:

```
# Check out the predictions our model is making
plot_decision_boundary(model_3, X, y)
```

doing binary classification...



**Looks like our model is trying to draw a straight line through the data.**

**What's wrong with doing this?**

**The main issue is our data isn't separable by a straight line.**

**In a regression problem, our model might work. In fact, let's try it.**

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Create some regression data
X_regression = np.arange(0, 1000, 5)
y_regression = np.arange(100, 1100, 5)

# Split it into training and test sets
X_reg_train = X_regression[:150]
X_reg_test = X_regression[150:]
y_reg_train = y_regression[:150]
y_reg_test = y_regression[150:]

# Fit our model to the data
model_3.fit(X_reg_train, y_reg_train, epochs=100)
```

Epoch 1/100

---

```

-----  

ValueError                                Traceback (most recent call last)  

<ipython-input-18-ac5f57a9e452> in <module>()  

    13  

    14 # Fit our model to the data  

--> 15 model_3.fit(X_reg_train, y_reg_train, epochs=100)

```

```
/usr/local/lib/python3.6/dist-packages/tensorflow/python/keras/engine/training.py in __met
hod_wrapper(self, *args, **kwargs)
  106     def _method_wrapper(self, *args, **kwargs):
  107         if not self._in_multi_worker_mode(): # pylint: disable=protected-access
--> 108             return method(self, *args, **kwargs)
  109
 110     # Running inside `run_distribute_coordinator` already.

/usr/local/lib/python3.6/dist-packages/tensorflow/python/keras/engine/training.py in fit(
self, x, y, batch_size, epochs, verbose, callbacks, validation_split, validation_data, sh
uffle, class_weight, sample_weight, initial_epoch, steps_per_epoch, validation_steps, val
idation_batch_size, validation_freq, max_queue_size, workers, use_multiprocessing)
 1096                 batch_size=batch_size):
 1097                     callbacks.on_train_batch_begin(step)
--> 1098                     tmp_logs = train_function(iterator)
 1099                     if data_handler.should_sync:
 1100                         context.async_wait()

/usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/def_function.py in __call__(
self, *args, **kwds)
  778             else:
  779                 compiler = "nonXla"
--> 780             result = self._call(*args, **kwds)
  781
 782             new_tracing_count = self._get_tracing_count()

/usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/def_function.py in __call__(s
elf, *args, **kwds)
  805                 # In this case we have created variables on the first call, so we run the
  806                 # defunned version which is guaranteed to never create variables.
--> 807             return self._stateless_fn(*args, **kwds) # pylint: disable=not-callable
  808         elif self._stateful_fn is not None:
  809             # Release the lock early so that multiple threads can perform the call

/usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/function.py in __call__(se
lf, *args, **kwargs)
 2826         """Calls a graph function specialized to the inputs."""
 2827         with self._lock:
--> 2828             graph_function, args, kwargs = self._maybe_define_function(args, kwargs)
 2829             return graph_function._filtered_call(args, kwargs) # pylint: disable=protect
ed-access
 2830

/usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/function.py in _maybe_defi
ne_function(self, args, kwargs)
 3208                 and self.input_signature is None
 3209                 and call_context_key in self._function_cache.missed):
--> 3210             return self._define_function_with_shape_relaxation(args, kwargs)
 3211
 3212             self._function_cache.missed.add(call_context_key)

/usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/function.py in _define_fun
ction_with_shape_relaxation(self, args, kwargs)
 3140
 3141     graph_function = self._create_graph_function(
--> 3142         args, kwargs, override_flat_arg_shapes=relaxed_arg_shapes)
 3143     self._function_cache.arg_relaxed[rank_only_cache_key] = graph_function
 3144

/usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/function.py in _create_gra
ph_function(self, args, kwargs, override_flat_arg_shapes)
 3073             arg_names=arg_names,
 3074             override_flat_arg_shapes=override_flat_arg_shapes,
--> 3075             capture_by_value=self._capture_by_value),
 3076             self._function_attributes,
 3077             function_spec=self.function_spec,

/usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/func_graph.py in func_
graph_from_py_func(name, python_func, args, kwargs, signature, func_graph, autograph, aut
ograph_options, add_control_dependencies, arg_names, op_return_value, collections, captur
e_by_value, override_flat_arg_shapes)
 984             _, original_func = tf_decorator.unwrap(python_func)
```

```

985
--> 986     func_outputs = python_func(*func_args, **func_kwargs)
987
988     # invariant: `func_outputs` contains only Tensors, CompositeTensors,
989
/usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/def_function.py in wrapped_fn(*args, **kwds)
  598         # __wrapped__ allows AutoGraph to swap in a converted function. We give
  599         # the function a weak reference to itself to avoid a reference cycle.
--> 600     return weak_wrapped_fn().__wrapped__(*args, **kwds)
  601     weak_wrapped_fn = weakref.ref(wrapped_fn)
  602

/usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/func_graph.py in wrapper(*args, **kwargs)
  971         except Exception as e: # pylint:disable=broad-except
  972             if hasattr(e, "ag_error_metadata"):
--> 973                 raise e.ag_error_metadata.to_exception(e)
  974             else:
  975                 raise

ValueError: in user code:

    /usr/local/lib/python3.6/dist-packages/tensorflow/python/keras/engine/training.py:806
train_function *
        return step_function(self, iterator)
    /usr/local/lib/python3.6/dist-packages/tensorflow/python/keras/engine/training.py:796
step_function **
        outputs = model.distribute_strategy.run(run_step, args=(data,))
    /usr/local/lib/python3.6/dist-packages/tensorflow/python/distribute/distribute_lib.py
:1211 run
        return self._extended.call_for_each_replica(fn, args=args, kwargs=kwargs)
    /usr/local/lib/python3.6/dist-packages/tensorflow/python/distribute/distribute_lib.py
:2585 call_for_each_replica
        return self._call_for_each_replica(fn, args, kwargs)
    /usr/local/lib/python3.6/dist-packages/tensorflow/python/distribute/distribute_lib.py
:2945 _call_for_each_replica
        return fn(*args, **kwargs)
    /usr/local/lib/python3.6/dist-packages/tensorflow/python/keras/engine/training.py:789
run_step **
        outputs = model.train_step(data)
    /usr/local/lib/python3.6/dist-packages/tensorflow/python/keras/engine/training.py:747
train_step
        y_pred = self(x, training=True)
    /usr/local/lib/python3.6/dist-packages/tensorflow/python/keras/engine/base_layer.py:9
76 __call__
        self.name)
    /usr/local/lib/python3.6/dist-packages/tensorflow/python/keras/engine/input_spec.py:2
16 assert_input_compatibility
        ' but received input with shape ' + str(shape))

    ValueError: Input 0 of layer sequential_2 is incompatible with the layer: expected ax
is -1 of input shape to have value 2 but received input with shape [None, 1]

```

**Oh wait... we compiled our model for a binary classification problem.**

**No trouble, we can recreate it for a regression problem.**

In [ ]:

```

# Setup random seed
tf.random.set_seed(42)

# Recreate the model
model_3 = tf.keras.Sequential([
    tf.keras.layers.Dense(100),
    tf.keras.layers.Dense(10),
    tf.keras.layers.Dense(1)
])

# Change the loss and metrics of our compiled model

```

```
model_3.compile(loss=tf.keras.losses.mae, # change the loss function to be regression-specific
                 optimizer=tf.keras.optimizers.Adam(),
                 metrics=['mae']) # change the metric to be regression-specific
```

```
# Fit the recompiled model
```

```
model_3.fit(X_reg_train, y_reg_train, epochs=100)
```

```
Epoch 1/100
5/5 [=====] - 0s 2ms/step - loss: 248.2155 - mae: 248.2155
Epoch 2/100
5/5 [=====] - 0s 2ms/step - loss: 138.9005 - mae: 138.9005
Epoch 3/100
5/5 [=====] - 0s 2ms/step - loss: 53.1039 - mae: 53.1039
Epoch 4/100
5/5 [=====] - 0s 1ms/step - loss: 73.5170 - mae: 73.5170
Epoch 5/100
5/5 [=====] - 0s 1ms/step - loss: 71.2358 - mae: 71.2358
Epoch 6/100
5/5 [=====] - 0s 2ms/step - loss: 47.0040 - mae: 47.0040
Epoch 7/100
5/5 [=====] - 0s 1ms/step - loss: 45.9386 - mae: 45.9386
Epoch 8/100
5/5 [=====] - 0s 2ms/step - loss: 42.3638 - mae: 42.3638
Epoch 9/100
5/5 [=====] - 0s 1ms/step - loss: 43.6831 - mae: 43.6831
Epoch 10/100
5/5 [=====] - 0s 2ms/step - loss: 42.6198 - mae: 42.6198
Epoch 11/100
5/5 [=====] - 0s 1ms/step - loss: 42.4797 - mae: 42.4797
Epoch 12/100
5/5 [=====] - 0s 2ms/step - loss: 41.5537 - mae: 41.5537
Epoch 13/100
5/5 [=====] - 0s 1ms/step - loss: 42.0972 - mae: 42.0972
Epoch 14/100
5/5 [=====] - 0s 1ms/step - loss: 41.8647 - mae: 41.8647
Epoch 15/100
5/5 [=====] - 0s 2ms/step - loss: 41.5342 - mae: 41.5342
Epoch 16/100
5/5 [=====] - 0s 2ms/step - loss: 41.4028 - mae: 41.4028
Epoch 17/100
5/5 [=====] - 0s 3ms/step - loss: 41.6887 - mae: 41.6887
Epoch 18/100
5/5 [=====] - 0s 1ms/step - loss: 41.6137 - mae: 41.6137
Epoch 19/100
5/5 [=====] - 0s 2ms/step - loss: 41.2796 - mae: 41.2796
Epoch 20/100
5/5 [=====] - 0s 2ms/step - loss: 41.1947 - mae: 41.1947
Epoch 21/100
5/5 [=====] - 0s 3ms/step - loss: 41.2130 - mae: 41.2130
Epoch 22/100
5/5 [=====] - 0s 4ms/step - loss: 41.0893 - mae: 41.0893
Epoch 23/100
5/5 [=====] - 0s 3ms/step - loss: 41.2019 - mae: 41.2019
Epoch 24/100
5/5 [=====] - 0s 2ms/step - loss: 40.9989 - mae: 40.9989
Epoch 25/100
5/5 [=====] - 0s 2ms/step - loss: 41.0130 - mae: 41.0130
Epoch 26/100
5/5 [=====] - 0s 2ms/step - loss: 41.0654 - mae: 41.0654
Epoch 27/100
5/5 [=====] - 0s 1ms/step - loss: 40.8764 - mae: 40.8764
Epoch 28/100
5/5 [=====] - 0s 3ms/step - loss: 41.0545 - mae: 41.0545
Epoch 29/100
5/5 [=====] - 0s 2ms/step - loss: 41.0480 - mae: 41.0480
Epoch 30/100
5/5 [=====] - 0s 2ms/step - loss: 40.8807 - mae: 40.8807
Epoch 31/100
5/5 [=====] - 0s 1ms/step - loss: 41.2695 - mae: 41.2695
Epoch 32/100
5/5 [=====] - 0s 2ms/step - loss: 40.9949 - mae: 40.9949
```

Epoch 33/100  
5/5 [=====] - 0s 2ms/step - loss: 41.0760 - mae: 41.0760  
Epoch 34/100  
5/5 [=====] - 0s 2ms/step - loss: 41.2471 - mae: 41.2471  
Epoch 35/100  
5/5 [=====] - 0s 2ms/step - loss: 40.6102 - mae: 40.6102  
Epoch 36/100  
5/5 [=====] - 0s 2ms/step - loss: 41.1093 - mae: 41.1093  
Epoch 37/100  
5/5 [=====] - 0s 2ms/step - loss: 40.8191 - mae: 40.8191  
Epoch 38/100  
5/5 [=====] - 0s 2ms/step - loss: 40.2485 - mae: 40.2485  
Epoch 39/100  
5/5 [=====] - 0s 2ms/step - loss: 41.0625 - mae: 41.0625  
Epoch 40/100  
5/5 [=====] - 0s 2ms/step - loss: 40.5311 - mae: 40.5311  
Epoch 41/100  
5/5 [=====] - 0s 2ms/step - loss: 40.5497 - mae: 40.5497  
Epoch 42/100  
5/5 [=====] - 0s 2ms/step - loss: 40.4322 - mae: 40.4322  
Epoch 43/100  
5/5 [=====] - 0s 2ms/step - loss: 40.5367 - mae: 40.5367  
Epoch 44/100  
5/5 [=====] - 0s 2ms/step - loss: 40.2487 - mae: 40.2487  
Epoch 45/100  
5/5 [=====] - 0s 2ms/step - loss: 40.5151 - mae: 40.5151  
Epoch 46/100  
5/5 [=====] - 0s 2ms/step - loss: 40.3702 - mae: 40.3702  
Epoch 47/100  
5/5 [=====] - 0s 2ms/step - loss: 40.4769 - mae: 40.4769  
Epoch 48/100  
5/5 [=====] - 0s 2ms/step - loss: 40.1532 - mae: 40.1532  
Epoch 49/100  
5/5 [=====] - 0s 2ms/step - loss: 40.7291 - mae: 40.7291  
Epoch 50/100  
5/5 [=====] - 0s 2ms/step - loss: 40.1536 - mae: 40.1536  
Epoch 51/100  
5/5 [=====] - 0s 2ms/step - loss: 40.2711 - mae: 40.2711  
Epoch 52/100  
5/5 [=====] - 0s 3ms/step - loss: 40.6572 - mae: 40.6572  
Epoch 53/100  
5/5 [=====] - 0s 2ms/step - loss: 40.6573 - mae: 40.6573  
Epoch 54/100  
5/5 [=====] - 0s 2ms/step - loss: 40.6894 - mae: 40.6894  
Epoch 55/100  
5/5 [=====] - 0s 1ms/step - loss: 41.2771 - mae: 41.2771  
Epoch 56/100  
5/5 [=====] - 0s 2ms/step - loss: 41.8519 - mae: 41.8519  
Epoch 57/100  
5/5 [=====] - 0s 2ms/step - loss: 40.7903 - mae: 40.7903  
Epoch 58/100  
5/5 [=====] - 0s 2ms/step - loss: 40.3128 - mae: 40.3128  
Epoch 59/100  
5/5 [=====] - 0s 2ms/step - loss: 40.7198 - mae: 40.7198  
Epoch 60/100  
5/5 [=====] - 0s 2ms/step - loss: 40.1478 - mae: 40.1478  
Epoch 61/100  
5/5 [=====] - 0s 3ms/step - loss: 40.1117 - mae: 40.1117  
Epoch 62/100  
5/5 [=====] - 0s 2ms/step - loss: 40.7800 - mae: 40.7800  
Epoch 63/100  
5/5 [=====] - 0s 2ms/step - loss: 39.7242 - mae: 39.7242  
Epoch 64/100  
5/5 [=====] - 0s 1ms/step - loss: 40.1465 - mae: 40.1465  
Epoch 65/100  
5/5 [=====] - 0s 1ms/step - loss: 39.6887 - mae: 39.6887  
Epoch 66/100  
5/5 [=====] - 0s 2ms/step - loss: 40.2840 - mae: 40.2840  
Epoch 67/100  
5/5 [=====] - 0s 2ms/step - loss: 39.5541 - mae: 39.5541  
Epoch 68/100  
5/5 [=====] - 0s 2ms/step - loss: 39.7378 - mae: 39.7378

```
Epoch 69/100
5/5 [=====] - 0s 2ms/step - loss: 39.9785 - mae: 39.9785
Epoch 70/100
5/5 [=====] - 0s 2ms/step - loss: 40.0016 - mae: 40.0016
Epoch 71/100
5/5 [=====] - 0s 2ms/step - loss: 40.0913 - mae: 40.0913
Epoch 72/100
5/5 [=====] - 0s 2ms/step - loss: 39.2547 - mae: 39.2547
Epoch 73/100
5/5 [=====] - 0s 2ms/step - loss: 39.6828 - mae: 39.6828
Epoch 74/100
5/5 [=====] - 0s 2ms/step - loss: 39.5373 - mae: 39.5373
Epoch 75/100
5/5 [=====] - 0s 2ms/step - loss: 39.6265 - mae: 39.6265
Epoch 76/100
5/5 [=====] - 0s 2ms/step - loss: 39.3110 - mae: 39.3110
Epoch 77/100
5/5 [=====] - 0s 2ms/step - loss: 39.1599 - mae: 39.1599
Epoch 78/100
5/5 [=====] - 0s 2ms/step - loss: 39.7550 - mae: 39.7550
Epoch 79/100
5/5 [=====] - 0s 3ms/step - loss: 39.2542 - mae: 39.2542
Epoch 80/100
5/5 [=====] - 0s 3ms/step - loss: 38.6968 - mae: 38.6968
Epoch 81/100
5/5 [=====] - 0s 2ms/step - loss: 39.5442 - mae: 39.5442
Epoch 82/100
5/5 [=====] - 0s 3ms/step - loss: 39.8686 - mae: 39.8686
Epoch 83/100
5/5 [=====] - 0s 3ms/step - loss: 39.1693 - mae: 39.1693
Epoch 84/100
5/5 [=====] - 0s 2ms/step - loss: 38.8840 - mae: 38.8840
Epoch 85/100
5/5 [=====] - 0s 2ms/step - loss: 38.8887 - mae: 38.8887
Epoch 86/100
5/5 [=====] - 0s 4ms/step - loss: 38.6614 - mae: 38.6614
Epoch 87/100
5/5 [=====] - 0s 2ms/step - loss: 38.8398 - mae: 38.8398
Epoch 88/100
5/5 [=====] - 0s 2ms/step - loss: 38.6604 - mae: 38.6604
Epoch 89/100
5/5 [=====] - 0s 2ms/step - loss: 38.7559 - mae: 38.7559
Epoch 90/100
5/5 [=====] - 0s 2ms/step - loss: 38.5442 - mae: 38.5442
Epoch 91/100
5/5 [=====] - 0s 2ms/step - loss: 38.3247 - mae: 38.3247
Epoch 92/100
5/5 [=====] - 0s 2ms/step - loss: 38.8431 - mae: 38.8431
Epoch 93/100
5/5 [=====] - 0s 2ms/step - loss: 39.1137 - mae: 39.1137
Epoch 94/100
5/5 [=====] - 0s 2ms/step - loss: 38.1463 - mae: 38.1463
Epoch 95/100
5/5 [=====] - 0s 2ms/step - loss: 38.3998 - mae: 38.3998
Epoch 96/100
5/5 [=====] - 0s 2ms/step - loss: 38.5599 - mae: 38.5599
Epoch 97/100
5/5 [=====] - 0s 2ms/step - loss: 38.1038 - mae: 38.1038
Epoch 98/100
5/5 [=====] - 0s 1ms/step - loss: 39.0081 - mae: 39.0081
Epoch 99/100
5/5 [=====] - 0s 1ms/step - loss: 38.3056 - mae: 38.3056
Epoch 100/100
5/5 [=====] - 0s 5ms/step - loss: 37.9976 - mae: 37.9976
```

Out[ ]:

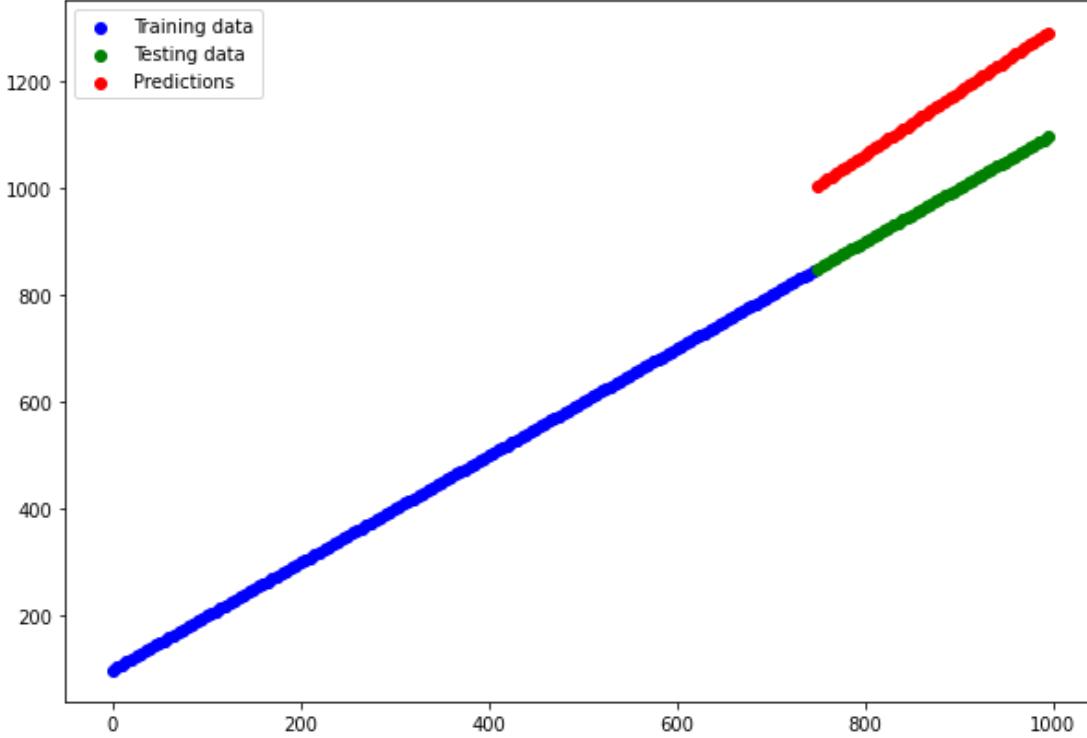
```
<tensorflow.python.keras.callbacks.History at 0x7f4254922f60>
```

Okay, it seems like our model is learning something (the `mae` value trends down with each epoch), let's plot its predictions.

In [ ]:

```
# Make predictions with our trained model
y_reg_preds = model_3.predict(y_reg_test)

# Plot the model's predictions against our regression data
plt.figure(figsize=(10, 7))
plt.scatter(X_reg_train, y_reg_train, c='b', label='Training data')
plt.scatter(X_reg_test, y_reg_test, c='g', label='Testing data')
plt.scatter(X_reg_test, y_reg_preds.squeeze(), c='r', label='Predictions')
plt.legend();
```



Okay, the predictions aren't perfect (if the predictions were perfect, the red would line up with the green), but they look better than complete guessing.

So this means our model must be learning something...

There must be something we're missing out on for our classification problem.

## The missing piece: Non-linearity

Okay, so we saw our neural network can model straight lines (with ability a little bit better than guessing).

What about non-straight (non-linear) lines?

If we're going to model our classification data (the red and blue circles), we're going to need some non-linear lines.

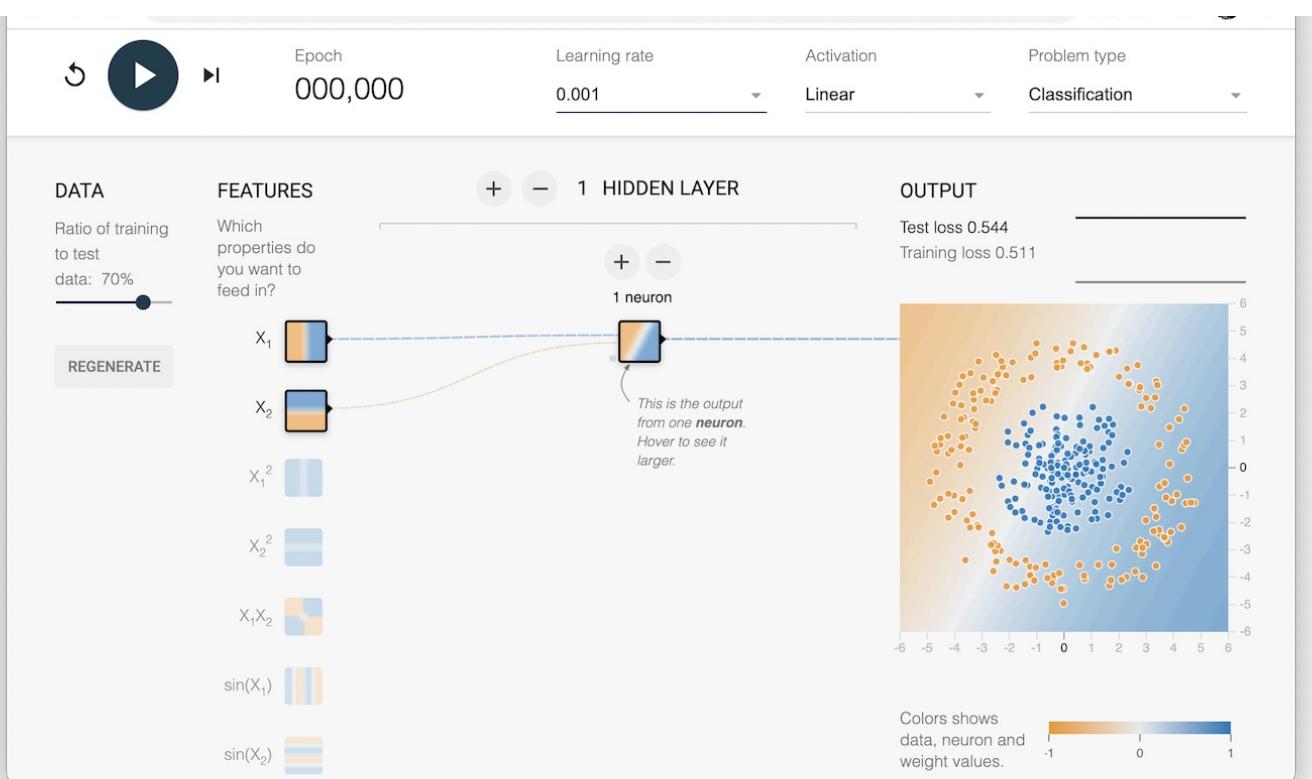
**I Practice:** Before we get to the next steps, I'd encourage you to play around with the [TensorFlow Playground](#) (check out what the data has in common with our own classification data) for 10-minutes. In particular the tab which says "activation". Once you're done, come back.

Did you try out the activation options? If so, what did you find?

If you didn't, don't worry, let's see it in code.

We're going to replicate the neural network you can see at this link: [TensorFlow Playground](#).





**The neural network we're going to recreate with TensorFlow code. See it live at [TensorFlow Playground](#).**

The main change we'll add to models we've built before is the use of the `activation` keyword.

In [ ]:

```
# Set the random seed
tf.random.set_seed(42)

# Create the model
model_4 = tf.keras.Sequential([
    tf.keras.layers.Dense(1, activation=tf.keras.activations.linear), # 1 hidden layer with linear activation
    tf.keras.layers.Dense(1) # output layer
])

# Compile the model
model_4.compile(loss=tf.keras.losses.binary_crossentropy,
                 optimizer=tf.keras.optimizers.Adam(lr=0.001), # "lr" is short for "learning rate"
                 metrics=["accuracy"])

# Fit the model
history = model_4.fit(X, y, epochs=100)
```

```
Epoch 1/100
32/32 [=====] - 0s 972us/step - loss: 4.2380 - accuracy: 0.5000
Epoch 2/100
32/32 [=====] - 0s 937us/step - loss: 4.0223 - accuracy: 0.5000
Epoch 3/100
32/32 [=====] - 0s 996us/step - loss: 3.8296 - accuracy: 0.5000
Epoch 4/100
32/32 [=====] - 0s 905us/step - loss: 3.7654 - accuracy: 0.5000
Epoch 5/100
32/32 [=====] - 0s 1ms/step - loss: 3.6464 - accuracy: 0.5000
Epoch 6/100
32/32 [=====] - 0s 924us/step - loss: 3.4960 - accuracy: 0.5000
Epoch 7/100
32/32 [=====] - 0s 894us/step - loss: 3.3804 - accuracy: 0.5000
Epoch 8/100
32/32 [=====] - 0s 943us/step - loss: 3.2279 - accuracy: 0.5000
Epoch 9/100
32/32 [=====] - 0s 1ms/step - loss: 2.7024 - accuracy: 0.5000
Epoch 10/100
```

```
-- -- -- -- --  
32/32 [=====] - 0s 911us/step - loss: 2.4002 - accuracy: 0.5000  
Epoch 11/100  
32/32 [=====] - 0s 956us/step - loss: 2.1984 - accuracy: 0.5000  
Epoch 12/100  
32/32 [=====] - 0s 933us/step - loss: 1.3257 - accuracy: 0.5000  
Epoch 13/100  
32/32 [=====] - 0s 1ms/step - loss: 1.0542 - accuracy: 0.5000  
Epoch 14/100  
32/32 [=====] - 0s 913us/step - loss: 1.0261 - accuracy: 0.5000  
Epoch 15/100  
32/32 [=====] - 0s 934us/step - loss: 1.0069 - accuracy: 0.5000  
Epoch 16/100  
32/32 [=====] - 0s 981us/step - loss: 0.9914 - accuracy: 0.5000  
Epoch 17/100  
32/32 [=====] - 0s 968us/step - loss: 0.9776 - accuracy: 0.5000  
Epoch 18/100  
32/32 [=====] - 0s 1ms/step - loss: 0.9656 - accuracy: 0.5000  
Epoch 19/100  
32/32 [=====] - 0s 957us/step - loss: 0.9549 - accuracy: 0.5000  
Epoch 20/100  
32/32 [=====] - 0s 969us/step - loss: 0.9448 - accuracy: 0.5000  
Epoch 21/100  
32/32 [=====] - 0s 938us/step - loss: 0.9357 - accuracy: 0.5000  
Epoch 22/100  
32/32 [=====] - 0s 1ms/step - loss: 0.9269 - accuracy: 0.5000  
Epoch 23/100  
32/32 [=====] - 0s 963us/step - loss: 0.9190 - accuracy: 0.5000  
Epoch 24/100  
32/32 [=====] - 0s 1ms/step - loss: 0.9115 - accuracy: 0.5000  
Epoch 25/100  
32/32 [=====] - 0s 1ms/step - loss: 0.9046 - accuracy: 0.5000  
Epoch 26/100  
32/32 [=====] - 0s 1ms/step - loss: 0.8976 - accuracy: 0.5000  
Epoch 27/100  
32/32 [=====] - 0s 970us/step - loss: 0.8912 - accuracy: 0.4990  
Epoch 28/100  
32/32 [=====] - 0s 948us/step - loss: 0.8850 - accuracy: 0.4960  
Epoch 29/100  
32/32 [=====] - 0s 1ms/step - loss: 0.8790 - accuracy: 0.4950  
Epoch 30/100  
32/32 [=====] - 0s 1ms/step - loss: 0.8732 - accuracy: 0.4950  
Epoch 31/100  
32/32 [=====] - 0s 978us/step - loss: 0.8678 - accuracy: 0.4870  
Epoch 32/100  
32/32 [=====] - 0s 1ms/step - loss: 0.8623 - accuracy: 0.4790  
Epoch 33/100  
32/32 [=====] - 0s 1ms/step - loss: 0.8573 - accuracy: 0.4750  
Epoch 34/100  
32/32 [=====] - 0s 1ms/step - loss: 0.8525 - accuracy: 0.4730  
Epoch 35/100  
32/32 [=====] - 0s 1ms/step - loss: 0.8477 - accuracy: 0.4650  
Epoch 36/100  
32/32 [=====] - 0s 984us/step - loss: 0.8432 - accuracy: 0.4590  
Epoch 37/100  
32/32 [=====] - 0s 957us/step - loss: 0.8388 - accuracy: 0.4560  
Epoch 38/100  
32/32 [=====] - 0s 959us/step - loss: 0.8347 - accuracy: 0.4470  
Epoch 39/100  
32/32 [=====] - 0s 972us/step - loss: 0.8305 - accuracy: 0.4420  
Epoch 40/100  
32/32 [=====] - 0s 1ms/step - loss: 0.8266 - accuracy: 0.4380  
Epoch 41/100  
32/32 [=====] - 0s 1ms/step - loss: 0.8227 - accuracy: 0.4340  
Epoch 42/100  
32/32 [=====] - 0s 1ms/step - loss: 0.8189 - accuracy: 0.4290  
Epoch 43/100  
32/32 [=====] - 0s 1ms/step - loss: 0.8151 - accuracy: 0.4280  
Epoch 44/100  
32/32 [=====] - 0s 968us/step - loss: 0.8116 - accuracy: 0.4280  
Epoch 45/100  
32/32 [=====] - 0s 930us/step - loss: 0.8082 - accuracy: 0.4200  
Epoch 46/100
```

```
-->-- 100%  
32/32 [=====] - 0s 958us/step - loss: 0.8049 - accuracy: 0.4160  
Epoch 47/100  
32/32 [=====] - 0s 1ms/step - loss: 0.8018 - accuracy: 0.4150  
Epoch 48/100  
32/32 [=====] - 0s 938us/step - loss: 0.7986 - accuracy: 0.4130  
Epoch 49/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7956 - accuracy: 0.4140  
Epoch 50/100  
32/32 [=====] - 0s 975us/step - loss: 0.7926 - accuracy: 0.4170  
Epoch 51/100  
32/32 [=====] - 0s 977us/step - loss: 0.7897 - accuracy: 0.4200  
Epoch 52/100  
32/32 [=====] - 0s 973us/step - loss: 0.7870 - accuracy: 0.4210  
Epoch 53/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7843 - accuracy: 0.4280  
Epoch 54/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7817 - accuracy: 0.4350  
Epoch 55/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7791 - accuracy: 0.4480  
Epoch 56/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7766 - accuracy: 0.4510  
Epoch 57/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7742 - accuracy: 0.4530  
Epoch 58/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7718 - accuracy: 0.4530  
Epoch 59/100  
32/32 [=====] - 0s 948us/step - loss: 0.7696 - accuracy: 0.4550  
Epoch 60/100  
32/32 [=====] - 0s 927us/step - loss: 0.7674 - accuracy: 0.4570  
Epoch 61/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7654 - accuracy: 0.4560  
Epoch 62/100  
32/32 [=====] - 0s 968us/step - loss: 0.7634 - accuracy: 0.4600  
Epoch 63/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7613 - accuracy: 0.4610  
Epoch 64/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7595 - accuracy: 0.4640  
Epoch 65/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7577 - accuracy: 0.4590  
Epoch 66/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7558 - accuracy: 0.4630  
Epoch 67/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7541 - accuracy: 0.4620  
Epoch 68/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7524 - accuracy: 0.4640  
Epoch 69/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7507 - accuracy: 0.4660  
Epoch 70/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7491 - accuracy: 0.4660  
Epoch 71/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7475 - accuracy: 0.4680  
Epoch 72/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7460 - accuracy: 0.4710  
Epoch 73/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7445 - accuracy: 0.4720  
Epoch 74/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7430 - accuracy: 0.4740  
Epoch 75/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7416 - accuracy: 0.4730  
Epoch 76/100  
32/32 [=====] - 0s 962us/step - loss: 0.7403 - accuracy: 0.4750  
Epoch 77/100  
32/32 [=====] - 0s 994us/step - loss: 0.7390 - accuracy: 0.4730  
Epoch 78/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7377 - accuracy: 0.4720  
Epoch 79/100  
32/32 [=====] - 0s 954us/step - loss: 0.7365 - accuracy: 0.4730  
Epoch 80/100  
32/32 [=====] - 0s 927us/step - loss: 0.7353 - accuracy: 0.4760  
Epoch 81/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7341 - accuracy: 0.4740  
Epoch 82/100
```

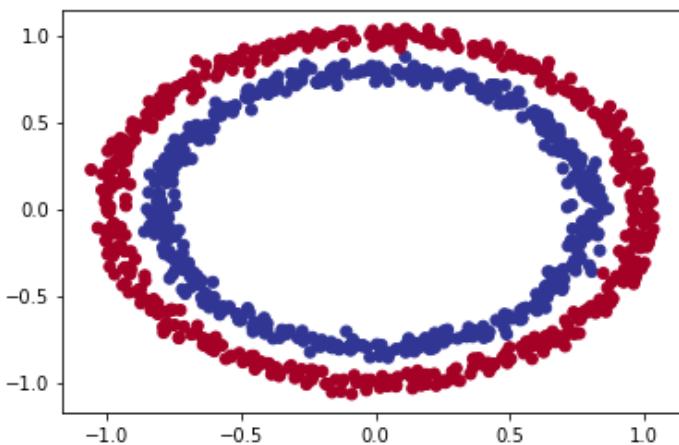
```
--> ... -->
32/32 [=====] - 0s 1ms/step - loss: 0.7330 - accuracy: 0.4750
Epoch 83/100
32/32 [=====] - 0s 1ms/step - loss: 0.7320 - accuracy: 0.4790
Epoch 84/100
32/32 [=====] - 0s 1ms/step - loss: 0.7309 - accuracy: 0.4790
Epoch 85/100
32/32 [=====] - 0s 1ms/step - loss: 0.7299 - accuracy: 0.4810
Epoch 86/100
32/32 [=====] - 0s 1ms/step - loss: 0.7290 - accuracy: 0.4820
Epoch 87/100
32/32 [=====] - 0s 1ms/step - loss: 0.7280 - accuracy: 0.4850
Epoch 88/100
32/32 [=====] - 0s 998us/step - loss: 0.7271 - accuracy: 0.4850
Epoch 89/100
32/32 [=====] - 0s 977us/step - loss: 0.7263 - accuracy: 0.4870
Epoch 90/100
32/32 [=====] - 0s 1ms/step - loss: 0.7254 - accuracy: 0.4880
Epoch 91/100
32/32 [=====] - 0s 1ms/step - loss: 0.7246 - accuracy: 0.4900
Epoch 92/100
32/32 [=====] - 0s 1ms/step - loss: 0.7238 - accuracy: 0.4890
Epoch 93/100
32/32 [=====] - 0s 1ms/step - loss: 0.7230 - accuracy: 0.4890
Epoch 94/100
32/32 [=====] - 0s 1ms/step - loss: 0.7223 - accuracy: 0.4860
Epoch 95/100
32/32 [=====] - 0s 1ms/step - loss: 0.7215 - accuracy: 0.4860
Epoch 96/100
32/32 [=====] - 0s 1ms/step - loss: 0.7208 - accuracy: 0.4860
Epoch 97/100
32/32 [=====] - 0s 1ms/step - loss: 0.7202 - accuracy: 0.4880
Epoch 98/100
32/32 [=====] - 0s 1ms/step - loss: 0.7195 - accuracy: 0.4870
Epoch 99/100
32/32 [=====] - 0s 1ms/step - loss: 0.7188 - accuracy: 0.4870
Epoch 100/100
32/32 [=====] - 0s 1ms/step - loss: 0.7181 - accuracy: 0.4850
```

**Okay, our model performs a little worse than guessing.**

**Let's remind ourselves what our data looks like.**

In [ ]:

```
# Check out our data
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.RdYlBu);
```

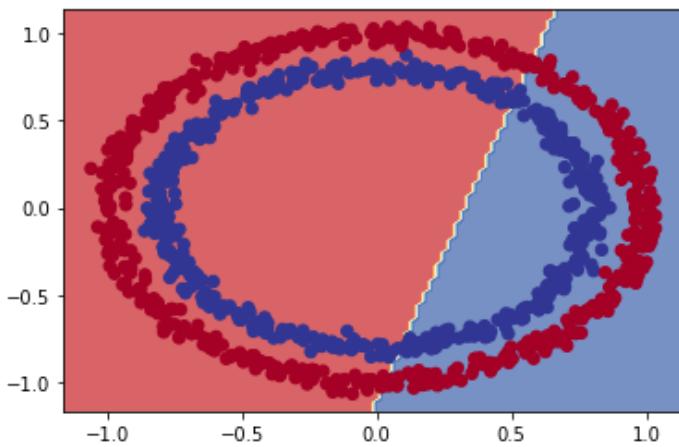


**And let's see how our model is making predictions on it.**

In [ ]:

```
# Check the decision boundary (blue is blue class, yellow is the crossover, red is red class)
plot_decision_boundary(model_4, X, y)
```

doing binary classification...



Well, it looks like we're getting a straight (linear) line prediction again.

But our data is non-linear (not a straight line)...

What we're going to have to do is add some non-linearity to our model.

To do so, we'll use the `activation` parameter in one of our layers.

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Create a model with a non-linear activation
model_5 = tf.keras.Sequential([
    tf.keras.layers.Dense(1, activation=tf.keras.activations.relu), # can also do activation='relu'
    tf.keras.layers.Dense(1) # output layer
])

# Compile the model
model_5.compile(loss=tf.keras.losses.binary_crossentropy,
                 optimizer=tf.keras.optimizers.Adam(),
                 metrics=["accuracy"])

# Fit the model
history = model_5.fit(X, y, epochs=100)
```

```
Epoch 1/100
32/32 [=====] - 0s 1ms/step - loss: 1.8377 - accuracy: 0.5000
Epoch 2/100
32/32 [=====] - 0s 920us/step - loss: 1.4449 - accuracy: 0.5000
Epoch 3/100
32/32 [=====] - 0s 975us/step - loss: 1.3410 - accuracy: 0.5000
Epoch 4/100
32/32 [=====] - 0s 1ms/step - loss: 1.2678 - accuracy: 0.4770
Epoch 5/100
32/32 [=====] - 0s 917us/step - loss: 1.2116 - accuracy: 0.4390
Epoch 6/100
32/32 [=====] - 0s 1ms/step - loss: 1.1664 - accuracy: 0.4180
Epoch 7/100
32/32 [=====] - 0s 990us/step - loss: 1.1294 - accuracy: 0.4250
Epoch 8/100
32/32 [=====] - 0s 1ms/step - loss: 1.0970 - accuracy: 0.4420
Epoch 9/100
32/32 [=====] - 0s 974us/step - loss: 1.0670 - accuracy: 0.4540
Epoch 10/100
32/32 [=====] - 0s 950us/step - loss: 1.0407 - accuracy: 0.4550
Epoch 11/100
32/32 [=====] - 0s 970us/step - loss: 1.0147 - accuracy: 0.4600
Epoch 12/100
32/32 [=====] - 0s 1ms/step - loss: 0.9872 - accuracy: 0.4630
Epoch 13/100
32/32 [=====] - 0s 1ms/step - loss: 0.9650 - accuracy: 0.4650
Epoch 14/100
32/32 [=====] - 0s 1ms/step - loss: 0.9430 - accuracy: 0.4670
Epoch 15/100
32/32 [=====] - 0s 1ms/step - loss: 0.9210 - accuracy: 0.4690
Epoch 16/100
32/32 [=====] - 0s 1ms/step - loss: 0.9000 - accuracy: 0.4710
Epoch 17/100
32/32 [=====] - 0s 1ms/step - loss: 0.8790 - accuracy: 0.4730
Epoch 18/100
32/32 [=====] - 0s 1ms/step - loss: 0.8580 - accuracy: 0.4750
Epoch 19/100
32/32 [=====] - 0s 1ms/step - loss: 0.8370 - accuracy: 0.4770
Epoch 20/100
32/32 [=====] - 0s 1ms/step - loss: 0.8160 - accuracy: 0.4790
Epoch 21/100
32/32 [=====] - 0s 1ms/step - loss: 0.7950 - accuracy: 0.4810
Epoch 22/100
32/32 [=====] - 0s 1ms/step - loss: 0.7740 - accuracy: 0.4830
Epoch 23/100
32/32 [=====] - 0s 1ms/step - loss: 0.7530 - accuracy: 0.4850
Epoch 24/100
32/32 [=====] - 0s 1ms/step - loss: 0.7320 - accuracy: 0.4870
Epoch 25/100
32/32 [=====] - 0s 1ms/step - loss: 0.7110 - accuracy: 0.4890
Epoch 26/100
32/32 [=====] - 0s 1ms/step - loss: 0.6900 - accuracy: 0.4910
Epoch 27/100
32/32 [=====] - 0s 1ms/step - loss: 0.6690 - accuracy: 0.4930
Epoch 28/100
32/32 [=====] - 0s 1ms/step - loss: 0.6480 - accuracy: 0.4950
Epoch 29/100
32/32 [=====] - 0s 1ms/step - loss: 0.6270 - accuracy: 0.4970
Epoch 30/100
32/32 [=====] - 0s 1ms/step - loss: 0.6060 - accuracy: 0.4990
Epoch 31/100
32/32 [=====] - 0s 1ms/step - loss: 0.5850 - accuracy: 0.5010
Epoch 32/100
32/32 [=====] - 0s 1ms/step - loss: 0.5640 - accuracy: 0.5030
Epoch 33/100
32/32 [=====] - 0s 1ms/step - loss: 0.5430 - accuracy: 0.5050
Epoch 34/100
32/32 [=====] - 0s 1ms/step - loss: 0.5220 - accuracy: 0.5070
Epoch 35/100
32/32 [=====] - 0s 1ms/step - loss: 0.5010 - accuracy: 0.5090
Epoch 36/100
32/32 [=====] - 0s 1ms/step - loss: 0.4800 - accuracy: 0.5110
Epoch 37/100
32/32 [=====] - 0s 1ms/step - loss: 0.4590 - accuracy: 0.5130
Epoch 38/100
32/32 [=====] - 0s 1ms/step - loss: 0.4380 - accuracy: 0.5150
Epoch 39/100
32/32 [=====] - 0s 1ms/step - loss: 0.4170 - accuracy: 0.5170
Epoch 40/100
32/32 [=====] - 0s 1ms/step - loss: 0.3960 - accuracy: 0.5190
Epoch 41/100
32/32 [=====] - 0s 1ms/step - loss: 0.3750 - accuracy: 0.5210
Epoch 42/100
32/32 [=====] - 0s 1ms/step - loss: 0.3540 - accuracy: 0.5230
Epoch 43/100
32/32 [=====] - 0s 1ms/step - loss: 0.3330 - accuracy: 0.5250
Epoch 44/100
32/32 [=====] - 0s 1ms/step - loss: 0.3120 - accuracy: 0.5270
Epoch 45/100
32/32 [=====] - 0s 1ms/step - loss: 0.2910 - accuracy: 0.5290
Epoch 46/100
32/32 [=====] - 0s 1ms/step - loss: 0.2700 - accuracy: 0.5310
Epoch 47/100
32/32 [=====] - 0s 1ms/step - loss: 0.2490 - accuracy: 0.5330
Epoch 48/100
32/32 [=====] - 0s 1ms/step - loss: 0.2280 - accuracy: 0.5350
Epoch 49/100
32/32 [=====] - 0s 1ms/step - loss: 0.2070 - accuracy: 0.5370
Epoch 50/100
32/32 [=====] - 0s 1ms/step - loss: 0.1860 - accuracy: 0.5390
Epoch 51/100
32/32 [=====] - 0s 1ms/step - loss: 0.1650 - accuracy: 0.5410
Epoch 52/100
32/32 [=====] - 0s 1ms/step - loss: 0.1440 - accuracy: 0.5430
Epoch 53/100
32/32 [=====] - 0s 1ms/step - loss: 0.1230 - accuracy: 0.5450
Epoch 54/100
32/32 [=====] - 0s 1ms/step - loss: 0.1020 - accuracy: 0.5470
Epoch 55/100
32/32 [=====] - 0s 1ms/step - loss: 0.0810 - accuracy: 0.5490
Epoch 56/100
32/32 [=====] - 0s 1ms/step - loss: 0.0600 - accuracy: 0.5510
Epoch 57/100
32/32 [=====] - 0s 1ms/step - loss: 0.0390 - accuracy: 0.5530
Epoch 58/100
32/32 [=====] - 0s 1ms/step - loss: 0.0180 - accuracy: 0.5550
Epoch 59/100
32/32 [=====] - 0s 1ms/step - loss: 0.0070 - accuracy: 0.5570
Epoch 60/100
32/32 [=====] - 0s 1ms/step - loss: 0.0000 - accuracy: 0.5590
```

32/32 [=====] - 0s 1ms/step - loss: 0.9579 - accuracy: 0.4620  
Epoch 14/100  
32/32 [=====] - 0s 1ms/step - loss: 0.9201 - accuracy: 0.4660  
Epoch 15/100  
32/32 [=====] - 0s 1ms/step - loss: 0.8514 - accuracy: 0.4660  
Epoch 16/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7888 - accuracy: 0.4720  
Epoch 17/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7580 - accuracy: 0.4740  
Epoch 18/100  
32/32 [=====] - 0s 937us/step - loss: 0.7392 - accuracy: 0.4760  
Epoch 19/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7273 - accuracy: 0.4850  
Epoch 20/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7180 - accuracy: 0.4870  
Epoch 21/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7121 - accuracy: 0.4880  
Epoch 22/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7070 - accuracy: 0.4880  
Epoch 23/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7036 - accuracy: 0.4870  
Epoch 24/100  
32/32 [=====] - 0s 1ms/step - loss: 0.7011 - accuracy: 0.4900  
Epoch 25/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6993 - accuracy: 0.4860  
Epoch 26/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6977 - accuracy: 0.4910  
Epoch 27/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6970 - accuracy: 0.4950  
Epoch 28/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6959 - accuracy: 0.4970  
Epoch 29/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6955 - accuracy: 0.4950  
Epoch 30/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6949 - accuracy: 0.4970  
Epoch 31/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6946 - accuracy: 0.4990  
Epoch 32/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6943 - accuracy: 0.4910  
Epoch 33/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6940 - accuracy: 0.5000  
Epoch 34/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6939 - accuracy: 0.4910  
Epoch 35/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6937 - accuracy: 0.4940  
Epoch 36/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6935 - accuracy: 0.4930  
Epoch 37/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6934 - accuracy: 0.4950  
Epoch 38/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6935 - accuracy: 0.4900  
Epoch 39/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6934 - accuracy: 0.4840  
Epoch 40/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6935 - accuracy: 0.5000  
Epoch 41/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6936 - accuracy: 0.5060  
Epoch 42/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6935 - accuracy: 0.4800  
Epoch 43/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6933 - accuracy: 0.5050  
Epoch 44/100  
32/32 [=====] - 0s 972us/step - loss: 0.6936 - accuracy: 0.4810  
Epoch 45/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6933 - accuracy: 0.4550  
Epoch 46/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6932 - accuracy: 0.4920  
Epoch 47/100  
32/32 [=====] - 0s 997us/step - loss: 0.6935 - accuracy: 0.4870  
Epoch 48/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6934 - accuracy: 0.5100  
Epoch 49/100

32/32 [=====] - 0s 1ms/step - loss: 0.6934 - accuracy: 0.4700  
Epoch 50/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6934 - accuracy: 0.5000  
Epoch 51/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6933 - accuracy: 0.4760  
Epoch 52/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6934 - accuracy: 0.5010  
Epoch 53/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6935 - accuracy: 0.4880  
Epoch 54/100  
32/32 [=====] - 0s 993us/step - loss: 0.6933 - accuracy: 0.5530  
Epoch 55/100  
32/32 [=====] - 0s 972us/step - loss: 0.6935 - accuracy: 0.5060  
Epoch 56/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6935 - accuracy: 0.5200  
Epoch 57/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6933 - accuracy: 0.4910  
Epoch 58/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6934 - accuracy: 0.4990  
Epoch 59/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6938 - accuracy: 0.5000  
Epoch 60/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6934 - accuracy: 0.5000  
Epoch 61/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6935 - accuracy: 0.4900  
Epoch 62/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6934 - accuracy: 0.4820  
Epoch 63/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6933 - accuracy: 0.4700  
Epoch 64/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6934 - accuracy: 0.4820  
Epoch 65/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6934 - accuracy: 0.4620  
Epoch 66/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6934 - accuracy: 0.4760  
Epoch 67/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6934 - accuracy: 0.4830  
Epoch 68/100  
32/32 [=====] - 0s 993us/step - loss: 0.6933 - accuracy: 0.4680  
Epoch 69/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6933 - accuracy: 0.5010  
Epoch 70/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6934 - accuracy: 0.4970  
Epoch 71/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6933 - accuracy: 0.4680  
Epoch 72/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6934 - accuracy: 0.5070  
Epoch 73/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6934 - accuracy: 0.5320  
Epoch 74/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6933 - accuracy: 0.5290  
Epoch 75/100  
32/32 [=====] - 0s 968us/step - loss: 0.6935 - accuracy: 0.5000  
Epoch 76/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6935 - accuracy: 0.4730  
Epoch 77/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6935 - accuracy: 0.4870  
Epoch 78/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6934 - accuracy: 0.4830  
Epoch 79/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6935 - accuracy: 0.4640  
Epoch 80/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6933 - accuracy: 0.5040  
Epoch 81/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6936 - accuracy: 0.5000  
Epoch 82/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6935 - accuracy: 0.5020  
Epoch 83/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6936 - accuracy: 0.4840  
Epoch 84/100  
32/32 [=====] - 0s 1ms/step - loss: 0.6932 - accuracy: 0.5070  
Epoch 85/100

```

32/32 [=====] - 0s 1ms/step - loss: 0.6933 - accuracy: 0.5000
Epoch 86/100
32/32 [=====] - 0s 1ms/step - loss: 0.6935 - accuracy: 0.5000
Epoch 87/100
32/32 [=====] - 0s 1ms/step - loss: 0.6933 - accuracy: 0.5000
Epoch 88/100
32/32 [=====] - 0s 1ms/step - loss: 0.6933 - accuracy: 0.4680
Epoch 89/100
32/32 [=====] - 0s 1ms/step - loss: 0.6934 - accuracy: 0.4590
Epoch 90/100
32/32 [=====] - 0s 1ms/step - loss: 0.6937 - accuracy: 0.4980
Epoch 91/100
32/32 [=====] - 0s 1ms/step - loss: 0.6933 - accuracy: 0.4910
Epoch 92/100
32/32 [=====] - 0s 1ms/step - loss: 0.6935 - accuracy: 0.4850
Epoch 93/100
32/32 [=====] - 0s 967us/step - loss: 0.6938 - accuracy: 0.4970
Epoch 94/100
32/32 [=====] - 0s 1ms/step - loss: 0.6934 - accuracy: 0.4710
Epoch 95/100
32/32 [=====] - 0s 1ms/step - loss: 0.6934 - accuracy: 0.4720
Epoch 96/100
32/32 [=====] - 0s 1ms/step - loss: 0.6934 - accuracy: 0.4880
Epoch 97/100
32/32 [=====] - 0s 1ms/step - loss: 0.6934 - accuracy: 0.4510
Epoch 98/100
32/32 [=====] - 0s 1ms/step - loss: 0.6934 - accuracy: 0.4640
Epoch 99/100
32/32 [=====] - 0s 1ms/step - loss: 0.6935 - accuracy: 0.4940
Epoch 100/100
32/32 [=====] - 0s 1ms/step - loss: 0.6936 - accuracy: 0.5180

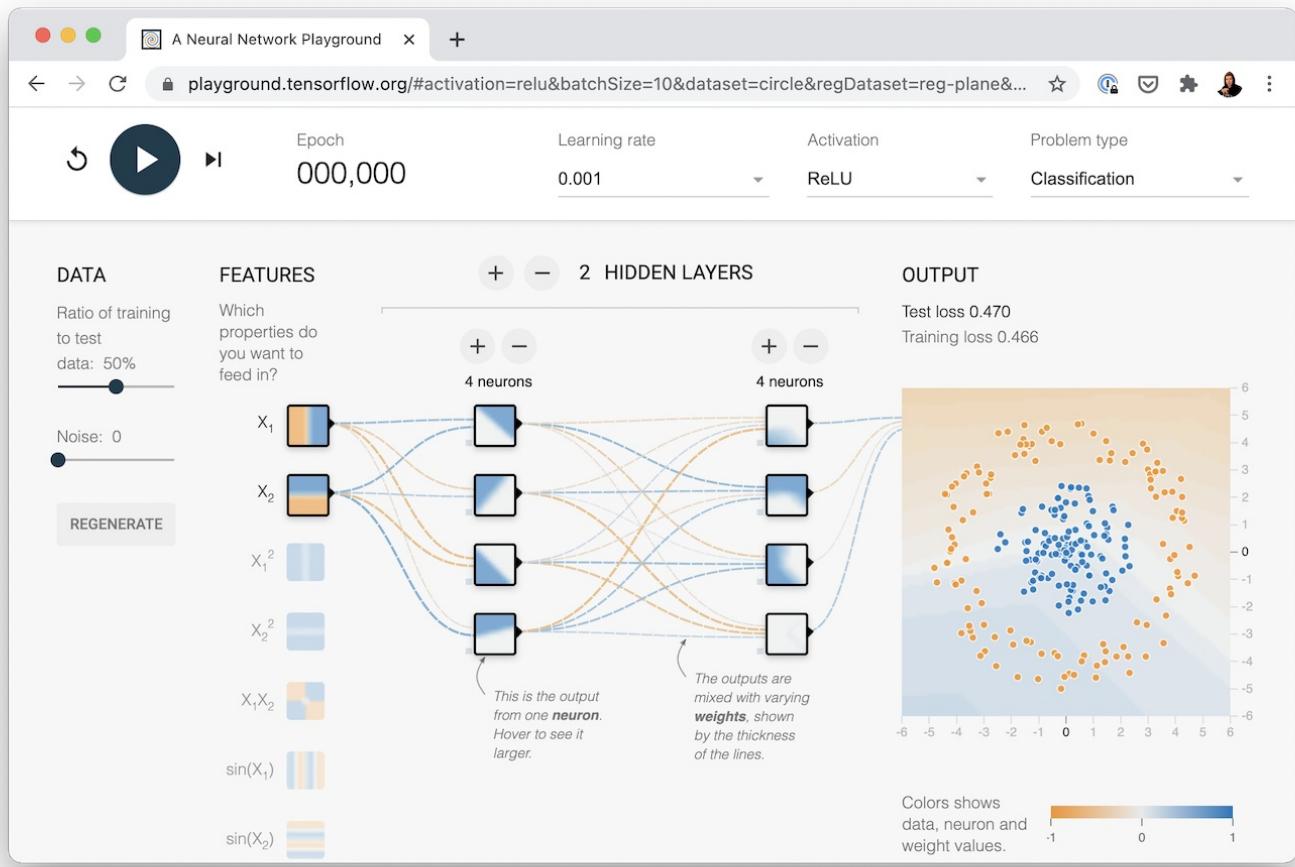
```

**Hmm... still not learning...**

**What we if increased the number of neurons and layers?**

**Say, 2 hidden layers, with ReLU, pronounced "rel-u", (short for rectified linear unit), activation on the first one, and 4 neurons each?**

To see this network in action, check out the [TensorFlow Playground demo](#).



The neural network we're going to recreate with TensorFlow code. See it live at [TensorFlow Playground](#).

Let's try.

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Create a model
model_6 = tf.keras.Sequential([
    tf.keras.layers.Dense(4, activation=tf.keras.activations.relu), # hidden layer 1, 4 neurons, ReLU activation
    tf.keras.layers.Dense(4, activation=tf.keras.activations.relu), # hidden layer 2, 4 neurons, ReLU activation
    tf.keras.layers.Dense(1) # output layer
])

# Compile the model
model_6.compile(loss=tf.keras.losses.binary_crossentropy,
                  optimizer=tf.keras.optimizers.Adam(lr=0.001), # Adam's default learning rate is 0.001
                  metrics=['accuracy'])

# Fit the model
history = model_6.fit(X, y, epochs=100)
```

```
Epoch 1/100
32/32 [=====] - 0s 1ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 2/100
32/32 [=====] - 0s 969us/step - loss: 7.7125 - accuracy: 0.5000
Epoch 3/100
32/32 [=====] - 0s 1ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 4/100
32/32 [=====] - 0s 975us/step - loss: 7.7125 - accuracy: 0.5000
Epoch 5/100
32/32 [=====] - 0s 934us/step - loss: 7.7125 - accuracy: 0.5000
Epoch 6/100
32/32 [=====] - 0s 967us/step - loss: 7.7125 - accuracy: 0.5000
Epoch 7/100
32/32 [=====] - 0s 1ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 8/100
32/32 [=====] - 0s 1ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 9/100
32/32 [=====] - 0s 1ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 10/100
32/32 [=====] - 0s 1ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 11/100
32/32 [=====] - 0s 1ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 12/100
32/32 [=====] - 0s 997us/step - loss: 7.7125 - accuracy: 0.5000
Epoch 13/100
32/32 [=====] - 0s 1ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 14/100
32/32 [=====] - 0s 1ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 15/100
32/32 [=====] - 0s 1ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 16/100
32/32 [=====] - 0s 1ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 17/100
32/32 [=====] - 0s 952us/step - loss: 7.7125 - accuracy: 0.5000
Epoch 18/100
32/32 [=====] - 0s 1ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 19/100
32/32 [=====] - 0s 1ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 20/100
32/32 [=====] - 0s 1ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 21/100
32/32 [=====] - 0s 1ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 22/100
```





```
32/32 [=====] - 0s 1ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 95/100
32/32 [=====] - 0s 1ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 96/100
32/32 [=====] - 0s 995us/step - loss: 7.7125 - accuracy: 0.5000
Epoch 97/100
32/32 [=====] - 0s 1ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 98/100
32/32 [=====] - 0s 985us/step - loss: 7.7125 - accuracy: 0.5000
Epoch 99/100
32/32 [=====] - 0s 1ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 100/100
32/32 [=====] - 0s 1ms/step - loss: 7.7125 - accuracy: 0.5000
```

In [ ]:

```
# Evaluate the model
model_6.evaluate(X, y)
```

```
32/32 [=====] - 0s 876us/step - loss: 7.7125 - accuracy: 0.5000
```

Out[ ]:

```
[7.712474346160889, 0.5]
```

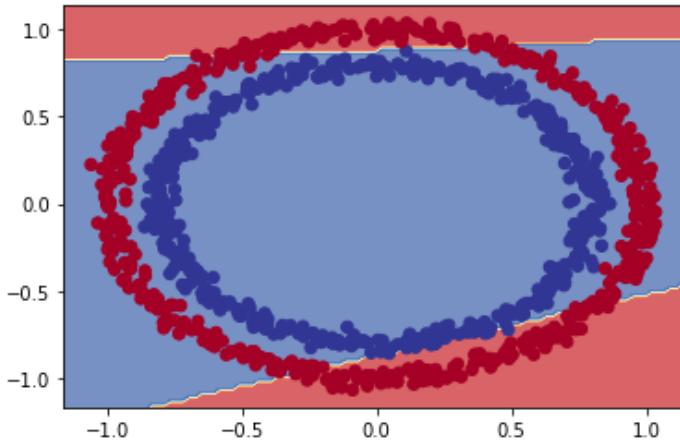
We're still hitting 50% accuracy, our model is still practically as good as guessing.

How do the predictions look?

In [ ]:

```
# Check out the predictions using 2 hidden layers
plot_decision_boundary(model_6, X, y)
```

doing binary classification...



What gives?

It seems like our model is the same as the one in the [TensorFlow Playground](#) but model it's still drawing straight lines...

Ideally, the yellow lines go on the inside of the red circle and the blue circle.

Okay, okay, let's model this circle once and for all.

One more model (I promise... actually, I'm going to have to break that promise... we'll be building plenty more models).

This time we'll change the activation function on our output layer too. Remember the architecture of a classification model? For binary classification, the output layer activation is usually the [Sigmoid activation function](#).

In [ ]:

```
# Set random seed
```

```

tf.random.set_seed(42)

# Create a model
model_7 = tf.keras.Sequential([
    tf.keras.layers.Dense(4, activation=tf.keras.activations.relu), # hidden layer 1, ReLU
    activation
    tf.keras.layers.Dense(4, activation=tf.keras.activations.relu), # hidden layer 2, ReLU
    activation
    tf.keras.layers.Dense(1, activation=tf.keras.activations.sigmoid) # output layer, sigmoid
    activation
])

# Compile the model
model_7.compile(loss=tf.keras.losses.binary_crossentropy,
                 optimizer=tf.keras.optimizers.Adam(),
                 metrics=['accuracy'])

# Fit the model
history = model_7.fit(X, y, epochs=100, verbose=0)

```

In [ ]:

```

# Evaluate our model
model_7.evaluate(X, y)

```

32/32 [=====] - 0s 870us/step - loss: 0.2948 - accuracy: 0.9910

Out[ ]:

[0.29480037093162537, 0.9909999966621399]

**Woah! It looks like our model is getting some incredible results, let's check them out.**

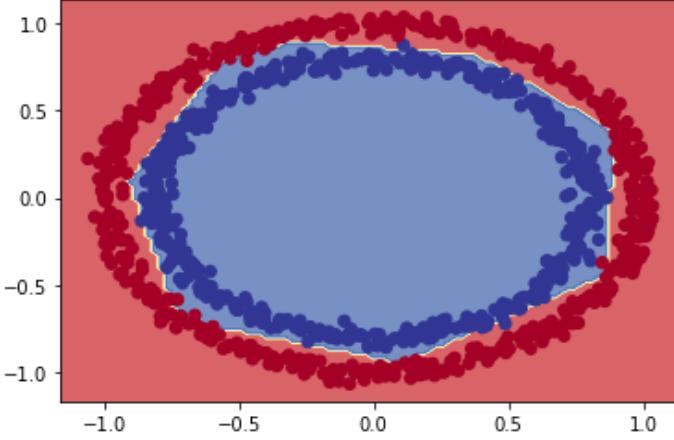
In [ ]:

```

# View the predictions of the model with relu and sigmoid activations
plot_decision_boundary(model_7, X, y)

```

doing binary classification...



Nice! It looks like our model is almost perfectly (apart from a few examples) separating the two circles.

❑ **Question:** What's wrong with the predictions we've made? Are we really evaluating our model correctly here? Hint: what data did the model learn on and what did we predict on?

Before we answer that, it's important to recognize what we've just covered.

❑ **Note:** The combination of linear (straight lines) and non-linear (non-straight lines) functions is one of the key fundamentals of neural networks.

Think of it like this:

If I gave you an unlimited amount of straight lines and non-straight lines, what kind of patterns could you draw?

That's essentially what neural networks do to find patterns in data.

Now you might be thinking, "but I haven't seen a linear function or a non-linear function before..."

Oh but you have.

We've been using them the whole time.

They're what power the layers in the models we just built.

To get some intuition about the activation functions we've just used, let's create them and then try them on some toy data.

In [ ]:

```
# Create a toy tensor (similar to the data we pass into our model)
A = tf.cast(tf.range(-10, 10), tf.float32)
A
```

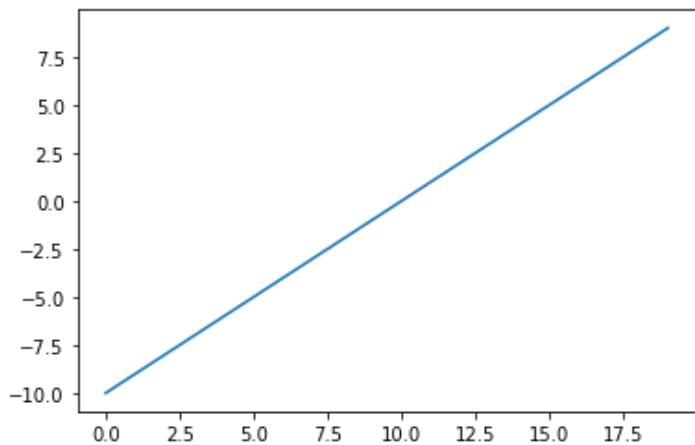
Out [ ]:

```
<tf.Tensor: shape=(20,), dtype=float32, numpy=
array([-10., -9., -8., -7., -6., -5., -4., -3., -2., -1.,  0.,
       1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.],
      dtype=float32)>
```

How does this look?

In [ ]:

```
# Visualize our toy tensor
plt.plot(A);
```



A straight (linear) line!

Nice, now let's recreate the [sigmoid function](#) and see what it does to our data. You can also find a pre-built sigmoid function at [tf.keras.activations.sigmoid](#).

In [ ]:

```
# Sigmoid - https://www.tensorflow.org/api_docs/python/tf/keras/activations/sigmoid
def sigmoid(x):
    return 1 / (1 + tf.exp(-x))

# Use the sigmoid function on our tensor
sigmoid(A)
```

Out [ ]:

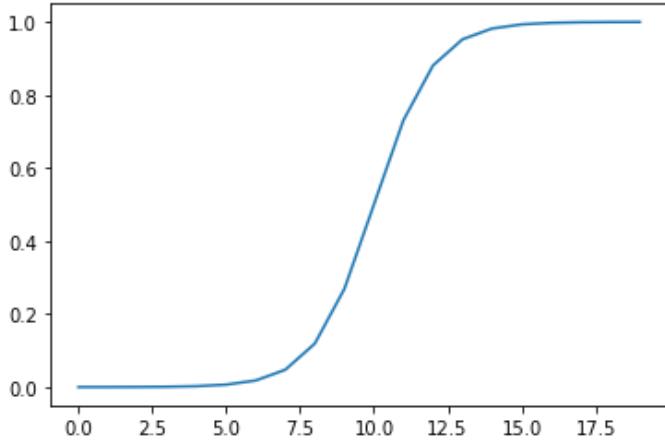
```
<tf.Tensor: shape=(20,), dtype=float32, numpy=
array([4.5397872e-05, 1.2339458e-04, 3.3535014e-04, 9.1105117e-04,
       2.4726233e-03, 6.6928510e-03, 1.7986210e-02, 4.7425874e-02,
       1.1920292e-01, 2.6894143e-01, 5.0000000e-01, 7.3105860e-01,
       8.8079703e-01, 9.5257413e-01, 9.8201376e-01, 9.9330717e-01,
```

```
9.9752742e-01, 9.9908900e-01, 9.9966466e-01, 9.9987662e-01],  
dtype=float32)>
```

## And how does it look?

In [ ]:

```
# Plot sigmoid modified tensor  
plt.plot(sigmoid(A));
```



A non-straight (non-linear) line!

Okay, how about the [ReLU function](#) (ReLU turns all negatives to 0 and positive numbers stay the same)?

In [ ]:

```
# ReLU - https://www.tensorflow.org/api\_docs/python/tf/keras/activations/relu  
def relu(x):  
    return tf.maximum(0, x)  
  
# Pass toy tensor through ReLU function  
relu(A)
```

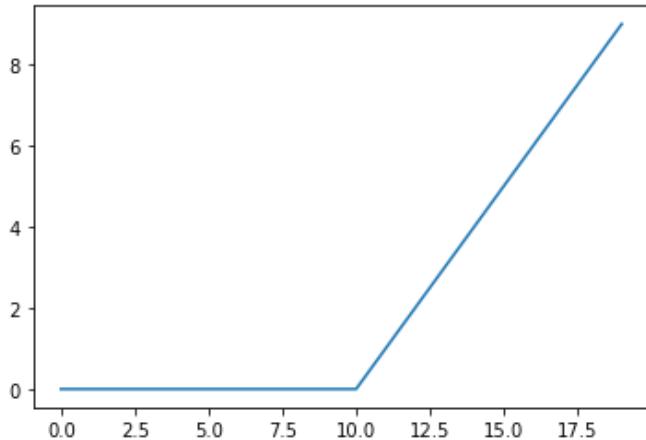
Out [ ]:

```
<tf.Tensor: shape=(20,), dtype=float32, numpy=  
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 2., 3., 4., 5., 6.,  
    7., 8., 9.], dtype=float32)>
```

How does the ReLU-modified tensor look?

In [ ]:

```
# Plot ReLU-modified tensor  
plt.plot(relu(A));
```



Another non-straight line!

## Well, how about TensorFlow's [linear activation function?](#)

In [ ]:

```
# Linear - https://www.tensorflow.org/api_docs/python/tf/keras/activations/linear (returns input non-modified...)
tf.keras.activations.linear(A)
```

Out[ ]:

```
<tf.Tensor: shape=(20,), dtype=float32, numpy=
array([-10., -9., -8., -7., -6., -5., -4., -3., -2., -1., 0.,
       1., 2., 3., 4., 5., 6., 7., 8., 9.],  
      dtype=float32)>
```

Hmm, it looks like our inputs are unmodified...

In [ ]:

```
# Does the linear activation change anything?
A == tf.keras.activations.linear(A)
```

Out[ ]:

```
<tf.Tensor: shape=(20,), dtype=bool, numpy=
array([ True,  True,  True,  True,  True,  True,  True,  True,
       True,  True,  True,  True,  True,  True,  True,  True,  
       True,  True])>
```

Okay, so it makes sense now the model doesn't really learn anything when using only linear activation functions, because the linear activation function doesn't change our input data in anyway.

Where as, with our non-linear functions, our data gets manipulated. A neural network uses these kind of transformations at a large scale to figure draw patterns between its inputs and outputs.

Now rather than dive into the guts of neural networks, we're going to keep coding applying what we've learned to different problems but if you want a more in-depth look at what's going on behind the scenes, check out the Extra Curriculum section below.

**Resource:** For more on activation functions, check out the [machine learning cheatsheet page](#) on them.

## Evaluating and improving our classification model

If you answered the question above, you might've picked up what we've been doing wrong.

We've been evaluating our model on the same data it was trained on.

A better approach would be to split our data into training, validation (optional) and test sets.

Once we've done that, we'll train our model on the training set (let it find patterns in the data) and then see how well it learned the patterns by using it to predict values on the test set.

Let's do it.

In [ ]:

```
# How many examples are in the whole dataset?
len(X)
```

Out[ ]:

1000

In [ ]:

```
# Split data into train and test sets
```

```
X_train, y_train = X[:800], y[:800] # 80% of the data for the training set
X_test, y_test = X[800:], y[800:] # 20% of the data for the test set

# Check the shapes of the data
X_train.shape, X_test.shape # 800 examples in the training set, 200 examples in the test set
```

Out [ ]:

```
((800, 2), (200, 2))
```

**Great, now we've got training and test sets, let's model the training data and evaluate what our model has learned on the test set.**

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Create the model (same as model_7)
model_8 = tf.keras.Sequential([
    tf.keras.layers.Dense(4, activation="relu"), # hidden layer 1, using "relu" for activation (same as tf.keras.activations.relu)
    tf.keras.layers.Dense(4, activation="relu"),
    tf.keras.layers.Dense(1, activation="sigmoid") # output layer, using 'sigmoid' for the output
])

# Compile the model
model_8.compile(loss=tf.keras.losses.binary_crossentropy,
                 optimizer=tf.keras.optimizers.Adam(lr=0.01), # increase learning rate from 0.001 to 0.01 for faster learning
                 metrics=['accuracy'])

# Fit the model
history = model_8.fit(X_train, y_train, epochs=25)
```

```
Epoch 1/25
25/25 [=====] - 0s 1ms/step - loss: 0.6847 - accuracy: 0.5425
Epoch 2/25
25/25 [=====] - 0s 1ms/step - loss: 0.6777 - accuracy: 0.5525
Epoch 3/25
25/25 [=====] - 0s 961us/step - loss: 0.6736 - accuracy: 0.5512
Epoch 4/25
25/25 [=====] - 0s 975us/step - loss: 0.6681 - accuracy: 0.5775
Epoch 5/25
25/25 [=====] - 0s 964us/step - loss: 0.6633 - accuracy: 0.5850
Epoch 6/25
25/25 [=====] - 0s 952us/step - loss: 0.6546 - accuracy: 0.5838
Epoch 7/25
25/25 [=====] - 0s 1ms/step - loss: 0.6413 - accuracy: 0.6750
Epoch 8/25
25/25 [=====] - 0s 961us/step - loss: 0.6264 - accuracy: 0.7013
Epoch 9/25
25/25 [=====] - 0s 969us/step - loss: 0.6038 - accuracy: 0.7487
Epoch 10/25
25/25 [=====] - 0s 978us/step - loss: 0.5714 - accuracy: 0.7738
Epoch 11/25
25/25 [=====] - 0s 1ms/step - loss: 0.5404 - accuracy: 0.7650
Epoch 12/25
25/25 [=====] - 0s 1ms/step - loss: 0.5015 - accuracy: 0.7837
Epoch 13/25
25/25 [=====] - 0s 1ms/step - loss: 0.4683 - accuracy: 0.7975
Epoch 14/25
25/25 [=====] - 0s 997us/step - loss: 0.4113 - accuracy: 0.8450
Epoch 15/25
25/25 [=====] - 0s 967us/step - loss: 0.3625 - accuracy: 0.9125
Epoch 16/25
25/25 [=====] - 0s 1ms/step - loss: 0.3209 - accuracy: 0.9312
Epoch 17/25
25/25 [=====] - 0s 1ms/step - loss: 0.2847 - accuracy: 0.9488
Epoch 18/25
```

```
Epoch 18/25
25/25 [=====] - 0s 1ms/step - loss: 0.2597 - accuracy: 0.9525
Epoch 19/25
25/25 [=====] - 0s 1ms/step - loss: 0.2375 - accuracy: 0.9563
Epoch 20/25
25/25 [=====] - 0s 988us/step - loss: 0.2135 - accuracy: 0.9663
Epoch 21/25
25/25 [=====] - 0s 1ms/step - loss: 0.1938 - accuracy: 0.9775
Epoch 22/25
25/25 [=====] - 0s 1ms/step - loss: 0.1752 - accuracy: 0.9737
Epoch 23/25
25/25 [=====] - 0s 1ms/step - loss: 0.1619 - accuracy: 0.9787
Epoch 24/25
25/25 [=====] - 0s 1ms/step - loss: 0.1550 - accuracy: 0.9775
Epoch 25/25
25/25 [=====] - 0s 1ms/step - loss: 0.1490 - accuracy: 0.9762
```

In [ ]:

```
# Evaluate our model on the test set
loss, accuracy = model_8.evaluate(X_test, y_test)
print(f"Model loss on the test set: {loss}")
print(f"Model accuracy on the test set: {100*accuracy:.2f}%")
```

```
7/7 [=====] - 0s 1ms/step - loss: 0.1247 - accuracy: 1.0000
Model loss on the test set: 0.12468849867582321
Model accuracy on the test set: 100.00%
```

100% accuracy? Nice!

Now, when we started to create `model_8` we said it was going to be the same as `model_7` but you might've found that to be a little lie.

That's because we changed a few things:

- The `activation` parameter - We used strings (`"relu"` & `"sigmoid"`) instead of using library paths (`tf.keras.activations.relu`), in TensorFlow, they both offer the same functionality.
- The `learning_rate` (also `lr`) parameter - We increased the `learning rate` parameter in the [Adam optimizer](#) to `0.01` instead of `0.001` (an increase of 10x).
  - You can think of the learning rate as how quickly a model learns. The higher the learning rate, the faster the model's capacity to learn, however, there's such a thing as a *too high* learning rate, where a model tries to learn too fast and doesn't learn anything. We'll see a trick to find the ideal learning rate soon.
- The number of epochs - We lowered the number of epochs (using the `epochs` parameter) from 100 to 25 but our model still got an incredible result on both the training and test sets.
  - One of the reasons our model performed well in even less epochs (remember a single epoch is the model trying to learn patterns in the data by looking at it once, so 25 epochs means the model gets 25 chances) than before is because we increased the learning rate.

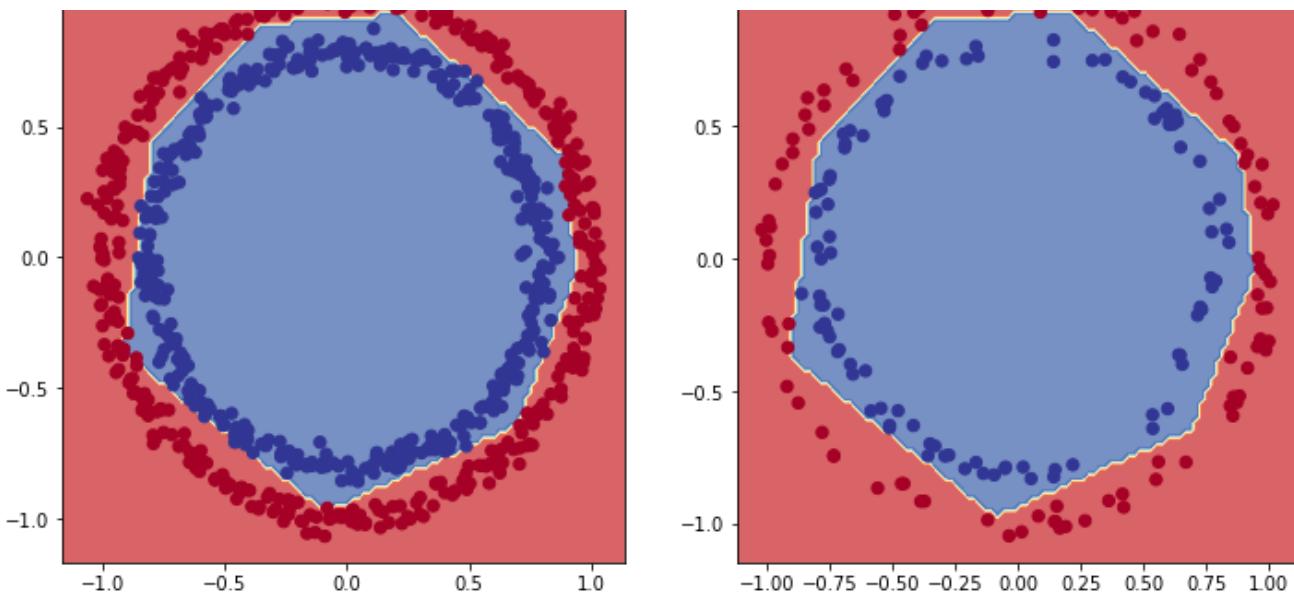
We know our model is performing well based on the evaluation metrics but let's see how it performs visually.

In [ ]:

```
# Plot the decision boundaries for the training and test sets
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.title("Train")
plot_decision_boundary(model_8, X=X_train, y=y_train)
plt.subplot(1, 2, 2)
plt.title("Test")
plot_decision_boundary(model_8, X=X_test, y=y_test)
plt.show()
```

doing binary classification...  
doing binary classification...





Check that out! How cool. With a few tweaks, our model is now predicting the blue and red circles almost perfectly.

## Plot the loss curves

Looking at the plots above, we can see the outputs of our model are very good.

But how did our model go whilst it was learning?

As in, how did the performance change everytime the model had a chance to look at the data (once every epoch)?

To figure this out, we can check the **loss curves** (also referred to as the **learning curves**).

You might've seen we've been using the variable `history` when calling the `fit()` function on a model ([fit\(\) returns a History object](#)).

This is where we'll get the information for how our model is performing as it learns.

Let's see how we might use it.

In [ ]:

```
# You can access the information in the history variable using the .history attribute
pd.DataFrame(history.history)
```

Out[ ]:

	loss	accuracy
0	0.684651	0.54250
1	0.677721	0.55250
2	0.673595	0.55125
3	0.668149	0.57750
4	0.663269	0.58500
5	0.654567	0.58375
6	0.641258	0.67500
7	0.626428	0.70125
8	0.603831	0.74875
9	0.571404	0.77375
10	0.540443	0.76500
11	0.501504	0.78375

	loss	accuracy
12	0.468332	0.79750
13	0.411302	0.84500
14	0.362506	0.91250
15	0.320904	0.93125
16	0.284708	0.94875
17	0.259720	0.95250
18	0.237469	0.95625
19	0.213520	0.96625
20	0.193820	0.97750
21	0.175244	0.97375
22	0.161893	0.97875
23	0.154989	0.97750
24	0.148973	0.97625

Inspecting the outputs, we can see the loss values going down and the accuracy going up.

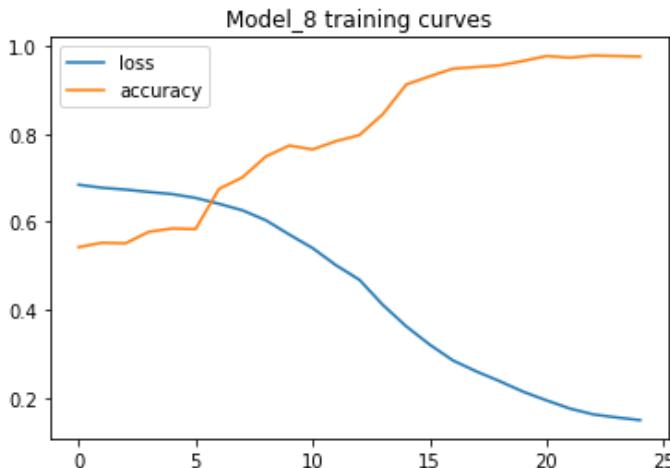
How's it look (visualize, visualize, visualize)?

In [ ]:

```
# Plot the loss curves
pd.DataFrame(history.history).plot()
plt.title("Model_8 training curves")
```

Out[ ]:

Text(0.5, 1.0, 'Model\_8 training curves')



Beautiful. This is the ideal plot we'd be looking for when dealing with a classification problem, loss going down, accuracy going up.

💡 Note: For many problems, the loss function going down means the model is improving (the predictions it's making are getting closer to the ground truth labels).

## Finding the best learning rate

Aside from the architecture itself (the layers, number of neurons, activations, etc), the most important hyperparameter you can tune for your neural network models is the **learning rate**.

In `model_8` you saw we lowered the Adam optimizer's learning rate from the default of `0.001 (default)` to `0.01`.

And you might be wondering why we did this.

Put it this way, it was a lucky guess.

I just decided to try a lower learning rate and see how the model went.

Now you might be thinking, "Seriously? You can do that?"

And the answer is yes. You can change any of the hyperparameters of your neural networks.

With practice, you'll start to see what kind of hyperparameters work and what don't.

That's an important thing to understand about machine learning and deep learning in general. It's very experimental. You build a model and evaluate it, build a model and evaluate it.

That being said, I want to introduce you a trick which will help you find the optimal learning rate (at least to begin training with) for your models going forward.

To do so, we're going to use the following:

- A [learning rate callback](#).
  - You can think of a callback as an extra piece of functionality you can add to your model while its training.
- Another model (we could use the same ones as above, we're practicing building models here).
- A modified loss curves plot.

We'll go through each with code, then explain what's going on.

**Note:** The default hyperparameters of many neural network building blocks in TensorFlow are setup in a way which usually work right out of the box (e.g. the [Adam optimizer's](#) default settings can usually get good results on many datasets). So it's a good idea to try the defaults first, then adjust as needed.

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Create a model (same as model_8)
model_9 = tf.keras.Sequential([
    tf.keras.layers.Dense(4, activation="relu"),
    tf.keras.layers.Dense(4, activation="relu"),
    tf.keras.layers.Dense(1, activation="sigmoid")
])

# Compile the model
model_9.compile(loss="binary_crossentropy", # we can use strings here too
                 optimizer="Adam", # same as tf.keras.optimizers.Adam() with default settings
                 metrics=["accuracy"])

# Create a learning rate scheduler callback
lr_scheduler = tf.keras.callbacks.LearningRateScheduler(lambda epoch: 1e-4 * 10**(epoch/20)) # traverse a set of learning rate values starting from 1e-4, increasing by 10**(epoch/20) every epoch

# Fit the model (passing the lr_scheduler callback)
history = model_9.fit(X_train,
                       y_train,
                       epochs=100,
                       callbacks=[lr_scheduler])
```

```
Epoch 1/100
25/25 [=====] - 0s 1ms/step - loss: 0.6945 - accuracy: 0.4988
Epoch 2/100
25/25 [=====] - 0s 1ms/step - loss: 0.6938 - accuracy: 0.4975
Epoch 3/100
25/25 [=====] - 0s 1ms/step - loss: 0.6930 - accuracy: 0.4963
Epoch 4/100
```

25/25 [=====] - 0s 1ms/step - loss: 0.6922 - accuracy: 0.4975  
Epoch 5/100  
25/25 [=====] - 0s 944us/step - loss: 0.6914 - accuracy: 0.5063  
Epoch 6/100  
25/25 [=====] - 0s 938us/step - loss: 0.6906 - accuracy: 0.5013  
Epoch 7/100  
25/25 [=====] - 0s 996us/step - loss: 0.6898 - accuracy: 0.4950  
Epoch 8/100  
25/25 [=====] - 0s 1ms/step - loss: 0.6889 - accuracy: 0.5038  
Epoch 9/100  
25/25 [=====] - 0s 1ms/step - loss: 0.6880 - accuracy: 0.5013  
Epoch 10/100  
25/25 [=====] - 0s 962us/step - loss: 0.6871 - accuracy: 0.5050  
Epoch 11/100  
25/25 [=====] - 0s 909us/step - loss: 0.6863 - accuracy: 0.5200  
Epoch 12/100  
25/25 [=====] - 0s 968us/step - loss: 0.6856 - accuracy: 0.5163  
Epoch 13/100  
25/25 [=====] - 0s 977us/step - loss: 0.6847 - accuracy: 0.5175  
Epoch 14/100  
25/25 [=====] - 0s 1ms/step - loss: 0.6842 - accuracy: 0.5200  
Epoch 15/100  
25/25 [=====] - 0s 1ms/step - loss: 0.6835 - accuracy: 0.5213  
Epoch 16/100  
25/25 [=====] - 0s 927us/step - loss: 0.6829 - accuracy: 0.5213  
Epoch 17/100  
25/25 [=====] - 0s 965us/step - loss: 0.6826 - accuracy: 0.5225  
Epoch 18/100  
25/25 [=====] - 0s 936us/step - loss: 0.6819 - accuracy: 0.5300  
Epoch 19/100  
25/25 [=====] - 0s 917us/step - loss: 0.6816 - accuracy: 0.5312  
Epoch 20/100  
25/25 [=====] - 0s 924us/step - loss: 0.6811 - accuracy: 0.5387  
Epoch 21/100  
25/25 [=====] - 0s 916us/step - loss: 0.6806 - accuracy: 0.5400  
Epoch 22/100  
25/25 [=====] - 0s 931us/step - loss: 0.6801 - accuracy: 0.5412  
Epoch 23/100  
25/25 [=====] - 0s 949us/step - loss: 0.6796 - accuracy: 0.5400  
Epoch 24/100  
25/25 [=====] - 0s 997us/step - loss: 0.6790 - accuracy: 0.5425  
Epoch 25/100  
25/25 [=====] - 0s 907us/step - loss: 0.6784 - accuracy: 0.5450  
Epoch 26/100  
25/25 [=====] - 0s 981us/step - loss: 0.6778 - accuracy: 0.5387  
Epoch 27/100  
25/25 [=====] - 0s 1ms/step - loss: 0.6770 - accuracy: 0.5425  
Epoch 28/100  
25/25 [=====] - 0s 939us/step - loss: 0.6760 - accuracy: 0.5537  
Epoch 29/100  
25/25 [=====] - 0s 1ms/step - loss: 0.6754 - accuracy: 0.5512  
Epoch 30/100  
25/25 [=====] - 0s 981us/step - loss: 0.6739 - accuracy: 0.5575  
Epoch 31/100  
25/25 [=====] - 0s 990us/step - loss: 0.6726 - accuracy: 0.5500  
Epoch 32/100  
25/25 [=====] - 0s 1ms/step - loss: 0.6711 - accuracy: 0.5512  
Epoch 33/100  
25/25 [=====] - 0s 972us/step - loss: 0.6688 - accuracy: 0.5562  
Epoch 34/100  
25/25 [=====] - 0s 1ms/step - loss: 0.6672 - accuracy: 0.5612  
Epoch 35/100  
25/25 [=====] - 0s 1ms/step - loss: 0.6660 - accuracy: 0.5888  
Epoch 36/100  
25/25 [=====] - 0s 984us/step - loss: 0.6625 - accuracy: 0.5625  
Epoch 37/100  
25/25 [=====] - 0s 1ms/step - loss: 0.6560 - accuracy: 0.5813  
Epoch 38/100  
25/25 [=====] - 0s 1ms/step - loss: 0.6521 - accuracy: 0.6025  
Epoch 39/100  
25/25 [=====] - 0s 1ms/step - loss: 0.6415 - accuracy: 0.7088  
Epoch 40/100

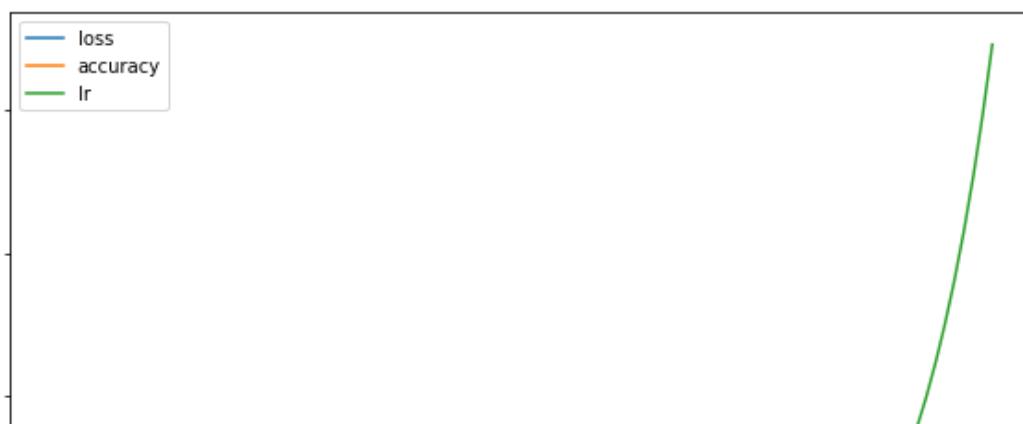
25/25 [=====] - 0s 1ms/step - loss: 0.6210 - accuracy: 0.7113  
Epoch 41/100  
25/25 [=====] - 0s 1ms/step - loss: 0.5904 - accuracy: 0.7487  
Epoch 42/100  
25/25 [=====] - 0s 1ms/step - loss: 0.5688 - accuracy: 0.7312  
Epoch 43/100  
25/25 [=====] - 0s 1ms/step - loss: 0.5346 - accuracy: 0.7563  
Epoch 44/100  
25/25 [=====] - 0s 1ms/step - loss: 0.4533 - accuracy: 0.8150  
Epoch 45/100  
25/25 [=====] - 0s 1ms/step - loss: 0.3455 - accuracy: 0.9112  
Epoch 46/100  
25/25 [=====] - 0s 1ms/step - loss: 0.2570 - accuracy: 0.9463  
Epoch 47/100  
25/25 [=====] - 0s 1ms/step - loss: 0.1968 - accuracy: 0.9575  
Epoch 48/100  
25/25 [=====] - 0s 956us/step - loss: 0.1336 - accuracy: 0.9700  
Epoch 49/100  
25/25 [=====] - 0s 1ms/step - loss: 0.1310 - accuracy: 0.9613  
Epoch 50/100  
25/25 [=====] - 0s 936us/step - loss: 0.1002 - accuracy: 0.9700  
Epoch 51/100  
25/25 [=====] - 0s 972us/step - loss: 0.1166 - accuracy: 0.9638  
Epoch 52/100  
25/25 [=====] - 0s 990us/step - loss: 0.1368 - accuracy: 0.9513  
Epoch 53/100  
25/25 [=====] - 0s 982us/step - loss: 0.0879 - accuracy: 0.9787  
Epoch 54/100  
25/25 [=====] - 0s 1ms/step - loss: 0.1187 - accuracy: 0.9588  
Epoch 55/100  
25/25 [=====] - 0s 1ms/step - loss: 0.0733 - accuracy: 0.9712  
Epoch 56/100  
25/25 [=====] - 0s 990us/step - loss: 0.1132 - accuracy: 0.9550  
Epoch 57/100  
25/25 [=====] - 0s 1ms/step - loss: 0.1057 - accuracy: 0.9613  
Epoch 58/100  
25/25 [=====] - 0s 1ms/step - loss: 0.0664 - accuracy: 0.9750  
Epoch 59/100  
25/25 [=====] - 0s 1ms/step - loss: 0.1901 - accuracy: 0.9275  
Epoch 60/100  
25/25 [=====] - 0s 1ms/step - loss: 0.1895 - accuracy: 0.9312  
Epoch 61/100  
25/25 [=====] - 0s 1ms/step - loss: 0.4128 - accuracy: 0.8625  
Epoch 62/100  
25/25 [=====] - 0s 1ms/step - loss: 0.1716 - accuracy: 0.9725  
Epoch 63/100  
25/25 [=====] - 0s 969us/step - loss: 0.0575 - accuracy: 0.9950  
Epoch 64/100  
25/25 [=====] - 0s 993us/step - loss: 0.1009 - accuracy: 0.9638  
Epoch 65/100  
25/25 [=====] - 0s 985us/step - loss: 0.1292 - accuracy: 0.9488  
Epoch 66/100  
25/25 [=====] - 0s 1ms/step - loss: 0.1552 - accuracy: 0.9438  
Epoch 67/100  
25/25 [=====] - 0s 1ms/step - loss: 0.2122 - accuracy: 0.9325  
Epoch 68/100  
25/25 [=====] - 0s 1ms/step - loss: 0.2178 - accuracy: 0.9100  
Epoch 69/100  
25/25 [=====] - 0s 1ms/step - loss: 0.0810 - accuracy: 0.9762  
Epoch 70/100  
25/25 [=====] - 0s 1ms/step - loss: 0.1457 - accuracy: 0.9388  
Epoch 71/100  
25/25 [=====] - 0s 1ms/step - loss: 0.3945 - accuracy: 0.8475  
Epoch 72/100  
25/25 [=====] - 0s 1ms/step - loss: 0.1845 - accuracy: 0.9212  
Epoch 73/100  
25/25 [=====] - 0s 1ms/step - loss: 0.1096 - accuracy: 0.9600  
Epoch 74/100  
25/25 [=====] - 0s 1ms/step - loss: 0.1100 - accuracy: 0.9588  
Epoch 75/100  
25/25 [=====] - 0s 1ms/step - loss: 0.1986 - accuracy: 0.9362  
Epoch 76/100

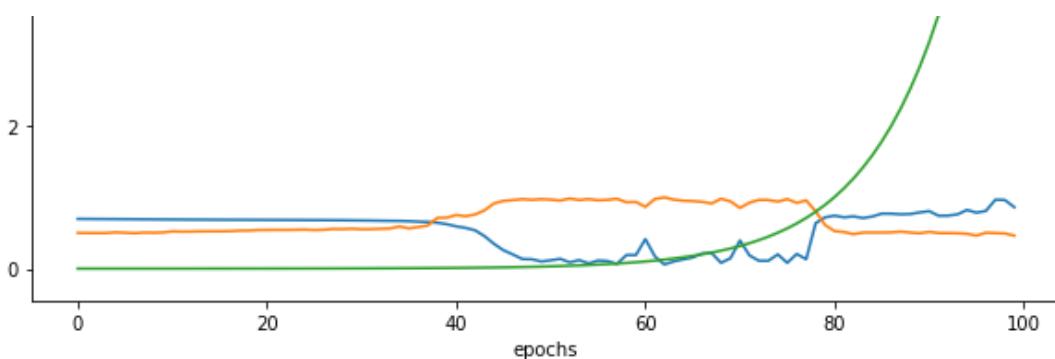
```
[4]: /usr/local/lib/python3.7/dist-packages/tensorflow/python/training/monitored_session.py:103: UserWarning: MonitoredSession: The session will be closed at the end of the monitoring loop. Consider using the 'close' method to close it manually.
  warnings.warn('MonitoredSession: The session will be closed at the end of the monitoring loop. Consider using the \'close\' method to close it manually.')
25/25 [=====] - 0s 1ms/step - loss: 0.0812 - accuracy: 0.9712
Epoch 77/100
25/25 [=====] - 0s 1ms/step - loss: 0.2043 - accuracy: 0.9187
Epoch 78/100
25/25 [=====] - 0s 1ms/step - loss: 0.1318 - accuracy: 0.9538
Epoch 79/100
25/25 [=====] - 0s 986us/step - loss: 0.6341 - accuracy: 0.8000
Epoch 80/100
25/25 [=====] - 0s 1ms/step - loss: 0.7168 - accuracy: 0.6087
Epoch 81/100
25/25 [=====] - 0s 1ms/step - loss: 0.7397 - accuracy: 0.5238
Epoch 82/100
25/25 [=====] - 0s 1ms/step - loss: 0.7165 - accuracy: 0.5113
Epoch 83/100
25/25 [=====] - 0s 1ms/step - loss: 0.7297 - accuracy: 0.4812
Epoch 84/100
25/25 [=====] - 0s 1ms/step - loss: 0.7066 - accuracy: 0.5038
Epoch 85/100
25/25 [=====] - 0s 1ms/step - loss: 0.7282 - accuracy: 0.5038
Epoch 86/100
25/25 [=====] - 0s 1ms/step - loss: 0.7687 - accuracy: 0.5038
Epoch 87/100
25/25 [=====] - 0s 1ms/step - loss: 0.7680 - accuracy: 0.5063
Epoch 88/100
25/25 [=====] - 0s 1ms/step - loss: 0.7586 - accuracy: 0.5163
Epoch 89/100
25/25 [=====] - 0s 1ms/step - loss: 0.7630 - accuracy: 0.5038
Epoch 90/100
25/25 [=====] - 0s 1ms/step - loss: 0.7854 - accuracy: 0.4938
Epoch 91/100
25/25 [=====] - 0s 1ms/step - loss: 0.8033 - accuracy: 0.5138
Epoch 92/100
25/25 [=====] - 0s 1ms/step - loss: 0.7382 - accuracy: 0.4963
Epoch 93/100
25/25 [=====] - 0s 1ms/step - loss: 0.7402 - accuracy: 0.4963
Epoch 94/100
25/25 [=====] - 0s 1ms/step - loss: 0.7582 - accuracy: 0.4938
Epoch 95/100
25/25 [=====] - 0s 1ms/step - loss: 0.8194 - accuracy: 0.4863
Epoch 96/100
25/25 [=====] - 0s 1ms/step - loss: 0.7818 - accuracy: 0.4613
Epoch 97/100
25/25 [=====] - 0s 961us/step - loss: 0.8059 - accuracy: 0.5013
Epoch 98/100
25/25 [=====] - 0s 1ms/step - loss: 0.9631 - accuracy: 0.4963
Epoch 99/100
25/25 [=====] - 0s 1ms/step - loss: 0.9587 - accuracy: 0.4913
Epoch 100/100
25/25 [=====] - 0s 1ms/step - loss: 0.8582 - accuracy: 0.4613
```

**Now our model has finished training, let's have a look at the training history.**

In [ ]:

```
# Checkout the history
pd.DataFrame(history.history).plot(figsize=(10,7), xlabel="epochs");
```





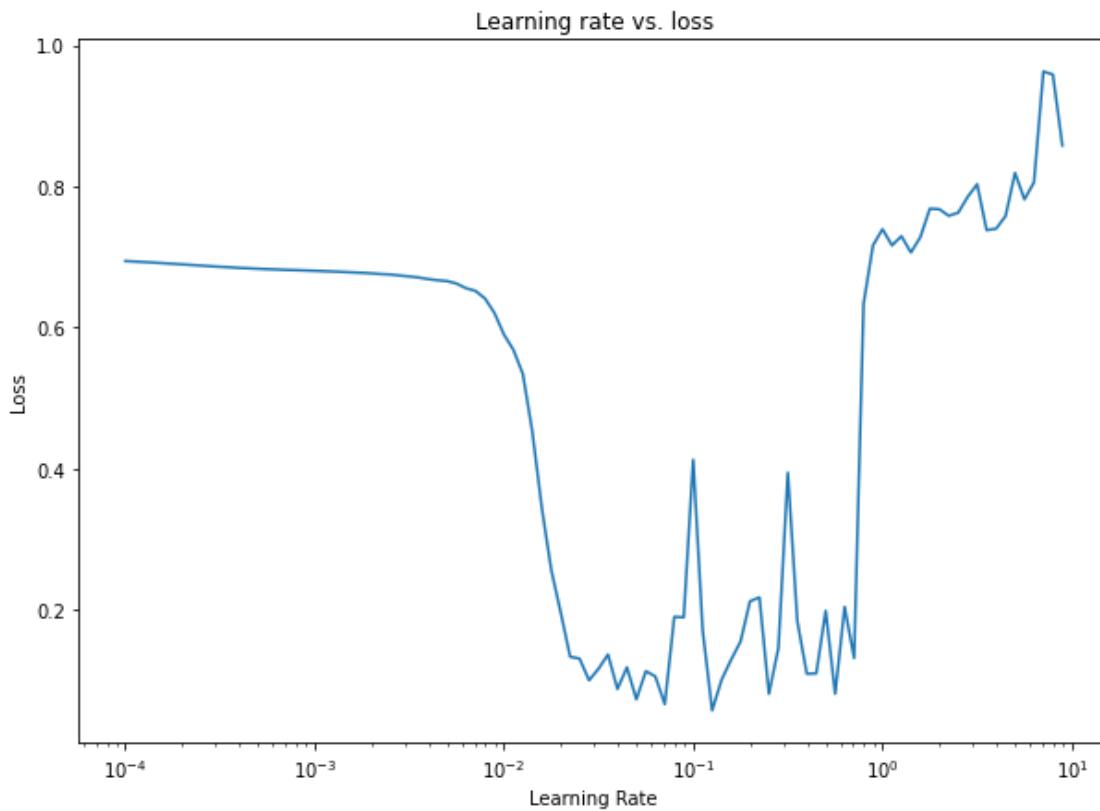
As you see the learning rate exponentially increases as the number of epochs increases.

And you can see the model's accuracy goes up (and loss goes down) at a specific point when the learning rate slowly increases.

To figure out where this inflection point is, we can plot the loss versus the log-scale learning rate.

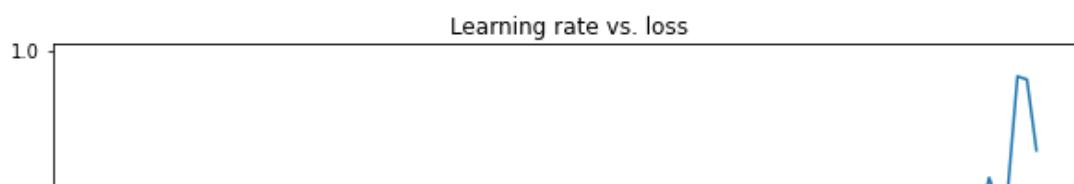
In [ ]:

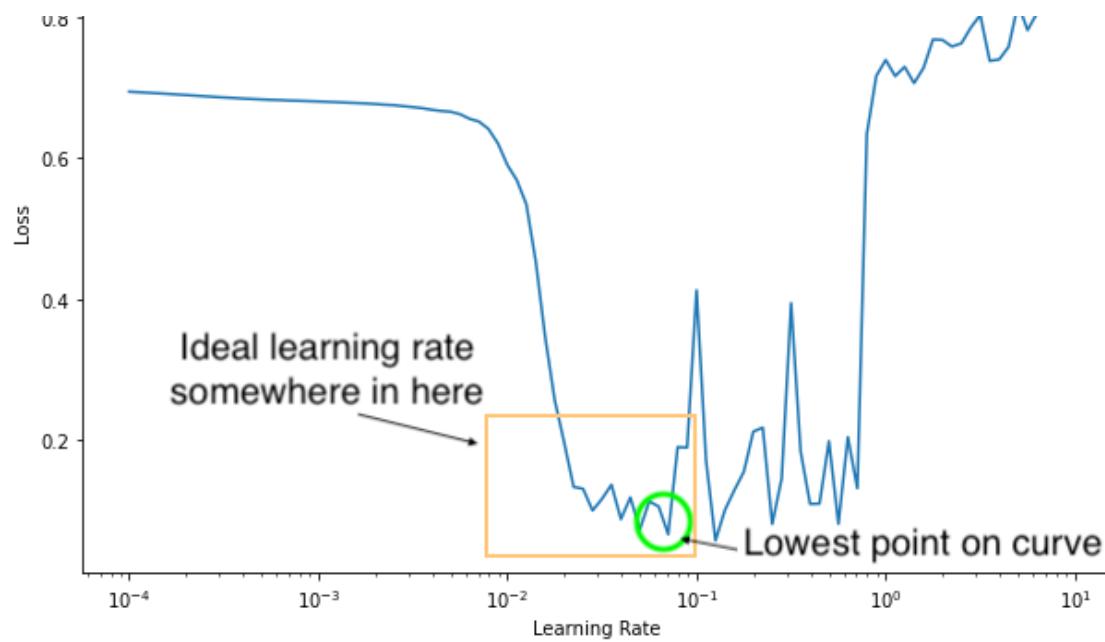
```
# Plot the learning rate versus the loss
lrs = 1e-4 * (10 ** (np.arange(100)/20))
plt.figure(figsize=(10, 7))
plt.semilogx(lrs, history.history["loss"]) # we want the x-axis (learning rate) to be log scale
plt.xlabel("Learning Rate")
plt.ylabel("Loss")
plt.title("Learning rate vs. loss");
```



To figure out the ideal value of the learning rate (at least the ideal value to begin training our model), the rule of thumb is to take the learning rate value where the loss is still decreasing but not quite flattened out (usually about 10x smaller than the bottom of the curve).

In this case, our ideal learning rate ends up between  $10^{-2}$  and  $10^{-1}$ .





**The ideal learning rate at the start of model training is somewhere just before the loss curve bottoms out (a value where the loss is still decreasing).**

In [ ]:

```
# Example of other typical learning rate values
10**0, 10**-1, 10**-2, 10**-3, 1e-4
```

Out[ ]:

```
(1, 0.1, 0.01, 0.001, 0.0001)
```

Now we've estimated the ideal learning rate (we'll use `0.02`) for our model, let's refit it.

In [ ]:

```
# Set the random seed
tf.random.set_seed(42)

# Create the model
model_10 = tf.keras.Sequential([
    tf.keras.layers.Dense(4, activation="relu"),
    tf.keras.layers.Dense(4, activation="relu"),
    tf.keras.layers.Dense(1, activation="sigmoid")
])

# Compile the model with the ideal learning rate
model_10.compile(loss="binary_crossentropy",
                  optimizer=tf.keras.optimizers.Adam(lr=0.02), # to adjust the learning rate,
# you need to use tf.keras.optimizers.Adam (not "adam")
                  metrics=["accuracy"])

# Fit the model for 20 epochs (5 less than before)
history = model_10.fit(X_train, y_train, epochs=20)
```

```
Epoch 1/20
25/25 [=====] - 0s 1ms/step - loss: 0.6837 - accuracy: 0.5600
Epoch 2/20
25/25 [=====] - 0s 970us/step - loss: 0.6744 - accuracy: 0.5750
Epoch 3/20
25/25 [=====] - 0s 969us/step - loss: 0.6626 - accuracy: 0.5875
Epoch 4/20
25/25 [=====] - 0s 992us/step - loss: 0.6332 - accuracy: 0.6388
Epoch 5/20
25/25 [=====] - 0s 1ms/step - loss: 0.5830 - accuracy: 0.7563
Epoch 6/20
25/25 [=====] - 0s 1ms/step - loss: 0.4907 - accuracy: 0.8313
Epoch 7/20
25/25 [=====] - 0s 1ms/step - loss: 0.4251 - accuracy: 0.8450
Epoch 8/20
```

```
25/25 [=====] - 0s 1ms/step - loss: 0.3596 - accuracy: 0.8875
Epoch 9/20
25/25 [=====] - 0s 1ms/step - loss: 0.3152 - accuracy: 0.9100
Epoch 10/20
25/25 [=====] - 0s 1ms/step - loss: 0.2512 - accuracy: 0.9500
Epoch 11/20
25/25 [=====] - 0s 1ms/step - loss: 0.2152 - accuracy: 0.9500
Epoch 12/20
25/25 [=====] - 0s 1ms/step - loss: 0.1721 - accuracy: 0.9750
Epoch 13/20
25/25 [=====] - 0s 1ms/step - loss: 0.1443 - accuracy: 0.9837
Epoch 14/20
25/25 [=====] - 0s 1ms/step - loss: 0.1232 - accuracy: 0.9862
Epoch 15/20
25/25 [=====] - 0s 1ms/step - loss: 0.1085 - accuracy: 0.9850
Epoch 16/20
25/25 [=====] - 0s 1ms/step - loss: 0.0940 - accuracy: 0.9937
Epoch 17/20
25/25 [=====] - 0s 988us/step - loss: 0.0827 - accuracy: 0.9962
Epoch 18/20
25/25 [=====] - 0s 1ms/step - loss: 0.0798 - accuracy: 0.9937
Epoch 19/20
25/25 [=====] - 0s 1ms/step - loss: 0.0845 - accuracy: 0.9875
Epoch 20/20
25/25 [=====] - 0s 991us/step - loss: 0.0790 - accuracy: 0.9887
```

Nice! With a little higher learning rate ( 0.02 instead of 0.01 ) we reach a higher accuracy than model\_8 in less epochs ( 20 instead of 25 ).

Practice: Now you've seen an example of what can happen when you change the learning rate, try changing the learning rate value in the [TensorFlow Playground](#) and see what happens. What happens if you increase it? What happens if you decrease it?

In [ ]:

```
# Evaluate model on the test dataset
model_10.evaluate(X_test, y_test)
```

```
7/7 [=====] - 0s 2ms/step - loss: 0.0574 - accuracy: 0.9900
```

Out [ ]:

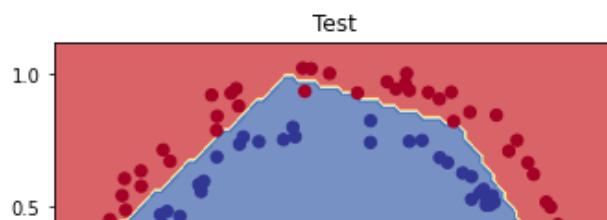
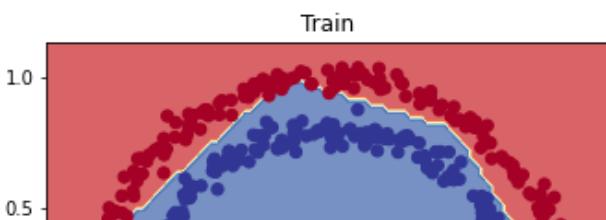
```
[0.05740181356668472, 0.9900000095367432]
```

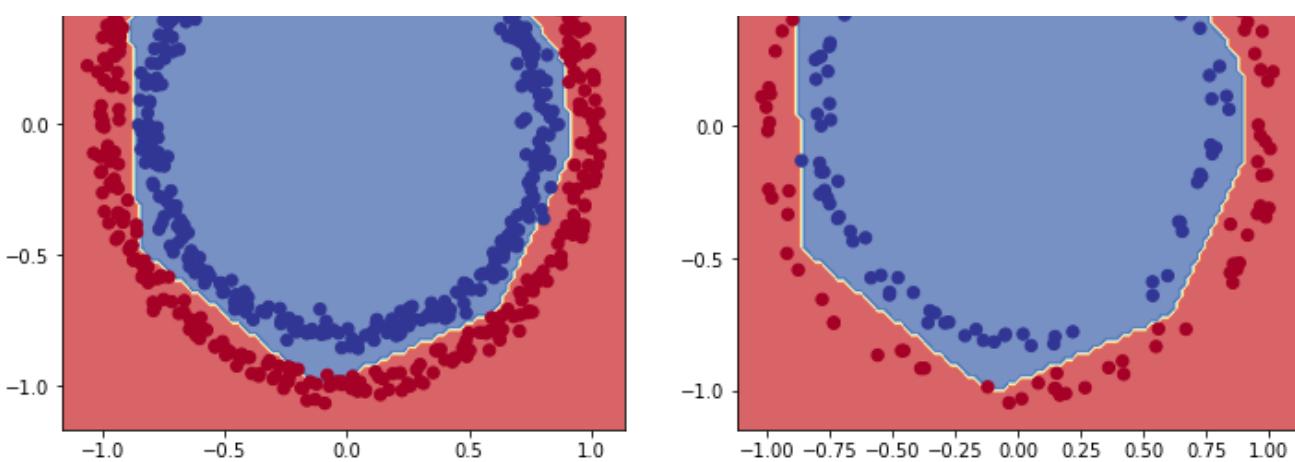
Let's see how the predictions look.

In [ ]:

```
# Plot the decision boundaries for the training and test sets
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.title("Train")
plot_decision_boundary(model_10, X=X_train, y=y_train)
plt.subplot(1, 2, 2)
plt.title("Test")
plot_decision_boundary(model_10, X=X_test, y=y_test)
plt.show()
```

```
doing binary classification...
doing binary classification...
```





And as we can see, almost perfect again.

These are the kind of experiments you'll be running often when building your own models.

Start with default settings and see how they perform on your data.

And if they don't perform as well as you'd like, improve them.

Let's look at a few more ways to evaluate our classification models.

## More classification evaluation methods

Alongside the visualizations we've been making, there are a number of different evaluation metrics we can use to evaluate our classification models.

Metric name/Evaluation method	Definition	Code
Accuracy	Out of 100 predictions, how many does your model get correct? E.g. 95% accuracy means it gets 95/100 predictions correct.	<a href="#">sklearn.metrics.accuracy_score()</a> or <a href="#">tf.keras.metrics.Accuracy()</a>
Precision	Proportion of true positives over total number of samples. Higher precision leads to less false positives (model predicts 1 when it should've been 0).	<a href="#">sklearn.metrics.precision_score()</a> or <a href="#">tf.keras.metrics.Precision()</a>
Recall	Proportion of true positives over total number of true positives and false negatives (model predicts 0 when it should've been 1). Higher recall leads to less false negatives.	<a href="#">sklearn.metrics.recall_score()</a> or <a href="#">tf.keras.metrics.Recall()</a>
F1-score	Combines precision and recall into one metric. 1 is best, 0 is worst.	<a href="#">sklearn.metrics.f1_score()</a>
Confusion matrix	Compares the predicted values with the true values in a tabular way, if 100% correct, all values in the matrix will be top left to bottom right (diagnol line).	Custom function or <a href="#">sklearn.metrics.plot_confusion_matrix()</a>
Classification report	Collection of some of the main classification metrics such as precision, recall and f1-score.	<a href="#">sklearn.metrics.classification_report()</a>

▀ Note: Every classification problem will require different kinds of evaluation methods. But you should be familiar with at least the ones above.

Let's start with accuracy.

Because we passed `["accuracy"]` to the `metrics` parameter when we compiled our model, calling `evaluate()` on it will return the loss as well as accuracy.

In [ ]:

```
# Check the accuracy of our model
loss, accuracy = model_10.evaluate(X_test, y_test)
print(f"Model loss on test set: {loss}")
```

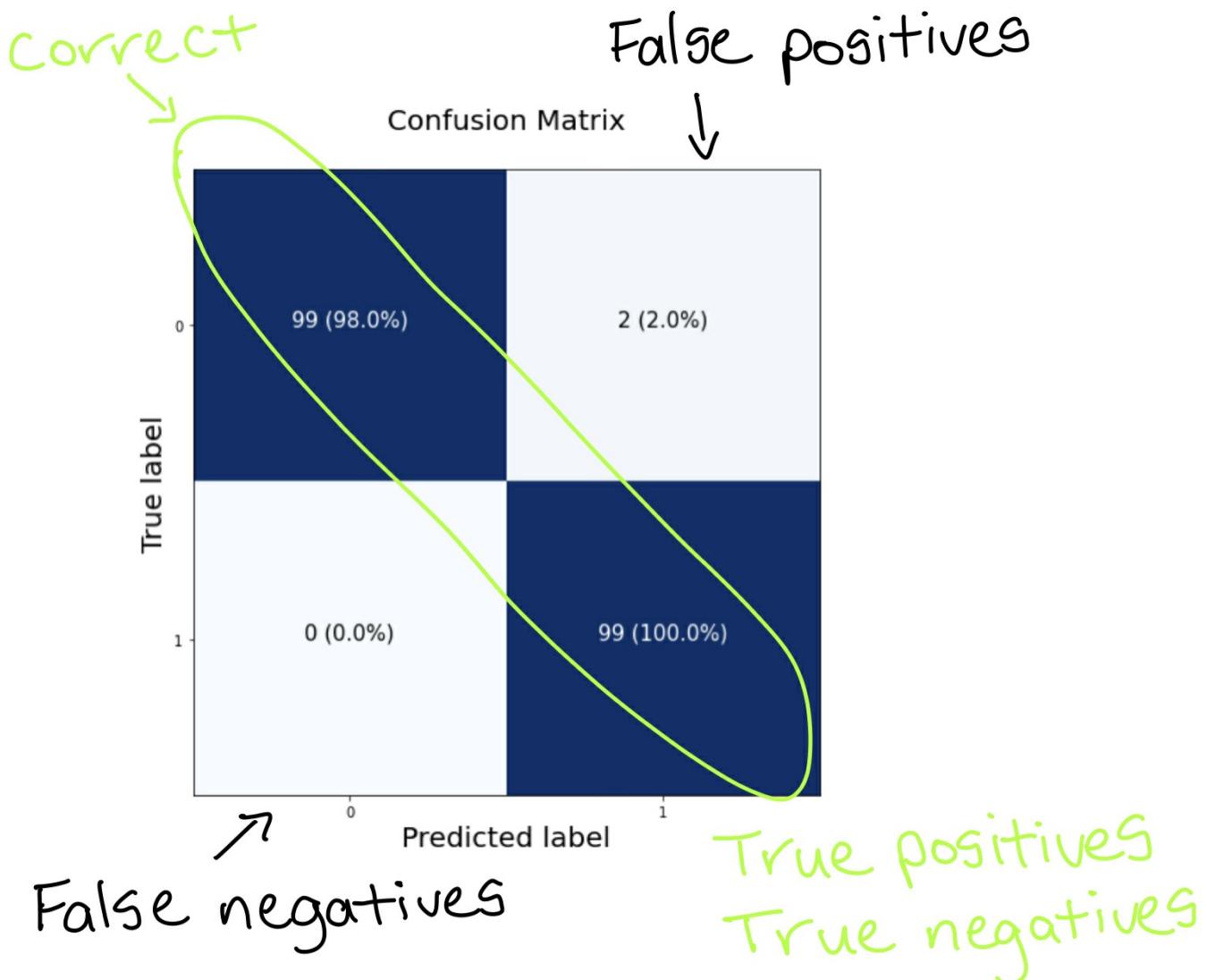
```
print(f"Model accuracy on test set: {(accuracy*100):.2f}%")
```

```
7/7 [=====] - 0s 1ms/step - loss: 0.0574 - accuracy: 0.9900
```

```
Model loss on test set: 0.05740181356668472
```

```
Model accuracy on test set: 99.00%
```

How about a confusion matrix?



Anatomy of a confusion matrix (what we're going to be creating). Correct predictions appear down the diagonal (from top left to bottom right).

We can make a confusion matrix using [Scikit-Learn's `confusion\_matrix`](#) method.

In [ ]:

```
# Create a confusion matrix
from sklearn.metrics import confusion_matrix

# Make predictions
y_preds = model_10.predict(X_test)

# Create confusion matrix
confusion_matrix(y_test, y_preds)
```

```
-----  
ValueError                                                 Traceback (most recent call last)  
<ipython-input-54-f9843efd97f5> in <module>()  
      6  
      7 # Create confusion matrix  
----> 8 confusion_matrix(y_test, y_preds)  
  
/usr/local/lib/python3.6/dist-packages/sklearn/metrics/_classification.py in confusion_matrix(y_true, y_pred, labels, sample_weight, normalize)
```

```
267 """
--> 268     y_type, y_true, y_pred = _check_targets(y_true, y_pred)
 269     if y_type not in ("binary", "multiclass"):
 270         raise ValueError("%s is not supported" % y_type)

/usr/local/lib/python3.6/dist-packages/sklearn/metrics/_classification.py in _check_targets(y_true, y_pred)
    88     if len(y_type) > 1:
    89         raise ValueError("Classification metrics can't handle a mix of {0} "
--> 90                         "and {1} targets".format(type_true, type_pred))
 91
 92     # We can't have more than one value on y_type => The set is no more needed
```

`ValueError`: Classification metrics can't handle a mix of binary and continuous targets

Ahh, it seems our predictions aren't in the format they need to be.

Let's check them out.

In [ ]:

```
# View the first 10 predictions
y_preds[:10]
```

Out[ ]:

```
array([[9.8526537e-01],
       [9.9923790e-01],
       [9.9032342e-01],
       [9.9706948e-01],
       [3.9622882e-01],
       [1.8126875e-02],
       [9.6829069e-01],
       [1.9746691e-02],
       [9.9967170e-01],
       [5.6460500e-04]], dtype=float32)
```

What about our test labels?

In [ ]:

```
# View the first 10 test labels
y_test[:10]
```

Out[ ]:

```
array([1, 1, 1, 1, 0, 0, 1, 0, 1, 0])
```

It looks like we need to get our predictions into the binary format (0 or 1).

But you might be wondering, what format are they currently in?

In their current format (`9.8526537e-01`), they're in a form called **prediction probabilities**.

You'll see this often with the outputs of neural networks. Often they won't be exact values but more a probability of how *likely* they are to be one value or another.

So one of the steps you'll often see after making predictions with a neural network is converting the prediction probabilities into labels.

In our case, since our ground truth labels (`y_test`) are binary (0 or 1), we can convert the prediction probabilities using to their binary form using `tf.round()`.

In [ ]:

```
# Convert prediction probabilities to binary format and view the first 10
tf.round(y_preds)[:10]
```

Out[ ]:

```
<tf.Tensor: shape=(10, 1), dtype=float32, numpy=
array([[1.],
       [1.],
       [1.],
       [0.],
       [0.],
       [1.],
       [0.],
       [1.],
       [0.]], dtype=float32)>
```

**Wonderful! Now we can use the `confusion_matrix` function.**

In [ ]:

```
# Create a confusion matrix
confusion_matrix(y_test, tf.round(y_preds))
```

Out[ ]:

```
array([[99,  2],
       [ 0, 99]])
```

Alright, we can see the highest numbers are down the diagonal (from top left to bottom right) so this a good sign, but the rest of the matrix doesn't really tell us much.

How about we make a function to make our confusion matrix a little more visual?

In [ ]:

```
# Note: The following confusion matrix code is a remix of Scikit-Learn's
# plot_confusion_matrix function - https://scikit-learn.org/stable/modules/generated/sklearn.metrics.plot_confusion_matrix.html
# and Made with ML's introductory notebook - https://github.com/GokuMohandas/MadeWithML/blob/main/notebooks/08_Neural_Networks.ipynb
```

```
figsize = (10, 10)
```

```
# Create the confusion matrix
cm = confusion_matrix(y_test, tf.round(y_preds))
cm_norm = cm.astype("float") / cm.sum(axis=1)[:, np.newaxis] # normalize it
n_classes = cm.shape[0]
```

```
# Let's prettyify it
fig, ax = plt.subplots(figsize=figsize)
# Create a matrix plot
cax = ax.matshow(cm, cmap=plt.cm.Blues) # https://matplotlib.org/3.2.0/api/_as_gen/matplotlib.axes.Axes.matshow.html
fig.colorbar(cax)
```

```
# Create classes
classes = False
```

```
if classes:
    labels = classes
else:
    labels = np.arange(cm.shape[0])
```

```
# Label the axes
ax.set(title="Confusion Matrix",
       xlabel="Predicted label",
       ylabel="True label",
       xticks=np.arange(n_classes),
       yticks=np.arange(n_classes),
       xticklabels=labels,
       yticklabels=labels)
```

```
# Set x-axis labels to bottom
ax.xaxis.set_label_position("bottom")
```

```

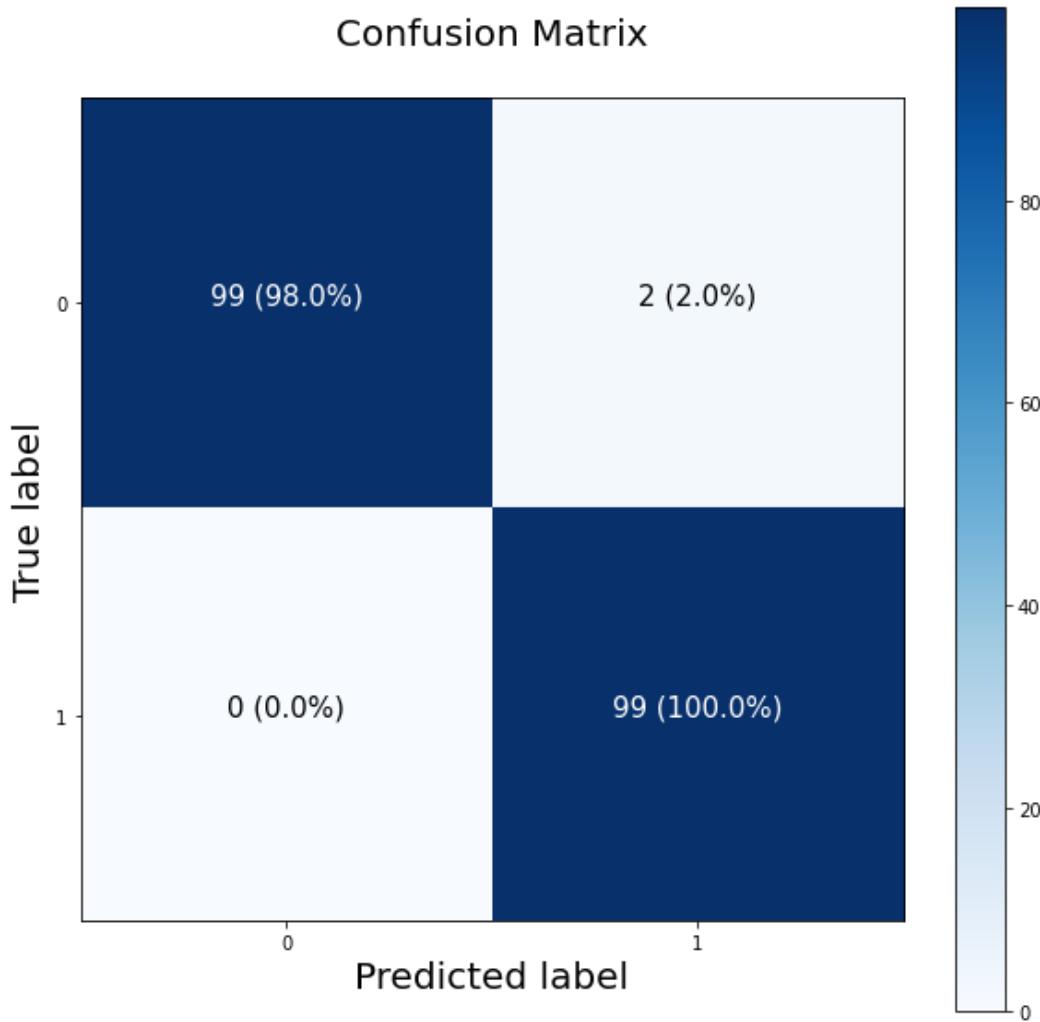
ax.xaxis.tick_bottom()

# Adjust label size
ax.xaxis.label.set_size(20)
ax.yaxis.label.set_size(20)
ax.title.set_size(20)

# Set threshold for different colors
threshold = (cm.max() + cm.min()) / 2.

# Plot the text on each cell
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, f"{cm[i, j]} ({cm_norm[i, j]*100:.1f}%)",
              horizontalalignment="center",
              color="white" if cm[i, j] > threshold else "black",
              size=15)

```



**That looks much better. It seems our model has made almost perfect predictions on the test set except for two false positives (top right corner).**

In [ ]:

```

# What does itertools.product do? Combines two things into each combination
import itertools
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    print(i, j)

```

```

0 0
0 1
1 0
1 1

```

## Working with a larger example (multiclass classification)

We've seen a binary classification example (predicting if a data point is part of a red circle or blue circle) but

**what if you had multiple different classes of things?**

For example, say you were a fashion company and you wanted to build a neural network to predict whether a piece of clothing was a shoe, a shirt or a jacket (3 different options).

**When you have more than two classes as an option, this is known as **multiclass classification**.**

The good news is, the things we've learned so far (with a few tweaks) can be applied to multiclass classification problems as well.

## Let's see it in action.

To start, we'll need some data. The good thing for us is TensorFlow has a multiclass classification dataset known as [Fashion MNIST built-in](#). Meaning we can get started straight away.

We can import it using the `tf.keras.datasets` module.

**Resource:** The following multiclass classification problem has been adapted from the [TensorFlow classification guide](#). A good exercise would be to once you've gone through the following example, replicate the TensorFlow guide.

In [ ]:

```
import tensorflow as tf
from tensorflow.keras.datasets import fashion_mnist

# The data has already been sorted into training and test sets for us
(train_data, train_labels), (test_data, test_labels) = fashion_mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
32768/29515 [=====] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26427392/26421880 [=====] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
8192/5148 [=====] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4423680/4422102 [=====] - 0s 0us/step
```

**Now let's check out an example.**

In [ ]:

```
# Show the first training example
print(f"Training sample:\n{train_data[0]}\n")
print(f"Training label: {train_labels[0]}")
```

Training sample:

```

223 223 215 213 164 127 123 196 229 0]
[ 0   0   0   0   0   0   0   0   0   0   0   0   0   0   183 225 216 223 228
235 227 224 222 224 221 223 245 173 0]
[ 0   0   0   0   0   0   0   0   0   0   0   0   0   0   193 228 218 213 198
180 212 210 211 213 223 220 243 202 0]
[ 0   0   0   0   0   0   0   0   0   0   1   3   0   12 219 220 212 218 192
169 227 208 218 224 212 226 197 209 52]
[ 0   0   0   0   0   0   0   0   0   0   0   6   0   99 244 222 220 218 203
198 221 215 213 222 220 245 119 167 56]
[ 0   0   0   0   0   0   0   0   0   0   4   0   0   55 236 228 230 228 240
232 213 218 223 234 217 217 209 92 0]
[ 0   0   1   4   6   7   2   0   0   0   0   0   0   0   237 226 217 223 222 219
222 221 216 223 229 215 218 255 77 0]
[ 0   3   0   0   0   0   0   0   0   0   62 145 204 228 207 213 221 218 208
211 218 224 223 219 215 224 244 159 0]
[ 0   0   0   0   18  44  82 107 189 228 220 222 217 226 200 205 211 230
224 234 176 188 250 248 233 238 215 0]
[ 0   57 187 208 224 221 224 208 204 214 208 209 200 159 245 193 206 223
255 255 221 234 221 211 220 232 246 0]
[ 3   202 228 224 221 211 211 214 205 205 205 220 240 80 150 255 229 221
188 154 191 210 204 209 222 228 225 0]
[ 98 233 198 210 222 229 229 234 249 220 194 215 217 241 65 73 106 117
168 219 221 215 217 223 223 224 229 29]
[ 75 204 212 204 193 205 211 225 216 185 197 206 198 213 240 195 227 245
239 223 218 212 209 222 220 221 230 67]
[ 48 203 183 194 213 197 185 190 194 192 202 214 219 221 220 236 225 216
199 206 186 181 177 172 181 205 206 115]
[ 0   122 219 193 179 171 183 196 204 210 213 207 211 210 200 196 194 191
195 191 198 192 176 156 167 177 210 92]
[ 0   0   74 189 212 191 175 172 175 181 185 188 189 188 193 198 204 209
210 210 211 188 188 194 192 216 170 0]
[ 2   0   0   0   66 200 222 237 239 242 246 243 244 221 220 193 191 179
182 182 181 176 166 168 99 58 0   0]
[ 0   0   0   0   0   0   0   40 61 44 72 41 35 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]

```

Training label: 9

**Woah, we get a large list of numbers, followed (the data) by a single number (the class label).**

## What about the shapes?

In [ ]:

```
# Check the shape of our data  
train_data.shape, train_labels.shape, test_data.shape, test_labels.shape
```

Out [ ] :

((60000, 28, 28), (60000,), (10000, 28, 28), (10000,))

In [ ]:

```
# Check shape of a single example  
train_data[0].shape, train_labels[0].shape
```

Out[ ]:

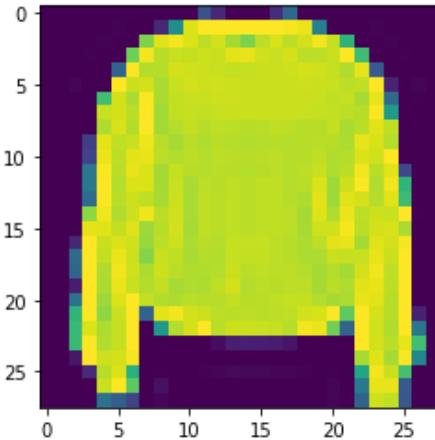
( (28, 28), () )

Okay, 60,000 training examples each with shape (28, 28) and a label each as well as 10,000 test examples of shape (28, 28).

**But these are just numbers, let's visualize.**

In [ ]:

```
# Plot a single example
import matplotlib.pyplot as plt
plt.imshow(train_data[7]);
```



Hmm, but what about its label?

In [ ]:

```
# Check our samples label
train_labels[7]
```

Out [ ]:

2

It looks like our labels are in numerical form. And while this is fine for a neural network, you might want to have them in human readable form.

Let's create a small list of the class names (we can find them on [the dataset's GitHub page](#)).

**Note:** Whilst this dataset has been prepared for us and ready to go, it's important to remember many datasets won't be ready to go like this one. Often you'll have to do a few preprocessing steps to have it ready to use with a neural network (we'll see more of this when we work with our own data later).

In [ ]:

```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

# How many classes are there (this'll be our output shape)?
len(class_names)
```

Out [ ]:

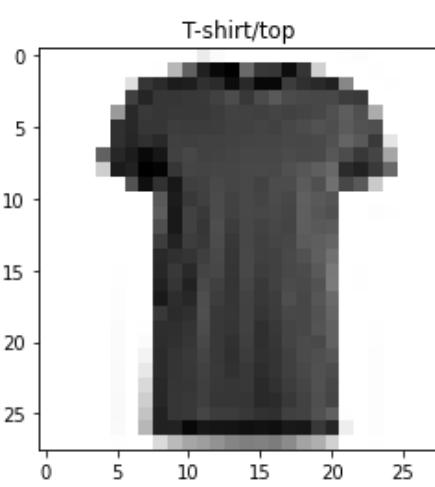
10

Now we have these, let's plot another example.

**Question:** Pay particular attention to what the data we're working with *looks* like. Is it only straight lines? Or does it have non-straight lines as well? Do you think if we wanted to find patterns in the photos of clothes (which are actually collections of pixels), will our model need non-linearities (non-straight lines) or not?

In [ ]:

```
# Plot an example image and its label
plt.imshow(train_data[17], cmap=plt.cm.binary) # change the colours to black & white
plt.title(class_names[train_labels[17]]);
```



In [ ]:

```
# Plot multiple random images of fashion MNIST
import random
plt.figure(figsize=(7, 7))
for i in range(4):
    ax = plt.subplot(2, 2, i + 1)
    rand_index = random.choice(range(len(train_data)))
    plt.imshow(train_data[rand_index], cmap=plt.cm.binary)
    plt.title(class_names[train_labels[rand_index]])
    plt.axis(False)
```



Alright, let's build a model to figure out the relationship between the pixel values and their labels.

Since this is a multiclass classification problem, we'll need to make a few changes to our architecture (inline with Table 1 above):

- The **input shape** will have to deal with 28x28 tensors (the height and width of our images).
  - We're actually going to squash the input into a tensor (vector) of shape `(784)`.
- The **output shape** will have to be 10 because we need our model to predict for 10 different classes.
  - We'll also change the `activation` parameter of our output layer to be `"softmax"` instead of `'sigmoid'`. As we'll see the `"softmax"` activation function outputs a series of values between 0 & 1 (the same shape as **output shape**, which together add up to ~1. The index with the highest value is predicted by the model to be the most *likely* class.
- We'll need to change our loss function from a binary loss function to a multiclass loss function.
  - More specifically, since our labels are in integer form, we'll use `tf.keras.losses.SparseCategoricalCrossentropy()`, if our labels were one-hot encoded (e.g.

they looked something like [0, 0, 1, 0, 0...], we'd use  
`tf.keras.losses.CategoricalCrossentropy()`.

- We'll also use the `validation_data` parameter when calling the `fit()` function. This will give us an idea of how the model performs on the test set during training.

You ready? Let's go.

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Create the model
model_11 = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)), # input layer (we had to reshape 28x28
    to 784, the Flatten layer does this for us)
    tf.keras.layers.Dense(4, activation="relu"),
    tf.keras.layers.Dense(4, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax") # output shape is 10, activation is softmax
])

# Compile the model
model_11.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(), # different loss function for multiclass classification
                  optimizer=tf.keras.optimizers.Adam(),
                  metrics=["accuracy"])

# Fit the model
non_norm_history = model_11.fit(train_data,
                                  train_labels,
                                  epochs=10,
                                  validation_data=(test_data, test_labels)) # see how the model performs on the test set during training
```

```
Epoch 1/10
1875/1875 [=====] - 3s 1ms/step - loss: 2.1631 - accuracy: 0.162
2 - val_loss: 1.7951 - val_accuracy: 0.2100
Epoch 2/10
1875/1875 [=====] - 3s 1ms/step - loss: 1.7094 - accuracy: 0.251
4 - val_loss: 1.6439 - val_accuracy: 0.3022
Epoch 3/10
1875/1875 [=====] - 3s 1ms/step - loss: 1.6353 - accuracy: 0.284
7 - val_loss: 1.6003 - val_accuracy: 0.2818
Epoch 4/10
1875/1875 [=====] - 3s 1ms/step - loss: 1.6099 - accuracy: 0.285
8 - val_loss: 1.5964 - val_accuracy: 0.2958
Epoch 5/10
1875/1875 [=====] - 3s 1ms/step - loss: 1.5958 - accuracy: 0.304
0 - val_loss: 1.5948 - val_accuracy: 0.3005
Epoch 6/10
1875/1875 [=====] - 2s 1ms/step - loss: 1.5842 - accuracy: 0.311
5 - val_loss: 1.5678 - val_accuracy: 0.3195
Epoch 7/10
1875/1875 [=====] - 3s 1ms/step - loss: 1.5700 - accuracy: 0.320
5 - val_loss: 1.5695 - val_accuracy: 0.3161
Epoch 8/10
1875/1875 [=====] - 3s 1ms/step - loss: 1.5605 - accuracy: 0.325
8 - val_loss: 1.5526 - val_accuracy: 0.3343
Epoch 9/10
1875/1875 [=====] - 3s 1ms/step - loss: 1.5626 - accuracy: 0.326
9 - val_loss: 1.5467 - val_accuracy: 0.3373
Epoch 10/10
1875/1875 [=====] - 2s 1ms/step - loss: 1.5451 - accuracy: 0.332
7 - val_loss: 1.5339 - val_accuracy: 0.3542
```

In [ ]:

```
# Check the shapes of our model
# Note: the "None" in (None, 784) is for batch_size, we'll cover this in a later module
```

```
model_11.summary()
```

Model: "sequential\_11"

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense_28 (Dense)	(None, 4)	3140
dense_29 (Dense)	(None, 4)	20
dense_30 (Dense)	(None, 10)	50
=====		
Total params:	3,210	
Trainable params:	3,210	
Non-trainable params:	0	

Alright, our model gets to about ~35% accuracy after 10 epochs using a similar style model to what we used on our binary classification problem.

Which is better than guessing (guessing with 10 classes would result in about 10% accuracy) but we can do better.

Do you remember when we talked about neural networks preferring numbers between 0 and 1? (if not, treat this as a reminder)

Well, right now, the data we have isn't between 0 and 1, in other words, it's not normalized (hence why we used the `non_norm_history` variable when calling `fit()`). It's pixel values are between 0 and 255.

Let's see.

In [ ]:

```
# Check the min and max values of the training data
train_data.min(), train_data.max()
```

Out[ ]:

```
(0, 255)
```

We can get these values between 0 and 1 by dividing the entire array by the maximum: 255.0 (dividing by a float also converts to a float).

Doing so will result in all of our data being between 0 and 1 (known as scaling or normalization).

In [ ]:

```
# Divide train and test images by the maximum value (normalize it)
train_data = train_data / 255.0
test_data = test_data / 255.0

# Check the min and max values of the training data
train_data.min(), train_data.max()
```

Out[ ]:

```
(0.0, 1.0)
```

Beautiful! Now our data is between 0 and 1. Let's see what happens when we model it.

We'll use the same model as before (`model_11`) except this time the data will be normalized.

In [ ]:

```
# Set random seed
tf.random.set_seed(42)
```

```

# Create the model
model_12 = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)), # input layer (we had to reshape 28x28
    to 784)
    tf.keras.layers.Dense(4, activation="relu"),
    tf.keras.layers.Dense(4, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax") # output shape is 10, activation is softmax
])

# Compile the model
model_12.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(),
                  optimizer=tf.keras.optimizers.Adam(),
                  metrics=["accuracy"])

# Fit the model (to the normalized data)
norm_history = model_12.fit(train_data,
                             train_labels,
                             epochs=10,
                             validation_data=(test_data, test_labels))

```

```

Epoch 1/10
1875/1875 [=====] - 3s 1ms/step - loss: 1.0348 - accuracy: 0.647
4 - val_loss: 0.6937 - val_accuracy: 0.7617
Epoch 2/10
1875/1875 [=====] - 2s 1ms/step - loss: 0.6376 - accuracy: 0.775
7 - val_loss: 0.6400 - val_accuracy: 0.7820
Epoch 3/10
1875/1875 [=====] - 2s 1ms/step - loss: 0.5942 - accuracy: 0.791
4 - val_loss: 0.6247 - val_accuracy: 0.7783
Epoch 4/10
1875/1875 [=====] - 2s 1ms/step - loss: 0.5750 - accuracy: 0.797
9 - val_loss: 0.6078 - val_accuracy: 0.7881
Epoch 5/10
1875/1875 [=====] - 2s 1ms/step - loss: 0.5641 - accuracy: 0.800
6 - val_loss: 0.6169 - val_accuracy: 0.7881
Epoch 6/10
1875/1875 [=====] - 2s 1ms/step - loss: 0.5544 - accuracy: 0.804
3 - val_loss: 0.5855 - val_accuracy: 0.7951
Epoch 7/10
1875/1875 [=====] - 2s 1ms/step - loss: 0.5488 - accuracy: 0.806
3 - val_loss: 0.6097 - val_accuracy: 0.7836
Epoch 8/10
1875/1875 [=====] - 2s 1ms/step - loss: 0.5428 - accuracy: 0.807
7 - val_loss: 0.5787 - val_accuracy: 0.7971
Epoch 9/10
1875/1875 [=====] - 2s 1ms/step - loss: 0.5373 - accuracy: 0.809
7 - val_loss: 0.5698 - val_accuracy: 0.7977
Epoch 10/10
1875/1875 [=====] - 2s 1ms/step - loss: 0.5360 - accuracy: 0.812
4 - val_loss: 0.5658 - val_accuracy: 0.8014

```

**Woah, we used the exact same model as before but we with normalized data we're now seeing a much higher accuracy value!**

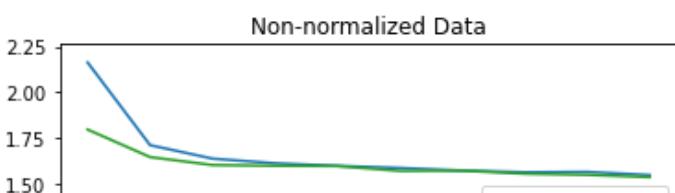
**Let's plot each model's history (their loss curves).**

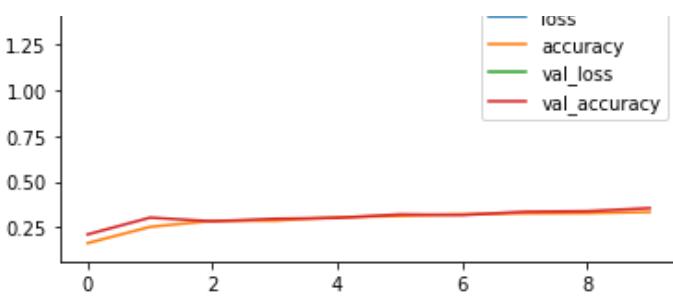
In [ ]:

```

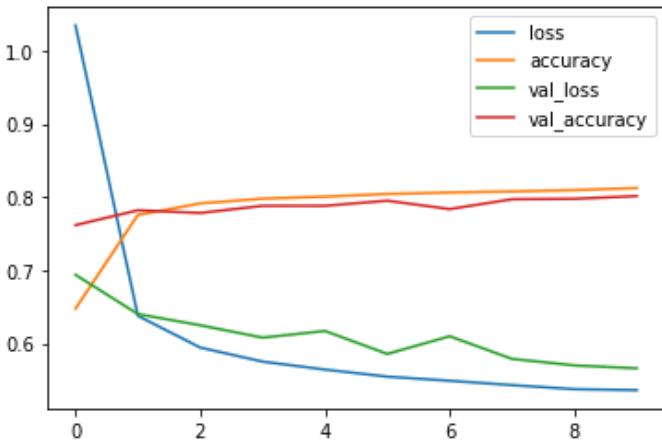
import pandas as pd
# Plot non-normalized data loss curves
pd.DataFrame(non_norm_history.history).plot(title="Non-normalized Data")
# Plot normalized data loss curves
pd.DataFrame(norm_history.history).plot(title="Normalized data");

```





Normalized data



Wow. From these two plots, we can see how much quicker our model with the normalized data (`model_12`) improved than the model with the non-normalized data (`model_11`).

**Note:** The same model with even *slightly* different data can produce *dramatically* different results. So when you're comparing models, it's important to make sure you're comparing them on the same criteria (e.g. same architecture but different data or same data but different architecture).

How about we find the ideal learning rate and see what happens?

We'll use the same architecture we've been using.

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Create the model
model_13 = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)), # input layer (we had to reshape 28x28 to 784)
    tf.keras.layers.Dense(4, activation="relu"),
    tf.keras.layers.Dense(4, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax") # output shape is 10, activation is softmax
])

# Compile the model
model_13.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(),
                  optimizer=tf.keras.optimizers.Adam(),
                  metrics=["accuracy"])

# Create the learning rate callback
lr_scheduler = tf.keras.callbacks.LearningRateScheduler(lambda epoch: 1e-3 * 10**(epoch/20))

# Fit the model
find_lr_history = model_13.fit(train_data,
                                 train_labels,
                                 epochs=40, # model already doing pretty good with current LR, probably don't need 100 epochs
                                 validation_data=(test_data, test_labels),
```

```
 callbacks=[lr_scheduler])
```

Epoch 1/40  
1875/1875 [=====] - 3s 1ms/step - loss: 1.0348 - accuracy: 0.647  
4 - val\_loss: 0.6937 - val\_accuracy: 0.7617  
Epoch 2/40  
1875/1875 [=====] - 2s 1ms/step - loss: 0.6366 - accuracy: 0.775  
9 - val\_loss: 0.6400 - val\_accuracy: 0.7808  
Epoch 3/40  
1875/1875 [=====] - 2s 1ms/step - loss: 0.5934 - accuracy: 0.791  
1 - val\_loss: 0.6278 - val\_accuracy: 0.7770  
Epoch 4/40  
1875/1875 [=====] - 2s 1ms/step - loss: 0.5749 - accuracy: 0.796  
9 - val\_loss: 0.6122 - val\_accuracy: 0.7871  
Epoch 5/40  
1875/1875 [=====] - 2s 1ms/step - loss: 0.5655 - accuracy: 0.798  
7 - val\_loss: 0.6061 - val\_accuracy: 0.7913  
Epoch 6/40  
1875/1875 [=====] - 2s 1ms/step - loss: 0.5569 - accuracy: 0.802  
2 - val\_loss: 0.5917 - val\_accuracy: 0.7940  
Epoch 7/40  
1875/1875 [=====] - 2s 1ms/step - loss: 0.5542 - accuracy: 0.803  
6 - val\_loss: 0.5898 - val\_accuracy: 0.7896  
Epoch 8/40  
1875/1875 [=====] - 2s 1ms/step - loss: 0.5509 - accuracy: 0.803  
9 - val\_loss: 0.5829 - val\_accuracy: 0.7949  
Epoch 9/40  
1875/1875 [=====] - 2s 1ms/step - loss: 0.5468 - accuracy: 0.804  
7 - val\_loss: 0.6036 - val\_accuracy: 0.7833  
Epoch 10/40  
1875/1875 [=====] - 2s 1ms/step - loss: 0.5478 - accuracy: 0.805  
8 - val\_loss: 0.5736 - val\_accuracy: 0.7974  
Epoch 11/40  
1875/1875 [=====] - 2s 1ms/step - loss: 0.5446 - accuracy: 0.805  
9 - val\_loss: 0.5672 - val\_accuracy: 0.8016  
Epoch 12/40  
1875/1875 [=====] - 2s 1ms/step - loss: 0.5432 - accuracy: 0.806  
7 - val\_loss: 0.5773 - val\_accuracy: 0.7950  
Epoch 13/40  
1875/1875 [=====] - 2s 1ms/step - loss: 0.5425 - accuracy: 0.805  
6 - val\_loss: 0.5775 - val\_accuracy: 0.7992  
Epoch 14/40  
1875/1875 [=====] - 2s 1ms/step - loss: 0.5407 - accuracy: 0.807  
8 - val\_loss: 0.5616 - val\_accuracy: 0.8075  
Epoch 15/40  
1875/1875 [=====] - 2s 1ms/step - loss: 0.5408 - accuracy: 0.805  
2 - val\_loss: 0.5773 - val\_accuracy: 0.8039  
Epoch 16/40  
1875/1875 [=====] - 2s 1ms/step - loss: 0.5437 - accuracy: 0.805  
8 - val\_loss: 0.5682 - val\_accuracy: 0.8015  
Epoch 17/40  
1875/1875 [=====] - 2s 1ms/step - loss: 0.5419 - accuracy: 0.807  
5 - val\_loss: 0.5995 - val\_accuracy: 0.7964  
Epoch 18/40  
1875/1875 [=====] - 2s 1ms/step - loss: 0.5488 - accuracy: 0.805  
8 - val\_loss: 0.5544 - val\_accuracy: 0.8087  
Epoch 19/40  
1875/1875 [=====] - 2s 1ms/step - loss: 0.5506 - accuracy: 0.804  
2 - val\_loss: 0.6068 - val\_accuracy: 0.7864  
Epoch 20/40  
1875/1875 [=====] - 2s 1ms/step - loss: 0.5537 - accuracy: 0.803  
0 - val\_loss: 0.5597 - val\_accuracy: 0.8076  
Epoch 21/40  
1875/1875 [=====] - 2s 1ms/step - loss: 0.5572 - accuracy: 0.803  
6 - val\_loss: 0.5998 - val\_accuracy: 0.7934  
Epoch 22/40  
1875/1875 [=====] - 2s 1ms/step - loss: 0.5615 - accuracy: 0.801  
3 - val\_loss: 0.5756 - val\_accuracy: 0.8034  
Epoch 23/40  
1875/1875 [=====] - 2s 1ms/step - loss: 0.5655 - accuracy: 0.801  
7 - val\_loss: 0.6386 - val\_accuracy: 0.7668  
Epoch 24/40

```

1875/1875 [=====] - 2s 1ms/step - loss: 0.5819 - accuracy: 0.796
3 - val_loss: 0.6356 - val_accuracy: 0.7869
Epoch 25/40
1875/1875 [=====] - 2s 1ms/step - loss: 0.5810 - accuracy: 0.797
7 - val_loss: 0.6481 - val_accuracy: 0.7865
Epoch 26/40
1875/1875 [=====] - 2s 1ms/step - loss: 0.5960 - accuracy: 0.790
1 - val_loss: 0.6997 - val_accuracy: 0.7802
Epoch 27/40
1875/1875 [=====] - 2s 1ms/step - loss: 0.6101 - accuracy: 0.787
0 - val_loss: 0.6124 - val_accuracy: 0.7917
Epoch 28/40
1875/1875 [=====] - 2s 1ms/step - loss: 0.6178 - accuracy: 0.784
6 - val_loss: 0.6137 - val_accuracy: 0.7962
Epoch 29/40
1875/1875 [=====] - 2s 1ms/step - loss: 0.6357 - accuracy: 0.777
1 - val_loss: 0.6655 - val_accuracy: 0.7621
Epoch 30/40
1875/1875 [=====] - 2s 1ms/step - loss: 0.6664 - accuracy: 0.767
3 - val_loss: 0.7274 - val_accuracy: 0.7454
Epoch 31/40
1875/1875 [=====] - 2s 1ms/step - loss: 0.6815 - accuracy: 0.761
1 - val_loss: 0.6861 - val_accuracy: 0.7527
Epoch 32/40
1875/1875 [=====] - 3s 1ms/step - loss: 0.7031 - accuracy: 0.757
0 - val_loss: 0.8097 - val_accuracy: 0.7441
Epoch 33/40
1875/1875 [=====] - 3s 1ms/step - loss: 0.7765 - accuracy: 0.733
7 - val_loss: 0.8163 - val_accuracy: 0.7702
Epoch 34/40
1875/1875 [=====] - 3s 1ms/step - loss: 0.7721 - accuracy: 0.740
9 - val_loss: 0.7519 - val_accuracy: 0.7000
Epoch 35/40
1875/1875 [=====] - 3s 1ms/step - loss: 0.8270 - accuracy: 0.716
9 - val_loss: 0.8102 - val_accuracy: 0.7342
Epoch 36/40
1875/1875 [=====] - 3s 1ms/step - loss: 0.8899 - accuracy: 0.693
4 - val_loss: 0.8824 - val_accuracy: 0.6822
Epoch 37/40
1875/1875 [=====] - 2s 1ms/step - loss: 0.9549 - accuracy: 0.661
1 - val_loss: 1.0329 - val_accuracy: 0.6430
Epoch 38/40
1875/1875 [=====] - 3s 1ms/step - loss: 1.0223 - accuracy: 0.628
2 - val_loss: 0.9631 - val_accuracy: 0.6314
Epoch 39/40
1875/1875 [=====] - 2s 1ms/step - loss: 1.2757 - accuracy: 0.481
0 - val_loss: 1.1771 - val_accuracy: 0.4974
Epoch 40/40
1875/1875 [=====] - 3s 1ms/step - loss: 1.5858 - accuracy: 0.326
5 - val_loss: 1.6092 - val_accuracy: 0.3048

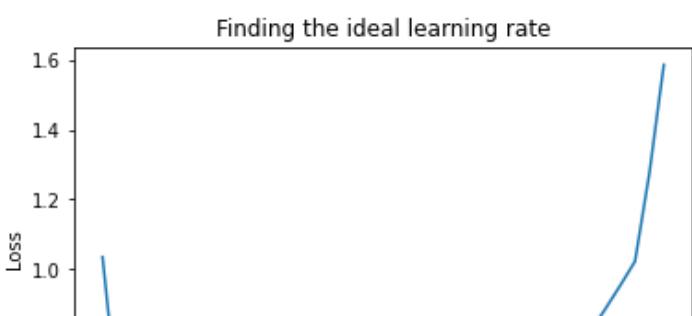
```

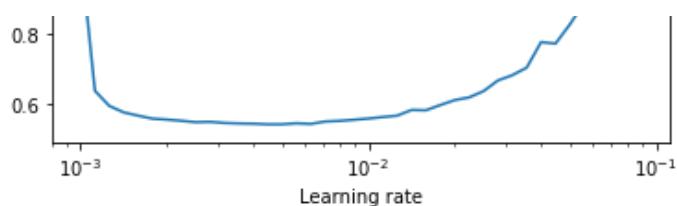
In [ ]:

```

# Plot the learning rate decay curve
import numpy as np
import matplotlib.pyplot as plt
lrs = 1e-3 * (10**(np.arange(40)/20))
plt.semilogx(lrs, find_lr.history.history["loss"]) # want the x-axis to be log-scale
plt.xlabel("Learning rate")
plt.ylabel("Loss")
plt.title("Finding the ideal learning rate");

```





In this case, it looks like somewhere close to the default learning rate of the [Adam optimizer](#) (0.001) is the ideal learning rate.

Let's refit a model using the ideal learning rate.

In [ ]:

```
# Set random seed
tf.random.set_seed(42)

# Create the model
model_14 = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)), # input layer (we had to reshape 28x28 to 784)
    tf.keras.layers.Dense(4, activation="relu"),
    tf.keras.layers.Dense(4, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax") # output shape is 10, activation is softmax
])

# Compile the model
model_14.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(),
                  optimizer=tf.keras.optimizers.Adam(lr=0.001), # ideal learning rate (same as default)
                  metrics=["accuracy"])

# Fit the model
history = model_14.fit(train_data,
                        train_labels,
                        epochs=20,
                        validation_data=(test_data, test_labels))
```

```
Epoch 1/20
1875/1875 [=====] - 3s 1ms/step - loss: 1.0348 - accuracy: 0.647
4 - val_loss: 0.6937 - val_accuracy: 0.7617
Epoch 2/20
1875/1875 [=====] - 2s 1ms/step - loss: 0.6376 - accuracy: 0.775
7 - val_loss: 0.6400 - val_accuracy: 0.7820
Epoch 3/20
1875/1875 [=====] - 2s 1ms/step - loss: 0.5942 - accuracy: 0.791
4 - val_loss: 0.6247 - val_accuracy: 0.7783
Epoch 4/20
1875/1875 [=====] - 2s 1ms/step - loss: 0.5750 - accuracy: 0.797
9 - val_loss: 0.6078 - val_accuracy: 0.7881
Epoch 5/20
1875/1875 [=====] - 2s 1ms/step - loss: 0.5641 - accuracy: 0.800
6 - val_loss: 0.6169 - val_accuracy: 0.7881
Epoch 6/20
1875/1875 [=====] - 2s 1ms/step - loss: 0.5544 - accuracy: 0.804
3 - val_loss: 0.5855 - val_accuracy: 0.7951
Epoch 7/20
1875/1875 [=====] - 3s 1ms/step - loss: 0.5488 - accuracy: 0.806
3 - val_loss: 0.6097 - val_accuracy: 0.7836
Epoch 8/20
1875/1875 [=====] - 2s 1ms/step - loss: 0.5428 - accuracy: 0.807
7 - val_loss: 0.5787 - val_accuracy: 0.7971
Epoch 9/20
1875/1875 [=====] - 2s 1ms/step - loss: 0.5373 - accuracy: 0.809
7 - val_loss: 0.5698 - val_accuracy: 0.7977
Epoch 10/20
1875/1875 [=====] - 2s 1ms/step - loss: 0.5360 - accuracy: 0.812
4 - val_loss: 0.5658 - val_accuracy: 0.8014
Epoch 11/20
1075/1075 [=====] - 2s 1ms/step - loss: 0.5211 - accuracy: 0.812
```

```

10/10/10/10 -----] - 2s 1ms/step - loss: 0.5511 - accuracy: 0.813
0 - val_loss: 0.5714 - val_accuracy: 0.8002
Epoch 12/20
1875/1875 [=====] - 2s 1ms/step - loss: 0.5284 - accuracy: 0.813
2 - val_loss: 0.5626 - val_accuracy: 0.8027
Epoch 13/20
1875/1875 [=====] - 2s 1ms/step - loss: 0.5271 - accuracy: 0.813
8 - val_loss: 0.5619 - val_accuracy: 0.8041
Epoch 14/20
1875/1875 [=====] - 2s 1ms/step - loss: 0.5249 - accuracy: 0.814
3 - val_loss: 0.5718 - val_accuracy: 0.7991
Epoch 15/20
1875/1875 [=====] - 2s 1ms/step - loss: 0.5231 - accuracy: 0.814
8 - val_loss: 0.5706 - val_accuracy: 0.8024
Epoch 16/20
1875/1875 [=====] - 2s 1ms/step - loss: 0.5203 - accuracy: 0.816
2 - val_loss: 0.5731 - val_accuracy: 0.8023
Epoch 17/20
1875/1875 [=====] - 2s 1ms/step - loss: 0.5191 - accuracy: 0.817
6 - val_loss: 0.5594 - val_accuracy: 0.8030
Epoch 18/20
1875/1875 [=====] - 2s 1ms/step - loss: 0.5176 - accuracy: 0.815
7 - val_loss: 0.5582 - val_accuracy: 0.8053
Epoch 19/20
1875/1875 [=====] - 2s 1ms/step - loss: 0.5156 - accuracy: 0.816
9 - val_loss: 0.5644 - val_accuracy: 0.8007
Epoch 20/20
1875/1875 [=====] - 2s 1ms/step - loss: 0.5146 - accuracy: 0.817
7 - val_loss: 0.5660 - val_accuracy: 0.8075

```

**Now we've got a model trained with a close-to-ideal learning rate and performing pretty well, we've got a couple of options.**

We could:

- Evaluate its performance using other classification metrics (such as a [confusion matrix](#) or [classification report](#)).
- Assess some of its predictions (through visualizations).
- Improve its accuracy (by training it for longer or changing the architecture).
- Save and export it for use in an application.

Let's go through the first two options.

First we'll create a classification matrix to visualize its predictions across the different classes.

In [ ]:

```

# Note: The following confusion matrix code is a remix of Scikit-Learn's
# plot_confusion_matrix function - https://scikit-learn.org/stable/modules/generated/sklearn.metrics.plot_confusion_matrix.html
# and Made with ML's introductory notebook - https://github.com/GokuMohandas/MadeWithML/blob/main/notebooks/08_Neural_Networks.ipynb
import itertools
from sklearn.metrics import confusion_matrix

# Our function needs a different name to sklearn's plot_confusion_matrix
def make_confusion_matrix(y_true, y_pred, classes=None, figsize=(10, 10), text_size=15):

    """Makes a labelled confusion matrix comparing predictions and ground truth labels.

    If classes is passed, confusion matrix will be labelled, if not, integer class values
    will be used.

    Args:
        y_true: Array of truth labels (must be same shape as y_pred).
        y_pred: Array of predicted labels (must be same shape as y_true).
        classes: Array of class labels (e.g. string form). If 'None', integer labels are used
        .
        figsize: Size of output figure (default=(10, 10)).
        text_size: Size of output figure text (default=15).
    """

    if classes is None:
        classes = np.unique(y_true)
    else:
        classes = np.array(classes)

    cm = confusion_matrix(y_true, y_pred)
    cm_norm = cm.astype("float") / cm.sum(axis=1)[:, np.newaxis]
    n_classes = cm.shape[0]

    plt.figure(figsize=figsize)
    sns.heatmap(cm_norm, annot=True, fmt=".2f", cmap="Blues")
    plt.title("Confusion Matrix", size=15)
    plt.ylabel('True Labels', size=13)
    plt.xlabel('Predicted Labels', size=13)
    plt.xticks(classes, size=13)
    plt.yticks(classes, size=13)
    plt.grid(False)
    plt.show()

```

Returns:

A labelled confusion matrix plot comparing `y_true` and `y_pred`.

Example usage:

```
make_confusion_matrix(y_true=test_labels, # ground truth test labels
                      y_pred=y_preds, # predicted labels
                      classes=class_names, # array of class label names
                      figsize=(15, 15),
                      text_size=10)

"""
# Create the confustion matrix
cm = confusion_matrix(y_true, y_pred)
cm_norm = cm.astype("float") / cm.sum(axis=1)[:, np.newaxis] # normalize it
n_classes = cm.shape[0] # find the number of classes we're dealing with

# Plot the figure and make it pretty
fig, ax = plt.subplots(figsize=figsize)
cax = ax.matshow(cm, cmap=plt.cm.Blues) # colors will represent how 'correct' a class
is, darker == better
fig.colorbar(cax)

# Are there a list of classes?
if classes:
    labels = classes
else:
    labels = np.arange(cm.shape[0])

# Label the axes
ax.set(title="Confusion Matrix",
       xlabel="Predicted label",
       ylabel="True label",
       xticks=np.arange(n_classes), # create enough axis slots for each class
       yticks=np.arange(n_classes),
       xticklabels=labels, # axes will labeled with class names (if they exist) or int
       yticklabels=labels)

# Make x-axis labels appear on bottom
ax.xaxis.set_label_position("bottom")
ax.xaxis.tick_bottom()

# Set the threshold for different colors
threshold = (cm.max() + cm.min()) / 2.

# Plot the text on each cell
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, f"{cm[i, j]} ({cm_norm[i, j]*100:.1f}%)",
             horizontalalignment="center",
             color="white" if cm[i, j] > threshold else "black",
             size=text_size)
```

Since a confusion matrix compares the truth labels (`test_labels`) to the predicted labels, we have to make some predictions with our model.

In [ ]:

```
# Make predictions with the most recent model
y_probs = model_14.predict(test_data) # "probs" is short for probabilities

# View the first 5 predictions
y_probs[:5]
```

Out [ ]:

```
array([[8.5630336e-11, 3.5361509e-13, 2.6633865e-05, 4.6356046e-08,
       5.0950021e-05, 9.6119225e-02, 8.1778381e-08, 9.1868617e-02,
       4.0605213e-03, 8.0787390e-01],
      [3.4278683e-06, 1.2899412e-16, 9.5989138e-01, 2.0516255e-07,
       1.5329245e-02, 2.4532243e-13, 2.4142915e-02, 1.1383623e-28,
       6.3271803e-04, 4.4789552e-08],
```

```
[6.10631e-05, 9.9651613e-01, 4.3861061e-08, 3.3405994e-03,
 1.3249499e-05, 1.4383491e-21, 8.2790693e-06, 7.3237471e-18,
 5.4811817e-08, 4.9225428e-14],
[7.5031145e-05, 9.9053687e-01, 4.2528288e-07, 9.2231687e-03,
 1.3623090e-04, 1.8276231e-18, 2.6808115e-05, 4.8124743e-14,
 1.4521548e-06, 2.2211462e-11],
[7.2190031e-02, 1.5495797e-06, 2.5566885e-01, 1.0363121e-02,
 4.3541368e-02, 1.1069260e-13, 6.1693019e-01, 6.7543135e-23,
 1.3049162e-03, 1.2140360e-09]], dtype=float32)
```

**Our model outputs a list of prediction probabilities, meaning, it outputs a number for how likely it thinks a particular class is to be the label.**

**The higher the number in the prediction probabilities list, the more likely the model believes that is the right class.**

To find the highest value we can use the `argmax()` method.

In [ ]:

```
# See the predicted class number and label for the first example
y_probs[0].argmax(), class_names[y_probs[0].argmax()]
```

Out [ ]:

```
(9, 'Ankle boot')
```

Now let's do the same for all of the predictions.

In [ ]:

```
# Convert all of the predictions from probabilities to labels
y_preds = y_probs.argmax(axis=1)

# View the first 10 prediction labels
y_preds[:10]
```

Out [ ]:

```
array([9, 2, 1, 1, 6, 1, 4, 6, 5, 7])
```

**Wonderful, now we've got our model's predictions in label form, let's create a confusion matrix to view them against the truth labels.**

In [ ]:

```
# Check out the non-prettified confusion matrix
from sklearn.metrics import confusion_matrix
confusion_matrix(y_true=test_labels,
                  y_pred=y_preds)
```

Out [ ]:

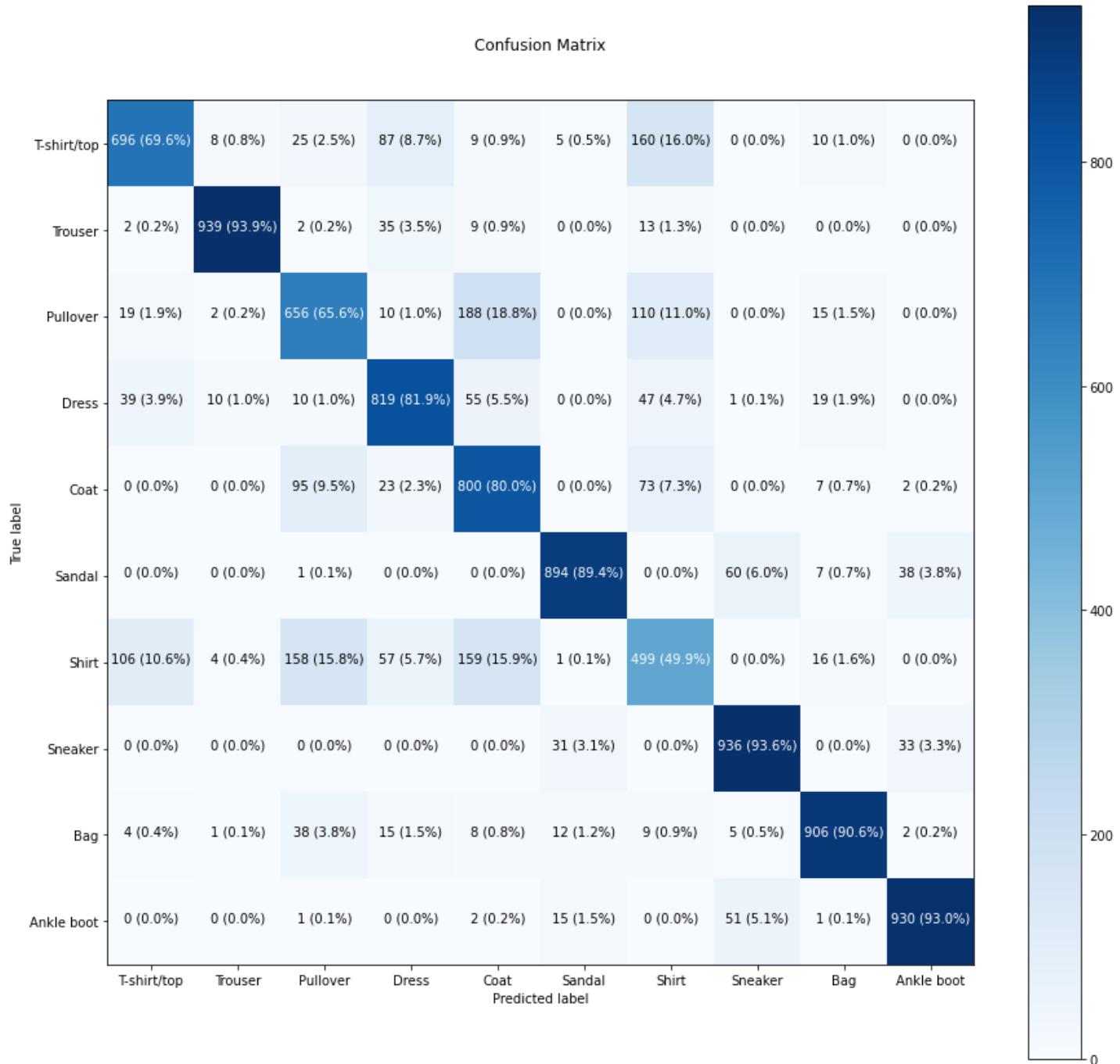
```
array([[696,    8,   25,   87,    9,    5,  160,    0,   10,    0],
       [  2, 939,    2,   35,    9,    0,   13,    0,    0,    0],
       [ 19,    2, 656,   10, 188,    0,  110,    0,   15,    0],
       [ 39,   10,   10, 819,   55,    0,   47,    1,   19,    0],
       [  0,    0,   95,   23, 800,    0,   73,    0,    7,    2],
       [  0,    0,    1,    0,    0, 894,    0,   60,    7,   38],
       [106,    4, 158,   57, 159,    1, 499,    0,   16,    0],
       [  0,    0,    0,    0,   31,    0, 936,    0,   33],
       [  4,    1,   38,   15,    8,   12,    9,    5, 906,    2],
       [  0,    0,    1,    0,   2,   15,    0,   51,    1, 930]])
```

**That confusion matrix is hard to comprehend, let's make it prettier using the function we created before.**

In [ ]:

```
# Make a prettier confusion matrix
```

```
make_confusion_matrix(y_true=test_labels,
                      y_pred=y_preds,
                      classes=class_names,
                      figsize=(15, 15),
                      text_size=10)
```



That looks much better! (one of my favourites sights in the world is a confusion matrix with dark squares down the diagonal)

Except the results aren't as good as they could be...

It looks like our model is getting confused between the Shirt and T-shirt/top classes (e.g. predicting Shirt when it's actually a T-shirt/top).

**Question:** Does it make sense that our model is getting confused between the Shirt and T-shirt/top classes? Why do you think this might be? What's one way you could investigate?

We've seen how our models predictions line up to the truth labels using a confusion matrix, but how about we visualize some?

Let's create a function to plot a random image along with its prediction.

**Note:** Often when working with images and other forms of visual data, it's a good idea to

**visualize as much as possible to develop a further understanding of the data and the outputs of your model.**

In [ ]:

```
import random

# Create a function for plotting a random image along with its prediction
def plot_random_image(model, images, true_labels, classes):
    """Picks a random image, plots it and labels it with a predicted and truth label.

    Args:
        model: a trained model (trained on data similar to what's in images).
        images: a set of random images (in tensor form).
        true_labels: array of ground truth labels for images.
        classes: array of class names for images.

    Returns:
        A plot of a random image from `images` with a predicted class label from `model` as well as the truth class label from `true_labels`.
    """
    # Setup random integer
    i = random.randint(0, len(images))

    # Create predictions and targets
    target_image = images[i]
    pred_probs = model.predict(target_image.reshape(1, 28, 28)) # have to reshape to get into right size for model
    pred_label = classes[pred_probs.argmax()]
    true_label = classes[true_labels[i]]

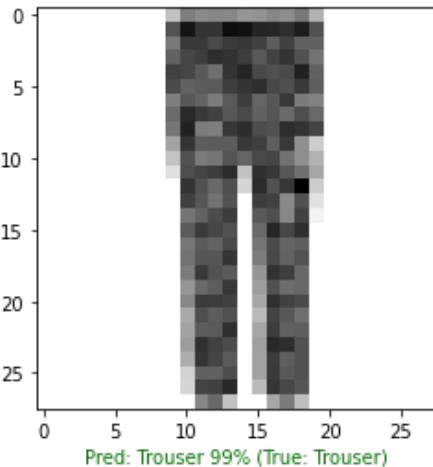
    # Plot the target image
    plt.imshow(target_image, cmap=plt.cm.binary)

    # Change the color of the titles depending on if the prediction is right or wrong
    if pred_label == true_label:
        color = "green"
    else:
        color = "red"

    # Add xlabel information (prediction/true label)
    plt.xlabel("Pred: {} {:.0f}% (True: {})".format(pred_label,
                                                       100*tf.reduce_max(pred_probs),
                                                       true_label),
               color=color) # set the color to green or red
```

In [ ]:

```
# Check out a random image as well as its prediction
plot_random_image(model=model_14,
                  images=test_data,
                  true_labels=test_labels,
                  classes=class_names)
```



After running the cell above a few times you'll start to get a visual understanding of the relationship between the model's predictions and the true labels.

Did you figure out which predictions the model gets confused on?

It seems to mix up classes which are similar, for example, Sneaker with Ankle boot .

Looking at the images, you can see how this might be the case.

The overall shape of a Sneaker and an Ankle Boot are similar.

The overall shape might be one of the patterns the model has learned and so therefore when two images have a similar shape, their predictions get mixed up.

## What patterns is our model learning?

We've been talking a lot about how a neural network finds patterns in numbers, but what exactly do these patterns look like?

Let's crack open one of our models and find out.

First, we'll get a list of layers in our most recent model ( model\_14 ) using the layers attribute.

In [ ]:

```
# Find the layers of our most recent model
model_14.layers
```

Out[ ]:

```
[<tensorflow.python.keras.layers.core.Flatten at 0x7f4254285780>,
 <tensorflow.python.keras.layers.core.Dense at 0x7f42542856a0>,
 <tensorflow.python.keras.layers.core.Dense at 0x7f4254285c50>,
 <tensorflow.python.keras.layers.core.Dense at 0x7f425428ecc0>]
```

We can access a target layer using indexing.

In [ ]:

```
# Extract a particular layer
model_14.layers[1]
```

Out[ ]:

```
<tensorflow.python.keras.layers.core.Dense at 0x7f42542856a0>
```

And we can find the patterns learned by a particular layer using the get\_weights() method.

The get\_weights() method returns the weights (also known as a weights matrix) and biases (also known as a bias vector) of a particular layer.

In [ ]:

```
# Get the patterns of a layer in our network
weights, biases = model_14.layers[1].get_weights()

# Shape = 1 weight matrix the size of our input data (28x28) per neuron (4)
weights, weights.shape
```

Out[ ]:

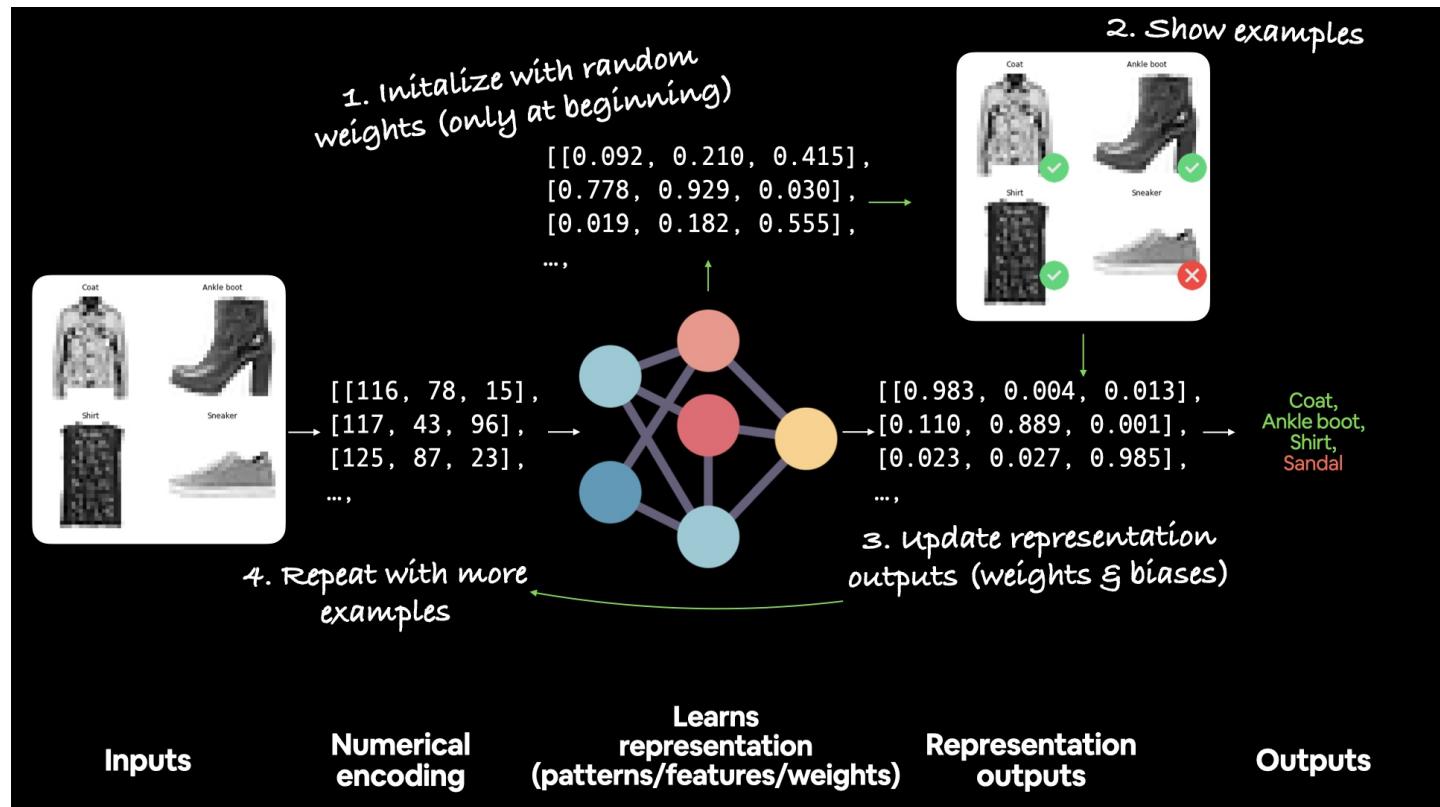
```
(array([[ 0.7150263 , -0.06077094, -0.99763054, -1.048431  ],
       [ 0.27732128, -0.47155392, -0.5291646 ,  0.02329262],
       [ 0.775243 ,  0.540276 , -1.1288569 , -0.7426157 ],
       ...,
       [-0.39453438,  0.47628698, -0.22641574,  0.25505954],
       [-0.4051576 ,  0.6181001 ,  0.23928389, -0.5038765 ],
       [ 0.23884599,  0.11606929, -0.12131333,  0.0435243511,
```

```
[...], [...], [...],  
dtype=float32), (784, 4))
```

The weights matrix is the same shape as the input data, which in our case is 784 (28x28 pixels). And there's a copy of the weights matrix for each neuron in the selected layer (our selected layer has 4 neurons).

Each value in the weights matrix corresponds to how a particular value in the input data influences the network's decisions.

These values start out as random numbers (they're set by the [kernel\\_initializer parameter](#) when creating a layer, the default is ["glorot\\_uniform"](#)) and are then updated to better representative values of the data (non-random) by the neural network during training.



*Example workflow of how a supervised neural network starts with random weights and updates them to better represent the data by looking at examples of ideal outputs.*

Now let's check out the bias vector.

In [ ]:

```
# Shape = 1 bias per neuron (we use 4 neurons in the first layer)  
biases, biases.shape
```

Out[ ]:

```
(array([ 2.4485605e-02, -6.1463297e-04, -2.7230164e-01, 8.1124890e-01],  
      dtype=float32), (4,))
```

Every neuron has a bias vector. Each of these is paired with a weight matrix.

The bias values get initialized as zeroes by default (using the [bias\\_initializer parameter](#)).

The bias vector dictates how much the patterns within the corresponding weights matrix should influence the next layer.

In [ ]:

```
# Can now calculate the number of parameters in our model  
model_14.summary()
```

Model: "sequential\_14"

Layer (type)	Output Shape	Param #
sequential_14	(None, 4)	32

flatten_3 (Flatten)	(None, 784)	0
dense_37 (Dense)	(None, 4)	3140
dense_38 (Dense)	(None, 4)	20
dense_39 (Dense)	(None, 10)	50
=====		
Total params:	3,210	
Trainable params:	3,210	
Non-trainable params:	0	

Now we've built a few deep learning models, it's a good time to point out the whole concept of inputs and outputs not only relates to a model as a whole but to *every layer* within a model.

You might've already guessed this, but starting from the input layer, each subsequent layer's input is the output of the previous layer.

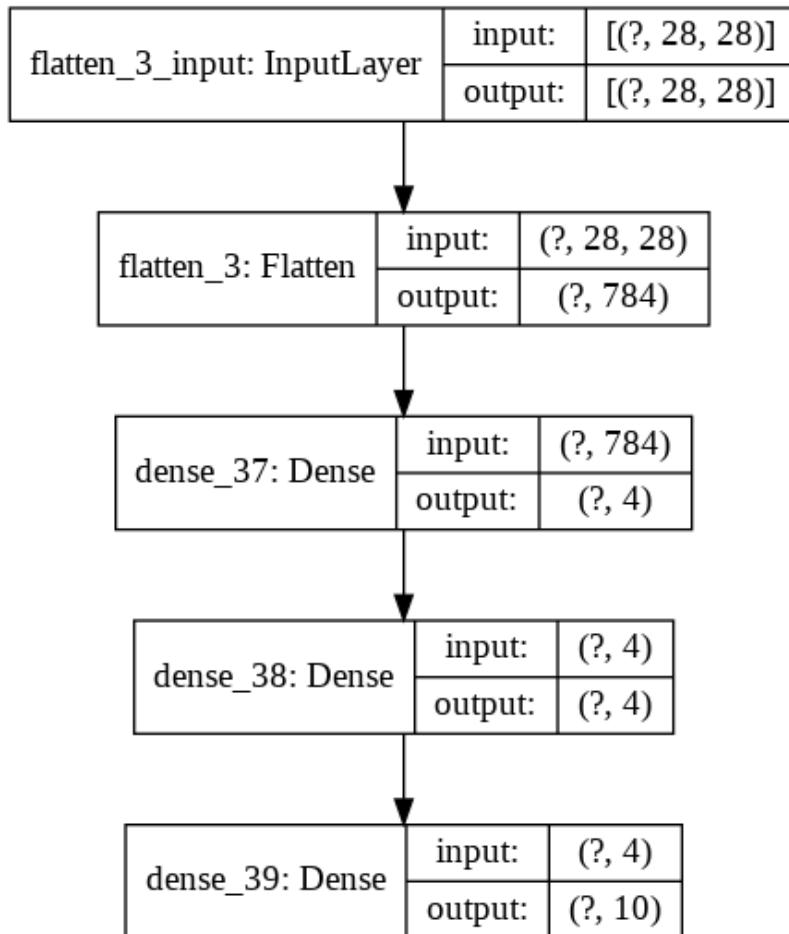
We can see this clearly using the utility `plot_model()`.

In [ ]:

```
from tensorflow.keras.utils import plot_model

# See the inputs and outputs of each layer
plot_model(model_14, show_shapes=True)
```

Out[ ]:



## How a model learns (in brief)

Alright, we've trained a bunch of models, but we've never really discussed what's going on under the hood. So how exactly does a model learn?

A model learns by updating and improving its weight matrices and biases values every epoch (in our case, when we call the `fit()` function).

It does so by comparing the patterns its learned between the data and labels to the actual labels.

If the current patterns (weight matrices and bias values) don't result in a desirable decrease in the loss function (higher loss means worse predictions), the optimizer tries to steer the model to update its patterns in the right way (using the real labels as a reference).

This process of using the real labels as a reference to improve the model's predictions is called [backpropagation](#).

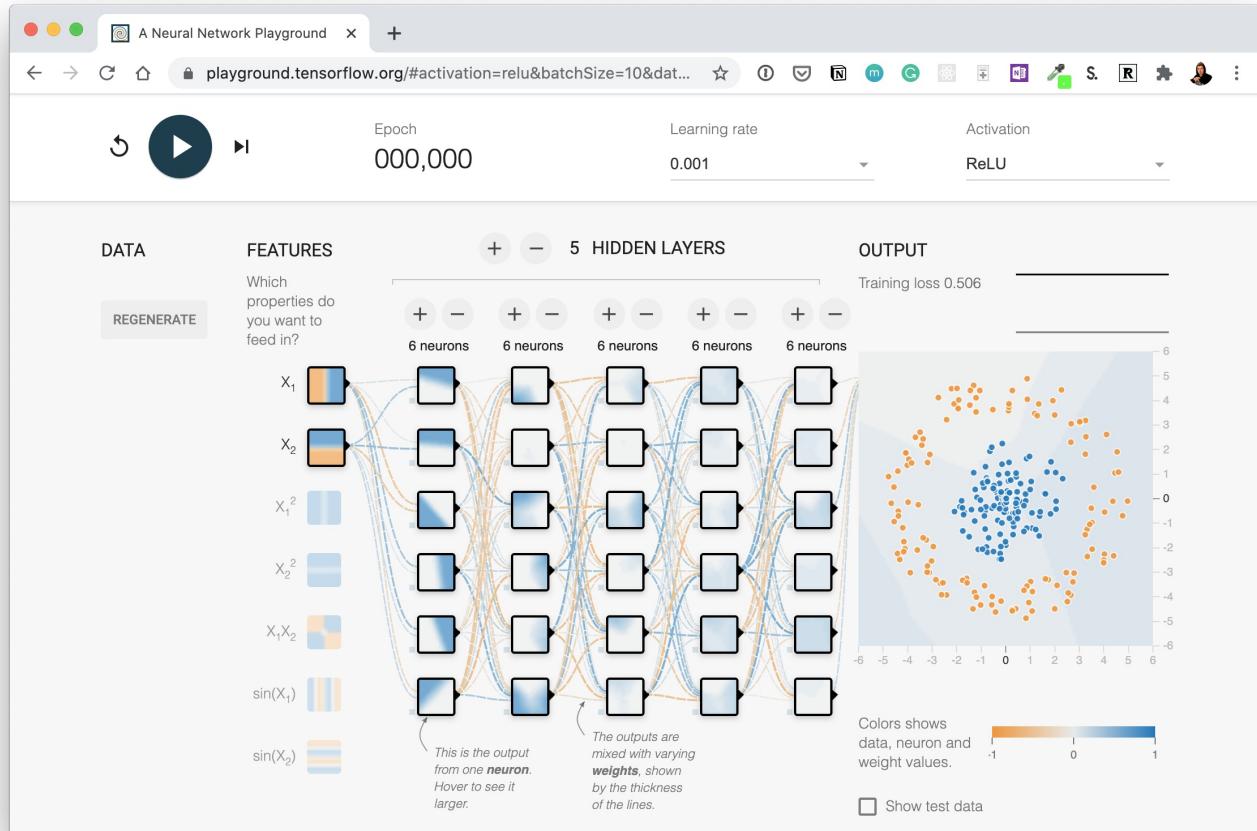
In other words, data and labels pass through a model (**forward pass**) and it attempts to learn the relationship between the data and labels.

And if this learned relationship isn't close to the actual relationship or it could be improved, the model does so by going back through itself (**backward pass**) and tweaking its weights matrices and bias values to better represent the data.

If all of this sounds confusing (and it's fine if it does, the above is a very succinct description), check out the resources in the extra-curriculum section for more.

## Exercises ▾

1. Play with neural networks in the [TensorFlow Playground](#) for 10-minutes. Especially try different values of the learning, what happens when you decrease it? What happens when you increase it?
2. Replicate the model pictured in the [TensorFlow Playground diagram](#) below using TensorFlow code. Compile it using the Adam optimizer, binary crossentropy loss and accuracy metric. Once it's compiled check a summary of the model.



Try this network out for yourself on the [TensorFlow Playground website](#). Hint: there are 5 hidden layers but the output layer isn't pictured, you'll have to decide what the output layer should be based on the input data.

3. Create a classification dataset using Scikit-Learn's `make_moons()` function, visualize it and then build a model to fit it at over 85% accuracy.
4. Create a function (or write code) to visualize multiple image predictions for the fashion MNIST at the same time. Plot at least three different images and their prediction labels at the same time. Hint: see the [classification tutorial in the TensorFlow documentation](#) for ideas.
5. Recreate [TensorFlow's softmax activation function](#) in your own code. Make sure it can accept a tensor and

return that tensor after having the softmax function applied to it.

6. Train a model to get 88%+ accuracy on the fashion MNIST test set. Plot a confusion matrix to see the results after.
7. Make a function to show an image of a certain class of the fashion MNIST dataset and make a prediction on it. For example, plot 3 images of the `T-shirt` class with their predictions.

## Extra curriculum □

- Watch 3Blue1Brown's neural networks video 2: [\*Gradient descent, how neural networks learn\*](#). After you're done, write 100 words about what you've learned.
  - If you haven't already, watch video 1: [\*But what is a Neural Network?\*](#). Note the activation function they talk about at the end.
- Watch [MIT's introduction to deep learning lecture 1](#) (if you haven't already) to get an idea of the concepts behind using linear and non-linear functions.
- Spend 1-hour reading [Michael Nielsen's Neural Networks and Deep Learning book](#).
- Read the [ML-Glossary documentation on activation functions](#). Which one is your favourite?
  - After you've read the ML-Glossary, see which activation functions are available in TensorFlow by searching "tensorflow activation functions".

# 03. Convolutional Neural Networks and Computer Vision with TensorFlow

So far we've covered the basics of TensorFlow and built a handful of models to work across different problems.

Now we're going to get specific and see how a special kind of neural network, [convolutional neural networks \(CNNs\)](#) can be used for computer vision (detecting patterns in visual data).

**Note:** In deep learning, many different kinds of model architectures can be used for different problems. For example, you could use a convolutional neural network for making predictions on image data and/or text data. However, in practice some architectures typically work better than others.

For example, you might want to:

- Classify whether a picture of food contains pizza or steak (we're going to do this)
- Detect whether or not an object appears in an image (e.g. did a specific car pass through a security camera?)

In this notebook, we're going to follow the TensorFlow modelling workflow we've been following so far whilst learning about how to build and use CNNs.

## What we're going to cover

Specifically, we're going to go through the following with TensorFlow:

- Getting a dataset to work with
- Architecture of a convolutional neural network
- A quick end-to-end example (what we're working towards)
- Steps in modelling for binary image classification with CNNs
  - Becoming one with the data
  - Preparing data for modelling
  - Creating a CNN model (starting with a baseline)
  - Fitting a model (getting it to find patterns in our data)
  - Evaluating a model
  - Improving a model
  - Making a prediction with a trained model
- Steps in modelling for multi-class image classification with CNNs
  - Same as above (but this time with a different dataset)

## How you can use this notebook

You can read through the descriptions and the code (it should all run, except for the cells which error on purpose), but there's a better option.

Write all of the code yourself.

Yes. I'm serious. Create a new notebook, and rewrite each line by yourself. Investigate it, see if you can break it, why does it break?

You don't have to write the text descriptions but writing the code yourself is a great way to get hands-on experience.

Don't worry if you make mistakes, we all do. The way to get better and make less mistakes is to [write more code](#).

# Get the data

Because convolutional neural networks work so well with images, to learn more about them, we're going to start with a dataset of images.

The images we're going to work with are from the [Food-101 dataset](#), a collection of 101 different categories of 101,000 (1000 images per category) real-world images of food dishes.

To begin, we're only going to use two of the categories, pizza  and steak  and build a binary classifier.

 **Note:** To prepare the data we're using, preprocessing steps such as, moving the images into different subset folders, have been done. To see these preprocessing steps check out [the preprocessing notebook](#).

We'll download the `pizza_steak` subset .zip file and unzip it.

In [1]:

```
import zipfile

# Download zip file of pizza_steak images
!wget https://storage.googleapis.com/ztm_tf_course/food_vision/pizza_steak.zip

# Unzip the downloaded file
zip_ref = zipfile.ZipFile("pizza_steak.zip", "r")
zip_ref.extractall()
zip_ref.close()

--2021-07-14 05:59:42-- https://storage.googleapis.com/ztm_tf_course/food_vision/pizza_s
teak.zip
Resolving storage.googleapis.com (storage.googleapis.com)... 142.251.33.208, 142.250.81.2
08, 172.217.13.240, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|142.251.33.208|:443... conn
ected.
HTTP request sent, awaiting response... 200 OK
Length: 109579078 (105M) [application/zip]
Saving to: 'pizza_steak.zip'

pizza_steak.zip      100%[=====] 104.50M   293MB/s    in 0.4s

2021-07-14 05:59:43 (293 MB/s) - 'pizza_steak.zip' saved [109579078/109579078]
```

 **Note:** If you're using Google Colab and your runtime disconnects, you may have to redownload the files. You can do this by rerunning the cell above.

## Inspect the data (become one with it)

A very crucial step at the beginning of any machine learning project is becoming one with the data. This usually means plenty of visualizing and folder scanning to understand the data you're working with.

With this being said, let's inspect the data we just downloaded.

The file structure has been formatted to be in a typical format you might use for working with images.

More specifically:

- A `train` directory which contains all of the images in the training dataset with subdirectories each named after a certain class containing images of that class.
- A `test` directory with the same structure as the `train` directory.

Example of file structure

```
pizza_steak <- top level folder
└── train <- training images
    ├── pizza
    │   ├── 1008104.jpg
    │   ├── 1638227.jpg
    │   ...
    └── steak
        ├── 1000205.jpg
        ├── 1647351.jpg
        ...
└── test <- testing images
    ├── pizza
    │   ├── 1001116.jpg
    │   ├── 1507019.jpg
    │   ...
    └── steak
        ├── 100274.jpg
        ├── 1653815.jpg
        ...
```

**Let's inspect each of the directories we've downloaded.**

**To so do, we can use the command `ls` which stands for list.**

In [2]:

```
!ls pizza_steak
```

test train

**We can see we've got a `train` and `test` folder.**

**Let's see what's inside one of them.**

In [3]:

```
!ls pizza_steak/train/
```

pizza steak

**And how about inside the `steak` directory?**

In [4]:

```
!ls pizza_steak/train/steak/
```

1000205.jpg	1647351.jpg	2238681.jpg	2824680.jpg	3375959.jpg	417368.jpg
100135.jpg	1650002.jpg	2238802.jpg	2825100.jpg	3381560.jpg	4176.jpg
101312.jpg	165639.jpg	2254705.jpg	2826987.jpg	3382936.jpg	42125.jpg
1021458.jpg	1658186.jpg	225990.jpg	2832499.jpg	3386119.jpg	421476.jpg
1032846.jpg	1658443.jpg	2260231.jpg	2832960.jpg	3388717.jpg	421561.jpg
10380.jpg	165964.jpg	2268692.jpg	285045.jpg	3389138.jpg	438871.jpg
1049459.jpg	167069.jpg	2271133.jpg	285147.jpg	3393547.jpg	43924.jpg
1053665.jpg	1675632.jpg	227576.jpg	2855315.jpg	3393688.jpg	440188.jpg
1068516.jpg	1678108.jpg	2283057.jpg	2856066.jpg	3396589.jpg	442757.jpg
1068975.jpg	168006.jpg	2286639.jpg	2859933.jpg	339891.jpg	443210.jpg
1081258.jpg	1682496.jpg	2287136.jpg	286219.jpg	3417789.jpg	444064.jpg
1090122.jpg	1684438.jpg	2291292.jpg	2862562.jpg	3425047.jpg	444709.jpg
1093966.jpg	168775.jpg	229323.jpg	2865730.jpg	3434983.jpg	447557.jpg
1098844.jpg	1697339.jpg	2300534.jpg	2878151.jpg	3435358.jpg	461187.jpg
1100074.jpg	1710569.jpg	2300845.jpg	2880035.jpg	3438319.jpg	461689.jpg
1105280.jpg	1714605.jpg	231296.jpg	2881783.jpg	3444407.jpg	465494.jpg
1117936.jpg	1724387.jpg	2315295.jpg	2884233.jpg	345734.jpg	468384.jpg
1126126.jpg	1724717.jpg	2323132.jpg	2890573.jpg	3460673.jpg	477486.jpg
114601.jpg	172936.jpg	2324994.jpg	2893832.jpg	3465327.jpg	482022.jpg

1147047.jpg	1736543.jpg	2327701.jpg	2893892.jpg	3466159.jpg	482465.jpg
1147883.jpg	1736968.jpg	2331076.jpg	2907177.jpg	3469024.jpg	483788.jpg
1155665.jpg	1746626.jpg	233964.jpg	290850.jpg	3470083.jpg	493029.jpg
1163977.jpg	1752330.jpg	2344227.jpg	2909031.jpg	3476564.jpg	503589.jpg
1190233.jpg	1761285.jpg	234626.jpg	2910418.jpg	3478318.jpg	510757.jpg
1208405.jpg	176508.jpg	234704.jpg	2912290.jpg	3488748.jpg	513129.jpg
1209120.jpg	1772039.jpg	2357281.jpg	2916448.jpg	3492328.jpg	513842.jpg
1212161.jpg	1777107.jpg	2361812.jpg	2916967.jpg	3518960.jpg	523535.jpg
1213988.jpg	1787505.jpg	2365287.jpg	2927833.jpg	3522209.jpg	525041.jpg
1219039.jpg	179293.jpg	2374582.jpg	2928643.jpg	3524429.jpg	534560.jpg
1225762.jpg	1816235.jpg	239025.jpg	2929179.jpg	3528458.jpg	534633.jpg
1230968.jpg	1822407.jpg	2390628.jpg	2936477.jpg	3531805.jpg	536535.jpg
1236155.jpg	1823263.jpg	2392910.jpg	2938012.jpg	3536023.jpg	541410.jpg
1241193.jpg	1826066.jpg	2394465.jpg	2938151.jpg	3538682.jpg	543691.jpg
1248337.jpg	1828502.jpg	2395127.jpg	2939678.jpg	3540750.jpg	560503.jpg
1257104.jpg	1828969.jpg	2396291.jpg	2940544.jpg	354329.jpg	561972.jpg
126345.jpg	1829045.jpg	2400975.jpg	2940621.jpg	3547166.jpg	56240.jpg
1264050.jpg	1829088.jpg	2403776.jpg	2949079.jpg	3553911.jpg	56409.jpg
1264154.jpg	1836332.jpg	2403907.jpg	295491.jpg	3556871.jpg	564530.jpg
1264858.jpg	1839025.jpg	240435.jpg	296268.jpg	355715.jpg	568972.jpg
127029.jpg	1839481.jpg	2404695.jpg	2964732.jpg	356234.jpg	576725.jpg
1289900.jpg	183995.jpg	2404884.jpg	2965021.jpg	3571963.jpg	588739.jpg
1290362.jpg	184110.jpg	2407770.jpg	2966859.jpg	3576078.jpg	590142.jpg
1295457.jpg	184226.jpg	2412263.jpg	2977966.jpg	3577618.jpg	60633.jpg
1312841.jpg	1846706.jpg	2425062.jpg	2979061.jpg	3577732.jpg	60655.jpg
1313316.jpg	1849364.jpg	2425389.jpg	2983260.jpg	3578934.jpg	606820.jpg
1324791.jpg	1849463.jpg	2435316.jpg	2984311.jpg	358042.jpg	612551.jpg
1327567.jpg	1849542.jpg	2437268.jpg	2988960.jpg	358045.jpg	614975.jpg
1327667.jpg	1853564.jpg	2437843.jpg	2989882.jpg	3591821.jpg	616809.jpg
1333055.jpg	1869467.jpg	2440131.jpg	2995169.jpg	359330.jpg	628628.jpg
1334054.jpg	1870942.jpg	2443168.jpg	2996324.jpg	3601483.jpg	632427.jpg
1335556.jpg	187303.jpg	2446660.jpg	3000131.jpg	3606642.jpg	636594.jpg
1337814.jpg	187521.jpg	2455944.jpg	3002350.jpg	3609394.jpg	637374.jpg
1340977.jpg	1888450.jpg	2458401.jpg	3007772.jpg	361067.jpg	640539.jpg
1343209.jpg	1889336.jpg	2487306.jpg	3008192.jpg	3613455.jpg	644777.jpg
134369.jpg	1907039.jpg	248841.jpg	3009617.jpg	3621464.jpg	644867.jpg
1344105.jpg	1925230.jpg	2489716.jpg	3011642.jpg	3621562.jpg	658189.jpg
134598.jpg	1927984.jpg	2490489.jpg	3020591.jpg	3621565.jpg	660900.jpg
1346387.jpg	1930577.jpg	2495884.jpg	3030578.jpg	3623556.jpg	663014.jpg
1348047.jpg	1937872.jpg	2495903.jpg	3047807.jpg	3640915.jpg	664545.jpg
1351372.jpg	1941807.jpg	2499364.jpg	3059843.jpg	3643951.jpg	667075.jpg
1362989.jpg	1942333.jpg	2500292.jpg	3074367.jpg	3653129.jpg	669180.jpg
1367035.jpg	1945132.jpg	2509017.jpg	3082120.jpg	3656752.jpg	669960.jpg
1371177.jpg	1961025.jpg	250978.jpg	3094354.jpg	3663518.jpg	6709.jpg
1375640.jpg	1966300.jpg	2514432.jpg	3095301.jpg	3663800.jpg	674001.jpg
1382427.jpg	1966967.jpg	2526838.jpg	3099645.jpg	3664376.jpg	676189.jpg
1392718.jpg	1969596.jpg	252858.jpg	3100476.jpg	3670607.jpg	681609.jpg
1395906.jpg	1971757.jpg	2532239.jpg	3110387.jpg	3671021.jpg	6926.jpg
1400760.jpg	1976160.jpg	2534567.jpg	3113772.jpg	3671877.jpg	703556.jpg
1403005.jpg	1984271.jpg	2535431.jpg	3116018.jpg	368073.jpg	703909.jpg
1404770.jpg	1987213.jpg	2535456.jpg	3128952.jpg	368162.jpg	704316.jpg
140832.jpg	1987639.jpg	2538000.jpg	3130412.jpg	368170.jpg	714298.jpg
141056.jpg	1995118.jpg	2543081.jpg	3136.jpg	3693649.jpg	720060.jpg
141135.jpg	1995252.jpg	2544643.jpg	313851.jpg	3700079.jpg	726083.jpg
1413972.jpg	199754.jpg	2547797.jpg	3140083.jpg	3704103.jpg	728020.jpg
1421393.jpg	2002400.jpg	2548974.jpg	3140147.jpg	3707493.jpg	732986.jpg
1428947.jpg	2011264.jpg	2549316.jpg	3142045.jpg	3716881.jpg	734445.jpg
1433912.jpg	2012996.jpg	2561199.jpg	3142618.jpg	3724677.jpg	735441.jpg
143490.jpg	2013535.jpg	2563233.jpg	3142674.jpg	3727036.jpg	740090.jpg
1445352.jpg	2017387.jpg	256592.jpg	3143192.jpg	3727491.jpg	745189.jpg
1446401.jpg	2018173.jpg	2568848.jpg	314359.jpg	3736065.jpg	752203.jpg
1453991.jpg	2020613.jpg	2573392.jpg	3157832.jpg	37384.jpg	75537.jpg
1456841.jpg	2032669.jpg	2592401.jpg	3159818.jpg	3743286.jpg	756655.jpg
146833.jpg	203450.jpg	2599817.jpg	3162376.jpg	3745515.jpg	762210.jpg
1476404.jpg	2034628.jpg	2603058.jpg	3168620.jpg	3750472.jpg	763690.jpg
1485083.jpg	2036920.jpg	2606444.jpg	3171085.jpg	3752362.jpg	767442.jpg
1487113.jpg	2038418.jpg	2614189.jpg	317206.jpg	3766099.jpg	786409.jpg
148916.jpg	2042975.jpg	2614649.jpg	3173444.jpg	3770370.jpg	80215.jpg
149087.jpg	2045647.jpg	2615718.jpg	3180182.jpg	377190.jpg	802348.jpg
1493169.jpg	2050584.jpg	2619625.jpg	31881.jpg	3777020.jpg	804684.jpg
149682.jpg	2052542.jpg	2622140.jpg	3191589.jpg	3777482.jpg	812163.jpg
1508094.jpg	2056627.jpg	262321.jpg	3204977.jpg	3781152.jpg	813486.jpg

1512226.jpg	2062248.jpg	2625330.jpg	320658.jpg	3787809.jpg	819027.jpg
1512347.jpg	2081995.jpg	2628106.jpg	3209173.jpg	3788729.jpg	822550.jpg
1524526.jpg	2087958.jpg	2629750.jpg	3223400.jpg	3790962.jpg	823766.jpg
1530833.jpg	2088030.jpg	2643906.jpg	3223601.jpg	3792514.jpg	827764.jpg
1539499.jpg	2088195.jpg	2644457.jpg	3241894.jpg	379737.jpg	830007.jpg
1541672.jpg	2090493.jpg	2648423.jpg	3245533.jpg	3807440.jpg	838344.jpg
1548239.jpg	2090504.jpg	2651300.jpg	3245622.jpg	381162.jpg	853327.jpg
1550997.jpg	2125877.jpg	2653594.jpg	3247009.jpg	3812039.jpg	854150.jpg
1552530.jpg	2129685.jpg	2661577.jpg	3253588.jpg	3829392.jpg	864997.jpg
15580.jpg	2133717.jpg	2668916.jpg	3260624.jpg	3830872.jpg	885571.jpg
1559052.jpg	2136662.jpg	268444.jpg	326587.jpg	38442.jpg	907107.jpg
1563266.jpg	213765.jpg	2691461.jpg	32693.jpg	3855584.jpg	908261.jpg
1567554.jpg	2138335.jpg	2706403.jpg	3271253.jpg	3857508.jpg	910672.jpg
1575322.jpg	2140776.jpg	270687.jpg	3274423.jpg	386335.jpg	911803.jpg
1588879.jpg	214320.jpg	2707522.jpg	3280453.jpg	3867460.jpg	91432.jpg
1594719.jpg	2146963.jpg	2711806.jpg	3298495.jpg	3868959.jpg	914570.jpg
1595869.jpg	215222.jpg	2716993.jpg	330182.jpg	3869679.jpg	922752.jpg
1598345.jpg	2154126.jpg	2724554.jpg	3306627.jpg	388776.jpg	923772.jpg
1598885.jpg	2154779.jpg	2738227.jpg	3315727.jpg	3890465.jpg	926414.jpg
1600179.jpg	2159975.jpg	2748917.jpg	331860.jpg	3894222.jpg	931356.jpg
1600794.jpg	2163079.jpg	2760475.jpg	332232.jpg	3895825.jpg	937133.jpg
160552.jpg	217250.jpg	2761427.jpg	3322909.jpg	389739.jpg	945791.jpg
1606596.jpg	2172600.jpg	2765887.jpg	332557.jpg	3916407.jpg	947877.jpg
1615395.jpg	2173084.jpg	2768451.jpg	3326734.jpg	393349.jpg	952407.jpg
1618011.jpg	217996.jpg	2771149.jpg	3330642.jpg	393494.jpg	952437.jpg
1619357.jpg	2193684.jpg	2779040.jpg	3333128.jpg	398288.jpg	955466.jpg
1621763.jpg	220341.jpg	2788312.jpg	3333735.jpg	40094.jpg	9555.jpg
1623325.jpg	22080.jpg	2788759.jpg	3334973.jpg	401094.jpg	961341.jpg
1624450.jpg	2216146.jpg	2796102.jpg	3335013.jpg	401144.jpg	97656.jpg
1624747.jpg	2222018.jpg	280284.jpg	3335267.jpg	401651.jpg	979110.jpg
1628861.jpg	2223787.jpg	2807888.jpg	3346787.jpg	405173.jpg	980247.jpg
1632774.jpg	2230959.jpg	2815172.jpg	3364420.jpg	405794.jpg	982988.jpg
1636831.jpg	2232310.jpg	2818805.jpg	336637.jpg	40762.jpg	987732.jpg
1645470.jpg	2233395.jpg	2823872.jpg	3372616.jpg	413325.jpg	996684.jpg

**Woah, a whole bunch of images. But how many?**

**Practice:** Try listing the same information for the `pizza` directory in the `test` folder.

In [5]:

```
import os

# Walk through pizza_steak directory and list number of files
for dirpath, dirnames, filenames in os.walk("pizza_steak"):
    print(f"There are {len(dirnames)} directories and {len(filenames)} images in '{dirpath}'")
```

There are 2 directories and 1 images in 'pizza\_steak'.  
 There are 2 directories and 1 images in 'pizza\_steak/test'.  
 There are 0 directories and 250 images in 'pizza\_steak/test/pizza'.  
 There are 0 directories and 250 images in 'pizza\_steak/test/steak'.  
 There are 2 directories and 1 images in 'pizza\_steak/train'.  
 There are 0 directories and 750 images in 'pizza\_steak/train/pizza'.  
 There are 0 directories and 750 images in 'pizza\_steak/train/steak'.

In [6]:

```
# Another way to find out how many images are in a file
num_steak_images_train = len(os.listdir("pizza_steak/train/steak"))

num_steak_images_train
```

Out[6]:

750

In [7]:

# Get the class names (programmatically - this is much more helpful with a longer list of classes)

```
# Get the class names (programmatically, this is much more helpful with a longer list of classes)
import pathlib
import numpy as np
data_dir = pathlib.Path("pizza_steak/train/") # turn our training path into a Python path
class_names = np.array(sorted([item.name for item in data_dir.glob('*')]))) # created a list of class_names from the subdirectories
print(class_names)
```

```
['.DS_Store' 'pizza' 'steak']
```

Okay, so we've got a collection of 750 training images and 250 testing images of pizza and steak.

Let's look at some.

**Note:** Whenever you're working with data, it's always good to visualize it as much as possible. Treat your first couple of steps of a project as becoming one with the data. **Visualize, visualize, visualize.**

In [8]:

```
# View an image
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import random

def view_random_image(target_dir, target_class):
    # Setup target directory (we'll view images from here)
    target_folder = target_dir+target_class

    # Get a random image path
    random_image = random.sample(os.listdir(target_folder), 1)

    # Read in the image and plot it using matplotlib
    img = mpimg.imread(target_folder + "/" + random_image[0])
    plt.imshow(img)
    plt.title(target_class)
    plt.axis("off");

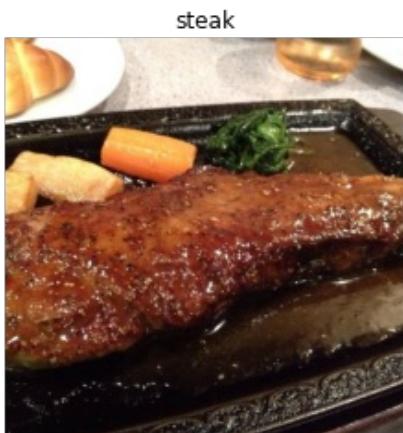
    print(f"Image shape: {img.shape}") # show the shape of the image

    return img
```

In [9]:

```
# View a random image from the training dataset
img = view_random_image(target_dir="pizza_steak/train/",
                        target_class="steak")
```

Image shape: (512, 512, 3)



After going through a dozen or so images from the different classes, you can start to get an idea of what we're working with.

The entire Food101 dataset comprises of similar images from 101 different classes.

You might've noticed we've been printing the image shape alongside the plotted image.

This is because the way our computer sees the image is in the form of a big array (tensor).

In [10]:

```
# View the img (actually just a big array/tensor)
img
```

Out[10]:

```
array([[[240, 150, 72],
       [232, 142, 66],
       [225, 132, 62],
       ...,
       [255, 255, 255],
       [255, 255, 255],
       [255, 255, 255]],

      [[244, 157, 78],
       [241, 151, 75],
       [235, 143, 70],
       ...,
       [255, 255, 255],
       [255, 255, 255],
       [255, 255, 255]],

      [[249, 162, 82],
       [248, 161, 82],
       [245, 154, 81],
       ...,
       [255, 255, 255],
       [255, 255, 255],
       [255, 255, 255]],

      ...,

      [[ 11,    6,   10],
       [  7,    2,    6],
       [  8,    3,    7],
       ...,
       [ 97,   88,   73],
       [ 96,   87,   72],
       [ 92,   83,   68]],

      [[ 11,    6,   10],
       [ 10,    5,    9],
       [  8,    3,    7],
       ...,
       [ 99,   90,   75],
       [ 97,   88,   73],
       [ 87,   78,   61]],

      [[  7,    2,    6],
       [ 11,    6,   10],
       [ 10,    5,    9],
       ...,
       [105,   96,   81],
       [101,   92,   75],
       [ 85,   76,   59]]], dtype=uint8)
```

In [11]:

```
# View the image shape
img.shape # returns (width, height, colour channels)
```

Out[11]:

```
(512, 512, 3)
```

Looking at the image shape more closely, you'll see it's in the form (Width, Height, Colour Channels).

In our case, the width and height vary but because we're dealing with colour images, the colour channels value is always 3. This is for different values of red, green and blue (RGB) pixels.

You'll notice all of the values in the `img` array are between 0 and 255. This is because that's the possible range for red, green and blue values.

For example, a pixel with a value `red=0, green=0, blue=255` will look very blue.

So when we build a model to differentiate between our images of `pizza` and `steak`, it will be finding patterns in these different pixel values which determine what each class looks like.

**Note:** As we've discussed before, many machine learning models, including neural networks prefer the values they work with to be between 0 and 1. Knowing this, one of the most common preprocessing steps for working with images is to scale (also referred to as normalize) their pixel values by dividing the image arrays by 255.

In [12]:

```
# Get all the pixel values between 0 & 1
img/255.
```

Out[12]:

```
array([[ [0.94117647, 0.58823529, 0.28235294],
       [0.90980392, 0.55686275, 0.25882353],
       [0.88235294, 0.51764706, 0.24313725],
       ...,
       [1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]],

      [[0.95686275, 0.61568627, 0.30588235],
       [0.94509804, 0.59215686, 0.29411765],
       [0.92156863, 0.56078431, 0.2745098],
       ...,
       [1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]],

      [[0.97647059, 0.63529412, 0.32156863],
       [0.97254902, 0.63137255, 0.32156863],
       [0.96078431, 0.60392157, 0.31764706],
       ...,
       [1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]],

      ...,

      [[0.04313725, 0.02352941, 0.03921569],
       [0.02745098, 0.00784314, 0.02352941],
       [0.03137255, 0.01176471, 0.02745098],
       ...,
       [0.38039216, 0.34509804, 0.28627451],
       [0.37647059, 0.34117647, 0.28235294],
       [0.36078431, 0.3254902, 0.26666667]],

      [[0.04313725, 0.02352941, 0.03921569],
       [0.03921569, 0.01960784, 0.03529412],
       [0.03137255, 0.01176471, 0.02745098],
       ...,
       [0.38823529, 0.35294118, 0.29411765],
       [0.38039216, 0.34509804, 0.28627451],
       [0.34117647, 0.30588235, 0.23921569]],

      [[0.02745098, 0.00784314, 0.02352941],
```

```
[0.04313725, 0.02352941, 0.03921569],  
[0.03921569, 0.01960784, 0.03529412],  
...  
[0.41176471, 0.37647059, 0.31764706],  
[0.39607843, 0.36078431, 0.29411765],  
[0.33333333, 0.29803922, 0.23137255]]])
```

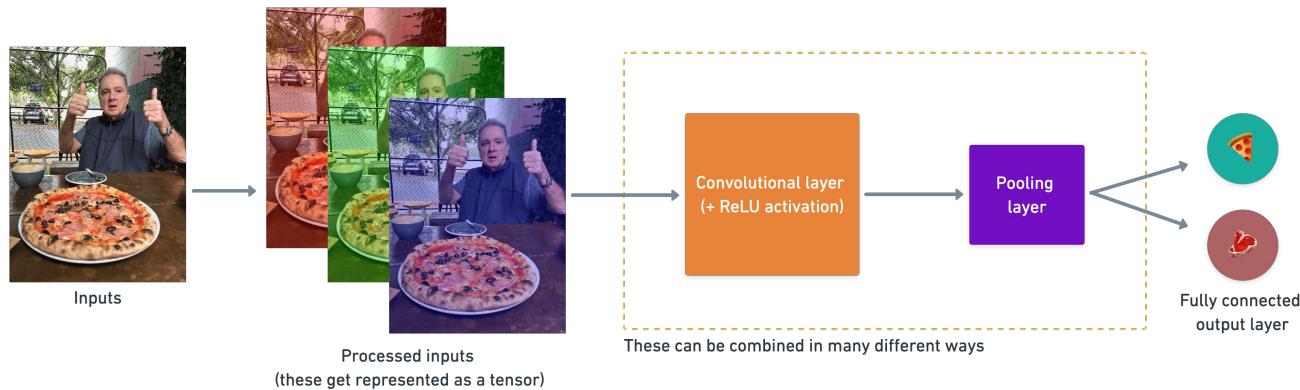
## A (typical) architecture of a convolutional neural network

Convolutional neural networks are no different to other kinds of deep learning neural networks in the fact they can be created in many different ways. What you see below are some components you'd expect to find in a traditional CNN.

Components of a convolutional neural network:

Hyperparameter/Layer type	What does it do?	Typical values
Input image(s)	Target images you'd like to discover patterns in	Whatever you can take a photo (or video) of
Input layer	Takes in target images and preprocesses them for further layers	<code>input_shape = [batch_size, image_height, image_width, color_channels]</code>
Convolution layer	Extracts/learns the most important features from target images	Multiple, can create with <code>tf.keras.layers.Conv2D</code> (X can be multiple values)
Hidden activation	Adds non-linearity to learned features (non-straight lines)	Usually ReLU ( <code>tf.keras.activations.relu</code> )
Pooling layer	Reduces the dimensionality of learned image features	Average ( <code>tf.keras.layers.AvgPool2D</code> ) or Max ( <code>tf.keras.layers.MaxPool2D</code> )
Fully connected layer	Further refines learned features from convolution layers	<code>tf.keras.layers.Dense</code>
Output layer	Takes learned features and outputs them in shape of target labels	<code>output_shape = [number_of_classes]</code> (e.g. 3 for pizza, steak or sushi)
Output activation	Adds non-linearities to output layer	<code>tf.keras.activations.sigmoid</code> (binary classification) or <code>tf.keras.activations.softmax</code>

How they stack together:



A simple example of how you might stack together the above layers into a convolutional neural network. Note the convolutional and pooling layers can often be arranged and rearranged into many different formations.

## An end-to-end example

We've checked out our data and found there's 750 training images, as well as 250 test images per class and they're all of various different shapes.

It's time to jump straight in the deep end.

Reading the [original dataset authors paper](#), we see they used a [Random Forest machine learning model](#) and averaged 50.76% accuracy at predicting what different foods different images had in them.

From now on, that 50.76% will be our baseline.

□ Note: A baseline is a score or evaluation metric you want to try and beat. Usually you'll start with a simple model, create a baseline and try to beat it by increasing the complexity of the model. A really fun way to learn machine learning is to find some kind of modelling paper with a published result and try to beat it.

The code in the following cell replicates an end-to-end way to model our `pizza_steak` dataset with a convolutional neural network (CNN) using the components listed above.

There will be a bunch of things you might not recognize but step through the code yourself and see if you can figure out what it's doing.

We'll go through each of the steps later on in the notebook.

For reference, the model we're using replicates TinyVGG, the computer vision architecture which fuels the [CNN explainer webpage](#).

□ Resource: The architecture we're using below is a scaled-down version of [VGG-16](#), a convolutional neural network which came 2nd in the 2014 [ImageNet classification competition](#).

In [13]:

```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Set the seed
tf.random.set_seed(42)

# Preprocess data (get all of the pixel values between 1 and 0, also called scaling/normalization)
train_datagen = ImageDataGenerator(rescale=1./255)
valid_datagen = ImageDataGenerator(rescale=1./255)

# Setup the train and test directories
train_dir = "pizza_steak/train/"
test_dir = "pizza_steak/test/"

# Import data from directories and turn it into batches
train_data = train_datagen.flow_from_directory(train_dir,
                                                batch_size=32, # number of images to process at a time
                                                target_size=(224, 224), # convert all images to be 224 x 224
                                                class_mode="binary", # type of problem we're working on
                                                seed=42)

valid_data = valid_datagen.flow_from_directory(test_dir,
                                                batch_size=32,
                                                target_size=(224, 224),
                                                class_mode="binary",
                                                seed=42)

# Create a CNN model (same as Tiny VGG - https://poloclub.github.io/cnn-explainer/)
model_1 = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(filters=10,
                          kernel_size=3, # can also be (3, 3)
                          activation="relu",
                          input_shape=(224, 224, 3)), # first layer specifies input shape (height, width, colour channels)
    tf.keras.layers.Conv2D(10, 3, activation="relu"),
    tf.keras.layers.MaxPool2D(pool_size=2, # pool_size can also be (2, 2)
                            padding="valid"), # padding can also be 'same'
    tf.keras.layers.Conv2D(10, 3, activation="relu"),
    tf.keras.layers.Conv2D(10, 3, activation="relu"), # activation='relu' == tf.keras.laye
```

```

rs.Activations(tf.nn.relu)
    tf.keras.layers.MaxPool2D(2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(1, activation="sigmoid") # binary activation output
])

# Compile the model
model_1.compile(loss="binary_crossentropy",
                 optimizer=tf.keras.optimizers.Adam(),
                 metrics=["accuracy"])

# Fit the model
history_1 = model_1.fit(train_data,
                        epochs=5,
                        steps_per_epoch=len(train_data),
                        validation_data=valid_data,
                        validation_steps=len(valid_data))

```

Found 1500 images belonging to 2 classes.  
 Found 500 images belonging to 2 classes.  
 Epoch 1/5  
 47/47 [=====] - 44s 224ms/step - loss: 0.5416 - accuracy: 0.7187  
 - val\_loss: 0.3886 - val\_accuracy: 0.8520  
 Epoch 2/5  
 47/47 [=====] - 9s 185ms/step - loss: 0.4008 - accuracy: 0.8200  
 - val\_loss: 0.3349 - val\_accuracy: 0.8600  
 Epoch 3/5  
 47/47 [=====] - 9s 188ms/step - loss: 0.3849 - accuracy: 0.8307  
 - val\_loss: 0.2867 - val\_accuracy: 0.8840  
 Epoch 4/5  
 47/47 [=====] - 9s 184ms/step - loss: 0.3422 - accuracy: 0.8633  
 - val\_loss: 0.2914 - val\_accuracy: 0.8780  
 Epoch 5/5  
 47/47 [=====] - 9s 185ms/step - loss: 0.3113 - accuracy: 0.8740  
 - val\_loss: 0.3029 - val\_accuracy: 0.8840

**Note:** If the cell above takes more than ~12 seconds per epoch to run, you might not be using a GPU accelerator. If you're using a Colab notebook, you can access a GPU accelerator by going to Runtime -> Change Runtime Type -> Hardware Accelerator and select "GPU". After doing so, you might have to rerun all of the above cells as changing the runtime type causes Colab to have to reset.

Nice! After 5 epochs, our model beat the baseline score of 50.76% accuracy (our model got ~85% accuracy on the training set and ~85% accuracy on the test set).

However, our model only went through a binary classification problem rather than all of the 101 classes in the Food101 dataset, so we can't directly compare these metrics. That being said, the results so far show that our model is learning something.

**Practice:** Step through each of the main blocks of code in the cell above, what do you think each is doing? It's okay if you're not sure, we'll go through this soon. In the meantime, spend 10-minutes playing around the incredible [CNN explainer website](#). What do you notice about the layer names at the top of the webpage?

Since we've already fit a model, let's check out its architecture.

In [14]:

```
# Check out the layers in our model
model_1.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #

conv2d (Conv2D)	(None, 222, 222, 10)	280
conv2d_1 (Conv2D)	(None, 220, 220, 10)	910
max_pooling2d (MaxPooling2D)	(None, 110, 110, 10)	0
conv2d_2 (Conv2D)	(None, 108, 108, 10)	910
conv2d_3 (Conv2D)	(None, 106, 106, 10)	910
max_pooling2d_1 (MaxPooling2D)	(None, 53, 53, 10)	0
flatten (Flatten)	(None, 28090)	0
dense (Dense)	(None, 1)	28091
<hr/>		
Total params:	31,101	
Trainable params:	31,101	
Non-trainable params:	0	

What do you notice about the names of `model_1`'s layers and the layer names at the top of the [CNN explainer website](#)?

I'll let you in on a little secret: we've replicated the exact architecture they use for their model demo.

Look at you go! You're already starting to replicate models you find in the wild.

Now there are a few new things here we haven't discussed, namely:

- The `ImageDataGenerator` class and the `rescale` parameter
- The `flow_from_directory()` method
  - The `batch_size` parameter
  - The `target_size` parameter
- `Conv2D` layers (and the parameters which come with them)
- `MaxPool2D` layers (and their parameters).
- The `steps_per_epoch` and `validation_steps` parameters in the `fit()` function

Before we dive into each of these, let's see what happens if we try to fit a model we've worked with previously to our data.

## Using the same model as before

To exemplify how neural networks can be adapted to many different problems, let's see how a binary classification model we've previously built might work with our data.

**Note:** If you haven't gone through the previous classification notebook, no troubles, we'll be bringing in the a simple 4 layer architecture used to separate dots replicated from the [TensorFlow Playground environment](#).

We can use all of the same parameters in our previous model except for changing two things:

- The data - we're now working with images instead of dots.
- The input shape - we have to tell our neural network the shape of the images we're working with.
  - A common practice is to reshape images all to one size. In our case, we'll resize the images to `(224, 224, 3)`, meaning a height and width of 224 pixels and a depth of 3 for the red, green, blue colour channels.

In [15]:

```
# Set random seed
tf.random.set_seed(42)
```

```

# Create a model to replicate the TensorFlow Playground model
model_2 = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(224, 224, 3)), # dense layers expect a 1-dimensional vector as input
    tf.keras.layers.Dense(4, activation='relu'),
    tf.keras.layers.Dense(4, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# Compile the model
model_2.compile(loss='binary_crossentropy',
                 optimizer=tf.keras.optimizers.Adam(),
                 metrics=["accuracy"])

# Fit the model
history_2 = model_2.fit(train_data, # use same training data created above
                        epochs=5,
                        steps_per_epoch=len(train_data),
                        validation_data=valid_data, # use same validation data created above
                        validation_steps=len(valid_data))

```

```

Epoch 1/5
47/47 [=====] - 9s 177ms/step - loss: 1.5409 - accuracy: 0.5067
- val_loss: 0.6932 - val_accuracy: 0.5000
Epoch 2/5
47/47 [=====] - 8s 175ms/step - loss: 0.6932 - accuracy: 0.5000
- val_loss: 0.6932 - val_accuracy: 0.5000
Epoch 3/5
47/47 [=====] - 8s 172ms/step - loss: 0.6932 - accuracy: 0.5000
- val_loss: 0.6932 - val_accuracy: 0.5000
Epoch 4/5
47/47 [=====] - 8s 171ms/step - loss: 0.6932 - accuracy: 0.5000
- val_loss: 0.6932 - val_accuracy: 0.5000
Epoch 5/5
47/47 [=====] - 10s 224ms/step - loss: 0.6932 - accuracy: 0.5000
- val_loss: 0.6932 - val_accuracy: 0.5000

```

**Hmmm... our model ran but it doesn't seem like it learned anything. It only reaches 50% accuracy on the training and test sets which in a binary classification problem is as good as guessing.**

**Let's see the architecture.**

In [16]:

```
# Check out our second model's architecture
model_2.summary()
```

Model: "sequential\_1"

Layer (type)	Output Shape	Param #
<hr/>		
flatten_1 (Flatten)	(None, 150528)	0
dense_1 (Dense)	(None, 4)	602116
dense_2 (Dense)	(None, 4)	20
dense_3 (Dense)	(None, 1)	5
<hr/>		

Total params: 602,141  
Trainable params: 602,141  
Non-trainable params: 0

**Wow. One of the most noticeable things here is the much larger number of parameters in model\_2 versus model\_1.**

model\_2 has 602,141 trainable parameters where as model\_1 has only 31,101. And despite this difference, model\_1 still far and large out performs model\_2.

**Note:** You can think of trainable parameters as *patterns a model can learn from data*. Intuitively, you might think more is better. And in some cases it is. But in this case, the difference here is in the two different styles of model we're using. Where a series of dense layers have a number of different learnable parameters connected to each other and hence a higher number of possible learnable patterns, a convolutional neural network seeks to sort out and learn the most important patterns in an image. So even though there are less learnable parameters in our convolutional neural network, these are often more helpful in deciphering between different features in an image.

Since our previous model didn't work, do you have any ideas of how we might make it work?

How about we increase the number of layers?

And maybe even increase the number of neurons in each layer?

More specifically, we'll increase the number of neurons (also called hidden units) in each dense layer from 4 to 100 and add an extra layer.

**Note:** Adding extra layers or increasing the number of neurons in each layer is often referred to as increasing the complexity of your model.

In [17]:

```
# Set random seed
tf.random.set_seed(42)

# Create a model similar to model_1 but add an extra layer and increase the number of hidden units in each layer
model_3 = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(224, 224, 3)), # dense layers expect a 1-dimensional vector as input
    tf.keras.layers.Dense(100, activation='relu'), # increase number of neurons from 4 to 100 (for each layer)
    tf.keras.layers.Dense(100, activation='relu'),
    tf.keras.layers.Dense(100, activation='relu'), # add an extra layer
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# Compile the model
model_3.compile(loss='binary_crossentropy',
                 optimizer=tf.keras.optimizers.Adam(),
                 metrics=["accuracy"])

# Fit the model
history_3 = model_3.fit(train_data,
                        epochs=5,
                        steps_per_epoch=len(train_data),
                        validation_data=valid_data,
                        validation_steps=len(valid_data))
```

```
Epoch 1/5
47/47 [=====] - 9s 178ms/step - loss: 2.7921 - accuracy: 0.6260
- val_loss: 1.7039 - val_accuracy: 0.5840
Epoch 2/5
47/47 [=====] - 8s 170ms/step - loss: 1.0619 - accuracy: 0.7067
- val_loss: 0.4707 - val_accuracy: 0.7720
Epoch 3/5
47/47 [=====] - 8s 175ms/step - loss: 0.6435 - accuracy: 0.7387
- val_loss: 0.5933 - val_accuracy: 0.7780
Epoch 4/5
47/47 [=====] - 8s 175ms/step - loss: 0.6605 - accuracy: 0.7347
- val_loss: 0.8361 - val_accuracy: 0.6040
Epoch 5/5
47/47 [=====] - 8s 174ms/step - loss: 0.6071 - accuracy: 0.7500
- val_loss: 0.7511 - val_accuracy: 0.7200
```

**Woah! Looks like our model is learning again. It got ~70% accuracy on the training set and ~70% accuracy on the validation set.**

**How does the architecture look?**

In [18]:

```
# Check out model_3 architecture
model_3.summary()
```

Model: "sequential\_2"

Layer (type)	Output Shape	Param #
flatten_2 (Flatten)	(None, 150528)	0
dense_4 (Dense)	(None, 100)	15052900
dense_5 (Dense)	(None, 100)	10100
dense_6 (Dense)	(None, 100)	10100
dense_7 (Dense)	(None, 1)	101
Total params: 15,073,201		
Trainable params: 15,073,201		
Non-trainable params: 0		

**My gosh, the number of trainable parameters has increased even more than model\_2. And even with close to 500x (~15,000,000 vs. ~31,000) more trainable parameters, model\_3 still doesn't outperform model\_1.**

This goes to show the power of convolutional neural networks and their ability to learn patterns despite using less parameters.

## Binary classification: Let's break it down

We just went through a whirlwind of steps:

1. Become one with the data (visualize, visualize, visualize...)
2. Preprocess the data (prepare it for a model)
3. Create a model (start with a baseline)
4. Fit the model
5. Evaluate the model
6. Adjust different parameters and improve model (try to beat your baseline)
7. Repeat until satisfied

Let's step through each.

### 1. Import and become one with the data

Whatever kind of data you're dealing with, it's a good idea to visualize at least 10-100 samples to start to building your own mental model of the data.

In our case, we might notice that the steak images tend to have darker colours where as pizza images tend to have a distinct circular shape in the middle. These might be patterns that our neural network picks up on.

You can also notice if some of your data is messed up (for example, has the wrong label) and start to consider ways you might go about fixing it.

**Resource:** To see how this data was processed into the file format we're using, see the [preprocessing notebook](#).

If the visualization cell below doesn't work, make sure you've got the data by uncommenting the cell below.

In [19]:

```
# import zipfile  
  
# # Download zip file of pizza_steak images  
# !wget https://storage.googleapis.com/ztm_tf_course/food_vision/pizza_steak.zip  
  
# # Unzip the downloaded file  
# zip_ref = zipfile.ZipFile("pizza_steak.zip", "r")  
# zip_ref.extractall()  
# zip_ref.close()
```

In [20]:

```
# Visualize data (requires function 'view_random_image' above)  
plt.figure()  
plt.subplot(1, 2, 1)  
steak_img = view_random_image("pizza_steak/train/", "steak")  
plt.subplot(1, 2, 2)  
pizza_img = view_random_image("pizza_steak/train/", "pizza")
```

Image shape: (512, 512, 3)  
Image shape: (512, 382, 3)



## 2. Preprocess the data (prepare it for a model)

One of the most important steps for a machine learning project is creating a training and test set.

In our case, our data is already split into training and test sets. Another option here might be to create a validation set as well, but we'll leave that for now.

For an image classification project, it's standard to have your data separated into `train` and `test` directories with subfolders in each for each class.

To start we define the training and test directory paths.

In [21]:

```
# Define training and test directory paths  
train_dir = "pizza_steak/train/"  
test_dir = "pizza_steak/test/"
```

Our next step is to turn our data into batches.

A **batch** is a small subset of the dataset a model looks at during training. For example, rather than looking at 10,000 images at one time and trying to figure out the patterns, a model might only look at 32 images at a time.

It does this for a couple of reasons:

- 10,000 images (or more) might not fit into the memory of your processor (GPU).
- Trying to learn the patterns in 10,000 images in one hit could result in the model not being able to learn very well.

## Why 32?

A [batch size of 32 is good for your health.](#)

No seriously, there are many different batch sizes you could use but 32 has proven to be very effective in many different use cases and is often the default for many data preprocessing functions.

To turn our data into batches, we'll first create an instance of [ImageDataGenerator](#) for each of our datasets.

In [22]:

```
# Create train and test data generators and rescale the data
from tensorflow.keras.preprocessing.image import ImageDataGenerator
train_datagen = ImageDataGenerator(rescale=1/255.)
test_datagen = ImageDataGenerator(rescale=1/255.)
```

The `ImageDataGenerator` class helps us prepare our images into batches as well as perform transformations on them as they get loaded into the model.

You might've noticed the `rescale` parameter. This is one example of the transformations we're doing.

Remember from before how we imported an image and it's pixel values were between 0 and 255?

The `rescale` parameter, along with `1/255.` is like saying "divide all of the pixel values by 255". This results in all of the image being imported and their pixel values being normalized (converted to be between 0 and 1).

**Note:** For more transformation options such as data augmentation (we'll see this later), refer to the [ImageDataGenerator documentation](#).

Now we've got a couple of `ImageDataGenerator` instances, we can load our images from their respective directories using the `flow_from_directory` method.

In [23]:

```
# Turn it into batches
train_data = train_datagen.flow_from_directory(directory=train_dir,
                                                target_size=(224, 224),
                                                class_mode='binary',
                                                batch_size=32)

test_data = test_datagen.flow_from_directory(directory=test_dir,
                                              target_size=(224, 224),
                                              class_mode='binary',
                                              batch_size=32)
```

Found 1500 images belonging to 2 classes.

Found 500 images belonging to 2 classes.

Wonderful! Looks like our training dataset has 1500 images belonging to 2 classes (pizza and steak) and our test dataset has 500 images also belonging to 2 classes.

Some things to here:

- Due to how our directories are structured, the classes get inferred by the subdirectory names in `train_dir` and `test_dir`.
- The `target_size` parameter defines the input size of our images in `(height, width)` format.
- The `class_mode` value of `'binary'` defines our classification problem type. If we had more than two classes, we would use `'categorical'`.
- The `batch_size` defines how many images will be in each batch, we've used 32 which is the same as the default.

We can take a look at our batched images and labels by inspecting the `train_data` object.

In [24]:

```
# Get a sample of the training data batch
images, labels = train_data.next() # get the 'next' batch of images/labels
len(images), len(labels)
```

Out [24] :

(32, 32)

**Wonderful, it seems our images and labels are in batches of 32.**

**Let's see what the images look like.**

In [25] :

```
# Get the first two images
images[:2], images[0].shape
```

Out [25] :

```
(array([[[[0.47058827, 0.40784317, 0.34509805],
          [0.4784314 , 0.427451 , 0.3647059 ],
          [0.48627454, 0.43529415, 0.37254903],
          ...,
          [0.8313726 , 0.70980394, 0.48627454],
          [0.8431373 , 0.73333335, 0.5372549 ],
          [0.87843144, 0.7725491 , 0.5882353 ]],
         [[0.50980395, 0.427451 , 0.36078432],
          [0.5058824 , 0.42352945, 0.35686275],
          [0.5137255 , 0.4431373 , 0.3647059 ],
          ...,
          [0.82745105, 0.7058824 , 0.48235297],
          [0.82745105, 0.70980394, 0.5058824 ],
          [0.8431373 , 0.73333335, 0.5372549 ]],
         [[0.5254902 , 0.427451 , 0.34901962],
          [0.5372549 , 0.43921572, 0.36078432],
          [0.5372549 , 0.45098042, 0.36078432],
          ...,
          [0.82745105, 0.7019608 , 0.4784314 ],
          [0.82745105, 0.7058824 , 0.49411768],
          [0.8352942 , 0.7176471 , 0.5137255 ]],
         ...,
         [[[0.77647066, 0.5647059 , 0.2901961 ],
           [0.7803922 , 0.53333336, 0.22352943],
           [0.79215693, 0.5176471 , 0.18039216],
           ...,
           [0.30588236, 0.2784314 , 0.24705884],
           [0.24705884, 0.23137257, 0.19607845],
           [0.2784314 , 0.27450982, 0.25490198]],
          [[0.7843138 , 0.57254905, 0.29803923],
           [0.79215693, 0.54509807, 0.24313727],
           [0.80000001, 0.5254902 , 0.18823531],
           ...,
           [0.2627451 , 0.23529413, 0.20392159],
           [0.24313727, 0.227451 , 0.19215688],
           [0.26666668, 0.2627451 , 0.24313727]],
          [[0.7960785 , 0.59607846, 0.3372549 ],
           [0.7960785 , 0.5647059 , 0.26666668],
           [0.81568635, 0.54901963, 0.22352943],
           ...,
           [0.23529413, 0.19607845, 0.16078432],
           [0.3019608 , 0.26666668, 0.24705884],
           [0.26666668, 0.2509804 , 0.24705884]]],
         [[[0.38823533, 0.4666667 , 0.36078432],
           [0.3921569 , 0.46274513, 0.36078432],
```

```

[0.38431376, 0.454902 , 0.36078432],
...,
[0.5294118 , 0.627451 , 0.54509807],
[0.5294118 , 0.627451 , 0.54509807],
[0.5411765 , 0.6392157 , 0.5568628 ]],

[[0.38431376, 0.454902 , 0.3529412 ],
[0.3921569 , 0.46274513, 0.36078432],
[0.39607847, 0.4666667 , 0.37254903],
...,
[0.54509807, 0.6431373 , 0.5686275 ],
[0.5529412 , 0.6509804 , 0.5764706 ],
[0.5647059 , 0.6627451 , 0.5882353 ]],

[[0.3921569 , 0.46274513, 0.36078432],
[0.38431376, 0.454902 , 0.3529412 ],
[0.4039216 , 0.47450984, 0.3803922 ],
...,
[0.5764706 , 0.67058825, 0.6156863 ],
[0.5647059 , 0.6666667 , 0.6156863 ],
[0.5647059 , 0.6666667 , 0.6156863 ]],

...,

[[0.47058827, 0.5647059 , 0.4784314 ],
[0.4784314 , 0.5764706 , 0.4901961 ],
[0.48235297, 0.5803922 , 0.49803925],
...,
[0.39607847, 0.42352945, 0.3019608 ],
[0.37647063, 0.40000004, 0.2901961 ],
[0.3803922 , 0.4039216 , 0.3019608 ]],

[[0.45098042, 0.5529412 , 0.454902 ],
[0.46274513, 0.5647059 , 0.4666667 ],
[0.47058827, 0.57254905, 0.47450984],
...,
[0.40784317, 0.43529415, 0.3137255 ],
[0.39607847, 0.41960788, 0.31764707],
[0.38823533, 0.40784317, 0.31764707]],

[[0.47450984, 0.5764706 , 0.47058827],
[0.47058827, 0.57254905, 0.4666667 ],
[0.46274513, 0.5647059 , 0.4666667 ],
...,
[0.4039216 , 0.427451 , 0.31764707],
[0.3921569 , 0.4156863 , 0.3137255 ],
[0.4039216 , 0.42352945, 0.3372549 ]]], dtype=float32),
(224, 224, 3))

```

**Due to our `rescale` parameter, the images are now in (224, 224, 3) shape tensors with values between 0 and 1.**

**How about the labels?**

In [26]:

```
# View the first batch of labels
labels
```

Out [26]:

```
array([1., 1., 0., 1., 0., 0., 0., 1., 0., 1., 0., 0., 0., 0., 1.,
       1., 0., 1., 0., 1., 1., 0., 0., 0., 0., 0., 1., 0., 1.],
      dtype=float32)
```

**Due to the `class_mode` parameter being 'binary' our labels are either 0 (pizza) or 1 (steak).**

**Now that our data is ready, our model is going to try and figure out the patterns between the image tensors and the labels.**

### 3. Create a model (start with a baseline)

You might be wondering what your default model architecture should be.

And the truth is, there's many possible answers to this question.

A simple heuristic for computer vision models is to use the model architecture which is performing best on [ImageNet](#) (a large collection of diverse images to benchmark different computer vision models).

However, to begin with, it's good to build a smaller model to acquire a baseline result which you try to improve upon.

**Note:** In deep learning a smaller model often refers to a model with less layers than the state of the art (SOTA). For example, a smaller model might have 3-4 layers whereas a state of the art model, such as, ResNet50 might have 50+ layers.

In our case, let's take a smaller version of the model that can be found on the [CNN explainer website](#) (model\_1 from above) and build a 3 layer convolutional neural network.

In [27]:

```
# Make the creating of our model a little easier
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPool2D, Activation
from tensorflow.keras import Sequential
```

In [28]:

```
# Create the model (this can be our baseline, a 3 layer Convolutional Neural Network)
model_4 = Sequential([
    Conv2D(filters=10,
           kernel_size=3,
           strides=1,
           padding='valid',
           activation='relu',
           input_shape=(224, 224, 3)), # input layer (specify input shape)
    Conv2D(10, 3, activation='relu'),
    Conv2D(10, 3, activation='relu'),
    Flatten(),
    Dense(1, activation='sigmoid') # output layer (specify output shape)
])
```

Great! We've got a simple convolutional neural network architecture ready to go.

And it follows the typical CNN structure of:

Input -> Conv + ReLU layers (non-linearities) -> Pooling layer -> Fully connected (dense layer) as Output

Let's discuss some of the components of the `Conv2D` layer:

- The "`2D`" means our inputs are two dimensional (height and width), even though they have 3 colour channels, the convolutions are run on each channel individually.
- `filters` - these are the number of "feature extractors" that will be moving over our images.
- `kernel_size` - the size of our filters, for example, a `kernel_size` of `(3, 3)` (or just `3`) will mean each filter will have the size  $3 \times 3$ , meaning it will look at a space of  $3 \times 3$  pixels each time. The smaller the kernel, the more fine-grained features it will extract.
- `stride` - the number of pixels a filter will move across as it covers the image. A `stride` of `1` means the filter moves across each pixel `1 by 1`. A `stride` of `2` means it moves `2` pixels at a time.
- `padding` - this can be either `'same'` or `'valid'`, `'same'` adds zeros to outside of the image so the resulting output of the convolutional layer is the same as the input, whereas `'valid'` (default) cuts off excess pixels where the filter doesn't fit (e.g.  $224$  pixels wide divided by a kernel size of  $3$  ( $224/3 = 74.6$ ) means a single pixel will get cut off the end).

A **feature** can be considered any significant part of an image. For example, in our case, a feature might be the circular shape of pizza. Or the rough edges on the outside of a steak.

It's important to note that these **features** are not defined by us, instead, the model learns them as it applies different filters across the image.

**I Resources:** For a great demonstration of these in action, be sure to spend some time going through the following:

- [CNN Explainer Webpage](#) - a great visual overview of many of the concepts we're replicating here with code.
- [A guide to convolutional arithmetic for deep learning](#) - a phenomenal introduction to the math going on behind the scenes of a convolutional neural network.
- For a great explanation of padding, see this [Stack Overflow answer](#).

Now our model is ready, let's compile it.

In [29]:

```
# Compile the model
model_4.compile(loss='binary_crossentropy',
                 optimizer=Adam(),
                 metrics=['accuracy'])
```

Since we're working on a binary classification problem (pizza vs. steak), the `loss` function we're using is `'binary_crossentropy'`, if it was multiclass, we might use something like `'categorical_crossentropy'`.

Adam with all the default settings is our optimizer and our evaluation metric is accuracy.

## 4. Fit a model

Our model is compiled, time to fit it.

You'll notice two new parameters here:

- `steps_per_epoch` - this is the number of batches a model will go through per epoch, in our case, we want our model to go through all batches so it's equal to the length of `train_data` (1500 images in batches of 32 =  $1500/32 = \sim 47$  steps)
- `validation_steps` - same as above, except for the `validation_data` parameter (500 test images in batches of 32 =  $500/32 = \sim 16$  steps)

In [30]:

```
# Check lengths of training and test data generators
len(train_data), len(test_data)
```

Out[30]:

(47, 16)

In [31]:

```
# Fit the model
history_4 = model_4.fit(train_data,
                        epochs=5,
                        steps_per_epoch=len(train_data),
                        validation_data=test_data,
                        validation_steps=len(test_data))
```

Epoch 1/5  
47/47 [=====] - 10s 200ms/step - loss: 1.4615 - accuracy: 0.6573  
- val\_loss: 0.4454 - val\_accuracy: 0.8160  
Epoch 2/5

```
Epoch 2/5
47/47 [=====] - 9s 192ms/step - loss: 0.4696 - accuracy: 0.7840
- val_loss: 0.4278 - val_accuracy: 0.8200
Epoch 3/5
47/47 [=====] - 9s 194ms/step - loss: 0.3410 - accuracy: 0.8633
- val_loss: 0.4009 - val_accuracy: 0.8000
Epoch 4/5
47/47 [=====] - 9s 191ms/step - loss: 0.1866 - accuracy: 0.9400
- val_loss: 0.4956 - val_accuracy: 0.8000
Epoch 5/5
47/47 [=====] - 9s 194ms/step - loss: 0.0493 - accuracy: 0.9873
- val_loss: 0.5993 - val_accuracy: 0.7960
```

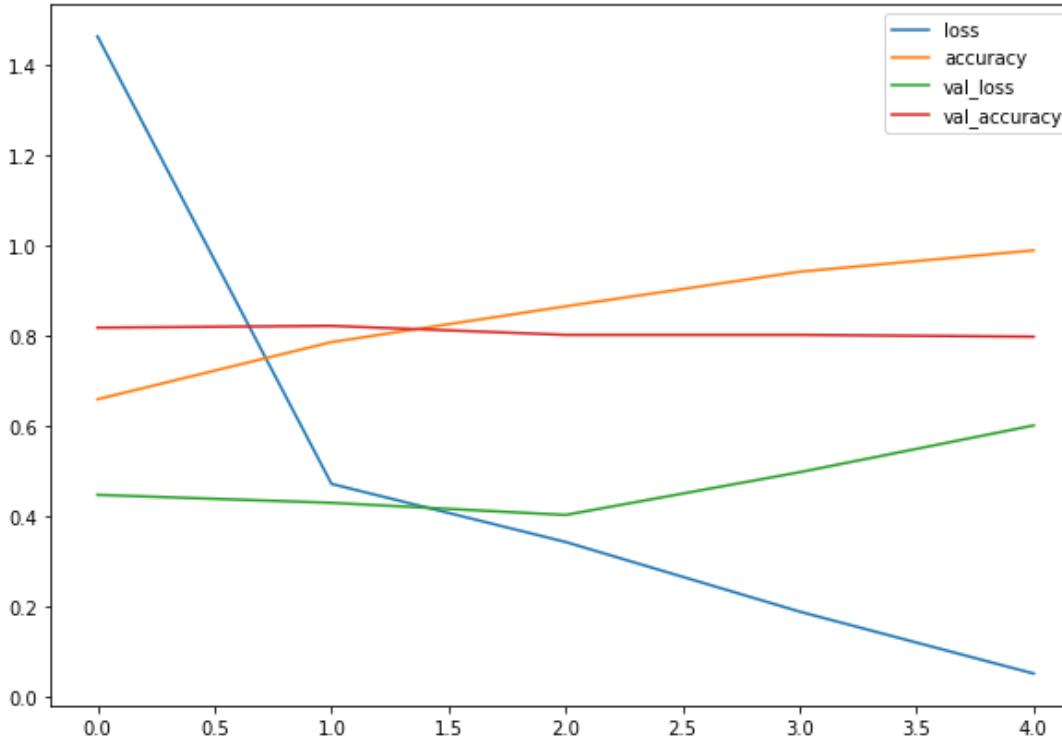
## 5. Evaluate the model

Oh yeah! Looks like our model is learning something.

Let's check out its training curves.

In [32]:

```
# Plot the training curves
import pandas as pd
pd.DataFrame(history_4.history).plot(figsize=(10, 7));
```



Hmm, judging by our loss curves, it looks like our model is **overfitting** the training dataset.

**Note:** When a model's validation loss starts to increase , it's likely that it's overfitting the training dataset. This means, it's learning the patterns in the training dataset *too well* and thus its ability to generalize to unseen data will be diminished.

To further inspect our model's training performance, let's separate the accuracy and loss curves.

In [33]:

```
# Plot the validation and training data separately
def plot_loss_curves(history):
    """
    Returns separate loss curves for training and validation metrics.
    """
    loss = history.history['loss']
    val_loss = history.history['val_loss']
```

```

accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']

epochs = range(len(history.history['loss']))

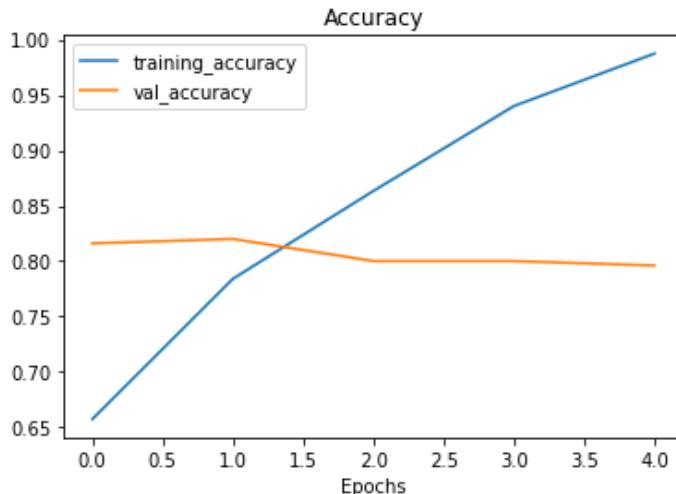
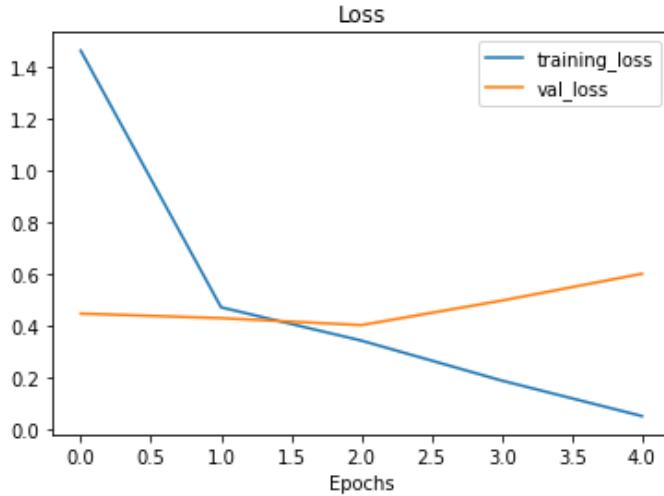
# Plot loss
plt.plot(epochs, loss, label='training_loss')
plt.plot(epochs, val_loss, label='val_loss')
plt.title('Loss')
plt.xlabel('Epochs')
plt.legend()

# Plot accuracy
plt.figure()
plt.plot(epochs, accuracy, label='training_accuracy')
plt.plot(epochs, val_accuracy, label='val_accuracy')
plt.title('Accuracy')
plt.xlabel('Epochs')
plt.legend();

```

In [34]:

```
# Check out the loss curves of model_4
plot_loss_curves(history_4)
```



The ideal position for these two curves is to follow each other. If anything, the validation curve should be slightly under the training curve. If there's a large gap between the training curve and validation curve, it means your model is probably overfitting.

In [35]:

```
# Check out our model's architecture
model_4.summary()
```

Model: "sequential\_3"

Layer (type)	Output Shape	Param #
conv2d_4 (Conv2D)	(None, 222, 222, 10)	280
conv2d_5 (Conv2D)	(None, 220, 220, 10)	910
conv2d_6 (Conv2D)	(None, 218, 218, 10)	910
flatten_3 (Flatten)	(None, 475240)	0
dense_8 (Dense)	(None, 1)	475241

Total params: 477,341  
Trainable params: 477,341  
Non-trainable params: 0

---

## 6. Adjust the model parameters

Fitting a machine learning model comes in 3 steps:

1. Create a baseline.
2. Beat the baseline by overfitting a larger model.
3. Reduce overfitting.

So far we've gone through steps 0 and 1.

And there are even a few more things we could try to further overfit our model:

- Increase the number of convolutional layers.
- Increase the number of convolutional filters.
- Add another dense layer to the output of our flattened layer.

But what we'll do instead is focus on getting our model's training curves to better align with eachother, in other words, we'll take on step 2.

Why is reducing overfitting important?

When a model performs too well on training data and poorly on unseen data, it's not much use to us if we wanted to use it in the real world.

Say we were building a pizza vs. steak food classifier app, and our model performs very well on our training data but when users tried it out, they didn't get very good results on their own food images, is that a good experience?

Not really...

So for the next few models we build, we're going to adjust a number of parameters and inspect the training curves along the way.

Namely, we'll build 2 more models:

- A ConvNet with [max pooling](#)
- A ConvNet with max pooling and data augmentation

For the first model, we'll follow the modified basic CNN structure:

Input -> Conv layers + ReLU layers (non-linearities) + Max Pooling layers -> Fully connected (dense layer) as Output

Let's built it. It'll have the same structure as `model_4` but with a [`MaxPool2D\(\)`](#) layer after each convolutional layer.

In [36]:

```
# Create the model (this can be our baseline, a 3 layer Convolutional Neural Network)
model_5 = Sequential([
    Conv2D(10, 3, activation='relu', input_shape=(222, 222, 3)),
    MaxPool2D(),
    Conv2D(10, 3, activation='relu'),
    MaxPool2D(),
    Flatten(),
    Dense(1)
])
```

```

Conv2D(10, 3, activation='relu', input_shape=(224, 224, 3)),
MaxPool2D(pool_size=2), # reduce number of features by half
Conv2D(10, 3, activation='relu'),
MaxPool2D(),
Conv2D(10, 3, activation='relu'),
MaxPool2D(),
Flatten(),
Dense(1, activation='sigmoid')
])

```

**Woah, we've got another layer type we haven't seen before.**

**If convolutional layers learn the features of an image you can think of a Max Pooling layer as figuring out the *most important* of those features. We'll see this an example of this in a moment.**

In [37]:

```

# Compile model (same as model_4)
model_5.compile(loss='binary_crossentropy',
                  optimizer=Adam(),
                  metrics=['accuracy'])

```

In [38]:

```

# Fit the model
history_5 = model_5.fit(train_data,
                        epochs=5,
                        steps_per_epoch=len(train_data),
                        validation_data=test_data,
                        validation_steps=len(test_data))

```

```

Epoch 1/5
47/47 [=====] - 9s 186ms/step - loss: 0.6214 - accuracy: 0.6353
- val_loss: 0.5094 - val_accuracy: 0.7420
Epoch 2/5
47/47 [=====] - 9s 181ms/step - loss: 0.4870 - accuracy: 0.7720
- val_loss: 0.4172 - val_accuracy: 0.8280
Epoch 3/5
47/47 [=====] - 9s 185ms/step - loss: 0.4188 - accuracy: 0.8160
- val_loss: 0.3403 - val_accuracy: 0.8580
Epoch 4/5
47/47 [=====] - 9s 190ms/step - loss: 0.4053 - accuracy: 0.8153
- val_loss: 0.3308 - val_accuracy: 0.8680
Epoch 5/5
47/47 [=====] - 9s 189ms/step - loss: 0.3878 - accuracy: 0.8373
- val_loss: 0.3461 - val_accuracy: 0.8680

```

**Okay, it looks like our model with max pooling ( `model_5` ) is performing worse on the training set but better on the validation set.**

**Before we checkout its training curves, let's check out its architecture.**

In [39]:

```

# Check out the model architecture
model_5.summary()

```

Model: "sequential\_4"

Layer (type)	Output Shape	Param #
conv2d_7 (Conv2D)	(None, 222, 222, 10)	280
max_pooling2d_2 (MaxPooling2D)	(None, 111, 111, 10)	0
conv2d_8 (Conv2D)	(None, 109, 109, 10)	910
max_pooling2d_3 (MaxPooling2D)	(None, 54, 54, 10)	0
conv2d_9 (Conv2D)	(None, 52, 52, 10)	910

max_pooling2d_4 (MaxPooling2 (None, 26, 26, 10)	0
flatten_4 (Flatten)	(None, 6760)
dense_9 (Dense)	(None, 1)
=====	
Total params: 8,861	
Trainable params: 8,861	
Non-trainable params: 0	

Do you notice what's going on here with the output shape in each MaxPooling2D layer?

It gets halved each time. This is effectively the MaxPooling2D layer taking the outputs of each Conv2D layer and saying "I only want the most important features, get rid of the rest".

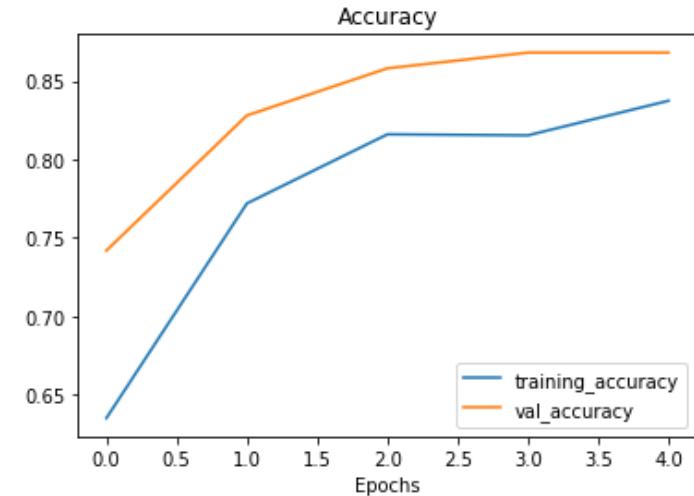
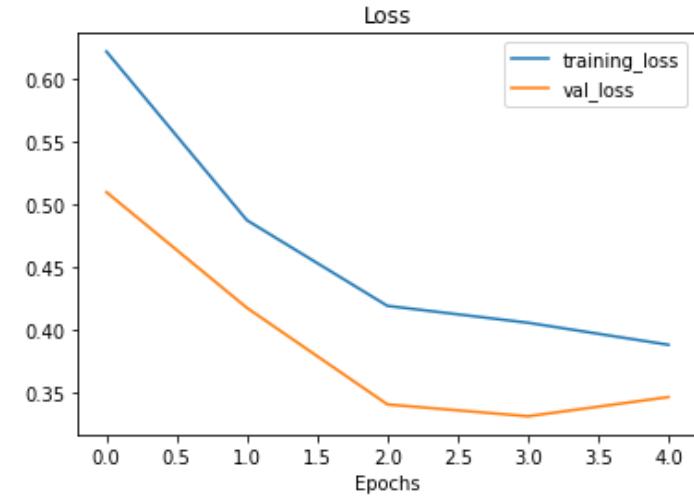
The bigger the pool\_size parameter, the more the max pooling layer will squeeze the features out of the image. However, too big and the model might not be able to learn anything.

The results of this pooling are seen in a major reduction of total trainable parameters (8,861 in model\_5 and 477,431 in model\_4).

Time to check out the loss curves.

In [40]:

```
# Plot loss curves of model_5 results
plot_loss_curves(history_5)
```



Nice! We can see the training curves get a lot closer to each other. However, our the validation loss looks to start increasing towards the end and in turn potentially leading to overfitting.

Time to dig into our bag of tricks and try another method of overfitting prevention, data augmentation.

**First, we'll see how it's done with code then we'll discuss what it's doing.**

To implement data augmentation, we'll have to reinstantiate our `ImageDataGenerator` instances.

In [41]:

```
# Create ImageDataGenerator training instance with data augmentation
train_datagen_augmented = ImageDataGenerator(rescale=1/255.,
                                              rotation_range=20, # rotate the image slightly between 0 and 20 degrees (note: this is an int not a float)
                                              shear_range=0.2, # shear the image
                                              zoom_range=0.2, # zoom into the image
                                              width_shift_range=0.2, # shift the image width ways
                                              height_shift_range=0.2, # shift the image height ways
                                              horizontal_flip=True) # flip the image on the horizontal axis

# Create ImageDataGenerator training instance without data augmentation
train_datagen = ImageDataGenerator(rescale=1/255.)

# Create ImageDataGenerator test instance without data augmentation
test_datagen = ImageDataGenerator(rescale=1/255.)
```

## QUESTION: What's data augmentation?

**Data augmentation** is the process of altering our training data, leading to it having more diversity and in turn allowing our models to learn more generalizable patterns. Altering might mean adjusting the rotation of an image, flipping it, cropping it or something similar.

**Doing this simulates the kind of data a model might be used on in the real world.**

If we're building a pizza vs. steak application, not all of the images our users take might be in similar setups to our training data. Using data augmentation gives us another way to prevent overfitting and in turn make our model more generalizable.

**Note:** Data augmentation is usually only performed on the training data. Using the `ImageDataGenerator` built-in data augmentation parameters our images are left as they are in the directories but are randomly manipulated when loaded into the model.

In [42]:

```
batch_size=32,  
class_mode='binary')
```

Augmented training images:  
Found 1500 images belonging to 2 classes.  
Non-augmented training images:  
Found 1500 images belonging to 2 classes.  
Unchanged test images:  
Found 500 images belonging to 2 classes.

**Better than talk about data augmentation, how about we see it?**

(remember our motto? **visualize, visualize, visualize...**)

In [43]:

```
# Get data batch samples  
images, labels = train_data.next()  
augmented_images, augmented_labels = train_data_augmented.next() # Note: labels aren't a  
ugmented, they stay the same
```

In [44]:

```
# Show original image and augmented image  
random_number = random.randint(0, 32) # we're making batches of size 32, so we'll get a  
random instance  
plt.imshow(images[random_number])  
plt.title(f"Original image")  
plt.axis(False)  
plt.figure()  
plt.imshow(augmented_images[random_number])  
plt.title(f"Augmented image")  
plt.axis(False);
```

Original image



Augmented image



After going through a sample of original and augmented images, you can start to see some of the example transformations on the training images.

Notice how some of the augmented images look like slightly warped versions of the original image. This means our model will be forced to try and learn patterns in less-than-perfect images, which is often the case when

using real-world images.

## Question: Should I use data augmentation? And how much should I augment?

Data augmentation is a way to try and prevent a model overfitting. If your model is overfitting (e.g. the validation loss keeps increasing), you may want to try using data augmentation.

As for how much to data augment, there's no set practice for this. Best to check out the options in the `ImageDataGenerator` class and think about how a model in your use case might benefit from some data augmentation.

Now we've got augmented data, let's try and refit a model on it and see how it affects training.

We'll use the same model as `model_5`.

In [45]:

```
# Create the model (same as model_5)
model_6 = Sequential([
    Conv2D(10, 3, activation='relu', input_shape=(224, 224, 3)),
    MaxPool2D(pool_size=2), # reduce number of features by half
    Conv2D(10, 3, activation='relu'),
    MaxPool2D(),
    Conv2D(10, 3, activation='relu'),
    MaxPool2D(),
    Flatten(),
    Dense(1, activation='sigmoid')
])

# Compile the model
model_6.compile(loss='binary_crossentropy',
                 optimizer=Adam(),
                 metrics=['accuracy'])

# Fit the model
history_6 = model_6.fit(train_data_augmented, # changed to augmented training data
                        epochs=5,
                        steps_per_epoch=len(train_data_augmented),
                        validation_data=test_data,
                        validation_steps=len(test_data))

Epoch 1/5
47/47 [=====] - 23s 474ms/step - loss: 0.7050 - accuracy: 0.4540
- val_loss: 0.6827 - val_accuracy: 0.6340
Epoch 2/5
47/47 [=====] - 21s 456ms/step - loss: 0.6958 - accuracy: 0.5027
- val_loss: 0.6748 - val_accuracy: 0.7420
Epoch 3/5
47/47 [=====] - 22s 461ms/step - loss: 0.6910 - accuracy: 0.5727
- val_loss: 0.6589 - val_accuracy: 0.5940
Epoch 4/5
47/47 [=====] - 21s 457ms/step - loss: 0.6580 - accuracy: 0.6567
- val_loss: 0.5421 - val_accuracy: 0.8180
Epoch 5/5
47/47 [=====] - 21s 457ms/step - loss: 0.7898 - accuracy: 0.7220
- val_loss: 0.6415 - val_accuracy: 0.5980
```

## Question: Why didn't our model get very good results on the training set to begin with?

It's because when we created `train_data_augmented` we turned off data shuffling using `shuffle=False` which means our model only sees a batch of a single kind of images at a time.

For example, the pizza class gets loaded in first because it's the first class. Thus its performance is measured on only a single class rather than both classes. The validation data performance improves steadily because it contains shuffled data.

Since we only set `shuffle=False` for demonstration purposes (so we could plot the same augmented and

**non-augmented image), we can fix this by setting `shuffle=True` on future data generators.**

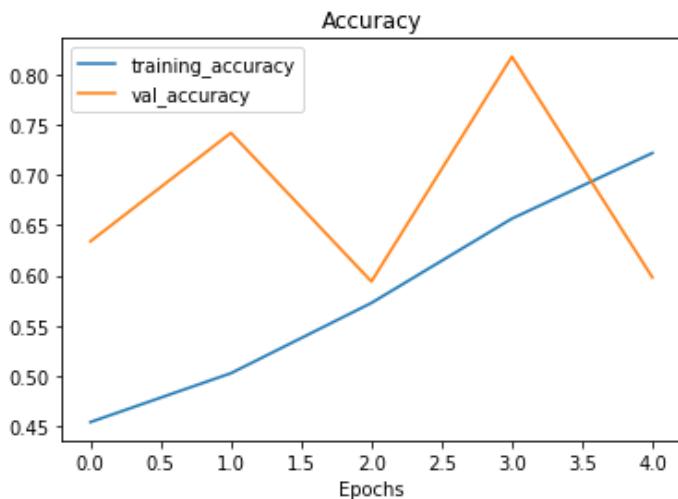
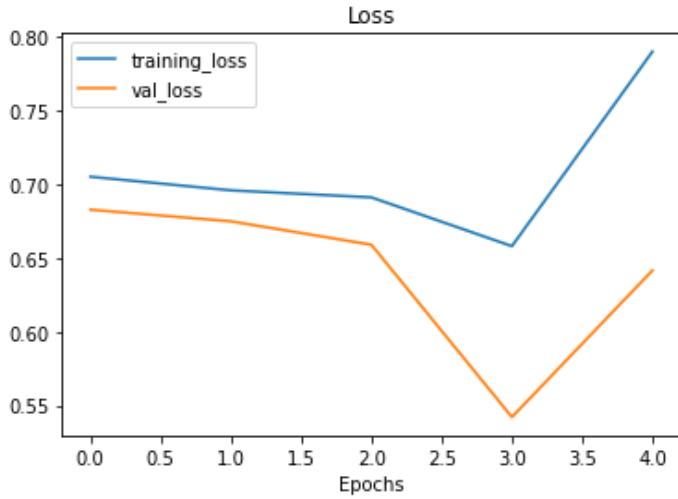
You may have also noticed each epoch taking longer when training with augmented data compared to when training with non-augmented data (~25s per epoch vs. ~10s per epoch).

This is because the `ImageDataGenerator` instance augments the data as it's loaded into the model. The benefit of this is that it leaves the original images unchanged. The downside is that it takes longer to load them in.

¶ Note: One possible method to speed up dataset manipulation would be to look into [TensorFlow's parallel reads and buffered prefetching options](#).

In [46]:

```
# Check model's performance history training on augmented data
plot_loss_curves(history_6)
```



It seems our validation loss curve is heading in the right direction but it's a bit jumpy (the most ideal loss curve isn't too spiky but a smooth descent, however, a perfectly smooth loss curve is the equivalent of a fairytale).

Let's see what happens when we shuffle the augmented training data.

In [47]:

e) # Shuffle data (default)

Found 1500 images belonging to 2 classes.

In [48]:

```
# Create the model (same as model_5 and model_6)
model_7 = Sequential([
    Conv2D(10, 3, activation='relu', input_shape=(224, 224, 3)),
    MaxPool2D(),
    Conv2D(10, 3, activation='relu'),
    MaxPool2D(),
    Conv2D(10, 3, activation='relu'),
    MaxPool2D(),
    Flatten(),
    Dense(1, activation='sigmoid')
])

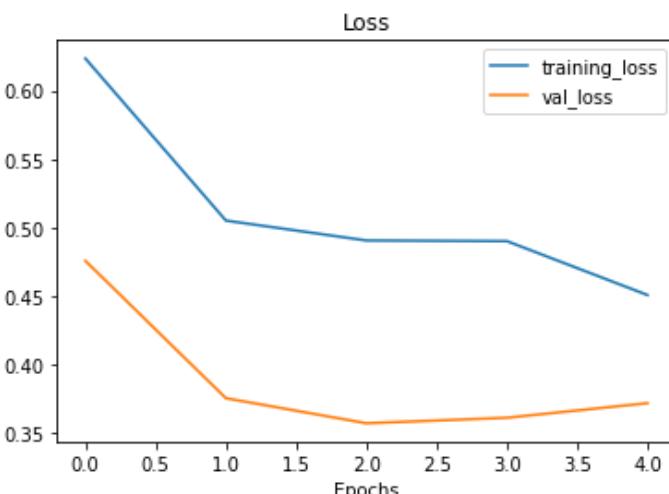
# Compile the model
model_7.compile(loss='binary_crossentropy',
                 optimizer=Adam(),
                 metrics=['accuracy'])

# Fit the model
history_7 = model_7.fit(train_data_augmented_shuffled, # now the augmented data is shuffled
                        epochs=5,
                        steps_per_epoch=len(train_data_augmented_shuffled),
                        validation_data=test_data,
                        validation_steps=len(test_data))
```

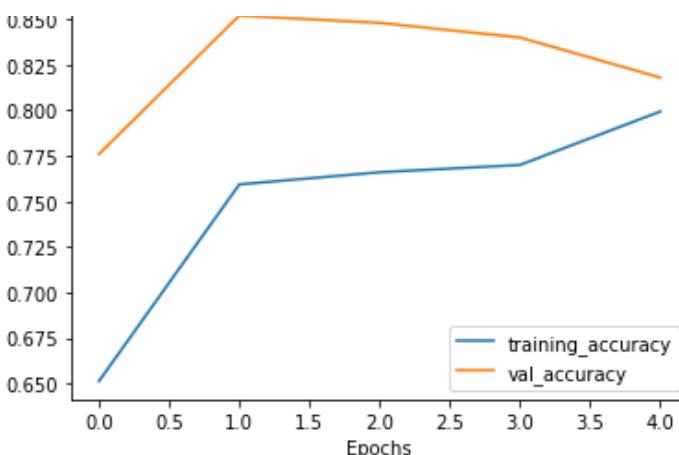
Epoch 1/5  
47/47 [=====] - 23s 472ms/step - loss: 0.6233 - accuracy: 0.6513  
- val\_loss: 0.4756 - val\_accuracy: 0.7760  
Epoch 2/5  
47/47 [=====] - 22s 469ms/step - loss: 0.5050 - accuracy: 0.7593  
- val\_loss: 0.3754 - val\_accuracy: 0.8520  
Epoch 3/5  
47/47 [=====] - 22s 465ms/step - loss: 0.4905 - accuracy: 0.7660  
- val\_loss: 0.3571 - val\_accuracy: 0.8480  
Epoch 4/5  
47/47 [=====] - 22s 458ms/step - loss: 0.4900 - accuracy: 0.7700  
- val\_loss: 0.3611 - val\_accuracy: 0.8400  
Epoch 5/5  
47/47 [=====] - 21s 457ms/step - loss: 0.4507 - accuracy: 0.7993  
- val\_loss: 0.3718 - val\_accuracy: 0.8180

In [49]:

```
# Check model's performance history training on augmented data
plot_loss_curves(history_7)
```



Accuracy



Notice with `model_7` how the performance on the training dataset improves almost immediately compared to `model_6`. This is because we shuffled the training data as we passed it to the model using the parameter `shuffle=True` in the `flow from directory` method.

This means the model was able to see examples of both pizza and steak images in each batch and in turn be evaluated on what it learned from both images rather than just one kind.

Also, our loss curves look a little bit smoother with shuffled data (comparing history 6 to history 7).

## **7. Repeat until satisfied**

We've trained a few model's on our dataset already and so far they're performing pretty good.

Since we've already beaten our baseline, there are a few things we could try to continue to improve our model:

- Increase the number of model layers (e.g. add more convolutional layers).
  - Increase the number of filters in each convolutional layer (e.g. from 10 to 32, 64, or 128, these numbers aren't set in stone either, they are usually found through trial and error).
  - Train for longer (more epochs).
  - Finding an ideal learning rate.
  - Get more data (give the model more opportunities to learn).
  - Use transfer learning to leverage what another image model has learned and adjust it for our own use case.

Adjusting each of these settings (except for the last two) during model development is usually referred to as **hyperparameter tuning**.

You can think of hyperparameter tuning as similar to adjusting the settings on your oven to cook your favourite dish. Although your oven does most of the cooking for you, you can help it by tweaking the dials.

Let's go back to right where we started and try our original model (`model_1` or the TinyVGG architecture from [CNN explainer](#)).

In [50]:

```

metrics=["accuracy"])

# Fit the model
history_8 = model_8.fit(train_data_augmented_shuffled,
                        epochs=5,
                        steps_per_epoch=len(train_data_augmented_shuffled),
                        validation_data=test_data,
                        validation_steps=len(test_data))

Epoch 1/5
47/47 [=====] - 37s 753ms/step - loss: 0.6479 - accuracy: 0.6267
- val_loss: 0.5632 - val_accuracy: 0.6480
Epoch 2/5
47/47 [=====] - 25s 536ms/step - loss: 0.5612 - accuracy: 0.7307
- val_loss: 0.4197 - val_accuracy: 0.8200
Epoch 3/5
47/47 [=====] - 22s 467ms/step - loss: 0.5459 - accuracy: 0.7393
- val_loss: 0.3961 - val_accuracy: 0.8380
Epoch 4/5
47/47 [=====] - 22s 463ms/step - loss: 0.5115 - accuracy: 0.7607
- val_loss: 0.3879 - val_accuracy: 0.8340
Epoch 5/5
47/47 [=====] - 22s 466ms/step - loss: 0.5123 - accuracy: 0.7573
- val_loss: 0.3865 - val_accuracy: 0.8360

```

**Note:** You might've noticed we used some slightly different code to build `model_8` as compared to `model_1`. This is because of the imports we did before, such as `from tensorflow.keras.layers import Conv2D` reduce the amount of code we had to write. Although the code is different, the architectures are the same.

In [51]:

```
# Check model_1 architecture (same as model_8)
model_1.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 222, 222, 10)	280
conv2d_1 (Conv2D)	(None, 220, 220, 10)	910
max_pooling2d (MaxPooling2D)	(None, 110, 110, 10)	0
conv2d_2 (Conv2D)	(None, 108, 108, 10)	910
conv2d_3 (Conv2D)	(None, 106, 106, 10)	910
max_pooling2d_1 (MaxPooling2D)	(None, 53, 53, 10)	0
flatten (Flatten)	(None, 28090)	0
dense (Dense)	(None, 1)	28091

Total params: 31,101  
Trainable params: 31,101  
Non-trainable params: 0

In [52]:

```
# Check model_8 architecture (same as model_1)
model_8.summary()
```

Model: "sequential\_7"

Layer (type)	Output Shape	Param #
--------------	--------------	---------

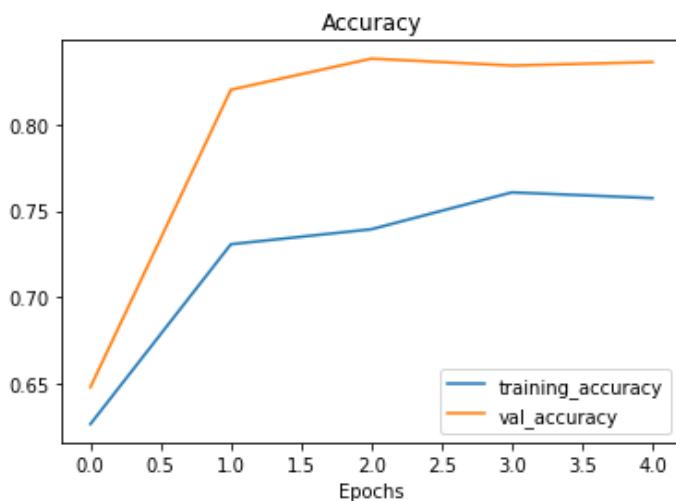
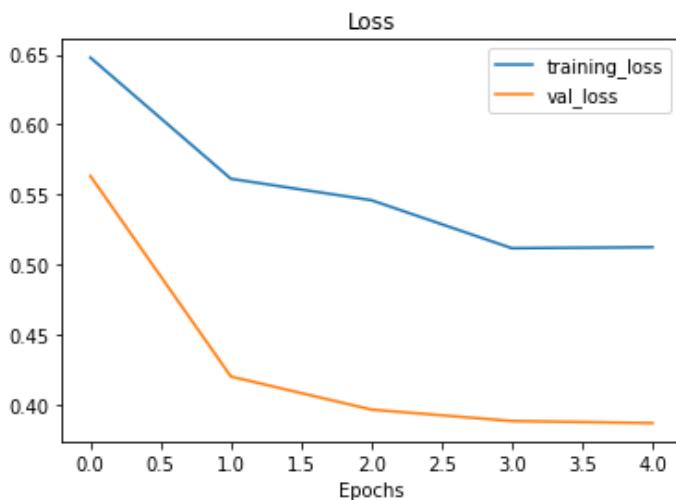
conv2d_16 (Conv2D)	(None, 222, 222, 10)	280
conv2d_17 (Conv2D)	(None, 220, 220, 10)	910
max_pooling2d_11 (MaxPooling)	(None, 110, 110, 10)	0
conv2d_18 (Conv2D)	(None, 108, 108, 10)	910
conv2d_19 (Conv2D)	(None, 106, 106, 10)	910
max_pooling2d_12 (MaxPooling)	(None, 53, 53, 10)	0
flatten_7 (Flatten)	(None, 28090)	0
dense_12 (Dense)	(None, 1)	28091

Total params: 31,101  
Trainable params: 31,101  
Non-trainable params: 0

**Now let's check out our TinyVGG model's performance.**

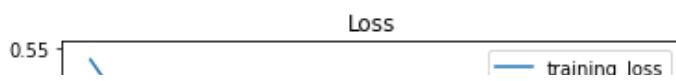
In [53]:

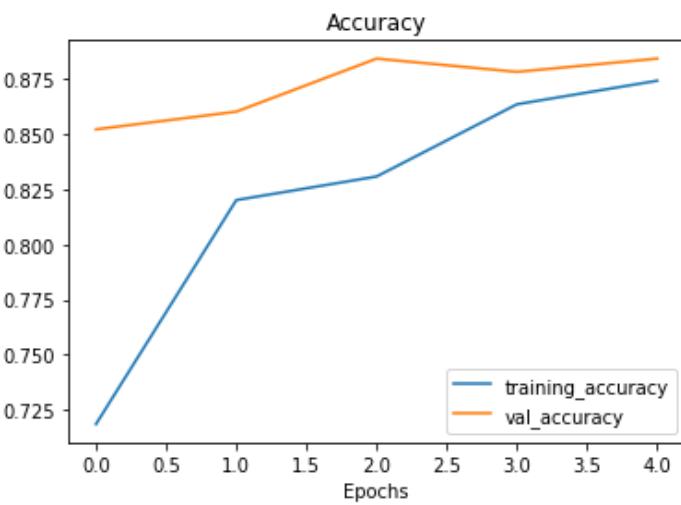
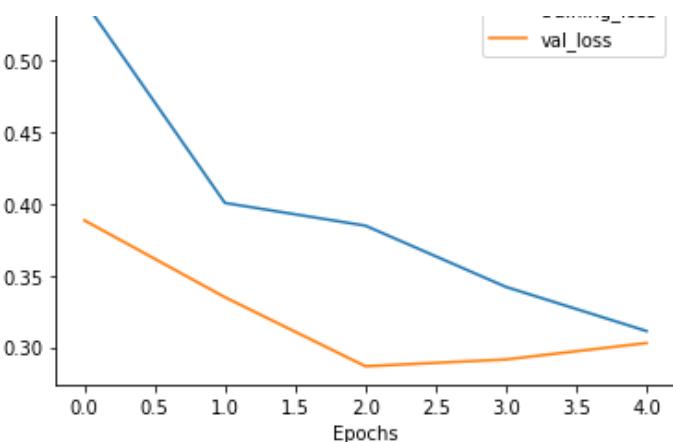
```
# Check out the TinyVGG model performance
plot_loss_curves(history_8)
```



In [54]:

```
# How does this training curve look compared to the one above?
plot_loss_curves(history_1)
```





Hmm, our training curves are looking good, but our model's performance on the training and test sets didn't improve much compared to the previous model.

Taking another look at the training curves, it looks like our model's performance might improve if we trained it a little longer (more epochs).

Perhaps that's something you like to try?

## Making a prediction with our trained model

What good is a trained model if you can't make predictions with it?

To really test it out, we'll upload a couple of our own images and see how the model goes.

First, let's remind ourselves of the classnames and view the image we're going to test on.

In [55]:

```
# Classes we're working with
print(class_names)

['.DS_Store' 'pizza' 'steak']
```

The first test image we're going to use is [a delicious steak](#) I cooked the other day.

In [56]:

```
# View our example image
!wget https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/images/03-steak.jpeg
steak = mpimg.imread("03-steak.jpeg")
plt.imshow(steak)
plt.axis(False);
```

--2021-07-14 06:11:00-- https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/images/03-steak.jpeg

resolving raw.githubusercontent.com (raw.githubusercontent.com) ... 185.199.108.133, 185.199.108.133, 99.109.133, 185.199.110.133, ...

Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443.

.. connected.

HTTP request sent, awaiting response... 200 OK

Length: 1978213 (1.9M) [image/jpeg]

Saving to: '03-steak.jpeg'

03-steak.jpeg 100% [=====] 1.89M --.-KB/s in 0.01s

2021-07-14 06:11:00 (193 MB/s) - '03-steak.jpeg' saved [1978213/1978213]



In [57]:

```
# Check the shape of our image  
steak.shape
```

Out[57]:

(4032, 3024, 3)

Since our model takes in images of shapes (224, 224, 3), we've got to reshape our custom image to use it with our model.

To do so, we can import and decode our image using `tf.io.read_file` (for reading files) and `tf.image` (for resizing our image and turning it into a tensor).

**Note:** For your model to make predictions on unseen data, for example, your own custom images, the custom image has to be in the same shape as your model has been trained on. In more general terms, to make predictions on custom data it has to be in the same form that your model has been trained on.

In [58]:

```
# Create a function to import an image and resize it to be able to be used with our model  
def load_and_prep_image(filename, img_shape=224):  
    """  
    Reads an image from filename, turns it into a tensor  
    and reshapes it to (img_shape, img_shape, colour_channel).  
    """  
    # Read in target file (an image)  
    img = tf.io.read_file(filename)  
  
    # Decode the read file into a tensor & ensure 3 colour channels  
    # (our model is trained on images with 3 colour channels and sometimes images have 4 colour channels)  
    img = tf.image.decode_image(img, channels=3)  
  
    # Resize the image (to the same size our model was trained on)  
    img = tf.image.resize(img, size = [img_shape, img_shape])  
  
    # Rescale the image (get all values between 0 and 1)  
    img = img/255.
```

```
return img
```

Now we've got a function to load our custom image, let's load it in.

In [59]:

```
# Load in and preprocess our custom image
steak = load_and_prep_image("03-steak.jpeg")
steak
```

Out[59]:

```
<tf.Tensor: shape=(224, 224, 3), dtype=float32, numpy=
array([[[[0.6377451 , 0.6220588 , 0.57892156],
       [0.6504902 , 0.63186276, 0.5897059 ],
       [0.63186276, 0.60833335, 0.5612745 ],
       ...,
       [0.52156866, 0.05098039, 0.09019608],
       [0.49509802, 0.04215686, 0.07058824],
       [0.52843136, 0.07745098, 0.10490196]],

      [[0.6617647 , 0.6460784 , 0.6107843 ],
       [0.6387255 , 0.6230392 , 0.57598037],
       [0.65588236, 0.63235295, 0.5852941 ],
       ...,
       [0.5352941 , 0.06862745, 0.09215686],
       [0.529902 , 0.05931373, 0.09460784],
       [0.5142157 , 0.05539216, 0.08676471]],

      [[0.6519608 , 0.6362745 , 0.5892157 ],
       [0.6392157 , 0.6137255 , 0.56764704],
       [0.65637255, 0.6269608 , 0.5828431 ],
       ...,
       [0.53137255, 0.06470589, 0.08039216],
       [0.527451 , 0.06862745, 0.1        ],
       [0.52254903, 0.05196078, 0.0872549 ]],

      ...,

      [[0.49313724, 0.42745098, 0.31029412],
       [0.05441177, 0.01911765, 0.          ],
       [0.2127451 , 0.16176471, 0.09509804],
       ...,
       [0.6132353 , 0.59362745, 0.57009804],
       [0.65294117, 0.6333333 , 0.6098039 ],
       [0.64166665, 0.62990195, 0.59460783]],

      [[0.65392154, 0.5715686 , 0.45        ],
       [0.6367647 , 0.54656863, 0.425       ],
       [0.04656863, 0.01372549, 0.          ],
       ...,
       [0.6372549 , 0.61764705, 0.59411764],
       [0.63529414, 0.6215686 , 0.5892157 ],
       [0.6401961 , 0.62058824, 0.59705883]],

      [[0.1        , 0.05539216, 0.          ],
       [0.48333332, 0.40882352, 0.29117647],
       [0.65        , 0.5686275 , 0.44019607],
       ...,
       [0.6308824 , 0.6161765 , 0.5808824 ],
       [0.6519608 , 0.63186276, 0.5901961 ],
       [0.6338235 , 0.6259804 , 0.57892156]]], dtype=float32)>
```

Wonderful, our image is in tensor format, time to try it with our model!

In [60]:

```
# Make a prediction on our custom image (spoiler: this won't work)
model_8.predict(steak)
```

```
ValueError                                     Traceback (most recent call last)
<ipython-input-60-fd7eef5274d1> in <module>()
      1 # Make a prediction on our custom image (spoiler: this won't work)
----> 2 model_8.predict(stake)

/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/training.py in predict(self, x, batch_size, verbose, steps, callbacks, max_queue_size, workers, use_multiprocessing)
   1725         for step in data_handler.steps():
   1726             callbacks.on_predict_batch_begin(step)
-> 1727             tmp_batch_outputs = self.predict_function(iterator)
   1728             if data_handler.should_sync:
   1729                 context.async_wait()

/usr/local/lib/python3.7/dist-packages/tensorflow/python/eager/def_function.py in __call__(self, *args, **kwds)
   887
   888     with OptionalXlaContext(self._jit_compile):
--> 889         result = self._call(*args, **kwds)
   890
   891     new_tracing_count = self.experimental_get_tracing_count()

/usr/local/lib/python3.7/dist-packages/tensorflow/python/eager/def_function.py in _call(self, *args, **kwds)
   931         # This is the first call of __call__, so we have to initialize.
   932         initializers = []
--> 933         self._initialize(args, kwds, add_initializers_to=initializers)
   934     finally:
   935         # At this point we know that the initialization is complete (or less

/usr/local/lib/python3.7/dist-packages/tensorflow/python/eager/def_function.py in _initialize(self, args, kwds, add_initializers_to)
   762     self._concrete_stateful_fn = (
   763         self._stateful_fn._get_concrete_function_internal_garbage_collected( # p
pylint: disable=protected-access
--> 764             *args, **kwds))
   765
   766     def invalid_creator_scope(*unused_args, **unused_kwds):

/usr/local/lib/python3.7/dist-packages/tensorflow/python/eager/function.py in _get_concrete_function_internal_garbage_collected(self, *args, **kwargs)
  3048         args, kwargs = None, None
  3049         with self._lock:
-> 3050             graph_function, _ = self._maybe_define_function(args, kwargs)
  3051             return graph_function
  3052

/usr/local/lib/python3.7/dist-packages/tensorflow/python/eager/function.py in _maybe_define_function(self, args, kwargs)
  3442             self._function_cache.missed.add(call_context_key)
  3443             graph_function = self._create_graph_function(args, kwargs)
--> 3444             self._function_cache.primary[cache_key] = graph_function
  3445
  3446

/usr/local/lib/python3.7/dist-packages/tensorflow/python/eager/function.py in _create_graph_function(self, args, kwargs, override_flat_arg_shapes)
  3287             arg_names=arg_names,
  3288             override_flat_arg_shapes=override_flat_arg_shapes,
-> 3289             capture_by_value=self._capture_by_value),
  3290             self._function_attributes,
  3291             function_spec=self.function_spec,

/usr/local/lib/python3.7/dist-packages/tensorflow/python/framework/func_graph.py in func_graph_from_py_func(name, python_func, args, kwargs, signature, func_graph, autograph, autograph_options, add_control_dependencies, arg_names, op_return_value, collections, capture_by_value, override_flat_arg_shapes)
  997             _, original_func = tf_decorator.unwrap(python_func)
  998
--> 999             func_outputs = python_func(*func_args, **func_kwargs)
1000
1001             # invariant: `func_outputs` contains only Tensors, CompositeTensors,
```

```

/usr/local/lib/python3.7/dist-packages/tensorflow/python/eager/def_function.py in wrapped_fn(*args, **kwds)
    670         # the function a weak reference to itself to avoid a reference cycle.
    671         with OptionalXlaContext(compile_with_xla):
--> 672             out = weak_wrapped_fn().__wrapped__(*args, **kwds)
    673         return out
    674

/usr/local/lib/python3.7/dist-packages/tensorflow/python/framework/func_graph.py in wrapper(*args, **kwargs)
    984         except Exception as e:  # pylint:disable=broad-except
    985             if hasattr(e, "ag_error_metadata"):
--> 986                 raise e.ag_error_metadata.to_exception(e)
    987             else:
    988                 raise

ValueError: in user code:

    /usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/training.py:156
9 predict_function  *
    return step_function(self, iterator)
    /usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/training.py:155
9 step_function  **
    outputs = model.distribute_strategy.run(run_step, args=(data,))
    /usr/local/lib/python3.7/dist-packages/tensorflow/python/distribute/distribute_lib.py
:1285 run
    return self._extended.call_for_each_replica(fn, args=args, kwargs=kwargs)
    /usr/local/lib/python3.7/dist-packages/tensorflow/python/distribute/distribute_lib.py
:2833 call_for_each_replica
    return self._call_for_each_replica(fn, args, kwargs)
    /usr/local/lib/python3.7/dist-packages/tensorflow/python/distribute/distribute_lib.py
:3608 _call_for_each_replica
    return fn(*args, **kwargs)
    /usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/training.py:155
2 run_step  **
    outputs = model.predict_step(data)
    /usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/training.py:152
5 predict_step
    return self(x, training=False)
    /usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/base_layer.py:1
013 __call__
    input_spec.assert_input_compatibility(self.input_spec, inputs, self.name)
    /usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/input_spec.py:2
35 assert_input_compatibility
    str(tuple(shape)))

    ValueError: Input 0 of layer sequential_7 is incompatible with the layer: : expected
min_ndim=4, found ndim=3. Full shape received: (32, 224, 3)

```

**There's one more problem...**

**Although our image is in the same shape as the images our model has been trained on, we're still missing a dimension.**

**Remember how our model was trained in batches?**

**Well, the batch size becomes the first dimension.**

**So in reality, our model was trained on data in the shape of `(batch_size, 224, 224, 3)`.**

**We can fix this by adding an extra to our custom image tensor using `tf.expand_dims`.**

In [61]:

```

# Add an extra axis
print(f"Shape before new dimension: {steak.shape}")
steak = tf.expand_dims(steak, axis=0) # add an extra dimension at axis 0
#steak = steak[tf.newaxis, ...] # alternative to the above, '...' is short for 'every other dimension'

```

```
print(f"Shape after new dimension: {steak.shape}")
steak
```

```
Shape before new dimension: (224, 224, 3)
Shape after new dimension: (1, 224, 224, 3)
```

Out[61]:

```
<tf.Tensor: shape=(1, 224, 224, 3), dtype=float32, numpy=
array([[[[0.6377451 , 0.6220588 , 0.57892156],
       [0.6504902 , 0.63186276, 0.5897059 ],
       [0.63186276, 0.60833335, 0.5612745 ],
       ...,
       [0.52156866, 0.05098039, 0.09019608],
       [0.49509802, 0.04215686, 0.07058824],
       [0.52843136, 0.07745098, 0.10490196]],

      [[0.6617647 , 0.6460784 , 0.6107843 ],
       [0.6387255 , 0.6230392 , 0.57598037],
       [0.65588236, 0.63235295, 0.5852941 ],
       ...,
       [0.5352941 , 0.06862745, 0.09215686],
       [0.529902 , 0.05931373, 0.09460784],
       [0.5142157 , 0.05539216, 0.08676471]],

      [[0.6519608 , 0.6362745 , 0.5892157 ],
       [0.6392157 , 0.6137255 , 0.56764704],
       [0.65637255, 0.6269608 , 0.5828431 ],
       ...,
       [0.53137255, 0.06470589, 0.08039216],
       [0.527451 , 0.06862745, 0.1          ],
       [0.52254903, 0.05196078, 0.0872549 ]],

      ...,

      [[0.49313724, 0.42745098, 0.31029412],
       [0.05441177, 0.01911765, 0.          ],
       [0.2127451 , 0.16176471, 0.09509804],
       ...,
       [0.6132353 , 0.59362745, 0.57009804],
       [0.65294117, 0.6333333 , 0.6098039 ],
       [0.64166665, 0.62990195, 0.59460783]],

      [[0.65392154, 0.5715686 , 0.45        ],
       [0.6367647 , 0.54656863, 0.425       ],
       [0.04656863, 0.01372549, 0.          ],
       ...,
       [0.6372549 , 0.61764705, 0.59411764],
       [0.63529414, 0.6215686 , 0.5892157 ],
       [0.6401961 , 0.62058824, 0.59705883]],

      [[0.1        , 0.05539216, 0.          ],
       [0.48333332, 0.40882352, 0.29117647],
       [0.65        , 0.5686275 , 0.44019607],
       ...,
       [0.6308824 , 0.6161765 , 0.5808824 ],
       [0.6519608 , 0.63186276, 0.5901961 ],
       [0.6338235 , 0.6259804 , 0.57892156]]], dtype=float32)>
```

**Our custom image has a batch size of 1! Let's make a prediction on it.**

In [62]:

```
# Make a prediction on custom image tensor
pred = model_8.predict(steak)
pred
```

Out[62]:

```
array([[0.73311806]], dtype=float32)
```

**Ahh. the predictions come out in prediction probability form. In other words. this means how likely the image is**

to be one class or another.

Since we're working with a binary classification problem, if the prediction probability is over 0.5, according to the model, the prediction is most likely to be the **positive class** (class 1).

And if the prediction probability is under 0.5, according to the model, the predicted class is most likely to be the **negative class** (class 0).

**Note:** The 0.5 cutoff can be adjusted to your liking. For example, you could set the limit to be 0.8 and over for the positive class and 0.2 for the negative class. However, doing this will almost always change your model's performance metrics so be sure to make sure they change in the right direction.

But saying positive and negative class doesn't make much sense when we're working with pizza and steak ...

So let's write a little function to convert predictions into their class names and then plot the target image.

In [63]:

```
# Remind ourselves of our class names
class_names
```

Out[63]:

```
array(['.DS_Store', 'pizza', 'steak'], dtype='<U9')
```

In [64]:

```
# We can index the predicted class by rounding the prediction probability
pred_class = class_names[int(tf.round(pred)[0][0])]
pred_class
```

Out[64]:

```
'pizza'
```

In [65]:

```
def pred_and_plot(model, filename, class_names):
    """
    Imports an image located at filename, makes a prediction on it with
    a trained model and plots the image with the predicted class as the title.
    """
    # Import the target image and preprocess it
    img = load_and_prep_image(filename)

    # Make a prediction
    pred = model.predict(tf.expand_dims(img, axis=0))

    # Get the predicted class
    pred_class = class_names[int(tf.round(pred)[0][0])]

    # Plot the image and predicted class
    plt.imshow(img)
    plt.title(f"Prediction: {pred_class}")
    plt.axis(False);
```

In [66]:

```
# Test our model on a custom image
pred_and_plot(model_8, "03-steak.jpeg", class_names)
```

Prediction: pizza





Nice! Our model got the prediction right.

The only downside of working with food is this is making me hungry.

Let's try one more image.

In [67]:

```
# Download another test image and make a prediction on it
!wget https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/images/0
3-pizza-dad.jpeg
pred_and_plot(model_8, "03-pizza-dad.jpeg", class_names)

--2021-07-14 06:13:56-- https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-lear
ning/main/images/03-pizza-dad.jpeg
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.1
99.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443.
.. connected.
HTTP request sent, awaiting response... 200 OK
Length: 2874848 (2.7M) [image/jpeg]
Saving to: '03-pizza-dad.jpeg'

03-pizza-dad.jpeg    100%[=====] 2.74M --.-KB/s   in 0.02s

2021-07-14 06:13:57 (162 MB/s) - '03-pizza-dad.jpeg' saved [2874848/2874848]
```



Two thumbs up! Woohoo!

## Multi-class Classification

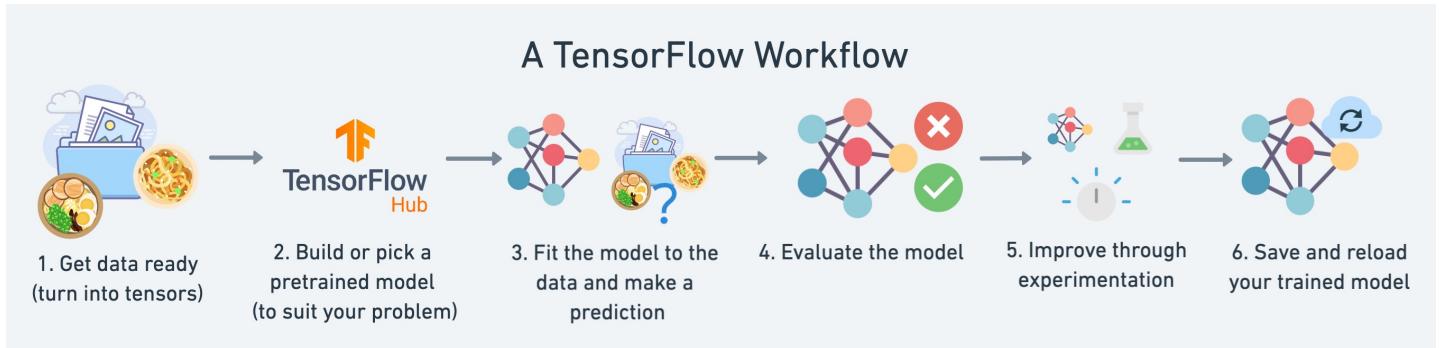
We've referenced the TinyVGG architecture from the CNN Explainer website multiple times through this notebook, however, the CNN Explainer website works with 10 different image classes, where as our current model only works with two classes (pizza and steak).

□ Practice: Before scrolling down, how do you think we might change our model to work with 10 classes of the same kind of images? Assume the data is in the same style as our two class problem.

Remember the steps we took before to build our pizza □ vs. steak □ classifier?

How about we go through those steps again, except this time, we'll work with 10 different types of food.

- 1. Become one with the data (visualize, visualize, visualize...)**
- 2. Preprocess the data (prepare it for a model)**
- 3. Create a model (start with a baseline)**
- 4. Fit the model**
- 5. Evaluate the model**
- 6. Adjust different parameters and improve model (try to beat your baseline)**
- 7. Repeat until satisfied**



*The workflow we're about to go through is a slightly modified version of the above image. As you keep going through deep learning problems, you'll find the workflow above is more of an outline than a step-by-step guide.*

## 1. Import and become one with the data

Again, we've got a subset of the [Food101 dataset](#). In addition to the pizza and steak images, we've pulled out another eight classes.

In [68]:

```
import zipfile

# Download zip file of 10_food_classes images
# See how this data was created - https://github.com/mrdbourke/tensorflow-deep-learning/blob/main/extras/image_data_modification.ipynb
!wget https://storage.googleapis.com/ztm_tf_course/food_vision/10_food_classes_all_data.zip

# Unzip the downloaded file
zip_ref = zipfile.ZipFile("10_food_classes_all_data.zip", "r")
zip_ref.extractall()
zip_ref.close()

--2021-07-14 06:13:57-- https://storage.googleapis.com/ztm_tf_course/food_vision/10_food_classes_all_data.zip
Resolving storage.googleapis.com (storage.googleapis.com)... 172.217.15.112, 142.250.65.8
0, 142.250.188.208, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|172.217.15.112|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 519183241 (495M) [application/zip]
Saving to: '10_food_classes_all_data.zip'

10_food_classes_all 100%[=====] 495.13M 239MB/s in 2.1s

2021-07-14 06:13:59 (239 MB/s) - '10_food_classes_all_data.zip' saved [519183241/519183241]
```

Now let's check out all of the different directories and sub-directories in the `10_food_classes` file.

In [69]:

```
import os

# Walk through 10_food_classes directory and list number of files
for dirpath, dirnames, filenames in os.walk("10_food_classes_all_data"):
```

```
print(f"There are {len(dirnames)} directories and {len(filenames)} images in '{dirpath}'\n'.'")

There are 2 directories and 0 images in '10_food_classes_all_data'.
There are 10 directories and 0 images in '10_food_classes_all_data/test'.
There are 0 directories and 250 images in '10_food_classes_all_data/test/chicken_curry'.
There are 0 directories and 250 images in '10_food_classes_all_data/test/ramen'.
There are 0 directories and 250 images in '10_food_classes_all_data/test/sushi'.
There are 0 directories and 250 images in '10_food_classes_all_data/test/grilled_salmon'.
There are 0 directories and 250 images in '10_food_classes_all_data/test/ice_cream'.
There are 0 directories and 250 images in '10_food_classes_all_data/test/chicken_wings'.
There are 0 directories and 250 images in '10_food_classes_all_data/test/pizza'.
There are 0 directories and 250 images in '10_food_classes_all_data/test/hamburger'.
There are 0 directories and 250 images in '10_food_classes_all_data/test/steak'.
There are 0 directories and 250 images in '10_food_classes_all_data/test/fried_rice'.
There are 10 directories and 0 images in '10_food_classes_all_data/train'.
There are 0 directories and 750 images in '10_food_classes_all_data/train/chicken_curry'.
There are 0 directories and 750 images in '10_food_classes_all_data/train/ramen'.
There are 0 directories and 750 images in '10_food_classes_all_data/train/sushi'.
There are 0 directories and 750 images in '10_food_classes_all_data/train/grilled_salmon'.
.
There are 0 directories and 750 images in '10_food_classes_all_data/train/ice_cream'.
There are 0 directories and 750 images in '10_food_classes_all_data/train/chicken_wings'.
There are 0 directories and 750 images in '10_food_classes_all_data/train/pizza'.
There are 0 directories and 750 images in '10_food_classes_all_data/train/hamburger'.
There are 0 directories and 750 images in '10_food_classes_all_data/train/steak'.
There are 0 directories and 750 images in '10_food_classes_all_data/train/fried_rice'.
```

**Looking good!**

**We'll now setup the training and test directory paths.**

In [70]:

```
train_dir = "10_food_classes_all_data/train/"
test_dir = "10_food_classes_all_data/test/"
```

**And get the class names from the subdirectories.**

In [71]:

```
# Get the class names for our multi-class dataset
import pathlib
import numpy as np
data_dir = pathlib.Path(train_dir)
class_names = np.array(sorted([item.name for item in data_dir.glob('*')]))
print(class_names)

['chicken_curry' 'chicken_wings' 'fried_rice' 'grilled_salmon' 'hamburger'
 'ice_cream' 'pizza' 'ramen' 'steak' 'sushi']
```

**How about we visualize an image from the training set?**

In [72]:

```
# View a random image from the training dataset
import random
img = view_random_image(target_dir=train_dir,
                        target_class=random.choice(class_names)) # get a random class name
```

Image shape: (384, 512, 3)

grilled\_salmon





## 2. Preprocess the data (prepare it for a model)

After going through a handful of images (it's good to visualize at least 10-100 different examples), it looks like our data directories are setup correctly.

Time to preprocess the data.

In [73]:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Rescale the data and create data generator instances
train_datagen = ImageDataGenerator(rescale=1/255.)
test_datagen = ImageDataGenerator(rescale=1/255.)

# Load data in from directories and turn it into batches
train_data = train_datagen.flow_from_directory(train_dir,
                                                target_size=(224, 224),
                                                batch_size=32,
                                                class_mode='categorical') # changed to categorical

test_data = test_datagen.flow_from_directory(test_dir,
                                              target_size=(224, 224),
                                              batch_size=32,
                                              class_mode='categorical')
```

Found 7500 images belonging to 10 classes.

Found 2500 images belonging to 10 classes.

As with binary classification, we've created image generators. The main change this time is that we've changed the `class_mode` parameter to `'categorical'` because we're dealing with 10 classes of food images.

Everything else like rescaling the images, creating the batch size and target image size stay the same.

Question: Why is the image size 224x224? This could actually be any size we wanted, however, 224x224 is a very common size for preprocessing images to. Depending on your problem you might want to use larger or smaller images.

## 3. Create a model (start with a baseline)

We can use the same model (TinyVGG) we used for the binary classification problem for our multi-class classification problem with a couple of small tweaks.

Namely:

- Changing the output layer to use have 10 output neurons (the same number as the number of classes we have).
- Changing the output layer to use `'softmax'` activation instead of `'sigmoid'` activation.
- Changing the loss function to be `'categorical_crossentropy'` instead of `'binary_crossentropy'`.

In [74]:

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
```

```

from tensorflow.keras.layers import Conv2D, MaxPool2D, Flatten, Dense

# Create our model (a clone of model_8, except to be multi-class)
model_9 = Sequential([
    Conv2D(10, 3, activation='relu', input_shape=(224, 224, 3)),
    Conv2D(10, 3, activation='relu'),
    MaxPool2D(),
    Conv2D(10, 3, activation='relu'),
    Conv2D(10, 3, activation='relu'),
    MaxPool2D(),
    Flatten(),
    Dense(10, activation='softmax') # changed to have 10 neurons (same as number of classes
) and 'softmax' activation
])

# Compile the model
model_9.compile(loss="categorical_crossentropy", # changed to categorical_crossentropy
                 optimizer=tf.keras.optimizers.Adam(),
                 metrics=["accuracy"])

```

## 4. Fit a model

**Now we've got a model suited for working with multiple classes, let's fit it to our data.**

In [75]:

```

# Fit the model
history_9 = model_9.fit(train_data, # now 10 different classes
                        epochs=5,
                        steps_per_epoch=len(train_data),
                        validation_data=test_data,
                        validation_steps=len(test_data))

```

```

Epoch 1/5
235/235 [=====] - 45s 192ms/step - loss: 2.2776 - accuracy: 0.13
69 - val_loss: 2.2468 - val_accuracy: 0.1804
Epoch 2/5
235/235 [=====] - 43s 185ms/step - loss: 2.1532 - accuracy: 0.21
81 - val_loss: 2.1431 - val_accuracy: 0.2500
Epoch 3/5
235/235 [=====] - 43s 185ms/step - loss: 1.7899 - accuracy: 0.39
23 - val_loss: 2.0544 - val_accuracy: 0.2948
Epoch 4/5
235/235 [=====] - 43s 185ms/step - loss: 1.1089 - accuracy: 0.63
27 - val_loss: 2.5314 - val_accuracy: 0.2708
Epoch 5/5
235/235 [=====] - 44s 186ms/step - loss: 0.4284 - accuracy: 0.86
53 - val_loss: 3.7439 - val_accuracy: 0.2484

```

**Why do you think each epoch takes longer than when working with only two classes of images?**

**It's because we're now dealing with more images than we were before. We've got 10 classes with 750 training images and 250 validation images each totalling 10,000 images. Where as when we had two classes, we had 1500 training images and 500 validation images, totalling 2000.**

**The intuitive reasoning here is the more data you have, the longer a model will take to find patterns.**

## 5. Evaluate the model

**Woohoo! We've just trained a model on 10 different classes of food images, let's see how it went.**

In [76]:

```

# Evaluate on the test data
model_9.evaluate(test_data)

```

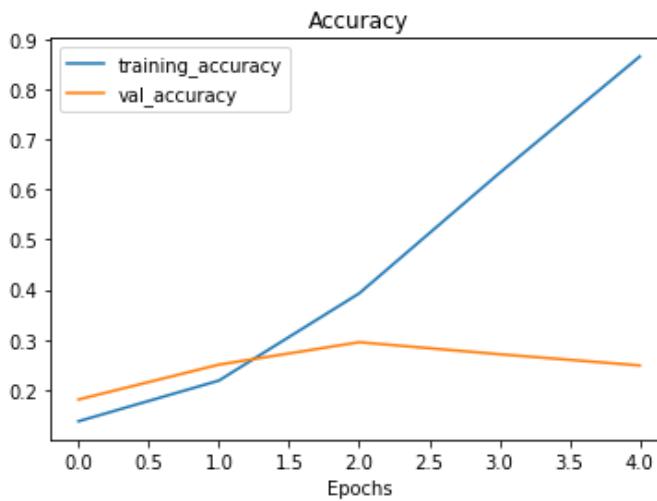
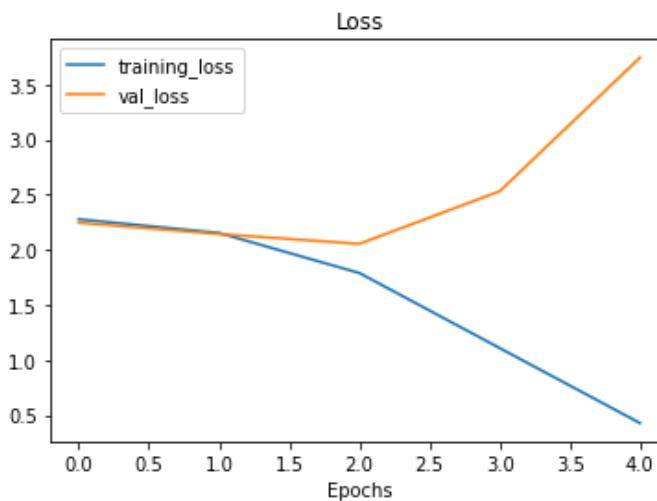
```

79/79 [=====] - 10s 129ms/step - loss: 3.7439 - accuracy: 0.2484

```

In [77]:

```
# Check out the model's loss curves on the 10 classes of data (note: this function comes from above in the notebook)
plot_loss_curves(history_9)
```



Woah, that's quite the gap between the training and validation loss curves.

What does this tell us?

It seems our model is **overfitting** the training set quite badly. In other words, it's getting great results on the training data but fails to generalize well to unseen data and performs poorly on the test data.

## 6. Adjust the model parameters

Due to its performance on the training data, it's clear our model is learning something. However, performing well on the training data is like going well in the classroom but failing to use your skills in real life.

Ideally, we'd like our model to perform as well on the test data as it does on the training data.

So our next steps will be to try and prevent our model overfitting. A couple of ways to prevent overfitting include:

- **Get more data** - Having more data gives the model more opportunities to learn patterns, patterns which may be more generalizable to new examples.
- **Simplify model** - If the current model is already overfitting the training data, it may be too complicated of a model. This means it's learning the patterns of the data too well and isn't able to generalize well to unseen data. One way to simplify a model is to reduce the number of layers it uses or to reduce the number of hidden units in each layer.
- **Use data augmentation** - Data augmentation manipulates the training data in a way so that's harder for the model to learn as it artificially adds more variety to the data. If a model is able to learn patterns in

augmented data, the model may be able to generalize better to unseen data.

- **Use transfer learning** - Transfer learning involves leverages the patterns (also called pretrained weights) one model has learned to use as the foundation for your own task. In our case, we could use one computer vision model pretrained on a large variety of images and then tweak it slightly to be more specialized for food images.

■ Note: Preventing overfitting is also referred to as **regularization**.

If you've already got an existing dataset, you're probably most likely to try one or a combination of the last three above options first.

Since collecting more data would involve us manually taking more images of food, let's try the ones we can do from right within the notebook.

How about we simplify our model first?

To do so, we'll remove two of the convolutional layers, taking the total number of convolutional layers from four to two.

In [78]:

```
# Try a simplified model (removed two layers)
model_10 = Sequential([
    Conv2D(10, 3, activation='relu', input_shape=(224, 224, 3)),
    MaxPool2D(),
    Conv2D(10, 3, activation='relu'),
    MaxPool2D(),
    Flatten(),
    Dense(10, activation='softmax')
])

model_10.compile(loss='categorical_crossentropy',
                  optimizer=tf.keras.optimizers.Adam(),
                  metrics=['accuracy'])

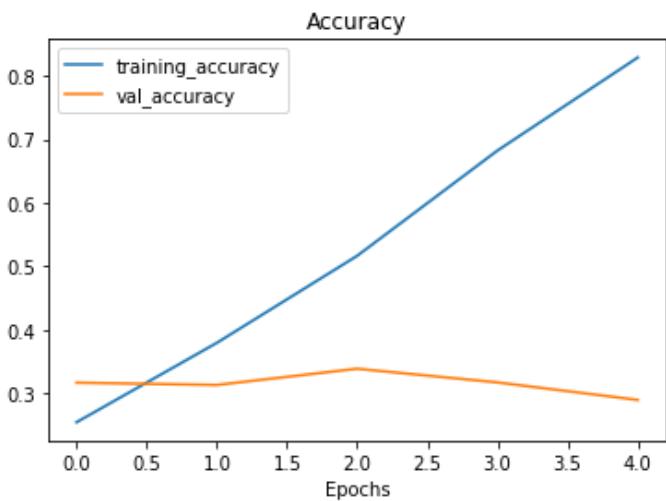
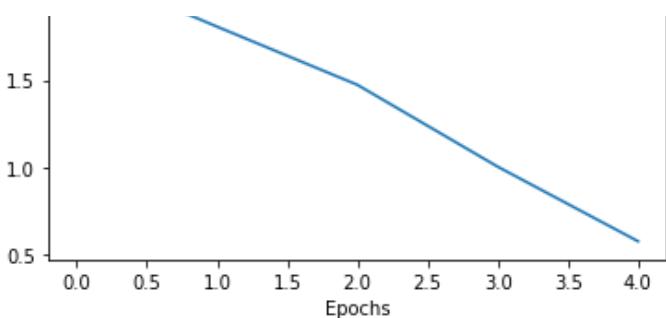
history_10 = model_10.fit(train_data,
                           epochs=5,
                           steps_per_epoch=len(train_data),
                           validation_data=test_data,
                           validation_steps=len(test_data))
```

```
Epoch 1/5
235/235 [=====] - 41s 175ms/step - loss: 2.1487 - accuracy: 0.25
40 - val_loss: 1.9855 - val_accuracy: 0.3164
Epoch 2/5
235/235 [=====] - 41s 174ms/step - loss: 1.8072 - accuracy: 0.37
92 - val_loss: 1.9812 - val_accuracy: 0.3128
Epoch 3/5
235/235 [=====] - 42s 177ms/step - loss: 1.4728 - accuracy: 0.51
64 - val_loss: 1.9150 - val_accuracy: 0.3384
Epoch 4/5
235/235 [=====] - 41s 176ms/step - loss: 1.0023 - accuracy: 0.68
24 - val_loss: 2.1659 - val_accuracy: 0.3168
Epoch 5/5
235/235 [=====] - 41s 177ms/step - loss: 0.5737 - accuracy: 0.82
91 - val_loss: 2.6703 - val_accuracy: 0.2892
```

In [79]:

```
# Check out the loss curves of model_10
plot_loss_curves(history_10)
```





Hmm... even with a simplified model, it looks like our model is still dramatically overfitting the training data.

What else could we try?

How about data augmentation?

Data augmentation makes it harder for the model to learn on the training data and in turn, hopefully making the patterns it learns more generalizable to unseen data.

To create augmented data, we'll recreate a new `ImageDataGenerator` instance, this time adding some parameters such as `rotation_range` and `horizontal_flip` to manipulate our images.

In [80]:

```
# Create augmented data generator instance
train_datagen_augmented = ImageDataGenerator(rescale=1/255.,
                                              rotation_range=20, # note: this is an int not a float
                                              width_shift_range=0.2,
                                              height_shift_range=0.2,
                                              zoom_range=0.2,
                                              horizontal_flip=True)

train_data_augmented = train_datagen_augmented.flow_from_directory(train_dir,
                                                                target_size=(224, 224),
                                                                batch_size=32,
                                                                class_mode='categorical')
```

Found 7500 images belonging to 10 classes.

Now we've got augmented data, let's see how it works with the same model as before (`model_10`).

Rather than rewrite the model from scratch, we can clone it using a handy function in TensorFlow called `clone_model` which can take an existing model and rebuild it in the same format.

The cloned version will not include any of the weights (patterns) the original model has learned. So when we train it, it'll be like training a model from scratch.

↳ **Note:** One of the key practices in deep learning and machine learning in general is to be a serial experimenter. That's what we're doing here. Trying something, seeing if it works, then trying something else. A good experiment setup also keeps track of the things you change, for example, that's why we're using the same model as before but with different data. The model stays the same but the data changes, this will let us know if augmented training data has any influence over performance.

In [81]:

```
# Clone the model (use the same architecture)
model_11 = tf.keras.models.clone_model(model_10)

# Compile the cloned model (same setup as used for model_10)
model_11.compile(loss="categorical_crossentropy",
                  optimizer=tf.keras.optimizers.Adam(),
                  metrics=["accuracy"])

# Fit the model
history_11 = model_11.fit(train_data_augmented, # use augmented data
                           epochs=5,
                           steps_per_epoch=len(train_data_augmented),
                           validation_data=test_data,
                           validation_steps=len(test_data))
```

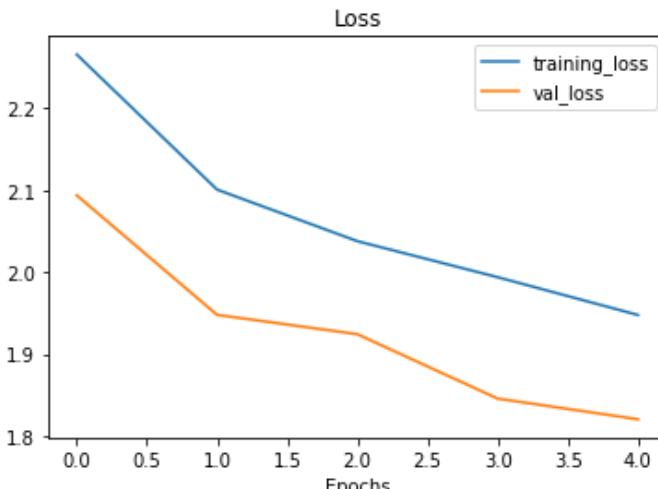
```
Epoch 1/5
235/235 [=====] - 105s 446ms/step - loss: 2.2657 - accuracy: 0.1
693 - val_loss: 2.0938 - val_accuracy: 0.2512
Epoch 2/5
235/235 [=====] - 104s 444ms/step - loss: 2.1007 - accuracy: 0.2
479 - val_loss: 1.9478 - val_accuracy: 0.3204
Epoch 3/5
235/235 [=====] - 104s 444ms/step - loss: 2.0377 - accuracy: 0.2
851 - val_loss: 1.9241 - val_accuracy: 0.3280
Epoch 4/5
235/235 [=====] - 104s 444ms/step - loss: 1.9937 - accuracy: 0.3
097 - val_loss: 1.8455 - val_accuracy: 0.3736
Epoch 5/5
235/235 [=====] - 104s 443ms/step - loss: 1.9476 - accuracy: 0.3
291 - val_loss: 1.8203 - val_accuracy: 0.3664
```

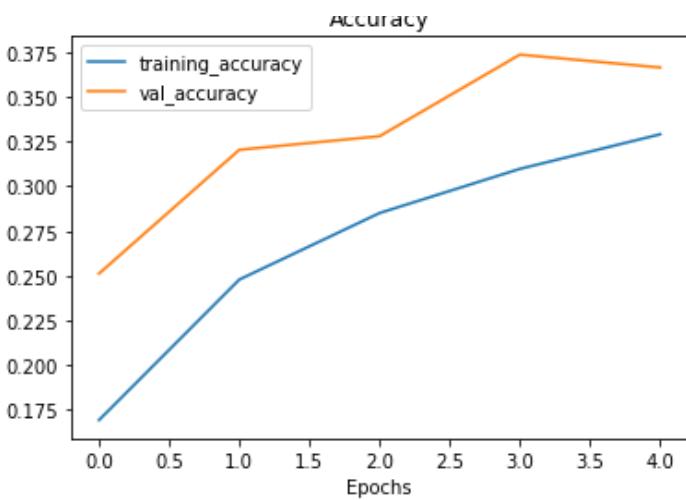
You can see it each epoch takes longer than the previous model. This is because our data is being augmented on the fly on the CPU as it gets loaded onto the GPU, in turn, increasing the amount of time between each epoch.

How do our model's training curves look?

In [82]:

```
# Check out our model's performance with augmented data
plot_loss_curves(history_11)
```





Woah! That's looking much better, the loss curves are much closer to eachother. Although our model didn't perform as well on the augmented training set, it performed much better on the validation dataset.

It even looks like if we kept it training for longer (more epochs) the evaluation metrics might continue to improve.



## 7. Repeat until satisfied

We could keep going here. Restructuring our model's architecture, adding more layers, trying it out, adjusting the learning rate, trying it out, trying different methods of data augmentation, training for longer. But as you could image, this could take a fairly long time.

Good thing there's still one trick we haven't tried yet and that's **transfer learning**.

However, we'll save that for the next notebook where you'll see how rather than design our own models from scratch we leverage the patterns another model has learned for our own task.

In the meantime, let's make a prediction with our trained multi-class model.

## Making a prediction with our trained model

What good is a model if you can't make predictions with it?

Let's first remind ourselves of the classes our multi-class model has been trained on and then we'll download some of own custom images to work with.

In [83]:

```
# What classes has our model been trained on?
class_names
```

Out [83]:

```
array(['chicken_curry', 'chicken_wings', 'fried_rice', 'grilled_salmon',
       'hamburger', 'ice_cream', 'pizza', 'ramen', 'steak', 'sushi'],
      dtype='|<U14')
```

Beautiful, now let's get some of our custom images.

If you're using Google Colab, you could also upload some of your own images via the files tab.

In [84]:

```
# -q is for "quiet"
!wget -q https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/images/03-pizza-dad.jpeg
!wget -q https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/images/03-steak.jpeg
!wget -q https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/images/03-salmon.jpeg
```

```
s/03-hamburger.jpeg  
!wget -q https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/images/03-sushi.jpeg
```

Okay, we've got some custom images to try, let's use the `pred_and_plot` function to make a prediction with `model_11` on one of the images and plot it.

In [85]:

```
# Make a prediction using model_11  
pred_and_plot(model=model_11,  
               filename="03-steak.jpeg",  
               class_names=class_names)
```

Prediction: chicken\_curry



Hmm... it looks like our model got the prediction wrong, how about we try another?

In [86]:

```
pred_and_plot(model_11, "03-sushi.jpeg", class_names)
```

Prediction: chicken\_curry



And again, it's predicting `chicken_curry` for some reason.

How about one more?

In [87]:

```
pred_and_plot(model_11, "03-pizza-dad.jpeg", class_names)
```

Prediction: chicken\_curry





chicken\_curry again? There must be something wrong...

I think it might have to do with our `pred_and_plot` function.

Let's makes a prediction without using the function and see where it might be going wrong.

In [88]:

```
# Load in and preprocess our custom image
img = load_and_prep_image("03-steak.jpeg")

# Make a prediction
pred = model_11.predict(tf.expand_dims(img, axis=0))

# Match the prediction class to the highest prediction probability
pred_class = class_names[pred.argmax()]
plt.imshow(img)
plt.title(pred_class)
plt.axis(False);
```

steak



Much better! There must be something up with our `pred_and_plot` function.

And I think I know what it is.

The `pred_and_plot` function was designed to be used with binary classification models where as our current model is a multi-class classification model.

The main difference lies in the output of the `predict` function.

In [89]:

```
# Check the output of the predict function
pred = model_11.predict(tf.expand_dims(img, axis=0))
pred
```

Out [89]:

```
array([[0.03887467, 0.15843102, 0.03170868, 0.1488752 , 0.0594982 ,
       0.04778374, 0.06917461, 0.04166124, 0.27236295, 0.13162972]],  
      dtype=float32)
```

Since our model has a '`softmax`' activation function and 10 output neurons, it outputs a prediction probability for each of the classes in our model.

The class with the highest probability is what the model believes the image contains.

We can find the maximum value index using `argmax` and then use that to index our `class_names` list to output the predicted class.

Output the predicted class.

In [90]:

```
# Find the predicted class name
class_names[pred.argmax()]
```

Out[90]:

```
'steak'
```

**Knowing this, we can readjust our `pred_and_plot` function to work with multiple classes as well as binary classes.**

In [91]:

```
# Adjust function to work with multi-class
def pred_and_plot(model, filename, class_names):
    """
    Imports an image located at filename, makes a prediction on it with
    a trained model and plots the image with the predicted class as the title.
    """
    # Import the target image and preprocess it
    img = load_and_prep_image(filename)

    # Make a prediction
    pred = model.predict(tf.expand_dims(img, axis=0))

    # Get the predicted class
    if len(pred[0]) > 1: # check for multi-class
        pred_class = class_names[pred.argmax()] # if more than one output, take the max
    else:
        pred_class = class_names[int(tf.round(pred)[0][0])] # if only one output, round

    # Plot the image and predicted class
    plt.imshow(img)
    plt.title(f"Prediction: {pred_class}")
    plt.axis(False);
```

**Let's try it out. If we've done it right, using different images should lead to different outputs (rather than chicken\_curry every time).**

In [92]:

```
pred_and_plot(model_11, "03-steak.jpeg", class_names)
```

Prediction: steak



In [93]:

```
pred_and_plot(model_11, "03-sushi.jpeg", class_names)
```

Prediction: chicken\_curry





In [94]:

```
pred_and_plot(model_11, "03-pizza-dad.jpeg", class_names)
```

Prediction: ice\_cream



In [95]:

```
pred_and_plot(model_11, "03-hamburger.jpeg", class_names)
```

Prediction: sushi



Our model's predictions aren't very good, this is because it's only performing at ~35% accuracy on the test dataset.

## Saving and loading our model

Once you've trained a model, you probably want to be able to save it and load it somewhere else.

To do so, we can use the `save` and `load_model` functions.

In [96]:

```
# Save a model
model_11.save("saved_trained_model")
```

INFO:tensorflow:Assets written to: saved\_trained\_model/assets

In [97]:

```
# Load in a model and evaluate it
```

```
loaded_model_11 = tf.keras.models.load_model("saved_trained_model")
loaded_model_11.evaluate(test_data)

79/79 [=====] - 10s 127ms/step - loss: 1.8203 - accuracy: 0.3664

Out[97]:
[1.8202669620513916, 0.36640000343322754]

In [98]:
# Compare our unsaved model's results (same as above)
model_11.evaluate(test_data)

79/79 [=====] - 10s 128ms/step - loss: 1.8203 - accuracy: 0.3664

Out[98]:
[1.8202663660049438, 0.36640000343322754]
```

## Exercises

1. Spend 20-minutes reading and interacting with the [CNN explainer website](#).
  - What are the key terms? e.g. explain convolution in your own words, pooling in your own words
2. Play around with the "understanding hyperparameters" section in the [CNN explainer](#) website for 10-minutes.
  - What is the kernel size?
  - What is the stride?
  - How could you adjust each of these in TensorFlow code?
3. Take 10 photos of two different things and build your own CNN image classifier using the techniques we've built here.
4. Find an ideal learning rate for a simple convolutional neural network model on your the 10 class dataset.

## Extra-curriculum

1. Watch: [MIT's Introduction to Deep Computer Vision](#) lecture. This will give you a great intuition behind convolutional neural networks.
2. Watch: Deep dive on [mini-batch gradient descent](#) by deeplearning.ai. If you're still curious about why we use batches to train models, this technical overview covers many of the reasons why.
3. Read: [CS231n Convolutional Neural Networks for Visual Recognition](#) class notes. This will give a very deep understanding of what's going on behind the scenes of the convolutional neural network architectures we're writing.
4. Read: "[A guide to convolution arithmetic for deep learning](#)". This paper goes through all of the mathematics running behind the scenes of our convolutional layers.
5. Code practice: [TensorFlow Data Augmentation Tutorial](#). For a more in-depth introduction on data augmentation with TensorFlow, spend an hour or two reading through this tutorial.

## 04. Transfer Learning with TensorFlow Part 1: Feature Extraction

We've built a bunch of convolutional neural networks from scratch and they all seem to be learning, however, there is still plenty of room for improvement.

To improve our model(s), we could spend a while trying different configurations, adding more layers, changing the learning rate, adjusting the number of neurons per layer and more.

However, doing this is very time consuming.

Luckily, there's a technique we can use to save time.

It's called **transfer learning**, in other words, taking the patterns (also called weights) another model has learned from another problem and using them for our own problem.

There are two main benefits to using transfer learning:

1. Can leverage an existing neural network architecture proven to work on problems similar to our own.
2. Can leverage a working neural network architecture which has **already learned** patterns on similar data to our own. This often results in achieving great results with less custom data.

What this means is, instead of hand-crafting our own neural network architectures or building them from scratch, we can utilise models which have worked for others.

And instead of training our own models from scratch on our own datasets, we can take the patterns a model has learned from datasets such as [ImageNet](#) (millions of images of different objects) and use them as the foundation of our own. Doing this often leads to getting great results with less data.

Over the next few notebooks, we'll see the power of transfer learning in action.

### What we're going to cover

We're going to go through the following with TensorFlow:

- Introduce transfer learning (a way to beat all of our old self-built models)
- Using a smaller dataset to experiment faster (10% of training samples of 10 classes of food)
- Build a transfer learning feature extraction model using TensorFlow Hub
- Introduce the TensorBoard callback to track model training results
- Compare model results using TensorBoard

### How you can use this notebook

You can read through the descriptions and the code (it should all run, except for the cells which error on purpose), but there's a better option.

Write all of the code yourself.

Yes. I'm serious. Create a new notebook, and rewrite each line by yourself. Investigate it, see if you can break it, why does it break?

You don't have to write the text descriptions but writing the code yourself is a great way to get hands-on experience.

Don't worry if you make mistakes, we all do. The way to get better and make less mistakes is to **write more code**.

## Using a GPU

To begin, let's check to see if we're using a GPU. Using a GPU will make sure our model trains faster than using just a CPU.

In [ ]:

```
# Are we using a GPU?  
!nvidia-smi
```

```
Fri Feb 12 03:39:41 2021  
+-----+  
| NVIDIA-SMI 460.39      Driver Version: 460.32.03    CUDA Version: 11.2 |  
+-----+  
| GPU  Name      Persistence-M| Bus-Id     Disp.A  | Volatile Uncorr. ECC | | | |
| Fan  Temp     Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |  
|          |          |          |          |          |          MIG M. |  
+=====+=====+=====+=====+=====+=====+  
| 0  Tesla T4        Off  | 00000000:00:04.0 Off |                0 | | | |
| N/A   37C     P8    9W / 70W |        0MiB / 15109MiB |      0%     Default |  
|          |          |          |          |          |          N/A |  
+-----+-----+-----+-----+-----+  
  
+-----+  
| Processes:  
| GPU  GI  CI      PID  Type  Process name          GPU Memory |  
|          ID  ID          |          |          |          Usage |  
+=====+=====+=====+=====+=====+=====+  
| No running processes found  
+-----+
```

If the cell above doesn't output something which looks like:

```
Fri Sep  4 03:35:21 2020  
+-----+  
| NVIDIA-SMI 450.66      Driver Version: 418.67      CUDA Version: 10.1 |  
+-----+  
| GPU  Name      Persistence-M| Bus-Id     Disp.A  | Volatile Uncorr. ECC | | | |
| Fan  Temp     Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |  
|          |          |          |          |          |          MIG M. |  
+=====+=====+=====+=====+=====+=====+  
| 0  Tesla P100-PCIE... Off  | 00000000:00:04.0 Off |                0 | | | |
| N/A   35C     P0    26W / 250W |        0MiB / 16280MiB |      0%     Default |  
|          |          |          |          |          |          ERR! |  
+-----+-----+-----+-----+-----+  
  
+-----+  
| Processes:  
| GPU  GI  CI      PID  Type  Process name          GPU Memory |  
|          ID  ID          |          |          |          Usage |  
+=====+=====+=====+=====+=====+=====+  
| No running processes found  
+-----+
```

Go to Runtime -> Change Runtime Type -> Hardware Accelerator and select "GPU", then rerun the cell above.

## Transfer learning with TensorFlow Hub: Getting great results with 10% of the data

If you've been thinking, "surely someone else has spent the time crafting the right model for the job..." then you're in luck.

For many of the problems you'll want to use deep learning for, chances are, a working model already exists.

And the good news is, you can access many of them on TensorFlow Hub.

[TensorFlow Hub](#) is a repository for existing model components. It makes it so you can import and use a fully trained model with as little as a URL.

Now, I really want to demonstrate the power of transfer learning to you.

To do so, what if I told you we could get much of the same results (or better) than our best model has gotten so far with only 10% of the original data, in other words, 10x less data.

This seems counterintuitive right?

Wouldn't you think more examples of what a picture of food looked like led to better results?

And you'd be right if you thought so, generally, more data leads to better results.

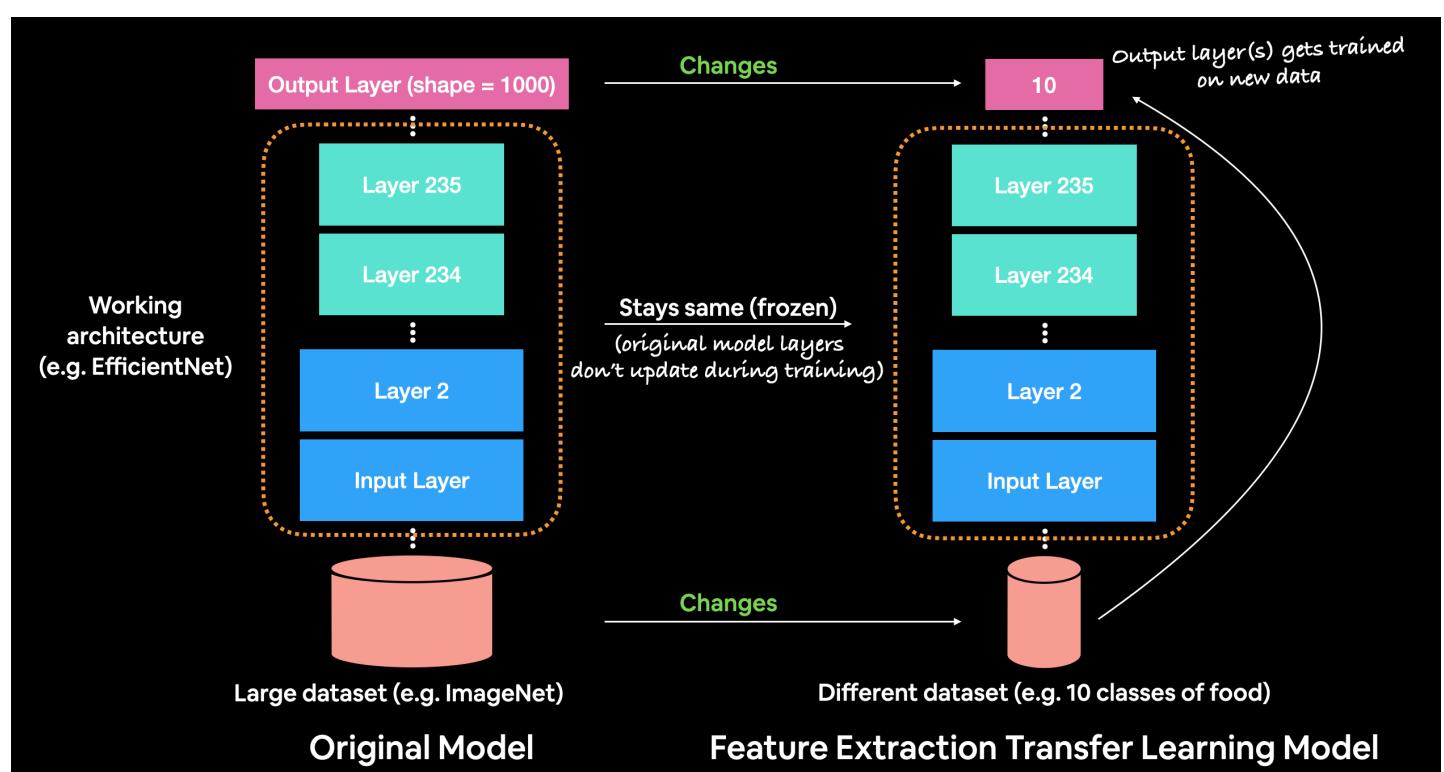
However, what if you didn't have more data? What if instead of 750 images per class, you had 75 images per class?

Collecting 675 more images of a certain class could take a long time.

So this is where another major benefit of transfer learning comes in.

**Transfer learning often allows you to get great results with less data.**

But don't just take my word for it. Let's download a subset of the data we've been using, namely 10% of the training data from the `10_food_classes` dataset and use it to train a food image classifier on.



*What we're working towards building. Taking a pre-trained model and adding our own custom layers on top, extracting all of the underlying patterns learned on another dataset our own images.*

## Downloading and becoming one with the data

In [ ]:

```
# Get data (10% of labels)
import zipfile

# Download data
!wget https://storage.googleapis.com/ztm_tf_course/food_vision/10_food_classes_10_percent.zip

# Unzip the downloaded file
zip_ref = zipfile.ZipFile("10_food_classes_10_percent.zip", "r")
zip_ref.extractall()
```

```
zip_ref.close()

--2021-02-12 03:39:42-- https://storage.googleapis.com/ztm_tf_course/food_vision/10_food_classes_10_percent.zip
Resolving storage.googleapis.com (storage.googleapis.com) ... 172.217.15.112, 172.253.62.1
28, 142.250.31.128, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|172.217.15.112|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 168546183 (161M) [application/zip]
Saving to: '10_food_classes_10_percent.zip'

10_food_classes_10_ 100%[=====] 160.74M 172MB/s in 0.9s

2021-02-12 03:39:43 (172 MB/s) - '10_food_classes_10_percent.zip' saved [168546183/168546183]
```

In [ ]:

```
# How many images in each folder?
import os

# Walk through 10 percent data directory and list number of files
for dirpath, dirnames, filenames in os.walk("10_food_classes_10_percent"):
    print(f"There are {len(dirnames)} directories and {len(filenames)} images in '{dirpath}'")
```

There are 2 directories and 0 images in '10\_food\_classes\_10\_percent'.  
There are 10 directories and 0 images in '10\_food\_classes\_10\_percent/test'.  
There are 0 directories and 250 images in '10\_food\_classes\_10\_percent/test/fried\_rice'.  
There are 0 directories and 250 images in '10\_food\_classes\_10\_percent/test/sushi'.  
There are 0 directories and 250 images in '10\_food\_classes\_10\_percent/test/chicken\_curry'.  
. .  
There are 0 directories and 250 images in '10\_food\_classes\_10\_percent/test/pizza'.  
There are 0 directories and 250 images in '10\_food\_classes\_10\_percent/test/ramen'.  
There are 0 directories and 250 images in '10\_food\_classes\_10\_percent/test/steak'.  
There are 0 directories and 250 images in '10\_food\_classes\_10\_percent/test/ice\_cream'.  
There are 0 directories and 250 images in '10\_food\_classes\_10\_percent/test/grilled\_salmon'.  
. .  
There are 0 directories and 250 images in '10\_food\_classes\_10\_percent/test/hamburger'.  
There are 0 directories and 250 images in '10\_food\_classes\_10\_percent/test/chicken\_wings'.  
. .  
There are 10 directories and 0 images in '10\_food\_classes\_10\_percent/train'.  
There are 0 directories and 75 images in '10\_food\_classes\_10\_percent/train/fried\_rice'.  
There are 0 directories and 75 images in '10\_food\_classes\_10\_percent/train/sushi'.  
There are 0 directories and 75 images in '10\_food\_classes\_10\_percent/train/chicken\_curry'.  
. .  
There are 0 directories and 75 images in '10\_food\_classes\_10\_percent/train/pizza'.  
There are 0 directories and 75 images in '10\_food\_classes\_10\_percent/train/ramen'.  
There are 0 directories and 75 images in '10\_food\_classes\_10\_percent/train/steak'.  
There are 0 directories and 75 images in '10\_food\_classes\_10\_percent/train/ice\_cream'.  
There are 0 directories and 75 images in '10\_food\_classes\_10\_percent/train/grilled\_salmon'.  
. .  
There are 0 directories and 75 images in '10\_food\_classes\_10\_percent/train/hamburger'.  
There are 0 directories and 75 images in '10\_food\_classes\_10\_percent/train/chicken\_wings'.  
. .

**Notice how each of the training directories now has 75 images rather than 750 images. This is key to demonstrating how well transfer learning can perform with less labelled images.**

**The test directories still have the same amount of images. This means we'll be training on less data but evaluating our models on the same amount of test data.**

## Creating data loaders (preparing the data)

**Now we've downloaded the data, let's use the `ImageDataGenerator` class along with the `flow_from_directory` method to load in our images.**

In [ ]:

```
# Setup data inputs
from tensorflow.keras.preprocessing.image import ImageDataGenerator

IMAGE_SHAPE = (224, 224)
BATCH_SIZE = 32

train_dir = "10_food_classes_10_percent/train/"
test_dir = "10_food_classes_10_percent/test/"

train_datagen = ImageDataGenerator(rescale=1/255.)
test_datagen = ImageDataGenerator(rescale=1/255.)

print("Training images:")
train_data_10_percent = train_datagen.flow_from_directory(train_dir,
                                                          target_size=IMAGE_SHAPE,
                                                          batch_size=BATCH_SIZE,
                                                          class_mode="categorical")

print("Testing images:")
test_data = test_datagen.flow_from_directory(test_dir,
                                              target_size=IMAGE_SHAPE,
                                              batch_size=BATCH_SIZE,
                                              class_mode="categorical")
```

Training images:

Found 750 images belonging to 10 classes.

Testing images:

Found 2500 images belonging to 10 classes.

**Excellent! Loading in the data we can see we've got 750 images in the training dataset belonging to 10 classes (75 per class) and 2500 images in the test set belonging to 10 classes (250 per class).**

## Setting up callbacks (things to run whilst our model trains)

Before we build a model, there's an important concept we're going to get familiar with because it's going to play a key role in our future model building experiments.

And that concept is **callbacks**.

**Callbacks** are extra functionality you can add to your models to be performed during or after training. Some of the most popular callbacks include:

- **Experiment tracking with TensorBoard** - log the performance of multiple models and then view and compare these models in a visual way on [TensorBoard](#) (a dashboard for inspecting neural network parameters). Helpful to compare the results of different models on your data.
- **Model checkpointing** - save your model as it trains so you can stop training if needed and come back to continue off where you left. Helpful if training takes a long time and can't be done in one sitting.
- **Early stopping** - leave your model training for an arbitrary amount of time and have it stop training automatically when it ceases to improve. Helpful when you've got a large dataset and don't know how long training will take.

We'll explore each of these overtime but for this notebook, we'll see how the TensorBoard callback can be used.

The TensorBoard callback can be accessed using `tf.keras.callbacks.TensorBoard()`.

Its main functionality is saving a model's training performance metrics to a specified `log_dir`.

By default, logs are recorded every epoch using the `update_freq='epoch'` parameter. This is a good default since tracking model performance too often can slow down model training.

To track our modelling experiments using TensorBoard, let's create a function which creates a TensorBoard callback for us.

**Note:** We create a function for creating a TensorBoard callback because as we'll see later on,

each model needs its own TensorBoard callback instance (so the function will create a new one each time it's run).

In [ ]:

```
# Create tensorboard callback (functionized because need to create a new one for each model)
import datetime
def create_tensorboard_callback(dir_name, experiment_name):
    log_dir = dir_name + "/" + experiment_name + "/" + datetime.datetime.now().strftime("%Y-%m-%d-%H%M%S")
    tensorboard_callback = tf.keras.callbacks.TensorBoard(
        log_dir=log_dir
    )
    print(f"Saving TensorBoard log files to: {log_dir}")
    return tensorboard_callback
```

Because you're likely to run multiple experiments, it's a good idea to be able to track them in some way.

In our case, our function saves a model's performance logs to a directory named

[dir\_name] / [experiment\_name] / [current\_timestamp], where:

- `dir_name` is the overall logs directory
- `experiment_name` is the particular experiment
- `current_timestamp` is the time the experiment started based on Python's `datetime.datetime().now()`

□ Note: Depending on your use case, the above experimenting tracking naming method may work or you might require something more specific. The good news is, the TensorBoard callback makes it easy to track modelling logs as long as you specify where to track them. So you can get as creative as you like with how you name your experiments, just make sure you or your team can understand them.

## Creating models using TensorFlow Hub

In the past we've used TensorFlow to create our own models layer by layer from scratch.

Now we're going to do a similar process, except the majority of our model's layers are going to come from [TensorFlow Hub](#).

In fact, we're going to use two models from TensorFlow Hub:

1. [ResNetV2](#) - a state of the art computer vision model architecture from 2016.
2. [EfficientNet](#) - a state of the art computer vision architecture from 2019.

State of the art means that at some point, both of these models have achieved the lowest error rate on [ImageNet \(ILSVRC-2012-CLS\)](#), the gold standard of computer vision benchmarks.

You might be wondering, how do you find these models on TensorFlow Hub?

Here are the steps I took:

1. Go to [tfhub.dev](#).
2. Choose your problem domain, e.g. "Image" (we're using food images).
3. Select your TF version, which in our case is TF2.
4. Remove all "Problem domain" filters except for the problem you're working on.
  - Note: "Image feature vector" can be used alongside almost any problem, we'll get to this soon.
5. The models listed are all models which could potentially be used for your problem.

□ Question: I see many options for image classification models, how do I know which is best?

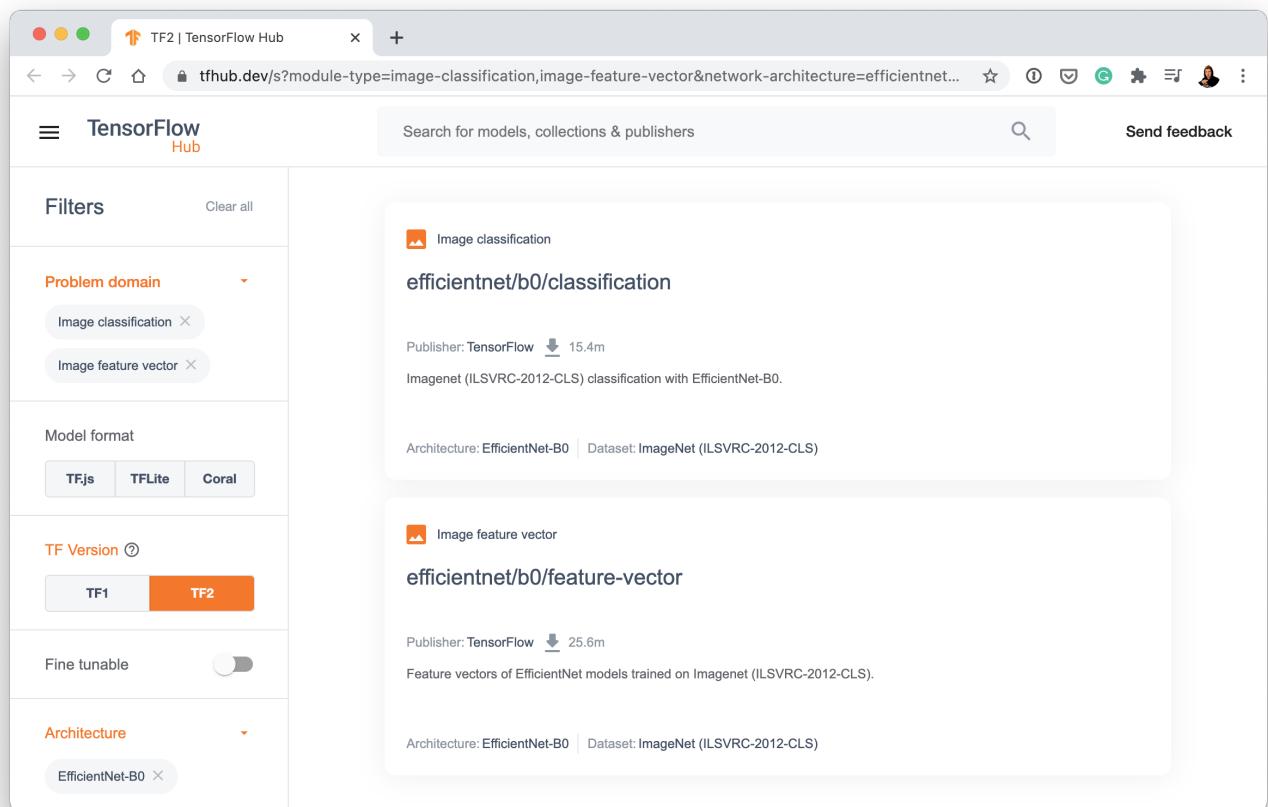
You can see a list of state-of-the-art models on [paperswithcode.com](http://paperswithcode.com), a resource for collecting the latest in deep learning paper results which have code implementations for the findings they report.

Since we're working with images, our target are the [models which perform best on ImageNet](#).

You'll probably find not all of the model architectures listed on paperswithcode appear on TensorFlow Hub. And this is okay, we can still use what's available.

To find our models, let's narrow down our search using the Architecture tab.

1. Select the Architecture tab on TensorFlow Hub and you'll see a dropdown menu of architecture names appear.
  - The rule of thumb here is generally, names with larger numbers mean better performing models. For example, EfficientNetB4 performs better than EfficientNetB0.
    - However, the tradeoff with larger numbers can mean they take longer to compute.
2. Select EfficientNetB0 and you should see [something like the following](#):



3. Clicking the one titled "[efficientnet/b0/feature-vector](#)" brings us to a page with a button that says "Copy URL". That URL is what we can use to harness the power of EfficientNetB0.

- Copying the URL should give you something like this:  
<https://tfhub.dev/tensorflow/efficientnet/b0/feature-vector/1>

**Question:** I thought we were doing *image classification*, why do we choose *feature vector* and not *classification*?

Great observation. This is where the different types of transfer learning come into play, as is, feature extraction and fine-tuning.

1. "As is" transfer learning is when you take a pretrained model as it is and apply it to your task without any changes.
  - For example, many computer vision models are pretrained on the ImageNet dataset which contains 1000 different classes of images. This means passing a single image to this model will produce 1000 different prediction probability values (1 for each class).
    - This is helpful if you have 1000 classes of images you'd like to classify and they're all the same as the ImageNet classes, however, it's not helpful if you want to classify only a small subset of classes (such as 10 different kinds of food). Models with "/classification" in their name on

TensorFlow Hub provide this kind of functionality.

2. **Feature extraction transfer learning** is when you take the underlying patterns (also called weights) a pretrained model has learned and adjust its outputs to be more suited to your problem.

- For example, say the pretrained model you were using had 236 different layers (EfficientNetB0 has 236 layers), but the top layer outputs 1000 classes because it was pretrained on ImageNet. To adjust this to your own problem, you might remove the original activation layer and replace it with your own but with the right number of output classes. The important part here is that **only the top few layers become trainable, the rest remain frozen**.

- This way all the underlying patterns remain in the rest of the layers and you can utilise them for your own problem. This kind of transfer learning is very helpful when your data is similar to the data a model has been pretrained on.

3. **Fine-tuning transfer learning** is when you take the underlying patterns (also called weights) of a pretrained model and adjust (fine-tune) them to your own problem.

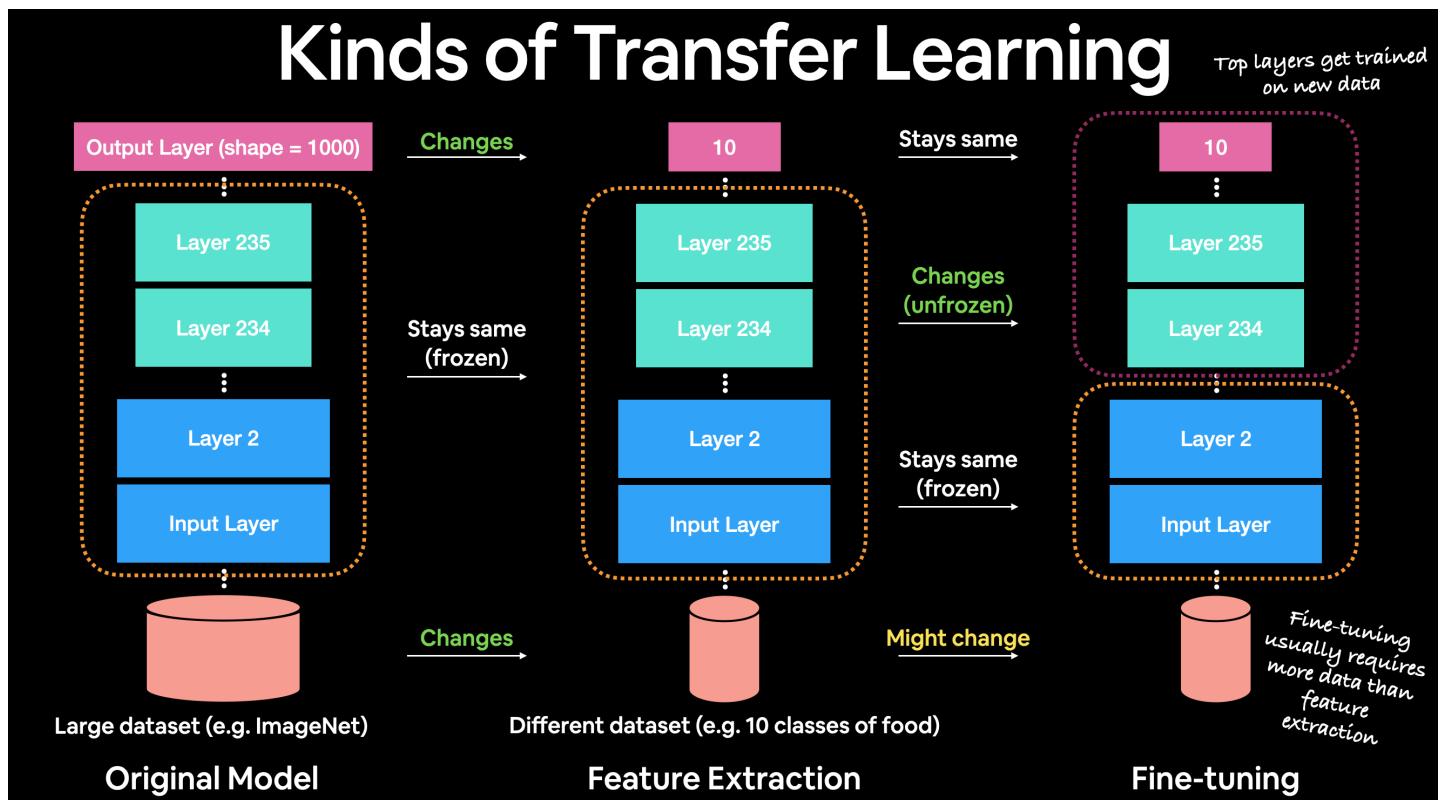
- This usually means training **some, many or all** of the layers in the pretrained model. This is useful when you've got a large dataset (e.g. 100+ images per class) where your data is slightly different to the data the original model was trained on.

A common workflow is to "freeze" all of the learned patterns in the bottom layers of a pretrained model so they're untrainable. And then train the top 2-3 layers of so the pretrained model can adjust its outputs to your custom data (**feature extraction**).

After you've trained the top 2-3 layers, you can then gradually "unfreeze" more and more layers and run the training process on your own data to further **fine-tune** the pretrained model.

□ **Question:** *Why train only the top 2-3 layers in feature extraction?*

The lower a layer is in a computer vision model as in, the closer it is to the input layer, the larger the features it learns. For example, a bottom layer in a computer vision model to identify images of cats or dogs might learn the outline of legs, whereas, layers closer to the output might learn the shape of teeth. Often, you'll want the larger features (learned patterns are also called features) to remain, since these are similar for both animals, whereas, the differences remain in the more fine-grained features.



*The different kinds of transfer learning. An original model, a feature extraction model (only top 2-3 layers change) and a fine-tuning model (many or all of original model get changed).*

Okay, enough talk, let's see this in action. Once we do, we'll explain what's happening.

First we'll import TensorFlow and TensorFlow Hub.

In [ ]:

```
import tensorflow as tf
import tensorflow_hub as hub
from tensorflow.keras import layers
```

Now we'll get the feature vector URLs of two common computer vision architectures, [EfficientNetB0 \(2019\)](#) and [ResNetV250 \(2016\)](#) from TensorFlow Hub using the steps above.

We're getting both of these because we're going to compare them to see which performs better on our data.

**Note:** Comparing different model architecture performance on the same data is a very common practice. The simple reason is because you want to know which model performs best for your problem.

**Update:** As of 14 August 2021, [EfficientNet V2 pretrained models are available on TensorFlow Hub](#). The original code in this notebook uses EfficientNet V1, it has been left unchanged. In [my experiments with this dataset](#), V1 outperforms V2. Best to experiment with your own data and see what suits you.

In [ ]:

```
# Resnet 50 V2 feature vector
resnet_url = "https://tfhub.dev/google/imagenet/resnet_v2_50/feature_vector/4"

# Original: EfficientNetB0 feature vector (version 1)
efficientnet_url = "https://tfhub.dev/tensorflow/efficientnet/b0/feature-vector/1"

# # New: EfficientNetB0 feature vector (version 2)
# efficientnet_url = "https://tfhub.dev/google/imagenet/efficientnet_v2_imagenet1k_b0/feature_vector/2"
```

These URLs link to a saved pretrained model on TensorFlow Hub.

When we use them in our model, the model will automatically be downloaded for us to use.

To do this, we can use the `KerasLayer()` model inside the TensorFlow hub library.

Since we're going to be comparing two models, to save ourselves code, we'll create a function `create_model()`. This function will take a model's TensorFlow Hub URL, instantiate a Keras Sequential model with the appropriate number of output layers and return the model.

In [ ]:

```
def create_model(model_url, num_classes=10):
    """Takes a TensorFlow Hub URL and creates a Keras Sequential model with it.

    Args:
        model_url (str): A TensorFlow Hub feature extraction URL.
        num_classes (int): Number of output neurons in output layer,
                           should be equal to number of target classes, default 10.

    Returns:
        An uncompiled Keras Sequential model with model_url as feature
        extractor layer and Dense output layer with num_classes outputs.
    """
    # Download the pretrained model and save it as a Keras layer
    feature_extractor_layer = hub.KerasLayer(model_url,
                                              trainable=False, # freeze the underlying patterns
                                              name='feature_extraction_layer',
                                              input_shape=IMAGE_SHAPE+(3,)) # define the input image shape
```

```
# Create our own model
model = tf.keras.Sequential([
    feature_extractor_layer, # use the feature extraction layer as the base
    layers.Dense(num_classes, activation='softmax', name='output_layer') # create our own output layer
])
return model
```

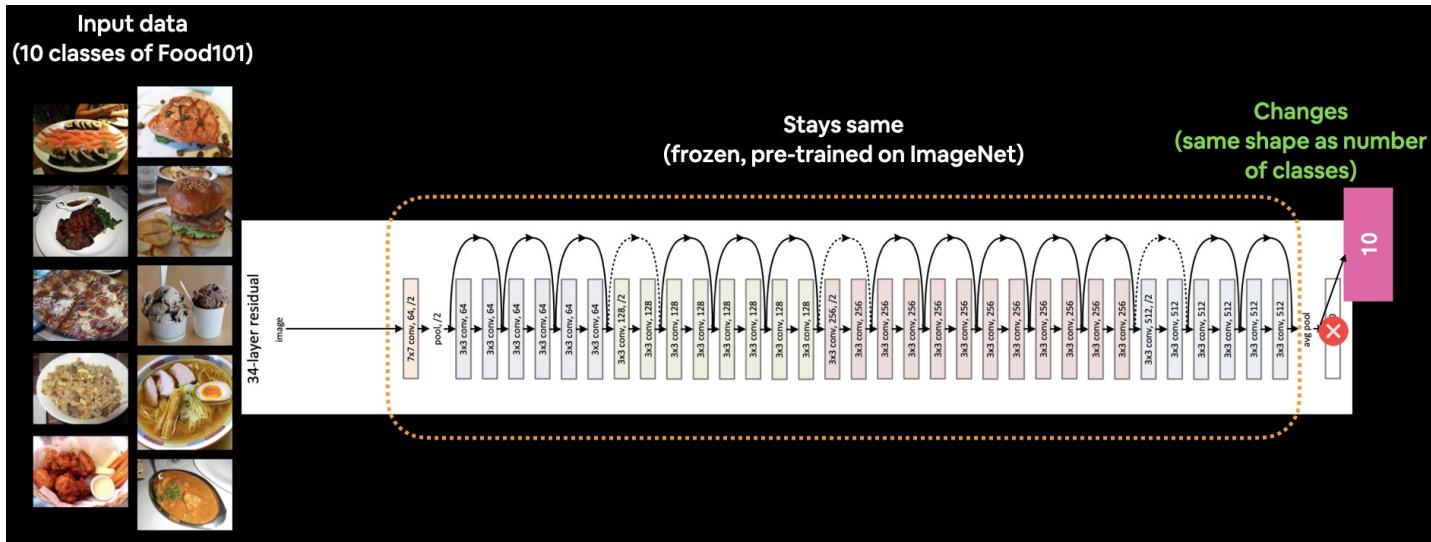
**Great! Now we've got a function for creating a model, we'll use it to first create a model using the ResNetV250 architecture as our feature extraction layer.**

Once the model is instantiated, we'll compile it using `categorical_crossentropy` as our loss function, the Adam optimizer and accuracy as our metric.

In [ ] :

```
# Create model
resnet_model = create_model(resnet_url, num_classes=train_data_10_percent.num_classes)

# Compile
resnet_model.compile(loss='categorical_crossentropy',
                      optimizer=tf.keras.optimizers.Adam(),
                      metrics=['accuracy'])
```



*What our current model looks like. A ResNet50V2 backbone with a custom dense layer on top (10 classes instead of 1000 ImageNet classes). Note: The image shows ResNet34 instead of ResNet50. Image source: <https://arxiv.org/abs/1512.03385>.*

**Beautiful. Time to fit the model.**

**We've got the training data ready in train data 10 percent as well as the test data saved as test data .**

But before we call the fit function, there's one more thing we're going to add, a callback. More specifically, a TensorBoard callback so we can track the performance of our model on TensorBoard.

We can add a callback to our model by using the `callbacks` parameter in the `fit` function.

In our case, we'll pass the `callbacks` parameter the `create_tensorboard_callback()` we created earlier with some specific inputs so we know what experiments we're running.

Let's keep this experiment short and train for 5 epochs.

In [ ]:

```

# Add TensorBoard callback to model (callbacks parameter takes a list)
        callbacks=[create_tensorboard_callback(dir_name="tensorflow_hub", # save experiment logs here
                                                experiment_name="resnet50V2")]) # name of log files

```

Saving TensorBoard log files to: tensorflow\_hub/resnet50V2/20210115-011336

Epoch 1/5  
24/24 [=====] - 29s 814ms/step - loss: 2.6383 - accuracy: 0.1690  
- val\_loss: 1.3164 - val\_accuracy: 0.5852

Epoch 2/5  
24/24 [=====] - 17s 737ms/step - loss: 1.0979 - accuracy: 0.6683  
- val\_loss: 0.9081 - val\_accuracy: 0.7124

Epoch 3/5  
24/24 [=====] - 17s 732ms/step - loss: 0.7306 - accuracy: 0.8073  
- val\_loss: 0.7874 - val\_accuracy: 0.7500

Epoch 4/5  
24/24 [=====] - 17s 735ms/step - loss: 0.5689 - accuracy: 0.8540  
- val\_loss: 0.7224 - val\_accuracy: 0.7640

Epoch 5/5  
24/24 [=====] - 17s 740ms/step - loss: 0.4724 - accuracy: 0.8913  
- val\_loss: 0.6969 - val\_accuracy: 0.7744

**Wow!**

**It seems that after only 5 epochs, the ResNetV250 feature extraction model was able to blow any of the architectures we made out of the water, achieving around 90% accuracy on the training set and nearly 80% accuracy on the test set...with only 10 percent of the training images!**

**That goes to show the power of transfer learning. And it's one of the main reasons whenever you're trying to model your own datasets, you should look into what pretrained models already exist.**

Let's check out our model's training curves using our `plot_loss_curves` function.

In [ ]:

```

# If you wanted to, you could really turn this into a helper function to load in with a helper.py script...
import matplotlib.pyplot as plt

# Plot the validation and training data separately
def plot_loss_curves(history):
    """
    Returns separate loss curves for training and validation metrics.
    """
    loss = history.history['loss']
    val_loss = history.history['val_loss']

    accuracy = history.history['accuracy']
    val_accuracy = history.history['val_accuracy']

    epochs = range(len(history.history['loss']))

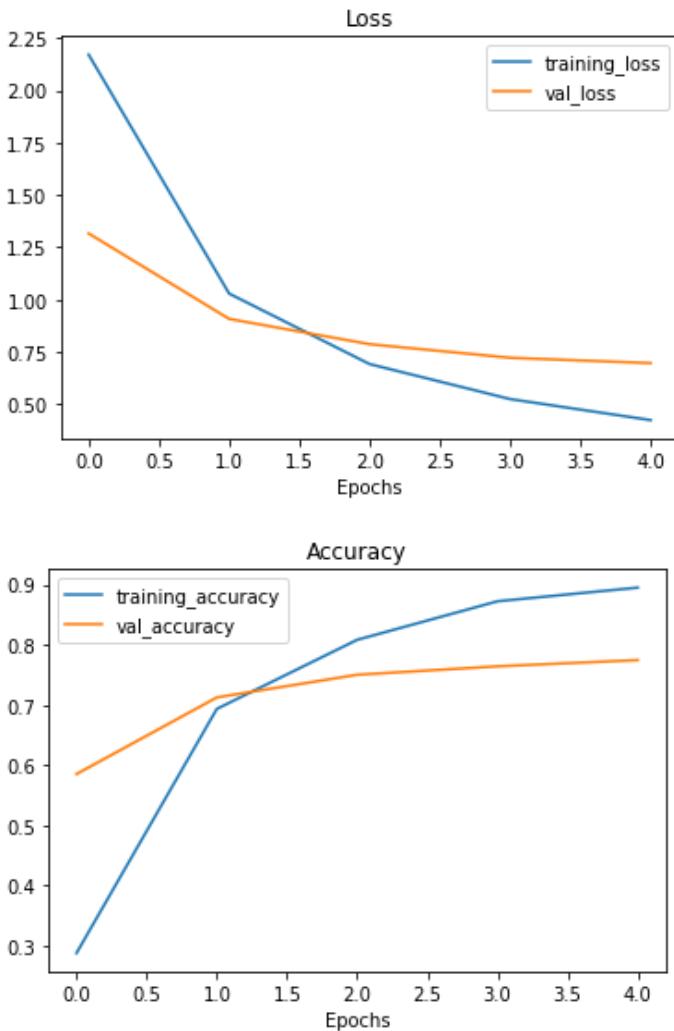
    # Plot loss
    plt.plot(epochs, loss, label='training_loss')
    plt.plot(epochs, val_loss, label='val_loss')
    plt.title('Loss')
    plt.xlabel('Epochs')
    plt.legend()

    # Plot accuracy
    plt.figure()
    plt.plot(epochs, accuracy, label='training_accuracy')
    plt.plot(epochs, val_accuracy, label='val_accuracy')
    plt.title('Accuracy')
    plt.xlabel('Epochs')
    plt.legend();

```

In [ ]:

```
plot_loss_curves(resnet_history)
```



**And what about a summary of our model?**

In [ ]:

```
# Resnet summary
resnet_model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
feature_extraction_layer (KerasLayer)	(None, 2048)	23564800
output_layer (Dense)	(None, 10)	20490
Total params:	23,585,290	
Trainable params:	20,490	
Non-trainable params:	23,564,800	

You can see the power of TensorFlow Hub here. The feature extraction layer has 23,564,800 parameters which are prelearned patterns the model has already learned on the ImageNet dataset. Since we set `trainable=False`, these patterns remain frozen (non-trainable) during training.

This means during training the model updates the 20,490 parameters in the output layer to suit our dataset.

Okay, we've trained a ResNetV250 model, time to do the same with EfficientNetB0 model.

The setup will be the exact same as before, except for the `model_url` parameter in the `create_model()` function and the `experiment_name` parameter in the `create_tensorboard_callback()` function.

In [ ]:

```

# Create model
efficientnet_model = create_model(model_url=efficientnet_url, # use EfficientNetB0 Tensor
Flow Hub URL
                                         num_classes=train_data_10_percent.num_classes)

# Compile EfficientNet model
efficientnet_model.compile(loss='categorical_crossentropy',
                            optimizer=tf.keras.optimizers.Adam(),
                            metrics=['accuracy'])

# Fit EfficientNet model
efficientnet_history = efficientnet_model.fit(train_data_10_percent, # only use 10% of training data
                                               epochs=5, # train for 5 epochs
                                               steps_per_epoch=len(train_data_10_percent),
                                               validation_data=test_data,
                                               validation_steps=len(test_data),
                                               callbacks=[create_tensorboard_callback(dir_name="tensorflow_hub",
                                                                                         # ex
periment_name="efficientnetB0")])

```

Saving TensorBoard log files to: tensorflow\_hub/efficientnetB0/20210115-011549

Epoch 1/5  
24/24 [=====] - 27s 819ms/step - loss: 2.0594 - accuracy: 0.3025  
- val\_loss: 1.2756 - val\_accuracy: 0.7524

Epoch 2/5  
24/24 [=====] - 16s 706ms/step - loss: 1.1388 - accuracy: 0.7528  
- val\_loss: 0.8559 - val\_accuracy: 0.8328

Epoch 3/5  
24/24 [=====] - 16s 707ms/step - loss: 0.7618 - accuracy: 0.8380  
- val\_loss: 0.6862 - val\_accuracy: 0.8496

Epoch 4/5  
24/24 [=====] - 16s 704ms/step - loss: 0.6424 - accuracy: 0.8662  
- val\_loss: 0.6034 - val\_accuracy: 0.8616

Epoch 5/5  
24/24 [=====] - 16s 708ms/step - loss: 0.4856 - accuracy: 0.9102  
- val\_loss: 0.5512 - val\_accuracy: 0.8668

**Holy smokes! The EfficientNetB0 model does even better than the ResNetV250 model! Achieving over 85% accuracy on the test set...again with only 10% of the training data.**

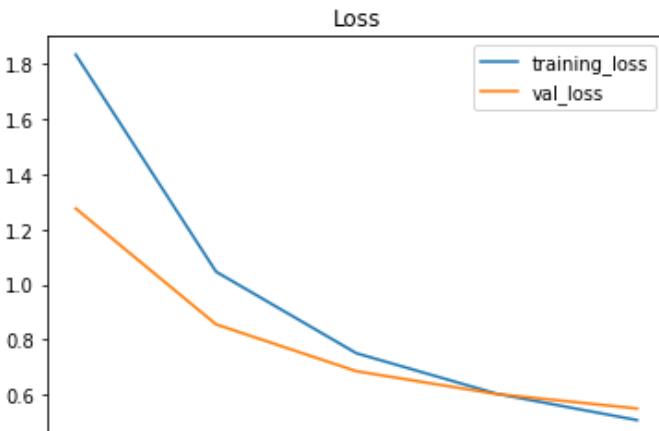
**How cool is that?**

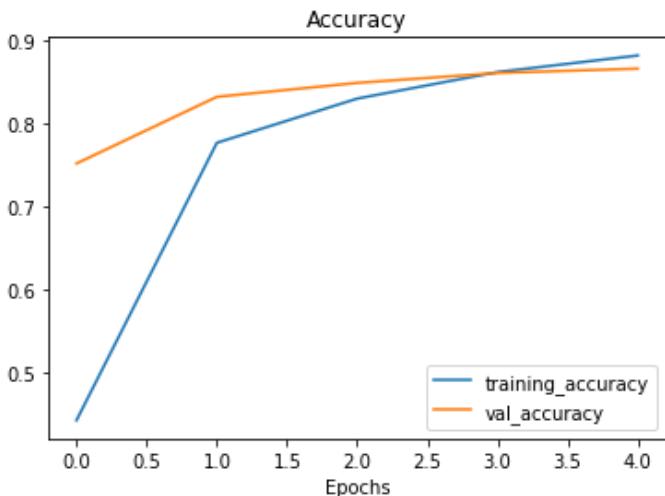
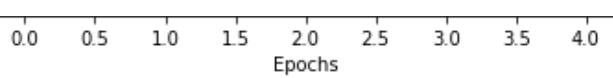
**With a couple of lines of code we're able to leverage state of the art models and adjust them to our own use case.**

**Let's check out the loss curves.**

In [ ]:

```
plot_loss_curves(efficientnet_history)
```





From the look of the EfficientNetB0 model's loss curves, it looks like if we kept training our model for longer, it might improve even further. Perhaps that's something you might want to try?

Let's check out the model summary.

In [ ]:

```
efficientnet_model.summary()
```

Model: "sequential\_1"

Layer (type)	Output Shape	Param #
feature_extraction_layer (KerasLayer)	(None, 1280)	4049564
output_layer (Dense)	(None, 10)	12810

Total params: 4,062,374  
Trainable params: 12,810  
Non-trainable params: 4,049,564

It seems despite having over four times less parameters (4,049,564 vs. 23,564,800) than the ResNet50V2 extraction layer, the EfficientNetB0 feature extraction layer yields better performance. Now it's clear where the "efficient" name came from.

## Comparing models using TensorBoard

Alright, even though we've already compared the performance of our two models by looking at the accuracy scores. But what if you had more than two models?

That's where an experiment tracking tool like [TensorBoard](#) (preinstalled in Google Colab) comes in.

The good thing is, since we set up a TensorBoard callback, all of our model's training logs have been saved automatically. To visualize them, we can upload the results to [TensorBoard.dev](#).

Uploading your results to TensorBoard.dev enables you to track and share multiple different modelling experiments. So if you needed to show someone your results, you could send them a link to your TensorBoard.dev as well as the accompanying Colab notebook.

**Note:** These experiments are public, do not upload sensitive data. You can delete experiments if needed.

## Uploading experiments to TensorBoard

To upload a series of TensorFlow logs to TensorBoard, we can use the following command:

```
Upload TensorBoard dev records
```

```
!tensorboard dev upload --logdir ./tensorflow_hub/ \
--name "EfficientNetB0 vs. ResNet50V2" \
--description "Comparing two different TF Hub feature extraction models architectures using 10% of training images" \
--one_shot
```

Where:

- `--logdir` is the target upload directory
- `--name` is the name of the experiment
- `--description` is a brief description of the experiment
- `--one_shot` exits the TensorBoard uploader once uploading is finished

Running the `tensorboard dev upload` command will first ask you to authorize the upload to TensorBoard.dev. After you've authorized the upload, your log files will be uploaded.

In [ ]:

```
# Upload TensorBoard dev records
!tensorboard dev upload --logdir ./tensorflow_hub/ \
--name "EfficientNetB0 vs. ResNet50V2" \
--description "Comparing two different TF Hub feature extraction models architectures using 10% of training images" \
--one_shot
```

2020-09-14 05:02:46.516878: I tensorflow/stream\_executor/platform/default/dso\_loader.cc:48] Successfully opened dynamic library libcudart.so.10.1  
Data for the "graphs" plugin is now uploaded to TensorBoard.dev! Note that uploaded data is public. If you do not want to upload data for this plugin, use the "--plugins" command line argument.  
Data for the "histograms" plugin is now uploaded to TensorBoard.dev! Note that uploaded data is public. If you do not want to upload data for this plugin, use the "--plugins" command line argument.  
Data for the "hparams" plugin is now uploaded to TensorBoard.dev! Note that uploaded data is public. If you do not want to upload data for this plugin, use the "--plugins" command line argument.  
Upload started and will continue reading any new data as it's added to the logdir. To stop uploading, press Ctrl-C.

View your TensorBoard live at: <https://tensorboard.dev/experiment/73taSKxXQeGPQsNBcVvY3g/>

```
[2020-09-14T05:02:48] Uploader started.
[2020-09-14T05:02:50] Total uploaded: 40 scalars, 0 tensors, 2 binary objects (3.2 MB)
Listening for new data in logdir...
Done. View your TensorBoard at https://tensorboard.dev/experiment/73taSKxXQeGPQsNBcVvY3g/
```

Every time you upload something to TensorBoard.dev you'll get a new experiment ID. The experiment ID will look something like this: <https://tensorboard.dev/experiment/73taSKxXQeGPQsNBcVvY3g/> (this is the actual experiment from this notebook).

If you upload the same directory again, you'll get a new experiment ID to go along with it.

This means to track your experiments, you may want to look into how you name your uploads. That way when you find them on TensorBoard.dev you can tell what happened during each experiment (e.g. "efficientnet0\_10\_percent\_data").

## Listing experiments you've saved to TensorBoard

To see all of the experiments you've uploaded you can use the command:

```
tensorboard dev list
```

In [ ]:

```
# Check out experiments  
!tensorboard dev list
```

```
2020-09-14 05:04:21.965097: I tensorflow/stream_executor/platform/default/dso_loader.cc:4  
8] Successfully opened dynamic library libcudart.so.10.1  
Data for the "graphs" plugin is now uploaded to TensorBoard.dev! Note that uploaded data  
is public. If you do not want to upload data for this plugin, use the "--plugins" command  
line argument.  
Data for the "histograms" plugin is now uploaded to TensorBoard.dev! Note that uploaded d  
ata is public. If you do not want to upload data for this plugin, use the "--plugins" comm  
and line argument.  
Data for the "hparams" plugin is now uploaded to TensorBoard.dev! Note that uploaded data  
is public. If you do not want to upload data for this plugin, use the "--plugins" command  
line argument.  
https://tensorboard.dev/experiment/73taSKxXQeGPQsNBcVvY3g/  
Name EfficientNetB0 vs. ResNet50V2  
Description Comparing two different TF Hub feature extraction models architectu  
res using 10% of training images  
Id 73taSKxXQeGPQsNBcVvY3g  
Created 2020-09-14 05:02:48 (1 minute ago)  
Updated 2020-09-14 05:02:50 (1 minute ago)  
Runs 4  
Tags 3  
Scalars 40  
Tensor bytes 0  
Binary object bytes 3402042  
https://tensorboard.dev/experiment/n6kd8XZ3Rdy1jSgSLH5WjA/  
Name EfficientNetB0 vs. ResNet50V2  
Description Comparing two different TF Hub feature extraction models architectu  
res using 10% of training images  
Id n6kd8XZ3Rdy1jSgSLH5WjA  
Created 2020-09-14 05:01:17 (3 minutes ago)  
Updated 2020-09-14 05:01:23 (3 minutes ago)  
Runs 10  
Tags 3  
Scalars 100  
Tensor bytes 0  
Binary object bytes 7619131  
Total: 2 experiment(s)
```

## Deleting experiments from TensorBoard

**Remember, all uploads to TensorBoard.dev are public, so to delete an experiment you can use the command:**

```
tensorboard dev delete --experiment_id [INSERT_EXPERIMENT_ID]
```

In [ ]:

```
# Delete an experiment  
!tensorboard dev delete --experiment_id n6kd8XZ3Rdy1jSgSLH5WjA
```

```
2020-09-14 05:06:06.959717: I tensorflow/stream_executor/platform/default/dso_loader.cc:4  
8] Successfully opened dynamic library libcudart.so.10.1  
Data for the "graphs" plugin is now uploaded to TensorBoard.dev! Note that uploaded data  
is public. If you do not want to upload data for this plugin, use the "--plugins" command  
line argument.  
Data for the "histograms" plugin is now uploaded to TensorBoard.dev! Note that uploaded d  
ata is public. If you do not want to upload data for this plugin, use the "--plugins" comm  
and line argument.  
Data for the "hparams" plugin is now uploaded to TensorBoard.dev! Note that uploaded data  
is public. If you do not want to upload data for this plugin, use the "--plugins" command  
line argument.  
Deleted experiment n6kd8XZ3Rdy1jSgSLH5WjA.
```

In [ ]:

```
# Check to see if experiments still exist  
!tensorboard dev list
```

```
2020-09-14 05:06:11.214919: I tensorflow/stream_executor/platform/default/dso_loader.cc:4
```

```
[8] Successfully opened dynamic library libcudart.so.10.1
Data for the "graphs" plugin is now uploaded to TensorBoard.dev! Note that uploaded data
is public. If you do not want to upload data for this plugin, use the "--plugins" command
line argument.
Data for the "histograms" plugin is now uploaded to TensorBoard.dev! Note that uploaded d
ata is public. If you do not want to upload data for this plugin, use the "--plugins" comm
and line argument.
Data for the "hparams" plugin is now uploaded to TensorBoard.dev! Note that uploaded data
is public. If you do not want to upload data for this plugin, use the "--plugins" command
line argument.
https://tensorboard.dev/experiment/73taSKxXQeGPQsNBcVvY3g/
Name           EfficientNetB0 vs. ResNet50V2
Description     Comparing two different TF Hub feature extraction models architectu
res using 10% of training images
Id             73taSKxXQeGPQsNBcVvY3g
Created        2020-09-14 05:02:48 (3 minutes ago)
Updated        2020-09-14 05:02:50 (3 minutes ago)
Runs           4
Tags           3
Scalars         40
Tensor bytes   0
Binary object bytes 3402042
Total: 1 experiment(s)
```

## Exercises

1. Build and fit a model using the same data we have here but with the MobileNetV2 architecture feature extraction ([mobilenet\\_v2\\_100\\_224/feature\\_vector](#)) from TensorFlow Hub, how does it perform compared to our other models?
2. Name 3 different image classification models on TensorFlow Hub that we haven't used.
3. Build a model to classify images of two different things you've taken photos of.
  - You can use any feature extraction layer from TensorFlow Hub you like for this.
  - You should aim to have at least 10 images of each class, for example to build a fridge versus oven classifier, you'll want 10 images of fridges and 10 images of ovens.
4. What is the current best performing model on ImageNet?
  - Hint: you might want to check [sotabench.com](#) for this.

## Extra-curriculum

- Read through the [TensorFlow Transfer Learning Guide](#) and define the main two types of transfer learning in your own words.
- Go through the [Transfer Learning with TensorFlow Hub tutorial](#) on the TensorFlow website and rewrite all of the code yourself into a new Google Colab notebook making comments about what each step does along the way.
- We haven't covered fine-tuning with TensorFlow Hub in this notebook, but if you'd like to know more, go through the [fine-tuning a TensorFlow Hub model tutorial](#) on the TensorFlow homepage. How to fine-tune a tensorflow hub model:
- Look into [experiment tracking with Weights & Biases](#), how could you integrate it with our existing TensorBoard logs?

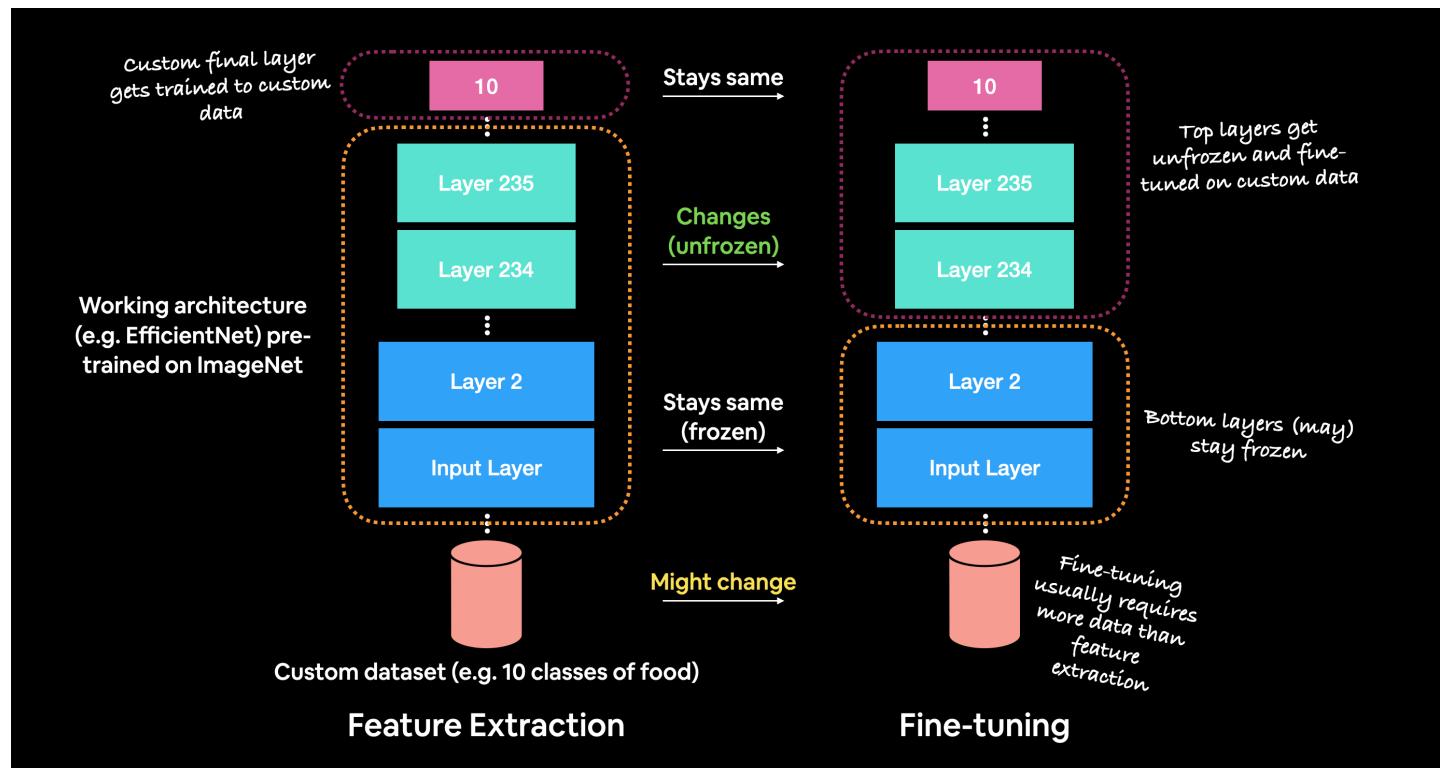
## 05. Transfer Learning with TensorFlow Part 2: Fine-tuning

In the previous section, we saw how we could leverage feature extraction transfer learning to get far better results on our Food Vision project than building our own models (even with less data).

Now we're going to cover another type of transfer learning: fine-tuning.

In **fine-tuning transfer learning** the pre-trained model weights from another model are unfrozen and tweaked during to better suit your own data.

For feature extraction transfer learning, you may only train the top 1-3 layers of a pre-trained model with your own data, in fine-tuning transfer learning, you might train 1-3+ layers of a pre-trained model (where the '+' indicates that many or all of the layers could be trained).



**Feature extraction transfer learning vs. fine-tuning transfer learning.** The main difference between the two is that in fine-tuning, more layers of the pre-trained model get unfrozen and tuned on custom data. This fine-tuning usually takes more data than feature extraction to be effective.

### What we're going to cover

We're going to go through the following with TensorFlow:

- Introduce fine-tuning, a type of transfer learning to modify a pre-trained model to be more suited to your data
- Using the Keras Functional API (a different way to build models in Keras)
- Using a smaller dataset to experiment faster (e.g. 1-10% of training samples of 10 classes of food)
- Data augmentation (how to make your training dataset more diverse without adding more data)
- Running a series of modelling experiments on our Food Vision data
  - Model 0: a transfer learning model using the Keras Functional API
  - Model 1: a feature extraction transfer learning model on 1% of the data with data augmentation
  - Model 2: a feature extraction transfer learning model on 10% of the data with data augmentation
  - Model 3: a fine-tuned transfer learning model on 10% of the data
  - Model 4: a fine-tuned transfer learning model on 100% of the data
- Introduce the ModelCheckpoint callback to save intermediate training results
- Compare model experiments results using TensorBoard

# How you can use this notebook

You can read through the descriptions and the code (it should all run, except for the cells which error on purpose), but there's a better option.

Write all of the code yourself.

Yes. I'm serious. Create a new notebook, and rewrite each line by yourself. Investigate it, see if you can break it, why does it break?

You don't have to write the text descriptions but writing the code yourself is a great way to get hands-on experience.

Don't worry if you make mistakes, we all do. The way to get better and make less mistakes is to **write more code**.

In [ ]:

```
# Are we using a GPU? (if not & you're using Google Colab, go to Runtime -> Change Runtime Type -> Hardware Accelerator: GPU )
!nvidia-smi
```

Tue Feb 16 02:14:29 2021

```
+-----+
| NVIDIA-SMI 460.39      Driver Version: 460.32.03     CUDA Version: 11.2      |
+-----+-----+-----+
| GPU  Name      Persistence-M| Bus-Id      Disp.A  | Volatile Uncorr. ECC  | | | | |
| Fan  Temp  Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M.  |
| |          |          |           |           |          | MIG M.   |
+=====+=====+=====+=====+=====+=====+=====+
| 0  Tesla T4        Off  | 00000000:00:04.0 Off |          0 | | | |
| N/A   73C   P8    13W / 70W |          0MiB / 15109MiB |     0%     Default |
|          |          |           |           |          | N/A      |
+-----+-----+-----+-----+-----+-----+
+-----+
| Processes:
| GPU  GI  CI      PID  Type      Process name          GPU Memory  |
|       ID  ID                  |                      Usage      |
+=====+=====+=====+=====+=====+=====+
|  No running processes found
+-----+
```

## Creating helper functions

Throughout your machine learning experiments, you'll likely come across snippets of code you want to use over and over again.

For example, a plotting function which plots a model's `history` object (see `plot_loss_curves()` below).

You could recreate these functions over and over again.

But as you might've guessed, rewriting the same functions becomes tedious.

One of the solutions is to store them in a helper script such as `helper_functions.py`. And then import the necessary functionality when you need it.

For example, you might write:

```
from helper_functions import plot_loss_curves

...
plot_loss_curves(history)
```

Let's see what this looks like.

In [ ]:

```
# Get helper_functions.py script from course GitHub
!wget https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/extras/helper_functions.py

# Import helper functions we're going to use
from helper_functions import create_tensorboard_callback, plot_loss_curves, unzip_data,
walk_through_dir

--2021-02-16 02:14:32-- https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/extras/helper_functions.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.111.133, 185.1
99.108.133, 185.199.109.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.111.133|:443...
.. connected.
HTTP request sent, awaiting response... 200 OK
Length: 9373 (9.2K) [text/plain]
Saving to: 'helper_functions.py.1'

helper_functions.py 100%[=====] 9.15K --.-KB/s in 0s

2021-02-16 02:14:32 (99.6 MB/s) - 'helper_functions.py.1' saved [9373/9373]
```

**Wonderful, now we've got a bunch of helper functions we can use throughout the notebook without having to rewrite them from scratch each time.**

**Note:** If you're running this notebook in Google Colab, when it times out Colab will delete the `helper_functions.py` file. So to use the functions imported above, you'll have to rerun the cell.

## 10 Food Classes: Working with less data

We saw in the [previous notebook](#) that we could get great results with only 10% of the training data using transfer learning with TensorFlow Hub.

In this notebook, we're going to continue to work with smaller subsets of the data, except this time we'll have a look at how we can use the in-built pretrained models within the `tf.keras.applications` module as well as how to fine-tune them to our own custom dataset.

We'll also practice using a new but similar dataloader function to what we've used before, `image_dataset_from_directory()` which is part of the `tf.keras.preprocessing` module.

Finally, we'll also be practicing using the [Keras Functional API](#) for building deep learning models. The Functional API is a more flexible way to create models than the `tf.keras.Sequential` API.

We'll explore each of these in more detail as we go.

Let's start by downloading some data.

In [ ]:

```
# Get 10% of the data of the 10 classes
!wget https://storage.googleapis.com/ztm_tf_course/food_vision/10_food_classes_10_percent.zip

unzip_data("10_food_classes_10_percent.zip")

--2021-02-16 02:14:53-- https://storage.googleapis.com/ztm_tf_course/food_vision/10_food_classes_10_percent.zip
Resolving storage.googleapis.com (storage.googleapis.com)... 172.217.12.240, 172.217.15.1
12, 172.253.62.128, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|172.217.12.240|:443... conn
ected.
HTTP request sent, awaiting response... 200 OK
-----
```

```
Length: 108546183 (101MB) [application/zip]
Saving to: '10_food_classes_10_percent.zip.1'
```

```
10_food_classes_10_ 100%[=====] 160.74M 186MB/s in 0.9s
```

```
2021-02-16 02:14:54 (186 MB/s) - '10_food_classes_10_percent.zip.1' saved [168546183/168546183]
```

The dataset we're downloading is the 10 food classes dataset (from Food 101) with 10% of the training images we used in the previous notebook.

**Note:** You can see how this dataset was created in the [image data modification notebook](#).

In [ ]:

```
# Walk through 10 percent data directory and list number of files
walk_through_dir("10_food_classes_10_percent")
```

```
There are 2 directories and 0 images in '10_food_classes_10_percent'.
There are 10 directories and 0 images in '10_food_classes_10_percent/train'.
There are 0 directories and 75 images in '10_food_classes_10_percent/train/ice_cream'.
There are 0 directories and 75 images in '10_food_classes_10_percent/train/ramen'.
There are 0 directories and 75 images in '10_food_classes_10_percent/train/chicken_wings'.
.
There are 0 directories and 75 images in '10_food_classes_10_percent/train/pizza'.
There are 0 directories and 75 images in '10_food_classes_10_percent/train/steak'.
There are 0 directories and 75 images in '10_food_classes_10_percent/train/fried_rice'.
There are 0 directories and 75 images in '10_food_classes_10_percent/train/hamburger'.
There are 0 directories and 75 images in '10_food_classes_10_percent/train/grilled_salmon'.
.
There are 0 directories and 75 images in '10_food_classes_10_percent/train/sushi'.
There are 0 directories and 75 images in '10_food_classes_10_percent/train/chicken_curry'.
.
There are 10 directories and 0 images in '10_food_classes_10_percent/test'.
There are 0 directories and 250 images in '10_food_classes_10_percent/test/ice_cream'.
There are 0 directories and 250 images in '10_food_classes_10_percent/test/ramen'.
There are 0 directories and 250 images in '10_food_classes_10_percent/test/chicken_wings'.
.
There are 0 directories and 250 images in '10_food_classes_10_percent/test/pizza'.
There are 0 directories and 250 images in '10_food_classes_10_percent/test/steak'.
There are 0 directories and 250 images in '10_food_classes_10_percent/test/fried_rice'.
There are 0 directories and 250 images in '10_food_classes_10_percent/test/hamburger'.
There are 0 directories and 250 images in '10_food_classes_10_percent/test/grilled_salmon'.
.
There are 0 directories and 250 images in '10_food_classes_10_percent/test/sushi'.
There are 0 directories and 250 images in '10_food_classes_10_percent/test/chicken_curry'.
.
```

We can see that each of the training directories contain 75 images and each of the testing directories contain 250 images.

Let's define our training and test filepaths.

In [ ]:

```
# Create training and test directories
train_dir = "10_food_classes_10_percent/train/"
test_dir = "10_food_classes_10_percent/test/"
```

Now we've got some image data, we need a way of loading it into a TensorFlow compatible format.

Previously, we've used the `ImageDataGenerator` class. And while this works well and is still very commonly used, this time we're going to use the `image_data_from_directory` function.

It works much the same way as `ImageDataGenerator`'s `flow_from_directory` method meaning your images need to be in the following file format:

Example of file structure

```
10_food_classes_10_percent <- top level folder
└── train <- training images
    ├── pizza
    │   ├── 1008104.jpg
    │   ├── 1638227.jpg
    │   ├── ...
    └── steak
        ├── 1000205.jpg
        ├── 1647351.jpg
        ├── ...
        ...
└── test <- testing images
    ├── pizza
    │   ├── 1001116.jpg
    │   ├── 1507019.jpg
    │   ├── ...
    └── steak
        ├── 100274.jpg
        ├── 1653815.jpg
        ├── ...
```

One of the main benefits of using `tf.keras.preprocessing.image_dataset_from_directory()` rather than `ImageDataGenerator` is that it creates a `tf.data.Dataset` object rather than a generator. The main advantage of this is the `tf.data.Dataset` API is much more efficient (faster) than the `ImageDataGenerator` API which is paramount for larger datasets.

Let's see it in action.

In [ ]:

```
# Create data inputs
import tensorflow as tf
IMG_SIZE = (224, 224) # define image size
train_data_10_percent = tf.keras.preprocessing.image_dataset_from_directory(directory=train_dir,
                                                                           image_size=IMG_SIZE,
                                                                           label_mode="categorical", # what type are the labels?
                                                                           batch_size=32) # batch_size is 32 by default, this is generally a good number
test_data_10_percent = tf.keras.preprocessing.image_dataset_from_directory(directory=test_dir,
                                                                           image_size=IMG_SIZE,
                                                                           label_mode="categorical")
```

Found 750 files belonging to 10 classes.

Found 2500 files belonging to 10 classes.

Wonderful! Looks like our dataloaders have found the correct number of images for each dataset.

For now, the main parameters we're concerned about in the `image_dataset_from_directory()` function are:

- `directory` - the filepath of the target directory we're loading images in from.
- `image_size` - the target size of the images we're going to load in (height, width).
- `batch_size` - the batch size of the images we're going to load in. For example if the `batch_size` is 32 (the default), batches of 32 images and labels at a time will be passed to the model.

There are more we could play around with if we needed to [in the `tf.keras.preprocessing` documentation](#).

If we check the training data datatype we should see it as a `BatchDataset` with shapes relating to our data.

In [ ]:

```
# Check the training data datatype
train_data_10_percent
```

Out [ ]:

```
<BatchDataset shapes: ((None, 224, 224, 3), (None, 10)), types: (tf.float32, tf.float32)>
```

In the above output:

- `(None, 224, 224, 3)` refers to the tensor shape of our images where `None` is the batch size, `224` is the height (and width) and `3` is the color channels (red, green, blue).
- `(None, 10)` refers to the tensor shape of the labels where `None` is the batch size and `10` is the number of possible labels (the 10 different food classes).
- Both image tensors and labels are of the datatype `tf.float32`.

The `batch_size` is `None` due to it only being used during model training. You can think of `None` as a placeholder waiting to be filled with the `batch_size` parameter from `image_dataset_from_directory()`.

Another benefit of using the `tf.data.Dataset` API are the associated methods which come with it.

For example, if we want to find the name of the classes we were working with, we could use the `class_names` attribute.

In [ ]:

```
# Check out the class names of our dataset
train_data_10_percent.class_names
```

Out [ ]:

```
['chicken_curry',
 'chicken_wings',
 'fried_rice',
 'grilled_salmon',
 'hamburger',
 'ice_cream',
 'pizza',
 'ramen',
 'steak',
 'sushi']
```

Or if we wanted to see an example batch of data, we could use the `take()` method.

In [ ]:

```
# See an example batch of data
for images, labels in train_data_10_percent.take(1):
    print(images, labels)
```

```
tf.Tensor(
[[[1.0000000e+00 0.0000000e+00 3.1000000e+01]
 [1.0000000e+00 0.0000000e+00 3.1000000e+01]
 [1.0000000e+00 0.0000000e+00 3.1000000e+01]
 ...
 [1.0000000e+00 0.0000000e+00 3.1000000e+01]
 [1.0000000e+00 0.0000000e+00 3.1000000e+01]
 [1.0000000e+00 0.0000000e+00 3.1000000e+01]]
 [[1.0000000e+00 0.0000000e+00 3.1000000e+01]
 [1.0000000e+00 0.0000000e+00 3.1000000e+01]
 [1.0000000e+00 0.0000000e+00 3.1000000e+01]
 ...
 [1.0000000e+00 0.0000000e+00 3.1000000e+01]
 [1.0000000e+00 0.0000000e+00 3.1000000e+01]
 [1.0000000e+00 0.0000000e+00 3.1000000e+01]]
```

```
[1.0000000e+00 0.0000000e+00 3.1000000e+01] ]  
[[1.0000000e+00 0.0000000e+00 3.1000000e+01]  
[1.0000000e+00 0.0000000e+00 3.1000000e+01]  
[1.0000000e+00 0.0000000e+00 3.1000000e+01]  
...  
[1.0000000e+00 0.0000000e+00 3.1000000e+01]  
[1.0000000e+00 0.0000000e+00 3.1000000e+01]  
[1.0000000e+00 0.0000000e+00 3.1000000e+01]]  
...  
[[1.07500107e+02 9.49286346e+01 8.90816803e+01]  
[1.15714394e+02 1.01285782e+02 9.41582489e+01]  
[1.17974548e+02 1.04188812e+02 9.51888123e+01]  
...  
[1.18617378e+02 2.90000000e+01 2.76939182e+01]  
[1.19000000e+02 2.90000000e+01 2.80000000e+01]  
[1.20076530e+02 3.00765305e+01 2.98622665e+01]]  
[[1.07045891e+02 9.20458908e+01 8.70458908e+01]  
[1.15852043e+02 1.00852043e+02 9.38520432e+01]  
[1.15841820e+02 1.02056099e+02 9.32703857e+01]  
...  
[1.17943863e+02 2.99438648e+01 2.87296009e+01]  
[1.17923454e+02 2.79234543e+01 2.79234543e+01]  
[1.17857117e+02 2.78571167e+01 2.78571167e+01]]  
[[1.00785606e+02 8.57856064e+01 8.07856064e+01]  
[1.13999931e+02 9.89999313e+01 9.28061218e+01]  
[1.11999931e+02 9.74284973e+01 9.06427841e+01]  
...  
[1.16857086e+02 2.88570862e+01 2.76428223e+01]  
[1.14571381e+02 2.65713806e+01 2.55713806e+01]  
[1.14642822e+02 2.46428223e+01 2.66428223e+01]]]  
  
[[ [9.76530609e+01 7.90663223e+01 6.50663223e+01]  
[1.04392860e+02 8.82500000e+01 7.33214264e+01]  
[1.09520409e+02 9.69591827e+01 8.36683655e+01]  
...  
[2.44423492e+02 2.37423492e+02 2.19423492e+02]  
[2.46760208e+02 2.39760208e+02 2.21760208e+02]  
[2.48000000e+02 2.41000000e+02 2.23000000e+02]]  
  
[[ [1.06326530e+02 8.95204086e+01 7.75867386e+01]  
[1.11239799e+02 9.68877640e+01 8.40918427e+01]  
[1.50612259e+02 1.41198990e+02 1.26897972e+02]  
...  
[2.40857162e+02 2.33857162e+02 2.15857162e+02]  
[2.45071442e+02 2.38071442e+02 2.20071442e+02]  
[2.47000000e+02 2.40000000e+02 2.22000000e+02]]  
  
[[ [1.15163269e+02 1.02295921e+02 9.33724518e+01]  
[1.54887756e+02 1.45117355e+02 1.34989807e+02]  
[2.00071426e+02 1.92836746e+02 1.79790833e+02]  
...  
[2.40357193e+02 2.33357193e+02 2.15357193e+02]  
[2.43642868e+02 2.36642868e+02 2.18642868e+02]  
[2.44571426e+02 2.37571426e+02 2.19571426e+02]]  
  
...  
[[ [1.41214325e+02 5.36428566e+01 5.27959023e+01]  
[1.37571487e+02 5.00000229e+01 4.82704086e+01]  
[1.39000046e+02 5.16428375e+01 4.90000000e+01]  
...  
[2.50168350e+02 2.41168350e+02 2.24168350e+02]  
[2.50000000e+02 2.41000000e+02 2.24000000e+02]  
[2.49785736e+02 2.40785736e+02 2.23785736e+02]]  
[[1.41974503e+02 5.39744949e+01 5.19744949e+01]
```

```

[1.40357147e+02 5.23571548e+01 5.03571548e+01]
[1.39642853e+02 5.16581650e+01 4.76122398e+01]
...
[2.51000000e+02 2.42000000e+02 2.25000000e+02]
[2.50933655e+02 2.41933655e+02 2.24933655e+02]
[2.50000000e+02 2.41000000e+02 2.24000000e+02]

[[1.42571426e+02 5.45714302e+01 5.25714302e+01]
[1.42617371e+02 5.46173782e+01 5.26173782e+01]
[1.39642853e+02 5.18571434e+01 4.72142868e+01]
...
[2.51494919e+02 2.42494919e+02 2.25494919e+02]
[2.51000000e+02 2.42000000e+02 2.25000000e+02]
[2.51000000e+02 2.42000000e+02 2.25000000e+02]]]

[[[3.25867348e+01 3.25867348e+01 3.05867348e+01]
[3.20000000e+01 3.20000000e+01 3.20000000e+01]
[3.24285736e+01 3.20000000e+01 3.42142868e+01]
...
[3.66428566e+01 3.14285927e+01 2.90713844e+01]
[3.03570900e+01 2.53570900e+01 2.23570900e+01]
[2.30560646e+01 1.80560646e+01 1.50560656e+01]]

[[3.14030609e+01 3.14030609e+01 2.94030609e+01]
[3.10714283e+01 3.10714283e+01 3.10714283e+01]
[3.24285736e+01 3.20000000e+01 3.42142868e+01]
...
[3.91428375e+01 3.39285736e+01 3.15713634e+01]
[3.35713768e+01 2.85713768e+01 2.55713768e+01]
[2.48570385e+01 1.98570385e+01 1.68570385e+01]]

[[3.00000000e+01 3.00000000e+01 2.80000000e+01]
[3.07295914e+01 3.07295914e+01 3.07295914e+01]
[3.24285736e+01 3.20000000e+01 3.42142868e+01]
...
[4.04285278e+01 3.52142639e+01 3.28570557e+01]
[3.82142487e+01 3.32142487e+01 3.02142467e+01]
[2.99998608e+01 2.49998608e+01 2.19998608e+01]]

...
[[2.54000000e+02 2.54000000e+02 2.54000000e+02]
[2.54000000e+02 2.54000000e+02 2.54000000e+02]
[2.54000000e+02 2.54000000e+02 2.54000000e+02]
...
[4.64285278e+01 2.27857361e+01 9.00000000e+00]
[4.59285583e+01 2.21428223e+01 9.71429443e+00]
[4.28570862e+01 1.90713501e+01 6.64282227e+00]]]

[[2.54000000e+02 2.54000000e+02 2.54000000e+02]
[2.54000000e+02 2.54000000e+02 2.54000000e+02]
[2.54000000e+02 2.54000000e+02 2.54000000e+02]
...
[4.63723946e+01 2.27296009e+01 8.94386578e+00]
[4.27856750e+01 1.87856750e+01 6.78567505e+00]
[3.95968971e+01 1.55968981e+01 3.59689808e+00]]]

[[2.54000000e+02 2.54000000e+02 2.54000000e+02]
[2.54000000e+02 2.54000000e+02 2.54000000e+02]
[2.54000000e+02 2.54000000e+02 2.54000000e+02]
...
[4.32090836e+01 1.87805557e+01 6.56629181e+00]
[4.10000000e+01 1.70000000e+01 5.00000000e+00]
[4.07143555e+01 1.67143555e+01 6.71435547e+00]]]

...
[[[2.35000000e+02 2.33000000e+02 1.84000000e+02]
[2.36928574e+02 2.34928574e+02 1.86928574e+02]

```

```
[2.38138702e+02 2.35924423e+02 1.88567276e+02]
...
[1.42204636e+02 1.06061691e+02 7.66331635e+01]
[1.36100983e+02 9.71009827e+01 6.61009827e+01]
[1.26105240e+02 8.51052399e+01 5.51052399e+01]

[[2.35642853e+02 2.31642853e+02 1.83642853e+02]
[2.37874680e+02 2.33874680e+02 1.86874680e+02]
[2.38941971e+02 2.34941971e+02 1.88370544e+02]
...
[1.49318756e+02 1.13961540e+02 8.21509323e+01]
[1.34856155e+02 9.58561478e+01 6.48561478e+01]
[1.27820007e+02 8.68200073e+01 5.68200111e+01]]

[[2.35642853e+02 2.31642853e+02 1.83642853e+02]
[2.36945480e+02 2.32945480e+02 1.85182083e+02]
[2.37236603e+02 2.33236603e+02 1.86338013e+02]
...
[1.48766525e+02 1.13409317e+02 8.11950531e+01]
[1.38290176e+02 9.92901764e+01 6.67633896e+01]
[1.32347656e+02 9.13476562e+01 5.98208733e+01]]

...
[[2.55000000e+02 2.12133591e+02 1.86660324e+02]
[2.55000000e+02 2.14763367e+02 1.86763367e+02]
[2.53323334e+02 2.12650497e+02 1.84385498e+02]
...
[2.26525040e+02 1.69525040e+02 1.62525040e+02]
[2.52622208e+02 1.98511551e+02 1.89116196e+02]
[2.53545837e+02 2.02877548e+02 1.91211685e+02]]

[[2.52510468e+02 2.10364456e+02 1.82758881e+02]
[2.53116028e+02 2.12116028e+02 1.84116028e+02]
[2.53416138e+02 2.14300110e+02 1.85131409e+02]
...
[1.46211349e+02 8.82693710e+01 8.40953293e+01]
[2.23439270e+02 1.68443405e+02 1.61430969e+02]
[2.53081512e+02 2.00401398e+02 1.91860809e+02]]

[[2.52710815e+02 2.12458923e+02 1.86584869e+02]
[2.51163895e+02 2.12163895e+02 1.83163895e+02]
[2.47927551e+02 2.10776413e+02 1.81776413e+02]
...
[1.41158997e+02 8.71762772e+01 8.26849136e+01]
[1.58315674e+02 1.05315666e+02 9.93156662e+01]
[2.39812149e+02 1.89132050e+02 1.82106827e+02]]]

[[[2.23229599e+02 1.86229599e+02 8.22295914e+01]
[2.27816330e+02 1.90816330e+02 8.68163300e+01]
[2.31607132e+02 1.93607132e+02 9.26071396e+01]
...
[6.21326485e+01 2.38417950e+01 5.98976803e+00]
[6.32602386e+01 2.16173820e+01 4.85206604e+00]
[6.12141113e+01 1.95712547e+01 2.98453188e+00]]]

[[2.20500000e+02 1.83525497e+02 7.73775558e+01]
[2.20214294e+02 1.83214294e+02 7.89336777e+01]
[2.21770416e+02 1.83770416e+02 8.33112259e+01]
...
[6.63571014e+01 2.94132233e+01 3.19893765e+00]
[6.80765457e+01 2.81377678e+01 2.78571439e+00]
[6.53365631e+01 2.34079933e+01 5.96906424e-01]]]

[[[2.15132660e+02 1.78994904e+02 6.78367386e+01]
[2.13642868e+02 1.76642868e+02 7.05714264e+01]
[2.16091843e+02 1.77923462e+02 7.95969391e+01]
...
[7.24439240e+01 3.48265762e+01 4.59136739e-02]
[7.98724670e+01 4.06734505e+01 3.87243915e+00]
[7.55713272e+01 3.37856102e+01 0.00000000e+00]]]
```

...

[[2.47362259e+02 2.40362259e+02 1.86010223e+02]  
[2.47127548e+02 2.40127548e+02 1.85428574e+02]  
[2.45025513e+02 2.38214279e+02 1.81025528e+02]

...

[2.49000000e+02 2.41168381e+02 1.91663239e+02]  
[2.48729568e+02 2.40943863e+02 1.91300980e+02]  
[2.48000000e+02 2.40494934e+02 1.89857056e+02]]

[[2.50163315e+02 2.43163315e+02 1.91071503e+02]  
[2.48000015e+02 2.41000015e+02 1.88867401e+02]  
[2.47301056e+02 2.39673508e+02 1.85801071e+02]

...

[2.48071411e+02 2.41056107e+02 1.89102020e+02]  
[2.48000000e+02 2.41000000e+02 1.88867325e+02]  
[2.48331665e+02 2.41331665e+02 1.87423477e+02]]

[[2.50301025e+02 2.42301025e+02 1.93301025e+02]  
[2.46382660e+02 2.38714294e+02 1.88719376e+02]  
[2.47005112e+02 2.39362244e+02 1.86586731e+02]

...

[2.47214264e+02 2.40214264e+02 1.87204041e+02]  
[2.47071442e+02 2.40071442e+02 1.86071442e+02]  
[2.49000000e+02 2.42000000e+02 1.87229584e+02]]

[[[6.97142868e+01 3.65000000e+01 3.78571415e+00]  
[7.13112259e+01 3.87397957e+01 4.09693861e+00]  
[6.95765305e+01 3.76479568e+01 2.29081607e+00]

...

[8.57296371e+01 7.10051575e+01 3.42143517e+01]  
[8.00407181e+01 6.71580582e+01 3.58723984e+01]  
[6.88163376e+01 5.69591942e+01 3.08164062e+01]]

[[7.57602081e+01 3.49030609e+01 2.83163333e+00]  
[7.48520432e+01 3.60663261e+01 2.99489808e+00]  
[6.94285660e+01 3.38571396e+01 5.61222248e-02]

...

[8.16378098e+01 5.68265762e+01 2.58265553e+01]  
[8.39948807e+01 5.93520241e+01 1.94999714e+01]  
[8.73317719e+01 6.36174850e+01 1.82603378e+01]]

[[6.67193832e+01 3.52193871e+01 2.21938777e+00]  
[6.24285660e+01 3.07142849e+01 1.53060742e-02]  
[5.93469391e+01 2.78316326e+01 9.18368548e-02]

...

[8.25203552e+01 6.37601357e+01 3.13570137e+01]  
[7.80000000e+01 5.59285469e+01 1.55867128e+01]  
[8.65716400e+01 6.06430664e+01 1.65002098e+01]]

...

[[2.00591873e+02 2.15158142e+02 2.14382751e+02]  
[1.96270523e+02 2.09841873e+02 2.02326660e+02]  
[1.92760223e+02 2.06545929e+02 1.90780655e+02]

...

[1.14596886e+02 1.18596886e+02 9.48316422e+01]  
[1.22800980e+02 1.27800980e+02 1.05800980e+02]  
[1.15928619e+02 1.20928619e+02 1.00928619e+02]]

[[1.94892899e+02 2.02750015e+02 2.14678635e+02]  
[2.08076462e+02 2.16928482e+02 2.22428528e+02]  
[1.96413361e+02 2.09556198e+02 2.00153214e+02]

...

[1.15285706e+02 1.18270401e+02 9.94592056e+01]  
[1.21846909e+02 1.26704025e+02 1.06775467e+02]  
[1.10285736e+02 1.15142853e+02 9.32142944e+01]]

[[1.85152374e+02 1.88152374e+02 2.06325897e+02]  
[2.01362289e+02 2.07959198e+02 2.18908234e+02]

Notice how the image arrays come out as tensors of pixel values where as the labels come out as one-hot encodings (e.g. [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.] for hamburger).

Model 0: Building a transfer learning model using the Keras Functional API

**Alright, our data is tensor-ified. Let's build a model.**

To do so we're going to be using the `tf.keras.applications` module as it contains a series of already trained (on ImageNet) computer vision models as well as the Keras Functional API to construct our model.

**We're going to go through the following steps:**

1. Instantiate a pre-trained base model object by choosing a target model such as `EfficientNetB0` from `tf.keras.applications`, setting the `include_top` parameter to `False` (we do this because we're going to create our own top, which are the output layers for the model).
  2. Set the base model's `trainable` attribute to `False` to freeze all of the weights in the pre-trained model.
  3. Define an input layer for our model, for example, what shape of data should our model expect?
  4. [Optional] Normalize the inputs to our model if it requires. Some computer vision models such as `ResNetV250` require their inputs to be between 0 & 1.

**Note:** As of writing, the `EfficientNet` models in the `tf.keras.applications` module do not require images to be normalized (pixel values between 0 and 1) on input, where as many of the other models do. I posted [an issue to the TensorFlow GitHub](#) about this and they confirmed this.

1. Pass the inputs to the base model.
  2. Pool the outputs of the base model into a shape compatible with the output activation layer (turn base model

- output tensors into same shape as label tensors). This can be done using `tf.keras.layers.GlobalAveragePooling2D()` or `tf.keras.layers.GlobalMaxPooling2D()`**
- though the former is more common in practice.**
3. Create an output activation layer using `tf.keras.layers.Dense()` with the appropriate activation function and number of neurons.
  4. Combine the inputs and outputs layer into a model using `tf.keras.Model()`.
  5. Compile the model using the appropriate loss function and choose of optimizer.
  6. Fit the model for desired number of epochs and with necessary callbacks (in our case, we'll start off with the TensorBoard callback).

Woah... that sounds like a lot. Before we get ahead of ourselves, let's see it in practice.

In [ ]:

```
# 1. Create base model with tf.keras.applications
base_model = tf.keras.applications.EfficientNetB0(include_top=False)

# 2. Freeze the base model (so the pre-learned patterns remain)
base_model.trainable = False

# 3. Create inputs into the base model
inputs = tf.keras.layers.Input(shape=(224, 224, 3), name="input_layer")

# 4. If using ResNet50V2, add this to speed up convergence, remove for EfficientNet
# x = tf.keras.layers.experimental.preprocessing.Rescaling(1./255)(inputs)

# 5. Pass the inputs to the base_model (note: using tf.keras.applications, EfficientNet i
nputs don't have to be normalized)
x = base_model(inputs)
# Check data shape after passing it to base_model
print(f"Shape after base_model: {x.shape}")

# 6. Average pool the outputs of the base model (aggregate all the most important information, reduce number of computations)
x = tf.keras.layers.GlobalAveragePooling2D(name="global_average_pooling_layer")(x)
print(f"After GlobalAveragePooling2D(): {x.shape}")

# 7. Create the output activation layer
outputs = tf.keras.layers.Dense(10, activation="softmax", name="output_layer")(x)

# 8. Combine the inputs with the outputs into a model
model_0 = tf.keras.Model(inputs, outputs)

# 9. Compile the model
model_0.compile(loss='categorical_crossentropy',
                 optimizer=tf.keras.optimizers.Adam(),
                 metrics=["accuracy"])

# 10. Fit the model (we use less steps for validation so it's faster)
history_10_percent = model_0.fit(train_data_10_percent,
                                   epochs=5,
                                   steps_per_epoch=len(train_data_10_percent),
                                   validation_data=test_data_10_percent,
                                   # Go through less of the validation data so epochs are
faster (we want faster experiments!)
                                   validation_steps=int(0.25 * len(test_data_10_percent)),
                                   # Track our model's training logs for visualization later
                                   callbacks=[create_tensorboard_callback("transfer_learning", "10_percent_feature_extract")])
```

Shape after base\_model: (None, 7, 7, 1280)  
After GlobalAveragePooling2D(): (None, 1280)  
Saving TensorBoard log files to: transfer\_learning/10\_percent\_feature\_extract/20210216-02  
1515  
Epoch 1/5  
24/24 [=====] - 14s 313ms/step - loss: 2.1271 - accuracy: 0.2823  
- val\_loss: 1.3399 - val\_accuracy: 0.6743  
Epoch 2/5

```
24/24 [=====] - 6s 223ms/step - loss: 1.26/6 - accuracy: 0.6/41
- val_loss: 0.8985 - val_accuracy: 0.8224
Epoch 3/5
24/24 [=====] - 6s 222ms/step - loss: 0.9113 - accuracy: 0.7824
- val_loss: 0.6983 - val_accuracy: 0.8520
Epoch 4/5
24/24 [=====] - 6s 224ms/step - loss: 0.7330 - accuracy: 0.8208
- val_loss: 0.6213 - val_accuracy: 0.8635
Epoch 5/5
24/24 [=====] - 6s 224ms/step - loss: 0.6232 - accuracy: 0.8519
- val_loss: 0.5716 - val_accuracy: 0.8651
```

**Nice! After a minute or so of training our model performs incredibly well on both the training (87%+ accuracy) and test sets (~83% accuracy).**

**This is incredible. All thanks to the power of transfer learning.**

**It's important to note the kind of transfer learning we used here is called feature extraction transfer learning, similar to what we did with the TensorFlow Hub models.**

**In other words, we passed our custom data to an already pre-trained model (`EfficientNetB0`), asked it "what patterns do you see?" and then put our own output layer on top to make sure the outputs were tailored to our desired number of classes.**

**We also used the Keras Functional API to build our model rather than the Sequential API. For now, the benefits of this main not seem clear but when you start to build more sophisticated models, you'll probably want to use the Functional API. So it's important to have exposure to this way of building models.**

**Resource:** To see the benefits and use cases of the Functional API versus the Sequential API, check out the [TensorFlow Functional API documentation](#).

**Let's inspect the layers in our model, we'll start with the base.**

In [ ]:

```
# Check layers in our base model
for layer_number, layer in enumerate(base_model.layers):
    print(layer_number, layer.name)
```

```
0 input_1
1 rescaling
2 normalization
3 stem_conv_pad
4 stem_conv
5 stem_bn
6 stem_activation
7 block1a_dwconv
8 block1a_bn
9 block1a_activation
10 block1a_se_squeeze
11 block1a_se_reshape
12 block1a_se_reduce
13 block1a_se_expand
14 block1a_se_excite
15 block1a_project_conv
16 block1a_project_bn
17 block2a_expand_conv
18 block2a_expand_bn
19 block2a_expand_activation
20 block2a_dwconv_pad
21 block2a_dwconv
22 block2a_bn
23 block2a_activation
24 block2a_se_squeeze
25 block2a_se_reshape
26 block2a_se_reduce
27 block2a_se_expand
28 block2a_se_excite
29 block2a_project_conv
```

```
30 block2a_project_bn
31 block2b_expand_conv
32 block2b_expand_bn
33 block2b_expand_activation
34 block2b_dwconv
35 block2b_bn
36 block2b_activation
37 block2b_se_squeeze
38 block2b_se_reshape
39 block2b_se_reduce
40 block2b_se_expand
41 block2b_se_excite
42 block2b_project_conv
43 block2b_project_bn
44 block2b_drop
45 block2b_add
46 block3a_expand_conv
47 block3a_expand_bn
48 block3a_expand_activation
49 block3a_dwconv_pad
50 block3a_dwconv
51 block3a_bn
52 block3a_activation
53 block3a_se_squeeze
54 block3a_se_reshape
55 block3a_se_reduce
56 block3a_se_expand
57 block3a_se_excite
58 block3a_project_conv
59 block3a_project_bn
60 block3b_expand_conv
61 block3b_expand_bn
62 block3b_expand_activation
63 block3b_dwconv
64 block3b_bn
65 block3b_activation
66 block3b_se_squeeze
67 block3b_se_reshape
68 block3b_se_reduce
69 block3b_se_expand
70 block3b_se_excite
71 block3b_project_conv
72 block3b_project_bn
73 block3b_drop
74 block3b_add
75 block4a_expand_conv
76 block4a_expand_bn
77 block4a_expand_activation
78 block4a_dwconv_pad
79 block4a_dwconv
80 block4a_bn
81 block4a_activation
82 block4a_se_squeeze
83 block4a_se_reshape
84 block4a_se_reduce
85 block4a_se_expand
86 block4a_se_excite
87 block4a_project_conv
88 block4a_project_bn
89 block4b_expand_conv
90 block4b_expand_bn
91 block4b_expand_activation
92 block4b_dwconv
93 block4b_bn
94 block4b_activation
95 block4b_se_squeeze
96 block4b_se_reshape
97 block4b_se_reduce
98 block4b_se_expand
99 block4b_se_excite
100 block4b_project_conv
101 block4b_project_bn
```

102 block4b\_drop  
103 block4b\_add  
104 block4c\_expand\_conv  
105 block4c\_expand\_bn  
106 block4c\_expand\_activation  
107 block4c\_dwconv  
108 block4c\_bn  
109 block4c\_activation  
110 block4c\_se\_squeeze  
111 block4c\_se\_reshape  
112 block4c\_se\_reduce  
113 block4c\_se\_expand  
114 block4c\_se\_excite  
115 block4c\_project\_conv  
116 block4c\_project\_bn  
117 block4c\_drop  
118 block4c\_add  
119 block5a\_expand\_conv  
120 block5a\_expand\_bn  
121 block5a\_expand\_activation  
122 block5a\_dwconv  
123 block5a\_bn  
124 block5a\_activation  
125 block5a\_se\_squeeze  
126 block5a\_se\_reshape  
127 block5a\_se\_reduce  
128 block5a\_se\_expand  
129 block5a\_se\_excite  
130 block5a\_project\_conv  
131 block5a\_project\_bn  
132 block5b\_expand\_conv  
133 block5b\_expand\_bn  
134 block5b\_expand\_activation  
135 block5b\_dwconv  
136 block5b\_bn  
137 block5b\_activation  
138 block5b\_se\_squeeze  
139 block5b\_se\_reshape  
140 block5b\_se\_reduce  
141 block5b\_se\_expand  
142 block5b\_se\_excite  
143 block5b\_project\_conv  
144 block5b\_project\_bn  
145 block5b\_drop  
146 block5b\_add  
147 block5c\_expand\_conv  
148 block5c\_expand\_bn  
149 block5c\_expand\_activation  
150 block5c\_dwconv  
151 block5c\_bn  
152 block5c\_activation  
153 block5c\_se\_squeeze  
154 block5c\_se\_reshape  
155 block5c\_se\_reduce  
156 block5c\_se\_expand  
157 block5c\_se\_excite  
158 block5c\_project\_conv  
159 block5c\_project\_bn  
160 block5c\_drop  
161 block5c\_add  
162 block6a\_expand\_conv  
163 block6a\_expand\_bn  
164 block6a\_expand\_activation  
165 block6a\_dwconv\_pad  
166 block6a\_dwconv  
167 block6a\_bn  
168 block6a\_activation  
169 block6a\_se\_squeeze  
170 block6a\_se\_reshape  
171 block6a\_se\_reduce  
172 block6a\_se\_expand  
173 block6a\_se\_excite

```
174 block6a_project_conv
175 block6a_project_bn
176 block6b_expand_conv
177 block6b_expand_bn
178 block6b_expand_activation
179 block6b_dwconv
180 block6b_bn
181 block6b_activation
182 block6b_se_squeeze
183 block6b_se_reshape
184 block6b_se_reduce
185 block6b_se_expand
186 block6b_se_excite
187 block6b_project_conv
188 block6b_project_bn
189 block6b_drop
190 block6b_add
191 block6c_expand_conv
192 block6c_expand_bn
193 block6c_expand_activation
194 block6c_dwconv
195 block6c_bn
196 block6c_activation
197 block6c_se_squeeze
198 block6c_se_reshape
199 block6c_se_reduce
200 block6c_se_expand
201 block6c_se_excite
202 block6c_project_conv
203 block6c_project_bn
204 block6c_drop
205 block6c_add
206 block6d_expand_conv
207 block6d_expand_bn
208 block6d_expand_activation
209 block6d_dwconv
210 block6d_bn
211 block6d_activation
212 block6d_se_squeeze
213 block6d_se_reshape
214 block6d_se_reduce
215 block6d_se_expand
216 block6d_se_excite
217 block6d_project_conv
218 block6d_project_bn
219 block6d_drop
220 block6d_add
221 block7a_expand_conv
222 block7a_expand_bn
223 block7a_expand_activation
224 block7a_dwconv
225 block7a_bn
226 block7a_activation
227 block7a_se_squeeze
228 block7a_se_reshape
229 block7a_se_reduce
230 block7a_se_expand
231 block7a_se_excite
232 block7a_project_conv
233 block7a_project_bn
234 top_conv
235 top_bn
236 top_activation
```

**Wow, that's a lot of layers... to handcode all of those would've taken a fairly long time to do, yet we can still take advantage of them thanks to the power of transfer learning.**

**How about a summary of the base model?**

In [ ]:

```
base_model.summary()
```

```
Model: "efficientnetb0"
```

Layer (type)	Output Shape	Param #	Connected to
<hr/>			
<hr/>			
input_1 (InputLayer)	[ (None, None, None, 0		
<hr/>			
rescaling (Rescaling)	(None, None, None, 3 0		input_1[0][0]
<hr/>			
normalization (Normalization)	(None, None, None, 3 7		rescaling[0][0]
<hr/>			
stem_conv_pad (ZeroPadding2D)	(None, None, None, 3 0		normalization[0][0]
<hr/>			
stem_conv (Conv2D)	(None, None, None, 3 864		stem_conv_pad[0][0]
<hr/>			
stem_bn (BatchNormalization)	(None, None, None, 3 128		stem_conv[0][0]
<hr/>			
stem_activation (Activation)	(None, None, None, 3 0		stem_bn[0][0]
<hr/>			
block1a_dwconv (DepthwiseConv2D)	(None, None, None, 3 288		stem_activation[0][0]
<hr/>			
block1a_bn (BatchNormalization)	(None, None, None, 3 128		block1a_dwconv[0][0]
<hr/>			
block1a_activation (Activation)	(None, None, None, 3 0		block1a_bn[0][0]
<hr/>			
block1a_se_squeeze (GlobalAvera	(None, 32)	0	block1a_activation[0][0]
<hr/>			
block1a_se_reshape (Reshape)	(None, 1, 1, 32)	0	block1a_se_squeeze[0][0]
<hr/>			
block1a_se_reduce (Conv2D)	(None, 1, 1, 8)	264	block1a_se_reshape[0][0]
<hr/>			
block1a_se_expand (Conv2D)	(None, 1, 1, 32)	288	block1a_se_reduce[0][0]
<hr/>			
block1a_se_excite (Multiply)	(None, None, None, 3 0		block1a_activation[0][0]
]			block1a_se_expand[0][0]
]			
block1a_project_conv (Conv2D)	(None, None, None, 1 512		block1a_se_excite[0][0]

block1a_project_bn (BatchNormal (None, None, None, 1 64 0])		block1a_project_conv[0][0]
block2a_expand_conv (Conv2D) (None, None, None, 9 1536)		block1a_project_bn[0][0]
block2a_expand_bn (BatchNormali (None, None, None, 9 384 )]		block2a_expand_conv[0][0]
block2a_expand_activation (Acti (None, None, None, 9 0)		block2a_expand_bn[0][0]
block2a_dwconv_pad (ZeroPadding (None, None, None, 9 0 n[0][0])		block2a_expand_activation[0][0]
block2a_dwconv (DepthwiseConv2D (None, None, None, 9 864)		block2a_dwconv_pad[0][0]
block2a_bn (BatchNormalization) (None, None, None, 9 384)		block2a_dwconv[0][0]
block2a_activation (Activation) (None, None, None, 9 0)		block2a_bn[0][0]
block2a_se_squeeze (GlobalAvera (None, 96)	0	block2a_activation[0][0]
block2a_se_reshape (Reshape) (None, 1, 1, 96)	0	block2a_se_squeeze[0][0]
block2a_se_reduce (Conv2D) (None, 1, 1, 4)	388	block2a_se_reshape[0][0]
block2a_se_expand (Conv2D) (None, 1, 1, 96)	480	block2a_se_reduce[0][0]
block2a_se_excite (Multiply) (None, None, None, 9 0 )]		block2a_activation[0][0]
block2a_se_excite (Multiply) (None, None, None, 9 0 )]		block2a_se_expand[0][0]
block2a_project_conv (Conv2D) (None, None, None, 2 2304)		block2a_se_excite[0][0]
block2a_project_bn (BatchNormal (None, None, None, 2 96 0])		block2a_project_conv[0][0]
block2b_expand_conv (Conv2D) (None, None, None, 1 3456)		block2a_project_bn[0][0]
block2b_expand_bn (BatchNormali (None, None, None, 1 576 )]		block2b_expand_conv[0][0]

block2b_expand_activation (Acti (None, None, None, 1 0		block2b_expand_bn[0] [0]
block2b_dwconv (DepthwiseConv2D (None, None, None, 1 1296		block2b_expand_activation[0] [0]
block2b_bn (BatchNormalization) (None, None, None, 1 576		block2b_dwconv[0] [0]
block2b_activation (Activation) (None, None, None, 1 0		block2b_bn[0] [0]
block2b_se_squeeze (GlobalAvera (None, 144)	0	block2b_activation[0] [0]
block2b_se_reshape (Reshape) (None, 1, 1, 144)	0	block2b_se_squeeze[0] [0]
block2b_se_reduce (Conv2D) (None, 1, 1, 6)	870	block2b_se_reshape[0] [0]
block2b_se_expand (Conv2D) (None, 1, 1, 144)	1008	block2b_se_reduce[0] [0]
block2b_se_excite (Multiply) (None, None, None, 1 0		block2b_activation[0] [0]
]		block2b_se_expand[0] [0]
block2b_project_conv (Conv2D) (None, None, None, 2 3456		block2b_se_excite[0] [0]
block2b_project_bn (BatchNormal (None, None, None, 2 96		block2b_project_conv[0] [0]
block2b_drop (Dropout) (None, None, None, 2 0		block2b_project_bn[0] [0]
block2b_add (Add) (None, None, None, 2 0		block2b_drop[0] [0]
]		block2a_project_bn[0] [0]
block3a_expand_conv (Conv2D) (None, None, None, 1 3456		block2b_add[0] [0]
block3a_expand_bn (BatchNormali (None, None, None, 1 576		block3a_expand_conv[0] [0]
]		
block3a_expand_activation (Acti (None, None, None, 1 0		block3a_expand_bn[0] [0]
block3a_dwconv_pad (ZeroPadding (None, None, None, 1 0		block3a_expand_activation[0] [0]

block3a_dwconv (DepthwiseConv2D (None, None, None, 1 3600			block3a_dwconv_pad[0][0]
block3a_bn (BatchNormalization) (None, None, None, 1 576			block3a_dwconv[0][0]
block3a_activation (Activation) (None, None, None, 1 0			block3a_bn[0][0]
block3a_se_squeeze (GlobalAvera (None, 144)	0		block3a_activation[0][0]
block3a_se_reshape (Reshape) (None, 1, 1, 144)	0		block3a_se_squeeze[0][0]
block3a_se_reduce (Conv2D) (None, 1, 1, 6)	870		block3a_se_reshape[0][0]
block3a_se_expand (Conv2D) (None, 1, 1, 144)	1008		block3a_se_reduce[0][0]
block3a_se_excite (Multiply) (None, None, None, 1 0			block3a_activation[0][0]
]			block3a_se_expand[0][0]
]			
block3a_project_conv (Conv2D) (None, None, None, 4 5760			block3a_se_excite[0][0]
block3a_project_bn (BatchNormal (None, None, None, 4 160			block3a_project_conv[0][0]
0)			
block3b_expand_conv (Conv2D) (None, None, None, 2 9600			block3a_project_bn[0][0]
block3b_expand_bn (BatchNormali (None, None, None, 2 960			block3b_expand_conv[0][0]
)]			
block3b_expand_activation (Acti (None, None, None, 2 0			block3b_expand_bn[0][0]
block3b_dwconv (DepthwiseConv2D (None, None, None, 2 6000			block3b_expand_activation[0][0]
n[0][0])			
block3b_bn (BatchNormaliz (None, None, None, 2 960			block3b_dwconv[0][0]
)			
block3b_activation (Activation) (None, None, None, 2 0			block3b_bn[0][0]
block3b_se_squeeze (GlobalAvera (None, 240)	0		block3b_activation[0][0]
)			
block3b_se_reshape (Reshape) (None, 1, 1, 240)	0		block3b_se_squeeze[0][0]
)			

block3b_se_reduce (Conv2D)	(None, 1, 1, 10)	2410	block3b_se_reshape[0][0]
block3b_se_expand (Conv2D)	(None, 1, 1, 240)	2640	block3b_se_reduce[0][0]
block3b_se_excite (Multiply)	(None, None, None, 20)		block3b_activation[0][0]
]			block3b_se_expand[0][0]
]			
block3b_project_conv (Conv2D)	(None, None, None, 4)	9600	block3b_se_excite[0][0]
block3b_project_bn (BatchNormal)	(None, None, None, 4)	160	block3b_project_conv[0][0]
block3b_drop (Dropout)	(None, None, None, 4)	0	block3b_project_bn[0][0]
block3b_add (Add)	(None, None, None, 4)	0	block3b_drop[0][0]
block3b	[0]		block3a_project_bn[0][0]
block4a_expand_conv (Conv2D)	(None, None, None, 2)	9600	block3b_add[0][0]
block4a_expand_bn (BatchNormali	(None, None, None, 2)	960	block4a_expand_conv[0][0]
]			
block4a_expand_activation (Acti	(None, None, None, 2)	0	block4a_expand_bn[0][0]
block4a_dwconv_pad (ZeroPadding	(None, None, None, 2)	0	block4a_expand_activation[0][0]
n[0][0]			
block4a_dwconv (DepthwiseConv2D)	(None, None, None, 2)	2160	block4a_dwconv_pad[0][0]
block4a_bn (BatchNormalization)	(None, None, None, 2)	960	block4a_dwconv[0][0]
block4a_activation (Activation)	(None, None, None, 2)	0	block4a_bn[0][0]
block4a_se_squeeze (GlobalAvera	(None, 240)	0	block4a_activation[0][0]
ge			
block4a_se_reshape (Reshape)	(None, 1, 1, 240)	0	block4a_se_squeeze[0][0]
block4a_se_reduce (Conv2D)	(None, 1, 1, 10)	2410	block4a_se_reshape[0][0]

block4a_se_expand (Conv2D)	(None, 1, 1, 240)	2640	block4a_se_reduce[0][0]
block4a_se_excite (Multiply)	(None, None, None, 20)	block4a_activation[0][0]	
]			block4a_se_expand[0][0]
]			
block4a_project_conv (Conv2D)	(None, None, None, 8)	19200	block4a_se_excite[0][0]
block4a_project_bn (BatchNormal)	(None, None, None, 8)	320	block4a_project_conv[0][0]
0]			
block4b_expand_conv (Conv2D)	(None, None, None, 4)	38400	block4a_project_bn[0][0]
block4b_expand_bn (BatchNormali	(None, None, None, 4)	1920	block4b_expand_conv[0][0]
]			
block4b_expand_activation (Acti	(None, None, None, 40)	0	block4b_expand_bn[0][0]
block4b_dwconv (DepthwiseConv2D)	(None, None, None, 4)	4320	block4b_expand_activatio
n[0][0]			
block4b_bn (BatchNormalizatio	(None, None, None, 4)	1920	block4b_dwconv[0][0]
n)			
block4b_activation (Activation)	(None, None, None, 40)	0	block4b_bn[0][0]
block4b_se_squeeze (GlobalAvera	(None, 480)	0	block4b_activation[0][0]
n)			
block4b_se_reshape (Reshape)	(None, 1, 1, 480)	0	block4b_se_squeeze[0][0]
block4b_se_reduce (Conv2D)	(None, 1, 1, 20)	9620	block4b_se_reshape[0][0]
block4b_se_expand (Conv2D)	(None, 1, 1, 480)	10080	block4b_se_reduce[0][0]
block4b_se_excite (Multiply)	(None, None, None, 40)	0	block4b_activation[0][0]
]			
]			block4b_se_expand[0][0]
block4b_project_conv (Conv2D)	(None, None, None, 8)	38400	block4b_se_excite[0][0]
block4b_project_bn (BatchNormal	(None, None, None, 8)	320	block4b_project_conv[0][0]
]			

0]

block4b_drop (Dropout)	(None, None, None, 8 0	block4b_project_bn[0] [0]
block4b_add (Add)	(None, None, None, 8 0	block4b_drop[0] [0]
block4a_project_bn[0] [0]		
block4c_expand_conv (Conv2D)	(None, None, None, 4 38400	block4b_add[0] [0]
block4c_expand_bn (BatchNormali	(None, None, None, 4 1920	block4c_expand_conv[0] [0]
]		
block4c_expand_activation (Acti	(None, None, None, 4 0	block4c_expand_bn[0] [0]
n[0] [0]		
block4c_dwconv (DepthwiseConv2D	(None, None, None, 4 4320	block4c_expand_activatio
n[0] [0]		
block4c_bn (BatchNormalizatio	(None, None, None, 4 1920	block4c_dwconv[0] [0]
n)		
block4c_activation (Activation)	(None, None, None, 4 0	block4c_bn[0] [0]
block4c_se_squeeze (GlobalAvera	(None, 480)	0
ge)		
block4c_se_reshape (Reshape)	(None, 1, 1, 480)	0
block4c_se_reduce (Conv2D)	(None, 1, 1, 20)	9620
block4c_se_expand (Conv2D)	(None, 1, 1, 480)	10080
block4c_se_excite (Multiply)	(None, None, None, 4 0	block4c_activation[0] [0]
]		
block4c_se_expand[0] [0]		
]		
block4c_project_conv (Conv2D)	(None, None, None, 8 38400	block4c_se_excite[0] [0]
block4c_project_bn (BatchNormal	(None, None, None, 8 320	block4c_project_conv[0] [0]
al)		
block4c_drop (Dropout)	(None, None, None, 8 0	block4c_project_bn[0] [0]
]		
block4c_add (Add)	(None, None, None, 8 0	block4c_drop[0] [0]

block4b\_add[0][0]

---

block5a\_expand\_conv (Conv2D) (None, None, None, 4 38400 block4c\_add[0][0]

---

block5a\_expand\_bn (BatchNormali (None, None, None, 4 1920 block5a\_expand\_conv[0][0]  
])

---

block5a\_expand\_activation (Acti (None, None, None, 4 0 block5a\_expand\_bn[0][0]

---

block5a\_dwconv (DepthwiseConv2D (None, None, None, 4 12000 block5a\_expand\_activatio  
n[0][0])

---

block5a\_bn (BatchNormalization) (None, None, None, 4 1920 block5a\_dwconv[0][0]

---

block5a\_activation (Activation) (None, None, None, 4 0 block5a\_bn[0][0]

---

block5a\_se\_squeeze (GlobalAvera (None, 480) 0 block5a\_activation[0][0]

---

block5a\_se\_reshape (Reshape) (None, 1, 1, 480) 0 block5a\_se\_squeeze[0][0]

---

block5a\_se\_reduce (Conv2D) (None, 1, 1, 20) 9620 block5a\_se\_reshape[0][0]

---

block5a\_se\_expand (Conv2D) (None, 1, 1, 480) 10080 block5a\_se\_reduce[0][0]

---

block5a\_se\_excite (Multiply) (None, None, None, 4 0 block5a\_activation[0][0]  
]  
block5a\_se\_expand[0][0]  
]

---

block5a\_project\_conv (Conv2D) (None, None, None, 1 53760 block5a\_se\_excite[0][0]

---

block5a\_project\_bn (BatchNormal (None, None, None, 1 448 block5a\_project\_conv[0][  
0])

---

block5b\_expand\_conv (Conv2D) (None, None, None, 6 75264 block5a\_project\_bn[0][0])

---

block5b\_expand\_bn (BatchNormali (None, None, None, 6 2688 block5b\_expand\_conv[0][0]  
])

---

block5b\_expand\_activation (Acti (None, None, None, 6 0 block5b\_expand\_bn[0][0])

---

block5b\_dwconv (DepthwiseConv2D (None, None, None, 6 16800 block5b\_expand\_activatio  
n[0][0])

n[0][0]			
block5b_bn (BatchNormalization) (None, None, None, 6 2688)			block5b_dwconv[0][0]
block5b_activation (Activation) (None, None, None, 6 0)			block5b_bn[0][0]
block5b_se_squeeze (GlobalAvera (None, 672)	0		block5b_activation[0][0]
block5b_se_reshape (Reshape) (None, 1, 1, 672)	0		block5b_se_squeeze[0][0]
block5b_se_reduce (Conv2D) (None, 1, 1, 28)	18844		block5b_se_reshape[0][0]
block5b_se_expand (Conv2D) (None, 1, 1, 672)	19488		block5b_se_reduce[0][0]
block5b_se_excite (Multiply) (None, None, None, 6 0)			block5b_activation[0][0]
]			block5b_se_expand[0][0]
]			
block5b_project_conv (Conv2D) (None, None, None, 1 75264)			block5b_se_excite[0][0]
block5b_project_bn (BatchNormal (None, None, None, 1 448			block5b_project_conv[0][0]
0)			
block5b_drop (Dropout) (None, None, None, 1 0)			block5b_project_bn[0][0]
]			
block5b_add (Add) (None, None, None, 1 0)			block5b_drop[0][0]
0)			block5a_project_bn[0][0]
block5c_expand_conv (Conv2D) (None, None, None, 6 75264)			block5b_add[0][0]
block5c_expand_bn (BatchNormali (None, None, None, 6 2688)			block5c_expand_conv[0][0]
]			
block5c_expand_activation (Acti (None, None, None, 6 0)			block5c_expand_bn[0][0]
block5c_dwconv (DepthwiseConv2D (None, None, None, 6 16800			block5c_expand_activation[0][0]
n[0][0]			
block5c_bn (BatchNormalization) (None, None, None, 6 2688)			block5c_dwconv[0][0]
block5c_activation (Activation) (None, None, None, 6 0)			block5c_bn[0][0]

block5c_se_squeeze (GlobalAvera (None, 672)	0	block5c_activation[0][0]
block5c_se_reshape (Reshape) (None, 1, 1, 672)	0	block5c_se_squeeze[0][0]
block5c_se_reduce (Conv2D) (None, 1, 1, 28)	18844	block5c_se_reshape[0][0]
block5c_se_expand (Conv2D) (None, 1, 1, 672)	19488	block5c_se_reduce[0][0]
block5c_se_excite (Multiply) (None, None, None, 6 0 ]	block5c_activation[0][0] block5c_se_expand[0][0]	
block5c_project_conv (Conv2D) (None, None, None, 1 75264		block5c_se_excite[0][0]
block5c_project_bn (BatchNormal (None, None, None, 1 448 0)		block5c_project_conv[0][0]
block5c_drop (Dropout) (None, None, None, 1 0 ]		block5c_project_bn[0][0]
block5c_add (Add) (None, None, None, 1 0		block5c_drop[0][0] block5b_add[0][0]
block6a_expand_conv (Conv2D) (None, None, None, 6 75264		block5c_add[0][0]
block6a_expand_bn (BatchNormali (None, None, None, 6 2688 ]		block6a_expand_conv[0][0]
block6a_expand_activation (Acti (None, None, None, 6 0		block6a_expand_bn[0][0]
block6a_dwconv_pad (ZeroPadding (None, None, None, 6 0 n[0][0])		block6a_expand_activation
block6a_dwconv (DepthwiseConv2D (None, None, None, 6 16800		block6a_dwconv_pad[0][0]
block6a_bn (BatchNormalization) (None, None, None, 6 2688		block6a_dwconv[0][0]
block6a_activation (Activation) (None, None, None, 6 0		block6a_bn[0][0]
block6a_se_squeeze (GlobalAvera (None, 672)	0	block6a_activation[0][0]

block6a_se_reshape (Reshape)	(None, 1, 1, 672)	0	block6a_se_squeeze[0][0]
block6a_se_reduce (Conv2D)	(None, 1, 1, 28)	18844	block6a_se_reshape[0][0]
block6a_se_expand (Conv2D)	(None, 1, 1, 672)	19488	block6a_se_reduce[0][0]
block6a_se_excite (Multiply)	(None, None, None, 6)	0	block6a_activation[0][0]
			block6a_se_expand[0][0]
block6a_project_conv (Conv2D)	(None, None, None, 1)	129024	block6a_se_excite[0][0]
block6a_project_bn (BatchNormal	(None, None, None, 1)	768	block6a_project_conv[0][0]
block6b_expand_conv (Conv2D)	(None, None, None, 1)	221184	block6a_project_bn[0][0]
block6b_expand_bn (BatchNormali	(None, None, None, 1)	4608	block6b_expand_conv[0][0]
block6b_expand_activation (Acti	(None, None, None, 1)	0	block6b_expand_bn[0][0]
block6b_dwconv (DepthwiseConv2D	(None, None, None, 1)	28800	block6b_expand_activation[0][0]
block6b_bn (BatchNormalization)	(None, None, None, 1)	4608	block6b_dwconv[0][0]
block6b_activation (Activation)	(None, None, None, 1)	0	block6b_bn[0][0]
block6b_se_squeeze (GlobalAvera	(None, 1152)	0	block6b_activation[0][0]
block6b_se_reshape (Reshape)	(None, 1, 1, 1152)	0	block6b_se_squeeze[0][0]
block6b_se_reduce (Conv2D)	(None, 1, 1, 48)	55344	block6b_se_reshape[0][0]
block6b_se_expand (Conv2D)	(None, 1, 1, 1152)	56448	block6b_se_reduce[0][0]
block6b_se_excite (Multiply)	(None, None, None, 1)	0	block6b_activation[0][0]
			block6b_se_expand[0][0]

]

block6b_project_conv (Conv2D)	(None, None, None, 1 221184	block6b_se_excite[0][0]	
block6b_project_bn (BatchNormal (None, None, None, 1 768	0]	block6b_project_conv[0][0]	
block6b_drop (Dropout)	(None, None, None, 1 0	block6b_project_bn[0][0]	
block6b_add (Add)	(None, None, None, 1 0	block6b_drop[0][0]	
block6a_project_bn[0][0]	0]	block6a_project_bn[0][0]	
block6c_expand_conv (Conv2D)	(None, None, None, 1 221184	block6b_add[0][0]	
block6c_expand_bn (BatchNormali (None, None, None, 1 4608	]	block6c_expand_conv[0][0]	
block6c_expand_activation (Acti (None, None, None, 1 0	0]	block6c_expand_bn[0][0]	
block6c_dwconv (DepthwiseConv2D (None, None, None, 1 28800	n[0][0]	block6c_expand_activation[0][0]	
block6c_bn (BatchNormalization)	(None, None, None, 1 4608	block6c_dwconv[0][0]	
block6c_activation (Activation)	(None, None, None, 1 0	block6c_bn[0][0]	
block6c_se_squeeze (GlobalAvera (None, 1152)	0	block6c_activation[0][0]	
block6c_se_reshape (Reshape)	(None, 1, 1, 1152)	0	block6c_se_squeeze[0][0]
block6c_se_reduce (Conv2D)	(None, 1, 1, 48)	55344	block6c_se_reshape[0][0]
block6c_se_expand (Conv2D)	(None, 1, 1, 1152)	56448	block6c_se_reduce[0][0]
block6c_se_excite (Multiply)	(None, None, None, 1 0	block6c_activation[0][0]	
]		block6c_se_expand[0][0]	
block6c_project_conv (Conv2D)	(None, None, None, 1 221184	block6c_se_excite[0][0]	
block6c_project_bn (BatchNormal (None, None, None, 1 768	]	block6c_project_conv[0][0]	

0]

block6c_drop (Dropout)	(None, None, None, 1 0	block6c_project_bn[0] [0]
block6c_add (Add)	(None, None, None, 1 0	block6c_drop[0] [0]
block6c_add (Add)		block6b_add[0] [0]
block6d_expand_conv (Conv2D)	(None, None, None, 1 221184	block6c_add[0] [0]
block6d_expand_bn (BatchNormali	(None, None, None, 1 4608	block6d_expand_conv[0] [0]
block6d_expand_activation (Acti	(None, None, None, 1 0	block6d_expand_bn[0] [0]
block6d_dwconv (DepthwiseConv2D	(None, None, None, 1 28800	block6d_expand_activatio
n[0] [0]		n[0] [0]
block6d_bn (BatchNormalizatio	(None, None, None, 1 4608	block6d_dwconv[0] [0]
block6d_activation (Activation)	(None, None, None, 1 0	block6d_bn[0] [0]
block6d_se_squeeze (GlobalAvera	(None, 1152) 0	block6d_activation[0] [0]
block6d_se_reshape (Reshape)	(None, 1, 1, 1152) 0	block6d_se_squeeze[0] [0]
block6d_se_reduce (Conv2D)	(None, 1, 1, 48) 55344	block6d_se_reshape[0] [0]
block6d_se_expand (Conv2D)	(None, 1, 1, 1152) 56448	block6d_se_reduce[0] [0]
block6d_se_excite (Multiply)	(None, None, None, 1 0	block6d_activation[0] [0]
]		block6d_se_expand[0] [0]
block6d_project_conv (Conv2D)	(None, None, None, 1 221184	block6d_se_excite[0] [0]
block6d_project_bn (BatchNormal	(None, None, None, 1 768	block6d_project_conv[0] [0]
0)		
block6d_drop (Dropout)	(None, None, None, 1 0	block6d_project_bn[0] [0]
block6d_add (Add)	(None, None, None, 1 0	block6d_drop[0] [0]

block6c\_add[0][0]

---

block7a\_expand\_conv (Conv2D) (None, None, None, 1 221184 block6d\_add[0][0]

---

block7a\_expand\_bn (BatchNormali (None, None, None, 1 4608 block7a\_expand\_conv[0][0]  
])

---

block7a\_expand\_activation (Acti (None, None, None, 1 0 block7a\_expand\_bn[0][0]

---

block7a\_dwconv (DepthwiseConv2D (None, None, None, 1 10368 block7a\_expand\_activation  
n[0][0])

---

block7a\_bn (BatchNormalization) (None, None, None, 1 4608 block7a\_dwconv[0][0]

---

block7a\_activation (Activation) (None, None, None, 1 0 block7a\_bn[0][0]

---

block7a\_se\_squeeze (GlobalAvera (None, 1152) 0 block7a\_activation[0][0]

---

block7a\_se\_reshape (Reshape) (None, 1, 1, 1152) 0 block7a\_se\_squeeze[0][0]

---

block7a\_se\_reduce (Conv2D) (None, 1, 1, 48) 55344 block7a\_se\_reshape[0][0]

---

block7a\_se\_expand (Conv2D) (None, 1, 1, 1152) 56448 block7a\_se\_reduce[0][0]

---

block7a\_se\_excite (Multiply) (None, None, None, 1 0 block7a\_activation[0][0]  
]  
block7a\_se\_expand[0][0]  
]

---

block7a\_project\_conv (Conv2D) (None, None, None, 3 368640 block7a\_se\_excite[0][0]

---

block7a\_project\_bn (BatchNormal (None, None, None, 3 1280 block7a\_project\_conv[0][  
0])

---

top\_conv (Conv2D) (None, None, None, 1 409600 block7a\_project\_bn[0][0])

---

top\_bn (BatchNormalization) (None, None, None, 1 5120 top\_conv[0][0])

---

top\_activation (Activation) (None, None, None, 1 0 top\_bn[0][0])

---

=====

=====

Total params: 4,049,571

Trainable params: 0  
Non-trainable params: 4,049,571

---

You can see how each of the different layers have a certain number of parameters each. Since we are using a pre-trained model, you can think of all of these parameters as patterns the base model has learned on another dataset. And because we set `base_model.trainable = False`, these patterns remain as they are during training (they're frozen and don't get updated).

Alright that was the base model, let's see the summary of our overall model.

In [ ]:

```
# Check summary of model constructed with Functional API
model_0.summary()
```

Model: "model"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	[None, 224, 224, 3]	0
efficientnetb0 (Functional)	(None, None, None, 1280)	4049571
global_average_pooling_layer (GlobalAveragePooling2D)	(None, 1280)	0
output_layer (Dense)	(None, 10)	12810

Total params: 4,062,381  
Trainable params: 12,810  
Non-trainable params: 4,049,571

Our overall model has five layers but really, one of those layers (`efficientnetb0`) has 236 layers.

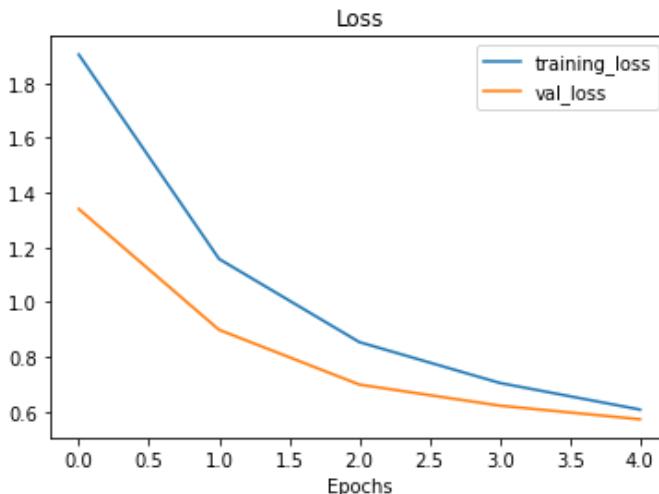
You can see how the output shape started out as `(None, 224, 224, 3)` for the input layer (the shape of our images) but was transformed to be `(None, 10)` by the output layer (the shape of our labels), where `None` is the placeholder for the batch size.

Notice too, the only trainable parameters in the model are those in the output layer.

How do our model's training curves look?

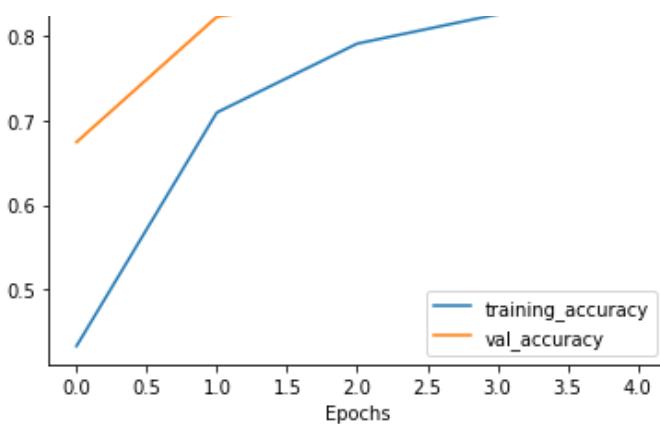
In [ ]:

```
# Check out our model's training curves
plot_loss_curves(history_10_percent)
```



Accuracy





## Getting a feature vector from a trained model

Question: What happens with the `tf.keras.layers.GlobalAveragePooling2D()` layer? I haven't seen it before.

The `tf.keras.layers.GlobalAveragePooling2D()` layer transforms a 4D tensor into a 2D tensor by averaging the values across the inner-axes.

The previous sentence is a bit of a mouthful, so let's see an example.

In [ ]:

```
# Define input tensor shape (same number of dimensions as the output of efficientnetb0)
input_shape = (1, 4, 4, 3)

# Create a random tensor
tf.random.set_seed(42)
input_tensor = tf.random.normal(input_shape)
print(f"Random input tensor:\n {input_tensor}\n")

# Pass the random tensor through a global average pooling 2D layer
global_average_pooled_tensor = tf.keras.layers.GlobalAveragePooling2D()(input_tensor)
print(f"2D global average pooled random tensor:\n {global_average_pooled_tensor}\n")

# Check the shapes of the different tensors
print(f"Shape of input tensor: {input_tensor.shape}")
print(f"Shape of 2D global averaged pooled input tensor: {global_average_pooled_tensor.shape}")
```

Random input tensor:

```
[[[[-0.3274685 -0.8426258  0.3194337]
  [-1.4075519 -2.3880599 -1.0392479]
  [-0.5573232  0.539707   1.6994323]
  [ 0.28893656 -1.5066116 -0.2645474]]]

[[[-0.59722406 -1.9171132 -0.62044144]
  [ 0.8504023  -0.40604794 -3.0258412]
  [ 0.9058464  0.29855987 -0.22561555]
  [-0.7616443  -1.891714   -0.9384712]]]

[[[ 0.77852213 -0.47338897  0.97772694]
  [ 0.24694404  0.20573747 -0.5256233]
  [ 0.32410017  0.02545409 -0.10638497]
  [-0.6369475  1.1603122   0.2507359]]]

[[[-0.41728497  0.40125778 -1.4145442]
  [-0.5931857  -1.6617213   0.33567193]
  [ 0.10815629  0.2347968  -0.56668764]
  [-0.35819843  0.88698614  0.52744764]]]]
```

2D global average pooled random tensor:

```
[[-0.09368646 -0.45840448 -0.2885598]]
```

Shape of input tensor: (1, 4, 4, 3)

Shape of 2D global averaged pooled input tensor: (1, 3)

You can see the `tf.keras.layers.GlobalAveragePooling2D()` layer condensed the input tensor from shape (1, 4, 4, 3) to (1, 3). It did so by averaging the `input_tensor` across the middle two axes.

We can replicate this operation using the `tf.reduce_mean()` operation and specifying the appropriate axes.

In [ ]:

```
# This is the same as GlobalAveragePooling2D()
tf.reduce_mean(input_tensor, axis=[1, 2]) # average across the middle axes
```

Out[ ]:

```
<tf.Tensor: shape=(1, 3), dtype=float32, numpy=array([-0.09368646, -0.45840448, -0.2885598]), dtype=float32>
```

Doing this not only makes the output of the base model compatible with the input shape requirement of our output layer (`tf.keras.layers.Dense()`), it also condenses the information found by the base model into a lower dimension **feature vector**.

□ Note: One of the reasons feature extraction transfer learning is named how it is because what often happens is a pretrained model outputs a **feature vector** (a long tensor of numbers, in our case, this is the output of the `tf.keras.layers.GlobalAveragePooling2D()` layer) which can then be used to extract patterns out of.

□ Practice: Do the same as the above cell but for `tf.keras.layers.GlobalMaxPool2D()`.

## Running a series of transfer learning experiments

We've seen the incredible results of transfer learning on 10% of the training data, what about 1% of the training data?

What kind of results do you think we can get using 100x less data than the original CNN models we built ourselves?

Why don't we answer that question while running the following modelling experiments:

1. `model_1`: Use feature extraction transfer learning on 1% of the training data with data augmentation.
2. `model_2`: Use feature extraction transfer learning on 10% of the training data with data augmentation.
3. `model_3`: Use fine-tuning transfer learning on 10% of the training data with data augmentation.
4. `model_4`: Use fine-tuning transfer learning on 100% of the training data with data augmentation.

While all of the experiments will be run on different versions of the training data, they will all be evaluated on the same test dataset, this ensures the results of each experiment are as comparable as possible.

All experiments will be done using the `EfficientNetB0` model within the `tf.keras.applications` module.

To make sure we're keeping track of our experiments, we'll use our `create_tensorboard_callback()` function to log all of the model training logs.

We'll construct each model using the Keras Functional API and instead of implementing data augmentation in the `ImageDataGenerator` class as we have previously, we're going to build it right into the model using the `tf.keras.layers.experimental.preprocessing` module.

Let's begin by downloading the data for experiment 1, using feature extraction transfer learning on 1% of the training data with data augmentation.

In [ ]:

```
# Download and unzip data
!wget https://storage.googleapis.com/ztm_tf_course/food_vision/10_food_classes_1_percent
```

```

.zip
unzip_data("10_food_classes_1_percent.zip")

# Create training and test dirs
train_dir_1_percent = "10_food_classes_1_percent/train/"
test_dir = "10_food_classes_1_percent/test/"

--2021-02-16 02:15:55-- https://storage.googleapis.com/ztm_tf_course/food_vision/10_food_classes_1_percent.zip
Resolving storage.googleapis.com (storage.googleapis.com)... 172.217.7.208, 172.217.9.208, 172.217.15.112, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|172.217.7.208|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 133612354 (127M) [application/zip]
Saving to: '10_food_classes_1_percent.zip.1'

10_food_classes_1_p 100%[=====] 127.42M 194MB/s in 0.7s

2021-02-16 02:15:56 (194 MB/s) - '10_food_classes_1_percent.zip.1' saved [133612354/133612354]

```

## How many images are we working with?

In [ ]:

```
# Walk through 1 percent data directory and list number of files
walk_through_dir("10_food_classes_1_percent")
```

```

There are 2 directories and 0 images in '10_food_classes_1_percent'.
There are 10 directories and 0 images in '10_food_classes_1_percent/train'.
There are 0 directories and 7 images in '10_food_classes_1_percent/train/ice_cream'.
There are 0 directories and 7 images in '10_food_classes_1_percent/train/ramen'.
There are 0 directories and 7 images in '10_food_classes_1_percent/train/chicken_wings'.
There are 0 directories and 7 images in '10_food_classes_1_percent/train/pizza'.
There are 0 directories and 7 images in '10_food_classes_1_percent/train/steak'.
There are 0 directories and 7 images in '10_food_classes_1_percent/train/fried_rice'.
There are 0 directories and 7 images in '10_food_classes_1_percent/train/hamburger'.
There are 0 directories and 7 images in '10_food_classes_1_percent/train/grilled_salmon'.
There are 0 directories and 7 images in '10_food_classes_1_percent/train/sushi'.
There are 0 directories and 7 images in '10_food_classes_1_percent/train/chicken_curry'.
There are 10 directories and 0 images in '10_food_classes_1_percent/test'.
There are 0 directories and 250 images in '10_food_classes_1_percent/test/ice_cream'.
There are 0 directories and 250 images in '10_food_classes_1_percent/test/ramen'.
There are 0 directories and 250 images in '10_food_classes_1_percent/test/chicken_wings'.
There are 0 directories and 250 images in '10_food_classes_1_percent/test/pizza'.
There are 0 directories and 250 images in '10_food_classes_1_percent/test/steak'.
There are 0 directories and 250 images in '10_food_classes_1_percent/test/fried_rice'.
There are 0 directories and 250 images in '10_food_classes_1_percent/test/hamburger'.
There are 0 directories and 250 images in '10_food_classes_1_percent/test/grilled_salmon'.
.
There are 0 directories and 250 images in '10_food_classes_1_percent/test/sushi'.
There are 0 directories and 250 images in '10_food_classes_1_percent/test/chicken_curry'.

```

**Alright, looks like we've only got seven images of each class, this should be a bit of a challenge for our model.**

**▀ Note: As with the 10% of data subset, the 1% of images were chosen at random from the original full training dataset. The test images are the same as the ones which have previously been used. If you want to see how this data was preprocessed, check out the [Food Vision Image Preprocessing notebook](#).**

**Time to load our images in as `tf.data.Dataset` objects, to do so, we'll use the `image_dataset_from_directory()` method.**

In [ ]:

```
import tensorflow as tf
```

```
IMG_SIZE = (224, 224)
train_data_1_percent = tf.keras.preprocessing.image_dataset_from_directory(train_dir_1_percent,
label_mode="categorical",
batch_size=3
2, # default
image_size=IMG_SIZE)
test_data = tf.keras.preprocessing.image_dataset_from_directory(test_dir,
label_mode="categorical",
image_size=IMG_SIZE)
```

Found 70 files belonging to 10 classes.  
Found 2500 files belonging to 10 classes.

## Data loaded. Time to augment it.

## **Adding data augmentation right into the model**

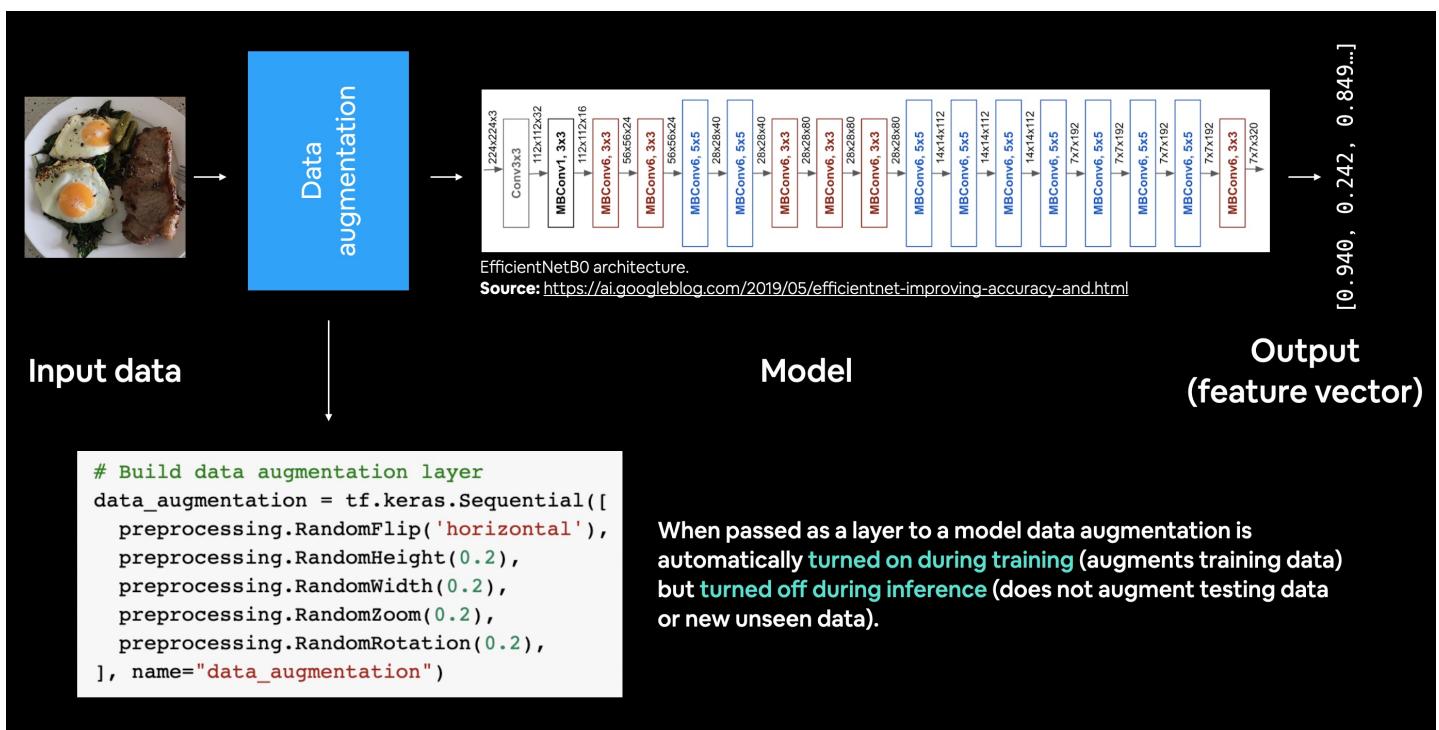
Previously we've used the different parameters of the `ImageDataGenerator` class to augment our training images, this time we're going to build data augmentation right into the model.

## How?

Using the `tf.keras.layers.experimental.preprocessing` module and creating a dedicated data augmentation layer.

This is a relatively new feature added to TensorFlow 2.2+ but it's very powerful. Adding a data augmentation layer to the model has the following benefits:

- Preprocessing of the images (augmenting them) happens on the GPU rather than on the CPU (much faster).
    - Images are best preprocessed on the GPU where as text and structured data are more suited to be preprocessed on the CPU.
  - Image data augmentation only happens during training so we can still export our whole model and use it elsewhere. And if someone else wanted to train the same model as us, including the same kind of data augmentation, they could.



**Example of using data augmentation as the first layer within a model (EfficientNetB0).**

**Note:** At the time of writing, the preprocessing layers we're using for data augmentation are in *experimental* status within the `tf.image` library. This means although the layers should be considered stable, the code may change slightly in a future version of TensorFlow. For more

information on the other preprocessing layers available and the different methods of data augmentation, check out the [Keras preprocessing layers guide](#) and the [TensorFlow data augmentation guide](#).

To use data augmentation right within our model we'll create a Keras Sequential model consisting of only data preprocessing layers, we can then use this Sequential model within another Functional model.

If that sounds confusing, it'll make sense once we create it in code.

The data augmentation transformations we're going to use are:

- [RandomFlip](#) - flips image on horizontal or vertical axis.
- [RandomRotation](#) - randomly rotates image by a specified amount.
- [RandomZoom](#) - randomly zooms into an image by specified amount.
- [RandomHeight](#) - randomly shifts image height by a specified amount.
- [RandomWidth](#) - randomly shifts image width by a specified amount.
- [Rescaling](#) - normalizes the image pixel values to be between 0 and 1, this is worth mentioning because it is required for some image models but since we're using the `tf.keras.applications` implementation of EfficientNetB0, it's not required.

There are more option but these will do for now.

In [ ]:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.layers.experimental import preprocessing

# Create a data augmentation stage with horizontal flipping, rotations, zooms
data_augmentation = keras.Sequential([
    preprocessing.RandomFlip("horizontal"),
    preprocessing.RandomRotation(0.2),
    preprocessing.RandomZoom(0.2),
    preprocessing.RandomHeight(0.2),
    preprocessing.RandomWidth(0.2),
    # preprocessing.Rescaling(1./255) # keep for ResNet50V2, remove for EfficientNetB0
], name ="data_augmentation")
```

And that's it! Our data augmentation Sequential model is ready to go. As you'll see shortly, we'll be able to slot this "model" as a layer into our transfer learning model later on.

But before we do that, let's test it out by passing random images through it.

In [ ]:

```
# View a random image
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import os
import random
target_class = random.choice(train_data_1_percent.class_names) # choose a random class
target_dir = "10_food_classes_1_percent/train/" + target_class # create the target directory
random_image = random.choice(os.listdir(target_dir)) # choose a random image from target directory
random_image_path = target_dir + "/" + random_image # create the chosen random image path
img = mpimg.imread(random_image_path) # read in the chosen target image
plt.imshow(img) # plot the target image
plt.title(f"Original random image from class: {target_class}")
plt.axis(False); # turn off the axes

# Augment the image
augmented_img = data_augmentation(tf.expand_dims(img, axis=0)) # data augmentation model requires shape (None, height, width, 3)
plt.figure()
```

```
plt.imshow(tf.squeeze(augmented_img)/255.) # requires normalization after augmentation
plt.title(f"Augmented random image from class: {target_class}")
plt.axis(False);
```

Original random image from class: grilled\_salmon



Augmented random image from class: grilled\_salmon



Run the cell above a few times and you can see the different random augmentations on different classes of images. Because we're going to add the data augmentation model as a layer in our upcoming transfer learning model, it'll apply these kind of random augmentations to each of the training images which passes through it.

Doing this will make our training dataset a little more varied. You can think of it as if you were taking a photo of food in real-life, not all of the images are going to be perfect, some of them are going to be orientated in strange ways. These are the kind of images we want our model to be able to handle.

Speaking of model, let's build one with the Functional API. We'll run through all of the same steps as before except for one difference, we'll add our data augmentation Sequential model as a layer immediately after the input layer.

## Model 1: Feature extraction transfer learning on 1% of the data with data augmentation

In [ ]:

```
# Setup input shape and base model, freezing the base model layers
input_shape = (224, 224, 3)
base_model = tf.keras.applications.EfficientNetB0(include_top=False)
base_model.trainable = False

# Create input layer
inputs = layers.Input(shape=input_shape, name="input_layer")

# Add in data augmentation Sequential model as a layer
x = data_augmentation(inputs)

# Give base_model inputs (after augmentation) and don't train it
x = base_model(x, training=False)

# Pool output features of base model
x = layers.GlobalAveragePooling2D(name="global_average_pooling_layer")(x)
```

```

# Put a dense layer on as the output
outputs = layers.Dense(10, activation="softmax", name="output_layer") (x)

# Make a model with inputs and outputs
model_1 = keras.Model(inputs, outputs)

# Compile the model
model_1.compile(loss="categorical_crossentropy",
                 optimizer=tf.keras.optimizers.Adam(),
                 metrics=["accuracy"])

# Fit the model
history_1_percent = model_1.fit(train_data_1_percent,
                                 epochs=5,
                                 steps_per_epoch=len(train_data_1_percent),
                                 validation_data=test_data,
                                 validation_steps=int(0.25* len(test_data)), # validate for less step
                                 callbacks=[create_tensorboard_callback("transfer_learning", "1_percent_data_aug")])

```

Saving TensorBoard log files to: transfer\_learning/1\_percent\_data\_aug/20210216-021736

Epoch 1/5  
 3/3 [=====] - 10s 2s/step - loss: 2.4533 - accuracy: 0.0299 - val\_loss: 2.2204 - val\_accuracy: 0.1645

Epoch 2/5  
 3/3 [=====] - 4s 2s/step - loss: 2.1931 - accuracy: 0.0897 - val\_loss: 2.0827 - val\_accuracy: 0.2796

Epoch 3/5  
 3/3 [=====] - 4s 2s/step - loss: 1.9338 - accuracy: 0.4006 - val\_loss: 1.9872 - val\_accuracy: 0.3536

Epoch 4/5  
 3/3 [=====] - 4s 2s/step - loss: 1.8485 - accuracy: 0.4954 - val\_loss: 1.8808 - val\_accuracy: 0.4095

Epoch 5/5  
 3/3 [=====] - 4s 2s/step - loss: 1.7186 - accuracy: 0.5845 - val\_loss: 1.7857 - val\_accuracy: 0.5099

**Wow! How cool is that? Using only 7 training images per class, using transfer learning our model was able to get ~40% accuracy on the validation set. This result is pretty amazing since the [original Food-101 paper](#) achieved 50.67% accuracy with all the data, namely, 750 training images per class (note: this metric was across 101 classes, not 10, we'll get to 101 classes soon).**

If we check out a summary of our model, we should see the data augmentation layer just after the input layer.

In [ ]:

```
# Check out model summary
model_1.summary()
```

Model: "model\_1"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	[(None, 224, 224, 3)]	0
data_augmentation (Sequentia)	(None, None, None, 3)	0
efficientnetb0 (Functional)	(None, None, None, 1280)	4049571
global_average_pooling_layer (None, 1280)		0
output_layer (Dense)	(None, 10)	12810
<hr/>		
Total params: 4,062,381		
Trainable params: 12,810		
Non-trainable params: 4,049,571		

There it is. We've now got data augmentation built right into the our model. This means if we saved it and reloaded it somewhere else, the data augmentation layers would come with it.

The important thing to remember is **data augmentation only runs during training**. So if we were to evaluate or use our model for inference (predicting the class of an image) the data augmentation layers will be automatically turned off.

To see this in action, let's evaluate our model on the test data.

In [ ]:

```
# Evaluate on the test data
results_1_percent_data_aug = model_1.evaluate(test_data)
results_1_percent_data_aug
```

79/79 [=====] - 10s 122ms/step - loss: 1.8055 - accuracy: 0.4728

Out[ ]:

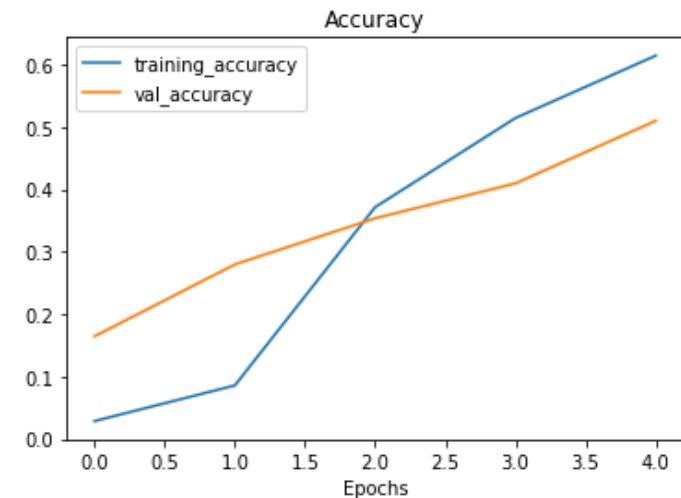
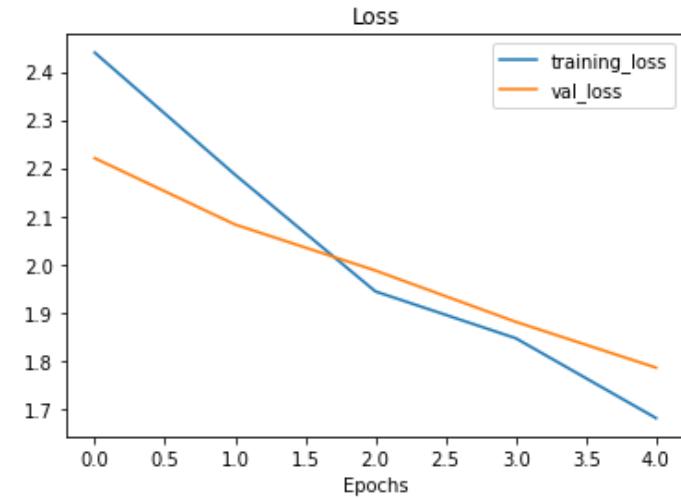
```
[1.8054912090301514, 0.47279998660087585]
```

The results here may be slightly better/worse than the log outputs of our model during training because during training we only evaluate our model on 25% of the test data using the line `validation_steps=int(0.25 * len(test_data))`. Doing this speeds up our epochs but still gives us enough of an idea of how our model is going.

Let's stay consistent and check out our model's loss curves.

In [ ]:

```
# How does the model go with a data augmentation layer with 1% of data
plot_loss_curves(history_1_percent)
```



It looks like the metrics on both datasets would improve if we kept training for more epochs. But we'll leave that

for now, we've got more experiments to do!

## Model 2: Feature extraction transfer learning with 10% of data and data augmentation

Alright, we've tested 1% of the training data with data augmentation, how about we try 10% of the data with data augmentation?

But wait...

Question: How do you know what experiments to run?

Great question.

The truth here is you often won't. Machine learning is still a very experimental practice. It's only after trying a fair few things that you'll start to develop an intuition of what to try.

My advice is to follow your curiosity as tenaciously as possible. If you feel like you want to try something, write the code for it and run it. See how it goes. The worst thing that'll happen is you'll figure out what doesn't work, the most valuable kind of knowledge.

From a practical standpoint, as we've talked about before, you'll want to reduce the amount of time between your initial experiments as much as possible. In other words, run a plethora of smaller experiments, using less data and less training iterations before you find something promising and then scale it up.

In the theme of scale, let's scale our 1% training data augmentation experiment up to 10% training data augmentation. That sentence doesn't really make sense but you get what I mean.

We're going to run through the exact same steps as the previous model, the only difference being using 10% of the training data instead of 1%.

In [ ]:

```
# Get 10% of the data of the 10 classes (uncomment if you haven't gotten "10_food_classes_10_percent.zip" already)
# !wget https://storage.googleapis.com/ztm_tf_course/food_vision/10_food_classes_10_percent.zip
# unzip_data("10_food_classes_10_percent.zip")

train_dir_10_percent = "10_food_classes_10_percent/train/"
test_dir = "10_food_classes_10_percent/test/"
```

Data downloaded. Let's create the dataloaders.

In [ ]:

```
# Setup data inputs
import tensorflow as tf
IMG_SIZE = (224, 224)
train_data_10_percent = tf.keras.preprocessing.image_dataset_from_directory(train_dir_10_percent,
                                                               label_mode="categorical",
                                                               image_size=IMG_SIZE)
# Note: the test data is the same as the previous experiment, we could
# skip creating this, but we'll leave this here to practice.
test_data = tf.keras.preprocessing.image_dataset_from_directory(test_dir,
                                                               label_mode="categorical",
                                                               image_size=IMG_SIZE)
```

Found 750 files belonging to 10 classes.  
Found 2500 files belonging to 10 classes.

Awesome! We've got 10x more images to work with, 75 per class instead of 7 per class.

Let's build a model with data augmentation built in. We could reuse the data augmentation Sequential model we created before but we'll recreate it to practice.

In [ ]:

```
# Create a functional model with data augmentation
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.layers.experimental import preprocessing
from tensorflow.keras.models import Sequential

# Build data augmentation layer
data_augmentation = Sequential([
    preprocessing.RandomFlip('horizontal'),
    preprocessing.RandomHeight(0.2),
    preprocessing.RandomWidth(0.2),
    preprocessing.RandomZoom(0.2),
    preprocessing.RandomRotation(0.2),
    # preprocessing.Rescaling(1./255) # keep for ResNet50V2, remove for EfficientNet
], name="data_augmentation")

# Setup the input shape to our model
input_shape = (224, 224, 3)

# Create a frozen base model
base_model = tf.keras.applications.EfficientNetB0(include_top=False)
base_model.trainable = False

# Create input and output layers
inputs = layers.Input(shape=input_shape, name="input_layer") # create input layer
x = data_augmentation(inputs) # augment our training images
x = base_model(x, training=False) # pass augmented images to base model but keep it in inference mode, so batchnorm layers don't get updated: https://keras.io/guides/transfer_learning/#build-a-model
x = layers.GlobalAveragePooling2D(name="global_average_pooling_layer")(x)
outputs = layers.Dense(10, activation="softmax", name="output_layer")(x)
model_2 = tf.keras.Model(inputs, outputs)

# Compile
model_2.compile(loss="categorical_crossentropy",
                  optimizer=tf.keras.optimizers.Adam(lr=0.001), # use Adam optimizer with base learning rate
                  metrics=["accuracy"])
```

## Creating a ModelCheckpoint callback

Our model is compiled and ready to be fit, so why haven't we fit it yet?

Well, for this experiment we're going to introduce a new callback, the `ModelCheckpoint` callback.

The `ModelCheckpoint` callback gives you the ability to save your model, as a whole in the `SavedModel` format or the `weights (patterns) only` to a specified directory as it trains.

This is helpful if you think your model is going to be training for a long time and you want to make backups of it as it trains. It also means if you think your model could benefit from being trained for longer, you can reload it from a specific checkpoint and continue training from there.

For example, say you fit a feature extraction transfer learning model for 5 epochs and you check the training curves and see it was still improving and you want to see if fine-tuning for another 5 epochs could help, you can load the checkpoint, unfreeze some (or all) of the base model layers and then continue training.

In fact, that's exactly what we're going to do.

But first, let's create a `ModelCheckpoint` callback. To do so, we have to specify a directory we'd like to save to.

In [ ]:

```
# Setup checkpoint path
checkpoint_path = "ten_percent_model_checkpoints_weights/checkpoint.ckpt" # note: remember saving directly to Colab is temporary

# Create a ModelCheckpoint callback that saves the model's weights only
checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_path,
                                                       save_weights_only=True, # set
                                                       to False to save the entire model
                                                       save_best_only=False, # set to
                                                       True to save only the best model instead of a model every epoch
                                                       save_freq="epoch", # save every
                                                       epoch
                                                       verbose=1)
```

## Question: What's the difference between saving the entire model (SavedModel format) and saving the weights only?

The [SavedModel](#) format saves a model's architecture, weights and training configuration all in one folder. It makes it very easy to reload your model exactly how it is elsewhere. However, if you do not want to share all of these details with others, you may want to save and share the weights only (these will just be large tensors of non-human interpretable numbers). If disk space is an issue, saving the weights only is faster and takes up less space than saving the whole model.

Time to fit the model.

Because we're going to be fine-tuning it later, we'll create a variable `initial_epochs` and set it to 5 to use later.

We'll also add in our `checkpoint_callback` in our list of `callbacks`.

In [ ]:

```
# Fit the model saving checkpoints every epoch
initial_epochs = 5
history_10_percent_data_aug = model_2.fit(train_data_10_percent,
                                             epochs=initial_epochs,
                                             validation_data=test_data,
                                             validation_steps=int(0.25 * len(test_data)), #
                                             callbacks=[create_tensorboard_callback("transfer_learning"),
                                                        "10_percent_data_aug"),
                                                        checkpoint_callback])
```

Saving TensorBoard log files to: transfer\_learning/10\_percent\_data\_aug/20210216-021854  
Epoch 1/5  
24/24 [=====] - 16s 452ms/step - loss: 2.1809 - accuracy: 0.2156  
- val\_loss: 1.5099 - val\_accuracy: 0.6299

Epoch 00001: saving model to ten\_percent\_model\_checkpoints\_weights/checkpoint.ckpt  
Epoch 2/5  
24/24 [=====] - 9s 358ms/step - loss: 1.4534 - accuracy: 0.6360  
- val\_loss: 1.0699 - val\_accuracy: 0.7582

Epoch 00002: saving model to ten\_percent\_model\_checkpoints\_weights/checkpoint.ckpt  
Epoch 3/5  
24/24 [=====] - 9s 375ms/step - loss: 1.0684 - accuracy: 0.7427  
- val\_loss: 0.8586 - val\_accuracy: 0.7878

Epoch 00003: saving model to ten\_percent\_model\_checkpoints\_weights/checkpoint.ckpt  
Epoch 4/5  
24/24 [=====] - 8s 338ms/step - loss: 0.9191 - accuracy: 0.7720  
- val\_loss: 0.7781 - val\_accuracy: 0.7993

Epoch 00004: saving model to ten\_percent\_model\_checkpoints\_weights/checkpoint.ckpt  
Epoch 5/5  
24/24 [=====] - 8s 332ms/step - loss: 0.7953 - accuracy: 0.7950

```
- val_loss: 0.7229 - val_accuracy: 0.7944
```

```
Epoch 00005: saving model to ten_percent_model_checkpoints_weights/checkpoint.ckpt
```

**Would you look at that! Looks like our `ModelCheckpoint` callback worked and our model saved its weights every epoch without too much overhead (saving the whole model takes longer than just the weights).**

Let's evaluate our model and check its loss curves.

In [ ]:

```
# Evaluate on the test data
results_10_percent_data_aug = model_2.evaluate(test_data)
results_10_percent_data_aug
```

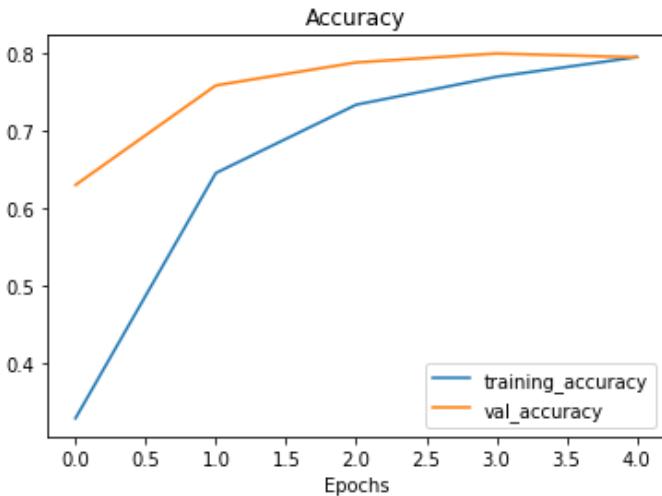
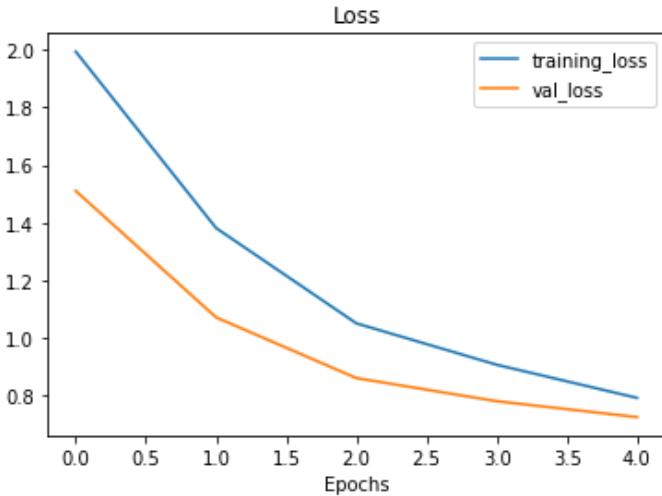
```
79/79 [=====] - 10s 119ms/step - loss: 0.7047 - accuracy: 0.8080
```

Out [ ]:

```
[0.7046541571617126, 0.8080000281333923]
```

In [ ]:

```
# Plot model loss curves
plot_loss_curves(history_10_percent_data_aug)
```



Looking at these, our model's performance with 10% of the data and data augmentation isn't as good as the model with 10% of the data without data augmentation (see `model_0` results above), however the curves are trending in the right direction, meaning if we decided to train for longer, its metrics would likely improve.

Since we checkpointed (is that a word?) our model's weights, we might as well see what it's like to load it back in. We'll be able to test if it saved correctly by evaluating it on the test data.

To load saved model weights you can use the the `load_weights()` method, passing it the path where your saved weights are stored.

In [ ]:

```
# Load in saved model weights and evaluate model
```

```
model_2.load_weights(checkpoint_path)
```

```
loaded_weights_model_results = model_2.evaluate(test_data)
```

```
79/79 [=====] - 10s 118ms/step - loss: 0.7047 - accuracy: 0.8080
```

**Now let's compare the results of our previously trained model and the loaded model. These results should very close if not exactly the same. The reason for minor differences comes down to the precision level of numbers calculated.**

In [ ]:

```
# If the results from our native model and the loaded weights are the same, this should output True
```

```
results_10_percent_data_aug == loaded_weights_model_results
```

Out [ ]:

```
False
```

**If the above cell doesn't output `True`, it's because the numbers are close but not the `exact` same (due to how computers store numbers with degrees of precision).**

**However, they should be `very` close...**

In [ ]:

```
import numpy as np
```

```
# Check to see if loaded model results are very close to native model results (should output True)
```

```
np.isclose(np.array(results_10_percent_data_aug), np.array(loaded_weights_model_results))
```

Out [ ]:

```
array([ True,  True])
```

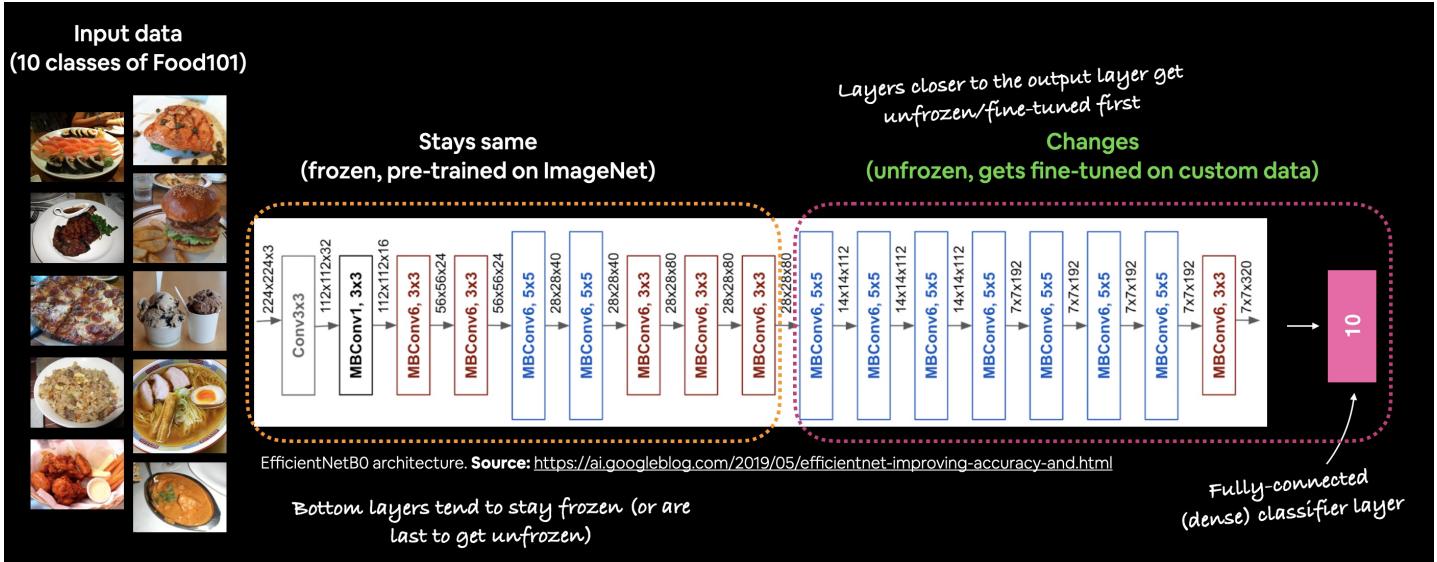
In [ ]:

```
# Check the difference between the two results
```

```
print(np.array(results_10_percent_data_aug) - np.array(loaded_weights_model_results))
```

```
[-1.1920929e-07  0.0000000e+00]
```

## Model 3: Fine-tuning an existing model on 10% of the data



**High-level example of fine-tuning an EfficientNet model. Bottom layers (layers closer to the input data) stay frozen where as top layers (layers closer to the output data) are updated during training.**

*Frozen where as top layers (layers closer to the output data) are updated during training.*

So far our saved model has been trained using feature extraction transfer learning for 5 epochs on 10% of the training data and data augmentation.

This means all of the layers in the base model (EfficientNetB0) were frozen during training.

For our next experiment we're going to switch to fine-tuning transfer learning. This means we'll be using the same base model except we'll be unfreezing some of its layers (ones closest to the top) and running the model for a few more epochs.

The idea with fine-tuning is to start customizing the pre-trained model more to our own data.

**Note:** Fine-tuning usually works best *after* training a feature extraction model for a few epochs and with large amounts of data. For more on this, check out [Keras' guide on Transfer learning & fine-tuning](#).

We've verified our loaded model's performance, let's check out its layers.

In [ ]:

```
# Layers in loaded model
model_2.layers
```

Out [ ]:

```
[<tensorflow.python.keras.engine.input_layer.InputLayer at 0x7fceca1c9208>,
<tensorflow.python.keras.engine.sequential.Sequential at 0x7fceca1c90b8>,
<tensorflow.python.keras.engine.functional.Functional at 0x7fcebe5bc630>,
<tensorflow.python.keras.layers.pooling.GlobalAveragePooling2D at 0x7fcebe62b8d0>,
<tensorflow.python.keras.core.Dense at 0x7fcebe5f1e80>]
```

In [ ]:

```
for layer in model_2.layers:
    print(layer.trainable)
```

```
True
True
False
True
True
```

Looking good. We've got an input layer, a Sequential layer (the data augmentation model), a Functional layer (EfficientNetB0), a pooling layer and a Dense layer (the output layer).

How about a summary?

In [ ]:

```
model_2.summary()
```

```
Model: "model_2"
```

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	[(None, 224, 224, 3)]	0
data_augmentation (Sequential)	(None, None, None, 3)	0
efficientnetb0 (Functional)	(None, None, None, 1280)	4049571
global_average_pooling_layer (None, 1280)		0
output_layer (Dense)	(None, 10)	12810

Total params: 4,062,381  
Trainable params: 12,810  
Non-trainable params: 4,049,571

Alright, it looks like all of the layers in the `efficientnetb0` layer are frozen. We can confirm this using the `trainable_variables` attribute.

In [ ]:

```
# How many layers are trainable in our base model?  
print(len(model_2.layers[2].trainable_variables)) # layer at index 2 is the EfficientNetB  
0 layer (the base model)
```

0

**This is the same as our base model.**

In [ ]:

```
print(len(base_model.trainable_variables))
```

0

We can even check layer by layer to see if they're trainable.

In [ ]:

```
# Check which layers are tuneable (trainable)
for layer_number, layer in enumerate(base_model.layers):
    print(layer_number, layer.name, layer.trainable)
```

```
0 input_3 False
1 rescaling_2 False
2 normalization_2 False
3 stem_conv_pad False
4 stem_conv False
5 stem_bn False
6 stem_activation False
7 block1a_dwconv False
8 block1a_bn False
9 block1a_activation False
10 block1a_se_squeeze False
11 block1a_se_reshape False
12 block1a_se_reduce False
13 block1a_se_expand False
14 block1a_se_excite False
15 block1a_project_conv False
16 block1a_project_bn False
17 block2a_expand_conv False
18 block2a_expand_bn False
19 block2a_expand_activation False
20 block2a_dwconv_pad False
21 block2a_dwconv False
22 block2a_bn False
23 block2a_activation False
24 block2a_se_squeeze False
25 block2a_se_reshape False
26 block2a_se_reduce False
27 block2a_se_expand False
28 block2a_se_excite False
29 block2a_project_conv False
30 block2a_project_bn False
31 block2b_expand_conv False
32 block2b_expand_bn False
33 block2b_expand_activation False
34 block2b_dwconv False
35 block2b_bn False
36 block2b_activation False
37 block2b_se_squeeze False
38 block2b_se_reshape False
39 block2b_se_reduce False
40 block2b_se_expand False
41 block2b_se_init False
```

```
41 block2b_se_excite False
42 block2b_project_conv False
43 block2b_project_bn False
44 block2b_drop False
45 block2b_add False
46 block3a_expand_conv False
47 block3a_expand_bn False
48 block3a_expand_activation False
49 block3a_dwconv_pad False
50 block3a_dwconv False
51 block3a_bn False
52 block3a_activation False
53 block3a_se_squeeze False
54 block3a_se_reshape False
55 block3a_se_reduce False
56 block3a_se_expand False
57 block3a_se_excite False
58 block3a_project_conv False
59 block3a_project_bn False
60 block3b_expand_conv False
61 block3b_expand_bn False
62 block3b_expand_activation False
63 block3b_dwconv False
64 block3b_bn False
65 block3b_activation False
66 block3b_se_squeeze False
67 block3b_se_reshape False
68 block3b_se_reduce False
69 block3b_se_expand False
70 block3b_se_excite False
71 block3b_project_conv False
72 block3b_project_bn False
73 block3b_drop False
74 block3b_add False
75 block4a_expand_conv False
76 block4a_expand_bn False
77 block4a_expand_activation False
78 block4a_dwconv_pad False
79 block4a_dwconv False
80 block4a_bn False
81 block4a_activation False
82 block4a_se_squeeze False
83 block4a_se_reshape False
84 block4a_se_reduce False
85 block4a_se_expand False
86 block4a_se_excite False
87 block4a_project_conv False
88 block4a_project_bn False
89 block4b_expand_conv False
90 block4b_expand_bn False
91 block4b_expand_activation False
92 block4b_dwconv False
93 block4b_bn False
94 block4b_activation False
95 block4b_se_squeeze False
96 block4b_se_reshape False
97 block4b_se_reduce False
98 block4b_se_expand False
99 block4b_se_excite False
100 block4b_project_conv False
101 block4b_project_bn False
102 block4b_drop False
103 block4b_add False
104 block4c_expand_conv False
105 block4c_expand_bn False
106 block4c_expand_activation False
107 block4c_dwconv False
108 block4c_bn False
109 block4c_activation False
110 block4c_se_squeeze False
111 block4c_se_reshape False
112 block4c_se_reduce False
```

```
113 block4c_se_expand raise
114 block4c_se_excite False
115 block4c_project_conv False
116 block4c_project_bn False
117 block4c_drop False
118 block4c_add False
119 block5a_expand_conv False
120 block5a_expand_bn False
121 block5a_expand_activation False
122 block5a_dwconv False
123 block5a_bn False
124 block5a_activation False
125 block5a_se_squeeze False
126 block5a_se_reshape False
127 block5a_se_reduce False
128 block5a_se_expand False
129 block5a_se_excite False
130 block5a_project_conv False
131 block5a_project_bn False
132 block5b_expand_conv False
133 block5b_expand_bn False
134 block5b_expand_activation False
135 block5b_dwconv False
136 block5b_bn False
137 block5b_activation False
138 block5b_se_squeeze False
139 block5b_se_reshape False
140 block5b_se_reduce False
141 block5b_se_expand False
142 block5b_se_excite False
143 block5b_project_conv False
144 block5b_project_bn False
145 block5b_drop False
146 block5b_add False
147 block5c_expand_conv False
148 block5c_expand_bn False
149 block5c_expand_activation False
150 block5c_dwconv False
151 block5c_bn False
152 block5c_activation False
153 block5c_se_squeeze False
154 block5c_se_reshape False
155 block5c_se_reduce False
156 block5c_se_expand False
157 block5c_se_excite False
158 block5c_project_conv False
159 block5c_project_bn False
160 block5c_drop False
161 block5c_add False
162 block6a_expand_conv False
163 block6a_expand_bn False
164 block6a_expand_activation False
165 block6a_dwconv_pad False
166 block6a_dwconv False
167 block6a_bn False
168 block6a_activation False
169 block6a_se_squeeze False
170 block6a_se_reshape False
171 block6a_se_reduce False
172 block6a_se_expand False
173 block6a_se_excite False
174 block6a_project_conv False
175 block6a_project_bn False
176 block6b_expand_conv False
177 block6b_expand_bn False
178 block6b_expand_activation False
179 block6b_dwconv False
180 block6b_bn False
181 block6b_activation False
182 block6b_se_squeeze False
183 block6b_se_reshape False
184 block6b_se_reduce False
185 block6b_se_expand False
```

```
185 block6a_se_expand False
186 block6b_se_excite False
187 block6b_project_conv False
188 block6b_project_bn False
189 block6b_drop False
190 block6b_add False
191 block6c_expand_conv False
192 block6c_expand_bn False
193 block6c_expand_activation False
194 block6c_dwconv False
195 block6c_bn False
196 block6c_activation False
197 block6c_se_squeeze False
198 block6c_se_reshape False
199 block6c_se_reduce False
200 block6c_se_expand False
201 block6c_se_excite False
202 block6c_project_conv False
203 block6c_project_bn False
204 block6c_drop False
205 block6c_add False
206 block6d_expand_conv False
207 block6d_expand_bn False
208 block6d_expand_activation False
209 block6d_dwconv False
210 block6d_bn False
211 block6d_activation False
212 block6d_se_squeeze False
213 block6d_se_reshape False
214 block6d_se_reduce False
215 block6d_se_expand False
216 block6d_se_excite False
217 block6d_project_conv False
218 block6d_project_bn False
219 block6d_drop False
220 block6d_add False
221 block7a_expand_conv False
222 block7a_expand_bn False
223 block7a_expand_activation False
224 block7a_dwconv False
225 block7a_bn False
226 block7a_activation False
227 block7a_se_squeeze False
228 block7a_se_reshape False
229 block7a_se_reduce False
230 block7a_se_expand False
231 block7a_se_excite False
232 block7a_project_conv False
233 block7a_project_bn False
234 top_conv False
235 top_bn False
236 top_activation False
```

**Beautiful. This is exactly what we're after.**

**Now to fine-tune the base model to our own data, we're going to unfreeze the top 10 layers and continue training our model for another 5 epochs.**

**This means all of the base model's layers except for the last 10 will remain frozen and untrainable. And the weights in the remaining unfrozen layers will be updated during training.**

**Ideally, we should see the model's performance improve.**

#### QUESTION: How many layers should you unfreeze when training?

**There's no set rule for this. You could unfreeze every layer in the pretrained model or you could try unfreezing one layer at a time. Best to experiment with different amounts of unfreezing and fine-tuning to see what happens. Generally, the less data you have, the less layers you want to unfreeze and the more gradually you want to fine-tune.**

Resource: The [ULMFiT \(Universal Language Model Fine-tuning for Text Classification\) paper](#) has a great series of experiments on fine-tuning models.

To begin fine-tuning, we'll unfreeze the entire base model by setting its `trainable` attribute to `True`. Then we'll refreeze every layer in the base model except for the last 10 by looping through them and setting their `trainable` attribute to `False`. Finally, we'll recompile the model.

In [ ]:

```
base_model.trainable = True

# Freeze all layers except for the
for layer in base_model.layers[:-10]:
    layer.trainable = False

# Recompile the model (always recompile after any adjustments to a model)
model_2.compile(loss="categorical_crossentropy",
                 optimizer=tf.keras.optimizers.Adam(lr=0.0001), # lr is 10x lower than before for fine-tuning
                 metrics=["accuracy"])
```

Wonderful, now let's check which layers of the pretrained model are trainable.

In [ ]:

```
# Check which layers are tuneable (trainable)
for layer_number, layer in enumerate(base_model.layers):
    print(layer_number, layer.name, layer.trainable)
```

```
0 input_3 False
1 rescaling_2 False
2 normalization_2 False
3 stem_conv_pad False
4 stem_conv False
5 stem_bn False
6 stem_activation False
7 block1a_dwconv False
8 block1a_bn False
9 block1a_activation False
10 block1a_se_squeeze False
11 block1a_se_reshape False
12 block1a_se_reduce False
13 block1a_se_expand False
14 block1a_se_excite False
15 block1a_project_conv False
16 block1a_project_bn False
17 block2a_expand_conv False
18 block2a_expand_bn False
19 block2a_expand_activation False
20 block2a_dwconv_pad False
21 block2a_dwconv False
22 block2a_bn False
23 block2a_activation False
24 block2a_se_squeeze False
25 block2a_se_reshape False
26 block2a_se_reduce False
27 block2a_se_expand False
28 block2a_se_excite False
29 block2a_project_conv False
30 block2a_project_bn False
31 block2b_expand_conv False
32 block2b_expand_bn False
33 block2b_expand_activation False
34 block2b_dwconv False
35 block2b_bn False
36 block2b_activation False
37 block2b_se_squeeze False
38 block2b_se_reshape False
39 block2b_se_reduce False
```

```
40 block2b_se_expand False
41 block2b_se_excite False
42 block2b_project_conv False
43 block2b_project_bn False
44 block2b_drop False
45 block2b_add False
46 block3a_expand_conv False
47 block3a_expand_bn False
48 block3a_expand_activation False
49 block3a_dwconv_pad False
50 block3a_dwconv False
51 block3a_bn False
52 block3a_activation False
53 block3a_se_squeeze False
54 block3a_se_reshape False
55 block3a_se_reduce False
56 block3a_se_expand False
57 block3a_se_excite False
58 block3a_project_conv False
59 block3a_project_bn False
60 block3b_expand_conv False
61 block3b_expand_bn False
62 block3b_expand_activation False
63 block3b_dwconv False
64 block3b_bn False
65 block3b_activation False
66 block3b_se_squeeze False
67 block3b_se_reshape False
68 block3b_se_reduce False
69 block3b_se_expand False
70 block3b_se_excite False
71 block3b_project_conv False
72 block3b_project_bn False
73 block3b_drop False
74 block3b_add False
75 block4a_expand_conv False
76 block4a_expand_bn False
77 block4a_expand_activation False
78 block4a_dwconv_pad False
79 block4a_dwconv False
80 block4a_bn False
81 block4a_activation False
82 block4a_se_squeeze False
83 block4a_se_reshape False
84 block4a_se_reduce False
85 block4a_se_expand False
86 block4a_se_excite False
87 block4a_project_conv False
88 block4a_project_bn False
89 block4b_expand_conv False
90 block4b_expand_bn False
91 block4b_expand_activation False
92 block4b_dwconv False
93 block4b_bn False
94 block4b_activation False
95 block4b_se_squeeze False
96 block4b_se_reshape False
97 block4b_se_reduce False
98 block4b_se_expand False
99 block4b_se_excite False
100 block4b_project_conv False
101 block4b_project_bn False
102 block4b_drop False
103 block4b_add False
104 block4c_expand_conv False
105 block4c_expand_bn False
106 block4c_expand_activation False
107 block4c_dwconv False
108 block4c_bn False
109 block4c_activation False
110 block4c_se_squeeze False
111 block4c_se_reshape False
```

```
112 block4c_se_reduce False
113 block4c_se_expand False
114 block4c_se_excite False
115 block4c_project_conv False
116 block4c_project_bn False
117 block4c_drop False
118 block4c_add False
119 block5a_expand_conv False
120 block5a_expand_bn False
121 block5a_expand_activation False
122 block5a_dwconv False
123 block5a_bn False
124 block5a_activation False
125 block5a_se_squeeze False
126 block5a_se_reshape False
127 block5a_se_reduce False
128 block5a_se_expand False
129 block5a_se_excite False
130 block5a_project_conv False
131 block5a_project_bn False
132 block5b_expand_conv False
133 block5b_expand_bn False
134 block5b_expand_activation False
135 block5b_dwconv False
136 block5b_bn False
137 block5b_activation False
138 block5b_se_squeeze False
139 block5b_se_reshape False
140 block5b_se_reduce False
141 block5b_se_expand False
142 block5b_se_excite False
143 block5b_project_conv False
144 block5b_project_bn False
145 block5b_drop False
146 block5b_add False
147 block5c_expand_conv False
148 block5c_expand_bn False
149 block5c_expand_activation False
150 block5c_dwconv False
151 block5c_bn False
152 block5c_activation False
153 block5c_se_squeeze False
154 block5c_se_reshape False
155 block5c_se_reduce False
156 block5c_se_expand False
157 block5c_se_excite False
158 block5c_project_conv False
159 block5c_project_bn False
160 block5c_drop False
161 block5c_add False
162 block6a_expand_conv False
163 block6a_expand_bn False
164 block6a_expand_activation False
165 block6a_dwconv_pad False
166 block6a_dwconv False
167 block6a_bn False
168 block6a_activation False
169 block6a_se_squeeze False
170 block6a_se_reshape False
171 block6a_se_reduce False
172 block6a_se_expand False
173 block6a_se_excite False
174 block6a_project_conv False
175 block6a_project_bn False
176 block6b_expand_conv False
177 block6b_expand_bn False
178 block6b_expand_activation False
179 block6b_dwconv False
180 block6b_bn False
181 block6b_activation False
182 block6b_se_squeeze False
183 block6b_se_reshape False
```

```
184 block6b_se_reduce False
185 block6b_se_expand False
186 block6b_se_excite False
187 block6b_project_conv False
188 block6b_project_bn False
189 block6b_drop False
190 block6b_add False
191 block6c_expand_conv False
192 block6c_expand_bn False
193 block6c_expand_activation False
194 block6c_dwconv False
195 block6c_bn False
196 block6c_activation False
197 block6c_se_squeeze False
198 block6c_se_reshape False
199 block6c_se_reduce False
200 block6c_se_expand False
201 block6c_se_excite False
202 block6c_project_conv False
203 block6c_project_bn False
204 block6c_drop False
205 block6c_add False
206 block6d_expand_conv False
207 block6d_expand_bn False
208 block6d_expand_activation False
209 block6d_dwconv False
210 block6d_bn False
211 block6d_activation False
212 block6d_se_squeeze False
213 block6d_se_reshape False
214 block6d_se_reduce False
215 block6d_se_expand False
216 block6d_se_excite False
217 block6d_project_conv False
218 block6d_project_bn False
219 block6d_drop False
220 block6d_add False
221 block7a_expand_conv False
222 block7a_expand_bn False
223 block7a_expand_activation False
224 block7a_dwconv False
225 block7a_bn False
226 block7a_activation False
227 block7a_se_squeeze True
228 block7a_se_reshape True
229 block7a_se_reduce True
230 block7a_se_expand True
231 block7a_se_excite True
232 block7a_project_conv True
233 block7a_project_bn True
234 top_conv True
235 top_bn True
236 top_activation True
```

Nice! It seems all layers except for the last 10 are frozen and untrainable. This means only the last 10 layers of the base model along with the output layer will have their weights updated during training.

#### Question: Why did we recompile the model?

Every time you make a change to your models, you need to recompile them.

In our case, we're using the exact same loss, optimizer and metrics as before, except this time the learning rate for our optimizer will be 10x smaller than before (0.0001 instead of Adam's default of 0.001).

We do this so the model doesn't try to overwrite the existing weights in the pretrained model too fast. In other words, we want learning to be more gradual.

Note: There's no set standard for setting the learning rate during fine-tuning, though reductions of **2.6x-10x+** seem to work well in practice .

## How many trainable variables do we have now?

In [ ]:

```
print(len(model_2.trainable_variables))
```

12

Wonderful, it looks like our model has a total of 10 trainable variables, the last 10 layers of the base model and the weight and bias parameters of the Dense output layer.

Time to fine-tune!

We're going to continue training on from where our previous model finished. Since it trained for 5 epochs, our fine-tuning will begin on the epoch 5 and continue for another 5 epochs.

To do this, we can use the `initial_epoch` parameter of the `fit()` method. We'll pass it the last epoch of the previous model's training history (`history_10_percent_data_aug.epoch[-1]`).

In [ ]:

```
# Fine tune for another 5 epochs
fine_tune_epochs = initial_epochs + 5

# Refit the model (same as model_2 except with more trainable layers)
history_fine_10_percent_data_aug = model_2.fit(train_data_10_percent,
                                                epochs=fine_tune_epochs,
                                                validation_data=test_data,
                                                initial_epoch=history_10_percent_data_aug
                                                .epoch[-1], # start from previous last epoch
                                                validation_steps=int(0.25 * len(test_data)),
                                                callbacks=[create_tensorboard_callback("transfer_learning", "10_percent_fine_tune_last_10")]) # name experiment appropriately
```

Saving TensorBoard log files to: transfer\_learning/10\_percent\_fine\_tune\_last\_10/20210216-022051  
Epoch 5/10  
24/24 [=====] - 15s 439ms/step - loss: 0.6963 - accuracy: 0.8062  
- val\_loss: 0.6032 - val\_accuracy: 0.8043  
Epoch 6/10  
24/24 [=====] - 8s 329ms/step - loss: 0.5747 - accuracy: 0.8390  
- val\_loss: 0.5580 - val\_accuracy: 0.8125  
Epoch 7/10  
24/24 [=====] - 8s 319ms/step - loss: 0.4972 - accuracy: 0.8458  
- val\_loss: 0.5543 - val\_accuracy: 0.8240  
Epoch 8/10  
24/24 [=====] - 8s 323ms/step - loss: 0.4262 - accuracy: 0.8814  
- val\_loss: 0.5403 - val\_accuracy: 0.8191  
Epoch 9/10  
24/24 [=====] - 8s 300ms/step - loss: 0.4234 - accuracy: 0.8855  
- val\_loss: 0.5262 - val\_accuracy: 0.8322  
Epoch 10/10  
24/24 [=====] - 8s 305ms/step - loss: 0.3665 - accuracy: 0.9056  
- val\_loss: 0.5218 - val\_accuracy: 0.8322

☐ Note: Fine-tuning usually takes far longer per epoch than feature extraction (due to updating more weights throughout a network).

Ho ho, looks like our model has gained a few percentage points of accuracy! Let's evaluate it.

In [ ]:

```
# Evaluate the model on the test data
results_fine_tune_10_percent = model_2.evaluate(test_data)
```

**Remember, the results from evaluating the model might be slightly different to the outputs from training since during training we only evaluate on 25% of the test data.**

**Alright, we need a way to evaluate our model's performance before and after fine-tuning. How about we write a function to compare the before and after?**

In [ ]:

```
def compare_histories(original_history, new_history, initial_epochs=5):
    """
    Compares two model history objects.
    """

    # Get original history measurements
    acc = original_history.history["accuracy"]
    loss = original_history.history["loss"]

    print(len(acc))

    val_acc = original_history.history["val_accuracy"]
    val_loss = original_history.history["val_loss"]

    # Combine original history with new history
    total_acc = acc + new_history.history["accuracy"]
    total_loss = loss + new_history.history["loss"]

    total_val_acc = val_acc + new_history.history["val_accuracy"]
    total_val_loss = val_loss + new_history.history["val_loss"]

    print(len(total_acc))
    print(total_acc)

    # Make plots
    plt.figure(figsize=(8, 8))
    plt.subplot(2, 1, 1)
    plt.plot(total_acc, label='Training Accuracy')
    plt.plot(total_val_acc, label='Validation Accuracy')
    plt.plot([initial_epochs-1, initial_epochs-1],
             plt.ylim(), label='Start Fine Tuning') # reshift plot around epochs
    plt.legend(loc='lower right')
    plt.title('Training and Validation Accuracy')

    plt.subplot(2, 1, 2)
    plt.plot(total_loss, label='Training Loss')
    plt.plot(total_val_loss, label='Validation Loss')
    plt.plot([initial_epochs-1, initial_epochs-1],
             plt.ylim(), label='Start Fine Tuning') # reshift plot around epochs
    plt.legend(loc='upper right')
    plt.title('Training and Validation Loss')
    plt.xlabel('epoch')
    plt.show()
```

**This is where saving the history variables of our model training comes in handy. Let's see what happened after fine-tuning the last 10 layers of our model.**

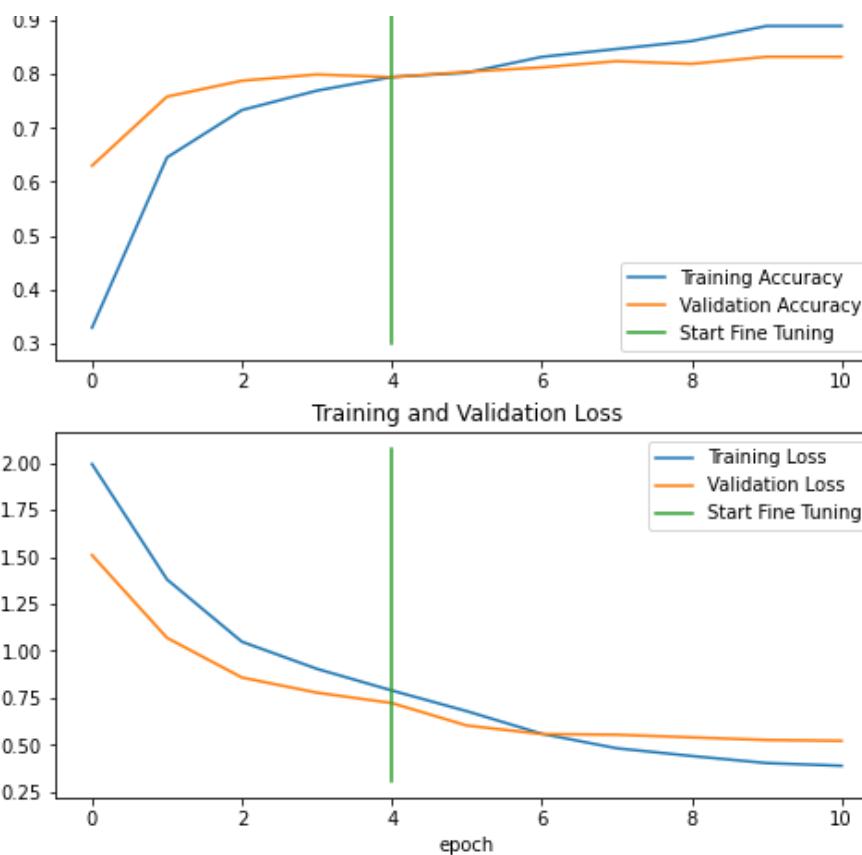
In [ ]:

```
compare_histories(original_history=history_10_percent_data_aug,
                  new_history=history_fine_10_percent_data_aug,
                  initial_epochs=5)
```

5

11

[0.329333351612091, 0.645333497047424, 0.733333492279053, 0.7693333625793457, 0.794666  
6479110718, 0.8026666641235352, 0.8320000171661377, 0.84666693687439, 0.8613333106040955  
, 0.8893333077430725, 0.8893333077430725]



Alright, alright, seems like the curves are heading in the right direction after fine-tuning. But remember, it should be noted that fine-tuning usually works best with larger amounts of data.

## Model 4: Fine-tuning an existing model all of the data

Enough talk about how fine-tuning a model usually works with more data, let's try it out.

We'll start by downloading the full version of our 10 food classes dataset.

In [ ]:

```
# Download and unzip 10 classes of data with all images
!wget https://storage.googleapis.com/ztm_tf_course/food_vision/10_food_classes_all_data.zip
unzip_data("10_food_classes_all_data.zip")

# Setup data directories
train_dir = "10_food_classes_all_data/train/"
test_dir = "10_food_classes_all_data/test/"

--2021-02-16 02:48:30-- https://storage.googleapis.com/ztm_tf_course/food_vision/10_food_classes_all_data.zip
Resolving storage.googleapis.com (storage.googleapis.com)... 172.217.164.144, 172.253.115.128, 172.253.63.128, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|172.217.164.144|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 519183241 (495M) [application/zip]
Saving to: '10_food_classes_all_data.zip'

10_food_classes_all 100%[=====] 495.13M 124MB/s in 4.1s

2021-02-16 02:48:35 (121 MB/s) - '10_food_classes_all_data.zip' saved [519183241/519183241]
```

In [ ]:

```
# How many images are we working with now?
walk_through_dir("10_food_classes_all_data")
```

```
There are 2 directories and 0 images in '10_food_classes_all_data'.
There are 10 directories and 0 images in '10_food_classes_all_data/train'.
There are 0 directories and 750 images in '10_food_classes_all_data/train/ice_cream'.
There are 0 directories and 750 images in '10_food_classes_all_data/train/ramen'.
There are 0 directories and 750 images in '10_food_classes_all_data/train/chicken_wings'.
There are 0 directories and 750 images in '10_food_classes_all_data/train/pizza'.
There are 0 directories and 750 images in '10_food_classes_all_data/train/steak'.
There are 0 directories and 750 images in '10_food_classes_all_data/train/fried_rice'.
There are 0 directories and 750 images in '10_food_classes_all_data/train/hamburger'.
There are 0 directories and 750 images in '10_food_classes_all_data/train/grilled_salmon'.
.
There are 0 directories and 750 images in '10_food_classes_all_data/train/sushi'.
There are 0 directories and 750 images in '10_food_classes_all_data/train/chicken_curry'.
There are 10 directories and 0 images in '10_food_classes_all_data/test'.
There are 0 directories and 250 images in '10_food_classes_all_data/test/ice_cream'.
There are 0 directories and 250 images in '10_food_classes_all_data/test/ramen'.
There are 0 directories and 250 images in '10_food_classes_all_data/test/chicken_wings'.
There are 0 directories and 250 images in '10_food_classes_all_data/test/pizza'.
There are 0 directories and 250 images in '10_food_classes_all_data/test/steak'.
There are 0 directories and 250 images in '10_food_classes_all_data/test/fried_rice'.
There are 0 directories and 250 images in '10_food_classes_all_data/test/hamburger'.
There are 0 directories and 250 images in '10_food_classes_all_data/test/grilled_salmon'.
There are 0 directories and 250 images in '10_food_classes_all_data/test/sushi'.
There are 0 directories and 250 images in '10_food_classes_all_data/test/chicken_curry'.
```

## And now we'll turn the images into tensors datasets.

In [ ]:

```
# Setup data inputs
import tensorflow as tf
IMG_SIZE = (224, 224)
train_data_10_classes_full = tf.keras.preprocessing.image_dataset_from_directory(train_dir,
                                                                           label_mode="categorical",
                                                                           image_size=IMG_SIZE)

# Note: this is the same test dataset we've been using for the previous experiments
test_data = tf.keras.preprocessing.image_dataset_from_directory(test_dir,
                                                               label_mode="categorical",
                                                               image_size=IMG_SIZE)
```

Found 7500 files belonging to 10 classes.  
Found 2500 files belonging to 10 classes.

Oh this is looking good. We've got 10x more images in of the training classes to work with.

The test dataset is the same we've been using for our previous experiments.

As it is now, our `model_2` has been fine-tuned on 10 percent of the data, so to begin fine-tuning on all of the data and keep our experiments consistent, we need to revert it back to the weights we checkpointed after 5 epochs of feature-extraction.

To demonstrate this, we'll first evaluate the current `model_2`.

In [ ]:

```
# Evaluate model (this is the fine-tuned 10 percent of data version)
model_2.evaluate(test_data)
```

79/79 [=====] - 10s 118ms/step - loss: 0.4870 - accuracy: 0.8388

Out[ ]:

[0.4869591295719147, 0.8388000130653381]

**These are the same values as** `results_fine_tune_10_percent`.

In [ ]:

```
results_fine_tune_10_percent
```

Out[ ]:

```
[0.48695918917655945, 0.8388000130653381]
```

**Now we'll revert the model back to the saved weights.**

In [ ]:

```
# Load model from checkpoint, that way we can fine-tune from the same stage the 10 percent data model was fine-tuned from
model_2.load_weights(checkpoint_path) # revert model back to saved weights
```

Out[ ]:

```
<tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x7fcd9dale4a8>
```

**And the results should be the same as** `results_10_percent_data_aug`.

In [ ]:

```
# After loading the weights, this should have gone down (no fine-tuning)
model_2.evaluate(test_data)
```

```
79/79 [=====] - 10s 117ms/step - loss: 0.7047 - accuracy: 0.8080
```

Out[ ]:

```
[0.7046542763710022, 0.8080000281333923]
```

In [ ]:

```
# Check to see if the above two results are the same (they should be)
results_10_percent_data_aug
```

Out[ ]:

```
[0.7046541571617126, 0.8080000281333923]
```

**Alright, the previous steps might seem quite confusing but all we've done is:**

1. Trained a feature extraction transfer learning model for 5 epochs on 10% of the data (with all base model layers frozen) and saved the model's weights using `ModelCheckpoint`.
2. Fine-tuned the same model on the same 10% of the data for a further 5 epochs with the top 10 layers of the base model unfrozen.
3. Saved the results and training logs each time.
4. Reloaded the model from 1 to do the same steps as 2 but with all of the data.

**The same steps as 2?**

**Yeah, we're going to fine-tune the last 10 layers of the base model with the full dataset for another 5 epochs but first let's remind ourselves which layers are trainable.**

In [ ]:

```
# Check which layers are tuneable in the whole model
for layer_number, layer in enumerate(model_2.layers):
    print(layer_number, layer.name, layer.trainable)
```

```
0 input_layer True
1 data_augmentation True
2 efficientnetb0 True
3 global_average_pooling_layer True
4 output_layer True
```

## Can we get a little more specific?

In [ ]:

```
# Check which layers are tuneable in the base model
for layer_number, layer in enumerate(base_model.layers):
    print(layer_number, layer.name, layer.trainable)

0 input_3 False
1 rescaling_2 False
2 normalization_2 False
3 stem_conv_pad False
4 stem_conv False
5 stem_bn False
6 stem_activation False
7 block1a_dwconv False
8 block1a_bn False
9 block1a_activation False
10 block1a_se_squeeze False
11 block1a_se_reshape False
12 block1a_se_reduce False
13 block1a_se_expand False
14 block1a_se_excite False
15 block1a_project_conv False
16 block1a_project_bn False
17 block2a_expand_conv False
18 block2a_expand_bn False
19 block2a_expand_activation False
20 block2a_dwconv_pad False
21 block2a_dwconv False
22 block2a_bn False
23 block2a_activation False
24 block2a_se_squeeze False
25 block2a_se_reshape False
26 block2a_se_reduce False
27 block2a_se_expand False
28 block2a_se_excite False
29 block2a_project_conv False
30 block2a_project_bn False
31 block2b_expand_conv False
32 block2b_expand_bn False
33 block2b_expand_activation False
34 block2b_dwconv False
35 block2b_bn False
36 block2b_activation False
37 block2b_se_squeeze False
38 block2b_se_reshape False
39 block2b_se_reduce False
40 block2b_se_expand False
41 block2b_se_excite False
42 block2b_project_conv False
43 block2b_project_bn False
44 block2b_drop False
45 block2b_add False
46 block3a_expand_conv False
47 block3a_expand_bn False
48 block3a_expand_activation False
49 block3a_dwconv_pad False
50 block3a_dwconv False
51 block3a_bn False
52 block3a_activation False
53 block3a_se_squeeze False
54 block3a_se_reshape False
55 block3a_se_reduce False
56 block3a_se_expand False
57 block3a_se_excite False
58 block3a_project_conv False
59 block3a_project_bn False
60 block3b_expand_conv False
61 block3b_expand_bn False
62 block3b_expand_activation False
```

```
63 block3b_dwconv False
64 block3b_bn False
65 block3b_activation False
66 block3b_se_squeeze False
67 block3b_se_reshape False
68 block3b_se_reduce False
69 block3b_se_expand False
70 block3b_se_excite False
71 block3b_project_conv False
72 block3b_project_bn False
73 block3b_drop False
74 block3b_add False
75 block4a_expand_conv False
76 block4a_expand_bn False
77 block4a_expand_activation False
78 block4a_dwconv_pad False
79 block4a_dwconv False
80 block4a_bn False
81 block4a_activation False
82 block4a_se_squeeze False
83 block4a_se_reshape False
84 block4a_se_reduce False
85 block4a_se_expand False
86 block4a_se_excite False
87 block4a_project_conv False
88 block4a_project_bn False
89 block4b_expand_conv False
90 block4b_expand_bn False
91 block4b_expand_activation False
92 block4b_dwconv False
93 block4b_bn False
94 block4b_activation False
95 block4b_se_squeeze False
96 block4b_se_reshape False
97 block4b_se_reduce False
98 block4b_se_expand False
99 block4b_se_excite False
100 block4b_project_conv False
101 block4b_project_bn False
102 block4b_drop False
103 block4b_add False
104 block4c_expand_conv False
105 block4c_expand_bn False
106 block4c_expand_activation False
107 block4c_dwconv False
108 block4c_bn False
109 block4c_activation False
110 block4c_se_squeeze False
111 block4c_se_reshape False
112 block4c_se_reduce False
113 block4c_se_expand False
114 block4c_se_excite False
115 block4c_project_conv False
116 block4c_project_bn False
117 block4c_drop False
118 block4c_add False
119 block5a_expand_conv False
120 block5a_expand_bn False
121 block5a_expand_activation False
122 block5a_dwconv False
123 block5a_bn False
124 block5a_activation False
125 block5a_se_squeeze False
126 block5a_se_reshape False
127 block5a_se_reduce False
128 block5a_se_expand False
129 block5a_se_excite False
130 block5a_project_conv False
131 block5a_project_bn False
132 block5b_expand_conv False
133 block5b_expand_bn False
134 block5b_expand_activation False
```

```
135 block5b_dwconv False
136 block5b_bn False
137 block5b_activation False
138 block5b_se_squeeze False
139 block5b_se_reshape False
140 block5b_se_reduce False
141 block5b_se_expand False
142 block5b_se_excite False
143 block5b_project_conv False
144 block5b_project_bn False
145 block5b_drop False
146 block5b_add False
147 block5c_expand_conv False
148 block5c_expand_bn False
149 block5c_expand_activation False
150 block5c_dwconv False
151 block5c_bn False
152 block5c_activation False
153 block5c_se_squeeze False
154 block5c_se_reshape False
155 block5c_se_reduce False
156 block5c_se_expand False
157 block5c_se_excite False
158 block5c_project_conv False
159 block5c_project_bn False
160 block5c_drop False
161 block5c_add False
162 block6a_expand_conv False
163 block6a_expand_bn False
164 block6a_expand_activation False
165 block6a_dwconv_pad False
166 block6a_dwconv False
167 block6a_bn False
168 block6a_activation False
169 block6a_se_squeeze False
170 block6a_se_reshape False
171 block6a_se_reduce False
172 block6a_se_expand False
173 block6a_se_excite False
174 block6a_project_conv False
175 block6a_project_bn False
176 block6b_expand_conv False
177 block6b_expand_bn False
178 block6b_expand_activation False
179 block6b_dwconv False
180 block6b_bn False
181 block6b_activation False
182 block6b_se_squeeze False
183 block6b_se_reshape False
184 block6b_se_reduce False
185 block6b_se_expand False
186 block6b_se_excite False
187 block6b_project_conv False
188 block6b_project_bn False
189 block6b_drop False
190 block6b_add False
191 block6c_expand_conv False
192 block6c_expand_bn False
193 block6c_expand_activation False
194 block6c_dwconv False
195 block6c_bn False
196 block6c_activation False
197 block6c_se_squeeze False
198 block6c_se_reshape False
199 block6c_se_reduce False
200 block6c_se_expand False
201 block6c_se_excite False
202 block6c_project_conv False
203 block6c_project_bn False
204 block6c_drop False
205 block6c_add False
206 block6d_expand_conv False
```

```
207 block6d_expand_bn False
208 block6d_expand_activation False
209 block6d_dwconv False
210 block6d_bn False
211 block6d_activation False
212 block6d_se_squeeze False
213 block6d_se_reshape False
214 block6d_se_reduce False
215 block6d_se_expand False
216 block6d_se_excite False
217 block6d_project_conv False
218 block6d_project_bn False
219 block6d_drop False
220 block6d_add False
221 block7a_expand_conv False
222 block7a_expand_bn False
223 block7a_expand_activation False
224 block7a_dwconv False
225 block7a_bn False
226 block7a_activation False
227 block7a_se_squeeze True
228 block7a_se_reshape True
229 block7a_se_reduce True
230 block7a_se_expand True
231 block7a_se_excite True
232 block7a_project_conv True
233 block7a_project_bn True
234 top_conv True
235 top_bn True
236 top_activation True
```

**Looking good! The last 10 layers are trainable (unfrozen).**

**We've got one more step to do before we can begin fine-tuning.**

**Do you remember what it is?**

**I'll give you a hint. We just reloaded the weights to our model and what do we need to do every time we make a change to our models?**

**Recompile them!**

**This will be just as before.**

In [ ]:

```
# Compile
model_2.compile(loss="categorical_crossentropy",
                  optimizer=tf.keras.optimizers.Adam(lr=0.0001), # divide learning rate by
10 for fine-tuning
                  metrics=[ "accuracy" ])
```

**Alright, time to fine-tune on all of the data!**

In [ ]:

```
# Continue to train and fine-tune the model to our data
fine_tune_epochs = initial_epochs + 5

history_fine_10_classes_full = model_2.fit(train_data_10_classes_full,
                                             epochs=fine_tune_epochs,
                                             initial_epoch=history_10_percent_data_aug.epoch[-1],
                                             validation_data=test_data,
                                             validation_steps=int(0.25 * len(test_data)),
                                             callbacks=[create_tensorboard_callback("transfer_learning", "full_10_classes_fine_tune_last_10"))]
```

Saving TensorBoard log files to: transfer\_learning/full\_10\_classes\_fine\_tune\_last\_10/20210216-025031

```

Epoch 5/10
235/235 [=====] - 49s 190ms/step - loss: 0.7943 - accuracy: 0.75
13 - val_loss: 0.4029 - val_accuracy: 0.8635
Epoch 6/10
235/235 [=====] - 51s 215ms/step - loss: 0.6391 - accuracy: 0.79
13 - val_loss: 0.3668 - val_accuracy: 0.8882
Epoch 7/10
235/235 [=====] - 47s 198ms/step - loss: 0.5564 - accuracy: 0.82
20 - val_loss: 0.3237 - val_accuracy: 0.9013
Epoch 8/10
235/235 [=====] - 46s 192ms/step - loss: 0.5030 - accuracy: 0.83
94 - val_loss: 0.3390 - val_accuracy: 0.8832
Epoch 9/10
235/235 [=====] - 45s 191ms/step - loss: 0.4611 - accuracy: 0.84
79 - val_loss: 0.3099 - val_accuracy: 0.8980
Epoch 10/10
235/235 [=====] - 43s 183ms/step - loss: 0.4507 - accuracy: 0.85
57 - val_loss: 0.2903 - val_accuracy: 0.9161

```

**Note:** Training took longer per epoch, but that makes sense because we're using 10x more training data than before.

Let's evaluate on all of the test data.

In [ ]:

```
results_fine_tune_full_data = model_2.evaluate(test_data)
results_fine_tune_full_data
```

```
79/79 [=====] - 10s 118ms/step - loss: 0.3264 - accuracy: 0.8988
```

Out[ ]:

```
[0.32638612389564514, 0.8988000154495239]
```

Nice! It looks like fine-tuning with all of the data has given our model a boost, how do the training curves look?

In [ ]:

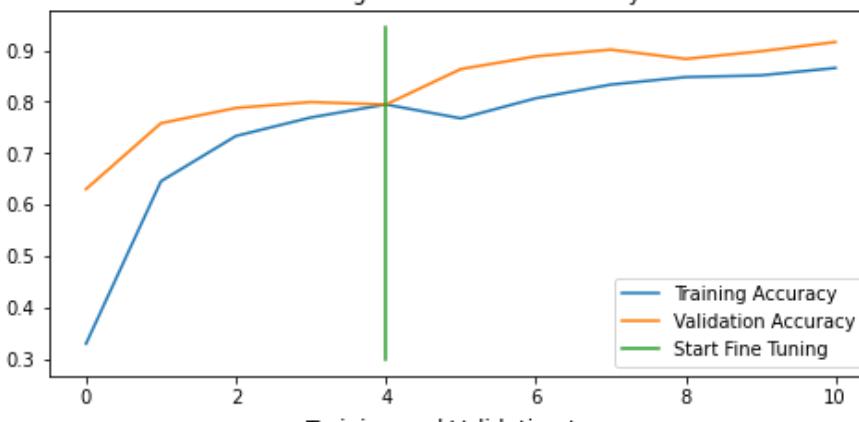
```
# How did fine-tuning go with more data?
compare_histories(original_history=history_10_percent_data_aug,
                  new_history=history_fine_10_classes_full,
                  initial_epochs=5)
```

```
5
```

```
11
```

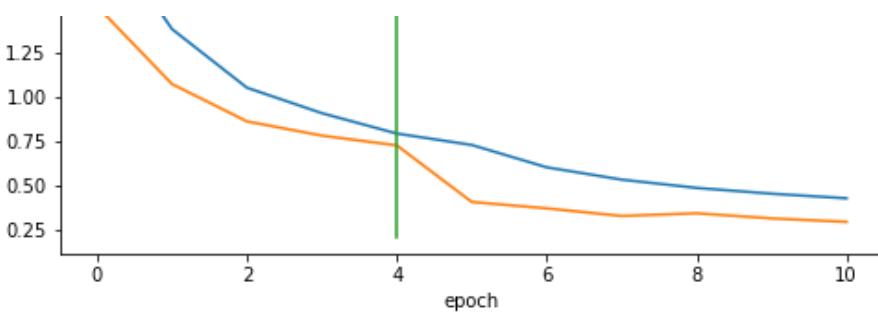
```
[0.329333351612091, 0.6453333497047424, 0.7333333492279053, 0.7693333625793457, 0.794666
6479110718, 0.767599999046326, 0.806666722297668, 0.833333134651184, 0.847866654396057
1, 0.8511999845504761, 0.8655999898910522]
```

Training and Validation Accuracy



Training and Validation Loss





Looks like that extra data helped! Those curves are looking great. And if we trained for longer, they might even keep improving.

## Viewing our experiment data on TensorBoard

Right now our experimental results are scattered all throughout our notebook. If we want to share them with someone, they'd be getting a bunch of different graphs and metrics... not a fun time.

But guess what?

Thanks to the TensorBoard callback we made with our helper function `create_tensorflow_callback()`, we've been tracking our modelling experiments the whole time.

How about we upload them to TensorBoard.dev and check them out?

We can do with the `tensorboard dev upload` command and passing it the directory where our experiments have been logged.

**Note:** Remember, whatever you upload to TensorBoard.dev becomes public. If there are training logs you don't want to share, don't upload them.

In [ ]:

```
# View tensorboard logs of transfer learning modelling experiments (should be 4 models)
# Upload TensorBoard dev records
!tensorboard dev upload --logdir ./transfer_learning \
--name "Transfer learning experiments" \
--description "A series of different transfer learning experiments with varying amounts \
of data and fine-tuning" \
--one_shot # exits the uploader when upload has finished
```

2020-09-17 22:51:36.043126: I tensorflow/stream\_executor/platform/default/dso\_loader.cc:48] Successfully opened dynamic library libcudart.so.10.1

Data for the "graphs" plugin is now uploaded to TensorBoard.dev! Note that uploaded data is public. If you do not want to upload data for this plugin, use the "--plugins" command line argument.

Data for the "histograms" plugin is now uploaded to TensorBoard.dev! Note that uploaded data is public. If you do not want to upload data for this plugin, use the "--plugins" command line argument.

Data for the "hparams" plugin is now uploaded to TensorBoard.dev! Note that uploaded data is public. If you do not want to upload data for this plugin, use the "--plugins" command line argument.

Upload started and will continue reading any new data as it's added to the logdir. To stop uploading, press Ctrl-C.

View your TensorBoard live at: <https://tensorboard.dev/experiment/2076kw3PQbKl01Byfg5B4w/>

[2020-09-17T22:51:37] Uploader started.

[2020-09-17T22:51:47] Total uploaded: 128 scalars, 0 tensors, 5 binary objects (9.1 MB)

Listening for new data in logdir...

Done. View your TensorBoard at <https://tensorboard.dev/experiment/2076kw3PQbKl01Byfg5B4w/>

Once we've uploaded the results to TensorBoard.dev we get a shareable link we can use to view and compare our experiments and share our results with others if needed.

You can view the original versions of the experiments we ran in this notebook here:

<https://tensorboard.dev/experiment/2076kw3PQbKl0lByfg5B4w/>

Question: Which model performed the best? Why do you think this is? How did fine-tuning go?

To find all of your previous TensorBoard.dev experiments using the command `tensorboard dev list`.

In [ ]:

```
# View previous experiments
!tensorboard dev list
```

```
2020-09-17 22:51:48.747476: I tensorflow/stream_executor/platform/default/dso_loader.cc:4
8] Successfully opened dynamic library libcudart.so.10.1
Data for the "graphs" plugin is now uploaded to TensorBoard.dev! Note that uploaded data
is public. If you do not want to upload data for this plugin, use the "--plugins" command
line argument.
Data for the "histograms" plugin is now uploaded to TensorBoard.dev! Note that uploaded d
ata is public. If you do not want to upload data for this plugin, use the "--plugins" comm
and line argument.
Data for the "hparams" plugin is now uploaded to TensorBoard.dev! Note that uploaded data
is public. If you do not want to upload data for this plugin, use the "--plugins" command
line argument.
https://tensorboard.dev/experiment/2076kw3PQbKl0lByfg5B4w/
Name           Transfer learning experiments
Description     A series of different transfer learning experiments with varying am
ounts of data and fine-tuning
Id             2076kw3PQbKl0lByfg5B4w
Created        2020-09-17 22:51:37 (15 seconds ago)
Updated        2020-09-17 22:51:47 (5 seconds ago)
Runs           10
Tags           3
Scalars        128
Tensor bytes   0
Binary object bytes 9520961
https://tensorboard.dev/experiment/73taSKxXQeGPQsNBcVvY3g/
Name           EfficientNetB0 vs. ResNet50V2
Description     Comparing two different TF Hub feature extraction models architectu
res using 10% of training images
Id             73taSKxXQeGPQsNBcVvY3g
Created        2020-09-14 05:02:48
Updated        2020-09-14 05:02:50
Runs           4
Tags           3
Scalars        40
Tensor bytes   0
Binary object bytes 3402042
Total: 2 experiment(s)
```

And if you want to remove a previous experiment (and delete it from public viewing) you can use the command:

```
tensorboard dev delete --experiment_id [INSERT_EXPERIMENT_ID_TO_DELETE]
```

In [ ]:

```
# Remove previous experiments
# !tensorboard dev delete --experiment_id OUbW0O3pRqqQgAphVBxi8Q
```

```
2020-09-17 22:51:53.982454: I tensorflow/stream_executor/platform/default/dso_loader.cc:4
8] Successfully opened dynamic library libcudart.so.10.1
Data for the "graphs" plugin is now uploaded to TensorBoard.dev! Note that uploaded data
is public. If you do not want to upload data for this plugin, use the "--plugins" command
line argument.
Data for the "histograms" plugin is now uploaded to TensorBoard.dev! Note that uploaded d
ata is public. If you do not want to upload data for this plugin, use the "--plugins" comm
and line argument.
Data for the "hparams" plugin is now uploaded to TensorBoard.dev! Note that uploaded data
is public. If you do not want to upload data for this plugin, use the "--plugins" command
line argument.
```

## Exercises

1. Write a function to visualize an image from any dataset (train or test file) and any class (e.g. "steak", "pizza"... etc), visualize it and make a prediction on it using a trained model.
2. Use feature-extraction to train a transfer learning model on 10% of the Food Vision data for 10 epochs using `tf.keras.applications.EfficientNetB0` as the base model. Use the `ModelCheckpoint` callback to save the weights to file.
3. Fine-tune the last 20 layers of the base model you trained in 2 for another 10 epochs. How did it go?
4. Fine-tune the last 30 layers of the base model you trained in 2 for another 10 epochs. How did it go?

## Extra-curriculum

- Read the [documentation on data augmentation](#) in TensorFlow.
- Read the [ULMFit paper](#) (technical) for an introduction to the concept of freezing and unfreezing different layers.
- Read up on learning rate scheduling (there's a [TensorFlow callback](#) for this), how could this influence our model training?
  - If you're training for longer, you probably want to reduce the learning rate as you go... the closer you get to the bottom of the hill, the smaller steps you want to take. Imagine it like finding a coin at the bottom of your couch. In the beginning your arm movements are going to be large and the closer you get, the smaller your movements become.

## 06. Transfer Learning with TensorFlow Part 3: Scaling up (Food Vision mini)

In the previous two notebooks ([transfer learning part 1: feature extraction](#) and [part 2: fine-tuning](#)) we've seen the power of transfer learning.

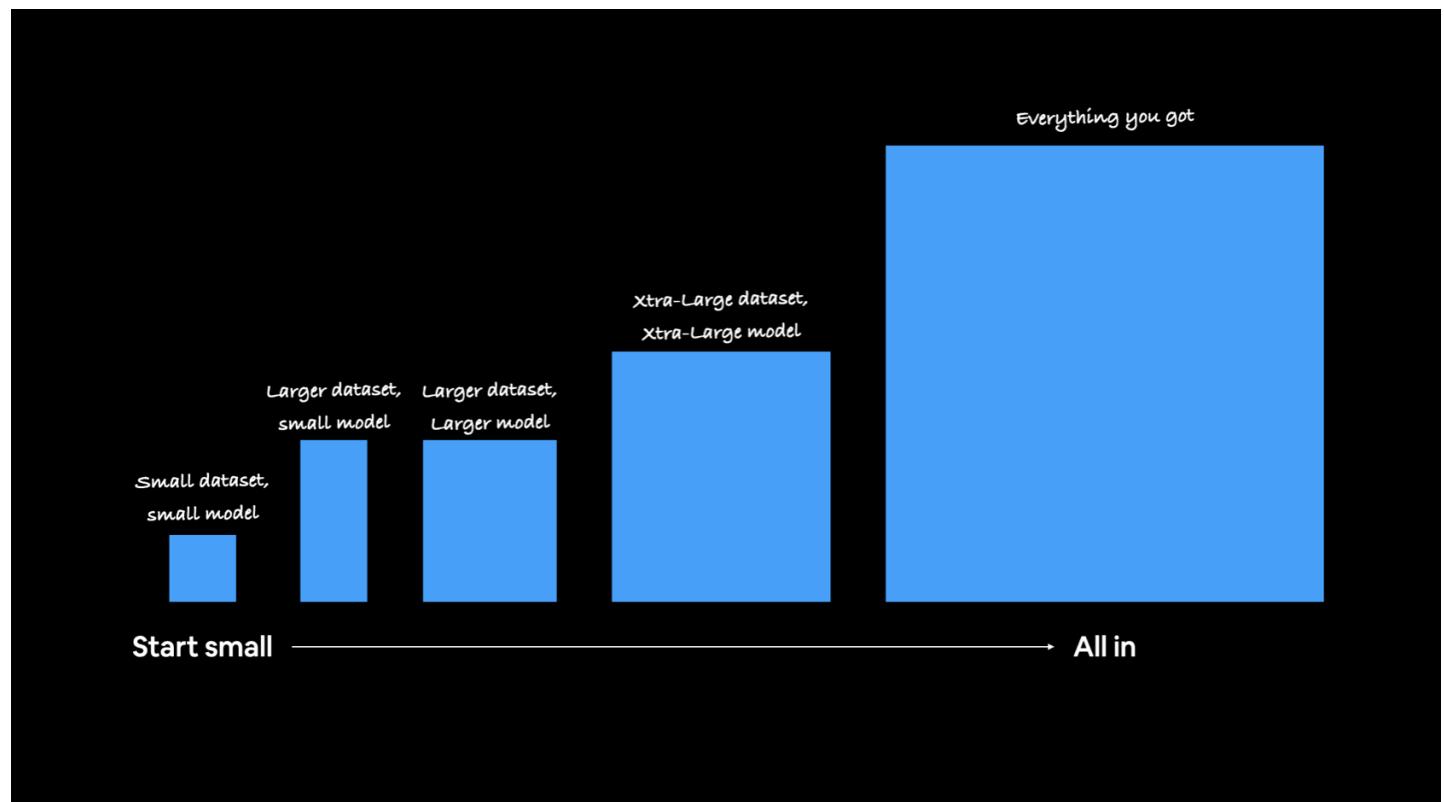
Now we know our smaller modelling experiments are working, it's time to step things up a notch with more data.

This is a common practice in machine learning and deep learning: get a model working on a small amount of data before scaling it up to a larger amount of data.

 **Note:** You haven't forgotten the machine learning practitioners motto have you? "Experiment, experiment, experiment."

It's time to get closer to our Food Vision project coming to life. In this notebook we're going to scale up from using 10 classes of the Food101 data to using all of the classes in the Food101 dataset.

Our goal is to beat the original [Food101 paper](#)'s results with 10% of data.



*Machine learning practitioners are serial experimenters. Start small, get a model working, see if your experiments work then gradually scale them up to where you want to go (we're going to be looking at scaling up throughout this notebook).*

### What we're going to cover

We're going to go through the follow with TensorFlow:

- Downloading and preparing 10% of the Food101 data (10% of training data)
- Training a feature extraction transfer learning model on 10% of the Food101 training data
- Fine-tuning our feature extraction model
- Saving and loaded our trained model
- Evaluating the performance of our Food Vision model trained on 10% of the training data
  - Finding our model's most wrong predictions
- Making predictions with our Food Vision model on custom images of food

## How you can use this notebook

You can read through the descriptions and the code (it should all run, except for the cells which error on purpose), but there's a better option.

Write all of the code yourself.

Yes. I'm serious. Create a new notebook, and rewrite each line by yourself. Investigate it, see if you can break it, why does it break?

You don't have to write the text descriptions but writing the code yourself is a great way to get hands-on experience.

Don't worry if you make mistakes, we all do. The way to get better and make less mistakes is to **write more code**.

Resource: See the full set of course materials on GitHub:

<https://github.com/mrdbourke/tensorflow-deep-learning>

```
In [ ]:

# Are we using a GPU?
!nvidia-smi

Thu Feb 25 03:38:16 2021
+-----+
| NVIDIA-SMI 460.39      Driver Version: 460.32.03    CUDA Version: 11.2 |
|-----+-----+-----+
| GPU  Name      Persistence-M| Bus-Id      Disp.A  | Volatile Uncorr. ECC | | | |
| Fan  Temp     Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |
| |          |          |              |           | MIG M.   |
|-----+-----+-----+-----+
|  0  Tesla P100-PCIE... Off  | 00000000:00:04.0 Off |          0 | | | |
| N/A   34C     P0    26W / 250W |          0MiB / 16280MiB |      0%  Default |
| |          |          |              |           | N/A      |
+-----+-----+-----+
+-----+
| Processes:                               |
| GPU  GI  CI      PID  Type      Process name          GPU Memory |
|       ID  ID                  |                      Usage  |
|-----+-----+-----+-----+
| No running processes found               |
+-----+
```

## Creating helper functions

We've created a series of helper functions throughout the previous notebooks. Instead of rewriting them (tedious), we'll import the `helper_functions.py` file from the GitHub repo.

In [ ]:

```
# Get helper functions file
!wget https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/extras/helper_functions.py

--2021-02-25 03:38:17-- https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/extras/helper_functions.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.1
99.110.133, 185.199.111.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443.
.. connected.
HTTP request sent, awaiting response... 200 OK
Length: 9304 (9.1K) [text/plain]
```

```
Saving to: 'helper_functions.py'
```

```
helper_functions.py 100%[=====] 9.09K --.-KB/s in 0s
```

```
2021-02-25 03:38:17 (117 MB/s) - 'helper_functions.py' saved [9304/9304]
```

```
In [ ]:
```

```
# Import series of helper functions for the notebook (we've created/used these in previous notebooks)
from helper_functions import create_tensorboard_callback, plot_loss_curves, unzip_data, compare_histories, walk_through_dir
```

## 101 Food Classes: Working with less data

So far we've confirmed the transfer learning model's we've been using work pretty well with the 10 Food Classes dataset. Now it's time to step it up and see how they go with the full 101 Food Classes.

In the original [Food101](#) dataset there's 1000 images per class (750 of each class in the training set and 250 of each class in the test set), totalling 101,000 images.

We could start modelling straight away on this large dataset but in the spirit of continually experimenting, we're going to see how our previously working model's go with 10% of the training data.

This means for each of the 101 food classes we'll be building a model on 75 training images and evaluating it on 250 test images.

## Downloading and preprocessing the data

Just as before we'll download a subset of the Food101 dataset which has been extracted from the original dataset (to see the preprocessing of the data check out the [Food Vision preprocessing notebook](#)).

We download the data as a zip file so we'll use our `unzip_data()` function to unzip it.

```
In [ ]:
```

```
# Download data from Google Storage (already preformatted)
!wget https://storage.googleapis.com/ztm_tf_course/food_vision/101_food_classes_10_percent.zip

unzip_data("101_food_classes_10_percent.zip")

train_dir = "101_food_classes_10_percent/train/"
test_dir = "101_food_classes_10_percent/test/"

--2021-02-25 03:38:24-- https://storage.googleapis.com/ztm_tf_course/food_vision/101_food_classes_10_percent.zip
Resolving storage.googleapis.com (storage.googleapis.com)... 64.233.189.128, 108.177.97.1
28, 108.177.125.128, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|64.233.189.128|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1625420029 (1.5G) [application/zip]
Saving to: '101_food_classes_10_percent.zip'

101_food_classes_10 100%[=====] 1.51G 92.3MB/s in 17s

2021-02-25 03:38:42 (93.2 MB/s) - '101_food_classes_10_percent.zip' saved [1625420029/1625420029]
```

```
In [ ]:
```

```
# How many images/classes are there?
walk_through_dir("101_food_classes_10_percent")
```

There are 2 directories and 0 images in '101\_food\_classes\_10\_percent'.

There are 101 directories and 0 images in '101\_food\_classes\_10\_percent/train'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/carrot\_cake'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/gyoza'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/sushi'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/deviled\_eggs'.

.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/falafel'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/croque\_madame'.

.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/greek\_salad'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/churros'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/frozen\_yogurt'.

.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/shrimp\_and\_grits'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/pizza'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/edamame'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/samosa'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/clam\_chowder'.

.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/red\_velvet\_cake'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/panna\_cotta'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/paella'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/seaweed\_salad'.

.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/pork\_chop'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/mussels'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/chicken\_wings'.

.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/tacos'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/chocolate\_mousse'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/miso\_soup'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/beef\_carpaccio'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/lasagna'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/chicken\_curry'.

.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/cannoli'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/cheesecake'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/lobster\_roll\_sandwich'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/lobster\_bisque'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/sashimi'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/tiramisu'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/strawberry\_shortcake'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/foie\_gras'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/beignets'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/onion\_rings'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/apple\_pie'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/bibimbap'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/french\_onion\_soup'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/french\_fries'.

.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/eggs\_benedict'.

.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/fish\_and\_chips'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/tuna\_tartare'.

.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/donuts'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/spaghetti\_bolognese'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/caprese\_salad'.

.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/creme\_brulee'.

.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/peking\_duck'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/takoyaki'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/waffles'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/scallops'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/grilled\_salmon'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/oysters'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/pad\_thai'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/omelette'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/club\_sandwich'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/gnocchi'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/spring\_rolls'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/ramen'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/grilled\_cheese\_sandwich'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/guacamole'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/prime\_rib'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/filet\_mignon'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/breakfast\_burrito'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/macaroni\_and\_cheese'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/dumplings'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/chicken\_quesadilla'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/macarons'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/spaghetti\_carbonara'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/huevos\_rancheros'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/fried\_calamari'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/garlic\_bread'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/escargots'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/risotto'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/cheese\_plate'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/ceviche'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/cup\_cakes'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/poutine'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/baby\_back\_ribs'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/nachos'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/pho'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/crab\_cakes'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/bruschetta'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/bread\_pudding'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/ravioli'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/ice\_cream'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/steak'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/hot\_and\_sour\_soup'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/beet\_salad'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/french\_toast'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/hamburger'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/pancakes'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/fried\_rice'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/hummus'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/hot\_dog'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/chocolate\_cake'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/pulled\_pork\_sandwich'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/caesar\_salad'.  
There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/beef\_tartare'.

There are 0 directories and 75 images in '101\_food\_classes\_10\_percent/train/baklava'.  
There are 101 directories and 0 images in '101\_food\_classes\_10\_percent/test'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/carrot\_cake'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/gyoza'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/sushi'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/deviled\_eggs'.  
. . .  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/falafel'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/croque\_madame'.  
. . .  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/greek\_salad'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/churros'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/frozen\_yogurt'.  
. . .  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/shrimp\_and\_grits'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/pizza'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/edamame'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/samosa'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/clam\_chowder'.  
. . .  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/red\_velvet\_cake'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/panna\_cotta'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/paella'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/seaweed\_salad'.  
. . .  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/pork\_chop'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/mussels'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/chicken\_wings'.  
. . .  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/tacos'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/chocolate\_mousse'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/miso\_soup'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/beef\_carpaccio'.  
. . .  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/lasagna'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/chicken\_curry'.  
. . .  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/cannoli'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/cheesecake'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/lobster\_roll\_sandwich'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/lobster\_bisque'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/sashimi'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/tiramisu'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/strawberry\_shortcake'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/foie\_gras'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/beignets'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/onion\_rings'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/apple\_pie'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/bibimbap'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/french\_onion\_soup'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/french\_fries'.  
. . .  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/eggs\_benedict'.  
. . .  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/fish\_and\_chips'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/tuna\_tartare'.  
. . .  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/donuts'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/spaghetti\_bolognese'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/caprese\_salad'.  
. . .  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/creme\_brulee'.

There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/peking\_duck'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/takoyaki'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/waffles'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/scallops'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/grilled\_salmon'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/oysters'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/pad\_thai'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/omelette'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/club\_sandwich'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/gnocchi'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/spring\_rolls'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/ramen'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/grilled\_cheese\_sandwich'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/guacamole'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/prime\_rib'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/filet\_mignon'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/breakfast\_burrito'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/macaroni\_and\_cheese'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/dumplings'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/chicken\_quesadilla'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/macarons'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/spaghetti\_carbonara'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/huevos\_rancheros'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/fried\_calamari'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/garlic\_bread'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/escargots'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/risotto'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/cheese\_plate'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/ceviche'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/cup\_cakes'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/poutine'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/baby\_back\_ribs'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/nachos'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/pho'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/crab\_cakes'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/bruschetta'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/bread\_pudding'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/ravioli'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/ice\_cream'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/steak'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/hot\_and\_sour\_soup'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/beet\_salad'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/french\_toast'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/hamburger'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/pancakes'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/fried\_rice'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/hummus'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/hot\_dog'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/chocolate\_cake'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/pulled\_pork\_sandwich'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/caesar\_salad'.

There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/beer\_tartare'.  
There are 0 directories and 250 images in '101\_food\_classes\_10\_percent/test/baklava'.

### As before our data comes in the common image classification data format of:

Example of file structure

```
10_food_classes_10_percent <- top level folder
  └── train <- training images
    ├── pizza
    │   ├── 1008104.jpg
    │   ├── 1638227.jpg
    │   ├── ...
    └── steak
        ├── 1000205.jpg
        ├── 1647351.jpg
        ├── ...
  └── test <- testing images
    ├── pizza
    │   ├── 1001116.jpg
    │   ├── 1507019.jpg
    │   ├── ...
    └── steak
        ├── 100274.jpg
        ├── 1653815.jpg
        ├── ...
```

Let's use the `image_dataset_from_directory()` function to turn our images and labels into a `tf.data.Dataset`, a TensorFlow datatype which allows for us to pass it directory to our model.

For the test dataset, we're going to set `shuffle=False` so we can perform repeatable evaluation and visualization on it later.

In [ ]:

```
# Setup data inputs
import tensorflow as tf
IMG_SIZE = (224, 224)
train_data_all_10_percent = tf.keras.preprocessing.image_dataset_from_directory(train_dir,
  label_mode="categorical",
  image_size=IMG_SIZE)

test_data = tf.keras.preprocessing.image_dataset_from_directory(test_dir,
  label_mode="categorical",
  image_size=IMG_SIZE,
  shuffle=False) # don't
shuffle test data for prediction analysis
```

Found 7575 files belonging to 101 classes.  
Found 25250 files belonging to 101 classes.

Wonderful! It looks like our data has been imported as expected with 75 images per class in the training set (75 images **101 classes = 7575 images**) and 25250 images in the test set (250 images 101 classes = 25250 images).

## Train a big dog model with transfer learning on 10% of 101 food classes

Our food image data has been imported into TensorFlow, time to model it.

To keep our experiments swift, we're going to start by using feature extraction transfer learning with a pre-trained model for a few epochs and then fine-tune for a few more epochs.

More specifically, our goal will be to see if we can beat the baseline from original [Food101 paper](#) (50.76% accuracy on 101 classes) with 10% of the training data and the following modelling setup:

- A `ModelCheckpoint` callback to save our progress during training, this means we could experiment with further training later without having to train from scratch every time
- Data augmentation built right into the model
- A headless (no top layers) `EfficientNetB0` architecture from `tf.keras.applications` as our base model
- A `Dense` layer with 101 hidden neurons (same as number of food classes) and softmax activation as the output layer
- Categorical crossentropy as the loss function since we're dealing with more than two classes
- The Adam optimizer with the default settings
- Fitting for 5 full passes on the training data while evaluating on 15% of the test data

It seems like a lot but these are all things we've covered before in the [Transfer Learning in TensorFlow Part 2: Fine-tuning notebook](#).

Let's start by creating the `ModelCheckpoint` callback.

Since we want our model to perform well on unseen data we'll set it to monitor the validation accuracy metric and save the model weights which score the best on that.

In [ ]:

```
# Create checkpoint callback to save model for later use
checkpoint_path = "101_classes_10_percent_data_model_checkpoint"
checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(checkpoint_path,
                                                       save_weights_only=True, # save
                                                       only the model weights
                                                       monitor="val_accuracy", # save
                                                       the model weights which score the best validation accuracy
                                                       save_best_only=True) # only keep
                                                       the best model weights on file (delete the rest)
```

Checkpoint ready. Now let's create a small data augmentation model with the Sequential API. Because we're working with a reduced sized training set, this will help prevent our model from overfitting on the training data.

In [ ]:

```
# Import the required modules for model creation
from tensorflow.keras import layers
from tensorflow.keras.layers.experimental import preprocessing
from tensorflow.keras.models import Sequential

# Setup data augmentation
data_augmentation = Sequential([
    preprocessing.RandomFlip("horizontal"), # randomly flip images on horizontal edge
    preprocessing.RandomRotation(0.2), # randomly rotate images by a specific amount
    preprocessing.RandomHeight(0.2), # randomly adjust the height of an image by a specific amount
    preprocessing.RandomWidth(0.2), # randomly adjust the width of an image by a specific amount
    preprocessing.RandomZoom(0.2), # randomly zoom into an image
    # preprocessing.Rescaling(1./255) # keep for models like ResNet50V2, remove for EfficientNet
], name="data_augmentation")
```

Beautiful! We'll be able to insert the `data_augmentation` Sequential model as a layer in our Functional API model. That way if we want to continue training our model at a later time, the data augmentation is already built right in.

Speaking of Functional API model's, time to put together a feature extraction transfer learning model using `tf.keras.applications.EfficientNetB0` as our base model.

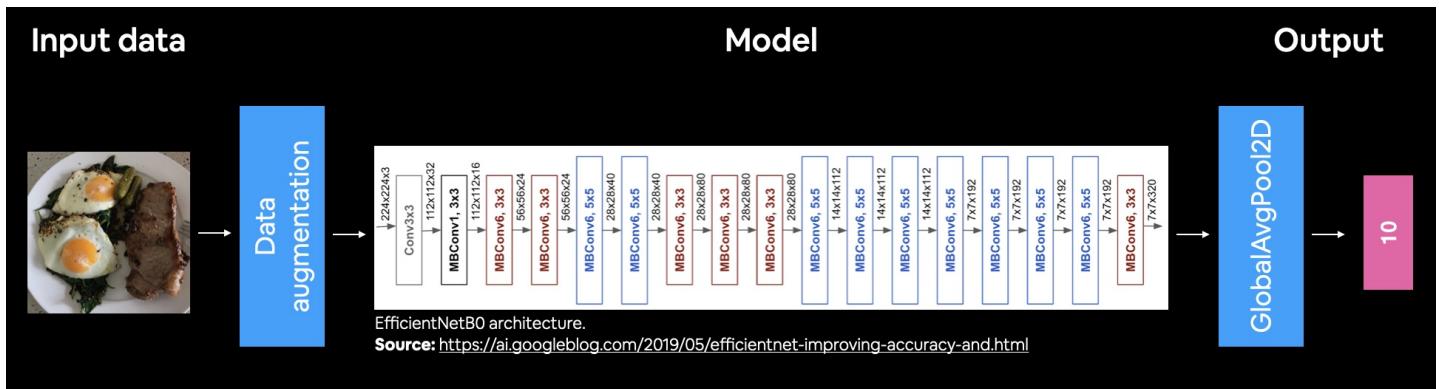
We'll import the base model using the parameter `include_top=False` so we can add on our own output layers, notably `GlobalAveragePooling2D()` (condense the outputs of the base model into a shape usable by the output layer) followed by a `Dense` layer.

In [ ]:

```
# Setup base model and freeze its layers (this will extract features)
base_model = tf.keras.applications.EfficientNetB0(include_top=False)
base_model.trainable = False

# Setup model architecture with trainable top layers
inputs = layers.Input(shape=(224, 224, 3), name="input_layer") # shape of input image
x = data_augmentation(inputs) # augment images (only happens during training)
x = base_model(x, training=False) # put the base model in inference mode so we can use it
# to extract features without updating the weights
x = layers.GlobalAveragePooling2D(name="global_average_pooling")(x) # pool the outputs of
# the base model
outputs = layers.Dense(len(train_data_all_10_percent.class_names), activation="softmax",
name="output_layer")(x) # same number of outputs as classes
model = tf.keras.Model(inputs, outputs)
```

Downloading data from [https://storage.googleapis.com/keras-applications/efficientnetb0\\_no\\_top.h5](https://storage.googleapis.com/keras-applications/efficientnetb0_no_top.h5)  
16711680/16705208 [=====] - 0s 0us/step



A colourful figure of the model we've created with: 224x224 images as input, data augmentation as a layer, EfficientNetB0 as a backbone, an averaging pooling layer as well as dense layer with 10 neurons (same as number of classes we're working with) as output.

Model created. Let's inspect it.

In [ ]:

```
# Get a summary of our model
model.summary()
```

Model: "model"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	[ (None, 224, 224, 3) ]	0
data_augmentation (Sequential)	(None, None, None, 3)	0
efficientnetb0 (Functional)	(None, None, None, 1280)	4049571
global_average_pooling (Glob)	(None, 1280)	0
output_layer (Dense)	(None, 101)	129381

Total params: 4,178,952

Trainable params: 129,381

Non-trainable params: 4,049,571

Looking good! Our Functional model has 5 layers but each of those layers have varying amounts of layers within

Looking good! Our functional model has 8 layers but each of those layers have varying amounts of layers within them.

Notice the number of trainable and non-trainable parameters. It seems the only trainable parameters are within the `output_layer` which is exactly what we're after with this first run of feature extraction; keep all the learned patterns in the base model (`EfficientNetB0`) frozen whilst allowing the model to tune its outputs to our custom data.

Time to compile and fit.

In [ ]:

```
# Compile
model.compile(loss="categorical_crossentropy",
                optimizer=tf.keras.optimizers.Adam(), # use Adam with default settings
                metrics=["accuracy"])

# Fit
history_all_classes_10_percent = model.fit(train_data_all_10_percent,
                                             epochs=5, # fit for 5 epochs to keep experiments quick
                                             validation_data=test_data,
                                             validation_steps=int(0.15 * len(test_data)),
# evaluate on smaller portion of test data
                                             callbacks=[checkpoint_callback]) # save best
model weights to file
```

```
Epoch 1/5
237/237 [=====] - 62s 217ms/step - loss: 3.9868 - accuracy: 0.15
15 - val_loss: 2.6857 - val_accuracy: 0.3954
Epoch 2/5
237/237 [=====] - 46s 194ms/step - loss: 2.4503 - accuracy: 0.45
16 - val_loss: 2.2349 - val_accuracy: 0.4613
Epoch 3/5
237/237 [=====] - 46s 191ms/step - loss: 2.0245 - accuracy: 0.52
42 - val_loss: 2.0737 - val_accuracy: 0.4799
Epoch 4/5
237/237 [=====] - 44s 184ms/step - loss: 1.7955 - accuracy: 0.57
01 - val_loss: 2.0238 - val_accuracy: 0.4846
Epoch 5/5
237/237 [=====] - 43s 182ms/step - loss: 1.6464 - accuracy: 0.59
74 - val_loss: 1.9240 - val_accuracy: 0.5077
```

Woah! It looks like our model is getting some impressive results, but remember, during training our model only evaluated on 15% of the test data. Let's see how it did on the whole test dataset.

In [ ]:

```
# Evaluate model
results_feature_extraction_model = model.evaluate(test_data)
results_feature_extraction_model
```

```
790/790 [=====] - 83s 105ms/step - loss: 1.7309 - accuracy: 0.54
76
```

Out [ ]:

```
[1.7308824062347412, 0.5476435422897339]
```

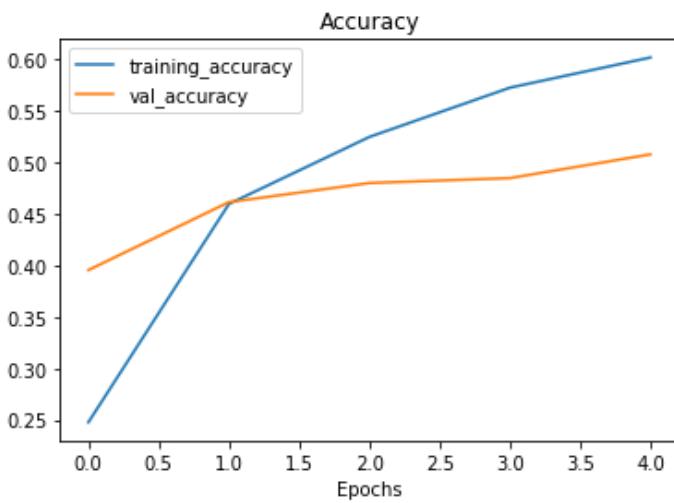
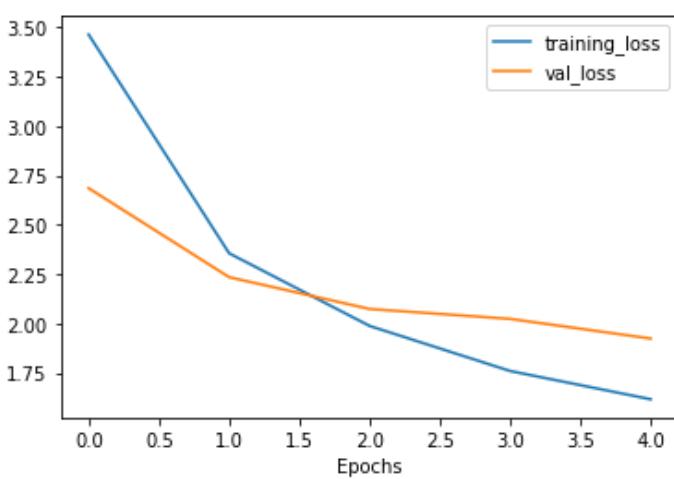
Well it looks like we just beat our baseline (the results from the original Food101 paper) with 10% of the data ! In under 5-minutes... that's the power of deep learning and more precisely, transfer learning: leveraging what one model has learned on another dataset for our own dataset.

How do the loss curves look?

In [ ]:

```
plot_loss_curves(history_all_classes_10_percent)
```

Loss



Question: What do these curves suggest? Hint: ideally, the two curves should be very similar to each other, if not, there may be some overfitting or underfitting.

## Fine-tuning

Our feature extraction transfer learning model is performing well. Why don't we try to fine-tune a few layers in the base model and see if we gain any improvements?

The good news is, thanks to the `ModelCheckpoint` callback, we've got the saved weights of our already well-performing model so if fine-tuning doesn't add any benefits, we can revert back.

To fine-tune the base model we'll first set its `trainable` attribute to `True`, unfreezing all of the frozen.

Then since we've got a relatively small training dataset, we'll refreeze every layer except for the last 5, making them trainable.

In [ ]:

```
# Unfreeze all of the layers in the base model
base_model.trainable = True

# Refreeze every layer except for the last 5
for layer in base_model.layers[:-5]:
    layer.trainable = False
```

We just made a change to the layers in our model and what do we have to do every time we make a change to our model?

Recompile it.

Because we're fine-tuning, we'll use a 10x lower learning rate to ensure the updates to the previous trained

weights aren't too large.



When fine-tuning and unfreezing layers of your pre-trained model, it's common practice to lower the learning rate you used for your feature extraction model. How much by? A 10x lower learning rate is usually a good place to start.

In [ ]:

```
# Recompile model with lower learning rate
model.compile(loss='categorical_crossentropy',
              optimizer=tf.keras.optimizers.Adam(1e-4), # 10x lower learning rate than default
              metrics=['accuracy'])
```

Model recompiled, how about we make sure the layers we want are trainable?

In [ ]:

```
# What layers in the model are trainable?
for layer in model.layers:
    print(layer.name, layer.trainable)
```

```
input_layer True
data_augmentation True
efficientnetb0 True
global_average_pooling True
output_layer True
```

In [ ]:

```
# Check which layers are trainable
for layer_number, layer in enumerate(base_model.layers):
    print(layer_number, layer.name, layer.trainable)
```

```
0 input_1 False
1 rescaling False
2 normalization False
3 stem_conv_pad False
4 stem_conv False
5 stem_bn False
6 stem_activation False
7 block1a_dwconv False
8 block1a_bn False
9 block1a_activation False
10 block1a_se_squeeze False
11 block1a_se_reshape False
12 block1a_se_reduce False
13 block1a_se_expand False
14 block1a_se_excite False
15 block1a_project_conv False
16 block1a_project_bn False
17 block2a_expand_conv False
```

```
-- ~~~~~_~~~~~_~~~~~_~~~~~ --~  
18 block2a_expand_bn False  
19 block2a_expand_activation False  
20 block2a_dwconv_pad False  
21 block2a_dwconv False  
22 block2a_bn False  
23 block2a_activation False  
24 block2a_se_squeeze False  
25 block2a_se_reshape False  
26 block2a_se_reduce False  
27 block2a_se_expand False  
28 block2a_se_excite False  
29 block2a_project_conv False  
30 block2a_project_bn False  
31 block2b_expand_conv False  
32 block2b_expand_bn False  
33 block2b_expand_activation False  
34 block2b_dwconv False  
35 block2b_bn False  
36 block2b_activation False  
37 block2b_se_squeeze False  
38 block2b_se_reshape False  
39 block2b_se_reduce False  
40 block2b_se_expand False  
41 block2b_se_excite False  
42 block2b_project_conv False  
43 block2b_project_bn False  
44 block2b_drop False  
45 block2b_add False  
46 block3a_expand_conv False  
47 block3a_expand_bn False  
48 block3a_expand_activation False  
49 block3a_dwconv_pad False  
50 block3a_dwconv False  
51 block3a_bn False  
52 block3a_activation False  
53 block3a_se_squeeze False  
54 block3a_se_reshape False  
55 block3a_se_reduce False  
56 block3a_se_expand False  
57 block3a_se_excite False  
58 block3a_project_conv False  
59 block3a_project_bn False  
60 block3b_expand_conv False  
61 block3b_expand_bn False  
62 block3b_expand_activation False  
63 block3b_dwconv False  
64 block3b_bn False  
65 block3b_activation False  
66 block3b_se_squeeze False  
67 block3b_se_reshape False  
68 block3b_se_reduce False  
69 block3b_se_expand False  
70 block3b_se_excite False  
71 block3b_project_conv False  
72 block3b_project_bn False  
73 block3b_drop False  
74 block3b_add False  
75 block4a_expand_conv False  
76 block4a_expand_bn False  
77 block4a_expand_activation False  
78 block4a_dwconv_pad False  
79 block4a_dwconv False  
80 block4a_bn False  
81 block4a_activation False  
82 block4a_se_squeeze False  
83 block4a_se_reshape False  
84 block4a_se_reduce False  
85 block4a_se_expand False  
86 block4a_se_excite False  
87 block4a_project_conv False  
88 block4a_project_bn False  
89 block4b_expand_conv False
```

```
90 block4b_expand_bn False
91 block4b_expand_activation False
92 block4b_dwconv False
93 block4b_bn False
94 block4b_activation False
95 block4b_se_squeeze False
96 block4b_se_reshape False
97 block4b_se_reduce False
98 block4b_se_expand False
99 block4b_se_excite False
100 block4b_project_conv False
101 block4b_project_bn False
102 block4b_drop False
103 block4b_add False
104 block4c_expand_conv False
105 block4c_expand_bn False
106 block4c_expand_activation False
107 block4c_dwconv False
108 block4c_bn False
109 block4c_activation False
110 block4c_se_squeeze False
111 block4c_se_reshape False
112 block4c_se_reduce False
113 block4c_se_expand False
114 block4c_se_excite False
115 block4c_project_conv False
116 block4c_project_bn False
117 block4c_drop False
118 block4c_add False
119 block5a_expand_conv False
120 block5a_expand_bn False
121 block5a_expand_activation False
122 block5a_dwconv False
123 block5a_bn False
124 block5a_activation False
125 block5a_se_squeeze False
126 block5a_se_reshape False
127 block5a_se_reduce False
128 block5a_se_expand False
129 block5a_se_excite False
130 block5a_project_conv False
131 block5a_project_bn False
132 block5b_expand_conv False
133 block5b_expand_bn False
134 block5b_expand_activation False
135 block5b_dwconv False
136 block5b_bn False
137 block5b_activation False
138 block5b_se_squeeze False
139 block5b_se_reshape False
140 block5b_se_reduce False
141 block5b_se_expand False
142 block5b_se_excite False
143 block5b_project_conv False
144 block5b_project_bn False
145 block5b_drop False
146 block5b_add False
147 block5c_expand_conv False
148 block5c_expand_bn False
149 block5c_expand_activation False
150 block5c_dwconv False
151 block5c_bn False
152 block5c_activation False
153 block5c_se_squeeze False
154 block5c_se_reshape False
155 block5c_se_reduce False
156 block5c_se_expand False
157 block5c_se_excite False
158 block5c_project_conv False
159 block5c_project_bn False
160 block5c_drop False
161 block5c_add False
```

```
-- ~~~~~ --~--  
162 block6a_expand_conv False  
163 block6a_expand_bn False  
164 block6a_expand_activation False  
165 block6a_dwconv_pad False  
166 block6a_dwconv False  
167 block6a_bn False  
168 block6a_activation False  
169 block6a_se_squeeze False  
170 block6a_se_reshape False  
171 block6a_se_reduce False  
172 block6a_se_expand False  
173 block6a_se_excite False  
174 block6a_project_conv False  
175 block6a_project_bn False  
176 block6b_expand_conv False  
177 block6b_expand_bn False  
178 block6b_expand_activation False  
179 block6b_dwconv False  
180 block6b_bn False  
181 block6b_activation False  
182 block6b_se_squeeze False  
183 block6b_se_reshape False  
184 block6b_se_reduce False  
185 block6b_se_expand False  
186 block6b_se_excite False  
187 block6b_project_conv False  
188 block6b_project_bn False  
189 block6b_drop False  
190 block6b_add False  
191 block6c_expand_conv False  
192 block6c_expand_bn False  
193 block6c_expand_activation False  
194 block6c_dwconv False  
195 block6c_bn False  
196 block6c_activation False  
197 block6c_se_squeeze False  
198 block6c_se_reshape False  
199 block6c_se_reduce False  
200 block6c_se_expand False  
201 block6c_se_excite False  
202 block6c_project_conv False  
203 block6c_project_bn False  
204 block6c_drop False  
205 block6c_add False  
206 block6d_expand_conv False  
207 block6d_expand_bn False  
208 block6d_expand_activation False  
209 block6d_dwconv False  
210 block6d_bn False  
211 block6d_activation False  
212 block6d_se_squeeze False  
213 block6d_se_reshape False  
214 block6d_se_reduce False  
215 block6d_se_expand False  
216 block6d_se_excite False  
217 block6d_project_conv False  
218 block6d_project_bn False  
219 block6d_drop False  
220 block6d_add False  
221 block7a_expand_conv False  
222 block7a_expand_bn False  
223 block7a_expand_activation False  
224 block7a_dwconv False  
225 block7a_bn False  
226 block7a_activation False  
227 block7a_se_squeeze False  
228 block7a_se_reshape False  
229 block7a_se_reduce False  
230 block7a_se_expand False  
231 block7a_se_excite False  
232 block7a_project_conv True  
233 block7a_project_bn True
```

```
234 top_conv True  
235 top_bn True  
236 top_activation True
```

**Excellent! Time to fine-tune our model.**

**Another 5 epochs should be enough to see whether any benefits come about (though we could always try more).**

**We'll start the training off where the feature extraction model left off using the `initial_epoch` parameter in the `fit()` function.**

In [ ]:

```
# Fine-tune for 5 more epochs  
fine_tune_epochs = 10 # model has already done 5 epochs, this is the total number of epochs we're after (5+5=10)  
  
history_all_classes_10_percent_fine_tune = model.fit(train_data_all_10_percent,  
                                                    epochs=fine_tune_epochs,  
                                                    validation_data=test_data,  
                                                    validation_steps=int(0.15 * len(test_data)), # validate on 15% of the test data  
                                                    initial_epoch=history_all_classes_10_percent.epoch[-1]) # start from previous last epoch
```

```
Epoch 5/10  
237/237 [=====] - 49s 184ms/step - loss: 1.4271 - accuracy: 0.64  
13 - val_loss: 1.9473 - val_accuracy: 0.4979  
Epoch 6/10  
237/237 [=====] - 42s 177ms/step - loss: 1.2777 - accuracy: 0.67  
06 - val_loss: 1.9605 - val_accuracy: 0.5008  
Epoch 7/10  
237/237 [=====] - 42s 177ms/step - loss: 1.1918 - accuracy: 0.68  
68 - val_loss: 1.9430 - val_accuracy: 0.5016  
Epoch 8/10  
237/237 [=====] - 42s 176ms/step - loss: 1.1052 - accuracy: 0.71  
12 - val_loss: 1.9121 - val_accuracy: 0.5074  
Epoch 9/10  
237/237 [=====] - 42s 175ms/step - loss: 1.0649 - accuracy: 0.72  
21 - val_loss: 1.9090 - val_accuracy: 0.5087  
Epoch 10/10  
237/237 [=====] - 41s 172ms/step - loss: 0.9865 - accuracy: 0.73  
86 - val_loss: 1.9170 - val_accuracy: 0.5077
```

**Once again, during training we were only evaluating on a small portion of the test data, let's find out how our model went on all of the test data.**

In [ ]:

```
# Evaluate fine-tuned model on the whole test dataset  
results_all_classes_10_percent_fine_tune = model.evaluate(test_data)  
results_all_classes_10_percent_fine_tune
```

```
790/790 [=====] - 83s 105ms/step - loss: 1.6188 - accuracy: 0.57  
60
```

Out [ ]:

```
[1.6187509298324585, 0.5759603977203369]
```

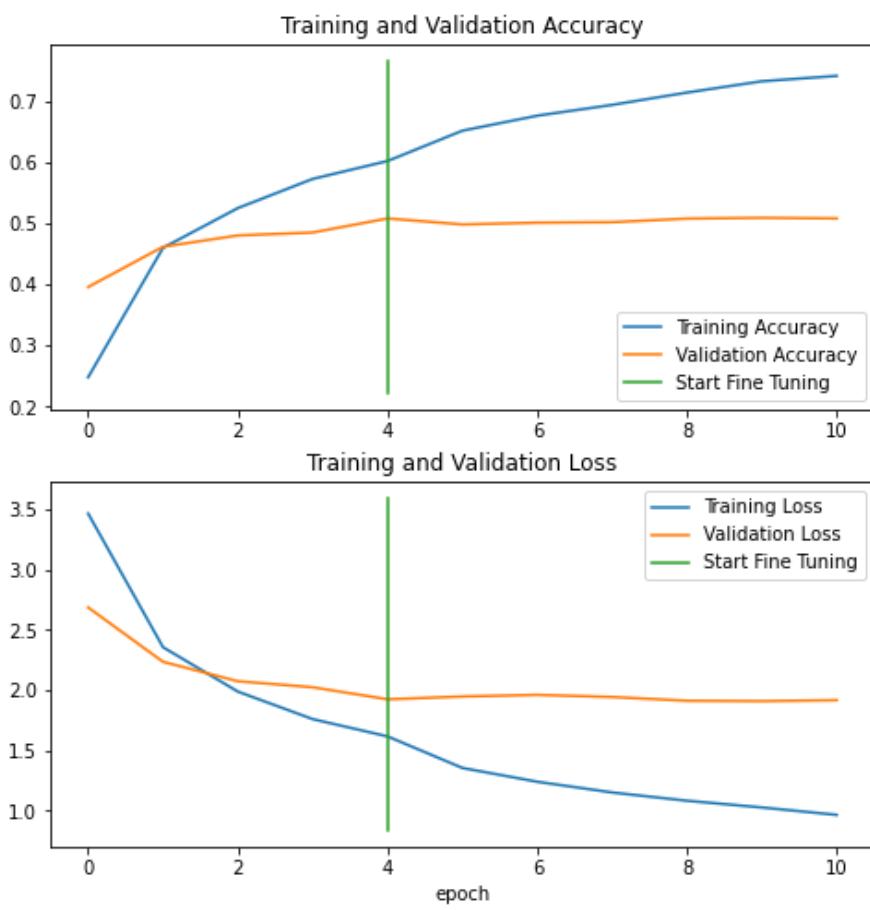
**Hmm... it seems like our model got a slight boost from fine-tuning.**

**We might get a better picture by using our `compare_histories()` function and seeing what the training curves say.**

In [ ]:

```
compare_histories(original_history=history_all_classes_10_percent,  
                  new_history=history_all_classes_10_percent_fine_tune,
```

```
initial_epochs=5)
```



It seems that after fine-tuning, our model's training metrics improved significantly but validation, not so much. Looks like our model is starting to overfit.

This is okay though, it's very often the case that fine-tuning leads to overfitting when the data a pre-trained model has been trained on is similar to your custom data.

In our case, our pre-trained model, `EfficientNetB0` was trained on [ImageNet](#) which contains many real life pictures of food just like our food dataset.

If feature extraction already works well, the improvements you see from fine-tuning may not be as great as if your dataset was significantly different from the data your base model was pre-trained on.

## Saving our trained model

To prevent having to retrain our model from scratch, let's save it to file using the `save()` method.

In [ ]:

```
# # Save model to drive so it can be used later
# model.save("drive/My Drive/tensorflow_course/101_food_class_10_percent_saved_big_dog_model")
```

## Evaluating the performance of the big dog model across all different classes

We've got a trained and saved model which according to the evaluation metrics we've used is performing fairly well.

But metrics schmetrics, let's dive a little deeper into our model's performance and get some visualizations going.

To do so, we'll load in the saved model and use it to make some predictions on the test dataset.

**Note:** Evaluating a machine learning model is as important as training one. Metrics can be deceiving. You should always visualize your model's performance on unseen data to make sure you aren't being fooled by good looking training numbers.

In [ ]:

```
import tensorflow as tf

# Download pre-trained model from Google Storage (like a cooking show, I trained this model earlier, so the results may be different than above)
!wget https://storage.googleapis.com/ztm_tf_course/food_vision/06_101_food_class_10_percent_saved_big_dog_model.zip
saved_model_path = "06_101_food_class_10_percent_saved_big_dog_model.zip"
unzip_data(saved_model_path)

# Note: loading a model will output a lot of 'WARNINGS', these can be ignored: https://www.tensorflow.org/tutorials/keras/save_and_load#save_checkpoints_during_training
# There's also a thread on GitHub trying to fix these warnings: https://github.com/tensorflow/tensorflow/issues/40166
# model = tf.keras.models.load_model("drive/My Drive/tensorflow_course/101_food_class_10_percent_saved_big_dog_model/") # path to drive model
model = tf.keras.models.load_model(saved_model_path.split(".") [0]) # don't include ".zip" in loaded model path
```

To make sure our loaded model is indeed a trained model, let's evaluate its performance on the test dataset.

In [ ]:

```
# Check to see if loaded model is a trained model
loaded_loss, loaded_accuracy = model.evaluate(test_data)
loaded_loss, loaded_accuracy
```

```
790/790 [=====] - 101s 126ms/step - loss: 1.8027 - accuracy: 0.6078
```

Out [ ]:

```
(1.8027207851409912, 0.6077623963356018)
```

Wonderful! It looks like our loaded model is performing just as well as it was before we saved it. Let's make some predictions.

## Making predictions with our trained model

To evaluate our trained model, we need to make some predictions with it and then compare those predictions to the test dataset.

Because the model has never seen the test dataset, this should give us an indication of how the model will perform in the real world on data similar to what it has been trained on.

To make predictions with our trained model, we can use the `predict()` method passing it the test data.

Since our data is multi-class, doing this will return a prediction probably tensor for each sample.

In other words, every time the trained model sees an image it will compare it to all of the patterns it learned during training and return an output for every class (all 101 of them) of how likely the image is to be that class.

In [ ]:

```
# Make predictions with model
pred_probs = model.predict(test_data, verbose=1) # set verbosity to see how long it will take
```

```
790/790 [=====] - 97s 121ms/step
```

We just passed all of the test images to our model and asked it to make a prediction on what food it thinks is in each.

**So if we had 25250 images in the test dataset, how many predictions do you think we should have?**

In [ ]:

```
# How many predictions are there?  
len(pred_probs)
```

Out[ ]:

25250

**And if each image could be one of 101 classes, how many predictions do you think we'll have for each image?**

In [ ]:

```
# What's the shape of our predictions?  
pred_probs.shape
```

Out[ ]:

(25250, 101)

**What we've got is often referred to as a **predictions probability tensor** (or array).**

**Let's see what the first 10 look like.**

In [ ]:

```
# How do they look?  
pred_probs[:10]
```

Out[ ]:

```
array([[5.95421158e-02, 3.57422527e-06, 4.13768552e-02, ...,  
       1.41387069e-09, 8.35303435e-05, 3.08974786e-03],  
      [9.64016914e-01, 1.37532208e-09, 8.47803021e-04, ...,  
       5.42867929e-05, 7.83622126e-12, 9.84660353e-10],  
      [9.59258795e-01, 3.25337423e-05, 1.48669060e-03, ...,  
       7.18914805e-07, 5.43971112e-07, 4.02760816e-05],  
      ...,  
      [4.73132193e-01, 1.29311957e-07, 1.48056773e-03, ...,  
       5.97502163e-04, 6.69690344e-05, 2.34693634e-05],  
      [4.45721857e-02, 4.72653312e-07, 1.22585274e-01, ...,  
       6.34984917e-06, 7.53184941e-06, 3.67786852e-03],  
      [7.24389613e-01, 1.92497329e-09, 5.23110903e-05, ...,  
       1.22913963e-03, 1.57927460e-09, 9.63957573e-05]], dtype=float32)
```

**Alright, it seems like we've got a bunch of tensors of really small numbers, how about we zoom into one of them?**

In [ ]:

```
# We get one prediction probability per class  
print(f"Number of prediction probabilities for sample 0: {len(pred_probs[0])}")  
print(f"What prediction probability sample 0 looks like:\n {pred_probs[0]})")  
print(f"The class with the highest predicted probability by the model for sample 0: {pred  
_probs[0].argmax()})")
```

Number of prediction probabilities for sample 0: 101

What prediction probability sample 0 looks like:

```
[5.95421158e-02 3.57422527e-06 4.13768552e-02 1.06605946e-09  
8.16144308e-09 8.66396554e-09 8.09271910e-07 8.56522377e-07  
1.98591461e-05 8.09778328e-07 3.17278626e-09 9.86739224e-07  
2.85323506e-04 7.80493392e-10 7.42299424e-04 3.89162269e-05  
6.47406114e-06 2.49773984e-06 3.78912046e-05 2.06782872e-07  
1.55384951e-05 8.15071758e-07 2.62305662e-06 2.00106840e-07  
8.38276890e-07 5.42158296e-06 3.73910325e-06 1.31505082e-08  
2.77614826e-03 2.80519962e-05 6.85624113e-10 2.55748073e-05  
1.66889426e-04 7.64074304e-10 4.04529506e-04 1.31507072e-08  
1.70574022e-05 1.44400567e-06 2.20020550e-07 2.21670052e-07
```

```
1.19514285e-06 1.4448256e-06 2.5062855e-06 8.2467005e-07  
8.53655195e-07 1.71386603e-06 7.05258344e-06 1.84022007e-08  
2.85532110e-07 7.94834523e-06 2.06816117e-06 1.85251508e-07  
3.36195107e-08 3.15225829e-04 1.04109231e-05 8.54482664e-07  
8.47418606e-01 1.05554454e-05 4.40947048e-07 3.74042465e-05  
3.53061914e-05 3.24890680e-05 6.73145405e-05 1.28525910e-08  
2.62198568e-10 1.03181483e-05 8.57435443e-05 1.05699053e-06  
2.12934742e-06 3.76375865e-05 7.59729986e-08 2.53406790e-04  
9.29062082e-07 1.25981722e-04 6.26215387e-06 1.24587505e-08  
4.05198007e-05 6.87281130e-08 1.25463293e-06 5.28873869e-08  
7.54249214e-08 7.53987842e-05 7.75403678e-05 6.40266194e-07  
9.90336730e-07 2.22258786e-05 1.50139331e-05 1.40384884e-07  
1.22325328e-05 1.90448221e-02 4.99993621e-05 4.62263915e-06  
1.53881501e-07 3.38241279e-07 3.92283273e-09 1.65637033e-07  
8.13207589e-05 4.89653439e-06 2.40683505e-07 2.31240901e-05  
3.10405565e-04 3.13800338e-05 1.41387069e-09 8.35303435e-05  
3.08974786e-03]
```

The class with the highest predicted probability by the model for sample 0: 52

As we discussed before, for each image tensor we pass to our model, because of the number of output neurons and activation function in the last layer (`layers.Dense(len(train_data_all_10_percent.class_names), activation="softmax")`), it outputs a prediction probability between 0 and 1 for all each of the 101 classes.

And the index of the highest prediction probability can be considered what the model thinks is the most likely label. Similarly, the lower prediction probability value, the less the model thinks that the target image is that specific class.

**Note:** Due to the nature of the softmax activation function, the sum of each of the prediction probabilities for a single sample will be 1 (or at least very close to 1). E.g. `pred_probs[0].sum() = 1`.

We can find the index of the maximum value in each prediction probability tensor using the `argmax()` method.

In [ ]:

```
# Get the class predictions of each label  
pred_classes = pred_probs.argmax(axis=1)
```

```
# How do they look?  
pred_classes[:10]
```

Out [ ]:

```
array([52,  0,  0, 80, 79, 61, 29,  0, 85,  0])
```

Beautiful! We've now got the predicted class index for each of the samples in our test dataset.

We'll be able to compare these to the test dataset labels to further evaluate our model.

To get the test dataset labels we can unravel our `test_data` object (which is in the form of a `tf.data.Dataset`) using the `unbatch()` method.

Doing this will give us access to the images and labels in the test dataset. Since the labels are in one-hot encoded format, we'll take use the `argmax()` method to return the index of the label.

**Note:** This unravelling is why we `shuffle=False` when creating the test data object. Otherwise, whenever we loaded the test dataset (like when making predictions), it would be shuffled every time, meaning if we tried to compare our predictions to the labels, they would be in different orders.

In [ ]:

```
# Note: This might take a minute or so due to unravelling 790 batches  
y_labels = []
```

```
for images, labels in test_data.unbatch(): # unbatch the test data and get images and labels
    y_labels.append(labels.numpy().argmax()) # append the index which has the largest value (labels are one-hot)
y_labels[:10] # check what they look like (unshuffled)
```

In [ ]:

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

Nice! Since `test_data` isn't shuffled, the `y_labels` array comes back in the same order as the `pred_classes` array.

The final check is to see how many labels we've got.

In [ ]:

```
# How many labels are there? (should be the same as how many prediction probabilities we have)
len(y_labels)
```

Out [ ]:

```
25250
```

As expected, the number of labels matches the number of images we've got. Time to compare our model's predictions with the ground truth labels.

## Evaluating our models predictions

A very simple evaluation is to use Scikit-Learn's `accuracy_score()` function which compares truth labels to predicted labels and returns an accuracy score.

If we've created our `y_labels` and `pred_classes` arrays correctly, this should return the same accuracy value (or at least very close) as the `evaluate()` method we used earlier.

In [ ]:

```
# Get accuracy score by comparing predicted classes to ground truth labels
from sklearn.metrics import accuracy_score
sklearn_accuracy = accuracy_score(y_labels, pred_classes)
sklearn_accuracy
```

Out [ ]:

```
0.6077623762376237
```

In [ ]:

```
# Does the evaluate method compare to the Scikit-Learn measured accuracy?
import numpy as np
print(f"Close? {np.isclose(loaded_accuracy, sklearn_accuracy)} | Difference: {loaded_accuracy - sklearn_accuracy}")
```

```
Close? True | Difference: 2.0097978059574473e-08
```

Okay, it looks like our `pred_classes` array and `y_labels` arrays are in the right orders.

How about we get a little bit more visual with a confusion matrix?

To do so, we'll use our `make_confusion_matrix` function we created in a previous notebook.

In [ ]:

```
# We'll import our make_confusion_matrix function from https://github.com/mrdbourke/tensorflow-deep-learning/blob/main/extras/helper_functions.py
# But if you run it out of the box, it doesn't really work for 101 classes...
```

```
# the cell below adds a little functionality to make it readable.  
from helper_functions import make_confusion_matrix
```

In [ ]:

```
# Note: The following confusion matrix code is a remix of Scikit-Learn's  
# plot_confusion_matrix function - https://scikit-learn.org/stable/modules/generated/sklearn.metrics.plot_confusion_matrix.html  
import itertools  
import matplotlib.pyplot as plt  
import numpy as np  
from sklearn.metrics import confusion_matrix  
  
# Our function needs a different name to sklearn's plot_confusion_matrix  
def make_confusion_matrix(y_true, y_pred, classes=None, figsize=(10, 10), text_size=15,  
norm=False, savefig=False):  
    """Makes a labelled confusion matrix comparing predictions and ground truth labels.  
  
    If classes is passed, confusion matrix will be labelled, if not, integer class values  
    will be used.  
  
    Args:  
        y_true: Array of truth labels (must be same shape as y_pred).  
        y_pred: Array of predicted labels (must be same shape as y_true).  
        classes: Array of class labels (e.g. string form). If 'None', integer labels are used  
        .  
        figsize: Size of output figure (default=(10, 10)).  
        text_size: Size of output figure text (default=15).  
        norm: normalize values or not (default=False).  
        savefig: save confusion matrix to file (default=False).  
  
    Returns:  
        A labelled confusion matrix plot comparing y_true and y_pred.  
  
    Example usage:  
        make_confusion_matrix(y_true=test_labels, # ground truth test labels  
                             y_pred=y_preds, # predicted labels  
                             classes=class_names, # array of class label names  
                             figsize=(15, 15),  
                             text_size=10)  
    """  
    # Create the confustion matrix  
    cm = confusion_matrix(y_true, y_pred)  
    cm_norm = cm.astype("float") / cm.sum(axis=1)[:, np.newaxis] # normalize it  
    n_classes = cm.shape[0] # find the number of classes we're dealing with  
  
    # Plot the figure and make it pretty  
    fig, ax = plt.subplots(figsize=figsize)  
    cax = ax.matshow(cm, cmap=plt.cm.Blues) # colors will represent how 'correct' a class  
is, darker == better  
    fig.colorbar(cax)  
  
    # Are there a list of classes?  
    if classes:  
        labels = classes  
    else:  
        labels = np.arange(cm.shape[0])  
  
    # Label the axes  
    ax.set(title="Confusion Matrix",  
          xlabel="Predicted label",  
          ylabel="True label",  
          xticks=np.arange(n_classes), # create enough axis slots for each class  
          yticks=np.arange(n_classes),  
          xticklabels=labels, # axes will labeled with class names (if they exist) or int  
s  
          yticklabels=labels)  
  
    # Make x-axis labels appear on bottom  
    ax.xaxis.set_label_position("bottom")  
    ax.xaxis.tick_bottom()
```

```

### Added: Rotate xticks for readability & increase font size (required due to such a large confusion matrix)
plt.xticks(rotation=70, fontsize=text_size)
plt.yticks(fontsize=text_size)

# Set the threshold for different colors
threshold = (cm.max() + cm.min()) / 2.

# Plot the text on each cell
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    if norm:
        plt.text(j, i, f"{cm[i, j]} ({cm_norm[i, j]*100:.1f}%)",
                  horizontalalignment="center",
                  color="white" if cm[i, j] > threshold else "black",
                  size=text_size)
    else:
        plt.text(j, i, f"{cm[i, j]}",
                  horizontalalignment="center",
                  color="white" if cm[i, j] > threshold else "black",
                  size=text_size)

# Save the figure to the current working directory
if savefig:
    fig.savefig("confusion_matrix.png")

```

**Right now our predictions and truth labels are in the form of integers, however, they'll be much easier to understand if we get their actual names. We can do so using the `class_names` attribute on our `test_data` object.**

In [ ]:

```
# Get the class names
class_names = test_data.class_names
class_names
```

Out[ ]:

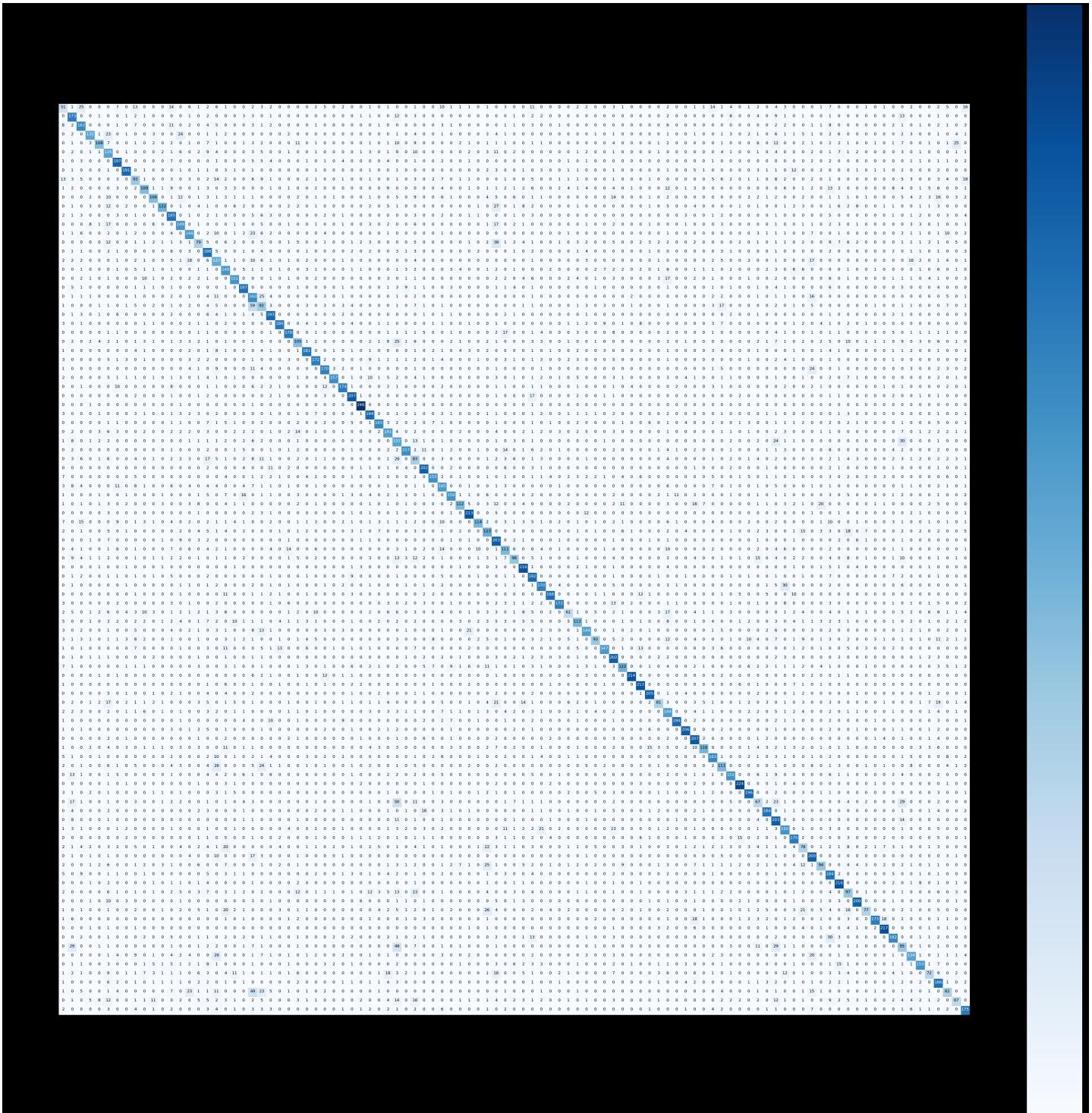
```
['apple_pie',
 'baby_back_ribs',
 'baklava',
 'beef_carpaccio',
 'beef_tartare',
 'beet_salad',
 'beignets',
 'bibimbap',
 'bread_pudding',
 'breakfast_burrito',
 'bruschetta',
 'caesar_salad',
 'cannoli',
 'caprese_salad',
 'carrot_cake',
 'ceviche',
 'cheese_plate',
 'cheesecake',
 'chicken_curry',
 'chicken_quesadilla',
 'chicken_wings',
 'chocolate_cake',
 'chocolate_mousse',
 'churros',
 'clam_chowder',
 'club_sandwich',
 'crab_cakes',
 'creme_brulee',
 'croque_madame',
 'cup_cakes',
 'deviled_eggs',
 'donuts',
 'dumplings',
```

```
'edamame',
'eggs_benedict',
'escargots',
'falafel',
'filet_mignon',
'fish_and_chips',
'foie_gras',
'french_fries',
'french_onion_soup',
'french_toast',
'fried_calamari',
'fried_rice',
'frozen_yogurt',
'garlic_bread',
'gnocchi',
'greek_salad',
'grilled_cheese_sandwich',
'grilled_salmon',
'guacamole',
'gyoza',
'hamburger',
'hot_and_sour_soup',
'hot_dog',
'huevos_rancheros',
'hummus',
'ice_cream',
'lasagna',
'lobster_bisque',
'lobster_roll_sandwich',
'macaroni_and_cheese',
'macarons',
'miso_soup',
'mussels',
'nachos',
'omelette',
'onion_rings',
'oysters',
'pad_thai',
'paella',
'pancakes',
'panna_cotta',
'peking_duck',
'pho',
'pizza',
'pork_chop',
'poutine',
'prime_rib',
'pulled_pork_sandwich',
'ramen',
'ravioli',
'red_velvet_cake',
'risotto',
'samosa',
'sashimi',
'scallops',
'seaweed_salad',
'shrimp_and_grits',
'spaghetti_bolognese',
'spaghetti_carbonara',
'spring_rolls',
'steak',
'strawberry_shortcake',
'sushi',
'tacos',
'takoyaki',
'tiramisu',
'tuna_tartare',
>waffles']
```

**101 class names and 25250 predictions and ground truth labels ready to go! Looks like our confusion matrix is going to be a big one!**

In [ ]:

```
# Plot a confusion matrix with all 25250 predictions, ground truth labels and 101 classes
make_confusion_matrix(y_true=y_labels,
                      y_pred=pred_classes,
                      classes=class_names,
                      figsize=(100, 100),
                      text_size=20,
                      norm=False,
                      savefig=True)
```



Woah! Now that's a big confusion matrix. It may look a little daunting at first but after zooming in a little, we can see how it gives us insight into which classes its getting "confused" on.

The good news is, the majority of the predictions are right down the top left to bottom right diagonal, meaning they're correct.

It looks like the model gets most confused on classes which look visually similar, such as predicting `filet_mignon` for instances of `pork_chop` and `chocolate_cake` for instances of `tiramisu`.

Since we're working on a classification problem, we can further evaluate our model's predictions using Scikit-

## Learn's `classification_report()` function.

In [ ]:

```
from sklearn.metrics import classification_report
print(classification_report(y_labels, pred_classes))
```

	precision	recall	f1-score	support
0	0.29	0.20	0.24	250
1	0.51	0.69	0.59	250
2	0.56	0.65	0.60	250
3	0.74	0.53	0.62	250
4	0.73	0.43	0.54	250
5	0.34	0.54	0.42	250
6	0.67	0.79	0.72	250
7	0.82	0.76	0.79	250
8	0.40	0.37	0.39	250
9	0.62	0.44	0.51	250
10	0.62	0.42	0.50	250
11	0.84	0.49	0.62	250
12	0.52	0.74	0.61	250
13	0.56	0.60	0.58	250
14	0.56	0.59	0.57	250
15	0.44	0.32	0.37	250
16	0.45	0.75	0.57	250
17	0.37	0.51	0.43	250
18	0.43	0.60	0.50	250
19	0.68	0.60	0.64	250
20	0.68	0.75	0.71	250
21	0.35	0.64	0.45	250
22	0.30	0.37	0.33	250
23	0.66	0.77	0.71	250
24	0.83	0.72	0.77	250
25	0.76	0.71	0.73	250
26	0.51	0.42	0.46	250
27	0.78	0.72	0.75	250
28	0.70	0.69	0.69	250
29	0.70	0.68	0.69	250
30	0.92	0.63	0.75	250
31	0.78	0.70	0.74	250
32	0.75	0.83	0.79	250
33	0.89	0.98	0.94	250
34	0.68	0.78	0.72	250
35	0.78	0.66	0.72	250
36	0.53	0.56	0.55	250
37	0.30	0.55	0.39	250
38	0.78	0.63	0.69	250
39	0.27	0.33	0.30	250
40	0.72	0.81	0.76	250
41	0.81	0.62	0.70	250
42	0.50	0.58	0.54	250
43	0.75	0.60	0.67	250
44	0.74	0.45	0.56	250
45	0.77	0.85	0.81	250
46	0.81	0.46	0.58	250
47	0.44	0.49	0.46	250
48	0.45	0.81	0.58	250
49	0.50	0.44	0.47	250
50	0.54	0.39	0.46	250
51	0.71	0.86	0.78	250
52	0.51	0.77	0.61	250
53	0.67	0.68	0.68	250
54	0.88	0.75	0.81	250
55	0.86	0.69	0.76	250
56	0.56	0.24	0.34	250
57	0.62	0.45	0.52	250
58	0.68	0.58	0.62	250
59	0.70	0.37	0.49	250
60	0.83	0.59	0.69	250
61	0.54	0.81	0.65	250
62	0.72	0.49	0.58	250

63	0.94	0.86	0.90	250
64	0.78	0.85	0.81	250
65	0.82	0.82	0.82	250
66	0.69	0.32	0.44	250
67	0.41	0.58	0.48	250
68	0.90	0.78	0.83	250
69	0.84	0.82	0.83	250
70	0.62	0.83	0.71	250
71	0.81	0.46	0.59	250
72	0.64	0.65	0.65	250
73	0.51	0.44	0.47	250
74	0.72	0.61	0.66	250
75	0.84	0.90	0.87	250
76	0.78	0.78	0.78	250
77	0.36	0.27	0.31	250
78	0.79	0.74	0.76	250
79	0.44	0.81	0.57	250
80	0.57	0.60	0.59	250
81	0.65	0.70	0.68	250
82	0.38	0.31	0.34	250
83	0.58	0.80	0.67	250
84	0.61	0.38	0.47	250
85	0.44	0.74	0.55	250
86	0.71	0.86	0.78	250
87	0.41	0.39	0.40	250
88	0.83	0.80	0.81	250
89	0.71	0.31	0.43	250
90	0.92	0.69	0.79	250
91	0.83	0.87	0.85	250
92	0.68	0.65	0.67	250
93	0.31	0.38	0.34	250
94	0.61	0.54	0.57	250
95	0.74	0.61	0.67	250
96	0.56	0.29	0.38	250
97	0.45	0.74	0.56	250
98	0.47	0.33	0.39	250
99	0.52	0.27	0.35	250
100	0.59	0.70	0.64	250
accuracy			0.61	25250
macro avg	0.63	0.61	0.61	25250
weighted avg	0.63	0.61	0.61	25250

The `classification_report()` outputs the precision, recall and f1-score's per class.

A reminder:

- **Precision** - Proportion of true positives over total number of samples. Higher precision leads to less false positives (model predicts 1 when it should've been 0).
- **Recall** - Proportion of true positives over total number of true positives and false negatives (model predicts 0 when it should've been 1). Higher recall leads to less false negatives.
- **F1 score** - Combines precision and recall into one metric. 1 is best, 0 is worst.

The above output is helpful but with so many classes, it's a bit hard to understand.

Let's see if we make it easier with the help of a visualization.

First, we'll get the output of `classification_report()` as a dictionary by setting `output_dict=True`.

In [ ]:

```
# Get a dictionary of the classification report
classification_report_dict = classification_report(y_labels, pred_classes, output_dict=True)
classification_report_dict
```

Out [ ]:

```
{'0': {'f1-score': 0.24056603773584903,
       'precision': 0.29310344827586204,
```

```
'precision': 0.6204007021000001,  
'recall': 0.204,  
'support': 250},  
'1': {'f1-score': 0.5864406779661017,  
'precision': 0.5088235294117647,  
'recall': 0.692,  
'support': 250},  
'10': {'f1-score': 0.5047619047619047,  
'precision': 0.6235294117647059,  
'recall': 0.424,  
'support': 250},  
'100': {'f1-score': 0.641025641025641,  
'precision': 0.5912162162162162,  
'recall': 0.7,  
'support': 250},  
'11': {'f1-score': 0.6161616161616161,  
'precision': 0.8356164383561644,  
'recall': 0.488,  
'support': 250},  
'12': {'f1-score': 0.6105610561056106,  
'precision': 0.5196629213483146,  
'recall': 0.74,  
'support': 250},  
'13': {'f1-score': 0.5775193798449612,  
'precision': 0.5601503759398496,  
'recall': 0.596,  
'support': 250},  
'14': {'f1-score': 0.574757281553398,  
'precision': 0.5584905660377358,  
'recall': 0.592,  
'support': 250},  
'15': {'f1-score': 0.36744186046511623,  
'precision': 0.4388888888888889,  
'recall': 0.316,  
'support': 250},  
'16': {'f1-score': 0.5654135338345864,  
'precision': 0.4530120481927711,  
'recall': 0.752,  
'support': 250},  
'17': {'f1-score': 0.42546063651591287,  
'precision': 0.3659942363112392,  
'recall': 0.508,  
'support': 250},  
'18': {'f1-score': 0.5008403361344538,  
'precision': 0.4318840579710145,  
'recall': 0.596,  
'support': 250},  
'19': {'f1-score': 0.6411889596602972,  
'precision': 0.6832579185520362,  
'recall': 0.604,  
'support': 250},  
'2': {'f1-score': 0.6022304832713754,  
'precision': 0.5625,  
'recall': 0.648,  
'support': 250},  
'20': {'f1-score': 0.7123809523809523,  
'precision': 0.68,  
'recall': 0.748,  
'support': 250},  
'21': {'f1-score': 0.45261669024045265,  
'precision': 0.350109409190372,  
'recall': 0.64,  
'support': 250},  
'22': {'f1-score': 0.3291592128801431,  
'precision': 0.2977346278317152,  
'recall': 0.368,  
'support': 250},  
'23': {'f1-score': 0.7134935304990757,  
'precision': 0.6632302405498282,  
'recall': 0.772,  
'support': 250},  
'24': {'f1-score': 0.7708779443254817,  
'precision': 0.8294930875576036
```

```
precision : 0.7574468085106383,
'recall': 0.72,
'support': 250},
'25': {'f1-score': 0.734020618556701,
'precision': 0.7574468085106383,
'recall': 0.712,
'support': 250},
'26': {'f1-score': 0.4625550660792952,
'precision': 0.5147058823529411,
'recall': 0.42,
'support': 250},
'27': {'f1-score': 0.7494824016563146,
'precision': 0.776824034334764,
'recall': 0.724,
'support': 250},
'28': {'f1-score': 0.6935483870967742,
'precision': 0.6991869918699187,
'recall': 0.688,
'support': 250},
'29': {'f1-score': 0.6910569105691057,
'precision': 0.7024793388429752,
'recall': 0.68,
'support': 250},
'3': {'f1-score': 0.616822429906542,
'precision': 0.7415730337078652,
'recall': 0.528,
'support': 250},
'30': {'f1-score': 0.7476190476190476,
'precision': 0.9235294117647059,
'recall': 0.628,
'support': 250},
'31': {'f1-score': 0.7357293868921776,
'precision': 0.7802690582959642,
'recall': 0.696,
'support': 250},
'32': {'f1-score': 0.7855787476280836,
'precision': 0.7472924187725631,
'recall': 0.828,
'support': 250},
'33': {'f1-score': 0.9371428571428572,
'precision': 0.8945454545454545,
'recall': 0.984,
'support': 250},
'34': {'f1-score': 0.7238805970149255,
'precision': 0.6783216783216783,
'recall': 0.776,
'support': 250},
'35': {'f1-score': 0.715835140997831,
'precision': 0.7819905213270142,
'recall': 0.66,
'support': 250},
'36': {'f1-score': 0.5475728155339805,
'precision': 0.5320754716981132,
'recall': 0.564,
'support': 250},
'37': {'f1-score': 0.3870056497175141,
'precision': 0.29912663755458513,
'recall': 0.548,
'support': 250},
'38': {'f1-score': 0.6946902654867257,
'precision': 0.7772277227722773,
'recall': 0.628,
'support': 250},
'39': {'f1-score': 0.29749103942652333,
'precision': 0.2694805194805195,
'recall': 0.332,
'support': 250},
'4': {'f1-score': 0.544080604534005,
'precision': 0.7346938775510204,
'recall': 0.432,
'support': 250},
'40': {'f1-score': 0.7622641509433963,
'precision': 0.7214285714285714
```

```
'recall': 0.808,
'support': 250},
'41': {'f1-score': 0.7029478458049886,
'precision': 0.8115183246073299,
'recall': 0.62,
'support': 250},
'42': {'f1-score': 0.537037037037037,
'precision': 0.5,
'recall': 0.58,
'support': 250},
'43': {'f1-score': 0.6651884700665188,
'precision': 0.746268656716418,
'recall': 0.6,
'support': 250},
'44': {'f1-score': 0.5586034912718205,
'precision': 0.7417218543046358,
'recall': 0.448,
'support': 250},
'45': {'f1-score': 0.8114285714285714,
'precision': 0.7745454545454545,
'recall': 0.852,
'support': 250},
'46': {'f1-score': 0.5831202046035805,
'precision': 0.8085106382978723,
'recall': 0.456,
'support': 250},
'47': {'f1-score': 0.4641509433962264,
'precision': 0.4392857142857143,
'recall': 0.492,
'support': 250},
'48': {'f1-score': 0.577524893314367,
'precision': 0.4481236203090508,
'recall': 0.812,
'support': 250},
'49': {'f1-score': 0.47234042553191485,
'precision': 0.5045454545454545,
'recall': 0.444,
'support': 250},
'5': {'f1-score': 0.41860465116279066,
'precision': 0.34177215189873417,
'recall': 0.54,
'support': 250},
'50': {'f1-score': 0.45581395348837206,
'precision': 0.5444444444444444,
'recall': 0.392,
'support': 250},
'51': {'f1-score': 0.7783783783783783,
'precision': 0.7081967213114754,
'recall': 0.864,
'support': 250},
'52': {'f1-score': 0.6124401913875598,
'precision': 0.5092838196286472,
'recall': 0.768,
'support': 250},
'53': {'f1-score': 0.6759443339960238,
'precision': 0.6719367588932806,
'recall': 0.68,
'support': 250},
'54': {'f1-score': 0.8103448275862069,
'precision': 0.8785046728971962,
'recall': 0.752,
'support': 250},
'55': {'f1-score': 0.7644444444444444,
'precision': 0.86,
'recall': 0.688,
'support': 250},
'56': {'f1-score': 0.3398328690807799,
'precision': 0.5596330275229358,
'recall': 0.244,
'support': 250},
'57': {'f1-score': 0.5209302325581396,
'precision': 0.6222222222222222
```

```
precision : 0.6233766233766233,
'recall': 0.448,
'support': 250},
'58': {'f1-score': 0.6233766233766233,
'precision': 0.6792452830188679,
'recall': 0.576,
'support': 250},
'59': {'f1-score': 0.486910994764398,
'precision': 0.7045454545454546,
'recall': 0.372,
'support': 250},
'6': {'f1-score': 0.7229357798165138,
'precision': 0.6677966101694915,
'recall': 0.788,
'support': 250},
'60': {'f1-score': 0.6885245901639344,
'precision': 0.8305084745762712,
'recall': 0.588,
'support': 250},
'61': {'f1-score': 0.6495176848874598,
'precision': 0.543010752688172,
'recall': 0.808,
'support': 250},
'62': {'f1-score': 0.5823389021479712,
'precision': 0.7218934911242604,
'recall': 0.488,
'support': 250},
'63': {'f1-score': 0.895397489539749,
'precision': 0.9385964912280702,
'recall': 0.856,
'support': 250},
'64': {'f1-score': 0.8129770992366412,
'precision': 0.7773722627737226,
'recall': 0.852,
'support': 250},
'65': {'f1-score': 0.82, 'precision': 0.82, 'recall': 0.82, 'support': 250},
'66': {'f1-score': 0.44141689373297005,
'precision': 0.6923076923076923,
'recall': 0.324,
'support': 250},
'67': {'f1-score': 0.47840531561461797,
'precision': 0.4090909090909091,
'recall': 0.576,
'support': 250},
'68': {'f1-score': 0.832618025751073,
'precision': 0.8981481481481481,
'recall': 0.776,
'support': 250},
'69': {'f1-score': 0.8340080971659919,
'precision': 0.8442622950819673,
'recall': 0.824,
'support': 250},
'7': {'f1-score': 0.7908902691511386,
'precision': 0.8197424892703863,
'recall': 0.764,
'support': 250},
'70': {'f1-score': 0.7101200686106347,
'precision': 0.6216216216216216,
'recall': 0.828,
'support': 250},
'71': {'f1-score': 0.5903307888040712,
'precision': 0.8111888111888111,
'recall': 0.464,
'support': 250},
'72': {'f1-score': 0.6468253968253969,
'precision': 0.6417322834645669,
'recall': 0.652,
'support': 250},
'73': {'f1-score': 0.4743589743589744,
'precision': 0.5091743119266054,
'recall': 0.444,
'support': 250},
'74': {'f1-score': 0.658008658008658
```

```
'precision': 0.7169811320754716,
'recall': 0.608,
'support': 250},
'75': {'f1-score': 0.8665377176015473,
'precision': 0.8389513108614233,
'recall': 0.896,
'support': 250},
'76': {'f1-score': 0.7808764940239045,
'precision': 0.7777777777777778,
'recall': 0.784,
'support': 250},
'77': {'f1-score': 0.30875576036866365,
'precision': 0.3641304347826087,
'recall': 0.268,
'support': 250},
'78': {'f1-score': 0.7603305785123966,
'precision': 0.7863247863247863,
'recall': 0.736,
'support': 250},
'79': {'f1-score': 0.571830985915493,
'precision': 0.44130434782608696,
'recall': 0.812,
'support': 250},
'8': {'f1-score': 0.3866943866943867,
'precision': 0.4025974025974026,
'recall': 0.372,
'support': 250},
'80': {'f1-score': 0.5870841487279843,
'precision': 0.5747126436781609,
'recall': 0.6,
'support': 250},
'81': {'f1-score': 0.6756756756756757,
'precision': 0.6529850746268657,
'recall': 0.7,
'support': 250},
'82': {'f1-score': 0.34285714285714286,
'precision': 0.3804878048780488,
'recall': 0.312,
'support': 250},
'83': {'f1-score': 0.6711409395973154,
'precision': 0.5780346820809249,
'recall': 0.8,
'support': 250},
'84': {'f1-score': 0.4653465346534653,
'precision': 0.6103896103896104,
'recall': 0.376,
'support': 250},
'85': {'f1-score': 0.5525525525525525,
'precision': 0.4423076923076923,
'recall': 0.736,
'support': 250},
'86': {'f1-score': 0.7783783783783783,
'precision': 0.7081967213114754,
'recall': 0.864,
'support': 250},
'87': {'f1-score': 0.3975409836065574,
'precision': 0.40756302521008403,
'recall': 0.388,
'support': 250},
'88': {'f1-score': 0.8130081300813008,
'precision': 0.8264462809917356,
'recall': 0.8,
'support': 250},
'89': {'f1-score': 0.4301675977653631,
'precision': 0.7129629629629629,
'recall': 0.308,
'support': 250},
'9': {'f1-score': 0.5117370892018779,
'precision': 0.6193181818181818,
'recall': 0.436,
'support': 250},
'90': {'f1-score': 0.7881548974943051
```

```

'90': {'f1-score': 0.84765625,
'precision': 0.9153439153439153,
'recall': 0.692,
'support': 250},
'91': {'f1-score': 0.84765625,
'precision': 0.8282442748091603,
'recall': 0.868,
'support': 250},
'92': {'f1-score': 0.6652977412731006,
'precision': 0.6835443037974683,
'recall': 0.648,
'support': 250},
'93': {'f1-score': 0.34234234234234234,
'precision': 0.3114754098360656,
'recall': 0.38,
'support': 250},
'94': {'f1-score': 0.5714285714285714,
'precision': 0.6118721461187214,
'recall': 0.536,
'support': 250},
'95': {'f1-score': 0.6710526315789473,
'precision': 0.7427184466019418,
'recall': 0.612,
'support': 250},
'96': {'f1-score': 0.3809523809523809,
'precision': 0.5625,
'recall': 0.288,
'support': 250},
'97': {'f1-score': 0.5644916540212443,
'precision': 0.4547677261613692,
'recall': 0.744,
'support': 250},
'98': {'f1-score': 0.3858823529411765,
'precision': 0.4685714285714286,
'recall': 0.328,
'support': 250},
'99': {'f1-score': 0.35356200527704484,
'precision': 0.5193798449612403,
'recall': 0.268,
'support': 250},
'accuracy': 0.6077623762376237,
'macro avg': {'f1-score': 0.6061252197245781,
'precision': 0.6328666845830312,
'recall': 0.6077623762376237,
'support': 25250},
'weighted avg': {'f1-score': 0.606125219724578,
'precision': 0.6328666845830311,
'recall': 0.6077623762376237,
'support': 25250}}

```

Alright, there's still a fair few values here, how about we narrow down?

Since the f1-score combines precision and recall in one metric, let's focus on that.

To extract it, we'll create an empty dictionary called `class_f1_scores` and then loop through each item in `classification_report_dict`, appending the class name and f1-score as the key, value pairs in `class_f1_scores`.

In [ ]:

```

# Create empty dictionary
class_f1_scores = {}
# Loop through classification report items
for k, v in classification_report_dict.items():
    if k == "accuracy": # stop once we get to accuracy key
        break
    else:
        # Append class names and f1-scores to new dictionary
        class_f1_scores[class_names[int(k)]] = v["f1-score"]
class_f1_scores

```

Out[ ]:

```
{'apple_pie': 0.24056603773584903,
 'baby_back_ribs': 0.5864406779661017,
 'baklava': 0.6022304832713754,
 'beef_carpaccio': 0.616822429906542,
 'beef_tartare': 0.544080604534005,
 'beet_salad': 0.41860465116279066,
 'beignets': 0.7229357798165138,
 'bibimbap': 0.7908902691511386,
 'bread_pudding': 0.3866943866943867,
 'breakfast_burrito': 0.5117370892018779,
 'bruschetta': 0.5047619047619047,
 'caesar_salad': 0.6161616161616161,
 'cannoli': 0.6105610561056106,
 'caprese_salad': 0.5775193798449612,
 'carrot_cake': 0.574757281553398,
 'ceviche': 0.36744186046511623,
 'cheese_plate': 0.5654135338345864,
 'cheesecake': 0.42546063651591287,
 'chicken_curry': 0.5008403361344538,
 'chicken_quesadilla': 0.6411889596602972,
 'chicken_wings': 0.7123809523809523,
 'chocolate_cake': 0.45261669024045265,
 'chocolate_mousse': 0.3291592128801431,
 'churros': 0.7134935304990757,
 'clam_chowder': 0.7708779443254817,
 'club_sandwich': 0.734020618556701,
 'crab_cakes': 0.4625550660792952,
 'creme_brulee': 0.7494824016563146,
 'croque_madame': 0.6935483870967742,
 'cup_cakes': 0.6910569105691057,
 'deviled_eggs': 0.7476190476190476,
 'donuts': 0.7357293868921776,
 'dumplings': 0.7855787476280836,
 'edamame': 0.9371428571428572,
 'eggs_benedict': 0.7238805970149255,
 'escargots': 0.715835140997831,
 'falafel': 0.5475728155339805,
 'filet_mignon': 0.3870056497175141,
 'fish_and_chips': 0.6946902654867257,
 'foie_gras': 0.29749103942652333,
 'french_fries': 0.7622641509433963,
 'french_onion_soup': 0.7029478458049886,
 'french_toast': 0.537037037037037,
 'fried_calamari': 0.6651884700665188,
 'fried_rice': 0.5586034912718205,
 'frozen_yogurt': 0.8114285714285714,
 'garlic_bread': 0.5831202046035805,
 'gnocchi': 0.4641509433962264,
 'greek_salad': 0.577524893314367,
 'grilled_cheese_sandwich': 0.47234042553191485,
 'grilled_salmon': 0.45581395348837206,
 'guacamole': 0.7783783783783783,
 'gyoza': 0.6124401913875598,
 'hamburger': 0.6759443339960238,
 'hot_and_sour_soup': 0.8103448275862069,
 'hot_dog': 0.7644444444444444,
 'huevos_rancheros': 0.3398328690807799,
 'hummus': 0.5209302325581396,
 'ice_cream': 0.6233766233766233,
 'lasagna': 0.486910994764398,
 'lobster_bisque': 0.6885245901639344,
 'lobster_roll_sandwich': 0.6495176848874598,
 'macaroni_and_cheese': 0.5823389021479712,
 'macarons': 0.895397489539749,
 'miso_soup': 0.8129770992366412,
 'mussels': 0.82,
 'nachos': 0.44141689373297005,
 'omelette': 0.47840531561461797,
 'onion_rings': 0.832618025751073,
 'oysters': 0.8340080971659919,
 '...': 0.7101200606106217}
```

```

'paella': 0.5903307888040712,
'pancakes': 0.6468253968253969,
'panna_cotta': 0.4743589743589744,
'peking_duck': 0.658008658008658,
'pho': 0.8665377176015473,
'pizza': 0.7808764940239045,
'pork_chop': 0.30875576036866365,
'poutine': 0.7603305785123966,
'prime_rib': 0.571830985915493,
'pulled_pork_sandwich': 0.5870841487279843,
'ramen': 0.6756756756756757,
'ravioli': 0.34285714285714286,
'red_velvet_cake': 0.6711409395973154,
'risotto': 0.4653465346534653,
'samosa': 0.5525525525525525,
'sashimi': 0.7783783783783783,
'scallops': 0.3975409836065574,
'seaweed_salad': 0.8130081300813008,
'shrimp_and_grits': 0.4301675977653631,
'spaghetti_bolognese': 0.7881548974943051,
'spaghetti_carbonara': 0.84765625,
'spring_rolls': 0.6652977412731006,
'steak': 0.34234234234234234,
'strawberry_shortcake': 0.5714285714285714,
'sushi': 0.6710526315789473,
'tacos': 0.3809523809523809,
'takoyaki': 0.5644916540212443,
'tiramisu': 0.3858823529411765,
'tuna_tartare': 0.35356200527704484,
>waffles': 0.641025641025641}

```

**Looking good!**

**It seems like our dictionary is ordered by the class names. However, I think if we're trying to visualize different scores, it might look nicer if they were in some kind of order.**

**How about we turn our `class_f1_scores` dictionary into a pandas DataFrame and sort it in ascending fashion?**

In [ ]:

```

# Turn f1-scores into dataframe for visualization
import pandas as pd
f1_scores = pd.DataFrame({ "class_name": list(class_f1_scores.keys()),
                           "f1-score": list(class_f1_scores.values()) }).sort_values("f1-score",
                           ascending=False)
f1_scores

```

Out[ ]:

class_name	f1-score
33	edamame 0.937143
63	macarons 0.895397
75	pho 0.866538
91	spaghetti_carbonara 0.847656
69	oysters 0.834008
...	...
56	huevos_rancheros 0.339833
22	chocolate_mousse 0.329159
77	pork_chop 0.308756
39	foie_gras 0.297491
0	apple_pie 0.240566

101 rows × 2 columns

Now we're talking! Let's finish it off with a nice horizontal bar chart.

In [ ]:

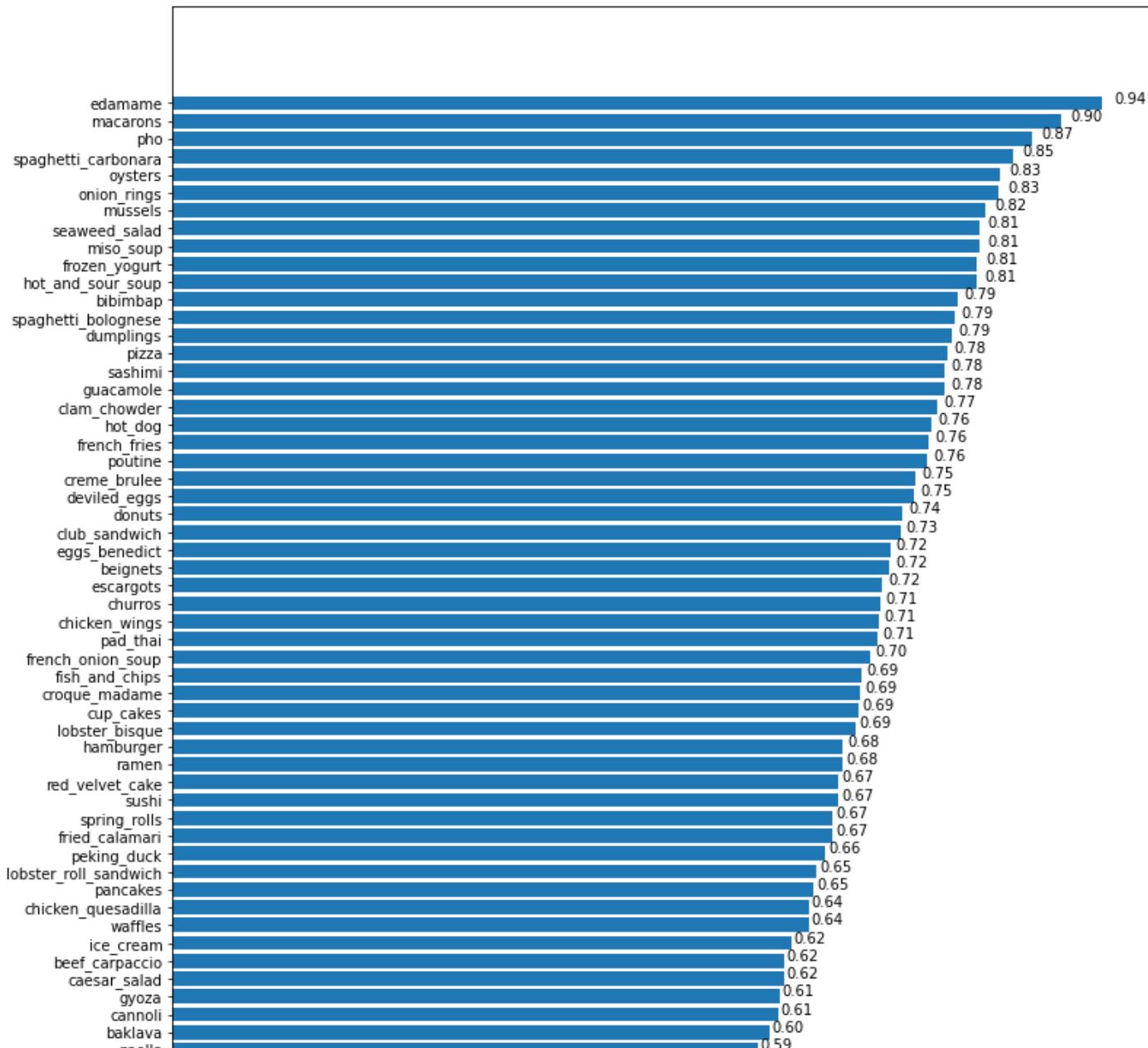
```
import matplotlib.pyplot as plt

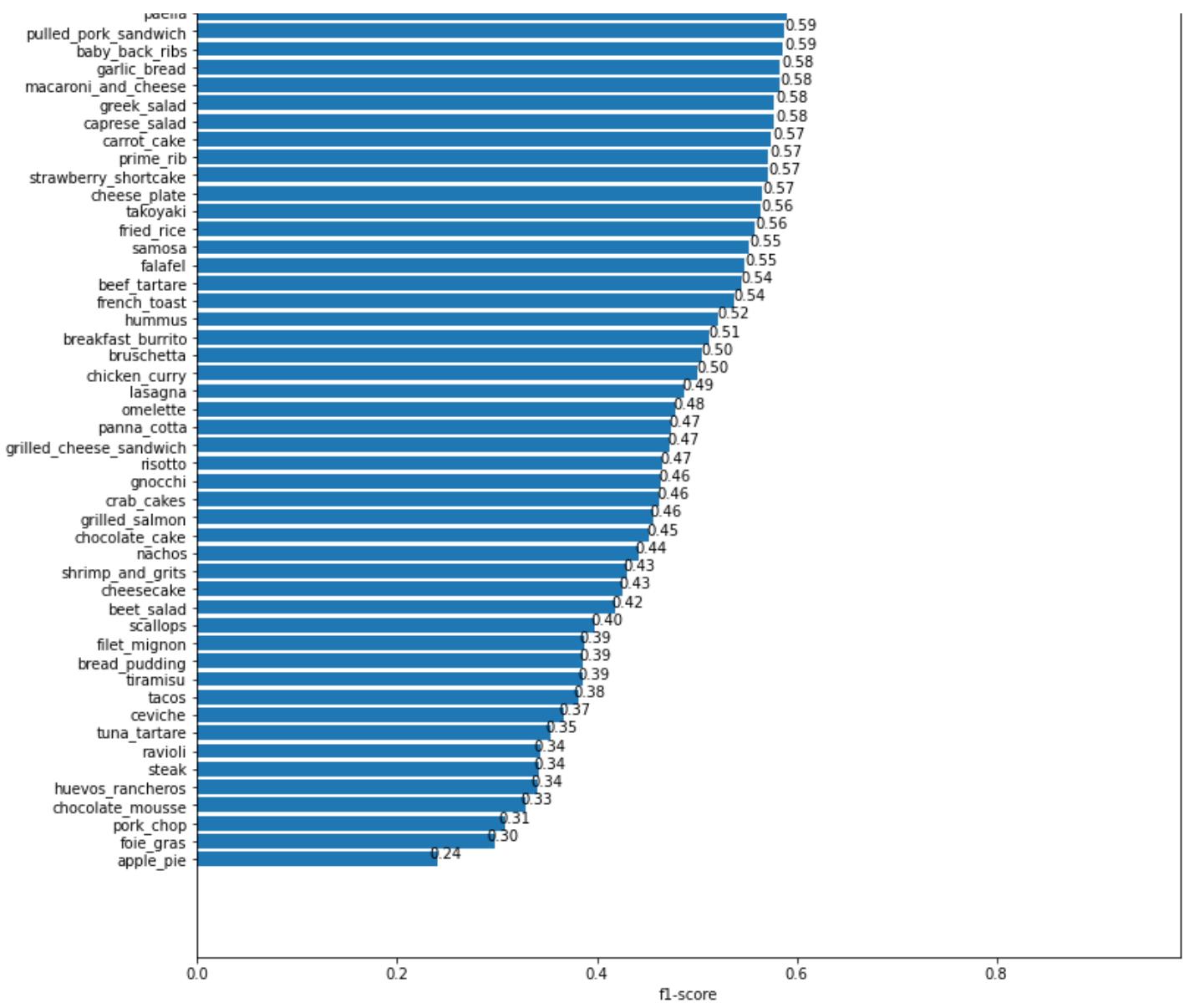
fig, ax = plt.subplots(figsize=(12, 25))
scores = ax.barh(range(len(f1_scores)), f1_scores["f1-score"].values)
ax.set_yticks(range(len(f1_scores)))
ax.set_yticklabels(list(f1_scores["class_name"]))
ax.set_xlabel("f1-score")
ax.set_title("F1-Scores for 10 Different Classes")
ax.invert_yaxis(); # reverse the order

def autolabel(rects): # Modified version of: https://matplotlib.org/examples/api/barchart_demo.html
    """
    Attach a text label above each bar displaying its height (it's value).
    """
    for rect in rects:
        width = rect.get_width()
        ax.text(1.03*width, rect.get_y() + rect.get_height()/1.5,
                f"{width:.2f}",
                ha='center', va='bottom')

autolabel(scores)
```

F1-Scores for 10 Different Classes





Now that's a good looking graph! I mean, the text positioning could be improved a little but it'll do for now.

Can you see how visualizing our model's predictions gives us a completely new insight into its performance?

A few moments ago we only had an accuracy score but now we've got an indication of how well our model is performing on a class by class basis.

It seems like our model performs fairly poorly on classes like `apple_pie` and `ravioli` while for classes like `edamame` and `pho` the performance is quite outstanding.

Findings like these give us clues into where we could go next with our experiments. Perhaps we may have to collect more data on poor performing classes or perhaps the worst performing classes are just hard to make predictions on.

**Exercise:** Visualize some of the most poor performing classes, do you notice any trends among them?

## Visualizing predictions on test images

Time for the real test. Visualizing predictions on actual images. You can look at all the metrics you want but until you've visualized some predictions, you won't really know how your model is performing.

As it stands, our model can't just predict on any image of our choice. The image first has to be loaded into a tensor.

So to begin predicting on any given image, we'll create a function to load an image into a tensor.

**Specifically, it'll:**

- **Read in a target image filepath using `tf.io.read_file()`.**
- **Turn the image into a `Tensor` using `tf.io.decode_image()`.**
- **Resize the image to be the same size as the images our model has been trained on (224 x 224) using `tf.image.resize()`.**
- **Scale the image to get all the pixel values between 0 & 1 if necessary.**

In [ ]:

```
def load_and_prep_image(filename, img_shape=224, scale=True):  
    """  
    Reads in an image from filename, turns it into a tensor and reshapes into  
(224, 224, 3).  
  
    Parameters  
    -----  
    filename (str): string filename of target image  
    img_shape (int): size to resize target image to, default 224  
    scale (bool): whether to scale pixel values to range(0, 1), default True  
    """  
  
    # Read in the image  
    img = tf.io.read_file(filename)  
    # Decode it into a tensor  
    img = tf.io.decode_image(img)  
    # Resize the image  
    img = tf.image.resize(img, [img_shape, img_shape])  
    if scale:  
        # Rescale the image (get all values between 0 and 1)  
        return img/255.  
    else:  
        return img
```

**Image loading and preprocessing function ready.**

**Now let's write some code to:**

1. **Load a few random images from the test dataset.**
2. **Make predictions on them.**
3. **Plot the original image(s) along with the model's predicted label, prediction probability and ground truth label.**

In [ ]:

```
# Make preds on a series of random images  
import os  
import random  
  
plt.figure(figsize=(17, 10))  
for i in range(3):  
    # Choose a random image from a random class  
    class_name = random.choice(class_names)  
    filename = random.choice(os.listdir(test_dir + "/" + class_name))  
    filepath = test_dir + class_name + "/" + filename  
  
    # Load the image and make predictions  
    img = load_and_prep_image(filepath, scale=False) # don't scale images for EfficientNet  
predictions  
    pred_prob = model.predict(tf.expand_dims(img, axis=0)) # model accepts tensors of shape [None, 224, 224, 3]  
    pred_class = class_names[pred_prob.argmax()] # find the predicted class  
  
    # Plot the image(s)  
    plt.subplot(1, 3, i+1)  
    plt.imshow(img/255.)  
    if class_name == pred_class: # Change the color of text based on whether prediction is  
right or wrong
```

```

        title_color = "g"
    else:
        title_color = "r"
    plt.title(f"actual: {class_name}, pred: {pred_class}, prob: {pred_prob.max():.2f}", color=title_color)
    plt.axis(False);

```

actual: lasagna, pred: croque\_madame, prob: 0.28  
actual: spaghetti\_carbonara, pred: spaghetti\_carbonara, prob: 0.72 actual: takoyaki, pred: poutine, prob: 0.50



After going through enough random samples, it starts to become clear that the model tends to make far worse predictions on classes which are visually similar such as `baby_back_ribs` getting mistaken as `steak` and vice versa.

## Finding the most wrong predictions

It's a good idea to go through at least 100+ random instances of your model's predictions to get a good feel for how it's doing.

After a while you might notice the model predicting on some images with a very high prediction probability, meaning it's very confident with its prediction but still getting the label wrong.

These **most wrong** predictions can help to give further insight into your model's performance.

So how about we write some code to collect all of the predictions where the model has output a high prediction probability for an image (e.g. 0.95+) but gotten the prediction wrong.

We'll go through the following steps:

1. Get all of the image file paths in the test dataset using the `list_files()` method.
2. Create a pandas DataFrame of the image filepaths, ground truth labels, prediction classes, max prediction probabilities, ground truth class names and predicted class names.
  - **Note:** We don't necessarily have to create a DataFrame like this but it'll help us visualize things as we go.
3. Use our DataFrame to find all the wrong predictions (where the ground truth doesn't match the prediction).
4. Sort the DataFrame based on wrong predictions and highest max prediction probabilities.
5. Visualize the images with the highest prediction probabilities but have the wrong prediction.

In [ ]:

```
# 1. Get the filenames of all of our test data
filepaths = []
for filepath in test_data.list_files("101_food_classes_10_percent/test/*/*.jpg",
                                     shuffle=False):
    filepaths.append(filepath.numpy())
filepaths[:10]
```

Out[ ]:

```
[b'101_food_classes_10_percent/test/apple_pie/1011328.jpg',
 b'101_food_classes_10_percent/test/apple_pie/101251.jpg',
 b'101_food_classes_10_percent/test/apple_pie/1034399.jpg',
 b'101_food_classes_10_percent/test/apple_pie/103801.jpg',
 b'101_food_classes_10_percent/test/apple_pie/1038694.jpg',
 b'101_food_classes_10_percent/test/apple_pie/1047447.jpg',
```

```
b'101_food_classes_10_percent/test/apple_pie/1068632.jpg',
b'101_food_classes_10_percent/test/apple_pie/110043.jpg',
b'101_food_classes_10_percent/test/apple_pie/1106961.jpg',
b'101_food_classes_10_percent/test/apple_pie/1113017.jpg']
```

Now we've got all of the test image filepaths, let's combine them into a DataFrame along with:

- Their ground truth labels (`y_labels`).
- The class the model predicted (`pred_classes`).
- The maximum prediction probability value (`pred_probs.max(axis=1)`).
- The ground truth class names.
- The predicted class names.

In [ ]:

```
# 2. Create a dataframe out of current prediction data for analysis
import pandas as pd
pred_df = pd.DataFrame({ "img_path": filepaths,
                         "y_true": y_labels,
                         "y_pred": pred_classes,
                         "pred_conf": pred_probs.max(axis=1), # get the maximum prediction probability value
                         "y_true_classname": [class_names[i] for i in y_labels],
                         "y_pred_classname": [class_names[i] for i in pred_classes] })
pred_df.head()
```

Out [ ]:

	img_path	y_true	y_pred	pred_conf	y_true_classname	y_pred_classname
0	b'101_food_classes_10_percent/test/apple_pie/1...	0	52	0.847419	apple_pie	gyoza
1	b'101_food_classes_10_percent/test/apple_pie/1...	0	0	0.964017	apple_pie	apple_pie
2	b'101_food_classes_10_percent/test/apple_pie/1...	0	0	0.959259	apple_pie	apple_pie
3	b'101_food_classes_10_percent/test/apple_pie/1...	0	80	0.658607	apple_pie	pulled_pork_sandwich
4	b'101_food_classes_10_percent/test/apple_pie/1...	0	79	0.367901	apple_pie	prime_rib

Nice! How about we make a simple column telling us whether or not the prediction is right or wrong?

In [ ]:

```
# 3. Is the prediction correct?
pred_df["pred_correct"] = pred_df["y_true"] == pred_df["y_pred"]
pred_df.head()
```

Out [ ]:

	img_path	y_true	y_pred	pred_conf	y_true_classname	y_pred_classname	pred_correct
0	b'101_food_classes_10_percent/test/apple_pie/1...	0	52	0.847419	apple_pie	gyoza	False
1	b'101_food_classes_10_percent/test/apple_pie/1...	0	0	0.964017	apple_pie	apple_pie	True
2	b'101_food_classes_10_percent/test/apple_pie/1...	0	0	0.959259	apple_pie	apple_pie	True
3	b'101_food_classes_10_percent/test/apple_pie/1...	0	80	0.658607	apple_pie	pulled_pork_sandwich	False
4	b'101_food_classes_10_percent/test/apple_pie/1...	0	79	0.367901	apple_pie	prime_rib	False

And now since we know which predictions were right or wrong and along with their prediction probabilities, how about we get the 100 "most wrong" predictions by sorting for wrong predictions and descending prediction probabilities?

In [ ]:

```
# 4. Get the top 100 wrong examples
```

```
top_100_wrong = pred_df[pred_df["pred_correct"] == False].sort_values("pred_conf", ascending=False) [:100]
top_100_wrong.head(20)
```

Out [ ]:

		img_path	y_true	y_pred	pred_conf	y_true_classname	y_pred_classname
21810	b'101_food_classes_10_percent/test/scallops/17...		87	29	0.999997	scallops	cup_cakes
231	b'101_food_classes_10_percent/test/apple_pie/8...		0	100	0.999995	apple_pie	waffles
15359	b'101_food_classes_10_percent/test/lobster_rol...		61	53	0.999988	lobster_roll_sandwich	hamburgers
23539	b'101_food_classes_10_percent/test/strawberry_...		94	83	0.999987	strawberry_shortcake	red_velvet_cake
21400	b'101_food_classes_10_percent/test/samosa/3140...		85	92	0.999981	samosa	spring_rolls
24540	b'101_food_classes_10_percent/test/tiramisu/16...		98	83	0.999947	tiramisu	red_velvet_cake
2511	b'101_food_classes_10_percent/test/bruschetta/...		10	61	0.999945	bruschetta	lobster_roll_sandwich
5574	b'101_food_classes_10_percent/test/chocolate_m...		22	21	0.999939	chocolate_mousse	chocolate_cake
17855	b'101_food_classes_10_percent/test/paella/2314...		71	65	0.999931	paella	mussels
23797	b'101_food_classes_10_percent/test/sushi/16593...		95	86	0.999904	sushi	sashimi
18001	b'101_food_classes_10_percent/test/pancakes/10...		72	67	0.999904	pancakes	omelets
11642	b'101_food_classes_10_percent/test/garlic_brea...		46	10	0.999877	garlic_bread	bruschetta
10847	b'101_food_classes_10_percent/test/fried_calam...		43	68	0.999872	fried_calamari	onion_rings
23631	b'101_food_classes_10_percent/test/strawberry_...		94	83	0.999858	strawberry_shortcake	red_velvet_cake
1155	b'101_food_classes_10_percent/test/beef_tartare...		4	5	0.999858	beef_tartare	beet_salad
10854	b'101_food_classes_10_percent/test/fried_calam...		43	68	0.999854	fried_calamari	onion_rings
23904	b'101_food_classes_10_percent/test/sushi/33652...		95	86	0.999823	sushi	sashimi
7316	b'101_food_classes_10_percent/test/cup_cakes/1...		29	83	0.999816	cup_cakes	red_velvet_cake
13144	b'101_food_classes_10_percent/test/gyoza/31214...		52	92	0.999799	gyoza	spring_rolls
10880	b'101_food_classes_10_percent/test/fried_calam...		43	68	0.999778	fried_calamari	onion_rings

Very interesting... just by comparing the ground truth classname (`y_true_classname`) and the prediction classname column (`y_pred_classname`), do you notice any trends?

It might be easier if we visualize them.

In [ ]:

```
# 5. Visualize some of the most wrong examples
images_to_view = 9
start_index = 10 # change the start index to view more
plt.figure(figsize=(15, 10))
for i, row in enumerate(top_100_wrong[start_index:start_index+images_to_view].itertuples()):
    plt.subplot(3, 3, i+1)
    img = load_and_prep_image(row[1], scale=True)
    _, _, _, _, pred_prob, y_true, y_pred, _ = row # only interested in a few parameters of each row
    plt.imshow(img)
    plt.title(f"actual: {y_true}, pred: {y_pred} \nprob: {pred_prob:.2f}")
    plt.axis(False)
```

actual: pancakes, pred: omelette  
prob: 1.00



actual: garlic\_bread, pred: bruschetta  
prob: 1.00



actual: fried\_calamari, pred: onion\_rings  
prob: 1.00





actual: strawberry\_shortcake, pred: red\_velvet\_cake  
prob: 1.00



actual: sushi, pred: sashimi  
prob: 1.00



actual: beef\_tartare, pred: beet\_salad  
prob: 1.00



actual: cup\_cakes, pred: red\_velvet\_cake  
prob: 1.00



actual: fried\_calamari, pred: onion\_rings  
prob: 1.00



actual: gyoza, pred: spring\_rolls  
prob: 1.00



Going through the model's most wrong predictions can usually help figure out a couple of things:

- **Some of the labels might be wrong** - If our model ends up being good enough, it may actually learn to predict very well on certain classes. This means some images which the model predicts the right label may show up as wrong if the ground truth label is wrong. If this is the case, we can often use our model to help us improve the labels in our dataset(s) and in turn, potentially making future models better. This process of using the model to help improve labels is often referred to as [active learning](#).
- **Could more samples be collected?** - If there's a recurring pattern for a certain class being poorly predicted on, perhaps it's a good idea to collect more samples of that particular class in different scenarios to improve further models.

## Test out the big dog model on test images as well as custom images of food

So far we've visualized some of our model's predictions from the test dataset but it's time for the real test: using our model to make predictions on our own custom images of food.

For this you might want to upload your own images to Google Colab or by putting them in a folder you can load into the notebook.

In my case, I've prepared my own small dataset of six or so images of various foods.

Let's download them and unzip them.

In [ ]:

```
# Download some custom images from Google Storage
# Note: you can upload your own custom images to Google Colab using the "upload" button in the Files tab
!wget https://storage.googleapis.com/ztm_tf_course/food_vision/custom_food_images.zip

unzip_data("custom_food_images.zip")

--2021-02-23 06:10:55-- https://storage.googleapis.com/ztm_tf_course/food_vision/custom_food_images.zip
Resolving storage.googleapis.com (storage.googleapis.com)... 172.217.7.176, 172.217.9.208, 172.217.2.112, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|172.217.7.176|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 13192985 (13M) [application/zip]
Saving to: 'custom_food_images.zip'
```

```
custom_food_images. 100%[=====] 12.58M --.-KB/s in 0.1s  
2021-02-23 06:10:55 (88.4 MB/s) - 'custom_food_images.zip' saved [13192985/13192985]
```

**Wonderful, we can load these in and turn them into tensors using our `load_and_prep_image()` function but first we need a list of image filepaths.**

In [ ]:

```
# Get custom food images filepaths  
custom_food_images = ["custom_food_images/" + img_path for img_path in os.listdir("custom_food_images")]  
custom_food_images
```

Out[ ]:

```
['custom_food_images/hamburger.jpeg',  
'custom_food_images/sushi.jpeg',  
'custom_food_images/ramen.jpeg',  
'custom_food_images/chicken_wings.jpeg',  
'custom_food_images/pizza-dad.jpeg',  
'custom_food_images/steak.jpeg']
```

**Now we can use similar code to what we used previously to load in our images, make a prediction on each using our trained model and then plot the image along with the predicted class.**

In [ ]:

```
# Make predictions on custom food images  
for img in custom_food_images:  
    img = load_and_prep_image(img, scale=False) # load in target image and turn it into tensor  
    pred_prob = model.predict(tf.expand_dims(img, axis=0)) # make prediction on image with shape [None, 224, 224, 3]  
    pred_class = class_names[pred_prob.argmax()] # find the predicted class label  
    # Plot the image with appropriate annotations  
    plt.figure()  
    plt.imshow(img/255.) # imshow() requires float inputs to be normalized  
    plt.title(f"pred: {pred_class}, prob: {pred_prob.max():.2f}")  
    plt.axis(False)
```

pred: hamburger, prob: 1.00



pred: sushi, prob: 0.73



pred: bibimbap, prob: 0.50



pred: chicken\_wings, prob: 1.00



pred: pizza, prob: 0.99



pred: filet\_mignon, prob: 0.61



Two thumbs up! How cool is that?! Our Food Vision model has come to life!

Seeing a machine learning model work on a premade test dataset is cool but seeing it work on your own data is mind blowing.

And guess what... our model got these incredible results (10%+ better than the baseline) with only 10% of the training images

I wonder what would happen if we trained a model with all of the data (100% of the training data from Food101 instead of 10%)? Hint: that's your task in the next notebook.

## Exercises

1. Take 3 of your own photos of food and use the trained model to make predictions on them, share your predictions with the other students in Discord and show off your Food Vision model 😊.
2. Train a feature-extraction transfer learning model for 10 epochs on the same data and compare its performance versus a model which used feature extraction for 5 epochs and fine-tuning for 5 epochs (like we've used in this notebook). Which method is better?
3. Recreate our first model (the feature extraction model) with `mixed_precision` turned on.
  - Does it make the model train faster?
  - Does it effect the accuracy or performance of our model?
  - What's the advantages of using `mixed_precision` training?

## Extra-curriculum

- Spend 15-minutes reading up on the [EarlyStopping callback](#). What does it do? How could we use it in our model training?
- Spend an hour reading about [Streamlit](#). What does it do? How might you integrate some of the things we've done in this notebook in a Streamlit app?

## 07. Milestone Project 1: Food Vision Big™

In the previous notebook ([transfer learning part 3: scaling up](#)) we built Food Vision mini: a transfer learning model which beat the original results of the [Food101 paper](#) with only 10% of the data.

But you might be wondering, what would happen if we used all the data?

Well, that's what we're going to find out in this notebook!

We're going to be building Food Vision Big™, using all of the data from the Food101 dataset.

Yep. All 75,750 training images and 25,250 testing images.

And guess what...

This time we've got the goal of beating [DeepFood](#), a 2016 paper which used a Convolutional Neural Network trained for 2-3 days to achieve 77.4% top-1 accuracy.

**Note:** Top-1 accuracy means "accuracy for the top softmax activation value output by the model" (because softmax outputs a value for every class, but top-1 means only the highest one is evaluated). Top-5 accuracy means "accuracy for the top 5 softmax activation values output by the model", in other words, did the true label appear in the top 5 activation values? Top-5 accuracy scores are usually noticeably higher than top-1.

	Food Vision Big™	Food Vision mini
Dataset source	TensorFlow Datasets	Preprocessed download from Kaggle
Train data	75,750 images	7,575 images
Test data	25,250 images	25,250 images
Mixed precision	Yes	No
Data loading	Perfomanant tf.data API	TensorFlow pre-built function
Target results	77.4% top-1 accuracy (beat <a href="#">DeepFood paper</a> )	50.76% top-1 accuracy (beat <a href="#">Food101 paper</a> )

*Table comparing difference between Food Vision Big (this notebook) versus Food Vision mini (previous notebook).*

Alongside attempting to beat the DeepFood paper, we're going to learn about two methods to significantly improve the speed of our model training:

1. Prefetching
2. Mixed precision training

But more on these later.

## What we're going to cover

- Using TensorFlow Datasets to download and explore data
- Creating preprocessing function for our data
- Batching & preparing datasets for modelling (making our datasets run fast)
- Creating modelling callbacks
- Setting up mixed precision training
- Building a feature extraction model (see [transfer learning part 1: feature extraction](#))
- Fine-tuning the feature extraction model (see [transfer learning part 2: fine-tuning](#))
- Viewing training results on TensorBoard

# How you should approach this notebook

You can read through the descriptions and the code (it should all run, except for the cells which error on purpose), but there's a better option.

Write all of the code yourself.

Yes. I'm serious. Create a new notebook, and rewrite each line by yourself. Investigate it, see if you can break it, why does it break?

You don't have to write the text descriptions but writing the code yourself is a great way to get hands-on experience.

Don't worry if you make mistakes, we all do. The way to get better and make less mistakes is to write more code.

▀ Resource: See the full set of course materials on GitHub:

<https://github.com/mrdbourke/tensorflow-deep-learning>

## Check GPU

For this notebook, we're going to be doing something different.

We're going to be using mixed precision training.

Mixed precision training was introduced in [TensorFlow 2.4.0](#) (a very new feature at the time of writing).

What does mixed precision training do?

Mixed precision training uses a combination of single precision (float32) and half-precision (float16) data types to speed up model training (up 3x on modern GPUs).

We'll talk about this more later on but in the meantime you can read the [TensorFlow documentation on mixed precision](#) for more details.

For now, before we can move forward if we want to use mixed precision training, we need to make sure the GPU powering our Google Colab instance (if you're using Google Colab) is compatible.

For mixed precision training to work, you need access to a GPU with a compute compatibility score of 7.0+.

Google Colab offers P100, K80 and T4 GPUs, however, the P100 and K80 aren't compatible with mixed precision training.

Therefore before we proceed we need to make sure we have access to a Tesla T4 GPU in our Google Colab instance.

If you're not using Google Colab, you can find a list of various [Nvidia GPU compute capabilities on Nvidia's developer website](#).

▀ Note: If you run the cell below and see a P100 or K80, try going to Runtime -> Factory Reset Runtime (note: this will remove any saved variables and data from your Colab instance) and then retry to get a T4.

In [ ]:

```
# If using Google Colab, this should output "Tesla T4" otherwise,  
# you won't be able to use mixed precision training  
!nvidia-smi -L
```

GPU 0: Tesla T4 (UUID: GPU-60ebc9b4-1cbc-ed62-9d0f-6e8ba24db56b)

Since mixed precision training was introduced in TensorFlow 2.4.0, make sure you've got at least TensorFlow 2.4.0+.

In [ ]:

```
# Check TensorFlow version (should be 2.4.0+)
import tensorflow as tf
print(tf.__version__)
```

2.4.1

## Get helper functions

We've created a series of helper functions throughout the previous notebooks in the course. Instead of rewriting them (tedious), we'll import the `helper_functions.py` file from the GitHub repo.

In [ ]:

```
# Get helper functions file
!wget https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/extras/helper_functions.py

--2021-03-22 03:22:37-- https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/extras/helper_functions.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com) ... 185.199.111.133, 185.199.108.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.111.133|:443...
.. connected.
HTTP request sent, awaiting response... 200 OK
Length: 9304 (9.1K) [text/plain]
Saving to: 'helper_functions.py'

helper_functions.py 100%[=====] 9.09K --.-KB/s in 0s

2021-03-22 03:22:37 (111 MB/s) - 'helper_functions.py' saved [9304/9304]
```

In [ ]:

```
# Import series of helper functions for the notebook (we've created/used these in previous notebooks)
from helper_functions import create_tensorboard_callback, plot_loss_curves, compare_histories
```

## Use TensorFlow Datasets to Download Data

In previous notebooks, we've downloaded our food images (from the [Food101 dataset](#)) from Google Storage.

And this is a typical workflow you'd use if you're working on your own datasets.

However, there's another way to get datasets ready to use with TensorFlow.

For many of the most popular datasets in the machine learning world (often referred to and used as benchmarks), you can access them through [TensorFlow Datasets \(TFDS\)](#).

**What is TensorFlow Datasets?**

A place for prepared and ready-to-use machine learning datasets.

**Why use TensorFlow Datasets?**

- Load data already in Tensors
- Practice on well established datasets
- Experiment with different data loading techniques (like we're going to use in this notebook)
- Experiment with new TensorFlow features quickly (such as mixed precision training)

**Why *not* use TensorFlow Datasets?**

- The datasets are static (they don't change, like your real-world datasets would)

- Might not be suited for your particular problem (but great for experimenting)

To begin using TensorFlow Datasets we can import it under the alias `tfds`.

In [ ]:

```
# Get TensorFlow Datasets
import tensorflow_datasets as tfds
```

To find all of the available datasets in TensorFlow Datasets, you can use the `list_builders()` method.

After doing so, we can check to see if the one we're after (`"food101"`) is present.

In [ ]:

```
# List available datasets
datasets_list = tfds.list_builders() # get all available datasets in TFDS
print("food101" in datasets_list) # is the dataset we're after available?
```

True

Beautiful! It looks like the dataset we're after is available (note there are plenty more available but we're on Food101).

To get access to the Food101 dataset from the TFDS, we can use the `tfds.load()` method.

In particular, we'll have to pass it a few parameters to let it know what we're after:

- `name (str)` : the target dataset (e.g. `"food101"`)
- `split (list, optional)` : what splits of the dataset we're after (e.g. `["train", "validation"]`)
  - the `split` parameter is quite tricky. See [the documentation for more](#).
- `shuffle_files (bool)` : whether or not to shuffle the files on download, defaults to `False`
- `as_supervised (bool)` : `True` to download data samples in tuple format (`(data, label)`) or `False` for dictionary format
- `with_info (bool)` : `True` to download dataset metadata (labels, number of samples, etc)

**Note:** Calling the `tfds.load()` method will start to download a target dataset to disk if the `download=True` parameter is set (default). This dataset could be 100GB+, so make sure you have space.

In [ ]:

```
# Load in the data (takes about 5-6 minutes in Google Colab)
(train_data, test_data), ds_info = tfds.load(name="food101", # target dataset to get from
                                             TFDS
                                             split=["train", "validation"], # what splits of data should we get? note: not all datasets have train, valid, test
                                             shuffle_files=True, # shuffle files on download?
                                             as_supervised=True, # download data in tuple format (sample, label), e.g. (image, label)
                                             with_info=True) # include dataset metadata?
if so, tfds.load() returns tuple (data, ds_info)
```

Downloading and preparing dataset food101/2.0.0 (download: 4.65 GiB, generated: Unknown size, total: 4.65 GiB) to /root/tensorflow\_datasets/food101/2.0.0...

```
Shuffling and writing examples to /root/tensorflow_datasets/food101/2.0.0.incompleteL6Zw0
E/food101-train.tfrecord
```

```
Shuffling and writing examples to /root/tensorflow_datasets/food101/2.0.0.incompleteL6Zw0
E/food101-validation.tfrecord
```

Dataset food101 downloaded and prepared to /root/tensorflow\_datasets/food101/2.0.0. Subsequent calls will reuse this data.

**Wonderful! After a few minutes of downloading, we've now got access to entire Food101 dataset (in tensor format) ready for modelling.**

Now let's get a little information from our dataset, starting with the class names.

Getting class names from a TensorFlow Datasets dataset requires downloading the "dataset\_info" variable (by using the `as_supervised=True` parameter in the `tfds.load()` method, note: this will only work for supervised datasets in TFDS).

We can access the class names of a particular dataset using the `dataset_info.features` attribute and accessing `names` attribute of the the "label" key.

In [ ]:

```
# Features of Food101 TFDS
ds_info.features
```

Out[ ]:

```
FeaturesDict({
    'image': Image(shape=(None, None, 3), dtype=tf.uint8),
    'label': ClassLabel(shape=(), dtype=tf.int64, num_classes=101),
})
```

In [ ]:

```
# Get class names
class_names = ds_info.features["label"].names
class_names[:10]
```

Out[ ]:

```
['apple_pie',
 'baby_back_ribs',
 'baklava',
 'beef_carpaccio',
 'beef_tartare',
 'beet_salad',
 'beignets',
 'bibimbap',
 'bread_pudding',
 'breakfast_burrito']
```

## Exploring the Food101 data from TensorFlow Datasets

Now we've downloaded the Food101 dataset from TensorFlow Datasets, how about we do what any good data explorer should?

In other words, "visualize, visualize, visualize".

Let's find out a few details about our dataset:

- The shape of our input data (image tensors)
- The datatype of our input data
- What the labels of our input data look like (e.g. one-hot encoded versus label-encoded)
- Do the labels match up with the class names?

To do, let's take one sample off the training data (using the `.take()` method) and explore it.

In [ ]:

```
# Take one sample off the training data
train_one_sample = train_data.take(1) # samples are in format (image_tensor, label)
```

**Because we used the `as_supervised=True` parameter in our `tfds.load()` method above, data samples come in the tuple format structure `(data, label)` or in our case `(image_tensor, label)`.**

In [ ]:

```
# What does one sample of our training data look like?
train_one_sample
```

Out[ ]:

```
<TakeDataset shapes: ((None, None, 3), ()), types: (tf.uint8, tf.int64)>
```

**Let's loop through our single training sample and get some info from the `image_tensor` and `label`.**

In [ ]:

```
# Output info about our training sample
for image, label in train_one_sample:
    print(f"""
        Image shape: {image.shape}
        Image dtype: {image.dtype}
        Target class from Food101 (tensor form): {label}
        Class name (str form): {class_names[label.numpy()]}
    """)
```

```
Image shape: (512, 512, 3)
Image dtype: <dtype: 'uint8'>
Target class from Food101 (tensor form): 25
Class name (str form): club_sandwich
```

**Because we set the `shuffle_files=True` parameter in our `tfds.load()` method above, running the cell above a few times will give a different result each time.**

**Checking these you might notice some of the images have different shapes, for example `(512, 342, 3)` and `(512, 512, 3)` (height, width, color\_channels).**

**Let's see what one of the image tensors from TFDS's Food101 dataset looks like.**

In [ ]:

```
# What does an image tensor from TFDS's Food101 look like?
image
```

Out[ ]:

```
<tf.Tensor: shape=(512, 512, 3), dtype=uint8, numpy=
array([[[135, 156, 175],
       [125, 148, 166],
       [114, 136, 159],
       ...,
       [ 26,    5,   12],
       [ 26,    3,   11],
       [ 27,    4,   12]],

      [[128, 150, 171],
       [115, 140, 160],
       [102, 127, 149],
       ...,
       [ 28,    7,   14],
       [ 29,    6,   14],
       [ 30,    7,   15]],
```

```
[[112, 139, 160],  
 [ 99, 127, 148],  
 [ 87, 115, 137],  
 ...,  
 [ 29,    6,   16],  
 [ 31,    5,   16],  
 [ 32,    6,   17]],  
  
...,  
  
[[ 48,   47,   53],  
 [ 53,   52,   58],  
 [ 52,   51,   59],  
 ...,  
 [111,   99,   99],  
 [108,   98,   97],  
 [106,   96,   97]],  
  
[[ 44,   45,   47],  
 [ 48,   49,   51],  
 [ 46,   47,   51],  
 ...,  
 [108,   96,   98],  
 [105,   94,   98],  
 [102,   93,   96]],  
  
[[ 40,   42,   41],  
 [ 45,   47,   46],  
 [ 44,   45,   49],  
 ...,  
 [105,   95,   96],  
 [104,   93,   99],  
 [100,   91,   96]]], dtype=uint8)>
```

In [ ]:

```
# What are the min and max values?  
tf.reduce_min(image), tf.reduce_max(image)
```

Out[ ]:

```
(<tf.Tensor: shape=(), dtype=uint8, numpy=0>,  
<tf.Tensor: shape=(), dtype=uint8, numpy=255>)
```

Alright looks like our image tensors have values of between 0 & 255 (standard red, green, blue colour values) and the values are of data type `unit8`.

We might have to preprocess these before passing them to a neural network. But we'll handle this later.

In the meantime, let's see if we can plot an image sample.

## Plot an image from TensorFlow Datasets

We've seen our image tensors in tensor format, now let's really adhere to our motto.

"Visualize, visualize, visualize!"

Let's plot one of the image samples using `matplotlib.pyplot.imshow()` and set the title to target class name.

In [ ]:

```
# Plot an image tensor  
import matplotlib.pyplot as plt  
plt.imshow(image)  
plt.title(class_names[label.numpy()]) # add title to image by indexing on class_names lis  
t  
plt.axis(False);
```

dub\_sandwich



Delicious!

Okay, looks like the Food101 data we've got from TFDS is similar to the datasets we've been using in previous notebooks.

Now let's preprocess it and get it ready for use with a neural network.

## Create preprocessing functions for our data

In previous notebooks, when our images were in folder format we used the method

`tf.keras.preprocessing.image_dataset_from_directory()` to load them in.

Doing this meant our data was loaded into a format ready to be used with our models.

However, since we've downloaded the data from TensorFlow Datasets, there are a couple of preprocessing steps we have to take before it's ready to model.

More specifically, our data is currently:

- In `uint8` data type
- Comprised of all different sized tensors (different sized images)
- Not scaled (the pixel values are between 0 & 255)

Whereas, models like data to be:

- In `float32` data type
- Have all of the same size tensors (batches require all tensors have the same shape, e.g. `(224, 224, 3)`)
- Scaled (values between 0 & 1), also called normalized

To take care of these, we'll create a `preprocess_img()` function which:

- Resizes an input image tensor to a specified size using `tf.image.resize()`
- Converts an input image tensor's current datatype to `tf.float32` using `tf.cast()`

**Note:** Pretrained EfficientNetBX models in `tf.keras.applications.efficientnet` (what we're going to be using) have rescaling built-in. But for many other model architectures you'll want to rescale your data (e.g. get its values between 0 & 1). This could be incorporated inside your `"preprocess_img()"` function (like the one below) or within your model as a `tf.keras.layers.experimental.preprocessing.Rescaling` layer.

In [ ]:

```
# Make a function for preprocessing images
def preprocess_img(image, label, img_shape=224):
    """
    Converts image datatype from 'uint8' -> 'float32' and reshapes image to
    [img_shape, img_shape, color_channels]
    """

```

```
image = tf.image.resize(image, [img_shape, img_shape]) # reshape to img_shape
return tf.cast(image, tf.float32), label # return (float32_image, label) tuple
```

Our `preprocess_img()` function above takes image and label as input (even though it does nothing to the label) because our dataset is currently in the tuple structure `(image, label)`.

Let's try our function out on a target image.

In [ ]:

```
# Preprocess a single sample image and check the outputs
preprocessed_img = preprocess_img(image, label)[0]
print(f"Image before preprocessing:\n {image[:2]}..., \nShape: {image.shape}, \nDatatype: {image.dtype}\n")
print(f"Image after preprocessing:\n {preprocessed_img[:2]}..., \nShape: {preprocessed_img.shape}, \nDatatype: {preprocessed_img.dtype}")
```

Image before preprocessing:

```
[[[135 156 175]
 [125 148 166]
 [114 136 159]

 ...
 [ 26   5   12]
 [ 26   3   11]
 [ 27   4   12]]]
```

```
[[128 150 171]
 [115 140 160]
 [102 127 149]
```

```
...
 [ 28   7   14]
 [ 29   6   14]
 [ 30   7   15]]...,
```

Shape: (512, 512, 3),  
Datatype: <dtype: 'uint8'>

Image after preprocessing:

```
[[[122.83163 146.17346 165.81633 ]
 [ 95.07653 122.122444 144.47958 ]
 [ 72.5051   106.994896 134.34694 ]
```

```
...
 [ 20.714308   2.3570995   3.9285717]
 [ 27.285715   6.285714    13.285714 ]
 [ 28.28575   5.2857494  13.285749 ]]
```

```
[[ 88.65305 119.41326 140.41327 ]
 [ 74.59694 108.30102 133.02042 ]
 [ 75.2551   112.57143 141.91325 ]
```

```
...
 [ 26.857143   6.285671   11.040798 ]
 [ 30.061235   6.86222   16.795908 ]
 [ 31.688843   5.688843   16.688843 ]]...,
```

Shape: (224, 224, 3),  
Datatype: <dtype: 'float32'>

Excellent! Looks like our `preprocess_img()` function is working as expected.

The input image gets converted from `uint8` to `float32` and gets reshaped from its current shape to `(224, 224, 3)`.

How does it look?

In [ ]:

```
# We can still plot our preprocessed image as long as we
# divide by 255 (for matplotlib compatibility)
plt.imshow(preprocessed_img/255.)
plt.title(class_names[label])
plt.axis(False);
```

club\_sandwich



All this food visualization is making me hungry. How about we start preparing to model it?

## Batch & prepare datasets

Before we can model our data, we have to turn it into batches.

Why?

Because computing on batches is memory efficient.

We turn our data from 101,000 image tensors and labels (train and test combined) into batches of 32 image and label pairs, thus enabling it to fit into the memory of our GPU.

To do this in effective way, we're going to be leveraging a number of methods from the [tf.data API](#).

**I Resource:** For loading data in the most performant way possible, see the TensorFlow documentation on [Better performance with the tf.data API](#).

Specifically, we're going to be using:

- `map()` - maps a predefined function to a target dataset (e.g. `preprocess_img()` to our image tensors)
- `shuffle()` - randomly shuffles the elements of a target dataset up `buffer_size` (ideally, the `buffer_size` is equal to the size of the dataset, however, this may have implications on memory)
- `batch()` - turns elements of a target dataset into batches (size defined by parameter `batch_size`)
- `prefetch()` - prepares subsequent batches of data whilst other batches of data are being computed on (improves data loading speed but costs memory)
- Extra: `cache()` - caches (saves them for later) elements in a target dataset, saving loading time (will only work if your dataset is small enough to fit in memory, standard Colab instances only have 12GB of memory)

Things to note:

- Can't batch tensors of different shapes (e.g. different image sizes, need to reshape images first, hence our `preprocess_img()` function)
- `shuffle()` keeps a buffer of the number you pass it images shuffled, ideally this number would be all of the samples in your training set, however, if your training set is large, this buffer might not fit in memory (a fairly large number like 1000 or 10000 is usually suffice for shuffling)
- For methods with the `num_parallel_calls` parameter available (such as `map()`), setting it to `num_parallel_calls=tf.data.AUTOTUNE` will parallelize preprocessing and significantly improve speed
- Can't use `cache()` unless your dataset can fit in memory

Woah, the above is alot. But once we've coded below, it'll start to make sense.

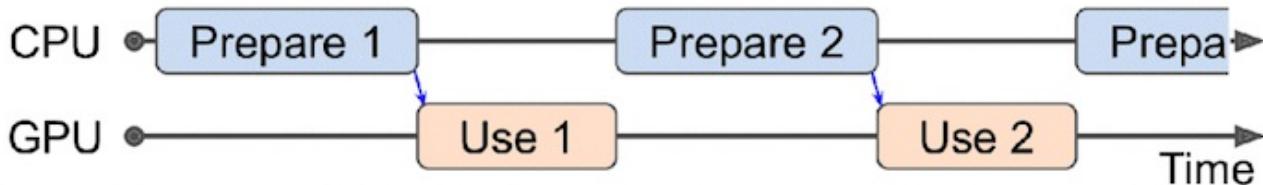
We're going to through things in the following order:

Original dataset (e.g. `train_data`) -> `map()` -> `shuffle()` -> `batch()` -> `prefetch()`  
-> `PrefetchDataset`

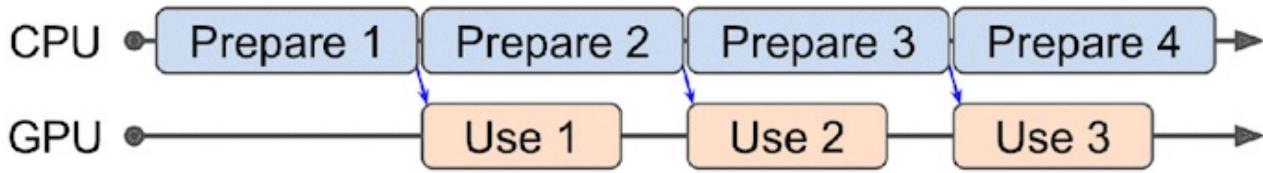
This is like saying,

"Hey, map this preprocessing function across our training dataset, then shuffle a number of elements before batching them together and make sure you prepare new batches (prefetch) whilst the model is looking through the current batch".

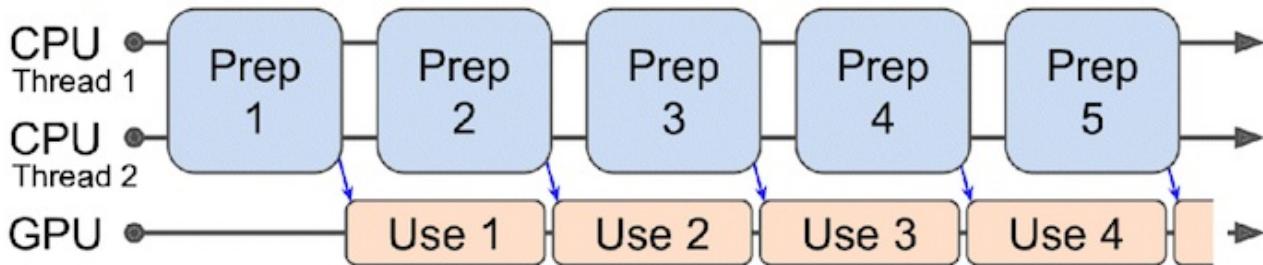
### Without prefetching



### With prefetching



### With prefetching + multithreaded loading & preprocessing



*What happens when you use prefetching (faster) versus what happens when you don't use prefetching (slower).*

**Source:** Page 422 of [Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow Book by Aurélien Géron](#).

In [ ]:

```
# Map preprocessing function to training data (and parallelize)
train_data = train_data.map(map_func=preprocess_img, num_parallel_calls=tf.data.AUTOTUNE)
# Shuffle train_data and turn it into batches and prefetch it (load it faster)
train_data = train_data.shuffle(buffer_size=1000).batch(batch_size=32).prefetch(buffer_size=tf.data.AUTOTUNE)

# Map preprocessing function to test data
test_data = test_data.map(preprocess_img, num_parallel_calls=tf.data.AUTOTUNE)
# Turn test data into batches (don't need to shuffle)
test_data = test_data.batch(32).prefetch(tf.data.AUTOTUNE)
```

And now let's check out what our prepared datasets look like.

In [ ]:

```
train_data, test_data
```

Out[ ]:

```
(<PrefetchDataset shapes: ((None, 224, 224, 3), (None,)), types: (tf.float32, tf.int64)>,
 <PrefetchDataset shapes: ((None, 224, 224, 3), (None,)), types: (tf.float32, tf.int64)>)
```

Excellent! Looks like our data is now in tuples of `(image, label)` with datatypes of `(tf.float32, tf.int64)`, just what our model is after.

**Note:** You can get away without calling the `prefetch()` method on the end of your datasets, however, you'd probably see significantly slower data loading speeds when building a model. So most of your dataset input pipelines should end with a call to `prefecth()`.

Onward.

## Create modelling callbacks

Since we're going to be training on a large amount of data and training could take a long time, it's a good idea to set up some modelling callbacks so we be sure of things like our model's training logs being tracked and our model being checkpointed (saved) after various training milestones.

To do each of these we'll use the following callbacks:

- `tf.keras.callbacks.TensorBoard()` - allows us to keep track of our model's training history so we can inspect it later (**note:** we've created this callback before have imported it from `helper_functions.py` as `create_tensorboard_callback()`)
- `tf.keras.callbacks.ModelCheckpoint()` - saves our model's progress at various intervals so we can load it and reuse it later without having to retrain it
  - Checkpointing is also helpful so we can start fine-tuning our model at a particular epoch and revert back to a previous state if fine-tuning offers no benefits

In [ ]:

```
# Create TensorBoard callback (already have "create_tensorboard_callback()" from a previous notebook)
from helper_functions import create_tensorboard_callback

# Create ModelCheckpoint callback to save model's progress
checkpoint_path = "model_checkpoints/cp.ckpt" # saving weights requires ".ckpt" extension
model_checkpoint = tf.keras.callbacks.ModelCheckpoint(checkpoint_path,
                                                       monitor="val_acc", # save the model weights with best validation accuracy
                                                       save_best_only=True, # only save the best weights
                                                       save_weights_only=True, # only save model weights (not whole model)
                                                       verbose=0) # don't print out whether or not model is being saved
```

## Setup mixed precision training

We touched on mixed precision training above.

However, we didn't quite explain it.

Normally, tensors in TensorFlow default to the float32 datatype (unless otherwise specified).

In computer science, float32 is also known as [single-precision floating-point format](#). The 32 means it usually occupies 32 bits in computer memory.

Your GPU has a limited memory, therefore it can only handle a number of float32 tensors at the same time.

This is where mixed precision training comes in.

Mixed precision training involves using a mix of float16 and float32 tensors to make better use of your GPU's memory.

Can you guess what float16 means?

Well, if you thought since float32 meant single-precision floating-point, you might've guessed float16 means [half-precision floating-point format](#). And if you did, you're right! And if not, no trouble, now you know.

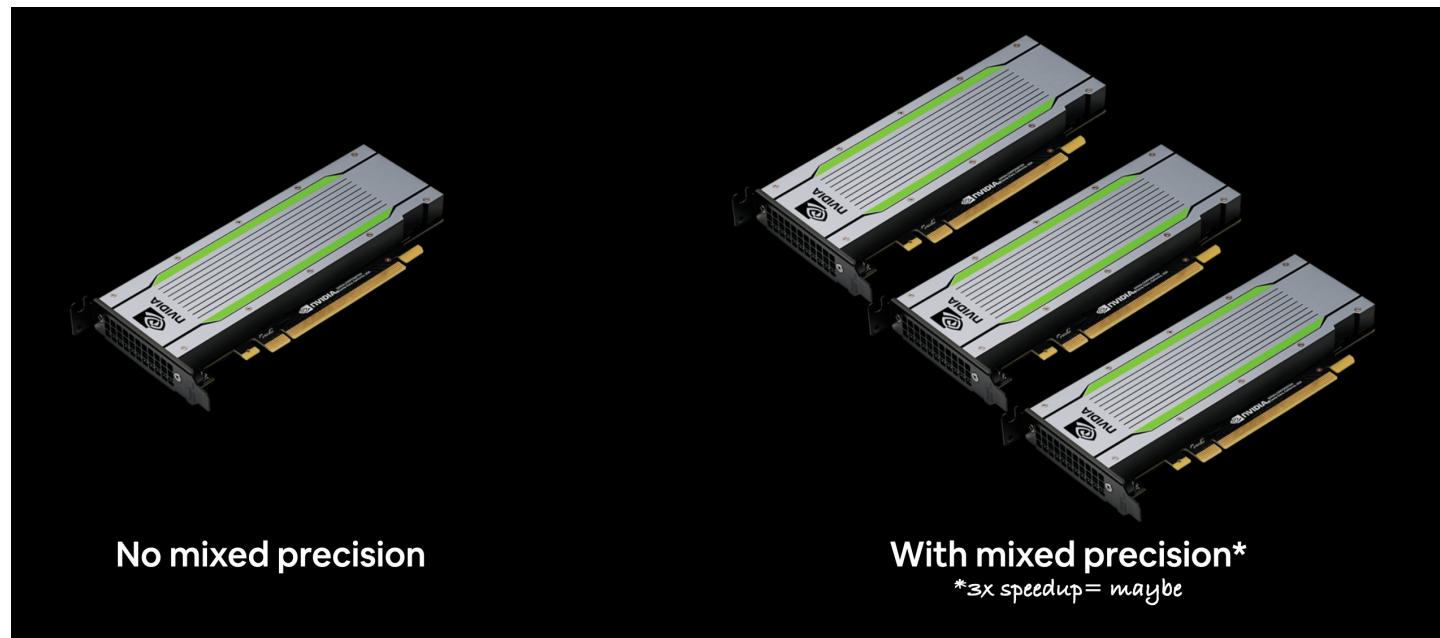
For tensors in float16 format, each element occupies 16 bits in computer memory.

So, where does this leave us?

As mentioned before, when using mixed precision training, your model will make use of float32 and float16 data types to use less memory where possible and in turn run faster (using less memory per tensor means more tensors can be computed on simultaneously).

As a result, using mixed precision training can improve your performance on modern GPUs (those with a compute capability score of 7.0+) by up to 3x.

For a more detailed explanation, I encourage you to read through the [TensorFlow mixed precision guide](#) (I'd highly recommend at least checking out the summary).



Because mixed precision training uses a combination of float32 and float16 data types, you may see up to a 3x speedup on modern GPUs.

□ **Note:** If your GPU doesn't have a score of over 7.0+ (e.g. P100 in Colab), mixed precision won't work (see: "[Supported Hardware](#)" in the mixed precision guide for more).

□ **Resource:** If you'd like to learn more about precision in computer science (the detail to which a numerical quantity is expressed by a computer), see the [Wikipedia page](#) (and accompanying resources).

Okay, enough talk, let's see how we can turn on mixed precision training in TensorFlow.

The beautiful thing is, the `tensorflow.keras.mixed_precision` API has made it very easy for us to get started.

First, we'll import the API and then use the `set_global_policy()` method to set the *dtype policy* to "mixed\_float16".

In [ ]:

```
# Turn on mixed precision training
from tensorflow.keras import mixed_precision
mixed_precision.set_global_policy(policy="mixed_float16") # set global policy to mixed precision
```

INFO:tensorflow:Mixed precision compatibility check (mixed\_float16): OK  
Your GPU will likely run quickly with dtype policy mixed\_float16 as it has compute capability of at least 7.0. Your GPU: Tesla T4, compute capability 7.5

INFO:tensorflow:Mixed precision compatibility check (mixed\_float16): OK  
Your GPU will likely run quickly with dtype policy mixed\_float16 as it has compute capabi

Nice! As long as the GPU you're using has a compute capability of 7.0+ the cell above should run without error.

Now we can check the global dtype policy (the policy which will be used by layers in our model) using the `mixed_precision.global_policy()` method.

In [ ]:

```
mixed_precision.global_policy() # should output "mixed_float16"
```

Out [ ]:

```
<Policy "mixed_float16">
```

Great, since the global dtype policy is now `"mixed_float16"` our model will automatically take advantage of float16 variables where possible and in turn speed up training.

## Build feature extraction model

Callbacks: ready to roll.

Mixed precision: turned on.

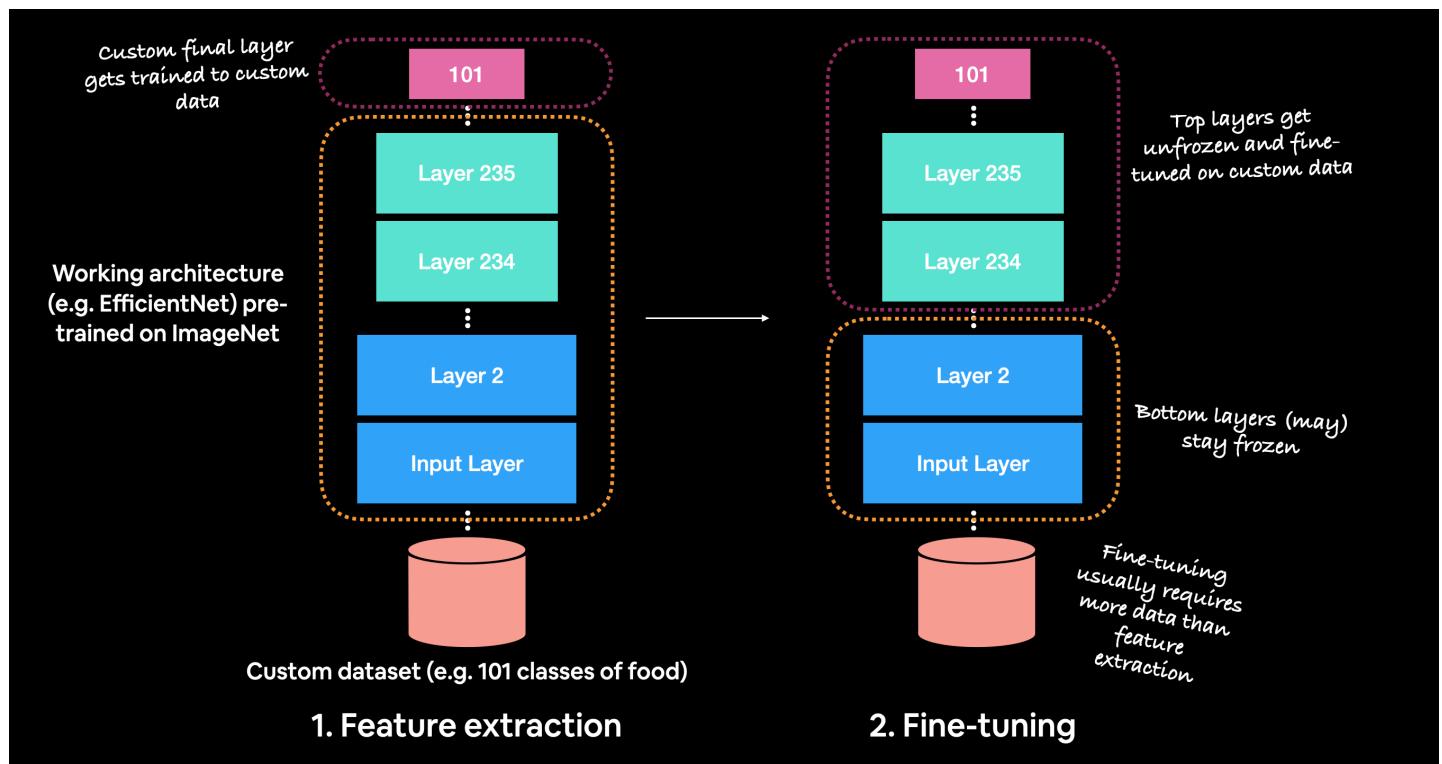
Let's build a model.

Because our dataset is quite large, we're going to move towards fine-tuning an existing pretrained model (EfficientNetB0).

But before we get into fine-tuning, let's set up a feature-extraction model.

Recall, the typical order for using transfer learning is:

1. Build a feature extraction model (replace the top few layers of a pretrained model)
2. Train for a few epochs with lower layers frozen
3. Fine-tune if necessary with multiple layers unfrozen



Before fine-tuning, it's best practice to train a feature extraction model with custom top layers.

To build the feature extraction model (covered in [Transfer Learning in TensorFlow Part 1: Feature extraction](#)), we'll:

- Use EfficientNetB0 from `tf.keras.applications` pre-trained on ImageNet as our base model

- We'll download this without the top layers using `include_top=False` parameter so we can create our own output layers
- Freeze the base model layers so we can use the pre-learned patterns the base model has found on ImageNet
- Put together the input, base model, pooling and output layers in a [Functional model](#)
- Compile the Functional model using the Adam optimizer and [sparse categorical crossentropy](#) as the loss function (since our labels aren't one-hot encoded)
- Fit the model for 3 epochs using the TensorBoard and ModelCheckpoint callbacks

▪ Note: Since we're using mixed precision training, our model needs a separate output layer with a hard-coded `dtype=float32`, for example, `layers.Activation("softmax", dtype=tf.float32)`. This ensures the outputs of our model are returned back to the float32 data type which is more numerically stable than the float16 datatype (important for loss calculations). See the "[Building the model](#)" section in the TensorFlow mixed precision guide for more.

```

import tensorflow as tf
from tensorflow.keras import layers

# Download base model and freeze underlying layers
base_model = tf.keras.applications.EfficientNetB0(include_top=False)
base_model.trainable = False

# Create functional model
inputs = layers.Input(shape=INPUT_SHAPE)
x = base_model(inputs, training=False)
x = layers.GlobalAveragePooling2D()(x)
outputs = layers.Dense(10, activation="softmax")(x)
model = tf.keras.Model(inputs, outputs)

# Compile
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=tf.keras.optimizers.Adam(),
              metrics=["accuracy"])

```

No mixed precision

```

import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras import mixed_precision 1

# Turn on mixed precision training
mixed_precision.set_global_policy("mixed_float16") 2

# Download base model and freeze underlying layers
base_model = tf.keras.applications.EfficientNetB0(include_top=False)
base_model.trainable = False

# Create functional model
inputs = layers.Input(shape=INPUT_SHAPE)
x = base_model(inputs, training=False)
x = layers.GlobalAveragePooling2D()(x)
x = layers.Dense(10)(x)
# Mixed precision requires output layer to have dtype=tf.float32 3
outputs = layers.Activation("softmax", dtype=tf.float32)(x)
model = tf.keras.Model(inputs, outputs)

# Compile
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=tf.keras.optimizers.Adam(),
              metrics=["accuracy"])

```

With mixed precision

Turning mixed precision on in TensorFlow with 3 lines of code.

In [ ]:

```

from tensorflow.keras import layers
from tensorflow.keras.layers.experimental import preprocessing

# Create base model
input_shape = (224, 224, 3)
base_model = tf.keras.applications.EfficientNetB0(include_top=False)
base_model.trainable = False # freeze base model layers

# Create Functional model
inputs = layers.Input(shape=input_shape, name="input_layer")
# Note: EfficientNetBX models have rescaling built-in but if your model didn't you could
# have a layer like below
# x = preprocessing.Rescaling(1./255)(x)
x = base_model(inputs, training=False) # set base_model to inference mode only
x = layers.GlobalAveragePooling2D(name="pooling_layer")(x)
x = layers.Dense(len(class_names))(x) # want one output neuron per class
# Separate activation of output layer so we can output float32 activations
outputs = layers.Activation("softmax", dtype=tf.float32, name="softmax_float32")(x)
model = tf.keras.Model(inputs, outputs)

# Compile the model
model.compile(loss="sparse_categorical_crossentropy", # Use sparse_categorical_crossentropy when labels are *not* one-hot
              optimizer=tf.keras.optimizers.Adam(),
              metrics=["accuracy"])

```

Downloading data from [https://storage.googleapis.com/keras-applications/efficientnetb0\\_no\\_top.h5](https://storage.googleapis.com/keras-applications/efficientnetb0_no_top.h5)  
16711680/16705208 [=====] - 0s 0us/step

In [ ]:

```
# Check out our model  
model.summary()
```

Model: "model"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	[(None, 224, 224, 3)]	0
efficientnetb0 (Functional)	(None, None, None, 1280)	4049571
pooling_layer (GlobalAverage)	(None, 1280)	0
dense (Dense)	(None, 101)	129381
softmax_float32 (Activation)	(None, 101)	0
=====		
Total params:	4,178,952	
Trainable params:	129,381	
Non-trainable params:	4,049,571	

## Checking layer dtype policies (are we using mixed precision?)

Model ready to go!

Before we said the mixed precision API will automatically change our layers' dtype policy's to whatever the global dtype policy is (in our case it's "mixed\_float16").

We can check this by iterating through our model's layers and printing layer attributes such as `dtype` and `dtype_policy`.

In [ ]:

```
# Check the dtype_policy attributes of layers in our model  
for layer in model.layers:  
    print(layer.name, layer.trainable, layer.dtype, layer.dtype_policy) # Check the dtype  
policy of layers
```

```
input_layer True float32 <Policy "float32">  
efficientnetb0 False float32 <Policy "mixed_float16">  
pooling_layer True float32 <Policy "mixed_float16">  
dense True float32 <Policy "mixed_float16">  
softmax_float32 True float32 <Policy "float32">
```

Going through the above we see:

- `layer.name` (str) : a layer's human-readable name, can be defined by the `name` parameter on construction
- `layer.trainable` (bool) : whether or not a layer is trainable (all of our layers are trainable except the `efficientnetb0` layer since we set its `trainable` attribute to `False`)
- `layer.dtype` : the data type a layer stores its variables in
- `layer.dtype_policy` : the data type a layer computes in

▀ Note: A layer can have a `dtype` of `float32` and a `dtype policy` of `"mixed_float16"` because it stores its variables (weights & biases) in `float32` (more numerically stable), however it computes in `float16` (faster).

We can also check the same details for our model's base model.

In [ ]:

```
# Check the layers in the base model and see what dtype policy they're using  
for layer in model.layers[1].layers[:20]: # only check the first 20 layers to save output space
```

```
print(layer.name, layer.trainable, layer.dtype, layer.dtype_policy)
```

```
input_1 False float32 <Policy "float32">
rescaling False float32 <Policy "mixed_float16">
normalization False float32 <Policy "float32">
stem_conv_pad False float32 <Policy "mixed_float16">
stem_conv False float32 <Policy "mixed_float16">
stem_bn False float32 <Policy "mixed_float16">
stem_activation False float32 <Policy "mixed_float16">
block1a_dwconv False float32 <Policy "mixed_float16">
block1a_bn False float32 <Policy "mixed_float16">
block1a_activation False float32 <Policy "mixed_float16">
block1a_se_squeeze False float32 <Policy "mixed_float16">
block1a_se_reshape False float32 <Policy "mixed_float16">
block1a_se_reduce False float32 <Policy "mixed_float16">
block1a_se_expand False float32 <Policy "mixed_float16">
block1a_se_excite False float32 <Policy "mixed_float16">
block1a_project_conv False float32 <Policy "mixed_float16">
block1a_project_bn False float32 <Policy "mixed_float16">
block2a_expand_conv False float32 <Policy "mixed_float16">
block2a_expand_bn False float32 <Policy "mixed_float16">
block2a_expand_activation False float32 <Policy "mixed_float16">
```

▀ **Note:** The mixed precision API automatically causes layers which can benefit from using the "mixed\_float16" dtype policy to use it. It also prevents layers which shouldn't use it from using it (e.g. the normalization layer at the start of the base model).

## Fit the feature extraction model

Now that's one good looking model. Let's fit it to our data shall we?

Three epochs should be enough for our top layers to adjust their weights enough to our food image data.

To save time per epoch, we'll also only validate on 15% of the test data.

In [ ]:

```
# Fit the model with callbacks
history_101_food_classes_feature_extract = model.fit(train_data,
                                                       epochs=3,
                                                       steps_per_epoch=len(train_data),
                                                       validation_data=test_data,
                                                       validation_steps=int(0.15 * len(te
st_data)),
                                                       callbacks=[create_tensorboard_call
back("training_logs",
      "efficientnetb0_101_classes_all_data_feature_extract"),
      model_checkpoint])
```

```
Saving TensorBoard log files to: training_logs/efficientnetb0_101_classes_all_data_featur
e_extract/20210317-032425
Epoch 1/3
2368/2368 [=====] - 218s 76ms/step - loss: 2.3139 - accuracy: 0.
4693 - val_loss: 1.2324 - val_accuracy: 0.6711
Epoch 2/3
2368/2368 [=====] - 156s 65ms/step - loss: 1.3124 - accuracy: 0.
6626 - val_loss: 1.1215 - val_accuracy: 0.7005
Epoch 3/3
2368/2368 [=====] - 154s 64ms/step - loss: 1.1467 - accuracy: 0.
7009 - val_loss: 1.0856 - val_accuracy: 0.7105
```

Nice, looks like our feature extraction model is performing pretty well. How about we evaluate it on the whole test dataset?

```
In [ ]:  
  
# Evaluate model (unsaved version) on whole test dataset  
results_feature_extract_model = model.evaluate(test_data)  
results_feature_extract_model  
  
790/790 [=====] - 53s 67ms/step - loss: 1.0854 - accuracy: 0.709  
8  
  
Out[ ]:  
[1.0854089260101318, 0.7098217606544495]
```

And since we used the `ModelCheckpoint` callback, we've got a saved version of our model in the `model_checkpoints` directory.

Let's load it in and make sure it performs just as well.

## Load and evaluate checkpoint weights

We can load in and evaluate our model's checkpoints by:

1. Cloning our model using `tf.keras.models.clone_model()` to make a copy of our feature extraction model with reset weights.
2. Calling the `load_weights()` method on our cloned model passing it the path to where our checkpointed weights are stored.
3. Calling `evaluate()` on the cloned model with loaded weights.

A reminder, checkpoints are helpful for when you perform an experiment such as fine-tuning your model. In the case you fine-tune your feature extraction model and find it doesn't offer any improvements, you can always revert back to the checkpointed version of your model.

In [ ]:

```
# Clone the model we created (this resets all weights)  
cloned_model = tf.keras.models.clone_model(model)  
cloned_model.summary()
```

Model: "model"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	[None, 224, 224, 3]	0
efficientnetb0 (Functional)	(None, None, None, 1280)	4049571
pooling_layer (GlobalAverage)	(None, 1280)	0
dense (Dense)	(None, 101)	129381
softmax_float32 (Activation)	(None, 101)	0

Total params: 4,178,952  
Trainable params: 129,381  
Non-trainable params: 4,049,571

In [ ]:

```
# Where are our checkpoints stored?  
checkpoint_path
```

Out[ ]:

```
'model_checkpoints/cp.ckpt'
```

In [ ]:

```
# Load checkpointed weights into cloned_model
```

```
cloned_model.load_weights(checkpoint_path)
```

Out [ ]:

```
<tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x7f186ca8ba90>
```

**Each time you make a change to your model (including loading weights), you have to recompile.**

In [ ]:

```
# Compile cloned_model (with same parameters as original model)
cloned_model.compile(loss="sparse_categorical_crossentropy",
                      optimizer=tf.keras.optimizers.Adam(),
                      metrics=["accuracy"])
```

In [ ]:

```
# Evaluate cloned model with loaded weights (should be same score as trained model)
results_cloned_model_with_loaded_weights = cloned_model.evaluate(test_data)
```

```
790/790 [=====] - 47s 56ms/step - loss: 1.0916 - accuracy: 0.704
3
```

**Our cloned model with loaded weight's results should be very close to the feature extraction model's results (if the cell below errors, something went wrong).**

In [ ]:

```
# Loaded checkpoint weights should return very similar results to checkpoint weights prior to saving
import numpy as np
assert np.isclose(results_feature_extract_model, results_cloned_model_with_loaded_weights).all() # check if all elements in array are close
```

**Cloning the model preserves `dtype_policy`'s of layers (but doesn't preserve weights) so if we wanted to continue fine-tuning with the cloned model, we could and it would still use the mixed precision `dtype policy`.**

In [ ]:

```
# Check the layers in the base model and see what dtype policy they're using
for layer in cloned_model.layers[1].layers[:20]: # check only the first 20 layers to save space
    print(layer.name, layer.trainable, layer.dtype, layer.dtype_policy)
```

```
input_1 True float32 <Policy "float32">
rescaling False float32 <Policy "mixed_float16">
normalization False float32 <Policy "float32">
stem_conv_pad False float32 <Policy "mixed_float16">
stem_conv False float32 <Policy "mixed_float16">
stem_bn False float32 <Policy "mixed_float16">
stem_activation False float32 <Policy "mixed_float16">
block1a_dwconv False float32 <Policy "mixed_float16">
block1a_bn False float32 <Policy "mixed_float16">
block1a_activation False float32 <Policy "mixed_float16">
block1a_se_squeeze False float32 <Policy "mixed_float16">
block1a_se_reshape False float32 <Policy "mixed_float16">
block1a_se_reduce False float32 <Policy "mixed_float16">
block1a_se_expand False float32 <Policy "mixed_float16">
block1a_se_excite False float32 <Policy "mixed_float16">
block1a_project_conv False float32 <Policy "mixed_float16">
block1a_project_bn False float32 <Policy "mixed_float16">
block2a_expand_conv False float32 <Policy "mixed_float16">
block2a_expand_bn False float32 <Policy "mixed_float16">
block2a_expand_activation False float32 <Policy "mixed_float16">
```

## Save the whole model to file

We can also save the whole model using the `save()` method.

Since our model is quite large, you might want to save it to Google Drive (if you're using Google Colab) so you can load it in for use later.

**Note:** Saving to Google Drive requires mounting Google Drive (go to Files -> Mount Drive).

In [ ]:

```
# ## Saving model to Google Drive (optional)

# # Create save path to drive
# save_dir = "drive/MyDrive/tensorflow_course/food_vision/07_efficientnetb0_feature_extra
ct_model_mixed_precision/"
# # os.makedirs(save_dir) # Make directory if it doesn't exist

# # Save model
# model.save(save_dir)
```

We can also save it directly to our Google Colab instance.

**Note:** Google Colab storage is ephemeral and your model will delete itself (along with any other saved files) when the Colab session expires.

In [ ]:

```
# Save model locally (if you're using Google Colab, your saved model will Colab instance
terminates)
save_dir = "07_efficientnetb0_feature_extract_model_mixed_precision"
model.save(save_dir)
```

INFO:tensorflow:Assets written to: 07\_efficientnetb0\_feature\_extract\_model\_mixed\_precision/assets

INFO:tensorflow:Assets written to: 07\_efficientnetb0\_feature\_extract\_model\_mixed\_precision/assets

And again, we can check whether or not our model saved correctly by loading it in and evaluating it.

In [ ]:

```
# Load model previously saved above
loaded_saved_model = tf.keras.models.load_model(save_dir)
```

Loading a `SavedModel` also retains all of the underlying layers `dtype_policy` (we want them to be `"mixed_float16"`).

In [ ]:

```
# Check the layers in the base model and see what dtype policy they're using
for layer in loaded_saved_model.layers[1].layers[:20]: # check only the first 20 layers
    print(layer.name, layer.trainable, layer.dtype, layer.dtype_policy)
```

```
input_1 True float32 <Policy "float32">
rescaling True float32 <Policy "mixed_float16">
normalization True float32 <Policy "float32">
stem_conv_pad True float32 <Policy "mixed_float16">
stem_conv True float32 <Policy "mixed_float16">
stem_bn True float32 <Policy "mixed_float16">
stem_activation True float32 <Policy "mixed_float16">
block1a_dwconv True float32 <Policy "mixed_float16">
block1a_bn True float32 <Policy "mixed_float16">
block1a_activation True float32 <Policy "mixed_float16">
block1a_se_squeeze True float32 <Policy "mixed_float16">
block1a_se_reshape True float32 <Policy "mixed_float16">
```

```
block1a_se_reduce True float32 <Policy "mixed_float16">
block1a_se_expand True float32 <Policy "mixed_float16">
block1a_se_excite True float32 <Policy "mixed_float16">
block1a_project_conv True float32 <Policy "mixed_float16">
block1a_project_bn True float32 <Policy "mixed_float16">
block2a_expand_conv True float32 <Policy "mixed_float16">
block2a_expand_bn True float32 <Policy "mixed_float16">
block2a_expand_activation True float32 <Policy "mixed_float16">
```

In [ ]:

```
# Check loaded model performance (this should be the same as results_feature_extract_model)
results_loaded_saved_model = loaded_saved_model.evaluate(test_data)
results_loaded_saved_model
```

```
790/790 [=====] - 46s 56ms/step - loss: 1.0854 - accuracy: 0.7098
```

Out[ ]:

```
[1.0854086875915527, 0.7098217606544495]
```

In [ ]:

```
# The loaded model's results should equal (or at least be very close) to the model's results prior to saving
# Note: this will only work if you've instantiated results variables
import numpy as np
assert np.isclose(results_feature_extract_model, results_loaded_saved_model).all()
```

That's what we want! Our loaded model performing as it should.

**Note:** We spent a fair bit of time making sure our model saved correctly because training on a lot of data can be time-consuming, so we want to make sure we don't have to continually train from scratch.

## Preparing our model's layers for fine-tuning

Our feature-extraction model is showing some great promise after three epochs. But since we've got so much data, it's probably worthwhile that we see what results we can get with fine-tuning (fine-tuning usually works best when you've got quite a large amount of data).

Remember our goal of beating the [DeepFood paper](#)?

They were able to achieve 77.4% top-1 accuracy on Food101 over 2-3 days of training.

Do you think fine-tuning will get us there?

Let's find out.

To start, let's load in our saved model.

**Note:** It's worth remembering a traditional workflow for fine-tuning is to freeze a pre-trained base model and then train only the output layers for a few iterations so their weights can be updated inline with your custom data (feature extraction). And then unfreeze a number or all of the layers in the base model and continue training until the model stops improving.

Like all good cooking shows, I've saved a model I prepared earlier (the feature extraction model from above) to Google Storage.

We can download it to make sure we're using the same model going forward.

In [ ]:

```
# Download the saved model from Google Storage
```

```
# Download the saved model from Google Storage
!wget https://storage.googleapis.com/ztm_tf_course/food_vision/07_efficientnetb0_feature_extract_model_mixed_precision.zip

--2021-03-17 03:44:38-- https://storage.googleapis.com/ztm_tf_course/food_vision/07_efficientnetb0_feature_extract_model_mixed_precision.zip
Resolving storage.googleapis.com (storage.googleapis.com) ... 74.125.195.128, 74.125.20.128, 74.125.142.128, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|74.125.195.128|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 16976857 (16M) [application/zip]
Saving to: '07_efficientnetb0_feature_extract_model_mixed_precision.zip'

07_efficientnetb0_f 100%[=====] 16.19M 79.1MB/s in 0.2s

2021-03-17 03:44:38 (79.1 MB/s) - '07_efficientnetb0_feature_extract_model_mixed_precision.zip' saved [16976857/16976857]
```

In [ ]:

```
# Unzip the SavedModel downloaded from Google Storage
!mkdir downloaded_gs_model # create new dir to store downloaded feature extraction model
!unzip 07_efficientnetb0_feature_extract_model_mixed_precision.zip -d downloaded_gs_model
```

```
Archive: 07_efficientnetb0_feature_extract_model_mixed_precision.zip
  creating: downloaded_gs_model/07_efficientnetb0_feature_extract_model_mixed_precision/
  creating: downloaded_gs_model/07_efficientnetb0_feature_extract_model_mixed_precision/
variables/
  inflating: downloaded_gs_model/07_efficientnetb0_feature_extract_model_mixed_precision/
variables/variables.data-00000-of-00001
  inflating: downloaded_gs_model/07_efficientnetb0_feature_extract_model_mixed_precision/
variables/variables.index
  inflating: downloaded_gs_model/07_efficientnetb0_feature_extract_model_mixed_precision/
saved_model.pb
  creating: downloaded_gs_model/07_efficientnetb0_feature_extract_model_mixed_precision/
assets/
```

In [ ]:

```
# Load and evaluate downloaded GS model
loaded_gs_model = tf.keras.models.load_model("/content/downloaded_gs_model/07_efficientnetb0_feature_extract_model_mixed_precision")
```

In [ ]:

```
# Get a summary of our downloaded model
loaded_gs_model.summary()
```

Model: "model"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	[(None, 224, 224, 3)]	0
efficientnetb0 (Functional)	(None, None, None, 1280)	4049571
pooling_layer (GlobalAverage)	(None, 1280)	0
dense (Dense)	(None, 101)	129381
softmax_float32 (Activation)	(None, 101)	0
Total params:	4,178,952	
Trainable params:	4,136,929	
Non-trainable params:	42,023	

And now let's make sure our loaded model is performing as expected.

In [ ]:

```
# How does the loaded model perform?  
results_loaded_gs_model = loaded_gs_model.evaluate(test_data)  
results_loaded_gs_model
```

790/790 [=====] - 46s 57ms/step - loss: 1.0881 - accuracy: 0.706  
5

Out [ ]:

```
[1.0881282091140747, 0.7064950466156006]
```

**Great, our loaded model is performing as expected.**

**When we first created our model, we froze all of the layers in the base model by setting**

```
base_model.trainable=False
```

**but since we've loaded in our model from file, let's check whether or not the layers are trainable or not.**

In [ ]:

```
# Are any of the layers in our model frozen?  
for layer in loaded_gs_model.layers:  
    layer.trainable = True # set all layers to trainable  
    print(layer.name, layer.trainable, layer.dtype, layer.dtype_policy) # make sure loaded  
model is using mixed precision dtype_policy ("mixed_float16")
```

```
input_layer True float32 <Policy "float32">  
efficientnetb0 True float32 <Policy "mixed_float16">  
pooling_layer True float32 <Policy "mixed_float16">  
dense True float32 <Policy "mixed_float16">  
softmax_float32 True float32 <Policy "float32">
```

**Alright, it seems like each layer in our loaded model is trainable. But what if we got a little deeper and inspected each of the layers in our base model?**

QUESTION: **Which layer in the loaded model is our base model?**

**Before saving the Functional model to file, we created it with five layers (layers below are 0-indexed):**

1. The input layer
2. The pre-trained base model layer (`tf.keras.applications.EfficientNetB0`)
3. The pooling layer
4. The fully-connected (dense) layer
5. The output softmax activation (with float32 dtype)

**Therefore to inspect our base model layer, we can access the `layers` attribute of the layer at index 1 in our model.**

In [ ]:

```
# Check the layers in the base model and see what dtype policy they're using  
for layer in loaded_gs_model.layers[1].layers[:20]:  
    print(layer.name, layer.trainable, layer.dtype, layer.dtype_policy)
```

```
input_1 True float32 <Policy "float32">  
rescaling True float32 <Policy "mixed_float16">  
normalization True float32 <Policy "float32">  
stem_conv_pad True float32 <Policy "mixed_float16">  
stem_conv True float32 <Policy "mixed_float16">  
stem_bn True float32 <Policy "mixed_float16">  
stem_activation True float32 <Policy "mixed_float16">  
block1a_dwconv True float32 <Policy "mixed_float16">  
block1a_bn True float32 <Policy "mixed_float16">  
block1a_activation True float32 <Policy "mixed_float16">  
block1a_se_squeeze True float32 <Policy "mixed_float16">  
block1a_se_reshape True float32 <Policy "mixed_float16">  
block1a_se_reduce True float32 <Policy "mixed_float16">
```

```
block1a_se_expand True float32 <Policy "mixed_float16">
block1a_se_excite True float32 <Policy "mixed_float16">
block1a_project_conv True float32 <Policy "mixed_float16">
block1a_project_bn True float32 <Policy "mixed_float16">
block2a_expand_conv True float32 <Policy "mixed_float16">
block2a_expand_bn True float32 <Policy "mixed_float16">
block2a_expand_activation True float32 <Policy "mixed_float16">
```

Wonderful, it looks like each layer in our base model is trainable (unfrozen) and every layer which should be using the dtype policy "mixed\_policy16" is using it.

Since we've got so much data (750 images x 101 training classes = 75750 training images), let's keep all of our base model's layers unfrozen.

**Note:** If you've got a small amount of data (less than 100 images per class), you may want to only unfreeze and fine-tune a small number of layers in the base model at a time. Otherwise, you risk overfitting.

## A couple more callbacks

We're about to start fine-tuning a deep learning model with over 200 layers using over 100,000 (75k+ training, 25K+ testing) images, which means our model's training time is probably going to be much longer than before.

**Question:** How long does training take?

It could be a couple of hours or in the case of the [DeepFood paper](#) (the baseline we're trying to beat), their best performing model took 2-3 days of training time.

You will really only know how long it'll take once you start training.

**Question:** When do you stop training?

Ideally, when your model stops improving. But again, due to the nature of deep learning, it can be hard to know when exactly a model will stop improving.

Luckily, there's a solution: the [EarlyStopping callback](#).

The EarlyStopping callback monitors a specified model performance metric (e.g. val\_loss) and when it stops improving for a specified number of epochs, automatically stops training.

Using the EarlyStopping callback combined with the ModelCheckpoint callback saving the best performing model automatically, we could keep our model training for an unlimited number of epochs until it stops improving.

Let's set both of these up to monitor our model's val\_loss.

In [ ]:

```
# Setup EarlyStopping callback to stop training if model's val_loss doesn't improve for 3 epochs
early_stopping = tf.keras.callbacks.EarlyStopping(monitor="val_loss", # watch the val loss metric
                                                 patience=3) # if val loss decreases for 3 epochs in a row, stop training

# Create ModelCheckpoint callback to save best model during fine-tuning
checkpoint_path = "fine_tune_checkpoints/"
model_checkpoint = tf.keras.callbacks.ModelCheckpoint(checkpoint_path,
                                                      save_best_only=True,
                                                      monitor="val_loss")
```

Woohoo! Fine-tuning callbacks ready.

If you're planning on training large models, the `ModelCheckpoint` and `EarlyStopping` are two callbacks you'll want to become very familiar with.

We're almost ready to start fine-tuning our model but there's one more callback we're going to implement: `ReduceLROnPlateau`.

Remember how the learning rate is the most important model hyperparameter you can tune? (if not, treat this as a reminder).

Well, the `ReduceLROnPlateau` callback helps to tune the learning rate for you.

Like the `ModelCheckpoint` and `EarlyStopping` callbacks, the `ReduceLROnPlateau` callback monitors a specified metric and when that metric stops improving, it reduces the learning rate by a specified factor (e.g. divides the learning rate by 10).

#### Question: Why lower the learning rate?

Imagine having a coin at the back of the couch and you're trying to grab with your fingers.

Now think of the learning rate as the size of the movements your hand makes towards the coin.

The closer you get, the smaller you want your hand movements to be, otherwise the coin will be lost.

Our model's ideal performance is the equivalent of grabbing the coin. So as training goes on and our model gets closer and closer to its ideal performance (also called **convergence**), we want the amount it learns to be less and less.

To do this we'll create an instance of the `ReduceLROnPlateau` callback to monitor the validation loss just like the `EarlyStopping` callback.

Once the validation loss stops improving for two or more epochs, we'll reduce the learning rate by a factor of 5 (e.g. `0.001` to `0.0002`).

And to make sure the learning rate doesn't get too low (and potentially result in our model learning nothing), we'll set the minimum learning rate to `1e-7`.

In [ ]:

```
# Creating learning rate reduction callback
reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(monitor="val_loss",
                                                 factor=0.2, # multiply the learning rate by 0.2 (reduce by 5x)
                                                 patience=2,
                                                 verbose=1, # print out when learning rate goes down
                                                 min_lr=1e-7)
```

Learning rate reduction ready to go!

Now before we start training, we've got to recompile our model.

We'll use sparse categorical crossentropy as the loss and since we're fine-tuning, we'll use a 10x lower learning rate than the Adam optimizers default (`1e-4` instead of `1e-3`).

In [ ]:

```
# Compile the model
loaded_gs_model.compile(loss="sparse_categorical_crossentropy", # sparse_categorical_crossentropy for labels that are *not* one-hot
                        optimizer=tf.keras.optimizers.Adam(0.0001), # 10x lower learning rate than the default
                        metrics=["accuracy"])
```

Okay, model compiled.

Now let's fit it on all of the data.

We'll set it up to run for up to 100 epochs.

Since we're going to be using the `EarlyStopping` callback, it might stop before reaching 100 epochs.

**Note:** Running the cell below will set the model up to fine-tune all of the pre-trained weights in the base model on all of the Food101 data. Doing so with unoptimized data pipelines and without mixed precision training will take a fairly long time per epoch depending on what type of GPU you're using (about 15-20 minutes on Colab GPUs). But don't worry, the code we've written above will ensure it runs much faster (more like 4-5 minutes per epoch).

In [ ]:

```
# Start to fine-tune (all layers)
history_101_food_classes_all_data_fine_tune = loaded_gs_model.fit(train_data,
                                                               epochs=100, # fine-tune for a maximum of 100 epochs
                                                               steps_per_epoch=len(train_data),
                                                               validation_data=test_data,
                                                               validation_steps=int(0.15 * len(test_data)), # validation during training on 15% of test data
                                                               callbacks=[create_tensorboard_callback("training_logs", "efficientb0_101_classes_all_data_fine_tuning"), # track the model training logs
                                                               model_checkpoint, # save only the best model during training
                                                               early_stopping, # stop model after X epochs of no improvements
                                                               reduce_lr]) # reduce the learning rate after X epochs of no improvements
```

Saving TensorBoard log files to: `training_logs/efficientb0_101_classes_all_data_fine_tuning/20210317-034947`  
Epoch 1/100  
2368/2368 [=====] - 304s 123ms/step - loss: 0.9813 - accuracy: 0.7357 - val\_loss: 0.7932 - val\_accuracy: 0.7842  
INFO:tensorflow:Assets written to: `fine_tune_checkpoints/assets`

INFO:tensorflow:Assets written to: `fine_tune_checkpoints/assets`

Epoch 2/100  
2368/2368 [=====] - 287s 121ms/step - loss: 0.5896 - accuracy: 0.8363 - val\_loss: 0.8097 - val\_accuracy: 0.7820  
Epoch 3/100  
2368/2368 [=====] - 287s 121ms/step - loss: 0.3337 - accuracy: 0.9068 - val\_loss: 0.8791 - val\_accuracy: 0.7850

Epoch 00003: ReduceLROnPlateau reducing learning rate to 1.9999999494757503e-05.  
Epoch 4/100  
2368/2368 [=====] - 287s 121ms/step - loss: 0.1099 - accuracy: 0.9724 - val\_loss: 0.9399 - val\_accuracy: 0.8008

**Note:** If you didn't use mixed precision or use techniques such as `prefetch()` in the `Batch & prepare datasets` section, your model fine-tuning probably takes up to 2.5-3x longer per epoch (see the output below for an example).

Prefetch and mixed precision   No prefetch and no mixed precision

Time per epoch	Prefetch and mixed precision	No prefetch and no mixed precision
Time per epoch	~280-300s	~1127-1397s

**Results from fine-tuning Food Vision Big™ on Food101 dataset using an EfficientNetB0 backbone using a Google Colab Tesla T4 GPU.**

Saving TensorBoard log files to: `training_logs/efficientb0_101_classes_all_data_fi`

```
ne_tuning/20200928-013008
Epoch 1/100
2368/2368 [=====] - 1397s 590ms/step - loss: 1.2068 - accuracy: 0.6820 - val_loss: 1.1623 - val_accuracy: 0.6894
Epoch 2/100
2368/2368 [=====] - 1193s 504ms/step - loss: 0.9459 - accuracy: 0.7444 - val_loss: 1.1549 - val_accuracy: 0.6872
Epoch 3/100
2368/2368 [=====] - 1143s 482ms/step - loss: 0.7848 - accuracy: 0.7838 - val_loss: 1.0402 - val_accuracy: 0.7142
Epoch 4/100
2368/2368 [=====] - 1127s 476ms/step - loss: 0.6599 - accuracy: 0.8149 - val_loss: 0.9599 - val_accuracy: 0.7373
```

**Example fine-tuning time for non-prefetched data as well as non-mixed precision training (~2.5-3x longer per epoch).**

**Let's make sure we save our model before we start evaluating it.**

In [ ]:

```
# # Save model to Google Drive (optional)
# loaded_gs_model.save("/content/drive/MyDrive/tensorflow_course/food_vision/07_efficientnetb0_fine_tuned_101_classes_mixed_precision/")
```

In [ ]:

```
# Save model locally (note: if you're using Google Colab and you save your model locally, it will be deleted when your Google Colab session ends)
loaded_gs_model.save("07_efficientnetb0_fine_tuned_101_classes_mixed_precision")
```

INFO:tensorflow:Assets written to: 07\_efficientnetb0\_fine\_tuned\_101\_classes\_mixed\_precision/assets

INFO:tensorflow:Assets written to: 07\_efficientnetb0\_fine\_tuned\_101\_classes\_mixed\_precision/assets

**Looks like our model has gained a few performance points from fine-tuning, let's evaluate on the whole test dataset and see if managed to beat the [DeepFood paper's](#) result of 77.4% accuracy.**

In [ ]:

```
# Evaluate mixed precision trained loaded model
results_loaded_gs_model_fine_tuned = loaded_gs_model.evaluate(test_data)
results_loaded_gs_model_fine_tuned
```

790/790 [=====] - 55s 69ms/step - loss: 0.9397 - accuracy: 0.796

Out[ ]:

```
[0.9397100210189819, 0.7963564395904541]
```

**Woohoo!!!! It looks like our model beat the results mentioned in the DeepFood paper for Food101 (DeepFood's 77.4% top-1 accuracy versus our ~79% top-1 accuracy).**

## Download fine-tuned model from Google Storage

**As mentioned before, training models can take a significant amount of time.**

**And again, like any good cooking show, here's something we prepared earlier...**

**It's a fine-tuned model exactly like the one we trained above but it's saved to Google Storage so it can be accessed, imported and evaluated.**

In [ ]:

```
# Download and evaluate fine-tuned model from Google Storage
!wget https://storage.googleapis.com/ztm_tf_course/food_vision/07_efficientnetb0_fine_tuned_101_classes_mixed_precision.zip

--2021-03-17 04:13:33-- https://storage.googleapis.com/ztm_tf_course/food_vision/07_efficientnetb0_fine_tuned_101_classes_mixed_precision.zip
Resolving storage.googleapis.com (storage.googleapis.com) ... 74.125.20.128, 74.125.197.128, 74.125.142.128, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|74.125.20.128|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 46790356 (45M) [application/zip]
Saving to: '07_efficientnetb0_fine_tuned_101_classes_mixed_precision.zip'

07_efficientnetb0_fine_tuned_101_classes_mixed_precision.zip 100%[=====] 44.62M 109MB/s in 0.4s

2021-03-17 04:13:34 (109 MB/s) - '07_efficientnetb0_fine_tuned_101_classes_mixed_precision.zip' saved [46790356/46790356]
```

**The downloaded model comes in zip format ( .zip ) so we'll unzip it into the Google Colab instance.**

In [ ]:

```
# Unzip fine-tuned model
!mkdir downloaded_fine_tuned_gs_model # create separate directory for fine-tuned model downloaded from Google Storage
!unzip /content/07_efficientnetb0_fine_tuned_101_classes_mixed_precision -d downloaded_fine_tuned_gs_model

Archive: /content/07_efficientnetb0_fine_tuned_101_classes_mixed_precision.zip
  creating: downloaded_fine_tuned_gs_model/07_efficientnetb0_fine_tuned_101_classes_mixed_precision/
  creating: downloaded_fine_tuned_gs_model/07_efficientnetb0_fine_tuned_101_classes_mixed_precision/variables/
    inflating: downloaded_fine_tuned_gs_model/07_efficientnetb0_fine_tuned_101_classes_mixed_precision/variables/variables.data-00000-of-00001
    inflating: downloaded_fine_tuned_gs_model/07_efficientnetb0_fine_tuned_101_classes_mixed_precision/variables/variables.index
    inflating: downloaded_fine_tuned_gs_model/07_efficientnetb0_fine_tuned_101_classes_mixed_precision/saved_model.pb
  creating: downloaded_fine_tuned_gs_model/07_efficientnetb0_fine_tuned_101_classes_mixed_precision/assets/
```

**Now we can load it using the `tf.keras.models.load_model()` method and get a summary (it should be the exact same as the model we created above).**

In [ ]:

```
# Load in fine-tuned model from Google Storage and evaluate
loaded_fine_tuned_gs_model = tf.keras.models.load_model("/content/downloaded_fine_tuned_gs_model/07_efficientnetb0_fine_tuned_101_classes_mixed_precision")
```

In [ ]:

```
# Get a model summary (same model architecture as above)
loaded_fine_tuned_gs_model.summary()
```

Model: "model"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	[(None, 224, 224, 3)]	0
efficientnetb0 (Functional)	(None, None, None, 1280)	4049571
pooling_layer (GlobalAverage)	(None, 1280)	0
dense (Dense)	(None, 101)	129381

```
softmax_float32 (Activation) (None, 101)
```

```
0
```

```
=====
```

```
Total params: 4,178,952
```

```
Trainable params: 4,136,929
```

```
Non-trainable params: 42,023
```

---

Finally, we can evaluate our model on the test data (this requires the `test_data` variable to be loaded).

In [ ]:

```
# Note: Even if you're loading in the model from Google Storage, you will still need to load the test_data variable for this cell to work
results_downloaded_fine_tuned_gs_model = loaded_fine_tuned_gs_model.evaluate(test_data)
results_downloaded_fine_tuned_gs_model
```

```
790/790 [=====] - 46s 56ms/step - loss: 0.9073 - accuracy: 0.8017
```

Out [ ]:

```
[0.907255232334137, 0.8017029762268066]
```

Excellent! Our saved model is performing as expected (better results than the DeepFood paper!).

Congratulations! You should be excited! You just trained a computer vision model with competitive performance to a research paper and in far less time (our model took ~20 minutes to train versus DeepFood's quoted 2-3 days).

In other words, you brought Food Vision life!

If you really wanted to step things up, you could try using the `EfficientNetB4` model (a larger version of `EfficientNetB0`). At the time of writing, the EfficientNet family has the [state of the art classification results](#) on the Food101 dataset.

**Resource:** To see which models are currently performing the best on a given dataset or problem type as well as the latest trending machine learning research, be sure to check out [paperswithcode.com](#) and [sotabench.com](#).

## View training results on TensorBoard

Since we tracked our model's fine-tuning training logs using the `TensorBoard` callback, let's upload them and inspect them on [TensorBoard.dev](#).

In [ ]:

```
!tensorboard dev upload --logdir ./training_logs \
--name "Fine-tuning EfficientNetB0 on all Food101 Data" \
--description "Training results for fine-tuning EfficientNetB0 on Food101 Data with learning rate 0.0001" \
--one_shot
```

In [ ]:

```
View experiment: https://tensorboard.dev/experiment/2KINdYxgSgW2bUg7dIvevw/
```

Viewing at our [model's training curves on TensorBoard.dev](#), it looks like our fine-tuning model gains boost in performance but starts to overfit as training goes on.

See the training curves on TensorBoard.dev here:

```
https://tensorboard.dev/experiment/2KINdYxgSgW2bUg7dIvevw/
```

To fix this, in future experiments, we might try things like:

- A different iteration of EfficientNet (e.g. EfficientNetB4 instead of EfficientNetB0).
- Unfreezing less layers of the base model and training them rather than unfreezing the whole base model in one go.

In [ ]:

```
# View past TensorBoard experiments
!tensorboard dev list
```

Data for the "text" plugin is now uploaded to TensorBoard.dev! Note that uploaded data is public. If you do not want to upload data for this plugin, use the "--plugins" command line argument.

```
https://tensorboard.dev/experiment/2KINdYxgSgW2bUg7dIvevw/
Name           Fine-tuning EfficientNetB0 on all Food101 Data
Description    Training results for fine-tuning EfficientNetB0 on Food101 Data with learning rate 0.0001
Id             2KINdYxgSgW2bUg7dIvevw
Created        2021-03-09 04:16:36 (2 minutes ago)
Updated        2021-03-09 04:16:39 (2 minutes ago)
Runs           4
Tags           3
Scalars        28
Tensor bytes   0
Binary object bytes 2079943

https://tensorboard.dev/experiment/vcySzjmkRkKBLVSDAQMO8g/
Name           Transfer Learning Experiments with 10 Food101 Classes
Description    A series of different transfer learning experiments with varying amounts of data and fine-tuning.
Id             vcySzjmkRkKBLVSDAQMO8g
Created        2021-02-18 05:43:45
Updated        2021-02-18 05:43:50
Runs           10
Tags           3
Scalars        108
Tensor bytes   0
Binary object bytes 4162671

https://tensorboard.dev/experiment/dQBrpdwIRgS2qI0Andv8Yg/
Name           EfficientNetB0 vs. ResNet50V2
Description    Comparing two different TF Hub feature extraction model architectures using 10% of the training data
Id             dQBrpdwIRgS2qI0Andv8Yg
Created        2021-02-15 02:13:39
Updated        2021-02-15 02:13:41
Runs           4
Tags           3
Scalars        40
Tensor bytes   0
Binary object bytes 6020432

https://tensorboard.dev/experiment/OAE6KXizQZKQxDiqI3cnUQ/
Name           Fine-tuning EfficientNetB0 on all Food101 Data
Description    Training results for fine-tuning EfficientNetB0 on Food101 Data with learning rate 0.0001
Id             OAE6KXizQZKQxDiqI3cnUQ
Created        2020-09-28 05:46:53
Updated        2020-09-28 05:46:55
Runs           2
Tags           3
Scalars        56
Tensor bytes   0
Binary object bytes 1338330

https://tensorboard.dev/experiment/2076kw3PQbK101Byfg5B4w/
Name           Transfer learning experiments
Description    A series of different transfer learning experiments with varying amounts of data and fine-tuning
Id             2076kw3PQbK101Byfg5B4w
Created        2020-09-17 22:51:37
Updated        2020-09-17 22:51:47
Runs           10
Tags           3
Scalars        128
Tensor bytes   0
```

```
TensorBoard
Binary object bytes 9520961
https://tensorboard.dev/experiment/73taSKxXQeGPQsNBcVvY3g/
Name          EfficientNetB0 vs. ResNet50V2
Description    Comparing two different TF Hub feature extraction models architec-
res using 10% of training images
Id            73taSKxXQeGPQsNBcVvY3g
Created        2020-09-14 05:02:48
Updated        2020-09-14 05:02:50
Runs           4
Tags           3
Scalars         40
Tensor bytes   0
Binary object bytes 3402042
Total: 6 experiment(s)
```

In [ ]:

```
# Delete past TensorBoard experiments
# !tensorboard dev delete --experiment_id YOUR_EXPERIMENT_ID

# Example
!tensorboard dev delete --experiment_id OAE6KXizQZKQxDiqI3cnUQ
```

Data for the "text" plugin is now uploaded to TensorBoard.dev! Note that uploaded data is public. If you do not want to upload data for this plugin, use the "--plugins" command line argument.

Deleted experiment OAE6KXizQZKQxDiqI3cnUQ.

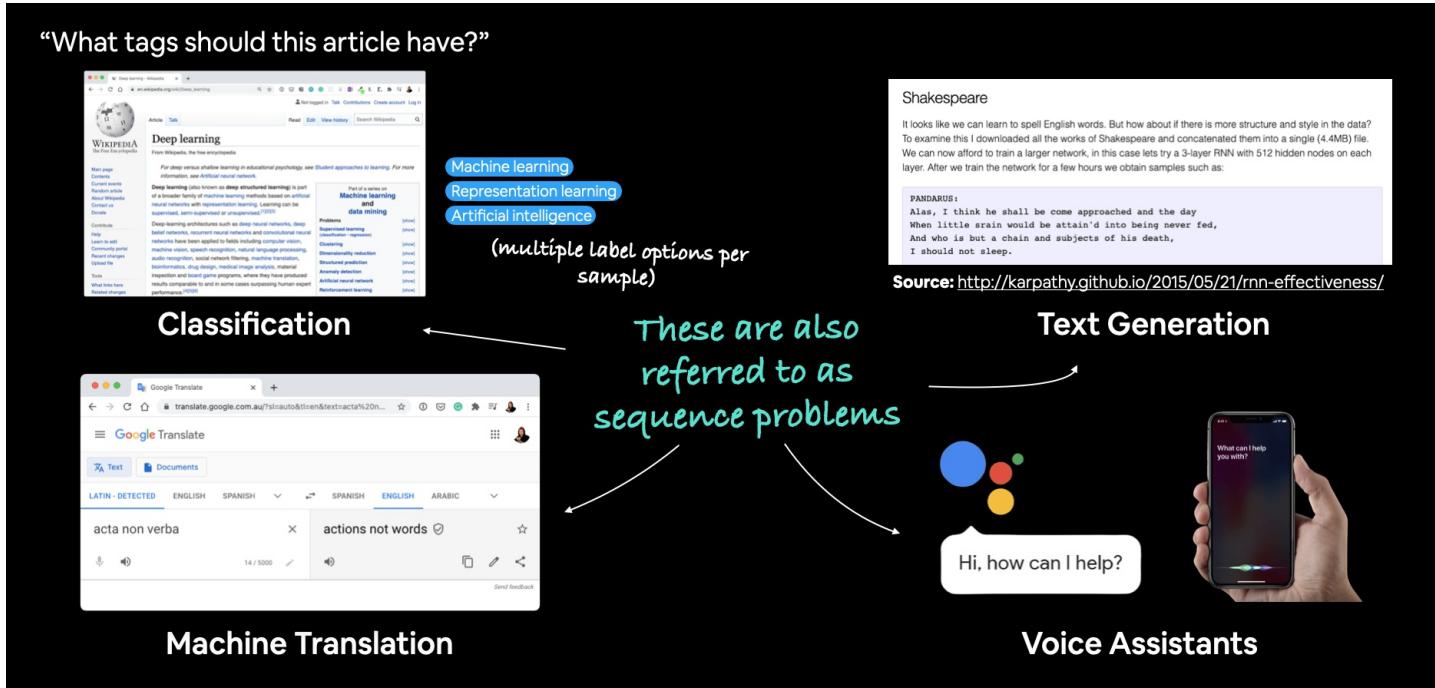
## Exercises

1. Use the same evaluation techniques on the large-scale Food Vision model as you did in the previous notebook ([Transfer Learning Part 3: Scaling up](#)). More specifically, it would be good to see:
  - A confusion matrix between all of the model's predictions and true labels.
  - A graph showing the f1-scores of each class.
  - A visualization of the model making predictions on various images and comparing the predictions to the ground truth.
    - For example, plot a sample image from the test dataset and have the title of the plot show the prediction, the prediction probability and the ground truth label.
2. Take 3 of your own photos of food and use the Food Vision model to make predictions on them. How does it go? Share your images/predictions with the other students.
3. Retrain the model (feature extraction and fine-tuning) we trained in this notebook, except this time use [EfficientNetB4](#) as the base model instead of [EfficientNetB0](#). Do you notice an improvement in performance? Does it take longer to train? Are there any tradeoffs to consider?
4. Name one important benefit of mixed precision training, how does this benefit take place?

## Extra-curriculum

- Read up on learning rate scheduling and the [learning rate scheduler callback](#). What is it? And how might it be helpful to this project?
- Read up on TensorFlow data loaders ([improving TensorFlow data loading performance](#)). Is there anything we've missed? What methods you keep in mind whenever loading data in TensorFlow? Hint: check the summary at the bottom of the page for a great round up of ideas.
- Read up on the documentation for [TensorFlow mixed precision training](#). What are the important things to keep in mind when using mixed precision training?

## 08. Natural Language Processing with TensorFlow



A handful of example natural language processing (**NLP**) and natural language understanding (**NLU**) problems. These are also often referred to as sequence problems (going from one sequence to another).

The main goal of [natural language processing \(NLP\)](#) is to derive information from natural language.

Natural language is a broad term but you can consider it to cover any of the following:

- Text (such as that contained in an email, blog post, book, Tweet)
- Speech (a conversation you have with a doctor, voice commands you give to a smart speaker)

Under the umbrellas of text and speech there are many different things you might want to do.

If you're building an email application, you might want to scan incoming emails to see if they're spam or not spam (classification).

If you're trying to analyse customer feedback complaints, you might want to discover which section of your business they're for.

**Note:** Both of these types of data are often referred to as *sequences* (a sentence is a sequence of words). So a common term you'll come across in NLP problems is called *seq2seq*, in other words, finding information in one sequence to produce another sequence (e.g. converting a speech command to a sequence of text-based steps).

To get hands-on with NLP in TensorFlow, we're going to practice the steps we've used previously but this time with text data:

Text -> turn into numbers -> build a model -> train the model to find patterns -> use patterns (make predictions)

**Resource:** For a great overview of NLP and the different problems within it, read the article [A Simple Introduction to Natural Language Processing](#).

### What we're going to cover

Let's get specific hey?

- Downloading a text dataset
- Visualizing text data
- Converting text into numbers using tokenization
- Turning our tokenized text into an embedding
- Modelling a text dataset
  - Starting with a baseline (TF-IDF)
  - Building several deep learning text models
    - Dense, LSTM, GRU, Conv1D, Transfer learning
- Comparing the performance of each our models
- Combining our models into an ensemble
- Saving and loading a trained model
- Find the most wrong predictions

## How you should approach this notebook

You can read through the descriptions and the code (it should all run, except for the cells which error on purpose), but there's a better option.

Write all of the code yourself.

Yes. I'm serious. Create a new notebook, and rewrite each line by yourself. Investigate it, see if you can break it, why does it break?

You don't have to write the text descriptions but writing the code yourself is a great way to get hands-on experience.

Don't worry if you make mistakes, we all do. The way to get better and make less mistakes is to write more code.

 **Resource:** See the full set of course materials on GitHub:  
<https://github.com/mrdbourke/tensorflow-deep-learning>

## Check for GPU

In order for our deep learning models to run as fast as possible, we'll need access to a GPU.

In Google Colab, you can set this up by going to Runtime -> Change runtime type -> Hardware accelerator -> GPU.

After selecting GPU, you may have to restart the runtime.

In [1]:

```
# Check for GPU
!nvidia-smi -L
```

GPU 0: Tesla K80 (UUID: GPU-7c8181f1-42c3-e0c6-0862-932bb75fde7b)

## Get helper functions

In past modules, we've created a bunch of helper functions to do small tasks required for our notebooks.

Rather than rewrite all of these, we can import a script and load them in from there.

The script containing our helper functions can be [found on GitHub](#).

In [2]:

```
# Download helper functions script
!wget https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/extras/helper_functions.py
```

```
--2021-09-23 05:25:54-- https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/extras/helper_functions.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com) ... 185.199.108.133, 185.199.110.133, 185.199.111.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443...
.. connected.
HTTP request sent, awaiting response... 200 OK
Length: 10246 (10K) [text/plain]
Saving to: 'helper_functions.py.4'

helper_functions.py 100%[=====] 10.01K --.-KB/s in 0s

2021-09-23 05:25:54 (43.6 MB/s) - 'helper_functions.py.4' saved [10246/10246]
```

In [3]:

```
# Import series of helper functions for the notebook
from helper_functions import unzip_data, create_tensorboard_callback, plot_loss_curves,
compare_histories
```

## Download a text dataset

Let's start by download a text dataset. We'll be using the [Real or Not?](#) dataset from Kaggle which contains text-based Tweets about natural disasters.

The Real Tweets are actually about diasters, for example:

Jetstar and Virgin forced to cancel Bali flights again because of ash from Mount Raung volcano

The Not Real Tweets are Tweets not about diasters (they can be on anything), for example:

'Education is the most powerful weapon which you can use to change the world.' Nelson Mandela #quote

For convenience, the dataset has been [downloaded from Kaggle](#) (doing this requires a Kaggle account) and uploaded as a downloadable zip file.

**Note:** The original downloaded data has not been altered to how you would download it from Kaggle.

In [4]:

```
# Download data (same as from Kaggle)
!wget "https://storage.googleapis.com/ztm_tf_course/nlp_getting_started.zip"

# Unzip data
unzip_data("nlp_getting_started.zip")

--2021-09-23 05:25:57-- https://storage.googleapis.com/ztm_tf_course/nlp_getting_started.zip
Resolving storage.googleapis.com (storage.googleapis.com) ... 66.102.1.128, 172.253.120.128, 74.125.206.128, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|66.102.1.128|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 607343 (593K) [application/zip]
Saving to: 'nlp_getting_started.zip.4'

nlp_getting_started 100%[=====] 593.11K --.-KB/s in 0.006s

2021-09-23 05:25:57 (101 MB/s) - 'nlp_getting_started.zip.4' saved [607343/607343]
```

Unzipping nlp\_getting\_started.zip gives the following 3 .csv files:

- `sample_submission.csv` - an example of the file you'd submit to the Kaggle competition of your model's predictions.
- `train.csv` - training samples of real and not real disaster Tweets.
- `test.csv` - testing samples of real and not real disaster Tweets.

## Visualizing a text dataset

Once you've acquired a new dataset to work with, what should you do first?

Explore it? Inspect it? Verify it? Become one with it?

All correct.

Remember the motto: visualize, visualize, visualize.

Right now, our text data samples are in the form of `.csv` files. For an easy way to make them visual, let's turn them into pandas DataFrame's.

**Reading:** You might come across text datasets in many different formats. Aside from CSV files (what we're working with), you'll probably encounter `.txt` files and `.json` files too. For working with these type of files, I'd recommend reading the two following articles by RealPython:

- [How to Read and Write Files in Python](#)
- [Working with JSON Data in Python](#)

In [5]:

```
# Turn .csv files into pandas DataFrame's
import pandas as pd
train_df = pd.read_csv("train.csv")
test_df = pd.read_csv("test.csv")
train_df.head()
```

Out [5]:

	<b>id</b>	<b>keyword</b>	<b>location</b>	<b>text</b>	<b>target</b>
0	1	NaN	NaN	Our Deeds are the Reason of this #earthquake M...	1
1	4	NaN	NaN	Forest fire near La Ronge Sask. Canada	1
2	5	NaN	NaN	All residents asked to 'shelter in place' are ...	1
3	6	NaN	NaN	13,000 people receive #wildfires evacuation or...	1
4	7	NaN	NaN	Just got sent this photo from Ruby #Alaska as ...	1

The training data we downloaded is probably shuffled already. But just to be sure, let's shuffle it again.

In [6]:

```
# Shuffle training dataframe
train_df_shuffled = train_df.sample(frac=1, random_state=42) # shuffle with random_state =42 for reproducibility
train_df_shuffled.head()
```

Out [6]:

	<b>id</b>	<b>keyword</b>	<b>location</b>	<b>text</b>	<b>target</b>
2644	3796	destruction	NaN	So you have a new weapon that can cause un-ima...	1
2227	3185	deluge	NaN	The f\$&@ing things I do for #GISHWHES Just...	0
5448	7769	police	UK	DT @georgegalloway: RT @Galloway4Mayor: ÜThe...	1
132	191	aftershock	NaN	Aftershock back to school kick off was great. ...	0

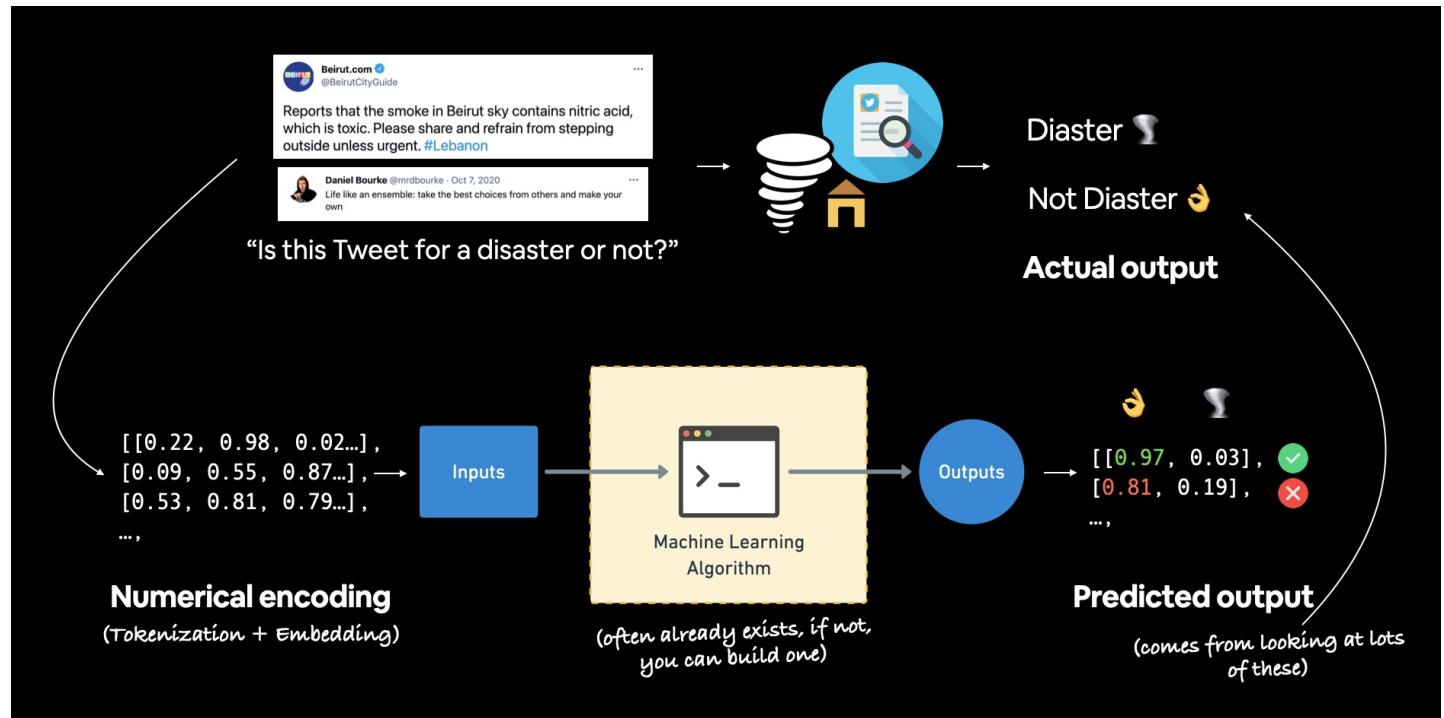
<b>id</b>	<b>keyword</b>	<b>location</b>	<b>text</b>	<b>target</b>
6845	9810	trauma	Montgomery County, MD	in response to trauma Children of Addicts deve...

Notice how the training data has a "target" column.

We're going to be writing code to find patterns (e.g. different combinations of words) in the "text" column of the training dataset to predict the value of the "target" column.

The test dataset doesn't have a "target" column.

Inputs (text column) -> Machine Learning Algorithm -> Outputs (target column)



Example text classification inputs and outputs for the problem of classifying whether a Tweet is about a disaster or not.

In [7]:

```
# The test data doesn't have a target (that's what we'd try to predict)
test_df.head()
```

Out [7]:

<b>id</b>	<b>keyword</b>	<b>location</b>	<b>text</b>
0	0	NaN	Just happened a terrible car crash
1	2	NaN	Heard about #earthquake is different cities, s...
2	3	NaN	there is a forest fire at spot pond, geese are...
3	9	NaN	Apocalypse lighting. #Spokane #wildfires
4	11	NaN	Typhoon Soudelor kills 28 in China and Taiwan

Let's check how many examples of each target we have.

In [8]:

```
# How many examples of each class?
train_df.target.value_counts()
```

Out [8]:

```
0      4342
1      3271
Name: target, dtype: int64
```

Since we have two target values, we're dealing with a **binary classification** problem.

It's fairly balanced too, about 60% negative class (`target = 0`) and 40% positive class (`target = 1`).

Where,

- `1` = a real disaster Tweet
- `0` = not a real disaster Tweet

And what about the total number of samples we have?

In [9]:

```
# How many samples total?
print(f"Total training samples: {len(train_df)}")
print(f"Total test samples: {len(test_df)}")
print(f"Total samples: {len(train_df) + len(test_df)}")
```

```
Total training samples: 7613
Total test samples: 3263
Total samples: 10876
```

Alright, seems like we've got a decent amount of training and test data. If anything, we've got an abundance of testing examples, usually a split of 90/10 (90% training, 10% testing) or 80/20 is suffice.

Okay, time to visualize, let's write some code to visualize random text samples.

□ **Question:** Why visualize random samples? You could visualize samples in order but this could lead to only seeing a certain subset of data. Better to visualize a substantial quantity (100+) of random samples to get an idea of the different kinds of data you're working with. In machine learning, never underestimate the power of randomness.

In [10]:

```
# Let's visualize some random training examples
import random
random_index = random.randint(0, len(train_df)-5) # create random indexes not higher than the total number of samples
for row in train_df_shuffled[["text", "target"]][random_index:random_index+5].itertuples():
    _, text, target = row
    print(f"Target: {target}, {(real disaster)" if target > 0 else "(not real disaster)"})
    print(f"Text:\n{text}\n")
    print("---\n")
```

```
Target: 0 (not real disaster)
Text:
https://t.co/eCMUjkKqX1 @ArianaGrande @ScreamQueens
Katherine's Death

---
Target: 0 (not real disaster)
Text:
@TinyJecht Are you another Stand-user? If you are I will have to detonate you with my Killer Queen.

---
Target: 1 (real disaster)
Text:
70 Years After Atomic Bombs Japan Still Struggles With War Past http://t.co/5wfXbAQMBK The anniversary of the devastation wrought by the U_
```

My lifelong all-time favorite song is 'Landslide'. This song has gotten me through a lot of though times &hellip; http://t.co/RfB3JXbiEJ

---

Target: 0 (not real disaster)

Text:

I hear the mumbling i hear the cackling i got em scared shook panicking

---

## Split data into training and validation sets

Since the test set has no labels and we need a way to evalaute our trained models, we'll split off some of the training data and create a validation set.

When our model trains (tries patterns in the Tweet samples), it'll only see data from the training set and we can see how it performs on unseen data using the validation set.

We'll convert our splits from pandas Series datatypes to lists of strings (for the text) and lists of ints (for the labels) for ease of use later.

To split our training dataset and create a validation dataset, we'll use Scikit-Learn's `train_test_split()` method and dedicate 10% of the training samples to the validation set.

In [11]:

```
from sklearn.model_selection import train_test_split

# Use train_test_split to split training data into training and validation sets
train_sentences, val_sentences, train_labels, val_labels = train_test_split(train_df_shuffled["text"].to_numpy(),
                                                                           train_df_shuffled["target"].to_numpy(),
                                                                           test_size=0.1, # dedicate 10% of samples to validation set
                                                                           random_state=42) # random state for reproducibility
```

In [12]:

```
# Check the lengths
len(train_sentences), len(train_labels), len(val_sentences), len(val_labels)
```

Out[12]:

(6851, 6851, 762, 762)

In [13]:

```
# View the first 10 training sentences and their labels
train_sentences[:10], train_labels[:10]
```

Out[13]:

```
(array(['@mogacola @zamtriossu i screamed after hitting tweet',
       'Imagine getting flattened by Kurt Zouma',
       '@Gurmeetramrahim #MSGDoing111WelfareWorks Green S welfare force ke appx 65000 members har time disaster victim ki help ke liye tyar hai....',
       "@shakjn @C7 @Magnums im shaking in fear he's gonna hack the planet",
       'Somehow find you and I collide http://t.co/Ee8RpOahPk',
       '@EvaHanderek @MarleyKnysh great times until the bus driver held us hostage in the mall parking lot lmfao',
       'destroy the free fandom honestly',
       'Weapons stolen from National Guard Armory in New Albany still missing #Gunsense http://t.co/lKNU8902JE',
       '@wfaawebat Pete when will the heat wave pass? Is it really going to be mid month? Frisco Boy Scouts have a canoe trip in Okla.',
       'Patient-reported outcomes in long-term survivors of metastatic colorectal cancer',
       'DRAFT - TUTORIAL OF SURVEYING WITH / / --EV1 ADG1MUL1'])
```

```
- BRITISH JOURNAL OF SURGERY URL: //WWW.BJS.CO.UK/2014/01/01/101111  
dtype=object), array([0, 0, 1, 0, 0, 1, 1, 0, 1, 1]))
```

## Converting text into numbers

Wonderful! We've got a training set and a validation set containing Tweets and labels.

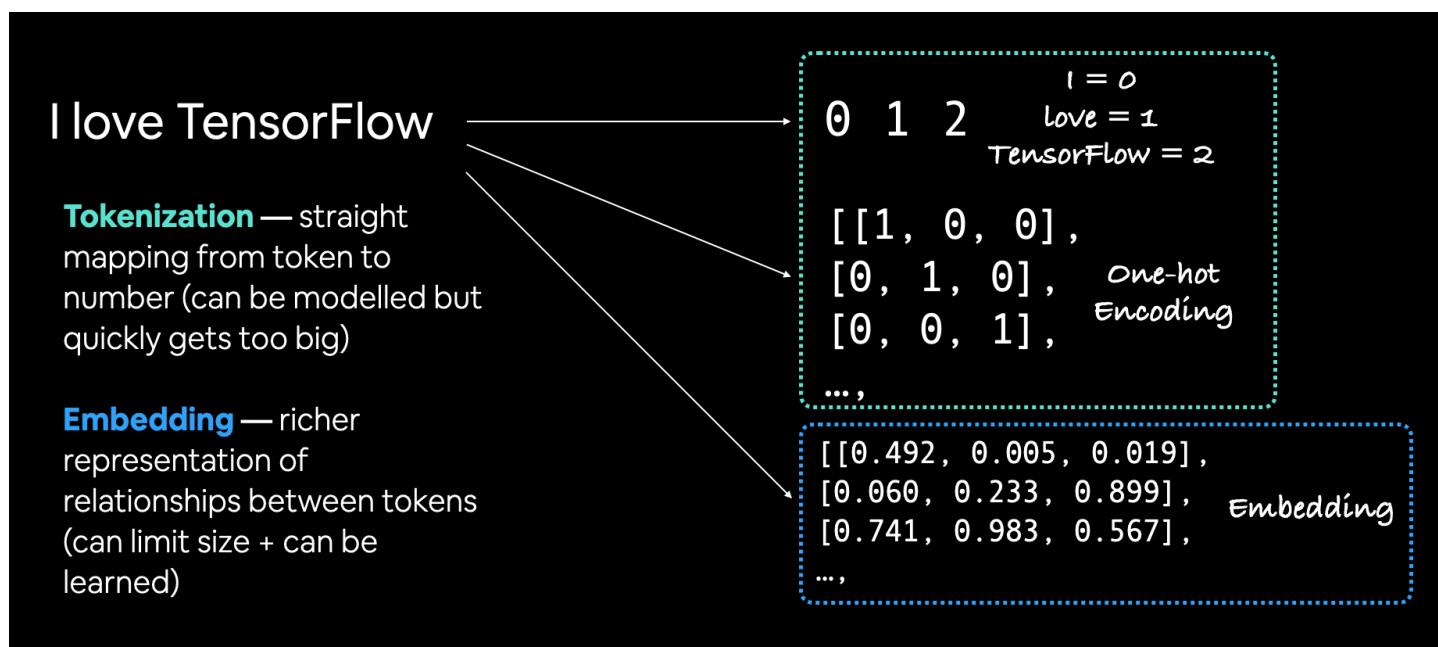
Our labels are in numerical form ( 0 and 1 ) but our Tweets are in string form.

Question: What do you think we have to do before we can use a machine learning algorithm with our text data?

If you answered something along the lines of "turn it into numbers", you're correct. A machine learning algorithm requires its inputs to be in numerical form.

In NLP, there are two main concepts for turning text into numbers:

- **Tokenization** - A straight mapping from word or character or sub-word to a numerical value. There are three main levels of tokenization:
  1. Using **word-level tokenization** with the sentence "I love TensorFlow" might result in "I" being 0 , "love" being 1 and "TensorFlow" being 2 . In this case, every word in a sequence considered a single **token**.
  2. **Character-level tokenization**, such as converting the letters A-Z to values 1-26 . In this case, every character in a sequence considered a single **token**.
  3. **Sub-word tokenization** is in between word-level and character-level tokenization. It involves breaking individual words into smaller parts and then converting those smaller parts into numbers. For example, "my favourite food is pineapple pizza" might become "my, fav, avour, rite, fo, oo, od, is, pin, ine, app, le, piz, za" . After doing this, these sub-words would then be mapped to a numerical value. In this case, every word could be considered multiple **tokens**.
- **Embeddings** - An embedding is a representation of natural language which can be learned. Representation comes in the form of a **feature vector**. For example, the word "dance" could be represented by the 5-dimensional vector [-0.8547, 0.4559, -0.3332, 0.9877, 0.1112] . It's important to note here, the size of the feature vector is tuneable. There are two ways to use embeddings:
  1. **Create your own embedding** - Once your text has been turned into numbers (required for an embedding), you can put them through an embedding layer (such as `tf.keras.layers.Embedding`) and an embedding representation will be learned during model training.
  2. **Reuse a pre-learned embedding** - Many pre-trained embeddings exist online. These pre-trained embeddings have often been learned on large corpuses of text (such as all of Wikipedia) and thus have a good underlying representation of natural language. You can use a pre-trained embedding to initialize your model and fine-tune it to your own specific task.



Example of tokenization (straight mapping from word to number) and embedding (richer representation of relationships between tokens)

## Question: What level of tokenization should I use? What embedding should I choose?

It depends on your problem. You could try character-level tokenization/embeddings and word-level tokenization/embeddings and see which perform best. You might even want to try stacking them (e.g. combining the outputs of your embedding layers using `tf.keras.layers.concatenate`).

If you're looking for pre-trained word embeddings, [Word2vec embeddings](#), [GloVe embeddings](#) and many of the options available on [TensorFlow Hub](#) are great places to start.

**Note:** Much like searching for a pre-trained computer vision model, you can search for pre-trained word embeddings to use for your problem. Try searching for something like "use pre-trained word embeddings in TensorFlow".

## Text vectorization (tokenization)

Enough talking about tokenization and embeddings, let's create some.

We'll practice tokenization (mapping our words to numbers) first.

To tokenize our words, we'll use the helpful preprocessing layer

`tf.keras.layers.experimental.preprocessing.TextVectorization`.

The `TextVectorization` layer takes the following parameters:

- `max_tokens` - The maximum number of words in your vocabulary (e.g. 20000 or the number of unique words in your text), includes a value for OOV (out of vocabulary) tokens.
- `standardize` - Method for standardizing text. Default is `"lower_and_strip_punctuation"` which lowers text and removes all punctuation marks.
- `split` - How to split text, default is `"whitespace"` which splits on spaces.
- `ngrams` - How many words to contain per token split, for example, `ngrams=2` splits tokens into continuous sequences of 2.
- `output_mode` - How to output tokens, can be `"int"` (integer mapping), `"binary"` (one-hot encoding), `"count"` or `"tf-idf"`. See documentation for more.
- `output_sequence_length` - Length of tokenized sequence to output. For example, if `output_sequence_length=150`, all tokenized sequences will be 150 tokens long.
- `pad_to_max_tokens` - Defaults to `False`, if `True`, the output feature axis will be padded to `max_tokens` even if the number of unique tokens in the vocabulary is less than `max_tokens`. Only valid in certain modes, see docs for more.

Let's see it in action.

In [14]:

```
import tensorflow as tf
from tensorflow.keras.layers.experimental.preprocessing import TextVectorization
# Note: in TensorFlow 2.6+, you no longer need "layers.experimental.preprocessing"
# you can use: "tf.keras.layers.TextVectorization", see https://github.com/tensorflow/tensorflow/releases/tag/v2.6.0 for more

# Use the default TextVectorization variables
text_vectorizer = TextVectorization(max_tokens=None, # how many words in the vocabulary (all of the different words in your text)
                                     standardize="lower_and_strip_punctuation", # how to process text
                                     split="whitespace", # how to split tokens
                                     ngrams=None, # create groups of n-words?
                                     output_mode="int", # how to map tokens to numbers
                                     output_sequence_length=None) # how long should the output sequence of tokens be?

# pad_to_max_tokens=True) # Not valid if using max_t
```

`okens=None`

We've initialized a `TextVectorization` object with the default settings but let's customize it a little bit for our own use case.

In particular, let's set values for `max_tokens` and `output_sequence_length`.

For `max_tokens` (the number of words in the vocabulary), multiples of 10,000 (`10,000`, `20,000`, `30,000`) or the exact number of unique words in your text (e.g. `32,179`) are common values.

For our use case, we'll use `10,000`.

And for the `output_sequence_length` we'll use the average number of tokens per Tweet in the training set. But first, we'll need to find it.

In [15]:

```
# Find average number of tokens (words) in training Tweets
round(sum([len(i.split()) for i in train_sentences])/len(train_sentences))
```

Out[15]:

15

Now let's create another `TextVectorization` object using our custom parameters.

In [16]:

```
# Setup text vectorization with custom variables
max_vocab_length = 10000 # max number of words to have in our vocabulary
max_length = 15 # max length our sequences will be (e.g. how many words from a Tweet does our model see?)

text_vectorizer = TextVectorization(max_tokens=max_vocab_length,
                                     output_mode="int",
                                     output_sequence_length=max_length)
```

Beautiful!

To map our `TextVectorization` instance `text_vectorizer` to our data, we can call the `adapt()` method on it whilst passing it our training text.

In [17]:

```
# Fit the text vectorizer to the training text
text_vectorizer.adapt(train_sentences)
```

Training data mapped! Let's try our `text_vectorizer` on a custom sentence (one similar to what you might see in the training data).

In [18]:

```
# Create sample sentence and tokenize it
sample_sentence = "There's a flood in my street!"
text_vectorizer([sample_sentence])
```

Out[18]:

```
<tf.Tensor: shape=(1, 15), dtype=int64, numpy=
array([[264,    3, 232,    4,   13, 698,    0,    0,    0,    0,    0,
       0,    0]])>
```

Wonderful, it seems we've got a way to turn our text into numbers (in this case, word-level tokenization). Notice the 0's at the end of the returned tensor, this is because we set `output_sequence_length=15`, meaning no matter the size of the sequence we pass to `text_vectorizer`, it always returns a sequence with a length of 15.

How about we try our `text_vectorizer` on a few random sentences?

In [19]:

```
# Choose a random sentence from the training dataset and tokenize it
random_sentence = random.choice(train_sentences)
print(f"Original text:\n{random_sentence}\n\nVectorized version:")
text_vectorizer([random_sentence])
```

Original text:

Black Eye 9: A space battle occurred at Star 0784 involving 3 fleets totaling 3942 ships with 14 destroyed

Vectorized version:

Out[19]:

```
<tf.Tensor: shape=(1, 15), dtype=int64, numpy=
array([[ 159,  898,  491,     3,  759,  442, 1068,    17,   874, 1629, 1129,
       118, 1524, 1457, 6327]])>
```

Looking good!

Finally, we can check the unique tokens in our vocabulary using the `get_vocabulary()` method.

In [20]:

```
# Get the unique words in the vocabulary
words_in_vocab = text_vectorizer.get_vocabulary()
top_5_words = words_in_vocab[:5] # most common tokens (notice the [UNK] token for "unknown" words)
bottom_5_words = words_in_vocab[-5:] # least common tokens
print(f"Number of words in vocab: {len(words_in_vocab)}")
print(f"Top 5 most common words: {top_5_words}")
print(f"Bottom 5 least common words: {bottom_5_words}")
```

Number of words in vocab: 10000

Top 5 most common words: ['', '[UNK]', 'the', 'a', 'in']

Bottom 5 least common words: ['pages', 'paeds', 'pads', 'padres', 'paddytomlinson1']

## Creating an Embedding using an Embedding Layer

We've got a way to map our text to numbers. How about we go a step further and turn those numbers into an embedding?

The powerful thing about an embedding is it can be learned during training. This means rather than just being static (e.g. 1 = I, 2 = love, 3 = TensorFlow), a word's numeric representation can be improved as a model goes through data samples.

We can see what an embedding of a word looks like by using the `tf.keras.layers.Embedding` layer.

The main parameters we're concerned about here are:

- `input_dim` - The size of the vocabulary (e.g. `len(text_vectorizer.get_vocabulary())`).
- `output_dim` - The size of the output embedding vector, for example, a value of 100 outputs a feature vector of size 100 for each word.
- `embeddings_initializer` - How to initialize the embeddings matrix, default is "uniform" which randomly initializes embedding matrix with uniform distribution. This can be changed for using pre-learned embeddings.
- `input_length` - Length of sequences being passed to embedding layer.

Knowing these, let's make an embedding layer.

In [21]:

```
tr.random.set_seed(42)
from tensorflow.keras import layers

embedding = layers.Embedding(input_dim=max_vocab_length, # set input shape
                             output_dim=128, # set size of embedding vector
                             embeddings_initializer="uniform", # default, initialize randomly
                             input_length=max_length, # how long is each input
                             name="embedding_1")

embedding
```

Out [21]:

```
<keras.layers.embeddings.Embedding at 0x7f5eb57cee50>
```

**Excellent, notice how `embedding` is a TensorFlow layer? This is important because we can use it as part of a model, meaning its parameters (word representations) can be updated and improved as the model learns.**

**How about we try it out on a sample sentence?**

In [22]:

```
# Get a random sentence from training set
random_sentence = random.choice(train_sentences)
print(f"Original text:\n{random_sentence}\n\nEmbedded version:")

# Embed the random sentence (turn it into numerical representation)
sample_embed = embedding(text_vectorizer([random_sentence]))
```

Original text:

```
UNR issues Severe Thunderstorm Warning [wind: 60 MPH hail: 0.75 IN] for Weston [WY] and Custer Fall River Lawrence Meade Pennington [SD]
```

Embedded version:

Out [22]:

```
<tf.Tensor: shape=(1, 15, 128), dtype=float32, numpy=
array([[[ -0.0235487 , -0.02911143, -0.01600165, ..., -0.03135703,
       -0.03661771,  0.02812702],
       [ 0.03848192, -0.04844777,  0.0345685 , ..., -0.01402212,
       -0.04769395, -0.0404261 ],
       [-0.04240407,  0.04235772,  0.04278237, ..., -0.01318695,
        0.04171768,  0.01777187],
       ...,
       [-0.00522641,  0.04871375, -0.03742788, ...,  0.00540795,
       -0.04380312, -0.01817607],
       [ 0.03406706, -0.00160446,  0.00894339, ..., -0.0356751 ,
       0.00541915,  0.00282475],
       [ 0.02248487, -0.02848336,  0.04786098, ...,  0.03069806,
      -0.04317403, -0.04145076]]], dtype=float32)>
```

**Each token in the sentence gets turned into a length 128 feature vector.**

In [23]:

```
# Check out a single token's embedding
sample_embed[0][0]
```

Out [23]:

```
<tf.Tensor: shape=(128,), dtype=float32, numpy=
array([-2.3548698e-02, -2.9111434e-02, -1.6001653e-02, -2.4854636e-02,
       -1.8839194e-02, -1.7742872e-02,  3.5991777e-02,  4.7734752e-03,
       -3.1464353e-02, -1.0192860e-02,  3.9272513e-02,  1.6213503e-02,
       3.5752203e-02, -7.9760700e-04, -4.6503343e-02,  4.1901264e-02,
       2.7158771e-02, -2.9694129e-02,  6.3859299e-04, -3.6073186e-02,
       3.7186686e-02,  8.1444494e-03, -3.4610189e-02, -1.2373447e-02,
       3.3506799e-02,  3.4542195e-03, -3.4555770e-02,  3.0121803e-03,
```

```

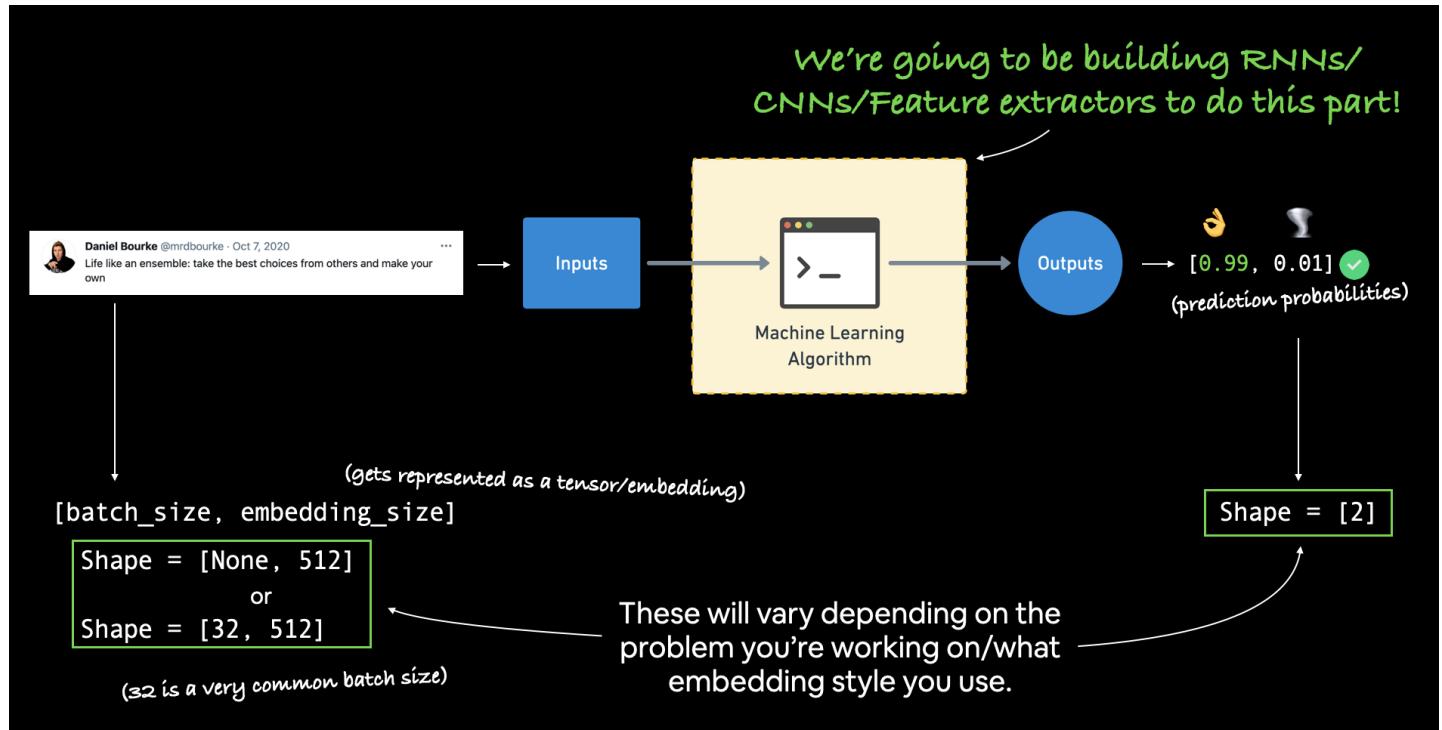
1.2546945e-02, 1.8180419e-02, -2.8727353e-02, 3.0131452e-03,
2.2011306e-02, 1.5216086e-02, 8.3960593e-05, -4.9976040e-02,
-4.1987814e-02, -1.4751814e-02, 3.1978119e-02, 3.0810181e-02,
1.3748173e-02, 1.3646554e-02, -1.8768311e-03, 5.6033619e-03,
-3.2450367e-02, -3.2819200e-02, 6.4723380e-03, 2.4402250e-02,
-4.9929023e-02, 8.7605603e-03, 3.7449453e-02, -3.0369056e-02,
2.8607275e-02, -8.9427829e-03, -2.6780851e-03, 1.9382443e-02,
-4.4139970e-02, -4.8123684e-02, 3.2326613e-02, 1.0355391e-02,
-6.2159896e-03, 3.3066813e-02, 4.1976977e-02, 9.8001361e-03,
9.7909793e-03, 1.8213544e-02, 1.6274918e-02, -1.7997943e-02,
1.4698040e-02, 1.0068141e-02, -2.3385560e-02, 1.7339502e-02,
3.5935570e-02, -4.9711645e-02, 3.2845590e-02, 3.8101044e-02,
3.9486382e-02, -3.1647660e-02, -4.8475552e-02, 4.4873584e-02,
2.7549271e-02, -4.1145109e-02, -3.3895336e-02, -3.6730655e-03,
4.9198270e-03, 9.6562132e-03, -2.2904599e-02, -1.3657093e-02,
1.5388299e-02, 8.1878789e-03, 1.8028166e-02, 3.1150069e-02,
4.7483686e-02, -3.7815310e-02, -4.5389161e-03, 4.1796099e-02,
4.3265197e-02, 3.1167094e-02, -4.9614847e-02, -5.8911927e-03,
4.3997217e-02, -2.2734845e-02, -4.1017674e-02, 1.7939974e-02,
2.3607183e-02, 1.5478458e-02, 7.7072531e-04, -4.3312550e-02,
-4.2333078e-02, -2.2680223e-02, 3.2546792e-02, -4.9846746e-02,
1.3042022e-02, -3.2268692e-02, -1.8501390e-02, 5.7965517e-03,
-6.9886930e-03, -1.9324971e-02, -4.5883238e-02, 3.7569497e-02,
1.4392149e-02, -1.0649189e-03, 2.4147406e-03, -2.7852738e-02,
-3.7008919e-02, -3.1357028e-02, -3.6617707e-02, 2.8127018e-02],
dtype=float32)>

```

**These values might not mean much to us but they're what our computer sees each word as. When our model looks for patterns in different samples, these values will be updated as necessary.**

**Note:** The previous two concepts (tokenization and embeddings) are the foundation for many NLP tasks. So if you're not sure about anything, be sure to research and conduct your own experiments to further help your understanding.

## Modelling a text dataset



Once you've got your inputs and outputs prepared, it's a matter of figuring out which machine learning model to build in between them to bridge the gap.

Now that we've got a way to turn our text data into numbers, we can start to build machine learning models to model it.

To get plenty of practice, we're going to build a series of different models, each as its own experiment. We'll

then compare the results of each model and see which one performed best.

**More specifically, we'll be building the following:**

- **Model 0:** Naive Bayes (baseline)
  - **Model 1:** Feed-forward neural network (dense model)
  - **Model 2:** LSTM model
  - **Model 3:** GRU model
  - **Model 4:** Bidirectional-LSTM model
  - **Model 5:** 1D Convolutional Neural Network
  - **Model 6:** TensorFlow Hub Pretrained Feature Extractor
  - **Model 7:** Same as model 6 with 10% of training data

**Model 0 is the simplest to acquire a baseline which we'll expect each other of the other deeper models to beat.**

**Each experiment will go through the following steps:**

- Construct the model
  - Train the model
  - Make predictions with the model
  - Track prediction evaluation metrics for later comparison

## Let's get started.

## Model 0: Getting a baseline

As with all machine learning modelling experiments, it's important to create a baseline model so you've got a benchmark for future experiments to build upon.

To create our baseline, we'll create a Scikit-Learn Pipeline using the TF-IDF (term frequency-inverse document frequency) formula to convert our words to numbers and then model them with the [Multinomial Naive Bayes algorithm](#). This was chosen via referring to the [Scikit-Learn machine learning map](#).

¶ **Reading:** The ins and outs of TF-IDF algorithm is beyond the scope of this notebook, however, the curious reader is encouraged to check out the [Scikit-Learn documentation for more](#).

In [24]:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline

# Create tokenization and modelling pipeline
model_0 = Pipeline([
    ("tfidf", TfidfVectorizer()), # convert words to numbers using tfidf
    ("clf", MultinomialNB()) # model the text
])

# Fit the pipeline to the training data
model_0.fit(train_sentences, train_labels)
```

Out [24] :

```
vocabulary=None)),  
        ('clf',  
         MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True))),  
        verbose=False)
```

The benefit of using a shallow model like Multinomial Naive Bayes is that training is very fast.

Let's evaluate our model and find our baseline metric.

In [25]:

```
baseline_score = model_0.score(val_sentences, val_labels)  
print(f"Our baseline model achieves an accuracy of: {baseline_score*100:.2f}%")
```

Our baseline model achieves an accuracy of: 79.27%

How about we make some predictions with our baseline model?

In [26]:

```
# Make predictions  
baseline_preds = model_0.predict(val_sentences)  
baseline_preds[:20]
```

Out[26]:

```
array([1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1])
```

## Creating an evaluation function for our model experiments

We could evaluate these as they are but since we're going to be evaluating several models in the same way going forward, let's create a helper function which takes an array of predictions and ground truth labels and computes the following:

- Accuracy
- Precision
- Recall
- F1-score

**Note:** Since we're dealing with a classification problem, the above metrics are the most appropriate. If we were working with a regression problem, other metrics such as MAE (mean absolute error) would be a better choice.

In [27]:

```
# Function to evaluate: accuracy, precision, recall, f1-score  
from sklearn.metrics import accuracy_score, precision_recall_fscore_support  
  
def calculate_results(y_true, y_pred):  
    """  
    Calculates model accuracy, precision, recall and f1 score of a binary classification model.  
  
    Args:  
    ----  
    y_true = true labels in the form of a 1D array  
    y_pred = predicted labels in the form of a 1D array  
  
    Returns a dictionary of accuracy, precision, recall, f1-score.  
    """  
    # Calculate model accuracy  
    model_accuracy = accuracy_score(y_true, y_pred) * 100  
    # Calculate model precision, recall and f1 score using "weighted" average  
    model_precision, model_recall, model_f1, _ = precision_recall_fscore_support(y_true, y_pred, average="weighted")  
    model_results = {"accuracy": model_accuracy,
```

```
        "precision": model_precision,  
        "recall": model_recall,  
        "f1": model_f1}  
    return model_results
```

In [28]:

```
# Get baseline results  
baseline_results = calculate_results(y_true=val_labels,  
                                      y_pred=baseline_preds)  
baseline_results
```

Out [28]:

```
{'accuracy': 79.26509186351706,  
'f1': 0.7862189758049549,  
'precision': 0.8111390004213173,  
'recall': 0.7926509186351706}
```

## Model 1: A simple dense model

The first "deep" model we're going to build is a single layer dense model. In fact, it's barely going to have a single layer.

It'll take our text and labels as input, tokenize the text, create an embedding, find the average of the embedding (using Global Average Pooling) and then pass the average through a fully connected layer with one output unit and a sigmoid activation function.

If the previous sentence sounds like a mouthful, it'll make sense when we code it out (remember, if in doubt, code it out).

And since we're going to be building a number of TensorFlow deep learning models, we'll import our `create_tensorboard_callback()` function from `helper_functions.py` to keep track of the results of each.

In [29]:

```
# Create tensorboard callback (need to create a new one for each model)  
from helper_functions import create_tensorboard_callback  
  
# Create directory to save TensorBoard logs  
SAVE_DIR = "model_logs"
```

Now we've got a TensorBoard callback function ready to go, let's build our first deep model.

In [30]:

```
# Build model with the Functional API  
from tensorflow.keras import layers  
inputs = layers.Input(shape=(1,), dtype="string") # inputs are 1-dimensional strings  
x = text_vectorizer(inputs) # turn the input text into numbers  
x = embedding(x) # create an embedding of the numerized numbers  
x = layers.GlobalAveragePooling1D()(x) # lower the dimensionality of the embedding (try running the model without this layer and see what happens)  
outputs = layers.Dense(1, activation="sigmoid")(x) # create the output layer, want binary outputs so use sigmoid activation  
model_1 = tf.keras.Model(inputs, outputs, name="model_1_dense") # construct the model
```

Looking good. Our model takes a 1-dimensional string as input (in our case, a Tweet), it then tokenizes the string using `text_vectorizer` and creates an embedding using `embedding`.

We then (optionally) pool the outputs of the embedding layer to reduce the dimensionality of the tensor we pass to the output layer.

Exercise: Try building `model_1` with and without a `GlobalAveragePooling1D()` layer after the `embedding` layer. What happens? Why do you think this is?

**Finally, we pass the output of the pooling layer to a dense layer with sigmoid activation (we use sigmoid since our problem is binary classification).**

**Before we can fit our model to the data, we've got to compile it. Since we're working with binary classification, we'll use "binary\_crossentropy" as our loss function and the Adam optimizer.**

In [31]:

```
# Compile model
model_1.compile(loss="binary_crossentropy",
                  optimizer=tf.keras.optimizers.Adam(),
                  metrics=["accuracy"])
```

**Model compiled. Let's get a summary.**

In [32]:

```
# Get a summary of the model
model_1.summary()
```

Model: "model\_1\_dense"

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[None, 1]	0
text_vectorization_1 (TextVe	(None, 15)	0
embedding_1 (Embedding)	(None, 15, 128)	1280000
global_average_pooling1d (Gl	(None, 128)	0
dense (Dense)	(None, 1)	129

Total params: 1,280,129  
Trainable params: 1,280,129  
Non-trainable params: 0

**Most of the trainable parameters are contained within the embedding layer. Recall we created an embedding of size 128 (`output_dim=128`) for a vocabulary of size 10,000 (`input_dim=10000`), hence the 1,280,000 trainable parameters.**

**Alright, our model is compiled, let's fit it to our training data for 5 epochs. We'll also pass our TensorBoard callback function to make sure our model's training metrics are logged.**

In [33]:

```
# Fit the model
model_1_history = model_1.fit(train_sentences, # input sentences can be a list of strings
                               due to text preprocessing layer built-in model
                               train_labels,
                               epochs=5,
                               validation_data=(val_sentences, val_labels),
                               callbacks=[create_tensorboard_callback(dir_name=SAVE_DIR,
                                                               experiment_name="simple_dense_model")])
```

Saving TensorBoard log files to: model\_logs/simple\_dense\_model/20210923-052559  
Epoch 1/5  
215/215 [=====] - 5s 17ms/step - loss: 0.6094 - accuracy: 0.6916  
- val\_loss: 0.5357 - val\_accuracy: 0.7572  
Epoch 2/5  
215/215 [=====] - 3s 13ms/step - loss: 0.4410 - accuracy: 0.8189  
- val\_loss: 0.4691 - val\_accuracy: 0.7848  
Epoch 3/5  
215/215 [=====] - 3s 13ms/step - loss: 0.3463 - accuracy: 0.8605  
- val\_loss: 0.4590 - val\_accuracy: 0.7900

```
Epoch 4/5
215/215 [=====] - 3s 14ms/step - loss: 0.2848 - accuracy: 0.8923
- val_loss: 0.4641 - val_accuracy: 0.7927
Epoch 5/5
215/215 [=====] - 3s 14ms/step - loss: 0.2380 - accuracy: 0.9118
- val_loss: 0.4767 - val_accuracy: 0.7874
```

Nice! Since we're using such a simple model, each epoch processes very quickly.

Let's check our model's performance on the validation set.

In [34]:

```
# Check the results
model_1.evaluate(val_sentences, val_labels)
```

```
24/24 [=====] - 0s 7ms/step - loss: 0.4767 - accuracy: 0.7874
```

Out [34]:

```
[0.4766846001148224, 0.787401556968689]
```

In [35]:

```
embedding.weights
```

Out [35]:

```
<tf.Variable 'embedding_1/embeddings:0' shape=(10000, 128) dtype=float32, numpy=
array([[ 0.00073166,  0.01504797, -0.03425457, ..., -0.04403538,
       -0.01042282,  0.01876436],
       [ 0.04135862, -0.03945084, -0.03811942, ...,  0.00464737,
       0.03163553,  0.029283 ],
       [ 0.00684031,  0.05363134, -0.00241555, ..., -0.07082176,
       -0.04750705,  0.01448254],
       ...,
       [-0.03301444, -0.0052493 , -0.04209725, ...,  0.02028764,
       0.00308807,  0.02215792],
       [ 0.00692343,  0.05942352, -0.01975194, ..., -0.06199061,
       -0.01018393,  0.03510419],
       [-0.0372346 ,  0.06267187, -0.07451146, ..., -0.02367217,
       -0.0864333 ,  0.01742156]], dtype=float32)>]
```

In [36]:

```
embed_weights = model_1.get_layer("embedding_1").get_weights()[0]
print(embed_weights.shape)
```

```
(10000, 128)
```

And since we tracked our model's training logs with TensorBoard, how about we visualize them?

We can do so by uploading our TensorBoard log files (contained in the `model_logs` directory) to [TensorBoard.dev](#).

**Note:** Remember, whatever you upload to TensorBoard.dev becomes public. If there are training logs you don't want to share, don't upload them.

In [37]:

```
# # View tensorboard logs of transfer learning modelling experiments (should be 4 models)
# # Upload TensorBoard dev records
# !tensorboard dev upload --logdir ./model_logs \
#   --name "First deep model on text data" \
#   --description "Trying a dense model with an embedding layer" \
#   --one_shot # exits the uploader when upload has finished
```

In [38]:

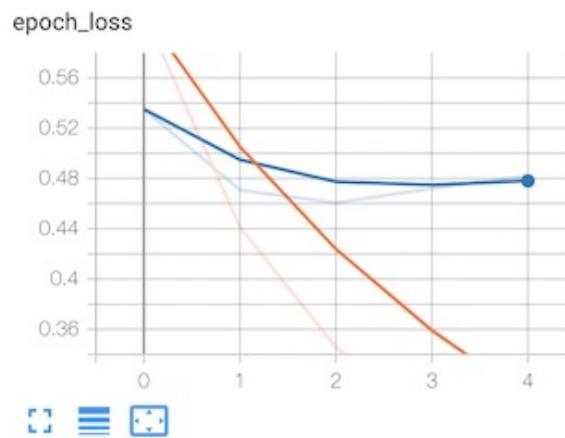
```
# If you need to remove previous experiments, you can do so using the following command
# !tensorboard dev delete --experiment_id EXPERIMENT_ID_TO_DELETE
```

The TensorBoard.dev experiment for our first deep model can be viewed here:

<https://tensorboard.dev/experiment/5d1Xm10aT6m6MgyW3HAGfw/>



epoch\_loss



What the training curves of our model look like on TensorBoard. From looking at the curves can you tell if the model is overfitting or underfitting?

Beautiful! Those are some colorful training curves. Would you say the model is overfitting or underfitting?

We've built and trained our first deep model, the next step is to make some predictions with it.

In [39]:

```
# Make predictions (these come back in the form of probabilities)
model_1_pred_probs = model_1.predict(val_sentences)
model_1_pred_probs[:10] # only print out the first 10 prediction probabilities
```

Out [39]:

```
array([[0.4048821 ],
       [0.7443312 ],
       [0.997895 ],
       [0.10889997],
       [0.11143532],
       [0.93556094],
       [0.9134595 ],
       [0.9925345 ],
       [0.97156817],
       [0.26570338]], dtype=float32)
```

Since our final layer uses a sigmoid activation function, we get our predictions back in the form of probabilities.

To convert them to prediction classes, we'll use `tf.round()`, meaning prediction probabilities below 0.5 will

be rounded to 0 and those above 0.5 will be rounded to 1.

**Note:** In practice, the output threshold of a sigmoid prediction probability doesn't necessarily have to 0.5. For example, through testing, you may find that a cut off of 0.25 is better for your chosen evaluation metrics. A common example of this threshold cutoff is the [precision-recall tradeoff](#).

In [40]:

```
# Turn prediction probabilities into single-dimension tensor of floats
model_1_preds = tf.squeeze(tf.round(model_1_pred_probs)) # squeeze removes single dimensions
model_1_preds[:20]
```

Out[40]:

```
<tf.Tensor: shape=(20,), dtype=float32, numpy=
array([0., 1., 1., 0., 0., 1., 1., 1., 0., 0., 1., 0., 0., 0., 0., 0.,
       0., 0., 1.], dtype=float32)>
```

Now we've got our model's predictions in the form of classes, we can use our `calculate_results()` function to compare them to the ground truth validation labels.

In [41]:

```
# Calculate model_1 metrics
model_1_results = calculate_results(y_true=val_labels,
                                      y_pred=model_1_preds)
model_1_results
```

Out[41]:

```
{'accuracy': 78.74015748031496,
 'f1': 0.7846966492209201,
 'precision': 0.7914920592553047,
 'recall': 0.7874015748031497}
```

How about we compare our first deep model to our baseline model?

In [42]:

```
# Is our simple Keras model better than our baseline model?
import numpy as np
np.array(list(model_1_results.values())) > np.array(list(baseline_results.values()))
```

Out[42]:

```
array([False, False, False, False])
```

Since we'll be doing this kind of comparison (baseline compared to new model) quite a few times, let's create a function to help us out.

In [43]:

```
# Create a helper function to compare our baseline results to new model results
def compare_baseline_to_new_results(baseline_results, new_model_results):
    for key, value in baseline_results.items():
        print(f"Baseline {key}: {value:.2f}, New {key}: {new_model_results[key]:.2f}, Difference: {new_model_results[key]-value:.2f}")

compare_baseline_to_new_results(baseline_results=baseline_results,
                               new_model_results=model_1_results)
```

```
Baseline accuracy: 79.27, New accuracy: 78.74, Difference: -0.52
Baseline precision: 0.81, New precision: 0.79, Difference: -0.02
Baseline recall: 0.79, New recall: 0.79, Difference: -0.01
Baseline f1: 0.79, New f1: 0.78, Difference: -0.01
```

# Visualizing learned embeddings

Our first model (`model_1`) contained an embedding layer (`embedding`) which learned a way of representing words as feature vectors by passing over the training data.

Hearing this for the first few times may sound confusing.

So to further help understand what a text embedding is, let's visualize the embedding our model learned.

To do so, let's remind ourselves of the words in our vocabulary.

In [44]:

```
# Get the vocabulary from the text vectorization layer
words_in_vocab = text_vectorizer.get_vocabulary()
len(words_in_vocab), words_in_vocab[:10]
```

Out[44]:

```
(10000, ['', '[UNK]', 'the', 'a', 'in', 'to', 'of', 'and', 'i', 'is'])
```

And now let's get our embedding layer's weights (these are the numerical representations of each word).

In [45]:

```
model_1.summary()
```

Model: "model\_1\_dense"

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[None, 1]	0
text_vectorization_1 (TextVe	(None, 15)	0
embedding_1 (Embedding)	(None, 15, 128)	1280000
global_average_pooling1d (G1	(None, 128)	0
dense (Dense)	(None, 1)	129

Total params: 1,280,129  
Trainable params: 1,280,129  
Non-trainable params: 0

In [46]:

```
# Get the weight matrix of embedding layer
# (these are the numerical patterns between the text in the training dataset the model has learned)
embed_weights = model_1.get_layer("embedding_1").get_weights()[0]
print(embed_weights.shape) # same size as vocab size and embedding_dim (each word is a embedding_dim size vector)
```

```
(10000, 128)
```

Now we've got these two objects, we can use the [Embedding Projector tool](#) to visualize our embedding.

To use the Embedding Projector tool, we need two files:

- The embedding vectors (same as embedding weights).
- The meta data of the embedding vectors (the words they represent - our vocabulary).

Right now, we've got of these files as Python objects. To download them to file, we're going to [use the code example available on the TensorFlow word embeddings tutorial page](#).

In [47]:

```

# # Code below is adapted from: https://www.tensorflow.org/tutorials/text/word_embeddings
#retrieve_the_trained_word_embeddings_and_save_them_to_disk
# import io

# # Create output writers
# out_v = io.open("embedding_vectors.tsv", "w", encoding="utf-8")
# out_m = io.open("embedding_metadata.tsv", "w", encoding="utf-8")

# # Write embedding vectors and words to file
# for num, word in enumerate(words_in_vocab):
#     if num == 0:
#         continue # skip padding token
#     vec = embed_weights[num]
#     out_m.write(word + "\n") # write words to file
#     out_v.write("\t".join([str(x) for x in vec]) + "\n") # write corresponding word vector to file
# out_v.close()
# out_m.close()

# # Download files locally to upload to Embedding Projector
# try:
#     from google.colab import files
# except ImportError:
#     pass
# else:
#     files.download("embedding_vectors.tsv")
#     files.download("embedding_metadata.tsv")

```

Once you've downloaded the embedding vectors and metadata, you can visualize them using Embedding Vector tool:

1. Go to <http://projector.tensorflow.org/>
2. Click on "Load data"
3. Upload the two files you downloaded (`embedding_vectors.tsv` and `embedding_metadata.tsv`)
4. Explore
5. Optional: You can share the data you've created by clicking "Publish"

What do you find?

Are words with similar meanings close together?

Remember, they might not be. The embeddings we downloaded are how our model interprets words, not necessarily how we interpret them.

Also, since the embedding has been learned purely from Tweets, it may contain some strange values as Tweets are a very unique style of natural language.

 **Question:** Do you have to visualize embeddings every time?

No. Although helpful for gaining an intuition of what natural language embeddings are, it's not completely necessary. Especially as the dimensions of your vocabulary and embeddings grow, trying to comprehend them would become an increasingly difficult task.

## Recurrent Neural Networks (RNN's)

For our next series of modelling experiments we're going to be using a special kind of neural network called a **Recurrent Neural Network (RNN)**.

The premise of an RNN is simple: use information from the past to help you with the future (this is where the term recurrent comes from). In other words, take an input (`x`) and compute an output (`y`) based on all previous inputs.

This concept is especially helpful when dealing with sequences such as passages of natural language text (such as our Tweets).

For example, when you read this sentence, you take into context the previous words when deciphering the

meaning of the current word dog.

See what happened there?

I put the word "dog" at the end which is a valid word but it doesn't make sense in the context of the rest of the sentence.

When an RNN looks at a sequence of text (already in numerical form), the patterns it learns are continually updated based on the order of the sequence.

For a simple example, take two sentences:

1. Massive earthquake last week, no?
2. No massive earthquake last week.

Both contain exactly the same words but have different meaning. The order of the words determines the meaning (one could argue punctuation marks also dictate the meaning but for simplicity sake, let's stay focused on the words).

Recurrent neural networks can be used for a number of sequence-based problems:

- **One to one:** one input, one output, such as image classification.
- **One to many:** one input, many outputs, such as image captioning (image input, a sequence of text as caption output).
- **Many to one:** many inputs, one outputs, such as text classification (classifying a Tweet as real disaster or not real disaster).
- **Many to many:** many inputs, many outputs, such as machine translation (translating English to Spanish) or speech to text (audio wave as input, text as output).

When you come across RNN's in the wild, you'll most likely come across variants of the following:

- Long short-term memory cells (LSTMs).
- Gated recurrent units (GRUs).
- Bidirectional RNN's (passes forward and backward along a sequence, left to right and right to left).

Going into the details of each these is beyond the scope of this notebook (we're going to focus on using them instead), the main thing you should know for now is that they've proven very effective at modelling sequences.

For a deeper understanding of what's happening behind the scenes of the code we're about to write, I'd recommend the following resources:

#### □ Resources:

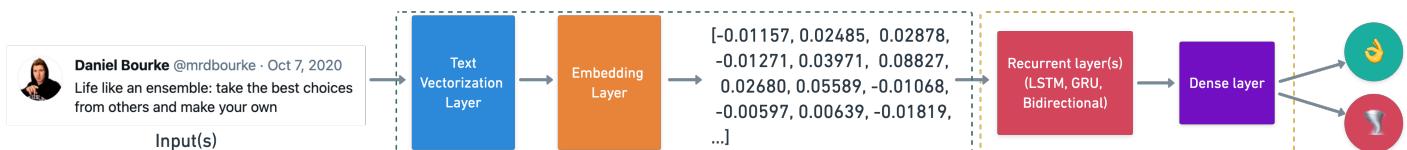
- [MIT Deep Learning Lecture on Recurrent Neural Networks](#) - explains the background of recurrent neural networks and introduces LSTMs.
- [The Unreasonable Effectiveness of Recurrent Neural Networks](#) by Andrej Karpathy - demonstrates the power of RNN's with examples generating various sequences.
- [Understanding LSTMs](#) by Chris Olah - an in-depth (and technical) look at the mechanics of the LSTM cell, possibly the most popular RNN building block.

## Model 2: LSTM

With all this talk of what RNN's are and what they're good for, I'm sure you're eager to build one.

We're going to start with an LSTM-powered RNN.

To harness the power of the LSTM cell (LSTM cell and LSTM layer are often used interchangably) in TensorFlow, we'll use `tensorflow.keras.layers.LSTM()`.



*Coloured block example of the structure of an recurrent neural network.*

Our model is going to take on a very similar structure to model 1:

Input (text) -> Tokenize -> Embedding -> Layers -> Output (label probability)

The main difference will be that we're going to add an LSTM layer between our embedding and output. And to make sure we're not getting reusing trained embeddings (this would involve data leakage between

models, leading to an uneven comparison later on), we'll create another embedding layer (`model_2_embedding`) for our model. The `text_vectorizer` layer can be reused since it doesn't get updated during training.

□ **Note:** The reason we use a new embedding layer for each model is since the embedding layer is a *learned* representation of words (as numbers), if we were to use the same embedding layer (`embedding_1`) for each model, we'd be mixing what one model learned with the next. And because we want to compare our models later on, starting them with their own embedding layer each time is a better idea.

In [48]:

```
# Set random seed and create embedding layer (new embedding layer for each model)
tf.random.set_seed(42)
from tensorflow.keras import layers
model_2_embedding = layers.Embedding(input_dim=max_vocab_length,
                                      output_dim=128,
                                      embeddings_initializer="uniform",
                                      input_length=max_length,
                                      name="embedding_2")

# Create LSTM model
inputs = layers.Input(shape=(1,), dtype="string")
x = text_vectorizer(inputs)
x = model_2_embedding(x)
print(x.shape)
# x = layers.LSTM(64, return_sequences=True)(x) # return vector for each word in the Tweet
# (you can stack RNN cells as long as return_sequences=True)
x = layers.LSTM(64)(x) # return vector for whole sequence
print(x.shape)
# x = layers.Dense(64, activation="relu")(x) # optional dense layer on top of output of LSTM cell
outputs = layers.Dense(1, activation="sigmoid")(x)
model_2 = tf.keras.Model(inputs, outputs, name="model_2_LSTM")
```

□ Note: Reading the documentation for the [TensorFlow LSTM layer](#), you'll find a plethora of parameters. Many of these have been tuned to make sure they compute as fast as possible. The main ones you'll be looking to adjust are `units` (number of hidden units) and `return_sequences` (set this to `True` when stacking LSTM or other recurrent layers).

**Now we've got our LSTM model built, let's compile it using "binary\_crossentropy" loss and the Adam optimizer.**

In [49]:

```
metrics=[ "accuracy" ])
```

And before we fit our model to the data, let's get a summary.

In [50]:

```
model_2.summary()
```

Model: "model\_2\_LSTM"

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[ (None, 1) ]	0
text_vectorization_1 (TextVe	(None, 15)	0
embedding_2 (Embedding)	(None, 15, 128)	1280000
lstm (LSTM)	(None, 64)	49408
dense_1 (Dense)	(None, 1)	65
=====		
Total params:	1,329,473	
Trainable params:	1,329,473	
Non-trainable params:	0	

Looking good! You'll notice a fair few more trainable parameters within our LSTM layer than `model_1`.

If you'd like to know where this number comes from, I recommend going through the above resources as well the following on calculating the number of parameters in an LSTM cell:

- [Stack Overflow answer to calculate the number of parameters in an LSTM cell](#) by Marcin Możejko
- [Calculating number of parameters in a LSTM unit and layer](#) by Shridhar Priyadarshi

Now our first RNN model's compiled let's fit it to our training data, validating it on the validation data and tracking its training parameters using our TensorBoard callback.

In [51]:

```
# Fit model
model_2_history = model_2.fit(train_sentences,
                               train_labels,
                               epochs=5,
                               validation_data=(val_sentences, val_labels),
                               callbacks=[create_tensorboard_callback(SAVE_DIR,
                                                               "LSTM")])
```

```
Saving TensorBoard log files to: model_logs/LSTM/20210923-052618
Epoch 1/5
215/215 [=====] - 12s 34ms/step - loss: 0.5100 - accuracy: 0.741
- val_loss: 0.4566 - val_accuracy: 0.7822
Epoch 2/5
215/215 [=====] - 4s 19ms/step - loss: 0.3176 - accuracy: 0.8717
- val_loss: 0.5138 - val_accuracy: 0.7756
Epoch 3/5
215/215 [=====] - 4s 18ms/step - loss: 0.2201 - accuracy: 0.9152
- val_loss: 0.5858 - val_accuracy: 0.7677
Epoch 4/5
215/215 [=====] - 4s 19ms/step - loss: 0.1556 - accuracy: 0.9428
- val_loss: 0.6041 - val_accuracy: 0.7743
Epoch 5/5
215/215 [=====] - 4s 20ms/step - loss: 0.1076 - accuracy: 0.9594
- val_loss: 0.8746 - val_accuracy: 0.7507
```

Nice! We've got our first trained RNN model using LSTM cells. Let's make some predictions with it.

The same thing will happen as before, due to the sigmoid activation function in the final layer, when we call the `predict()` method on our model, it'll return prediction probabilities rather than classes.

In [52]:

```
# Make predictions on the validation dataset
model_2_pred_probs = model_2.predict(val_sentences)
model_2_pred_probs.shape, model_2_pred_probs[:10] # view the first 10
```

Out[52]:

```
((762, 1), array([[0.00712602],
 [0.7873681 ],
 [0.9996376 ],
 [0.05679193],
 [0.0025822 ],
 [0.9996238 ],
 [0.9217023 ],
 [0.9997993 ],
 [0.9994954 ],
 [0.6645735 ]], dtype=float32))
```

We can turn these prediction probabilities into prediction classes by rounding to the nearest integer (by default, prediction probabilities under 0.5 will go to 0 and those over 0.5 will go to 1).

In [53]:

```
# Round out predictions and reduce to 1-dimensional array
model_2_preds = tf.squeeze(tf.round(model_2_pred_probs))
model_2_preds[:10]
```

Out[53]:

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=array([0., 1., 1., 0., 0., 1., 1., 1., 1., 1.], dtype=float32)>
```

**Beautiful, now let's use our `calculate_results()` function to evaluate our LSTM model and our `compare_baseline_to_new_results()` function to compare it to our baseline model.**

In [54]:

```
# Calculate LSTM model results
model_2_results = calculate_results(y_true=val_labels,
                                     y_pred=model_2_preds)
model_2_results
```

Out[54]:

```
{'accuracy': 75.06561679790026,
 'f1': 0.7489268622514025,
 'precision': 0.7510077975908164,
 'recall': 0.7506561679790026}
```

In [55]:

```
# Compare model 2 to baseline
compare_baseline_to_new_results(baseline_results, model_2_results)
```

```
Baseline accuracy: 79.27, New accuracy: 75.07, Difference: -4.20
Baseline precision: 0.81, New precision: 0.75, Difference: -0.06
Baseline recall: 0.79, New recall: 0.75, Difference: -0.04
Baseline f1: 0.79, New f1: 0.75, Difference: -0.04
```

## Model 3: GRU

Another popular and effective RNN component is the GRU or gated recurrent unit.

The GRU cell has similar features to an LSTM cell but has less parameters.

**Resource:** A full explanation of the GRU cell is beyond the scope of this notebook but I'd suggest the following resources to learn more:

- [Gated Recurrent Unit Wikipedia page](#)
- [Understanding GRU networks by Simeon Kostadinov](#)

To use the GRU cell in TensorFlow, we can call the `tensorflow.keras.layers.GRU()` class.

The architecture of the GRU-powered model will follow the same structure we've been using:

Input (text) -> Tokenize -> Embedding -> Layers -> Output (label probability)

Again, the only difference will be the layer(s) we use between the embedding and the output.

In [56]:

```
# Set random seed and create embedding layer (new embedding layer for each model)
tf.random.set_seed(42)
from tensorflow.keras import layers
model_3_embedding = layers.Embedding(input_dim=max_vocab_length,
                                      output_dim=128,
                                      embeddings_initializer="uniform",
                                      input_length=max_length,
                                      name="embedding_3")

# Build an RNN using the GRU cell
inputs = layers.Input(shape=(1,), dtype="string")
x = text_vectorizer(inputs)
x = model_3_embedding(x)
# x = layers.GRU(64, return_sequences=True) # stacking recurrent cells requires return_sequences=True
x = layers.GRU(64)(x)
# x = layers.Dense(64, activation="relu")(x) # optional dense layer after GRU cell
outputs = layers.Dense(1, activation="sigmoid")(x)
model_3 = tf.keras.Model(inputs, outputs, name="model_3_GRU")
```

TensorFlow makes it easy to use powerful components such as the GRU cell in our models. And now our third model is built, let's compile it, just as before.

In [57]:

```
# Compile GRU model
model_3.compile(loss="binary_crossentropy",
                 optimizer=tf.keras.optimizers.Adam(),
                 metrics=["accuracy"])
```

What does a summary of our model look like?

In [58]:

```
# Get a summary of the GRU model
model_3.summary()
```

Model: "model\_3\_GRU"

Layer (type)	Output Shape	Param #
<hr/>		
input_3 (InputLayer)	[ (None, 1) ]	0
<hr/>		
text_vectorization_1 (TextVe	(None, 15)	0
<hr/>		
embedding_3 (Embedding)	(None, 15, 128)	1280000
<hr/>		
gru (GRU)	(None, 64)	37248
<hr/>		
dense_2 (Dense)	(None, 1)	65
<hr/>		
Total params: 1,317,313		
Trainable params: 1,317,313		
Non-trainable params: 0		

**Notice the difference in number of trainable parameters between `model_2` (LSTM) and `model_3` (GRU). The difference comes from the LSTM cell having more trainable parameters than the GRU cell.**

**We'll fit our model just as we've been doing previously. We'll also track our models results using our `create_tensorboard_callback()` function.**

In [59]:

```
# Fit model
model_3_history = model_3.fit(train_sentences,
                               train_labels,
                               epochs=5,
                               validation_data=(val_sentences, val_labels),
                               callbacks=[create_tensorboard_callback(SAVE_DIR, "GRU")])
```

```
Saving TensorBoard log files to: model_logs/GRU/20210923-052650
Epoch 1/5
215/215 [=====] - 12s 24ms/step - loss: 0.5242 - accuracy: 0.731
- val_loss: 0.4553 - val_accuracy: 0.7769
Epoch 2/5
215/215 [=====] - 3s 15ms/step - loss: 0.3195 - accuracy: 0.8694
- val_loss: 0.4937 - val_accuracy: 0.7808
Epoch 3/5
215/215 [=====] - 2s 10ms/step - loss: 0.2197 - accuracy: 0.9181
- val_loss: 0.5607 - val_accuracy: 0.7743
Epoch 4/5
215/215 [=====] - 2s 10ms/step - loss: 0.1599 - accuracy: 0.9441
- val_loss: 0.6220 - val_accuracy: 0.7782
Epoch 5/5
215/215 [=====] - 2s 10ms/step - loss: 0.1221 - accuracy: 0.9584
- val_loss: 0.6205 - val_accuracy: 0.7677
```

**Due to the optimized default settings of the GRU cell in TensorFlow, training doesn't take long at all.**

**Time to make some predictions on the validation samples.**

In [60]:

```
# Make predictions on the validation data
model_3_pred_probs = model_3.predict(val_sentences)
model_3_pred_probs.shape, model_3_pred_probs[:10]
```

Out[60]:

```
((762, 1), array([[0.33325258],
 [0.87741184],
 [0.9980252 ],
 [0.11561754],
 [0.01235959],
 [0.9925639 ],
 [0.6214262 ],
 [0.99813336],
 [0.9982377 ],
 [0.50181067]], dtype=float32))
```

**Again we get an array of prediction probabilities back which we can convert to prediction classes by rounding them.**

In [61]:

```
# Convert prediction probabilities to prediction classes
model_3_preds = tf.squeeze(tf.round(model_3_pred_probs))
model_3_preds[:10]
```

Out[61]:

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=array([0., 1., 1., 0., 0., 1., 1., 1., 1., 1.], dtype=float32)>
```

Now we've got predicted classes, let's evaluate them against the ground truth labels.

In [62]:

```
# Calculate model_3 results
model_3_results = calculate_results(y_true=val_labels,
                                      y_pred=model_3_preds)
model_3_results
```

Out[62]:

```
{'accuracy': 76.77165354330708,
 'f1': 0.7667932666650168,
 'precision': 0.7675450859410361,
 'recall': 0.7677165354330708}
```

Finally we can compare our GRU model's results to our baseline.

In [63]:

```
# Compare to baseline
compare_baseline_to_new_results(baseline_results, model_3_results)
```

```
Baseline accuracy: 79.27, New accuracy: 76.77, Difference: -2.49
Baseline precision: 0.81, New precision: 0.77, Difference: -0.04
Baseline recall: 0.79, New recall: 0.77, Difference: -0.02
Baseline f1: 0.79, New f1: 0.77, Difference: -0.02
```

## Model 4: Bidirectional RNN model

Look at us go! We've already built two RNN's with GRU and LSTM cells. Now we're going to look into another kind of RNN, the bidirectional RNN.

A standard RNN will process a sequence from left to right, whereas a bidirectional RNN will process the sequence from left to right and then again from right to left.

Intuitively, this can be thought of as if you were reading a sentence for the first time in the normal fashion (left to right) but for some reason it didn't make sense so you traverse back through the words and go back over them again (right to left).

In practice, many sequence models often see an improvement in performance when using bidirectional RNN's.

However, this improvement in performance often comes at the cost of longer training times and increased model parameters (since the model goes left to right and right to left, the number of trainable parameters doubles).

Okay enough talk, let's build a bidirectional RNN.

Once again, TensorFlow helps us out by providing the `tensorflow.keras.layers.Bidirectional` class. We can use the `Bidirectional` class to wrap our existing RNNs, instantly making them bidirectional.

In [64]:

```
# Set random seed and create embedding layer (new embedding layer for each model)
tf.random.set_seed(42)
from tensorflow.keras import layers
model_4_embedding = layers.Embedding(input_dim=max_vocab_length,
                                      output_dim=128,
                                      embeddings_initializer="uniform",
                                      input_length=max_length,
                                      name="embedding_4")

# Build a Bidirectional RNN in TensorFlow
inputs = layers.Input(shape=(1,), dtype="string")
x = text_vectorizer(inputs)
x = model_4_embedding(x)
# x = layers.Bidirectional(layers.LSTM(64, return_sequences=True))(x) # stacking RNN laye
```

```

rs requires return_sequences=True
x = layers.Bidirectional(layers.LSTM(64))(x) # bidirectional goes both ways so has double the parameters of a regular LSTM layer
outputs = layers.Dense(1, activation="sigmoid")(x)
model_4 = tf.keras.Model(inputs, outputs, name="model_4_Bidirectional")

```

**Note:** You can use the `Bidirectional` wrapper on any RNN cell in TensorFlow. For example, `layers.Bidirectional(layers.GRU(64))` creates a bidirectional GRU cell.

Our bidirectional model is built, let's compile it.

In [65]:

```

# Compile
model_4.compile(loss="binary_crossentropy",
                 optimizer=tf.keras.optimizers.Adam(),
                 metrics=["accuracy"])

```

And of course, we'll check out a summary.

In [66]:

```

# Get a summary of our bidirectional model
model_4.summary()

```

Model: "model\_4\_Bidirectional"

Layer (type)	Output Shape	Param #
input_4 (InputLayer)	[(None, 1)]	0
text_vectorization_1 (TextVectorization)	(None, 15)	0
embedding_4 (Embedding)	(None, 15, 128)	1280000
bidirectional (Bidirectional)	(None, 128)	98816
dense_3 (Dense)	(None, 1)	129

Total params: 1,378,945  
Trainable params: 1,378,945  
Non-trainable params: 0

Notice the increased number of trainable parameters in `model_4` (bidirectional LSTM) compared to `model_2` (regular LSTM). This is due to the bidirectionality we added to our RNN.

Time to fit our bidirectional model and track its performance.

In [67]:

```

# Fit the model (takes longer because of the bidirectional layers)
model_4_history = model_4.fit(train_sentences,
                               train_labels,
                               epochs=5,
                               validation_data=(val_sentences, val_labels),
                               callbacks=[create_tensorboard_callback(SAVE_DIR, "bidirectional_RNN")])

```

Saving TensorBoard log files to: `model_logs/bidirectional_RNN/20210923-052719`  
Epoch 1/5  
215/215 [=====] - 8s 21ms/step - loss: 0.5093 - accuracy: 0.7481  
- val\_loss: 0.4606 - val\_accuracy: 0.7795  
Epoch 2/5  
215/215 [=====] - 3s 14ms/step - loss: 0.3135 - accuracy: 0.8708  
- val\_loss: 0.5144 - val\_accuracy: 0.7690  
Epoch 3/5

```
215/215 [=====] - 3s 14ms/step - loss: 0.2150 - accuracy: 0.9178
- val_loss: 0.5626 - val_accuracy: 0.7677
Epoch 4/5
215/215 [=====] - 3s 14ms/step - loss: 0.1523 - accuracy: 0.9469
- val_loss: 0.6365 - val_accuracy: 0.7769
Epoch 5/5
215/215 [=====] - 3s 14ms/step - loss: 0.1083 - accuracy: 0.9639
- val_loss: 0.6509 - val_accuracy: 0.7664
```

**Due to the bidirectionality of our model we see a slight increase in training time.**

**Not to worry, it's not too dramatic of an increase.**

**Let's make some predictions with it.**

In [68]:

```
# Make predictions with bidirectional RNN on the validation data
model_4_pred_probs = model_4.predict(val_sentences)
model_4_pred_probs[:10]
```

Out[68]:

```
array([[0.04000043],
       [0.827929],
       [0.99842227],
       [0.1353109],
       [0.00311337],
       [0.99220747],
       [0.9552836],
       [0.99945647],
       [0.99898285],
       [0.28141677]], dtype=float32)
```

**And we'll convert them to prediction classes and evaluate them against the ground truth labels and baseline model.**

In [69]:

```
# Convert prediction probabilities to labels
model_4_preds = tf.squeeze(tf.round(model_4_pred_probs))
model_4_preds[:10]
```

Out[69]:

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=array([0., 1., 1., 0., 0., 1., 1., 1., 1., 0.], dtype=float32)>
```

In [70]:

```
# Calculate bidirectional RNN model results
model_4_results = calculate_results(val_labels, model_4_preds)
model_4_results
```

Out[70]:

```
{'accuracy': 76.64041994750657,
 'f1': 0.7651213533864446,
 'precision': 0.7665895370389821,
 'recall': 0.7664041994750657}
```

In [71]:

```
# Check to see how the bidirectional model performs against the baseline
compare_baseline_to_new_results(baseline_results, model_4_results)
```

```
Baseline accuracy: 79.27, New accuracy: 76.64, Difference: -2.62
Baseline precision: 0.81, New precision: 0.77, Difference: -0.04
Baseline recall: 0.79, New recall: 0.77, Difference: -0.03
Baseline f1: 0.79, New f1: 0.77, Difference: -0.02
```

# Convolutional Neural Networks for Text

You might've used convolutional neural networks (CNNs) for images before but they can also be used for sequences.

The main difference between using CNNs for images and sequences is the shape of the data. Images come in 2-dimensions (height x width) whereas sequences are often 1-dimensional (a string of text).

So to use CNNs with sequences, we use a 1-dimensional convolution instead of a 2-dimensional convolution.

A typical CNN architecture for sequences will look like the following:

Inputs (text) -> Tokenization -> Embedding -> Layers -> Outputs (class probabilities)

You might be thinking "that just looks like the architecture layout we've been using for the other models..."

And you'd be right.

The difference again is in the layers component. Instead of using an LSTM or GRU cell, we're going to use a `tensorflow.keras.layers.Conv1D()` layer followed by a `tensorflow.keras.layers.GlobalMaxPool1D()` layer.

Resource: The intuition here is explained succinctly in the paper [Understanding Convolutional Neural Networks for Text Classification](#), where they state that CNNs classify text through the following steps:

1. 1-dimensional convolving filters are used as ngram detectors, each filter specializing in a closely-related family of ngrams (an ngram is a collection of n-words, for example, an ngram of 5 might result in "hello, my name is Daniel").
2. Max-pooling over time extracts the relevant ngrams for making a decision.
3. The rest of the network classifies the text based on this information.

## Model 5: Conv1D

Before we build a full 1-dimensional CNN model, let's see a 1-dimensional convolutional layer (also called a **temporal convolution**) in action.

We'll first create an embedding of a sample of text and experiment passing it through a `Conv1D()` layer and `GlobalMaxPool1D()` layer.

In [72]:

```
# Test out the embedding, 1D convolutional and max pooling
embedding_test = embedding(text_vectorizer(["this is a test sentence"])) # turn target sentence into embedding
conv_1d = layers.Conv1D(filters=32, kernel_size=5, activation="relu") # convolve over target sequence 5 words at a time
conv_1d_output = conv_1d(embedding_test) # pass embedding through 1D convolutional layer
max_pool = layers.GlobalMaxPool1D()
max_pool_output = max_pool(conv_1d_output) # get the most important features
embedding_test.shape, conv_1d_output.shape, max_pool_output.shape
```

Out [72]:

```
(TensorShape([1, 15, 128]), TensorShape([1, 11, 32]), TensorShape([1, 32]))
```

Notice the output shapes of each layer.

The embedding has an output shape dimension of the parameters we set it to (`input_length=15` and `output_dim=128`).

The 1-dimensional convolutional layer has an output which has been compressed inline with its parameters. And the same does for the max pooling layer output.

**Our text starts out as a string but gets converted to a feature vector of length 64 through various transformation steps (from tokenization to embedding to 1-dimensional convolution to max pool).**

**Let's take a peak at what each of these transformations looks like.**

In [73]:

```
# See the outputs of each layer
embedding_test[:1], conv_1d_output[:1], max_pool_output[:1]
```

Out [73]:

```
<tf.Tensor: shape=(1, 15, 128), dtype=float32, numpy=
array([[[ 0.02534914, -0.03109061,  0.00285616, ..., -0.00783159,
         -0.02685575, -0.0443413 ],
       [-0.0658626 ,  0.09451495, -0.01477603, ..., -0.00657781,
        -0.04238792,  0.07777896],
       [-0.04803652, -0.00709756, -0.02330894, ..., -0.0180733 ,
        0.02351036,  0.02676384],
       ...,
       [ 0.00073166,  0.01504797, -0.03425457, ..., -0.04403538,
        -0.01042282,  0.01876436],
       [ 0.00073166,  0.01504797, -0.03425457, ..., -0.04403538,
        -0.01042282,  0.01876436],
       [ 0.00073166,  0.01504797, -0.03425457, ..., -0.04403538,
        -0.01042282,  0.01876436]], dtype=float32)>,
<tf.Tensor: shape=(1, 11, 32), dtype=float32, numpy=
array([[0.08324985, 0.00648716, 0.          , 0.03983572, 0.          ,
        0.01144416, 0.00416251, 0.0228839 , 0.          , 0.00900978,
        0.          , 0.          , 0.03401771, 0.06408274, 0.08103722,
        0.00409014, 0.01579616, 0.          , 0.07930177, 0.          ,
        0.          , 0.          , 0.14525084, 0.          , 0.          ,
        0.          , 0.03682078, 0.06534287, 0.          , 0.          ,
        0.05094624, 0.          ],
       [0.          , 0.05387188, 0.          , 0.11491331, 0.          ,
        0.          , 0.1623708 , 0.          , 0.          , 0.00171254,
        0.14336711, 0.          , 0.          , 0.          , 0.          ,
        0.01197936, 0.          , 0.          , 0.13551372, 0.0040106 ,
        0.10309819, 0.09445544, 0.08390297, 0.          , 0.04213036,
        0.04487597, 0.06560461, 0.          , 0.02272684, 0.          ,
        0.          , 0.          ],
       [0.03683221, 0.04895764, 0.          , 0.1532475 , 0.          ,
        0.          , 0.          , 0.          , 0.          , 0.          ,
        0.          , 0.04650313, 0.00496456, 0.07349401, 0.01608641,
        0.          , 0.02779119, 0.          , 0.0808056 , 0.01403176,
        0.          , 0.03768815, 0.1038278 , 0.          , 0.03361662,
        0.          , 0.02577607, 0.00140354, 0.          , 0.          ,
        0.03211498, 0.          ],
       [0.0088782 , 0.10450974, 0.          , 0.06974535, 0.02328686,
        0.          , 0.04052207, 0.          , 0.          , 0.02733764,
        0.08674346, 0.          , 0.          , 0.06129852, 0.02007267,
        0.          , 0.          , 0.          , 0.03364263, 0.          ,
        0.04525332, 0.05219702, 0.06375706, 0.          , 0.          ,
        0.00774407, 0.00273467, 0.          , 0.          , 0.00499633,
        0.          , 0.          ],
       [0.          , 0.02369069, 0.          , 0.05827617, 0.05297644,
        0.          , 0.          , 0.          , 0.          , 0.          ,
        0.01719718, 0.02936822, 0.00466103, 0.06879887, 0.01944808,
        0.01585533, 0.01294545, 0.          , 0.06866529, 0.          ,
        0.00623766, 0.0351405 , 0.02407533, 0.          , 0.05979815,
        0.          , 0.01170142, 0.          , 0.          , 0.          ,
        0.04444929, 0.          ],
       [0.03544863, 0.          , 0.          , 0.05054973, 0.06105441,
        0.          , 0.00997427, 0.01403005, 0.          , 0.01680727,
        0.0314851 , 0.03889389, 0.          , 0.07710679, 0.0059097 ,
        0.          , 0.00263033, 0.          , 0.08935824, 0.          ,
        0.          , 0.05331149, 0.0522795 , 0.          , 0.06658384,
        0.01881707, 0.02448696, 0.          , 0.          , 0.          ,
        0.02008456, 0.          ],
       [0.03544863, 0.          , 0.          , 0.05054973, 0.06105442,
        0.          , 0.00997426, 0.01403006, 0.          , 0.01680727,
```

```

0.03148509, 0.03889391, 0. , 0.07710679, 0.0059097 ,
0. , 0.00263035, 0. , 0.08935823, 0. ,
0. , 0.05331149, 0.05227951, 0. , 0.06658384,
0.01881707, 0.02448694, 0. , 0. , 0. ,
0.02008457, 0. , ],
[0.03544864, 0. , 0. , 0.05054973, 0.06105441,
0. , 0.00997426, 0.01403005, 0. , 0.01680726,
0.0314851 , 0.03889389, 0. , 0.07710679, 0.0059097 ,
0. , 0.00263034, 0. , 0.08935826, 0. ,
0. , 0.0533115 , 0.0522795 , 0. , 0.06658384,
0.01881707, 0.02448694, 0. , 0. , 0. ,
0.02008457, 0. , ],
[0.03544863, 0. , 0. , 0.05054973, 0.06105442,
0. , 0.00997426, 0.01403005, 0. , 0.01680727,
0.0314851 , 0.0388939 , 0. , 0.07710679, 0.0059097 ,
0. , 0.00263034, 0. , 0.08935825, 0. ,
0. , 0.05331149, 0.05227951, 0. , 0.06658386,
0.01881707, 0.02448695, 0. , 0. , 0. ,
0.02008456, 0. , ],
[0.03544863, 0. , 0. , 0.05054973, 0.0610544 ,
0. , 0.00997427, 0.01403005, 0. , 0.01680727,
0.0314851 , 0.0388939 , 0. , 0.0771068 , 0.0059097 ,
0. , 0.00263034, 0. , 0.08935825, 0. ,
0. , 0.05331149, 0.05227951, 0. , 0.06658386,
0.01881707, 0.02448695, 0. , 0. , 0. ,
0.02008456, 0. , ],
[0.03544863, 0. , 0. , 0.05054973, 0.06105442,
0. , 0.00997426, 0.01403006, 0. , 0.01680726,
0.03148509, 0.0388939 , 0. , 0.0771068 , 0.0059097 ,
0. , 0.00263034, 0. , 0.08935824, 0. ,
0. , 0.05331149, 0.05227952, 0. , 0.06658386,
0.01881706, 0.02448695, 0. , 0. , 0. ,
0.02008456, 0. , ],
<tf.Tensor: shape=(1, 32), dtype=float32>,
<tf.Tensor: shape=(1, 32), dtype=float32, numpy=
array([[0.08324985, 0.10450974, 0. , 0.1532475 , 0.06105442,
       0.01144416, 0.1623708 , 0.0228839 , 0. , 0.02733764,
       0.14336711, 0.04650313, 0.03401771, 0.0771068 , 0.08103722,
       0.01585533, 0.02779119, 0. , 0.13551372, 0.01403176,
       0.10309819, 0.09445544, 0.14525084, 0. , 0.06658386,
       0.04487597, 0.06560461, 0.06534287, 0.02272684, 0.00499633,
       0.05094624, 0. , ]], dtype=float32)>

```

**Alright, we've seen the outputs of several components of a CNN for sequences, let's put them together and construct a full model, compile it (just as we've done with our other models) and get a summary.**

In [74]:

```

# Set random seed and create embedding layer (new embedding layer for each model)
tf.random.set_seed(42)
from tensorflow.keras import layers
model_5_embedding = layers.Embedding(input_dim=max_vocab_length,
                                       output_dim=128,
                                       embeddings_initializer="uniform",
                                       input_length=max_length,
                                       name="embedding_5")

# Create 1-dimensional convolutional layer to model sequences
from tensorflow.keras import layers
inputs = layers.Input(shape=(1,), dtype="string")
x = text_vectorizer(inputs)
x = model_5_embedding(x)
x = layers.Conv1D(filters=32, kernel_size=5, activation="relu")(x)
x = layers.GlobalMaxPool1D()(x)
# x = layers.Dense(64, activation="relu")(x) # optional dense layer
outputs = layers.Dense(1, activation="sigmoid")(x)
model_5 = tf.keras.Model(inputs, outputs, name="model_5_Conv1D")

# Compile Conv1D model
model_5.compile(loss="binary_crossentropy",
                 optimizer=tf.keras.optimizers.Adam(),

```

```
metrics=["accuracy"])

# Get a summary of our 1D convolution model
model_5.summary()
```

Model: "model\_5\_Conv1D"

Layer (type)	Output Shape	Param #
input_5 (InputLayer)	[None, 1]	0
text_vectorization_1 (TextVe	(None, 15)	0
embedding_5 (Embedding)	(None, 15, 128)	1280000
conv1d_1 (Conv1D)	(None, 11, 32)	20512
global_max_pooling1d_1 (Glob	(None, 32)	0
dense_4 (Dense)	(None, 1)	33
=====		
Total params: 1,300,545		
Trainable params: 1,300,545		
Non-trainable params: 0		

**Woohoo! Looking great! Notice how the number of trainable parameters for the 1-dimensional convolutional layer is similar to that of the LSTM layer in `model_2`.**

**Let's fit our 1D CNN model to our text data. In line with previous experiments, we'll save its results using our `create_tensorboard_callback()` function.**

In [75]:

```
# Fit the model
model_5_history = model_5.fit(train_sentences,
                               train_labels,
                               epochs=5,
                               validation_data=(val_sentences, val_labels),
                               callbacks=[create_tensorboard_callback(SAVE_DIR,
                                                                       "Conv1D")])
```

```
Saving TensorBoard log files to: model_logs/Conv1D/20210923-052810
Epoch 1/5
215/215 [=====] - 5s 10ms/step - loss: 0.5652 - accuracy: 0.7141
- val_loss: 0.4733 - val_accuracy: 0.7795
Epoch 2/5
215/215 [=====] - 2s 7ms/step - loss: 0.3380 - accuracy: 0.8615
- val_loss: 0.4758 - val_accuracy: 0.7730
Epoch 3/5
215/215 [=====] - 2s 8ms/step - loss: 0.2070 - accuracy: 0.9234
- val_loss: 0.5457 - val_accuracy: 0.7730
Epoch 4/5
215/215 [=====] - 2s 7ms/step - loss: 0.1314 - accuracy: 0.9578
- val_loss: 0.6163 - val_accuracy: 0.7730
Epoch 5/5
215/215 [=====] - 2s 7ms/step - loss: 0.0933 - accuracy: 0.9691
- val_loss: 0.6779 - val_accuracy: 0.7782
```

**Nice! Thanks to GPU acceleration, our 1D convolutional model trains nice and fast. Let's make some predictions with it and evaluate them just as before.**

In [76]:

```
# Make predictions with model_5
model_5_pred_probs = model_5.predict(val_sentences)
model_5_pred_probs[:10]
```

Out[76]:

```
array([[[0.225345  ],
       [0.7534112 ],
       [0.9995602 ],
       [0.05562792],
       [0.01449848],
       [0.9858518 ],
       [0.98418933],
       [0.99758804],
       [0.99862623],
       [0.26914373]], dtype=float32)
```

In [77]:

```
# Convert model_5 prediction probabilities to labels
model_5_preds = tf.squeeze(tf.round(model_5_pred_probs))
model_5_preds[:10]
```

Out[77]:

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=array([0., 1., 1., 0., 0., 1., 1., 1., 1., 0.], dtype=float32)>
```

In [78]:

```
# Calculate model_5 evaluation metrics
model_5_results = calculate_results(y_true=val_labels,
                                     y_pred=model_5_preds)
model_5_results
```

Out[78]:

```
{'accuracy': 77.82152230971128,
 'f1': 0.7758810170952618,
 'precision': 0.7807522349051432,
 'recall': 0.7782152230971129}
```

In [79]:

```
# Compare model_5 results to baseline
compare_baseline_to_new_results(baseline_results, model_5_results)
```

```
Baseline accuracy: 79.27, New accuracy: 77.82, Difference: -1.44
Baseline precision: 0.81, New precision: 0.78, Difference: -0.03
Baseline recall: 0.79, New recall: 0.78, Difference: -0.01
Baseline f1: 0.79, New f1: 0.78, Difference: -0.01
```

## Using Pretrained Embeddings (transfer learning for NLP)

For all of the previous deep learning models we've built and trained, we've created and used our own embeddings from scratch each time.

However, a common practice is to leverage pretrained embeddings through **transfer learning**. This is one of the main benefits of using deep models: being able to take what one (often larger) model has learned (often on a large amount of data) and adjust it for our own use case.

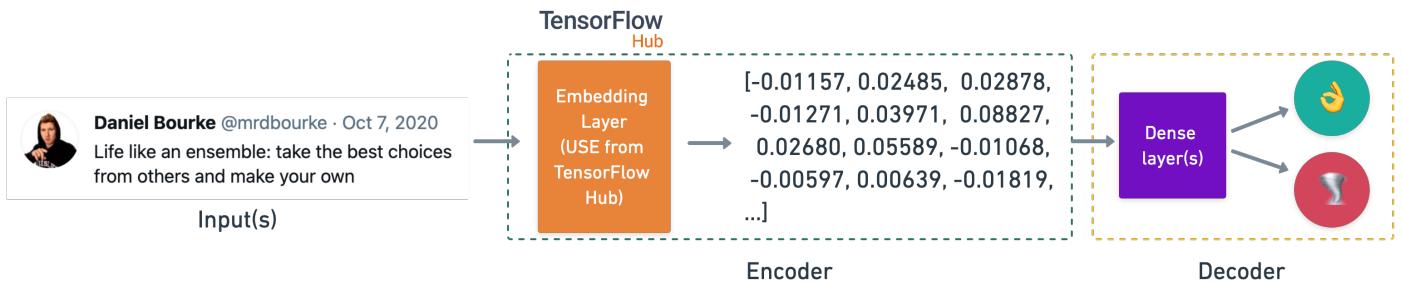
For our next model, instead of using our own embedding layer, we're going to replace it with a pretrained embedding layer.

More specifically, we're going to be using the [Universal Sentence Encoder](#) from [TensorFlow Hub](#) (a great resource containing a plethora of pretrained model resources for a variety of tasks).

**Note:** There are many different pretrained text embedding options on TensorFlow Hub, however, some require different levels of text preprocessing than others. Best to experiment with a few and see which best suits your use case.

The main difference between the embedding layer we created and the Universal Sentence Encoder is that rather than create a word-level embedding, the Universal Sentence Encoder, as you might've guessed, creates a whole sentence-level embedding.

Our embedding layer also outputs an a 128 dimensional vector for each word, where as, the Universal Sentence Encoder outputs a 512 dimensional vector for each sentence.



***The feature extractor model we're building through the eyes of an encoder/decoder model.***

■ Note: An **encoder** is the name for a model which converts raw data such as text into a numerical representation (feature vector), a **decoder** converts the numerical representation to a desired output.

As usual, this is best demonstrated with an example.

We can load in a TensorFlow Hub module using the `hub.load()` method and passing it the target URL of the module we'd like to use, in our case, it's "<https://tfhub.dev/google/universal-sentence-encoder/4>".

Let's load the Universal Sentence Encoder model and test it on a couple of sentences.

In [80]:

```
# Example of pretrained embedding with universal sentence encoder - https://tfhub.dev/google/universal-sentence-encoder/4
import tensorflow_hub as hub
embed = hub.load("https://tfhub.dev/google/universal-sentence-encoder/4") # load Universal Sentence Encoder
embed_samples = embed([sample_sentence,
    "When you call the universal sentence encoder on a sentence, it turns it into numbers."])
print(embed_samples[0][:50])
```

```
tf.Tensor(
[-0.01157024  0.0248591   0.0287805  -0.01271502  0.03971543  0.08827759
 0.02680986  0.05589837 -0.01068731 -0.0059729   0.00639324 -0.01819523
 0.00030817  0.09105891  0.05874644 -0.03180627  0.01512476 -0.05162928
 0.00991369 -0.06865346 -0.04209306  0.0267898   0.03011008  0.00321069
-0.00337969 -0.04787359  0.02266718 -0.00985924 -0.04063614 -0.01292095
-0.04666384  0.056303   -0.03949255  0.00517685  0.02495828 -0.07014439
 0.02871508  0.04947682 -0.00633971 -0.08960191  0.02807117 -0.00808362
-0.01360601  0.05998649 -0.10361786 -0.05195372  0.00232955 -0.02332528
-0.03758105  0.0332773 ], shape=(50,), dtype=float32)
```

In [81]:

```
# Each sentence has been encoded into a 512 dimension vector
embed_samples[0].shape
```

Out[81]:

```
TensorShape([512])
```

Passing our sentences to the Universal Sentence Encoder (USE) encodes them from strings to 512 dimensional vectors, which make no sense to us but hopefully make sense to our machine learning models.

Speaking of models, let's build one with the USE as our embedding layer.

We can convert the TensorFlow Hub USE module into a Keras layer using the `hub.KerasLayer` class.

**Note:** Due to the size of the USE TensorFlow Hub module, it may take a little while to download. Once it's downloaded though, it'll be cached and ready to use. And as with many TensorFlow Hub modules, there is a "lite" version of the USE which takes up less space but sacrifices some performance and requires more preprocessing steps. However, depending on your available compute power, the lite version may be better for your application use case.

In [82]:

```
# We can use this encoding layer in place of our text_vectorizer and embedding layer
sentence_encoder_layer = hub.KerasLayer("https://tfhub.dev/google/universal-sentence-encoder/4",
                                         input_shape=[], # shape of inputs coming to our
                                         model
                                         dtype=tf.string, # data type of inputs coming to
                                         trainable=False, # keep the pretrained weights (
                                         name="USE")
```

Beautiful! Now we've got the USE as a Keras layer, we can use it in a Keras Sequential model.

In [83]:

```
# Create model using the Sequential API
model_6 = tf.keras.Sequential([
    sentence_encoder_layer, # take in sentences and then encode them into an embedding
    layers.Dense(64, activation="relu"),
    layers.Dense(1, activation="sigmoid")
], name="model_6_USE")

# Compile model
model_6.compile(loss="binary_crossentropy",
                  optimizer=tf.keras.optimizers.Adam(),
                  metrics=["accuracy"])

model_6.summary()
```

Model: "model\_6\_USE"

Layer (type)	Output Shape	Param #
USE (KerasLayer)	(None, 512)	256797824
dense_5 (Dense)	(None, 64)	32832
dense_6 (Dense)	(None, 1)	65
Total params:	256,830,721	
Trainable params:	32,897	
Non-trainable params:	256,797,824	

Notice the number of parameters in the USE layer, these are the pretrained weights its learned on various text sources (Wikipedia, web news, web question-answer forums, etc, see the [Universal Sentence Encoder paper](#) for more).

The trainable parameters are only in our output layers, in other words, we're keeping the USE weights frozen and using it as a feature-extractor. We could fine-tune these weights by setting `trainable=True` when creating the `hub.KerasLayer` instance.

Now we've got a feature extractor model ready, let's train it and track its results to TensorBoard using our `create_tensorboard_callback()` function.

In [84]:

```
# Train a classifier on top of pretrained embeddings
model_6_history = model_6.fit(train_sentences,
                               train_labels,
                               epochs=5,
                               validation_data=(val_sentences, val_labels),
                               callbacks=[create_tensorboard_callback(SAVE_DIR,
                                                               "tf_hub_sentence_encoder")])
```

```
Saving TensorBoard log files to: model_logs/tf_hub_sentence_encoder/20210923-052854
Epoch 1/5
215/215 [=====] - 10s 32ms/step - loss: 0.5008 - accuracy: 0.789
- val_loss: 0.4478 - val_accuracy: 0.7966
Epoch 2/5
215/215 [=====] - 4s 19ms/step - loss: 0.4144 - accuracy: 0.8133
- val_loss: 0.4369 - val_accuracy: 0.8058
Epoch 3/5
215/215 [=====] - 4s 19ms/step - loss: 0.3998 - accuracy: 0.8212
- val_loss: 0.4329 - val_accuracy: 0.8110
Epoch 4/5
215/215 [=====] - 4s 18ms/step - loss: 0.3925 - accuracy: 0.8266
- val_loss: 0.4288 - val_accuracy: 0.8110
Epoch 5/5
215/215 [=====] - 4s 17ms/step - loss: 0.3860 - accuracy: 0.8276
- val_loss: 0.4309 - val_accuracy: 0.8123
```

**USE model trained! Let's make some predictions with it and evaluate them as we've done with our other models.**

In [85]:

```
# Make predictions with USE TF Hub model
model_6_pred_probs = model_6.predict(val_sentences)
model_6_pred_probs[:10]
```

Out[85]:

```
array([[0.14443193],
       [0.7271502 ],
       [0.9856655 ],
       [0.19740924],
       [0.73417026],
       [0.6859663 ],
       [0.9808888 ],
       [0.97411025],
       [0.91573215],
       [0.08070081]], dtype=float32)
```

In [86]:

```
# Convert prediction probabilities to labels
model_6_preds = tf.squeeze(tf.round(model_6_pred_probs))
model_6_preds[:10]
```

Out[86]:

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=array([0., 1., 1., 0., 1., 1., 1., 1., 1., 0.], dtype=float32)>
```

In [87]:

```
# Calculate model 6 performance metrics
model_6_results = calculate_results(val_labels, model_6_preds)
model_6_results
```

Out[87]:

```
{'accuracy': 81.23359580052494,
 'f1': 0.810686575717776,
 'precision': 0.8148798668657973,
 'recall': 0.8123359580052494}
```

In [88]:

```
# Compare TF Hub model to baseline
compare_baseline_to_new_results(baseline_results, model_6_results)

Baseline accuracy: 79.27, New accuracy: 81.23, Difference: 1.97
Baseline precision: 0.81, New precision: 0.81, Difference: 0.00
Baseline recall: 0.79, New recall: 0.81, Difference: 0.02
Baseline f1: 0.79, New f1: 0.81, Difference: 0.02
```

## Model 7: TensorFlow Hub Pretrained Sentence Encoder 10% of the training data

One of the benefits of using transfer learning methods, such as, the pretrained embeddings within the USE is the ability to get great results on a small amount of data (the USE paper even mentions this in the abstract).

To put this to the test, we're going to make a small subset of the training data (10%), train a model and evaluate it.

In [89]:

```
### NOTE: Making splits like this will lead to data leakage ###
### (some of the training examples in the validation set) ###

### WRONG WAY TO MAKE SPLITS (train_df_shuffled has already been split) ###

# # Create subsets of 10% of the training data
# train_10_percent = train_df_shuffled[["text", "target"]].sample(frac=0.1, random_state=42)
# train_sentences_10_percent = train_10_percent["text"].to_list()
# train_labels_10_percent = train_10_percent["target"].to_list()
# len(train_sentences_10_percent), len(train_labels_10_percent)
```

In [90]:

```
# One kind of correct way (there are more) to make data subset
# (split the already split train_sentences/train_labels)
train_sentences_90_percent, train_sentences_10_percent, train_labels_90_percent, train_labels_10_percent = train_test_split(np.array(train_sentences),
train_labels,
test_size=0.1,
random_state=42)
```

In [91]:

```
# Check length of 10 percent datasets
print(f"Total training examples: {len(train_sentences)}")
print(f"Length of 10% training examples: {len(train_sentences_10_percent)}")
```

Total training examples: 6851  
Length of 10% training examples: 686

Because we've selected a random subset of the training samples, the classes should be roughly balanced (as they are in the full training dataset).

In [92]:

```
# Check the number of targets in our subset of data
# (this should be close to the distribution of labels in the original train_labels)
pd.Series(train_labels_10_percent).value_counts()
```

Out[92]:

0	415
1	271
	dtype: int64

To make sure we're making an appropriate comparison between our model's ability to learn from the full training set and 10% subset, we'll clone our USE model (`model_6`) using the `tf.keras.models.clone_model()` method.

Doing this will create the same architecture but reset the learned weights of the clone target (pretrained weights from the USE will remain but all others will be reset).

In [93]:

```
# Clone model_6 but reset weights
model_7 = tf.keras.models.clone_model(model_6)

# Compile model
model_7.compile(loss="binary_crossentropy",
                 optimizer=tf.keras.optimizers.Adam(),
                 metrics=["accuracy"])

# Get a summary (will be same as model_6)
model_7.summary()
```

Model: "model\_6\_USE"

Layer (type)	Output Shape	Param #
=====		
USE (KerasLayer)	(None, 512)	256797824
dense_5 (Dense)	(None, 64)	32832
dense_6 (Dense)	(None, 1)	65
=====		
Total params:	256,830,721	
Trainable params:	32,897	
Non-trainable params:	256,797,824	

Notice the layout of `model_7` is the same as `model_6`. Now let's train the newly created model on our 10% training data subset.

In [94]:

```
# Fit the model to 10% of the training data
model_7_history = model_7.fit(x=train_sentences_10_percent,
                               y=train_labels_10_percent,
                               epochs=5,
                               validation_data=(val_sentences, val_labels),
                               callbacks=[create_tensorboard_callback(SAVE_DIR, "10_percent_tf_hub_sentence_encoder")])
```

Saving TensorBoard log files to: `model_logs/10_percent_tf_hub_sentence_encoder/20210923-052925`  
Epoch 1/5  
22/22 [=====] - 6s 147ms/step - loss: 0.6716 - accuracy: 0.6574 - val\_loss: 0.6526 - val\_accuracy: 0.6903  
Epoch 2/5  
22/22 [=====] - 1s 47ms/step - loss: 0.5972 - accuracy: 0.8032 - val\_loss: 0.5944 - val\_accuracy: 0.7362  
Epoch 3/5  
22/22 [=====] - 1s 45ms/step - loss: 0.5178 - accuracy: 0.8149 - val\_loss: 0.5398 - val\_accuracy: 0.7625  
Epoch 4/5  
22/22 [=====] - 1s 31ms/step - loss: 0.4526 - accuracy: 0.8265 - val\_loss: 0.5084 - val\_accuracy: 0.7677  
Epoch 5/5  
22/22 [=====] - 1s 46ms/step - loss: 0.4094 - accuracy: 0.8382 - val\_loss: 0.4915 - val\_accuracy: 0.7703

Due to the smaller amount of training data, training happens even quicker than before.

Let's evaluate our model's performance after learning on 10% of the training data.

In [95]:

```
# Make predictions with the model trained on 10% of the data
model_7_pred_probs = model_7.predict(val_sentences)
model_7_pred_probs[:10]
```

Out[95]:

```
array([[0.24043235],
       [0.76837844],
       [0.90137184],
       [0.29067948],
       [0.57149994],
       [0.8356514 ],
       [0.8062943 ],
       [0.83358175],
       [0.85545677],
       [0.11749928]], dtype=float32)
```

In [96]:

```
# Convert prediction probabilities to labels
model_7_preds = tf.squeeze(tf.round(model_7_pred_probs))
model_7_preds[:10]
```

Out[96]:

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=array([0., 1., 1., 0., 1., 1., 1., 1., 1., 0.], dtype=float32)>
```

In [97]:

```
# Calculate model results
model_7_results = calculate_results(val_labels, model_7_preds)
model_7_results
```

Out[97]:

```
{'accuracy': 77.03412073490814,
 'f1': 0.7667059443150692,
 'precision': 0.7755630249535594,
 'recall': 0.7703412073490814}
```

In [98]:

```
# Compare to baseline
compare_baseline_to_new_results(baseline_results, model_7_results)
```

```
Baseline accuracy: 79.27, New accuracy: 77.03, Difference: -2.23
Baseline precision: 0.81, New precision: 0.78, Difference: -0.04
Baseline recall: 0.79, New recall: 0.77, Difference: -0.02
Baseline f1: 0.79, New f1: 0.77, Difference: -0.02
```

## Comparing the performance of each of our models

Woah. We've come a long way! From training a baseline to several deep models.

Now it's time to compare our model's results.

But just before we do, it's worthwhile mentioning, this type of practice is a standard deep learning workflow. Training various different models, then comparing them to see which one performed best and continuing to train it if necessary.

The important thing to note is that for all of our modelling experiments we used the same training data (except for `model_7` where we used 10% of the training data).

To visualize our model's performances, let's create a pandas DataFrame with our results dictionaries and then plot it.

In [99]:

```
# Combine model results into a DataFrame
all_model_results = pd.DataFrame({ "baseline": baseline_results,
                                    "simple_dense": model_1_results,
                                    "lstm": model_2_results,
                                    "gru": model_3_results,
                                    "bidirectional": model_4_results,
                                    "conv1d": model_5_results,
                                    "tf_hub_sentence_encoder": model_6_results,
                                    "tf_hub_10_percent_data": model_7_results})
all_model_results = all_model_results.transpose()
all_model_results
```

Out [99]:

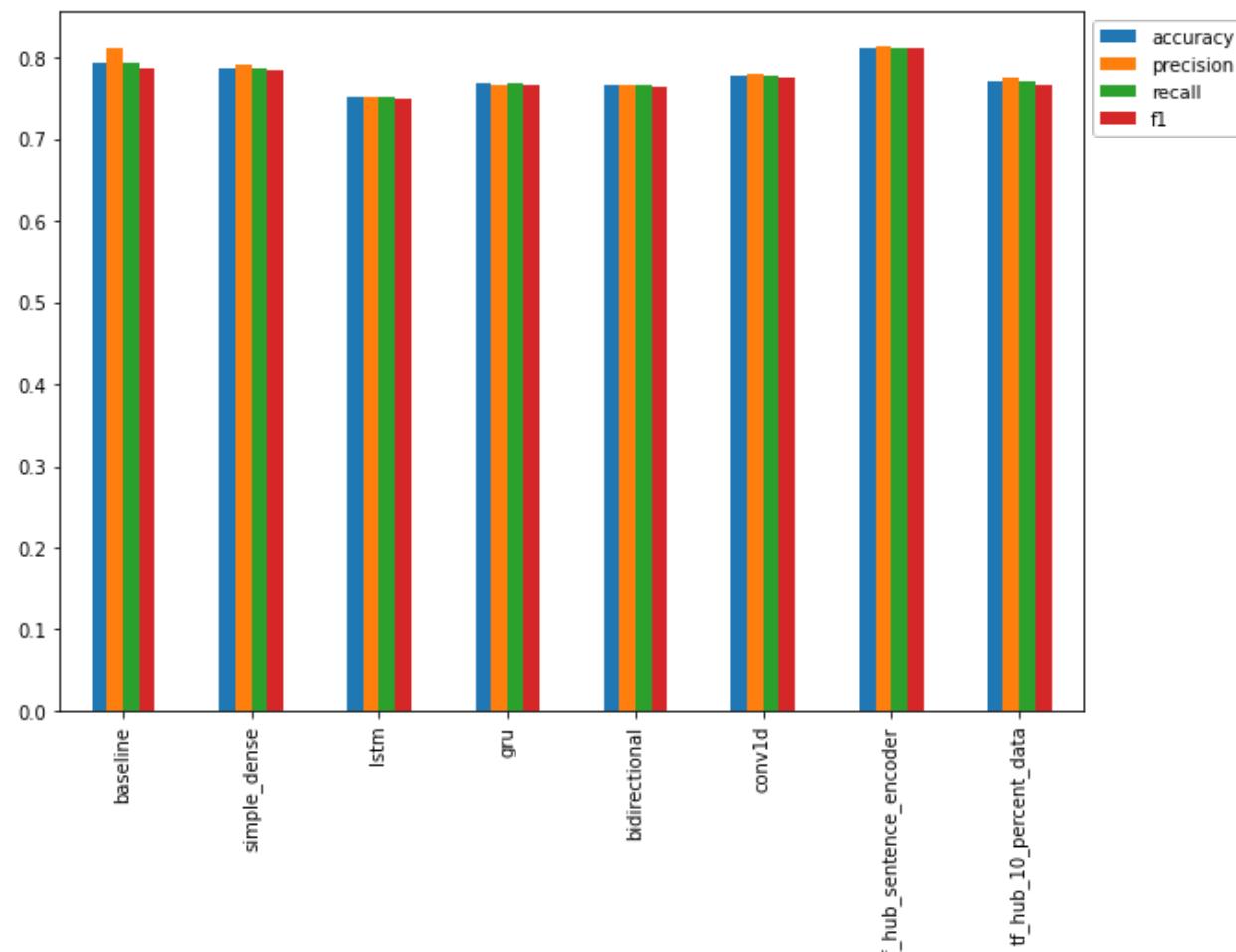
	<b>accuracy</b>	<b>precision</b>	<b>recall</b>	<b>f1</b>
<b>baseline</b>	79.265092	0.811139	0.792651	0.786219
<b>simple_dense</b>	78.740157	0.791492	0.787402	0.784697
<b>lstm</b>	75.065617	0.751008	0.750656	0.748927
<b>gru</b>	76.771654	0.767545	0.767717	0.766793
<b>bidirectional</b>	76.640420	0.766590	0.766404	0.765121
<b>conv1d</b>	77.821522	0.780752	0.778215	0.775881
<b>tf_hub_sentence_encoder</b>	81.233596	0.814880	0.812336	0.810687
<b>tf_hub_10_percent_data</b>	77.034121	0.775563	0.770341	0.766706

In [100]:

```
# Reduce the accuracy to same scale as other metrics
all_model_results["accuracy"] = all_model_results["accuracy"]/100
```

In [101]:

```
# Plot and compare all of the model results
all_model_results.plot(kind="bar", figsize=(10, 7)).legend(bbox_to_anchor=(1.0, 1.0));
```

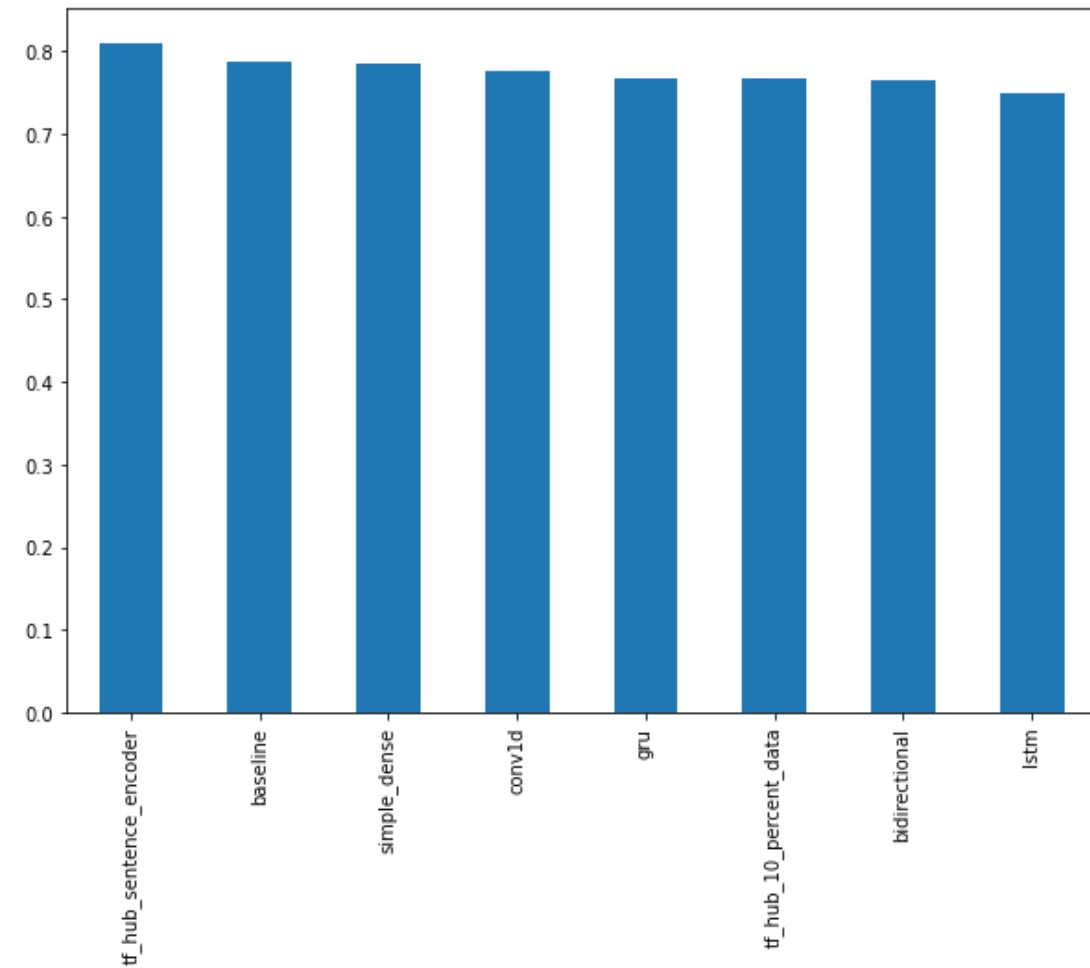


Looks like our pretrained USE TensorFlow Hub models have the best performance, even the one with only 10% of the training data seems to outperform the other models. This goes to show the power of transfer learning.

How about we drill down and get the F1-score's of each model?

In [102]:

```
# Sort model results by f1-score
all_model_results.sort_values("f1", ascending=False) ["f1"].plot(kind="bar", figsize=(10, 7));
```



Drilling down into a single metric we see our USE TensorFlow Hub models performing better than all of the other models. Interestingly, the baseline's F1-score isn't too far off the rest of the deeper models.

We can also visualize all of our model's training logs using TensorBoard.dev.

In [103]:

```
# # View tensorboard logs of transfer learning modelling experiments (should be 4 models)
# # Upload TensorBoard dev records
# !tensorboard dev upload --logdir ./model_logs \
#   --name "NLP modelling experiments" \
#   --description "A series of different NLP modellings experiments with various models"
#   --one_shot # exits the uploader when upload has finished
```

The TensorBoard logs of the different modelling experiments we ran can be viewed here:

<https://tensorboard.dev/experiment/LkoAakb7QIKBZ0RL97cXbw/>

In [104]:

```
# If you need to remove previous experiments, you can do so using the following command
# !tensorboard dev delete --experiment_id EXPERIMENT_ID_TO_DELETE
```

# Combining our models (model ensembling/stacking)

Many production systems use an **ensemble** (multiple different models combined) of models to make a prediction.

The idea behind model stacking is that if several uncorrelated models agree on a prediction, then the prediction must be more robust than a prediction made by a singular model.

The keyword in the sentence above is **uncorrelated**, which is another way of saying, different types of models. For example, in our case, we might combine our baseline, our bidirectional model and our TensorFlow Hub USE model.

Although these models are all trained on the same data, they all have a different way of finding patterns.

If we were to use three similarly trained models, such as three LSTM models, the predictions they output will likely be very similar.

Think of it as trying to decide where to eat with your friends. If you all have similar tastes, you'll probably all pick the same restaurant. But if you've all got different tastes and still end up picking the same restaurant, the restaurant must be good.

Since we're working with a classification problem, there are a few of ways we can combine our models:

1. **Averaging** - Take the output prediction probabilities of each model for each sample, combine them and then average them.
2. **Majority vote (mode)** - Make class predictions with each of your models on all samples, the predicted class is the one in majority. For example, if three different models predict [1, 0, 1] respectively, the majority class is 1, therefore, that would be the predicted label.
3. **Model stacking** - Take the outputs of each of your chosen models and use them as inputs to another model.

**Resource:** The above methods for model stacking/ensembling were adapted from Chapter 6 of the [Machine Learning Engineering Book](#) by Andriy Burkov. If you're looking to enter the field of machine learning engineering, not only building models but production-scale machine learning systems, I'd highly recommend reading it in its entirety.

Again, the concept of model stacking is best seen in action.

We're going to combine our baseline model (`model_0`), LSTM model (`model_2`) and our USE model trained on the full training data (`model_6`) by averaging the combined prediction probabilities of each.

In [105]:

```
# Get mean pred probs for 3 models
baseline_pred_probs = np.max(model_0.predict_proba(val_sentences), axis=1) # get the prediction probabilities from baseline model
combined_pred_probs = baseline_pred_probs + tf.squeeze(model_2_pred_probs, axis=1) + tf.squeeze(model_6_pred_probs)
combined_preds = tf.round(combined_pred_probs/3) # average and round the prediction probabilities to get prediction classes
combined_preds[:20]
```

Out[105]:

```
<tf.Tensor: shape=(20,), dtype=float32, numpy=
array([0., 1., 1., 0., 0., 1., 1., 1., 1., 0., 0., 0., 1., 0., 0., 0.,
       0., 0., 1.], dtype=float32)>
```

Wonderful! We've got a combined predictions array of different classes, let's evaluate them against the true labels and add our stacked model's results to our `all_model_results` DataFrame.

In [106]:

```
# Calculate results from averaging the prediction probabilities
ensemble_results = calculate_results(val_labels, combined_preds)
ensemble_results
```

Out[106]:

```
{'accuracy': 78.08398950131233,  
'f1': 0.7805169025578647,  
'precision': 0.7805216999297674,  
'recall': 0.7808398950131233}
```

In [107]:

```
# Add our combined model's results to the results DataFrame  
all_model_results.loc["ensemble_results"] = ensemble_results
```

In [108]:

```
# Convert the accuracy to the same scale as the rest of the results  
all_model_results.loc["ensemble_results"]["accuracy"] = all_model_results.loc["ensemble_results"]["accuracy"]/100
```

In [109]:

```
all_model_results
```

Out[109]:

	accuracy	precision	recall	f1
<b>baseline</b>	0.792651	0.811139	0.792651	0.786219
<b>simple_dense</b>	0.787402	0.791492	0.787402	0.784697
<b>lstm</b>	0.750656	0.751008	0.750656	0.748927
<b>gru</b>	0.767717	0.767545	0.767717	0.766793
<b>bidirectional</b>	0.766404	0.766590	0.766404	0.765121
<b>conv1d</b>	0.778215	0.780752	0.778215	0.775881
<b>tf_hub_sentence_encoder</b>	0.812336	0.814880	0.812336	0.810687
<b>tf_hub_10_percent_data</b>	0.770341	0.775563	0.770341	0.766706
<b>ensemble_results</b>	0.780840	0.780522	0.780840	0.780517

How did the stacked model go against the other models?

**Note:** It seems many of our model's results are similar. This may mean there are some limitations to what can be learned from our data. When many of your modelling experiments return similar results, it's a good idea to revisit your data, we'll do this shortly.

## Saving and loading a trained model

Although training time didn't take very long, it's good practice to save your trained models to avoid having to retrain them.

Saving your models also enables you to export them for use elsewhere outside of your notebooks, such as in a web application.

There are two main ways of [saving a model in TensorFlow](#):

1. The `HDF5` format.
2. The `SavedModel` format (default).

Let's take a look at both.

In [110]:

```
# Save TF Hub Sentence Encoder model to HDF5 format  
model_6.save("model_6.h5")
```

If you save a model as a `HDF5`, when loading it back in, you need to let [TensorFlow know about any custom objects you've used](#) (e.g. components which aren't built from pure TensorFlow, such as TensorFlow Hub components).

In [111]:

```
# Load model with custom Hub Layer (required with HDF5 format)
loaded_model_6 = tf.keras.models.load_model("model_6.h5",
                                            custom_objects={"KerasLayer": hub.KerasLayer
})
```

In [112]:

```
# How does our loaded model perform?
loaded_model_6.evaluate(val_sentences, val_labels)
```

24/24 [=====] - 1s 14ms/step - loss: 0.4309 - accuracy: 0.8123

Out[112]:

```
[0.43088313937187195, 0.8123359680175781]
```

Calling the `save()` method on our target model and passing it a filepath allows us to save our model in the `SavedModel` format.

In [113]:

```
# Save TF Hub Sentence Encoder model to SavedModel format (default)
model_6.save("model_6_SavedModel_format")
```

WARNING:absl:Function `wrapped\_model` contains input name(s) USE\_input with unsupported characters which will be renamed to use\_input in the SavedModel.

INFO:tensorflow:Assets written to: model\_6\_SavedModel\_format/assets

INFO:tensorflow:Assets written to: model\_6\_SavedModel\_format/assets

If you use `SavedModel` format (default), you can reload your model without specifying custom objects using the [`tensorflow.keras.models.load\_model\(\)` function](#).

In [114]:

```
# Load TF Hub Sentence Encoder SavedModel
loaded_model_6_SavedModel = tf.keras.models.load_model("model_6_SavedModel_format")
```

In [115]:

```
# Evaluate loaded SavedModel format
loaded_model_6_SavedModel.evaluate(val_sentences, val_labels)
```

24/24 [=====] - 1s 14ms/step - loss: 0.4309 - accuracy: 0.8123

Out[115]:

```
[0.43088313937187195, 0.8123359680175781]
```

As you can see saving and loading our model with either format results in the same performance.

Question: Should you used the `SavedModel` format or `HDF5` format?

For most use cases, the `SavedModel` format will suffice. However, this is a TensorFlow specific standard. If you need a more general-purpose data standard, `HDF5` might be better. For more, check out the [TensorFlow documentation on saving and loading models](#).

Finding the most wrong examples

We mentioned before that if many of our modelling experiments are returning similar results, despite using different kinds of models, it's a good idea to return to the data and inspect why this might be.

One of the best ways to inspect your data is to sort your model's predictions and find the samples it got *most* wrong, meaning, what predictions had a high prediction probability but turned out to be wrong.

Once again, visualization is your friend. Visualize, visualize, visualize.

To make things visual, let's take our best performing model's prediction probabilities and classes along with the validation samples (text and ground truth labels) and combine them in a pandas DataFrame.

- If our best model still isn't perfect, what examples is it getting wrong?
- Which ones are the *most* wrong?
- Are there some labels which are wrong? E.g. the model gets it right but the ground truth label doesn't reflect this

In [116] :

```
# Create dataframe with validation sentences and best performing model predictions
val_df = pd.DataFrame({"text": val_sentences,
                       "target": val_labels,
                       "pred": model_6_preds,
                       "pred_prob": tf.squeeze(model_6_pred_probs)})
```

Out[116] :

		text	target	pred	pred_prob
0		DFR EP016 Monthly Meltdown - On Dnbheaven 2015...	0	0.0	0.144432
1		FedEx no longer to transport bioterror germs i...	0	1.0	0.727150
2		Gunmen kill four in El Salvador bus attack: Su...	1	1.0	0.985666
3		@camilacabello97 Internally and externally scr...	1	0.0	0.197409
4		Radiation emergency #preparedness starts with ...	1	1.0	0.734170

Oh yeah! Now let's find our model's wrong predictions (where `target != pred`) and sort them by their prediction probability (the `pred_prob` column).

In [117] :

```
# Find the wrong predictions and sort by prediction probabilities
most_wrong = val_df[val_df["target"] != val_df["pred"]].sort_values("pred_prob", ascending=False)
most_wrong[:10]
```

Out[117] :

		text	target	pred	pred_prob
31		? High Skies - Burning Buildings ? http://t.co...	0	1.0	0.910481
759		FedEx will no longer transport bioterror patho...	0	1.0	0.864676
209		Ashes 2015: Australia's collapse at Trent Br...	0	1.0	0.837961
393		@SonofLiberty357 all illuminated by the bright...	0	1.0	0.836361
628		@noah_anynname That's where the concentration c...	0	1.0	0.835225
49		@madonnamking RSPCA site multiple 7 story high...	0	1.0	0.834875
109	[55436]	1950 LIONEL TRAINS SMOKE LOCOMOTIVES W...	0	1.0	0.800890
251		@AshGhebranious civil rights continued in the ...	0	1.0	0.782611
698		â€˜EMGN-AFRICAâ€™ pin:263789F4 â€˜ Correction: Ten...	0	1.0	0.782433

Finally, we can write some code to visualize the sample text, truth label, prediction class and prediction probability. Because we've sorted our samples by prediction probability, viewing samples from the head of our most\_wrong DataFrame will show us false positives.

A reminder:

- 0 = Not a real disaster Tweet
- 1 = Real disaster Tweet

In [118]:

```
# Check the false positives (model predicted 1 when should've been 0)
for row in most_wrong[:10].itertuples(): # loop through the top 10 rows (change the index
                                         to view different rows)
    _, text, target, pred, prob = row
    print(f"Target: {target}, Pred: {int(pred)}, Prob: {prob}")
    print(f"Text:\n{text}\n")
    print("----\n")
```

```
Target: 0, Pred: 1, Prob: 0.9104808568954468
Text:
? High Skies - Burning Buildings ? http://t.co/uVq41i3Kx2 #nowplaying

-----
Target: 0, Pred: 1, Prob: 0.8646755218505859
Text:
FedEx will no longer transport bioterror pathogens in wake of anthrax lab mishaps http://t.co/lHpgxc4b8J

-----
Target: 0, Pred: 1, Prob: 0.8379608988761902
Text:
Ashes 2015: Australia's collapse at Trent Bridge among worst in history: England bundled out Australia for 60 ... http://t.co/t5TrhjUAU0

-----
Target: 0, Pred: 1, Prob: 0.8363614082336426
Text:
@SonofLiberty357 all illuminated by the brightly burning buildings all around the town!

-----
Target: 0, Pred: 1, Prob: 0.8352250456809998
Text:
@noah_anynname That's where the concentration camps and mass murder come in.

EVERY. FUCKING. TIME.

-----
Target: 0, Pred: 1, Prob: 0.8348745107650757
Text:
@madonna_mking RSPCA site multiple 7 story high rise buildings next to low density character residential in an area that floods

-----
Target: 0, Pred: 1, Prob: 0.800889790058136
Text:
[55436] 1950 LIONEL TRAINS SMOKE LOCOMOTIVES WITH MAGNE-TRACTION INSTRUCTIONS http://t.co/xEZBs3sq0y http://t.co/C2x0QoKGly

-----
Target: 0, Pred: 1, Prob: 0.7826112508773804
```

Text:  
@AshGhebranious civil rights continued in the 60s. And what about trans-generational trauma? if anything we should listen to the Americans.

----

Target: 0, Pred: 1, Prob: 0.7824334502220154

Text:  
â€œMGN-AFRICAâ€ pin:263789F4 â€ Correction: Tent Collapse Story: Correction: Tent Collapse story â€ http://t.co/fDJUYvZMrv @wizkidayo

----

Target: 0, Pred: 1, Prob: 0.7713427543640137

Text:  
The Sound of Arson

----

**We can view the bottom end of our `most_wrong` DataFrame to inspect false negatives (model predicts 0, not a real disaster Tweet, when it should've predicted 1, real disaster Tweet).**

In [119]:

```
# Check the most wrong false negatives (model predicted 0 when should've predict 1)
for row in most_wrong[-10:].itertuples():
    _, text, target, pred, prob = row
    print(f"Target: {target}, Pred: {int(pred)}, Prob: {prob}")
    print(f"Text:\n{text}\n")
    print("----\n")
```

Target: 1, Pred: 0, Prob: 0.06304337829351425

Text:  
@BoyInAHorsemask its a panda trapped in a dogs body

----

Target: 1, Pred: 0, Prob: 0.06279505044221878

Text:  
going to redo my nails and watch behind the scenes of desolation of smaug ayyy

----

Target: 1, Pred: 0, Prob: 0.06060810014605522

Text:  
VICTORINOX SWISS ARMY DATE WOMEN'S RUBBER MOP WATCH 241487 http://t.co/yFy3nkkcoH http://t.co/KNEhVvOHVK

----

Target: 1, Pred: 0, Prob: 0.0573178268969059

Text:  
@willienelson We need help! Horses will die!Please RT & sign petition!Take a stand & be a voice for them! #gilbert23 https://t.co/e8dl1lNCVu

----

Target: 1, Pred: 0, Prob: 0.04535556212067604

Text:  
You can never escape me. Bullets don't harm me. Nothing harms me. But I know pain. I know pain. Sometimes I share it. With someone like you.

----

Target: 1, Pred: 0, Prob: 0.04145137220621109

Text:  
I get to smoke my shit in peace

----

Target: 1, Pred: 0, Prob: 0.03926113247871399  
Text:  
@SoonerMagic\_ I mean I'm a fan but I don't need a girl sounding off like a damn siren  
----  
Target: 1, Pred: 0, Prob: 0.0385933592915535  
Text:  
Why are you deluged with low self-image? Take the quiz: <http://t.co/XsPqdOrIqj> <http://t.co/oCQYvFR4UCy>  
----  
Target: 1, Pred: 0, Prob: 0.03627230226993561  
Text:  
Reddit Will Now Quarantine <http://t.co/pkUAMXw6pm> #onlinecommunities #reddit #amageddon  
#freespeech #Business <http://t.co/PAWvNJ4sAP>  
----  
Target: 1, Pred: 0, Prob: 0.032887961715459824  
Text:  
Ron & Fez - Dave's High School Crush <https://t.co/aN3W16c8F6> via @YouTube  
----

**Do you notice anything interesting about the most wrong samples?**

**Are the ground truth labels correct? What do you think would happen if we went back and corrected the labels which aren't?**

## Making predictions on the test dataset

Alright we've seen how our model's perform on the validation set.

But how about the test dataset?

We don't have labels for the test dataset so we're going to have to make some predictions and inspect them for ourselves.

Let's write some code to make predictions on random samples from the test dataset and visualize them.

In [120]:

```
# Making predictions on the test dataset
test_sentences = test_df["text"].to_list()
test_samples = random.sample(test_sentences, 10)
for test_sample in test_samples:
    pred_prob = tf.squeeze(model_6.predict([test_sample])) # has to be list
    pred = tf.round(pred_prob)
    print(f"Pred: {int(pred)}, Prob: {pred_prob}")
    print(f"Text:\n{test_sample}\n")
    print("----\n")
```

Pred: 1, Prob: 0.538340151309967  
Text:  
Flash Flood Watch in effect through 7:00am Thursday morning/12:00pm Thursday afternoon.  
For: Perry Wayne Cape... <http://t.co/fs7vro5seS>  
----

Pred: 1, Prob: 0.9250481128692627  
Text:  
NONSENSE >> famine memories -- strong exaggeration of Ukrainian MSM  
#ukraine #russia #????????? #sanctions <https://t.co/dDOTd7W2o8>  
----

Pred: 1, Prob: 0.894009351730346 /  
Text:  
New warning for Central Hills 1' hail 60 mph winds. NOT affecting Sturgis but could later tonight. #KOTAWeather http://t.co/E8oUxVKuTE

----  
  
Pred: 0, Prob: 0.07600127905607224  
Text:  
@imaginator1dx currently reading after. as you can see after we collided is on my dresser waiting to get read http://t.co/QwrASZ6LHO

----  
  
Pred: 0, Prob: 0.026800094172358513  
Text:  
Don't ruin a good today by thinking about a bad yesterday ????  
----

Pred: 0, Prob: 0.21049749851226807  
Text:  
I hope I get electrocuted today at work

----  
  
Pred: 0, Prob: 0.07167388498783112  
Text:  
http://t.co/16EC1WrW84 Asics GT-II Super Red 2.0 11 Ronnie Fieg Kith Red White 3M x gel grey volcano 2

----  
  
Pred: 0, Prob: 0.04366963729262352  
Text:  
I swear my eyes be bloody red but bitch I feel amazing.

----  
  
Pred: 1, Prob: 0.9714540839195251  
Text:  
Japan marks 70th anniversary of Hiroshima atomic bombing: Bells tolled in Hiroshima on Thursday as Japan marke... http://t.co/IqAIRPdIhg

----  
  
Pred: 0, Prob: 0.10844964534044266  
Text:  
@USCOURT If 90BLKs&amp;8WHTs colluded 2 take WHT F @USAgov AUTH Hostage&amp;2 make her look BLK w/Bioterrorism&amp;use her lgl/org IDis ID still hers?

## How do our model's predictions look on the test dataset?

**It's important to do these kind of visualization checks as often as possible to get a glance of how your model performs on unseen data and subsequently how it might perform on the real test: Tweets from the wild.**

## Predicting on Tweets from the wild

How about we find some Tweets and use our model to predict whether or not they're about a disaster or not?

To start, let's take one of my own [Tweets on living life like an ensemble model.](#)

In [121]:

```
# Turn Tweet into string
daniels_tweet = "Life like an ensemble: take the best choices from others and make your o
```

**Now we'll write a small function to take a model and an example sentence and return a prediction.**

In [122]:

```
def predict_on_sentence(model, sentence):
    """
    Uses model to make a prediction on sentence.

    Returns the sentence, the predicted label and the prediction probability.
    """
    pred_prob = model.predict([sentence])
    pred_label = tf.squeeze(tf.round(pred_prob)).numpy()
    print(f"Pred: {pred_label} {\"(real disaster)\" if pred_label > 0 else \"(not real disaster)\"}, Prob: {pred_prob[0][0]}")
    print(f"Text:\n{sentence}")
```

**Great! Time to test our model out.**

In [123]:

```
# Make a prediction on Tweet from the wild
predict_on_sentence(model=model_6, # use the USE model
                    sentence=daniels_tweet)
```

Pred: 0.0 (not real disaster) Prob: 0.046233948320150375  
Text:  
Life like an ensemble: take the best choices from others and make your own

**Woohoo! Our model predicted correctly. My Tweet wasn't about a diaster.**

**How about we find a few Tweets about actual disasters?**

**Such as the following two Tweets about the 2020 Beirut explosions.**

In [124]:

```
# Source - https://twitter.com/BeirutCityGuide/status/1290696551376007168
beirut_tweet_1 = "Reports that the smoke in Beirut sky contains nitric acid, which is toxic. Please share and refrain from stepping outside unless urgent. #Lebanon"

# Source - https://twitter.com/BeirutCityGuide/status/1290773498743476224
beirut_tweet_2 = "#Beirut declared a \"devastated city\", two-week state of emergency officially declared. #Lebanon"
```

In [125]:

```
# Predict on disaster Tweet 1
predict_on_sentence(model=model_6,
                    sentence=beirut_tweet_1)
```

Pred: 1.0 (real disaster) Prob: 0.9625465869903564  
Text:  
Reports that the smoke in Beirut sky contains nitric acid, which is toxic. Please share and refrain from stepping outside unless urgent. #Lebanon

In [126]:

```
# Predict on disaster Tweet 2
predict_on_sentence(model=model_6,
                    sentence=beirut_tweet_2)
```

Pred: 1.0 (real disaster) Prob: 0.9678557515144348  
Text:  
#Beirut declared a "devastated city", two-week state of emergency officially declared. #Lebanon

**Looks like our model is performing as expected, predicting both of the disaster Tweets as actual disasters.**

**Note:** The above examples are cherry-picked and are cases where you'd expect a model to function at high performance. For actual production systems, you'll want to continually perform tests to see how your model is performing.

## The speed/score tradeoff

One of the final tests we're going to do is to find the speed/score tradeoffs between our best model and baseline model.

Why is this important?

Although it can be tempting to just choose the best performing model you find through experimentation, this model might not actually work in a production setting.

Put it this way, imagine you're Twitter and receive 1 million Tweets per hour (this is a made up number, the actual number is much higher). And you're trying to build a disaster detection system to read Tweets and alert authorities with details about a disaster in close to real-time.

Compute power isn't free so you're limited to a single compute machine for the project. On that machine, one of your models makes 10,000 predictions per second at 80% accuracy whereas another one of your models (a larger model) makes 100 predictions per second at 85% accuracy.

Which model do you choose?

Is the second model's performance boost worth missing out on the extra capacity?

Of course, there are many options you could try here, such as sending as many Tweets as possible to the first model and then sending the ones which the model is least certain of to the second model.

The point here is to illustrate the best model you find through experimentation, might not be the model you end up using in production.

To make this more concrete, let's write a function to take a model and a number of samples and time how long the given model takes to make predictions on those samples.

In [127]:

```
# Calculate the time of predictions
import time
def pred_timer(model, samples):
    """
    Times how long a model takes to make predictions on samples.

    Args:
    ----
    model = a trained model
    sample = a list of samples

    Returns:
    ----
    total_time = total elapsed time for model to make predictions on samples
    time_per_pred = time in seconds per single sample
    """
    start_time = time.perf_counter() # get start time
    model.predict(samples) # make predictions
    end_time = time.perf_counter() # get finish time
    total_time = end_time-start_time # calculate how long predictions took to make
    time_per_pred = total_time/len(val_sentences) # find prediction time per sample
    return total_time, time_per_pred
```

Looking good!

Now let's use our `pred_timer()` function to evaluate the prediction times of our best performing model (`model_6`) and our baseline model (`model_0`).

In [128]:

```
# Calculate TF Hub Sentence Encoder prediction times
model_6_total_pred_time, model_6_time_per_pred = pred_timer(model_6, val_sentences)
model_6_total_pred_time, model_6_time_per_pred
```

Out[128]:

```
(0.3529780789999677, 0.0004632258254592752)
```

In [129]:

```
# Calculate Naive Bayes prediction times
baseline_total_pred_time, baseline_time_per_pred = pred_timer(model_0, val_sentences)
baseline_total_pred_time, baseline_time_per_pred
```

Out[129]:

```
(0.018752853000023606, 2.4610043307117593e-05)
```

**It seems with our current hardware (in my case, I'm using a Google Colab notebook) our best performing model takes over 10x the time to make predictions as our baseline model.**

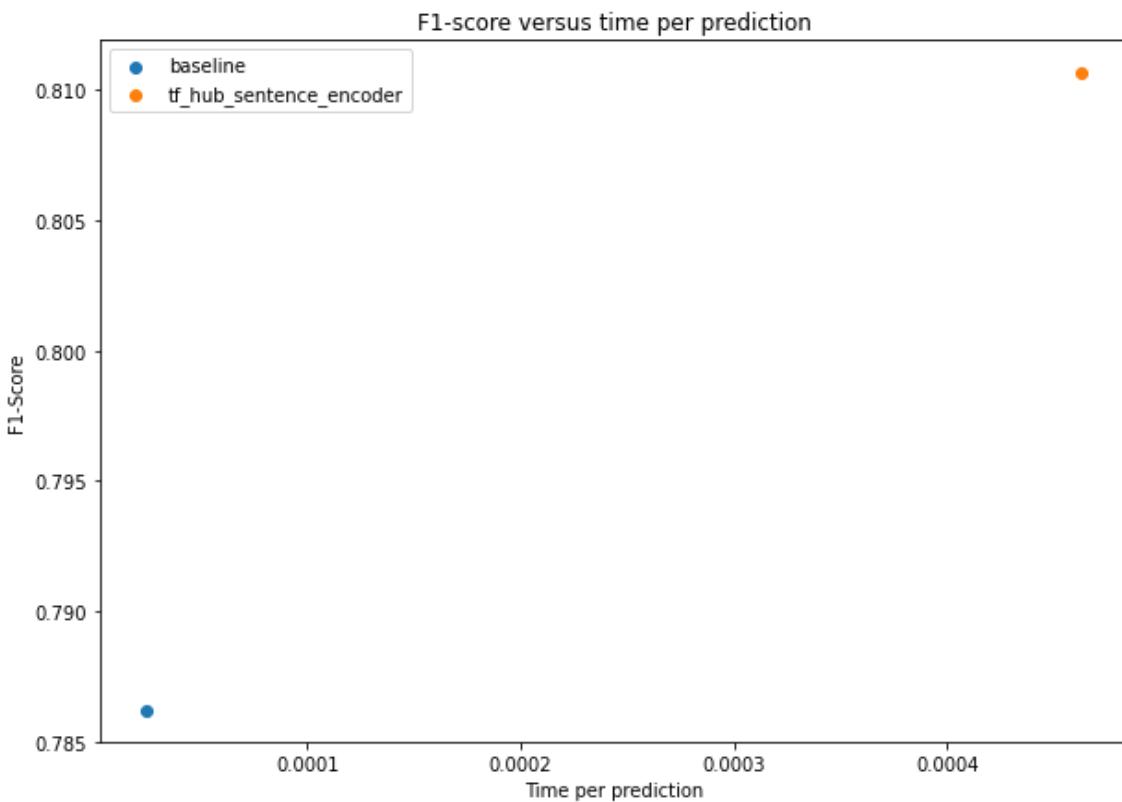
**Is that extra prediction time worth it?**

Let's compare time per prediction versus our model's F1-scores.

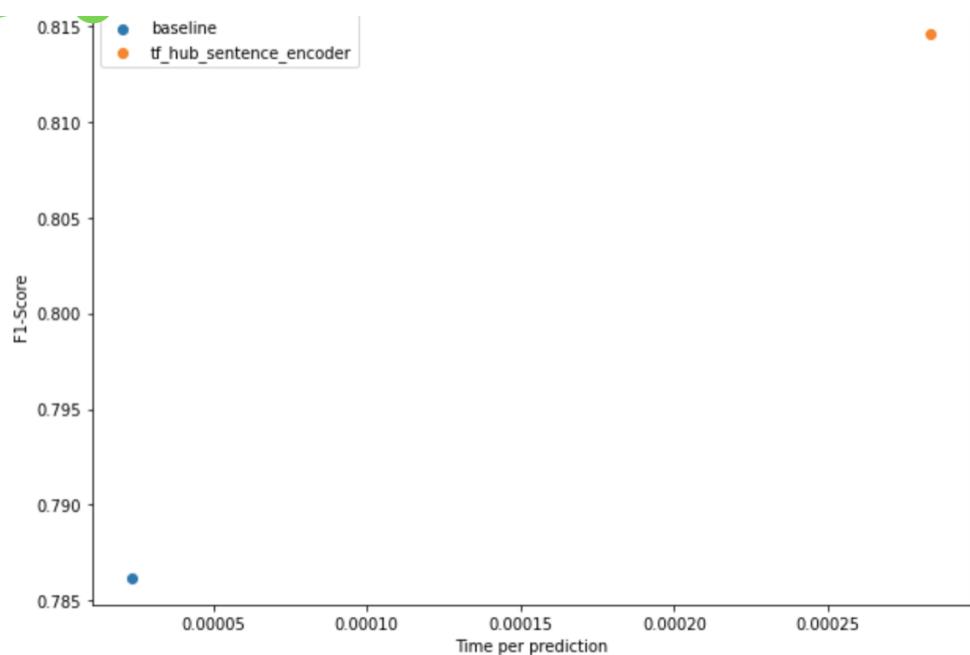
In [130]:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 7))
plt.scatter(baseline_time_per_pred, baseline_results["f1"], label="baseline")
plt.scatter(model_6_time_per_pred, model_6_results["f1"], label="tf_hub_sentence_encoder")
plt.legend()
plt.title("F1-score versus time per prediction")
plt.xlabel("Time per prediction")
plt.ylabel("F1-Score");
```



Ideal position for speed/performance (high performance + high speed)



*Ideal position for speed and performance tradeoff model (fast predictions with great results).*

Of course, the ideal position for each of these dots is to be in the top left of the plot (low time per prediction, high F1-score).

In our case, there's a clear tradeoff for time per prediction and performance. Our best performing model takes an order of magnitude longer per prediction but only results in a few F1-score point increase.

This kind of tradeoff is something you'll need to keep in mind when incorporating machine learning models into your own applications.

## Exercises

1. Rebuild, compile and train `model_1`, `model_2` and `model_5` using the [Keras Sequential API](#) instead of the Functional API.
2. Retrain the baseline model with 10% of the training data. How does perform compared to the Universal Sentence Encoder model with 10% of the training data?
3. Try fine-tuning the TF Hub Universal Sentence Encoder model by setting `training=True` when instantiating it as a Keras layer.

We can use this encoding layer in place of our `text_vectorizer` and embedding layer

```
sentence_encoder_layer = hub.KerasLayer("https://tfhub.dev/google/universal-sentence-encoder/4",
                                         input_shape=[],
                                         dtype=tf.string,
                                         trainable=True) # turn training on to fine-tune the TensorFlow Hub model
```

1. Retrain the best model you've got so far on the whole training set (no validation split). Then use this trained model to make predictions on the test dataset and format the predictions into the same format as the `sample_submission.csv` file from Kaggle (see the Files tab in Colab for what the `sample_submission.csv` file looks like). Once you've done this, [make a submission to the Kaggle competition](#), how did your model perform?
2. Combine the ensemble predictions using the majority vote (mode), how does this perform compare to averaging the prediction probabilities of each model?
3. Make a confusion matrix with the best performing model's predictions on the validation set and the validation ground truth labels.

## Extra-curriculum

To practice what you've learned, a good idea would be to spend an hour on 3 of the following (3-hours total, you could go through them all if you want) and then write a blog post about what you've learned.

- For an overview of the different problems within NLP and how to solve them read through:
  - [A Simple Introduction to Natural Language Processing](#)
  - [How to solve 90% of NLP problems: a step-by-step guide](#)
- Go through [MIT's Recurrent Neural Networks lecture](#). This will be one of the greatest additions to what's happening behind the RNN model's you've been building.
- Read through the [word embeddings page on the TensorFlow website](#). Embeddings are such a large part of NLP. We've covered them throughout this notebook but extra practice would be well worth it. A good exercise would be to write out all the code in the guide in a new notebook.
- For more on RNN's in TensorFlow, read and reproduce [the TensorFlow RNN guide](#). We've covered many of the concepts in this guide, but it's worth writing the code again for yourself.
- Text data doesn't always come in a nice package like the data we've downloaded. So if you're after more on preparing different text sources for being with your TensorFlow deep learning models, it's worth checking out the following:
  - [TensorFlow text loading tutorial](#).
  - [Reading text files with Python](#) by Real Python.
- This notebook has focused on writing NLP code. For a mathematically rich overview of how NLP with Deep Learning happens, read [Standford's Natural Language Processing with Deep Learning lecture notes Part 1](#).
  - For an even deeper dive, you could even do the whole [CS224n](#) (Natural Language Processing with Deep Learning) course.
- Great blog posts to read:
  - Andrei Karpathy's [The Unreasonable Effectiveness of RNNs](#) dives into generating Shakespeare text with RNNs.
  - [Text Classification with NLP: Tf-Idf vs Word2Vec vs BERT](#) by Mauro Di Pietro. An overview of different techniques for turning text into numbers and then classifying it.
  - [What are word embeddings?](#) by Machine Learning Mastery.
- Other topics worth looking into:
  - [Attention mechanisms](#). These are a foundational component of the transformer architecture and also often add improvements to deep NLP models.
  - [Transformer architectures](#). This model architecture has recently taken the NLP world by storm, achieving state of the art on many benchmarks. However, it does take a little more processing to get off the ground, the [HuggingFace Models \(formerly HuggingFace Transformers\) library](#) is probably your best quick start.
    - And now [HuggingFace even have their own course](#) on how their library works! I haven't done it but anything HuggingFace makes is world-class.

□ Resource: See the full set of course materials on GitHub:

<https://github.com/mrdbourke/tensorflow-deep-learning>

## 09. Milestone Project 2: SkimLit

In the previous notebook ([NLP fundamentals in TensorFlow](#)), we went through some fundamental natural language processing concepts. The main ones being **tokenization** (turning words into numbers) and **creating embeddings** (creating a numerical representation of words).

In this project, we're going to be putting what we've learned into practice.

More specifically, we're going to be replicating the deep learning model behind the 2017 paper [PubMed 200k RCT: a Dataset for Sequential Sentence Classification in Medical Abstracts](#).

When it was released, the paper presented a new dataset called PubMed 200k RCT which consists of ~200,000 labelled Randomized Controlled Trial (RCT) abstracts.

The goal of the dataset was to explore the ability for NLP models to classify sentences which appear in sequential order.

In other words, given the abstract of a RCT, what role does each sentence serve in the abstract?

**Nutritional psychiatry: the present state of the evidence**

Wolfgang Marx <sup>1</sup>, Genevieve Moseley <sup>2</sup>, Michael Berk <sup>2</sup>, Felice Jacka <sup>2</sup>

Affiliations + expand  
PMID: 28942748 DOI: 10.1017/S0029665117002026

**Abstract**

Mental illness, including depression, anxiety and bipolar disorder, accounts for a significant proportion of global disability and poses a substantial social, economic and health burden. Treatment is presently dominated by pharmacotherapy, such as antidepressants, and psychotherapy, such as cognitive behavioural therapy; however, such treatments avert less than half of the disease burden, suggesting that additional strategies are needed to prevent and treat mental disorders. There are now consistent mechanistic, observational and interventional data to suggest diet quality may be a modifiable risk factor for mental illness. This review provides an overview of the nutritional psychiatry field. It includes a discussion of the neurobiological mechanisms likely modulated by diet, the use of dietary and nutraceutical interventions in mental disorders, and recommendations for further research. Potential biological pathways related to mental disorders include inflammation, oxidative stress, the gut microbiome, epigenetic modifications and neuroplasticity. Consistent epidemiological evidence, particularly for depression, suggests an association between measures of diet quality and mental health, across multiple populations and age groups; these do not appear to be explained by other demographic, lifestyle factors or reverse causality. Our recently published intervention trial provides preliminary clinical evidence that dietary interventions in clinically diagnosed populations are feasible and can provide significant clinical benefit. Furthermore, nutraceuticals including n-3 fatty acids, folate, S-adenosylmethionine, N-acetyl cysteine and probiotics, among others, are promising avenues for future research. Continued research is now required to investigate the efficacy of intervention studies in large cohorts and within clinically relevant populations, particularly in patients with schizophrenia, bipolar and anxiety disorders.

**Source:** <https://pubmed.ncbi.nlm.nih.gov/28942748/>

**Harder to read**

wall of text broken into chunks

Model

SkimLit

Considerations for a surgical RCT for diffuse low-grade glioma: a survey

Alireza Mansouri <sup>1</sup>, Karanbir Brar <sup>2</sup>, Michael D Cusimano <sup>3</sup>

Affiliations + expand  
PMID: 32537182 PMCID: PMC7274180 (available on 2021-06-01) DOI: 10.1093/nop/npz058

**Abstract**

**Background:** Diffuse low-grade gliomas (DLGGs) are heterogeneous tumors that inevitably differentiate into malignant entities, leading to disability and death. Recently, a shift toward up-front maximal safe resection of DLGGs has been favored. However, this transition is not supported by randomized controlled trial (RCT) data. Here, we sought to survey the neuro-oncology community on considerations for a surgical RCT for DLGGs.

**Methods:** A 21-question survey focusing on a surgical RCT for DLGGs was developed and validated by 2 neurosurgeons. A sample case of a patient for whom management might be debatable was presented to gather additional insight. The survey was disseminated to members of the Society for Neuro-Oncology (SNO) and responses were collected from March 16 to July 10, 2018.

**Results:** A total of 131 responses were collected. Sixty-three of 117 (54%) respondents thought an RCT would not be ethical, 39 of 117 (33%) would consider participating, and 56 of 117 (48%) believed an RCT would be valuable for determining the differing roles of biopsy, surgery, and observation. This was exemplified by an evenly distributed selection of the latter management options for our sample case. Eighty-three of 120 (69.2%) respondents did not believe in equipoise for DLGG patients. Quality of life and overall survival were deemed equally important end points for a putative RCT.

**Conclusions:** Based on our survey, it is evident that management of certain DLGG patients is not well defined and an RCT may be justified. As with any surgical RCT, logistic challenges are anticipated. Robust patient-relevant end points and standardization of perioperative adjuncts are necessary if a surgical RCT is undertaken.

**Source:** <https://pubmed.ncbi.nlm.nih.gov/32537182/>

**Easier to read (skimmable)**

**Example inputs (harder to read abstract from PubMed) and outputs (easier to read abstract) of the model we're going to build. The model will take an abstract wall of text and predict the section label each sentence should have.**

### Model Input

For example, can we train an NLP model which takes the following input (note: the following sample has had all numerical symbols replaced with "@"):

To investigate the efficacy of @ weeks of daily low-dose oral prednisolone in improving pain , mobility , and systemic low-grade inflammation in the short term and whether the effect would be sustained at @ weeks in older adults with moderate to severe knee osteoarthritis ( OA ). A total of @ patients with primary knee OA were randomized @:@ ; @ received @ mg/day of prednisolone and @ received placebo for @ weeks. Outcome measures included pain reduction and improvement in function scores and systemic inflammation markers. Pain was assessed using the visual analog pain scale ( @-@ mm ). Secondary outcome measures included the Western Ontario and McMaster Universities Osteoarthritis Index scores , patient global assessment ( PGA ) of the severity of knee

OA , and @-min walk distance ( @MWD ), Serum levels of interleukin @ ( IL-@ ) , IL-@ , tumor necrosis factor ( TNF ) - , and high-sensitivity C-reactive protein ( hsCRP ) were measured. There was a clinically relevant reduction in the intervention group compared to the placebo group for knee pain , physical function , PGA , and @MWD at @ weeks. The mean difference between treatment arms ( @ % CI ) was @ ( @-@ @ ) , p < @ ; @ ( @-@ @ ) , p < @ ; @ ( @-@ @ ) , p < @ ; and @ ( @-@ @ ) , p < @ , respectively. Further , there was a clinically relevant reduction in the serum levels of IL-@ , IL-@ , TNF - , and hsCRP at @ weeks in the intervention group when compared to the placebo group. These differences remained significant at @ weeks. The Outcome Measures in Rheumatology Clinical Trials-Osteoarthritis Research Society International responder rate was @ % in the intervention group and @ % in the placebo group ( p < @ ). Low-dose oral prednisolone had both a short-term and a longer sustained effect resulting in less knee pain , better physical function , and attenuation of systemic inflammation in older patients with knee OA ( ClinicalTrials.gov identifier NCT@ ).

## Model output

And returns the following output:

```
[ '###24293578\n',
  'OBJECTIVE\tTo investigate the efficacy of @ weeks of daily low-dose oral prednisolone in improving pain , mobility , and systemic low-grade inflammation in the short term and whether the effect would be sustained at @ weeks in older adults with moderate to severe knee osteoarthritis ( OA ) .\n',
  'METHODS\tA total of @ patients with primary knee OA were randomized @:@ ; @ received @ mg/day of prednisolone and @ received placebo for @ weeks .\n',
  'METHODS\tOutcome measures included pain reduction and improvement in function scores and systemic inflammation markers .\n',
  'METHODS\tPain was assessed using the visual analog pain scale ( @-@ mm ) .\n',
  'METHODS\tSecondary outcome measures included the Western Ontario and McMaster Universities Osteoarthritis Index scores , patient global assessment ( PGA ) of the severity of knee OA , and @-min walk distance ( @MWD ) .\n',
  'METHODS\tSerum levels of interleukin @ ( IL-@ ) , IL-@ , tumor necrosis factor ( TNF ) - , and high-sensitivity C-reactive protein ( hsCRP ) were measured .\n',
  'RESULTS\tThere was a clinically relevant reduction in the intervention group compared to the placebo group for knee pain , physical function , PGA , and @MWD at @ weeks .\n',
  'RESULTS\tThe mean difference between treatment arms ( @ % CI ) was @ ( @-@ @ ) , p < @ ; @ ( @-@ @ ) , p < @ ; @ ( @-@ @ ) , p < @ ; and @ ( @-@ @ ) , p < @ , respectively .\n',
  'RESULTS\tFurther , there was a clinically relevant reduction in the serum levels of IL-@ , IL-@ , TNF - , and hsCRP at @ weeks in the intervention group when compared to the placebo group .\n',
  'RESULTS\tThese differences remained significant at @ weeks .\n',
  'RESULTS\tThe Outcome Measures in Rheumatology Clinical Trials-Osteoarthritis Research Society International responder rate was @ % in the intervention group and @ % in the placebo group ( p < @ ) .\n',
  'CONCLUSIONS\tLow-dose oral prednisolone had both a short-term and a longer sustained effect resulting in less knee pain , better physical function , and attenuation of systemic inflammation in older patients with knee OA ( ClinicalTrials.gov identifier NCT@ ) .\n',
  '\n']
```

## Problem in a sentence

The number of RCT papers released is continuing to increase, those without structured abstracts can be hard to read and in turn slow down researchers moving through the literature.

## Solution in a sentence

Create an NLP model to classify abstract sentences into the role they play (e.g. objective, methods, results, etc)

to enable researchers to skim through the literature (hence SkimLit ☰) and dive deeper when necessary.

☐ **Resources:** Before going through the code in this notebook, you might want to get a background of what we're going to be doing. To do so, spend an hour (or two) going through the following papers and then return to this notebook:

1. Where our data is coming from: [\*PubMed 200k RCT: a Dataset for Sequential Sentence Classification in Medical Abstracts\*](#)
2. Where our model is coming from: [\*Neural networks for joint sentence classification in medical paper abstracts\*](#).

## What we're going to cover

Time to take what we've learned in the NLP fundamentals notebook and build our biggest NLP model yet:

- Downloading a text dataset ([PubMed RCT200k from GitHub](#))
- Writing a preprocessing function to prepare our data for modelling
- Setting up a series of modelling experiments
  - Making a baseline (TF-IDF classifier)
  - Deep models with different combinations of: token embeddings, character embeddings, pretrained embeddings, positional embeddings
- Building our first multimodal model (taking multiple types of data inputs)
  - Replicating the model architecture from <https://arxiv.org/pdf/1612.05251.pdf>
- Find the most wrong predictions
- Making predictions on PubMed abstracts from the wild

## How you should approach this notebook

You can read through the descriptions and the code (it should all run, except for the cells which error on purpose), but there's a better option.

Write all of the code yourself.

Yes. I'm serious. Create a new notebook, and rewrite each line by yourself. Investigate it, see if you can break it, why does it break?

You don't have to write the text descriptions but writing the code yourself is a great way to get hands-on experience.

Don't worry if you make mistakes, we all do. The way to get better and make less mistakes is to write more code.

☐ **Resource:** See the full set of course materials on GitHub:

<https://github.com/mrdbourke/tensorflow-deep-learning>

## Confirm access to a GPU

Since we're going to be building deep learning models, let's make sure we have a GPU.

In Google Colab, you can set this up by going to Runtime -> Change runtime type -> Hardware accelerator -> GPU.

If you don't have access to a GPU, the models we're building here will likely take up to 10x longer to run.

In [ ]:

```
# Check for GPU
!nvidia-smi -L
```

GPU 0: Tesla T4 (UUID: GPU-90b6bfd2-2dbc-6214-b3b0-835ecd7fd102)

# Get data

Before we can start building a model, we've got to download the PubMed 200k RCT dataset.

In a phenomenal act of kindness, the authors of the paper have made the data they used for their research available publically and for free in the form of .txt files [on GitHub](#).

We can copy them to our local directory using `git clone https://github.com/Franck-Dernoncourt/pubmed-rct`.

In [ ]:

```
!git clone https://github.com/Franck-Dernoncourt/pubmed-rct.git  
!ls pubmed-rct
```

```
Cloning into 'pubmed-rct'...  
remote: Enumerating objects: 33, done.  
remote: Counting objects: 100% (3/3), done.  
remote: Compressing objects: 100% (3/3), done.  
remote: Total 33 (delta 0), reused 0 (delta 0), pack-reused 30  
Unpacking objects: 100% (33/33), done.  
PubMed_200k_RCT  
PubMed_200k_RCT_numbers_replaced_with_at_sign  
PubMed_20k_RCT  
PubMed_20k_RCT_numbers_replaced_with_at_sign  
README.md
```

Checking the contents of the downloaded repository, you can see there are four folders.

Each contains a different version of the PubMed 200k RCT dataset.

Looking at the [README file](#) from the GitHub page, we get the following information:

- PubMed 20k is a subset of PubMed 200k. I.e., any abstract present in PubMed 20k is also present in PubMed 200k.
- PubMed\_200k\_RCT is the same as PubMed\_200k\_RCT\_numbers\_replaced\_with\_at\_sign, except that in the latter all numbers had been replaced by @. (same for PubMed\_20k\_RCT vs. PubMed\_20k\_RCT\_numbers\_replaced\_with\_at\_sign).
- Since Github file size limit is 100 MiB, we had to compress PubMed\_200k\_RCT\train.7z and PubMed\_200k\_RCT\_numbers\_replaced\_with\_at\_sign\train.zip. To uncompress train.7z, you may use 7-Zip on Windows, Keka on Mac OS X, or p7zip on Linux.

To begin with, the dataset we're going to be focused on is

```
PubMed_20k_RCT_numbers_replaced_with_at_sign.
```

Why this one?

Rather than working with the whole 200k dataset, we'll keep our experiments quick by starting with a smaller subset. We could've chosen the dataset with numbers instead of having them replaced with @ but we didn't.

Let's check the file contents.

In [ ]:

```
# Check what files are in the PubMed_20K dataset  
!ls pubmed-rct/PubMed_20k_RCT_numbers_replaced_with_at_sign  
  
dev.txt test.txt train.txt
```

Beautiful, looks like we've got three separate text files:

- train.txt - training samples.
- dev.txt - dev is short for development set, which is another name for validation set (in our case, we'll be using and referring to this file as our validation set).
- test.txt - test samples.

To save ourselves typing out the filepath to our target directory each time, let's turn it into a variable.

In [ ]:

```
# Start by using the 20k dataset
data_dir = "pubmed-rct/PubMed_20k_RCT_numbers_replaced_with_at_sign/"
```

In [ ]:

```
# Check all of the filenames in the target directory
import os
filenames = [data_dir + filename for filename in os.listdir(data_dir)]
filenames
```

Out[ ]:

```
['pubmed-rct/PubMed_20k_RCT_numbers_replaced_with_at_sign/train.txt',
 'pubmed-rct/PubMed_20k_RCT_numbers_replaced_with_at_sign/test.txt',
 'pubmed-rct/PubMed_20k_RCT_numbers_replaced_with_at_sign/dev.txt']
```

## Preprocess data

Okay, now we've downloaded some text data, do you think we're ready to model it?

Wait...

We've downloaded the data but we haven't even looked at it yet.

What's the motto for getting familiar with any new dataset?

I'll give you a clue, the word begins with "v" and we say it three times.

Vibe, vibe, vibe?

Sort of... we've definitely got to the feel the vibe of our data.

Values, values, values?

Right again, we want to see lots of values but not quite what we're looking for.

Visualize, visualize, visualize?

Boom! That's it. To get familiar and understand how we have to prepare our data for our deep learning models, we've got to visualize it.

Because our data is in the form of text files, let's write some code to read each of the lines in a target file.

In [ ]:

```
# Create function to read the lines of a document
def get_lines(filename):
    """
    Reads filename (a text file) and returns the lines of text as a list.

    Args:
        filename: a string containing the target filepath to read.

    Returns:
        A list of strings with one string per line from the target filename.
        For example:
        ["this is the first line of filename",
         "this is the second line of filename",
         "..."]
    """
    with open(filename, "r") as f:
```

```
return f.readlines()
```

Alright, we've got a little function, `get_lines()` which takes the filepath of a text file, opens it, reads each of the lines and returns them.

Let's try it out on the training data (`train.txt`).

In [ ]:

```
train_lines = get_lines(data_dir+"train.txt")
train_lines[:20] # the whole first example of an abstract + a little more of the next one
```

Out[ ]:

```
[##24293578\n',
 'OBJECTIVE\tTo investigate the efficacy of @ weeks of daily low-dose oral prednisolone in improving pain , mobility , and systemic low-grade inflammation in the short term and whether the effect would be sustained at @ weeks in older adults with moderate to severe knee osteoarthritis ( OA ) .\n',
 'METHODS\tA total of @ patients with primary knee OA were randomized @:@ ; @ received @ mg/day of prednisolone and @ received placebo for @ weeks .\n',
 'METHODS\tOutcome measures included pain reduction and improvement in function scores and systemic inflammation markers .\n',
 'METHODS\tPain was assessed using the visual analog pain scale ( @-@ mm ) .\n',
 'METHODS\tSecondary outcome measures included the Western Ontario and McMaster Universities Osteoarthritis Index scores , patient global assessment ( PGA ) of the severity of knee OA , and @-min walk distance ( @MWD ) .\n',
 'METHODS\tSerum levels of interleukin @ ( IL-@ ) , IL-@ , tumor necrosis factor ( TNF ) - , and high-sensitivity C-reactive protein ( hsCRP ) were measured .\n',
 'RESULTS\tThere was a clinically relevant reduction in the intervention group compared to the placebo group for knee pain , physical function , PGA , and @MWD at @ weeks .\n',
 'RESULTS\tThe mean difference between treatment arms ( @ % CI ) was @ ( @-@ @ ) , p < @ ; @ ( @-@ @ ) , p < @ ; @ ( @-@ @ ) , p < @ , respectively .\n',
 'RESULTS\tFurther , there was a clinically relevant reduction in the serum levels of IL-@ , IL-@ , TNF - , and hsCRP at @ weeks in the intervention group when compared to the placebo group .\n',
 'RESULTS\tThese differences remained significant at @ weeks .\n',
 'RESULTS\tThe Outcome Measures in Rheumatology Clinical Trials-Osteoarthritis Research Society International responder rate was @ % in the intervention group and @ % in the placebo group ( p < @ ) .\n',
 'CONCLUSIONS\tLow-dose oral prednisolone had both a short-term and a longer sustained effect resulting in less knee pain , better physical function , and attenuation of systemic inflammation in older patients with knee OA ( ClinicalTrials.gov identifier NCT@ ) .\n',
 '\n',
 '##24854809\n',
 'BACKGROUND\tEmotional eating is associated with overeating and the development of obesity .\n',
 'BACKGROUND\tYet , empirical evidence for individual ( trait ) differences in emotional eating and cognitive mechanisms that contribute to eating during sad mood remain equivocal .\n',
 'OBJECTIVE\tThe aim of this study was to test if attention bias for food moderates the effect of self-reported emotional eating during sad mood ( vs neutral mood ) on actual food intake .\n',
 'OBJECTIVE\tIt was expected that emotional eating is predictive of elevated attention for food and higher food intake after an experimentally induced sad mood and that attentional maintenance on food predicts food intake during a sad versus a neutral mood .\n',
 'METHODS\tParticipants ( N = @ ) were randomly assigned to one of the two experimental mood induction conditions ( sad/neutral ) .\n']
```

Reading the lines from the training text file results in a list of strings containing different abstract samples, the sentences in a sample along with the role the sentence plays in the abstract.

The role of each sentence is prefixed at the start of each line separated by a tab (`\t`) and each sentence finishes with a new line (`\n`).

Different abstracts are separated by abstract ID's (lines beginning with `###`) and newlines (`\n`).

Knowing this, it looks like we've got a couple of steps to do to get our samples ready to pass as training data to our future machine learning model

**Let's write a function to perform the following steps:**

- **Take a target file of abstract samples.**
- **Read the lines in the target file.**
- **For each line in the target file:**
  - If the line begins with `###` mark it as an abstract ID and the beginning of a new abstract.
    - Keep count of the number of lines in a sample.
  - If the line begins with `\n` mark it as the end of an abstract sample.
    - Keep count of the total lines in a sample.
  - Record the text before the `\t` as the label of the line.
  - Record the text after the `\t` as the text of the line.
- **Return all of the lines in the target text file as a list of dictionaries containing the key/value pairs:**
  - `"line_number"` - the position of the line in the abstract (e.g. `3`).
  - `"target"` - the role of the line in the abstract (e.g. `OBJECTIVE`).
  - `"text"` - the text of the line in the abstract.
  - `"total_lines"` - the total lines in an abstract sample (e.g. `14`).
- **Abstract ID's and newlines should be omitted from the returned preprocessed data.**

**Example returned preprocessed sample (a single line from an abstract):**

```
[{'line_number': 0,
 'target': 'OBJECTIVE',
 'text': 'to investigate the efficacy of @ weeks of daily low-dose oral prednisolo
ne in improving pain , mobility , and systemic low-grade inflammation in the short
term and whether the effect would be sustained at @ weeks in older adults with mode
rate to severe knee osteoarthritis ( oa ) .',
 'total_lines': 11},
...]
```

In [ ]:

```
def preprocess_text_with_line_numbers(filename):
    """Returns a list of dictionaries of abstract line data.

    Takes in filename, reads its contents and sorts through each line,
    extracting things like the target label, the text of the sentence,
    how many sentences are in the current abstract and what sentence number
    the target line is.

    Args:
        filename: a string of the target text file to read and extract line data
        from.

    Returns:
        A list of dictionaries each containing a line from an abstract,
        the lines label, the lines position in the abstract and the total number
        of lines in the abstract where the line is from. For example:

        [{"target": "CONCLUSION",
          "text": "The study couldn't have gone better, turns out people are kinder than you
think",
          "line_number": 8,
          "total_lines": 8}]
    """

    input_lines = get_lines(filename) # get all lines from filename
    abstract_lines = "" # create an empty abstract
    abstract_samples = [] # create an empty list of abstracts

    # Loop through each line in target file
    for line in input_lines:
        if line.startswith("###"): # check to see if line is an ID line
            abstract_id = line
            abstract_lines = "" # reset abstract string
        elif line.isspace(): # check to see if line is a new line
```

```

abstract_line_split = abstract_lines.splitlines() # split abstract into separate lines

# Iterate through each line in abstract and count them at the same time
for abstract_line_number, abstract_line in enumerate(abstract_line_split):
    line_data = {} # create empty dict to store data from line
    target_text_split = abstract_line.split("\t") # split target label from text
    line_data["target"] = target_text_split[0] # get target label
    line_data["text"] = target_text_split[1].lower() # get target text and lower it
    line_data["line_number"] = abstract_line_number # what number line does the line appear in the abstract?
    line_data["total_lines"] = len(abstract_line_split) - 1 # how many total lines are in the abstract? (start from 0)
    abstract_samples.append(line_data) # add line data to abstract samples list

else: # if the above conditions aren't fulfilled, the line contains a labelled sentence
    abstract_lines += line

return abstract_samples

```

**Beautiful! That's one good looking function. Let's use it to preprocess each of our RCT 20k datasets.**

In [ ]:

```

# Get data from file and preprocess it
%%time
train_samples = preprocess_text_with_line_numbers(data_dir + "train.txt")
val_samples = preprocess_text_with_line_numbers(data_dir + "dev.txt") # dev is another name for validation set
test_samples = preprocess_text_with_line_numbers(data_dir + "test.txt")
len(train_samples), len(val_samples), len(test_samples)

```

CPU times: user 450 ms, sys: 89.4 ms, total: 540 ms  
Wall time: 540 ms

**How do our training samples look?**

In [ ]:

```
# Check the first abstract of our training data
train_samples[:14]
```

Out[ ]:

```
[{'line_number': 0,
 'target': 'OBJECTIVE',
 'text': 'to investigate the efficacy of @ weeks of daily low-dose oral prednisolone in improving pain , mobility , and systemic low-grade inflammation in the short term and whether the effect would be sustained at @ weeks in older adults with moderate to severe knee osteoarthritis ( oa ) .',
 'total_lines': 11},
 {'line_number': 1,
 'target': 'METHODS',
 'text': 'a total of @ patients with primary knee oa were randomized @:@ ; @ received @ mg/day of prednisolone and @ received placebo for @ weeks .',
 'total_lines': 11},
 {'line_number': 2,
 'target': 'METHODS',
 'text': 'outcome measures included pain reduction and improvement in function scores and systemic inflammation markers .',
 'total_lines': 11},
 {'line_number': 3,
 'target': 'METHODS',
 'text': 'pain was assessed using the visual analog pain scale ( @-@ mm ) .',
 'total_lines': 11},
 {'line_number': 4,
 'target': 'METHODS',
 'text': 'secondary outcome measures included the western ontario and mcmaster universities osteoarthritis index scores , patient global assessment ( pga ) of the severity of knee oa , and @-min walk distance ( @mwd ) .',
```

```

'total_lines': 11},
{'line_number': 5,
'target': 'METHODS',
'text': 'serum levels of interleukin @ ( il-@ ) , il-@ , tumor necrosis factor ( tnf ) - , and high-sensitivity c-reactive protein ( hscrp ) were measured .',
'total_lines': 11},
{'line_number': 6,
'target': 'RESULTS',
'text': 'there was a clinically relevant reduction in the intervention group compared to the placebo group for knee pain , physical function , pga , and @mwd at @ weeks .',
'total_lines': 11},
{'line_number': 7,
'target': 'RESULTS',
'text': 'the mean difference between treatment arms ( @ % ci ) was @ ( @-@ @ ) , p < @ ; @ ( @-@ @ ) , p < @ ; @ ( @-@ @ ) , p < @ ; and @ ( @-@ @ ) , p < @ , respectively .',
'total_lines': 11},
{'line_number': 8,
'target': 'RESULTS',
'text': 'further , there was a clinically relevant reduction in the serum levels of il-@ , il-@ , tnf - , and hscrp at @ weeks in the intervention group when compared to the placebo group .',
'total_lines': 11},
{'line_number': 9,
'target': 'RESULTS',
'text': 'these differences remained significant at @ weeks .',
'total_lines': 11},
{'line_number': 10,
'target': 'RESULTS',
'text': 'the outcome measures in rheumatology clinical trials-osteoarthritis research society international responder rate was @ % in the intervention group and @ % in the placebo group ( p < @ ) .',
'total_lines': 11},
{'line_number': 11,
'target': 'CONCLUSIONS',
'text': 'low-dose oral prednisolone had both a short-term and a longer sustained effect resulting in less knee pain , better physical function , and attenuation of systemic inflammation in older patients with knee oa ( clinicaltrials.gov identifier nct@ ) .',
'total_lines': 11},
{'line_number': 0,
'target': 'BACKGROUND',
'text': 'emotional eating is associated with overeating and the development of obesity .',
'total_lines': 10},
{'line_number': 1,
'target': 'BACKGROUND',
'text': 'yet , empirical evidence for individual ( trait ) differences in emotional eating and cognitive mechanisms that contribute to eating during sad mood remain equivocal .',
'total_lines': 10}]

```

**Fantastic! Looks like our `preprocess_text_with_line_numbers()` function worked great.**

**How about we turn our list of dictionaries into pandas DataFrame's so we visualize them better?**

In [ ]:

```

import pandas as pd
train_df = pd.DataFrame(train_samples)
val_df = pd.DataFrame(val_samples)
test_df = pd.DataFrame(test_samples)
train_df.head(14)

```

Out[ ]:

	target	text	line_number	total_lines
0	OBJECTIVE	to investigate the efficacy of @ weeks of dail...	0	11
1	METHODS	a total of @ patients with primary knee oa wer...	1	11
2	METHODS	outcome measures included pain reduction and i...	2	11

		pain was assessed using the visual analog pain...	text	line_number	3	11
4	METHODS	secondary outcome measures included the wester...		4	11	
5	METHODS	serum levels of interleukin @ ( il-@ ) , il-@ ...		5	11	
6	RESULTS	there was a clinically relevant reduction in t...		6	11	
7	RESULTS	the mean difference between treatment arms ( @...		7	11	
8	RESULTS	further , there was a clinically relevant redu...		8	11	
9	RESULTS	these differences remained significant at @ we...		9	11	
10	RESULTS	the outcome measures in rheumatology clinical ...		10	11	
11	CONCLUSIONS	low-dose oral prednisolone had both a short-te...		11	11	
12	BACKGROUND	emotional eating is associated with overeating...		0	10	
13	BACKGROUND	yet , empirical evidence for individual ( trai...		1	10	

Now our data is in DataFrame form, we can perform some data analysis on it.

In [ ]:

```
# Distribution of labels in training data
train_df.target.value_counts()
```

Out[ ]:

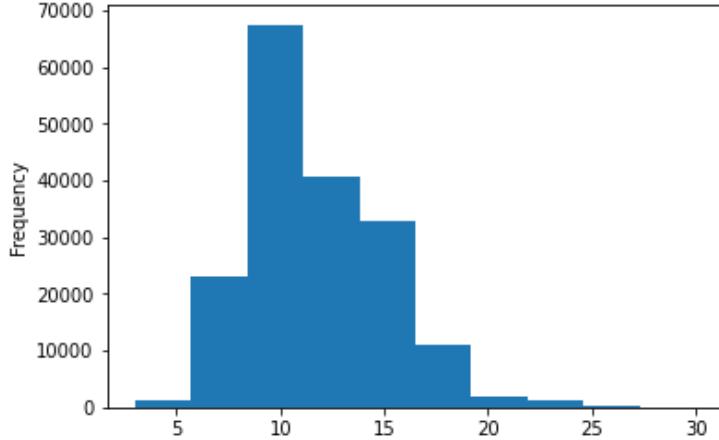
```
METHODS      59353
RESULTS      57953
CONCLUSIONS  27168
BACKGROUND    21727
OBJECTIVE    13839
Name: target, dtype: int64
```

Looks like sentences with the OBJECTIVE label are the least common.

How about we check the distribution of our abstract lengths?

In [ ]:

```
train_df.total_lines.plot.hist();
```



Okay, looks like most of the abstracts are around 7 to 15 sentences in length.

It's good to check these things out to make sure when we do train a model or test it on unseen samples, our results aren't outlandish.

## Get lists of sentences

When we build our deep learning model, one of its main inputs will be a list of strings (the lines of an abstract).

We can get these easily from our DataFrames by calling the tolist() method on our "text" columns.

In [ ]:

```
# Convert abstract text lines into lists
train_sentences = train_df["text"].tolist()
val_sentences = val_df["text"].tolist()
test_sentences = test_df["text"].tolist()
len(train_sentences), len(val_sentences), len(test_sentences)
```

Out[ ]:

```
(180040, 30212, 30135)
```

In [ ]:

```
# View first 10 lines of training sentences
train_sentences[:10]
```

Out[ ]:

```
['to investigate the efficacy of @ weeks of daily low-dose oral prednisolone in improving pain , mobility , and systemic low-grade inflammation in the short term and whether the effect would be sustained at @ weeks in older adults with moderate to severe knee osteoarthritis ( oa ) .',
 'a total of @ patients with primary knee oa were randomized @:@ ; @ received @ mg/day of prednisolone and @ received placebo for @ weeks .',
 'outcome measures included pain reduction and improvement in function scores and systemic inflammation markers .',
 'pain was assessed using the visual analog pain scale ( @-@ mm ) .',
 'secondary outcome measures included the western ontario and mcmaster universities osteoarthritis index scores , patient global assessment ( pga ) of the severity of knee oa , and @-min walk distance ( @mwd ) .',
 'serum levels of interleukin @ ( il-@ ) , il-@ , tumor necrosis factor ( tnf ) - , and high-sensitivity c-reactive protein ( hscrp ) were measured .',
 'there was a clinically relevant reduction in the intervention group compared to the placebo group for knee pain , physical function , pga , and @mwd at @ weeks .',
 'the mean difference between treatment arms ( @ % ci ) was @ ( @-@ @ ) , p < @ ; @ ( @-@ @ ) , p < @ ; @ ( @-@ @ ) , p < @ , respectively .',
 'further , there was a clinically relevant reduction in the serum levels of il-@ , il-@ , tnf - , and hscrp at @ weeks in the intervention group when compared to the placebo group .',
 'these differences remained significant at @ weeks .']
```

**Alright, we've separated our text samples. As you might've guessed, we'll have to write code to convert the text to numbers before we can use it with our machine learning models, we'll get to this soon.**

## Make numeric labels (ML models require numeric labels)

We're going to create one hot and label encoded labels.

We could get away with just making label encoded labels, however, TensorFlow's CategoricalCrossentropy loss function likes to have one hot encoded labels (this will enable us to use label smoothing later on).

To numerically encode labels we'll use Scikit-Learn's [OneHotEncoder](#) and [LabelEncoder](#) classes.

In [ ]:

```
# One hot encode labels
from sklearn.preprocessing import OneHotEncoder
one_hot_encoder = OneHotEncoder(sparse=False)
train_labels_one_hot = one_hot_encoder.fit_transform(train_df["target"].to_numpy().reshape(-1, 1))
val_labels_one_hot = one_hot_encoder.transform(val_df["target"].to_numpy().reshape(-1, 1))
test_labels_one_hot = one_hot_encoder.transform(test_df["target"].to_numpy().reshape(-1, 1))

# Check what training labels look like
train_labels_one_hot
```

```
Out[ ]:
```

```
array([[0., 0., 0., 1., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 1., 0., 0.],
       ...,
       [0., 0., 0., 0., 1.],
       [0., 1., 0., 0., 0.],
       [0., 1., 0., 0., 0.]])
```

## Label encode labels

```
In [ ]:
```

```
# Extract labels ("target" columns) and encode them into integers
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
train_labels_encoded = label_encoder.fit_transform(train_df["target"].to_numpy())
val_labels_encoded = label_encoder.transform(val_df["target"].to_numpy())
test_labels_encoded = label_encoder.transform(test_df["target"].to_numpy())

# Check what training labels look like
train_labels_encoded
```

```
Out[ ]:
```

```
array([3, 2, 2, ..., 4, 1, 1])
```

Now we've trained an instance of `LabelEncoder`, we can get the class names and number of classes using the `classes_` attribute.

```
In [ ]:
```

```
# Get class names and number of classes from LabelEncoder instance
num_classes = len(label_encoder.classes_)
class_names = label_encoder.classes_
num_classes, class_names
```

```
Out[ ]:
```

```
(5, array(['BACKGROUND', 'CONCLUSIONS', 'METHODS', 'OBJECTIVE', 'RESULTS'],
          dtype=object))
```

## Creating a series of model experiments

We've proprocessed our data so now, in true machine learning fashion, it's time to setup a series of modelling experiments.

We'll start by creating a simple baseline model to obtain a score we'll try to beat by building more and more complex models as we move towards replicating the sequence model outlined in [Neural networks for joint sentence classification in medical paper abstracts](#).

For each model, we'll train it on the training data and evaluate it on the validation data.

## Model 0: Getting a baseline

Our first model we'll be a TF-IDF Multinomial Naive Bayes as recommended by [Scikit-Learn's machine learning map](#).

To build it, we'll create a Scikit-Learn Pipeline which uses the `TfidfVectorizer` class to convert our abstract sentences to numbers using the TF-IDF (term frequency-inverse document frequency) algorithm and then learns to classify our sentences using the `MultinomialNB` aglorithm.

```
In [ ]:
```

```

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline

# Create a pipeline
model_0 = Pipeline([
    ("tf-idf", TfidfVectorizer()),
    ("clf", MultinomialNB())
])

# Fit the pipeline to the training data
model_0.fit(X=train_sentences,
             y=train_labels_encoded);

```

**Due to the speed of the Multinomial Naive Bayes algorithm, it trains very quickly.**

We can evaluate our model's accuracy on the validation dataset using the `score()` method.

In [ ]:

```

# Evaluate baseline on validation dataset
model_0.score(X=val_sentences,
               y=val_labels_encoded)

```

Out[ ]:

0.7218323844829869

Nice! Looks like 72.1% accuracy will be the number to beat with our deeper models.

Now let's make some predictions with our baseline model to further evaluate it.

In [ ]:

```

# Make predictions
baseline_preds = model_0.predict(val_sentences)
baseline_preds

```

Out[ ]:

array([4, 1, 3, ..., 4, 4, 1])

To evaluate our baseline's predictions, we'll import the `calculate_results()` function we created in the [previous notebook](#) and added it to our [helper\\_functions.py script](#) to compare them to the ground truth labels.

More specifically the `calculate_results()` function will help us obtain the following:

- Accuracy
- Precision
- Recall
- F1-score

## Download helper functions script

Let's get our `helper_functions.py` script we've been using to store helper functions we've created in previous notebooks.

In [ ]:

```

# Download helper functions script
!wget https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/extras/helper_functions.py

```

```
--2021-08-24 23:56:53-- https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/extras/helper_functions.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.110.133, 185.1
```

```
99.111.133, 185.199.108.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.110.133|:443.
.. connected.
HTTP request sent, awaiting response... 200 OK
Length: 10246 (10K) [text/plain]
Saving to: 'helper_functions.py'

helper_functions.py 100%[=====] 10.01K --.-KB/s in 0s

2021-08-24 23:56:53 (89.8 MB/s) - 'helper_functions.py' saved [10246/10246]
```

**Now we've got the helper functions script we can import the `calculate_results()` function and see how our baseline model went.**

In [ ]:

```
# Import calculate_results helper function
from helper_functions import calculate_results
```

In [ ]:

```
# Calculate baseline results
baseline_results = calculate_results(y_true=val_labels_encoded,
                                      y_pred=baseline_preds)
baseline_results
```

Out [ ]:

```
{'accuracy': 72.1832384482987,
 'f1': 0.6989250353450294,
 'precision': 0.7186466952323352,
 'recall': 0.7218323844829869}
```

## Preparing our data for deep sequence models

Excellent! We've got a working baseline to try and improve upon.

But before we start building deeper models, we've got to create vectorization and embedding layers.

The vectorization layer will convert our text to numbers and the embedding layer will capture the relationships between those numbers.

To start creating our vectorization and embedding layers, we'll need to import the appropriate libraries (namely TensorFlow and NumPy).

In [ ]:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
```

Since we'll be turning our sentences into numbers, it's a good idea to figure out how many words are in each sentence.

When our model goes through our sentences, it works best when they're all the same length (this is important for creating batches of the same size tensors).

For example, if one sentence is eight words long and another is 29 words long, we want to pad the eight word sentence with zeros so it ends up being the same length as the 29 word sentence.

Let's write some code to find the average length of sentences in the training set.

In [ ]:

```
# How long is each sentence on average?
sent_lens = [len(sentence.split()) for sentence in train_sentences]
```

```
avg_sent_len = np.mean(sent_lens)
avg_sent_len # return average sentence length (in tokens)
```

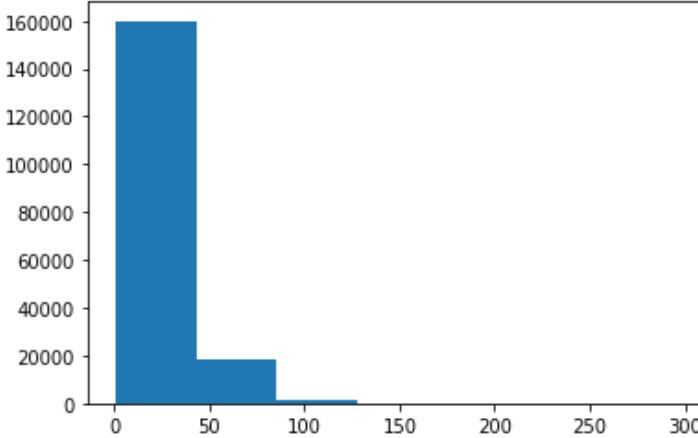
Out[ ]:

26.338269273494777

## How about the distribution of sentence lengths?

In [ ]:

```
# What's the distribution look like?
import matplotlib.pyplot as plt
plt.hist(sent_lens, bins=7);
```



Looks like the vast majority of sentences are between 0 and 50 tokens in length.

We can use NumPy's [percentile](#) to find the value which covers 95% of the sentence lengths.

In [ ]:

```
# How long of a sentence covers 95% of the lengths?
output_seq_len = int(np.percentile(sent_lens, 95))
output_seq_len
```

Out[ ]:

55

Wonderful! It looks like 95% of the sentences in our training set have a length of 55 tokens or less.

When we create our tokenization layer, we'll use this value to turn all of our sentences into the same length. Meaning sentences with a length below 55 get padded with zeros and sentences with a length above 55 get truncated (words after 55 get cut off).

### Question: Why 95%?

We could use the max sentence length of the sentences in the training set.

In [ ]:

```
# Maximum sentence length in the training set
max(sent_lens)
```

Out[ ]:

296

However, since hardly any sentences even come close to the max length, it would mean the majority of the data we pass to our model would be zeros (since all sentences below the max length would get padded with zeros).

### Note: The steps we've gone through are good practice when working with a text corpus for a

**NLP problem.** You want to know how long your samples are and what the distribution of them is. See section 4 Data Analysis of the [PubMed 200k RCT paper](#) for further examples.

## Create text vectorizer

Now we've got a little more information about our texts, let's create a way to turn it into numbers.

To do so, we'll use the [TextVectorization](#) layer from TensorFlow.

We'll keep all the parameters default except for `max_tokens` (the number of unique words in our dataset) and `output_sequence_length` (our desired output length for each vectorized sentence).

Section 3.2 of the [PubMed 200k RCT paper](#) states the vocabulary size of the PubMed 20k dataset as 68,000. So we'll use that as our `max_tokens` parameter.

In [ ]:

```
# How many words are in our vocabulary? (taken from 3.2 in https://arxiv.org/pdf/1710.06071.pdf)
max_tokens = 68000
```

And since discovered a sentence length of 55 covers 95% of the training sentences, we'll use that as our `output_sequence_length` parameter.

In [ ]:

```
# Create text vectorizer
from tensorflow.keras.layers.experimental.preprocessing import TextVectorization

text_vectorizer = TextVectorization(max_tokens=max_tokens, # number of words in vocabulary
                                     output_sequence_length=55) # desired output length of vectorized sequences
```

Great! Looks like our `text_vectorizer` is ready, let's adapt it to the training data (let it read the training data and figure out what number should represent what word) and then test it out.

In [ ]:

```
# Adapt text vectorizer to training sentences
text_vectorizer.adapt(train_sentences)
```

In [ ]:

```
# Test out text vectorizer
import random
target_sentence = random.choice(train_sentences)
print(f"Text:\n{target_sentence}")
print(f"\nLength of text: {len(target_sentence.split())}")
print(f"\nVectorized text:\n{text_vectorizer([target_sentence])}")
```

Text:

http://www.clinicaltrials.gov .

Length of text: 2

Vectorized text:

```
[[2243 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]]
```

Cool, we've now got a way to turn our sequences into numbers.

**Exercise:** Try running the cell above a dozen or so times. What do you notice about sequences

with a length less than 55?

Using the `get_vocabulary()` method of our `text_vectorizer` we can find out a few different tidbits about our text.

In [ ]:

```
# How many words in our training vocabulary?  
rct_20k_text_vocab = text_vectorizer.get_vocabulary()  
print(f"Number of words in vocabulary: {len(rct_20k_text_vocab)}")  
print(f"Most common words in the vocabulary: {rct_20k_text_vocab[:5]}")  
print(f"Least common words in the vocabulary: {rct_20k_text_vocab[-5:]}")
```

Number of words in vocabulary: 64841  
Most common words in the vocabulary: ['', '[UNK]', 'the', 'and', 'of']  
Least common words in the vocabulary: ['aainduced', 'aaigroup', 'aachener', 'aachen', 'aacp']

And if we wanted to figure out the configuration of our `text_vectorizer` we can use the `get_config()` method.

In [ ]:

```
# Get the config of our text vectorizer  
text_vectorizer.get_config()
```

Out[ ]:

```
{'batch_input_shape': (None,),  
'dtype': 'string',  
'max_tokens': 68000,  
'name': 'text_vectorization',  
'ngrams': None,  
'output_mode': 'int',  
'output_sequence_length': 55,  
'pad_to_max_tokens': False,  
'split': 'whitespace',  
'standardize': 'lower_and_strip_punctuation',  
'trainable': True}
```

## Create custom text embedding

Our `token_vectorization` layer maps the words in our text directly to numbers. However, this doesn't necessarily capture the relationships between those numbers.

To create a richer numerical representation of our text, we can use an **embedding**.

As our model learns (by going through many different examples of abstract sentences and their labels), it'll update its embedding to better represent the relationships between tokens in our corpus.

We can create a trainable embedding layer using TensorFlow's `Embedding` layer.

Once again, the main parameters we're concerned with here are the inputs and outputs of our `Embedding` layer.

The `input_dim` parameter defines the size of our vocabulary. And the `output_dim` parameter defines the dimension of the embedding output.

Once created, our embedding layer will take the integer outputs of our `text_vectorization` layer as inputs and convert them to feature vectors of size `output_dim`.

Let's see it in action.

In [ ]:

```
# Create token embedding layer
```

```

token_embed = layers.Embedding(input_dim=len(rct_20k_text_vocab), # length of vocabulary
                                output_dim=128, # Note: different embedding sizes result
                                in drastically different numbers of parameters to train
                                # Use masking to handle variable sequence lengths (save space)
                                mask_zero=True,
                                name="token_embedding")

# Show example embedding
print(f"Sentence before vectorization:\n{target_sentence}\n")
vectorized_sentence = text_vectorizer([target_sentence])
print(f"Sentence after vectorization (before embedding):\n{vectorized_sentence}\n")
embedded_sentence = token_embed(vectorized_sentence)
print(f"Sentence after embedding:\n{embedded_sentence}\n")
print(f"Embedded sentence shape: {embedded_sentence.shape}")

```

Sentence before vectorization:  
<http://www.clinicaltrials.gov> .

Sentence after vectorization (before embedding):

```

[[2243    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]
]
```

Sentence after embedding:

```

[[[ 0.0105269  0.04210378  0.04194726 ...  0.0111946  0.02379807
  -0.0433928 ]
 [-0.02391002  0.04098249 -0.0333933 ...  0.01311035 -0.02968328
  -0.04724472]
 [-0.02391002  0.04098249 -0.0333933 ...  0.01311035 -0.02968328
  -0.04724472]
 ...
 [-0.02391002  0.04098249 -0.0333933 ...  0.01311035 -0.02968328
  -0.04724472]
 [-0.02391002  0.04098249 -0.0333933 ...  0.01311035 -0.02968328
  -0.04724472]
 [-0.02391002  0.04098249 -0.0333933 ...  0.01311035 -0.02968328
  -0.04724472]]]

```

Embedded sentence shape: (1, 55, 128)

## Create datasets (as fast as possible)

We've gone through all the trouble of preprocessing our datasets to be used with a machine learning model, however, there are still a few steps we can use to make them work faster with our models.

Namely, the `tf.data` API provides methods which enable faster data loading.

 **Resource:** For best practices on data loading in TensorFlow, check out the following:

- [tf.data: Build TensorFlow input pipelines](#)
- [Better performance with the tf.data API](#)

The main steps we'll want to use with our data is to turn it into a `PrefetchDataset` of batches.

Doing so we'll ensure TensorFlow loads our data onto the GPU as fast as possible, in turn leading to faster training time.

To create a batched `PrefetchDataset` we can use the methods `batch()` and `prefetch()`, the parameter `tf.data.AUTOTUNE` will also allow TensorFlow to determine the optimal amount of compute to use to prepare datasets.

In [ ]:

```

# Turn our data into TensorFlow Datasets
train_dataset = tf.data.Dataset.from_tensor_slices((train_sentences, train_labels_one_hot))

```

```
)  
valid_dataset = tf.data.Dataset.from_tensor_slices((val_sentences, val_labels_one_hot))  
test_dataset = tf.data.Dataset.from_tensor_slices((test_sentences, test_labels_one_hot))  
  
train_dataset
```

Out [ ]:

```
<TensorSliceDataset shapes: (((), (5,)), types: (tf.string, tf.float64)>
```

In [ ]:

```
# Take the TensorSliceDataset's and turn them into prefetched batches  
train_dataset = train_dataset.batch(32).prefetch(tf.data.AUTOTUNE)  
valid_dataset = valid_dataset.batch(32).prefetch(tf.data.AUTOTUNE)  
test_dataset = test_dataset.batch(32).prefetch(tf.data.AUTOTUNE)  
  
train_dataset
```

Out [ ]:

```
<PrefetchDataset shapes: ((None,), (None, 5)), types: (tf.string, tf.float64)>
```

## Model 1: Conv1D with token embeddings

Alright, we've now got a way to numerically represent our text and labels, time to build a series of deep models to try and improve upon our baseline.

All of our deep models will follow a similar structure:

```
Input (text) -> Tokenize -> Embedding -> Layers -> Output (label probability)
```

The main component we'll be changing throughout is the `Layers` component. Because any modern deep NLP model requires text to be converted into an embedding before meaningful patterns can be discovered within.

The first model we're going to build is a 1-dimensional Convolutional Neural Network.

We're also going to be following the standard machine learning workflow of:

- Build model
- Train model
- Evaluate model (make predictions and compare to ground truth)

In [ ]:

```
# Create 1D convolutional model to process sequences  
inputs = layers.Input(shape=(1,), dtype=tf.string)  
text_vectors = text_vectorizer(inputs) # vectorize text inputs  
token_embeddings = token_embed(text_vectors) # create embedding  
x = layers.Conv1D(64, kernel_size=5, padding="same", activation="relu")(token_embeddings)  
x = layers.GlobalAveragePooling1D()(x) # condense the output of our feature vector  
outputs = layers.Dense(num_classes, activation="softmax")(x)  
model_1 = tf.keras.Model(inputs, outputs)  
  
# Compile  
model_1.compile(loss="categorical_crossentropy", # if your labels are integer form (not one hot) use sparse_categorical_crossentropy  
                 optimizer=tf.keras.optimizers.Adam(),  
                 metrics=["accuracy"])
```

In [ ]:

```
# Get summary of Conv1D model  
model_1.summary()
```

Model: "model"

Layer (type)	Output Shape	Param #

input_1 (InputLayer)	[None, 1]	0
text_vectorization (TextVect)	(None, 55)	0
token_embedding (Embedding)	(None, 55, 128)	8299648
conv1d (Conv1D)	(None, 55, 64)	41024
global_average_pooling1d (G1)	(None, 64)	0
dense (Dense)	(None, 5)	325
<hr/>		
Total params:	8,340,997	
Trainable params:	8,340,997	
Non-trainable params:	0	

**Wonderful! We've got our first deep sequence model built and ready to go.**

**Checking out the model summary, you'll notice the majority of the trainable parameters are within the embedding layer. If we were to increase the size of the embedding (by increasing the `output_dim` parameter of the `Embedding` layer), the number of trainable parameters would increase dramatically.**

**It's time to fit our model to the training data but we're going to make a mindful change.**

**Since our training data contains nearly 200,000 sentences, fitting a deep model may take a while even with a GPU. So to keep our experiments swift, we're going to run them on a subset of the training dataset.**

**More specifically, we'll only use the first 10% of batches (about 18,000 samples) of the training set to train on and the first 10% of batches from the validation set to validate on.**

**Note: It's a standard practice in machine learning to test your models on smaller subsets of data first to make sure they work before scaling them to larger amounts of data. You should aim to run many smaller experiments rather than only a handful of large experiments. And since your time is limited, one of the best ways to run smaller experiments is to reduce the amount of data you're working with (10% of the full dataset is usually a good amount, as long as it covers a similar distribution).**

In [ ]:

```
# Fit the model
model_1_history = model_1.fit(train_dataset,
                               steps_per_epoch=int(0.1 * len(train_dataset)), # only fit
on 10% of batches for faster training time
                               epochs=3,
                               validation_data=valid_dataset,
                               validation_steps=int(0.1 * len(valid_dataset))) # only val
idate on 10% of batches
```

```
Epoch 1/3
562/562 [=====] - 36s 8ms/step - loss: 0.9290 - accuracy: 0.6280
- val_loss: 0.6956 - val_accuracy: 0.7350
Epoch 2/3
562/562 [=====] - 5s 8ms/step - loss: 0.6647 - accuracy: 0.7528
- val_loss: 0.6353 - val_accuracy: 0.7666
Epoch 3/3
562/562 [=====] - 5s 8ms/step - loss: 0.6231 - accuracy: 0.7742
- val_loss: 0.6017 - val_accuracy: 0.7839
```

**Brilliant! We've got our first trained deep sequence model, and it didn't take too long (and if we didn't prefetch our batched data, it would've taken longer).**

**Time to make some predictions with our model and then evaluate them.**

In [ ]:

```
# Evaluate on whole validation dataset (we only validated on 10% of batches during training)
model_1.evaluate(valid_dataset)
```

```
945/945 [=====] - 3s 3ms/step - loss: 0.6040 - accuracy: 0.7845
```

Out [ ]:

```
[0.6039669513702393, 0.784489631652832]
```

In [ ]:

```
# Make predictions (our model outputs prediction probabilities for each class)
model_1_pred_probs = model_1.predict(valid_dataset)
model_1_pred_probs
```

Out [ ]:

```
array([[4.65810180e-01, 1.68030873e-01, 9.11973268e-02, 2.48495579e-01,
       2.64659580e-02],
       [3.85841161e-01, 3.26967925e-01, 1.09655075e-02, 2.69699097e-01,
       6.52625971e-03],
       [1.50063470e-01, 1.18270982e-02, 1.80714345e-03, 8.36278021e-01,
       2.42987626e-05],
       ...,
       [5.72754607e-06, 8.22585251e-04, 5.39675879e-04, 1.08408301e-06,
       9.98630941e-01],
       [5.39277196e-02, 4.82838809e-01, 9.59893093e-02, 5.87528460e-02,
       3.08491379e-01],
       [2.05477521e-01, 5.33861935e-01, 5.17199412e-02, 8.40113238e-02,
       1.24929301e-01]], dtype=float32)
```

In [ ]:

```
# Convert pred probs to classes
model_1_preds = tf.argmax(model_1_pred_probs, axis=1)
model_1_preds
```

Out [ ]:

```
<tf.Tensor: shape=(30212,), dtype=int64, numpy=array([0, 0, 3, ..., 4, 1, 1])>
```

In [ ]:

```
# Calculate model_1 results
model_1_results = calculate_results(y_true=val_labels_encoded,
                                      y_pred=model_1_preds)
model_1_results
```

Out [ ]:

```
{'accuracy': 78.44896067787634,
 'f1': 0.7822985872046467,
 'precision': 0.7814941086130403,
 'recall': 0.7844896067787634}
```

## Model 2: Feature extraction with pretrained token embeddings

Training our own embeddings took a little while to run, slowing our experiments down.

Since we're moving towards replicating the model architecture in [Neural Networks for Joint Sentence Classification in Medical Paper Abstracts](#), it mentions they used a [pretrained GloVe embedding](#) as a way to initialise their token embeddings.

To emulate this, let's see what results we can get with the [pretrained Universal Sentence Encoder embeddings from TensorFlow Hub](#).

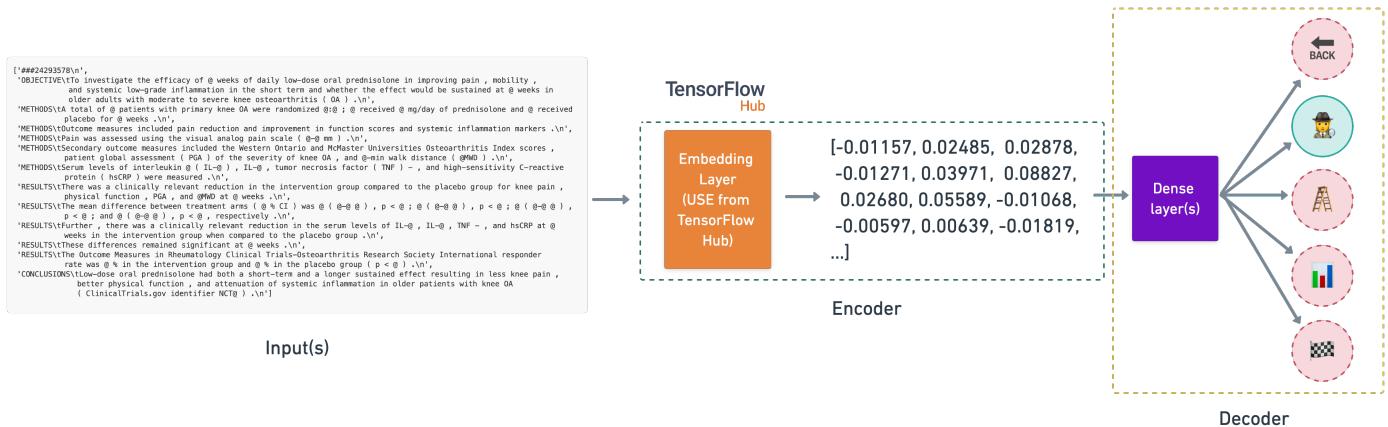
**Note:** We could use GloVe embeddings as per the paper but since we're working with TensorFlow, we'll use what's available from TensorFlow Hub (GloVe embeddings aren't). We'll save using [pretrained GloVe embeddings](#) as an extension.

## The model structure will look like:

Inputs (string) -> Pretrained embeddings from TensorFlow Hub (Universal Sentence Encoder) -> Layers -> Output (prediction probabilities)

You'll notice the lack of tokenization layer we've used in a previous model. This is because the Universal Sentence Encoder (USE) takes care of tokenization for us.

This type of model is called transfer learning, or more specifically, **feature extraction transfer learning**. In other words, taking the patterns a model has learned elsewhere and applying it to our own problem.



The feature extractor model we're building using a pretrained embedding from TensorFlow Hub.

To download the pretrained USE into a layer we can use in our model, we can use the `hub.KerasLayer` class.

We'll keep the pretrained embeddings frozen (by setting `trainable=False`) and add a trainable couple of layers on the top to tailor the model outputs to our own data.

**Note:** Due to having to download a relatively large model (~916MB), the cell below may take a little while to run.

In [ ]:

```
# Download pretrained TensorFlow Hub USE
import tensorflow_hub as hub
tf_hub_embedding_layer = hub.KerasLayer("https://tfhub.dev/google/universal-sentence-encoder/4",
                                         trainable=False,
                                         name="universal_sentence_encoder")
```

Beautiful, now our pretrained USE is downloaded and instantiated as a `hub.KerasLayer` instance, let's test it out on a random sentence.

In [ ]:

```
# Test out the embedding on a random sentence
random_training_sentence = random.choice(train_sentences)
print(f"Random training sentence:\n{random_training_sentence}\n")
use_embedded_sentence = tf_hub_embedding_layer([random_training_sentence])
print(f"Sentence after embedding:\n{use_embedded_sentence[0][:30]} (truncated output)...\n")
print(f"Length of sentence embedding:\n{len(use_embedded_sentence[0])}")
```

Random training sentence:  
data were collected from @, @ @th - and @th-grade students in these communities using anonymous cross-sectional surveys in @ and @ and analyzed in @ .

Sentence after embedding:

```
[-0.03330918  0.06432742 -0.04426299 -0.03340627  0.01580565  0.04260787
 0.05223562  0.02998829 -0.09009711  0.01004721 -0.01640431 -0.02970598
 0.06300306  0.0500071   -0.02137795  0.01532154  0.00676913  0.04174187
 0.00070422 -0.08601078  0.02839869  0.07599191 -0.05524107 -0.00862489]
```

```
-0.04704431 0.05998407 -0.00028414 -0.01645767 0.05340267 -0.06473802] (truncated output)...
```

Length of sentence embedding:  
512

**Nice! As we mentioned before the pretrained USE module from TensorFlow Hub takes care of tokenizing our text for us and outputs a 512 dimensional embedding vector.**

Let's put together and compile a model using our `tf_hub_embedding_layer`.

## Building and fitting an NLP feature extraction model from TensorFlow Hub

In [ ]:

```
# Define feature extractor model using TF Hub layer
inputs = layers.Input(shape=[], dtype=tf.string)
pretrained_embedding = tf_hub_embedding_layer(inputs) # tokenize text and create embedding
x = layers.Dense(128, activation="relu")(pretrained_embedding) # add a fully connected layer on top of the embedding
# Note: you could add more layers here if you wanted to
outputs = layers.Dense(5, activation="softmax")(x) # create the output layer
model_2 = tf.keras.Model(inputs=inputs,
                         outputs=outputs)

# Compile the model
model_2.compile(loss="categorical_crossentropy",
                 optimizer=tf.keras.optimizers.Adam(),
                 metrics=["accuracy"])
```

In [ ]:

```
# Get a summary of the model
model_2.summary()
```

Model: "model\_1"

Layer (type)	Output Shape	Param #
=====		
input_2 (InputLayer)	[(None, )]	0
universal_sentence_encoder (	(None, 512)	256797824
dense_1 (Dense)	(None, 128)	65664
dense_2 (Dense)	(None, 5)	645
=====		
Total params: 256,864,133		
Trainable params: 66,309		
Non-trainable params: 256,797,824		

**Checking the summary of our model we can see there's a large number of total parameters, however, the majority of these are non-trainable. This is because we set `training=False` when we instantiated our USE feature extractor layer.**

So when we train our model, only the top two output layers will be trained.

In [ ]:

```
# Fit feature extractor model for 3 epochs
model_2.fit(train_dataset,
            steps_per_epoch=int(0.1 * len(train_dataset)),
            epochs=3,
            validation_data=valid_dataset,
            validation_steps=int(0.1 * len(valid_dataset)))
```

```
Epoch 1/3
562/562 [=====] - 9s 12ms/step - loss: 0.9150 - accuracy: 0.6498
- val_loss: 0.7939 - val_accuracy: 0.6895
Epoch 2/3
562/562 [=====] - 7s 12ms/step - loss: 0.7681 - accuracy: 0.7022
- val_loss: 0.7523 - val_accuracy: 0.7058
Epoch 3/3
562/562 [=====] - 7s 12ms/step - loss: 0.7509 - accuracy: 0.7127
- val_loss: 0.7367 - val_accuracy: 0.7138
```

Out [ ]:

```
<keras.callbacks.History at 0x7f1f82200850>
```

In [ ]:

```
# Evaluate on whole validation dataset
model_2.evaluate(valid_dataset)
```

```
945/945 [=====] - 10s 10ms/step - loss: 0.7403 - accuracy: 0.7143
```

Out [ ]:

```
[0.7403141856193542, 0.7142525911331177]
```

**Since we aren't training our own custom embedding layer, training is much quicker.**

**Let's make some predictions and evaluate our feature extraction model.**

In [ ]:

```
# Make predictions with feature extraction model
model_2_pred_probs = model_2.predict(valid_dataset)
model_2_pred_probs
```

Out [ ]:

```
array([[4.3034002e-01, 3.5648319e-01, 2.3842952e-03, 2.0301045e-01,
       7.7820425e-03],
       [3.5814181e-01, 4.8381555e-01, 3.4082821e-03, 1.5132000e-01,
       3.3143654e-03],
       [2.2659931e-01, 1.5397042e-01, 2.1248475e-02, 5.6063354e-01,
       3.7548285e-02],
       ...,
       [1.7298853e-03, 5.6997794e-03, 5.2002735e-02, 8.3732005e-04,
       9.3973035e-01],
       [3.5631014e-03, 4.8854645e-02, 1.8445115e-01, 1.2047966e-03,
       7.6192635e-01],
       [1.7391451e-01, 3.0626863e-01, 4.5657125e-01, 5.7948320e-03,
       5.7450745e-02]], dtype=float32)
```

In [ ]:

```
# Convert the predictions with feature extraction model to classes
model_2_preds = tf.argmax(model_2_pred_probs, axis=1)
model_2_preds
```

Out [ ]:

```
<tf.Tensor: shape=(30212,), dtype=int64, numpy=array([0, 1, 3, ..., 4, 4, 2])>
```

In [ ]:

```
# Calculate results from TF Hub pretrained embeddings results on validation set
model_2_results = calculate_results(y_true=val_labels_encoded,
                                      y_pred=model_2_preds)
model_2_results
```

Out [ ]:

```
{'accuracy': 71.42526148550245,
 'f1': 0.7114550329161863,
 'precision': 0.7112206211011516}
```

```
Precision : 0.7142526148550244,
'recall': 0.7142526148550244}
```

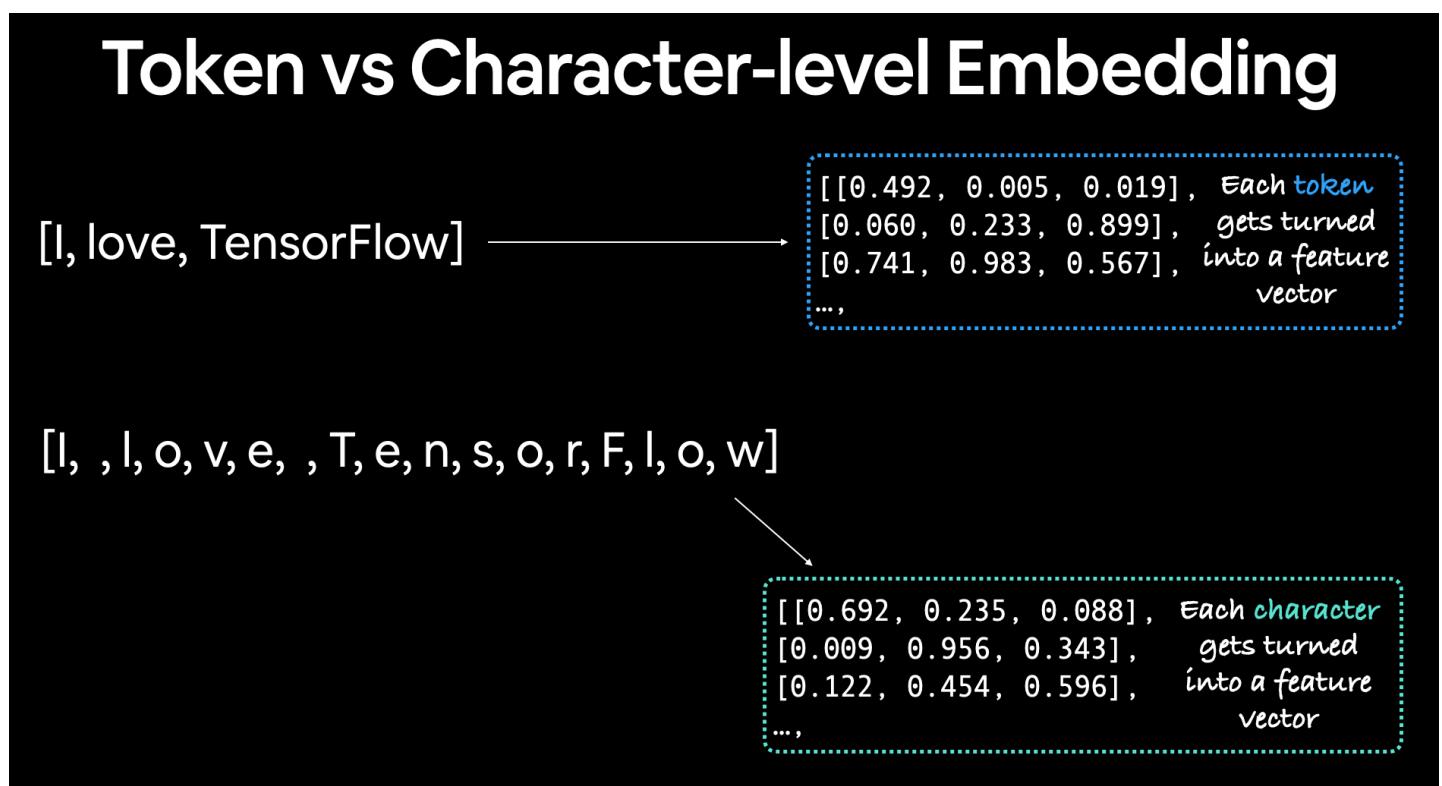
## Model 3: Conv1D with character embeddings

### Creating a character-level tokenizer

The [Neural Networks for Joint Sentence Classification in Medical Paper Abstracts](#) paper mentions their model uses a hybrid of token and character embeddings.

We've built models with a custom token embedding and a pretrained token embedding, how about we build one using a character embedding?

The difference between a character and token embedding is that the **character embedding** is created using sequences split into characters (e.g. `hello -> [ h , e , l , l , o ]`) where as a **token embedding** is created on sequences split into tokens.



*Token level embeddings split sequences into tokens (words) and embeddings each of them, character embeddings split sequences into characters and creates a feature vector for each.*

We can create a character-level embedding by first vectorizing our sequences (after they've been split into characters) using the [TextVectorization](#) class and then passing those vectorized sequences through an [Embedding](#) layer.

Before we can vectorize our sequences on a character-level we'll need to split them into characters. Let's write a function to do so.

In [ ]:

```
# Make function to split sentences into characters
def split_chars(text):
    return " ".join(list(text))

# Test splitting non-character-level sequence into characters
split_chars(random_training_sentence)
```

Out[ ]:

```
'data were collected from @, @ @th - and @th-
grade students in these communities using an
anonymous cross-sectional surveys in @ and @
and analyzed in @.'
```

**Great! Looks like our character-splitting function works. Let's create character-level datasets by splitting our sequence datasets into characters.**

In [ ]:

```
# Split sequence-level data splits into character-level data splits
train_chars = [split_chars(sentence) for sentence in train_sentences]
val_chars = [split_chars(sentence) for sentence in val_sentences]
test_chars = [split_chars(sentence) for sentence in test_sentences]
print(train_chars[0])
```

to investigate the efficacy of @ weeks of daily low-dose oral prednisolone in improving pain, mobility, and systemic low-grade inflammation in the short term and whether the effect would be sustained at @ weeks in older adults with moderate to severe knee osteoarthritis (oa).

**To figure out how long our vectorized character sequences should be, let's check the distribution of our character sequence lengths.**

In [ ]:

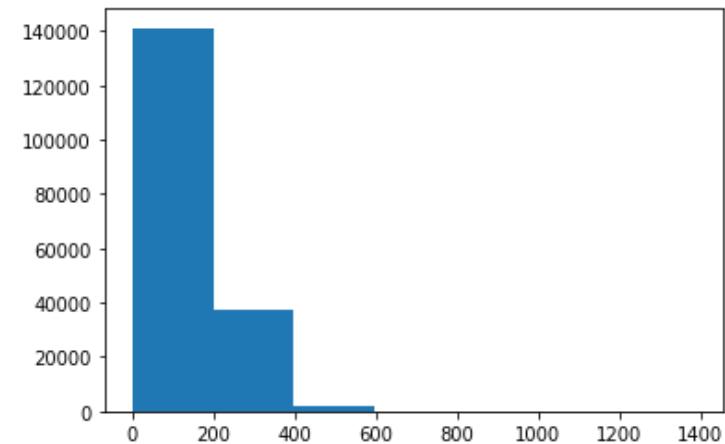
```
# What's the average character length?
char_lens = [len(sentence) for sentence in train_sentences]
mean_char_len = np.mean(char_lens)
mean_char_len
```

Out [ ]:

149.3662574983337

In [ ]:

```
# Check the distribution of our sequences at character-level
import matplotlib.pyplot as plt
plt.hist(char_lens, bins=7);
```



Okay, looks like most of our sequences are between 0 and 200 characters long.

**Let's use NumPy's percentile to figure out what length covers 95% of our sequences.**

In [ ]:

```
# Find what character length covers 95% of sequences
output_seq_char_len = int(np.percentile(char_lens, 95))
output_seq_char_len
```

Out [ ]:

290

Wonderful, now we know the sequence length which covers 95% of sequences. We'll use that in our

Wonderful, now we know the sequence length which covers 95% of sequences, we'll use that in our TextVectorization layer as the output\_sequence\_length parameter.

**Note:** You can experiment here to figure out what the optimal output\_sequence\_length should be, perhaps using the mean results in as good results as using the 95% percentile.

We'll set max\_tokens (the total number of different characters in our sequences) to 28, in other words, 26 letters of the alphabet + space + OOV (out of vocabulary or unknown) tokens.

In [ ]:

```
# Get all keyboard characters for char-level embedding
import string
alphabet = string.ascii_lowercase + string.digits + string.punctuation
alphabet
```

Out [ ]:

```
'abcdefghijklmnopqrstuvwxyz0123456789!#$%&\'()*+,-./:;<=>?@[\\\]^_`{|}~'
```

In [ ]:

```
# Create char-level token vectorizer instance
NUM_CHAR_TOKENS = len(alphabet) + 2 # num characters in alphabet + space + OOV token
char_vectorizer = TextVectorization(max_tokens=NUM_CHAR_TOKENS,
                                     output_sequence_length=output_seq_char_len,
                                     standardize="lower_and_strip_punctuation",
                                     name="char_vectorizer")

# Adapt character vectorizer to training characters
char_vectorizer.adapt(train_chars)
```

Nice! Now we've adapted our char\_vectorizer to our character-level sequences, let's check out some characteristics about it using the get\_vocabulary() method.

In [ ]:

```
# Check character vocabulary characteristics
char_vocab = char_vectorizer.get_vocabulary()
print(f"Number of different characters in character vocab: {len(char_vocab)}")
print(f"5 most common characters: {char_vocab[:5]}")
print(f"5 least common characters: {char_vocab[-5:]}")
```

```
Number of different characters in character vocab: 28
5 most common characters: ['', '[UNK]', 'e', 't', 'i']
5 least common characters: ['k', 'x', 'z', 'q', 'j']
```

We can also test it on random sequences of characters to make sure it's working.

In [ ]:

```
# Test out character vectorizer
random_train_chars = random.choice(train_chars)
print(f"\nCharified text:\n{random_train_chars}")
print(f"\nLength of chars: {len(random_train_chars.split())}")
vectorized_chars = char_vectorizer([random_train_chars])
print(f"\nVectorized chars:\n{vectorized_chars}")
print(f"\nLength of vectorized chars: {len(vectorized_chars[0])}")
```

```
Charified text:
a p e r s i s t e n t q u e s t i o n i s w h e t h e r p a r a p r o f e s s
i o n a l h o m e v i s i t o r s m i g h t p r o d u c e c o m p a r a b l e
e f f e c t s .
```

```
Length of chars: 86
```

```
Vectorized chars:
```

```
[[ 5 14 2 8 9 4 9 3 2 6 3 26 16 2 9 3 4 7 6 4 9 20 13 2
  3 13 2 8 14 5 8 5 14 8 7 17 2 9 9 4 7 6 5 12 13 7 15 2
```

Length of vectorized chars: 290

You'll notice sequences with a length shorter than 290 (`output_seq_char_length`) get padded with zeros on the end, this ensures all sequences passed to our model are the same length.

Also, due to the `standardize` parameter of `TextVectorization` being `"lower_and_strip_punctuation"` and the `split` parameter being `"whitespace"` by default, symbols (such as `@`) and spaces are removed.

**Note:** If you didn't want punctuation to be removed (keep the @, % etc), you can create a custom standardization callable and pass it as the `standardize` parameter. See the [TextVectorization](#) class documentation for more.

## Creating a character-level embedding

We've got a way to vectorize our character-level sequences, now's time to create a character-level embedding.

**Just like our custom token embedding, we can do so using the `tensorflow.keras.layers.Embedding` class.**

**Our character-level embedding layer requires an input dimension and output dimension.**

The input dimension (`input_dim`) will be equal to the number of different characters in our `char_vocab` (28). And since we're following the structure of the model in Figure 1 of [Neural Networks for Joint Sentence Classification in Medical Paper Abstracts](#), the output dimension of the character embedding (`output_dim`) will be 25.

In [ ] :

```

# Create char embedding layer
char_embed = layers.Embedding(input_dim=NUM_CHAR_TOKENS, # number of different characters
                               output_dim=25, # embedding dimension of each character (same as Figure 1 in https://arxiv.org/pdf/1612.05251.pdf)
                               mask_zero=False, # don't use masks (this messes up model_5 if set to True)
                               name="char_embed")

# Test out character embedding layer
print(f"Charified text (before vectorization and embedding):\n{random_train_chars}\n")
char_embed_example = char_embed(char_vectorizer([random_train_chars]))
print(f"Embedded chars (after vectorization and embedding):\n{char_embed_example}\n")
print(f"Character embedding shape: {char_embed_example.shape}")

```

Charified text (before vectorization and embedding):  
a persistent question is whether paraprofessional home visitors might produce comparable effects.

Embedded chars (after vectorization and embedding):

```
[[[-0.01923175 -0.01720572  0.04752548 ... -0.00952251 -0.03900822  
-0.01606651]  
[ 0.02746468 -0.0157405 -0.03408597 ...  0.00957409 -0.04426242  
0.02547267]  
[-0.00342412  0.04883511 -0.02929975 ... -0.04722989  0.04408958]
```

```

0.02328778]
...
[-0.02053725  0.04947415 -0.03963646 ...  0.04780323  0.00831659
-0.00033282]
[-0.02053725  0.04947415 -0.03963646 ...  0.04780323  0.00831659
-0.00033282]
[-0.02053725  0.04947415 -0.03963646 ...  0.04780323  0.00831659
-0.00033282]]

```

Character embedding shape: (1, 290, 25)

**Wonderful! Each of the characters in our sequences gets turned into a 25 dimension embedding.**

## Building a Conv1D model to fit on character embeddings

Now we've got a way to turn our character-level sequences into numbers (`char_vectorizer`) as well as numerically represent them as an embedding (`char_embed`) let's test how effective they are at encoding the information in our sequences by creating a character-level sequence model.

The model will have the same structure as our custom token embedding model (`model_1`) except it'll take character-level sequences as input instead of token-level sequences.

Input (character-level text) -> Tokenize -> Embedding -> Layers (Conv1D, GlobalMaxPool1D) -> Output (label probability)

In [ ]:

```

# Make Conv1D on chars only
inputs = layers.Input(shape=(1,), dtype="string")
char_vectors = char_vectorizer(inputs)
char_embeddings = char_embed(char_vectors)
x = layers.Conv1D(64, kernel_size=5, padding="same", activation="relu")(char_embeddings)
x = layers.GlobalMaxPool1D()(x)
outputs = layers.Dense(num_classes, activation="softmax")(x)
model_3 = tf.keras.Model(inputs=inputs,
                         outputs=outputs,
                         name="model_3_conv1D_char_embedding")

# Compile model
model_3.compile(loss="categorical_crossentropy",
                  optimizer=tf.keras.optimizers.Adam(),
                  metrics=["accuracy"])

```

In [ ]:

```
# Check the summary of conv1d_char_model
model_3.summary()
```

Model: "model\_3\_conv1D\_char\_embedding"

Layer (type)	Output Shape	Param #
<hr/>		
input_3 (InputLayer)	[(None, 1)]	0
char_vectorizer (TextVectori	(None, 290)	0
char_embed (Embedding)	(None, 290, 25)	1750
conv1d_1 (Conv1D)	(None, 290, 64)	8064
global_max_pooling1d (Global	(None, 64)	0
dense_3 (Dense)	(None, 5)	325
<hr/>		

Total params: 10,139  
Trainable params: 10,139  
Non-trainable params: 0

**Before fitting our model on the data, we'll create char-level batched PrefetchedDataset 's.**

In [ ]:

```
# Create char datasets
train_char_dataset = tf.data.Dataset.from_tensor_slices((train_chars, train_labels_one_hot)).batch(32).prefetch(tf.data.AUTOTUNE)
val_char_dataset = tf.data.Dataset.from_tensor_slices((val_chars, val_labels_one_hot)).batch(32).prefetch(tf.data.AUTOTUNE)

train_char_dataset
```

Out[ ]:

```
<PrefetchDataset shapes: ((None,), (None, 5)), types: (tf.string, tf.float64)>
```

**Just like our token-level sequence model, to save time with our experiments, we'll fit the character-level model on 10% of batches.**

In [ ]:

```
# Fit the model on chars only
model_3_history = model_3.fit(train_char_dataset,
                               steps_per_epoch=int(0.1 * len(train_char_dataset)),
                               epochs=3,
                               validation_data=val_char_dataset,
                               validation_steps=int(0.1 * len(val_char_dataset)))
```

```
Epoch 1/3
562/562 [=====] - 4s 5ms/step - loss: 1.2697 - accuracy: 0.4940
- val_loss: 1.0555 - val_accuracy: 0.5864
Epoch 2/3
562/562 [=====] - 3s 5ms/step - loss: 1.0124 - accuracy: 0.5996
- val_loss: 0.9542 - val_accuracy: 0.6267
Epoch 3/3
562/562 [=====] - 3s 5ms/step - loss: 0.9228 - accuracy: 0.6410
- val_loss: 0.8712 - val_accuracy: 0.6722
```

In [ ]:

```
# Evaluate model_3 on whole validation char dataset
model_3.evaluate(val_char_dataset)
```

```
945/945 [=====] - 3s 4ms/step - loss: 0.8873 - accuracy: 0.6588
```

Out[ ]:

```
[0.8873457908630371, 0.6587779521942139]
```

**Nice! Looks like our character-level model is working, let's make some predictions with it and evaluate them.**

In [ ]:

```
# Make predictions with character model only
model_3_pred_probs = model_3.predict(val_char_dataset)
model_3_pred_probs
```

Out[ ]:

```
array([[0.14757556, 0.40572175, 0.06463172, 0.33147973, 0.05059117],
       [0.33927342, 0.25103244, 0.01815611, 0.35482603, 0.03671199],
       [0.15584646, 0.11993653, 0.10841523, 0.5846738 , 0.03112798],
       ...,
       [0.02389161, 0.04591153, 0.07370146, 0.01870138, 0.83779407],
       [0.01393123, 0.10044182, 0.55801105, 0.05441127, 0.2732046 ],
       [0.28039306, 0.582124 , 0.04549843, 0.04313584, 0.04884864]],
      dtype=float32)
```

In [ ]:

```
# Convert predictions to classes
model_3_preds = tf.argmax(model_3_pred_probs, axis=1)
model_3_preds
```

Out[ ]:

```
<tf.Tensor: shape=(30212,), dtype=int64, numpy=array([1, 3, 3, ..., 4, 2, 1])>
```

In [ ]:

```
# Calculate ConvID char only model results
model_3_results = calculate_results(y_true=val_labels_encoded,
                                     y_pred=model_3_preds)
model_3_results
```

Out[ ]:

```
{'accuracy': 65.87779690189329,
 'f1': 0.6516863332082763,
 'precision': 0.6545009464773592,
 'recall': 0.6587779690189329}
```

## Model 4: Combining pretrained token embeddings + character embeddings (hybrid embedding layer)

Alright, now things are going to get spicy.

In moving closer to build a model similar to the one in Figure 1 of [Neural Networks for Joint Sentence Classification in Medical Paper Abstracts](#), it's time we tackled the hybrid token embedding layer they speak of.

This hybrid token embedding layer is a combination of token embeddings and character embeddings. In other words, they create a stacked embedding to represent sequences before passing them to the sequence label prediction layer.

So far we've built two models which have used token and character-level embeddings, however, these two models have used each of these embeddings exclusively.

To start replicating (or getting close to replicating) the model in Figure 1, we're going to go through the following steps:

1. Create a token-level model (similar to `model_1`)
2. Create a character-level model (similar to `model_3` with a slight modification to reflect the paper)
3. Combine (using `layers.Concatenate`) the outputs of 1 and 2
4. Build a series of output layers on top of 3 similar to Figure 1 and section 4.2 of [Neural Networks for Joint Sentence Classification in Medical Paper Abstracts](#)
5. Construct a model which takes token and character-level sequences as input and produces sequence label probabilities as output

In [ ]:

```
# 1. Setup token inputs/model
token_inputs = layers.Input(shape=[], dtype=tf.string, name="token_input")
token_embeddings = tf_hub_embedding_layer(token_inputs)
token_output = layers.Dense(128, activation="relu")(token_embeddings)
token_model = tf.keras.Model(inputs=token_inputs,
                             outputs=token_output)

# 2. Setup char inputs/model
char_inputs = layers.Input(shape=(1,), dtype=tf.string, name="char_input")
char_vectors = char_vectorizer(char_inputs)
char_embeddings = char_embed(char_vectors)
char_bi_lstm = layers.Bidirectional(layers.LSTM(25))(char_embeddings) # bi-LSTM shown in
# Figure 1 of https://arxiv.org/pdf/1612.05251.pdf
char_model = tf.keras.Model(inputs=char_inputs,
                           outputs=char_bi_lstm)

# 3. Concatenate token and char inputs (create hybrid token embedding)
```

```

token_char_concat = layers.concatenate(name="token_char_hybrid")([token_model.output,
                                                               char_model.output])

# 4. Create output layers - addition of dropout discussed in 4.2 of https://arxiv.org/pdf/1612.05251.pdf
combined_dropout = layers.Dropout(0.5)(token_char_concat)
combined_dense = layers.Dense(200, activation="relu")(combined_dropout) # slightly different to Figure 1 due to different shapes of token/char embedding layers
final_dropout = layers.Dropout(0.5)(combined_dense)
output_layer = layers.Dense(num_classes, activation="softmax")(final_dropout)

# 5. Construct model with char and token inputs
model_4 = tf.keras.Model(inputs=[token_model.input, char_model.input],
                         outputs=output_layer,
                         name="model_4_token_and_char_embeddings")

```

**Woah... There's a lot going on here, let's get a summary and plot our model to visualize what's happening.**

In [ ]:

```
# Get summary of token and character model
model_4.summary()
```

Model: "model\_4\_token\_and\_char\_embeddings"

Layer (type)	Output Shape	Param #	Connected to
<hr/>			
<hr/>			
char_input (InputLayer)	[ (None, 1) ]	0	
<hr/>			
token_input (InputLayer)	[ (None,) ]	0	
<hr/>			
char_vectorizer (TextVectorizat	(None, 290)	0	char_input[0][0]
<hr/>			
universal_sentence_encoder (Ker	(None, 512)	256797824	token_input[0][0]
<hr/>			
char_embed (Embedding)	(None, 290, 25)	1750	char_vectorizer[1][0]
<hr/>			
dense_4 (Dense)	(None, 128)	65664	universal_sentence_enco
der[1][0]			
<hr/>			
bidirectional (Bidirectional)	(None, 50)	10200	char_embed[1][0]
<hr/>			
token_char_hybrid (Concatenate)	(None, 178)	0	dense_4[0][0]
			bidirectional[0][0]
<hr/>			
dropout (Dropout)	(None, 178)	0	token_char_hybrid[0][0]
<hr/>			
dense_5 (Dense)	(None, 200)	35800	dropout[0][0]
<hr/>			

```
dropout_1 (Dropout)           (None, 200)          0          dense_5[0][0]
```

```
dense_6 (Dense)             (None, 5)            1005        dropout_1[0][0]
```

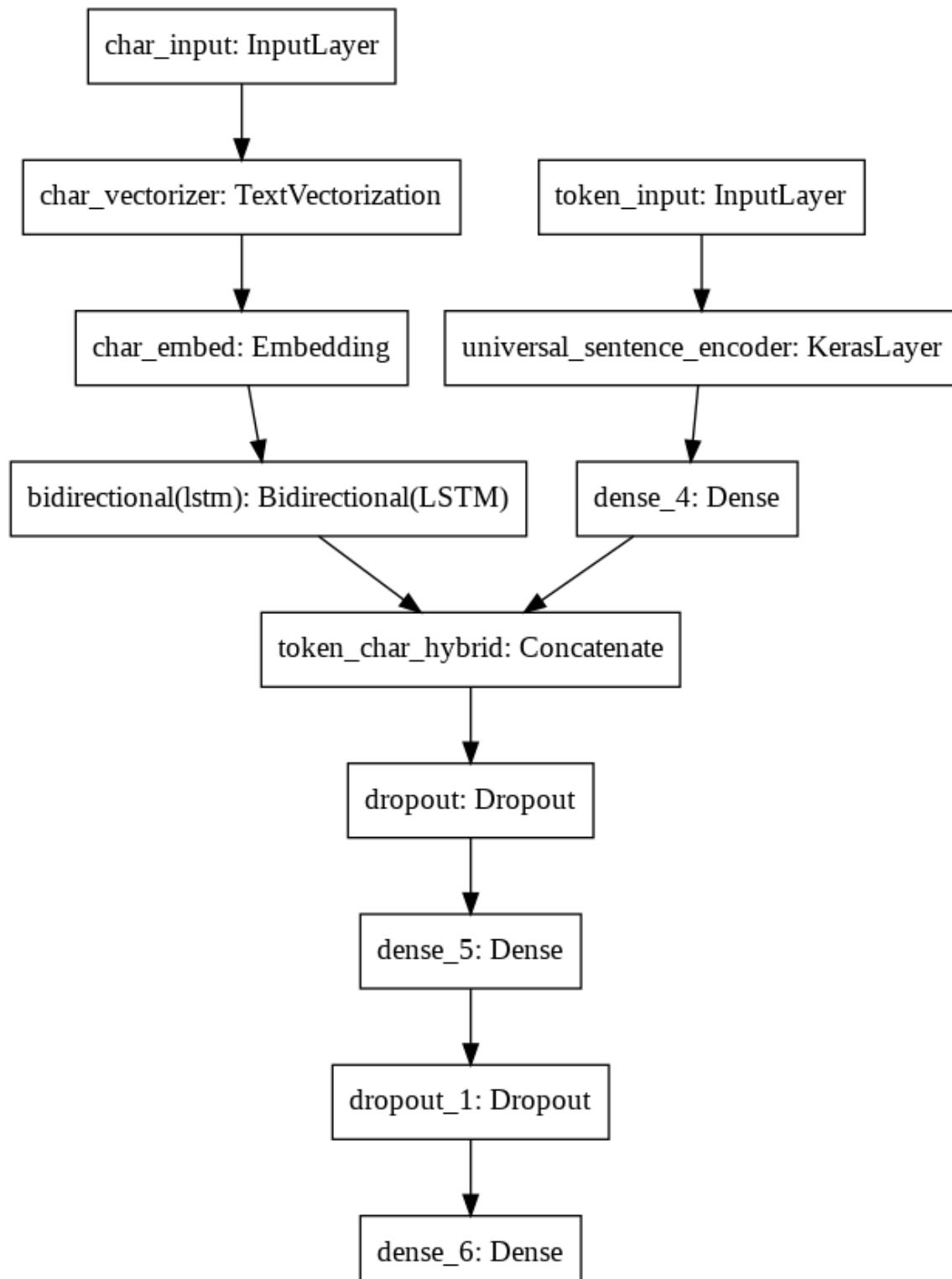
```
=====  
=====
```

Total params: 256,912,243  
Trainable params: 114,419  
Non-trainable params: 256,797,824

In [ ]:

```
# Plot hybrid token and character model
from tensorflow.keras.utils import plot_model
plot_model(model_4)
```

Out[ ]:



Now that's a good looking model. Let's compile it just as we have the rest of our models.

**Note:** Section 4.2 of [Neural Networks for Joint Sentence Classification in Medical Paper Abstracts](#) mentions using the SGD (stochastic gradient descent) optimizer, however, to stay consistent with our other models, we're going to use the Adam optimizer. As an exercise, you could try using `tf.keras.optimizers.SGD` instead of `tf.keras.optimizers.Adam` and compare the results.

In [ ]:

```
# Compile token char model
model_4.compile(loss="categorical_crossentropy",
                  optimizer=tf.keras.optimizers.Adam(), # section 4.2 of https://arxiv.org/pdf/1612.05251.pdf mentions using SGD but we'll stick with Adam
                  metrics=["accuracy"])
```

And again, to keep our experiments fast, we'll fit our token-character-hybrid model on 10% of training and validate on 10% of validation batches. However, the difference with this model is that it requires two inputs, token-level sequences and character-level sequences.

We can do this by create a `tf.data.Dataset` with a tuple as its first input, for example:

- `((token_data, char_data), (label))`

Let's see it in action.

## Combining token and character data into a `tf.data` dataset

In [ ]:

```
# Combine chars and tokens into a dataset
train_char_token_data = tf.data.Dataset.from_tensor_slices((train_sentences, train_chars)) # make data
train_char_token_labels = tf.data.Dataset.from_tensor_slices(train_labels_one_hot) # make labels
train_char_token_dataset = tf.data.Dataset.zip((train_char_token_data, train_char_token_labels)) # combine data and labels

# Prefetch and batch train data
train_char_token_dataset = train_char_token_dataset.batch(32).prefetch(tf.data.AUTOTUNE)

# Repeat same steps validation data
val_char_token_data = tf.data.Dataset.from_tensor_slices((val_sentences, val_chars))
val_char_token_labels = tf.data.Dataset.from_tensor_slices(val_labels_one_hot)
val_char_token_dataset = tf.data.Dataset.zip((val_char_token_data, val_char_token_labels))
val_char_token_dataset = val_char_token_dataset.batch(32).prefetch(tf.data.AUTOTUNE)
```

In [ ]:

```
# Check out training char and token embedding dataset
train_char_token_dataset, val_char_token_dataset
```

Out[ ]:

```
(<PrefetchDataset shapes: (((None,), (None,)), (None, 5)), types: ((tf.string, tf.string), tf.float64)>,
 <PrefetchDataset shapes: (((None,), (None,)), (None, 5)), types: ((tf.string, tf.string), tf.float64)>)
```

## Fitting a model on token and character-level sequences

In [ ]:

```
# Fit the model on tokens and chars
model_4_history = model_4.fit(train_char_token_dataset, # train on dataset of token and characters
                             steps_per_epoch=int(0.1 * len(train_char_token_dataset)),
                             epochs=3,
                             validation_data=val_char_token_dataset,
                             validation_steps=int(0.1 * len(val_char_token_dataset)))
```

```
Epoch 1/3
562/562 [=====] - 24s 36ms/step - loss: 0.9654 - accuracy: 0.615
9 - val_loss: 0.7859 - val_accuracy: 0.6898
Epoch 2/3
562/562 [=====] - 19s 34ms/step - loss: 0.7914 - accuracy: 0.695
8 - val_loss: 0.7139 - val_accuracy: 0.7301
Epoch 3/3
562/562 [=====] - 19s 34ms/step - loss: 0.7649 - accuracy: 0.705
7 - val_loss: 0.6826 - val_accuracy: 0.7410
```

In [ ]:

```
# Evaluate on the whole validation dataset
model_4.evaluate(val_char_token_dataset)
```

```
945/945 [=====] - 20s 21ms/step - loss: 0.6899 - accuracy: 0.7362
2
```

Out[ ]:

```
[0.6899493336677551, 0.7362306118011475]
```

**Nice! Our token-character hybrid model has come to life!**

To make predictions with it, since it takes multiple inputs, we can pass the `predict()` method a tuple of token-level sequences and character-level sequences.

We can then evaluate the predictions as we've done before.

In [ ]:

```
# Make predictions using the token-character model hybrid
model_4_pred_probs = model_4.predict(val_char_token_dataset)
model_4_pred_probs
```

Out[ ]:

```
array([[4.5224771e-01, 3.3035564e-01, 2.7360097e-03, 2.0863818e-01,
       6.0224836e-03],
       [2.7638477e-01, 5.4655772e-01, 3.0270391e-03, 1.7166287e-01,
       2.3676162e-03],
       [3.1192392e-01, 1.4092967e-01, 4.3908428e-02, 4.5166326e-01,
       5.1574774e-02],
       ...,
       [6.1459269e-04, 5.8821058e-03, 4.2544805e-02, 1.3780949e-04,
       9.5082068e-01],
       [9.3976045e-03, 7.2958224e-02, 1.8798770e-01, 4.0314477e-03,
       7.2562498e-01],
       [2.6598454e-01, 3.5141158e-01, 2.8458366e-01, 3.5951469e-02,
       6.2068779e-02]], dtype=float32)
```

In [ ]:

```
# Turn prediction probabilities into prediction classes
model_4_preds = tf.argmax(model_4_pred_probs, axis=1)
model_4_preds
```

Out[ ]:

```
<tf.Tensor: shape=(30212,), dtype=int64, numpy=array([0, 1, 3, ..., 4, 4, 1])>
```

In [ ]:

```
# Get results of token-character-hybrid model
```

```
# GET RESULTS OF TOKEN-LEVEL MODEL
model_4_results = calculate_results(y_true=val_labels_encoded,
                                    y_pred=model_4_preds)
model_4_results
```

Out[ ]:

```
{'accuracy': 73.62306368330465,
 'f1': 0.7335613153664351,
 'precision': 0.7370751903809197,
 'recall': 0.7362306368330465}
```

## Model 5: Transfer Learning with pretrained token embeddings + character embeddings + positional embeddings

It seems like combining token embeddings and character embeddings gave our model a little performance boost.

But there's one more piece of the puzzle we can add in.

What if we engineered our own features into the model?

Meaning, what if we took our own knowledge about the data and encoded it in a numerical way to give our model more information about our samples?

The process of applying your own knowledge to build features as input to a model is called **feature engineering**.

Can you think of something important about the sequences we're trying to classify?

If you were to look at an abstract, would you expect the sentences to appear in order? Or does it make sense if they were to appear sequentially? For example, sequences labelled CONCLUSIONS at the beginning and sequences labelled OBJECTIVE at the end?

Abstracts typically come in a sequential order, such as:

- OBJECTIVE ...
- METHODS ...
- METHODS ...
- METHODS ...
- RESULTS ...
- CONCLUSIONS ...

Or

- BACKGROUND ...
- OBJECTIVE ...
- METHODS ...
- METHODS ...
- RESULTS ...
- RESULTS ...
- CONCLUSIONS ...
- CONCLUSIONS ...

Of course, we can't engineer the sequence labels themselves into the training data (we don't have these at test time), but we can encode the order of a set of sequences in an abstract.

For example,

- Sentence 1 of 10 ...
- Sentence 2 of 10 ...
- Sentence 3 of 10 ...
- Sentence 4 of 10 ...
- ...

You might've noticed this when we created our `preprocess_text_with_line_numbers()` function. When we read in a text file of abstracts, we counted the number of lines in an abstract as well as the number of each line itself.

Doing this led to the `"line_number"` and `"total_lines"` columns of our DataFrames.

In [ ]:

```
# Inspect training dataframe  
train_df.head()
```

Out [ ]:

target	text	line_number	total_lines
0 OBJECTIVE	to investigate the efficacy of @ weeks of dail...	0	11
1 METHODS	a total of @ patients with primary knee oa wer...	1	11
2 METHODS	outcome measures included pain reduction and i...	2	11
3 METHODS	pain was assessed using the visual analog pain...	3	11
4 METHODS	secondary outcome measures included the wester...	4	11

The `"line_number"` and `"total_lines"` columns are features which didn't necessarily come with the training data but can be passed to our model as a **positional embedding**. In other words, the positional embedding is where the sentence appears in an abstract.

We can use these features because they will be available at test time.

Nutritional psychiatry: the present state of the evidence

Wolfgang Marx <sup>1</sup>, Genevieve Moseley <sup>2</sup>, Michael Berk <sup>2</sup>, Felice Jacka <sup>2</sup>

Affiliations + expand

PMID: 28942748 DOI: 10.1017/S0029665117002026

**Abstract**

1 Mental illness, including depression, anxiety and bipolar disorder, accounts for a significant proportion of global disability and poses a substantial social, economic and health burden.

2 Treatment is presently dominated by pharmacotherapy, such as antidepressants, and psychotherapy, such as cognitive behavioural therapy; however, such treatments avert less than half of the disease burden, suggesting that additional strategies are needed to prevent and treat mental disorders. 3 here are now consistent mechanistic, observational and interventional data to suggest diet quality may be a modifiable risk factor for mental illness. 4 this review provides an overview of the nutritional psychiatry field. It includes a discussion of the neurobiological 5 mechanisms likely modulated by diet, the use of dietary and nutraceutical interventions in mental disorders, and recommendations for further research. Potential biological pathways related to 6 mental disorders include inflammation, oxidative stress, the gut microbiome, epigenetic 7 modifications and neuroplasticity. Consistent epidemiological evidence, particularly for depression, suggests an association between measures of diet quality and mental health, across multiple populations and age groups; these do not appear to be explained by other demographic, lifestyle 8 factors or reverse causality. Our recently published intervention trial provides preliminary clinical 8 evidence that dietary interventions in clinically diagnosed populations are feasible and can provide significant clinical benefit. Furthermore, nutraceuticals including n-3 fatty acids, folate, S- 9 adenosylmethionine, N-acetyl cysteine and probiotics, among others, are promising avenues for future research. Continued research is now required to investigate the efficacy of intervention 10 studies in large cohorts and within clinically relevant populations, particularly in patients with schizophrenia, bipolar and anxiety disorders.

Source: <https://pubmed.ncbi.nlm.nih.gov/28942748/>

- Are the features we've engineered available at test time?
  - Line numbers ✓
  - Total lines ✓

Since abstracts typically have a sequential order about them (for example, background, objective, methods, results, conclusion), it makes sense to add the line number of where a particular sentence occurs to our model. The beautiful thing is, these features will be available at test time (we can just count the number of sentences in an abstract and the number of each one).

Meaning, if we were to predict the labels of sequences in an abstract our model had never seen, we could count the number of lines and the track the position of each individual line and pass it to our model.

Exercise: Another way of creating our positional embedding feature would be to combine the `"line_number"` and `"total_lines"` columns into one, for example a `"line_position"` column may contain values like `1_of_11`, `2_of_11`, etc. Where `1_of_11` would be the first line in an abstract 11 sentences long. After going through the following steps, you might want to revisit this positional embedding stage and see how a combined column of `"line_position"` goes against two separate columns.

## Create positional embeddings

Okay, enough talk about positional embeddings, let's create them.

Since our "line\_number" and "total\_line" columns are already numerical, we could pass them as they are to our model.

But to avoid our model thinking a line with "line\_number"=5 is five times greater than a line with "line\_number"=1, we'll use one-hot-encoding to encode our "line\_number" and "total\_lines" features.

To do this, we can use the `tf.one_hot` utility.

`tf.one_hot` returns a one-hot-encoded tensor. It accepts an array (or tensor) as input and the `depth` parameter determines the dimension of the returned tensor.

To figure out what we should set the `depth` parameter to, let's investigate the distribution of the "line\_number" column.

**Note:** When it comes to one-hot-encoding our features, Scikit-Learn's `OneHotEncoder` class is another viable option here.

In [ ]:

```
# How many different line numbers are there?  
train_df["line_number"].value_counts()
```

Out[ ]:

```
0      15000  
1      15000  
2      15000  
3      15000  
4      14992  
5      14949  
6      14758  
7      14279  
8      13346  
9      11981  
10     10041  
11     7892  
12     5853  
13     4152  
14     2835  
15     1861  
16     1188  
17      751  
18      462  
19      286  
20      162  
21      101  
22       66  
23       33  
24       22  
25       14  
26        7  
27        4  
28        3  
29        1  
30        1  
Name: line_number, dtype: int64
```

In [ ]:



```
[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],  
[1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],  
[0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],  
[0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],  
[0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],  
[0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],  
[0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],  
[0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],  
[0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0.]],  
dtype=float32)>)
```

We can do the same as we've done for our "line\_number" column with the "total\_lines" column. First, let's find an appropriate value for the depth parameter of `tf.one_hot`.

In [ ]:

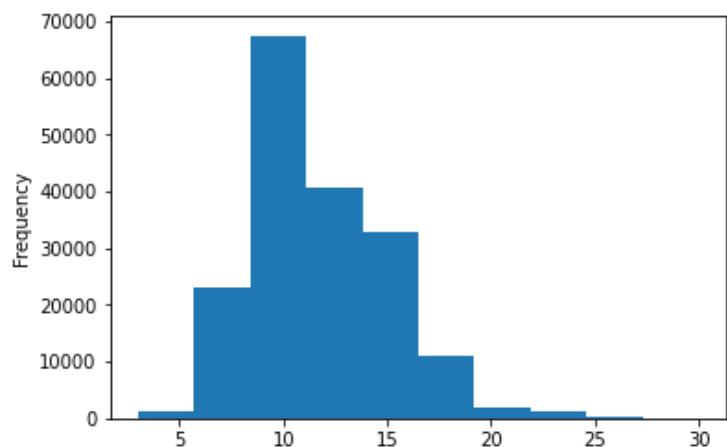
```
# How many different numbers of lines are there?  
train_df["total_lines"].value_counts()
```

Out[ ]:

```
11    24468  
10    23639  
12    22113  
9     19400  
13    18438  
14    14610  
8     12285  
15    10768  
7     7464  
16    7429  
17    5202  
6     3353  
18    3344  
19    2480  
20    1281  
5     1146  
21    770  
22    759  
23    264  
4     215  
24    200  
25    182  
26     81  
28     58  
3      32  
30     31  
27     28  
Name: total_lines, dtype: int64
```

In [ ]:

```
# Check the distribution of total lines  
train_df.total_lines.plot.hist();
```



Looking at the distribution of our `"total_lines"` column, a value of 20 looks like it covers the majority of samples.

We can confirm this with `np.percentile()`.

In [ ]:

```
# Check the coverage of a "total_lines" value of 20  
np.percentile(train_df.total_lines, 98) # a value of 20 covers 98% of samples
```

Out[ ]:

20.0

**Beautiful! Plenty of coverage. Let's one-hot-encode our "total\_lines" column just as we did our "line number" column.**

In [ ] :

```
# Use TensorFlow to create one-hot-encoded tensors of our "total_lines" column
train_total_lines_one_hot = tf.one_hot(train_df["total_lines"].to_numpy(), depth=20)
val_total_lines_one_hot = tf.one_hot(val_df["total_lines"].to_numpy(), depth=20)
test_total_lines_one_hot = tf.one_hot(test_df["total_lines"].to_numpy(), depth=20)

# Check shape and samples of total lines one-hot tensor
train_total_lines_one_hot.shape, train_total_lines_one_hot[:10]
```

Out[ ]:

## Building a tribrid embedding model

**Woohoo! Positional embedding tensors ready.**

**It's time to build the biggest model we've built yet. One which incorporates token embeddings, character embeddings and our newly crafted positional embeddings.**

**We'll be venturing into uncovered territory but there will be nothing here you haven't practiced before.**

**More specifically we're going to go through the following steps:**

1. Create a token-level model (similar to `model_1`)
  2. Create a character-level model (similar to `model_3` with a slight modification to reflect the paper)
  3. Create a `"line_number"` model (takes in one-hot-encoded `"line_number"` tensor and passes it through a non-linear layer)
  4. Create a `"total_lines"` model (takes in one-hot-encoded `"total_lines"` tensor and passes it through a non-linear layer)

5. Combine (using `layers.concatenate`) the outputs of 1 and 2 into a token-character-hybrid embedding and pass it series of output to Figure 1 and section 4.2 of [Neural Networks for Joint Sentence Classification in Medical Paper Abstracts](#)
6. Combine (using `layers.concatenate`) the outputs of 3, 4 and 5 into a token-character-positional tribrid embedding
7. Create an output layer to accept the tribrid embedding and output predicted label probabilities
8. Combine the inputs of 1, 2, 3, 4 and outputs of 7 into a `tf.keras.Model`

Woah! That's alot... but nothing we're not capable of. Let's code it.

In [ ]:

```
# 1. Token inputs
token_inputs = layers.Input(shape=[], dtype="string", name="token_inputs")
token_embeddings = tf_hub_embedding_layer(token_inputs)
token_outputs = layers.Dense(128, activation="relu")(token_embeddings)
token_model = tf.keras.Model(inputs=token_inputs,
                             outputs=token_embeddings)

# 2. Char inputs
char_inputs = layers.Input(shape=(1,), dtype="string", name="char_inputs")
char_vectors = char_vectorizer(char_inputs)
char_embeddings = char_embed(char_vectors)
char_bi_lstm = layers.Bidirectional(layers.LSTM(32))(char_embeddings)
char_model = tf.keras.Model(inputs=char_inputs,
                            outputs=char_bi_lstm)

# 3. Line numbers inputs
line_number_inputs = layers.Input(shape=(15,), dtype=tf.int32, name="line_number_input")
x = layers.Dense(32, activation="relu")(line_number_inputs)
line_number_model = tf.keras.Model(inputs=line_number_inputs,
                                   outputs=x)

# 4. Total lines inputs
total_lines_inputs = layers.Input(shape=(20,), dtype=tf.int32, name="total_lines_input")
y = layers.Dense(32, activation="relu")(total_lines_inputs)
total_line_model = tf.keras.Model(inputs=total_lines_inputs,
                                   outputs=y)

# 5. Combine token and char embeddings into a hybrid embedding
combined_embeddings = layers.concatenate(name="token_char_hybrid_embedding")([token_model.output,
                           char_model.output])
z = layers.Dense(256, activation="relu")(combined_embeddings)
z = layers.Dropout(0.5)(z)

# 6. Combine positional embeddings with combined token and char embeddings into a tribrid embedding
z = layers.concatenate(name="token_char_positional_embedding")([line_number_model.output,
                           total_line_model.output,
                           z])

# 7. Create output layer
output_layer = layers.Dense(5, activation="softmax", name="output_layer")(z)

# 8. Put together model
model_5 = tf.keras.Model(inputs=[line_number_model.input,
                                 total_line_model.input,
                                 token_model.input,
                                 char_model.input],
                           outputs=output_layer)
```

There's a lot going on here... let's visualize what's happening with a summary by plotting our model.

In [ ]:

```
# Get a summary of our token, char and positional embedding model  
model_5.summary()
```

Model: "model\_8"

Layer (type)	Output Shape	Param #	Connected to
char_inputs (InputLayer)	[None, 1]	0	
char_vectorizer (TextVectorizat	(None, 290)	0	char_inputs[0][0]
token_inputs (InputLayer)	[None, ]	0	
char_embed (Embedding)	(None, 290, 25)	1750	char_vectorizer[2][0]
universal_sentence_encoder (Ker	(None, 512)	256797824	token_inputs[0][0]
bidirectional_1 (Bidirectional)	(None, 64)	14848	char_embed[2][0]
token_char_hybrid_embedding (Co	(None, 576)	0	universal_sentence_enco
der[2][0]			bidirectional_1[0][0]
line_number_input (InputLayer)	[None, 15]	0	
total_lines_input (InputLayer)	[None, 20]	0	
dense_10 (Dense)	(None, 256)	147712	token_char_hybrid_embed
ding[0][0]			
dense_8 (Dense)	(None, 32)	512	line_number_input[0][0]
dense_9 (Dense)	(None, 32)	672	total_lines_input[0][0]
dropout_2 (Dropout)	(None, 256)	0	dense_10[0][0]
token_char_positional_embedding	(None, 320)	0	dense_8[0][0]
			dense_9[0][0]
			dropout_2[0][0]

```

output_layer (Dense)           (None, 5)          1605      token_char_positional_e
mbedding[0]
=====
=====
Total params: 256,964,923
Trainable params: 167,099
Non-trainable params: 256,797,824

```

---

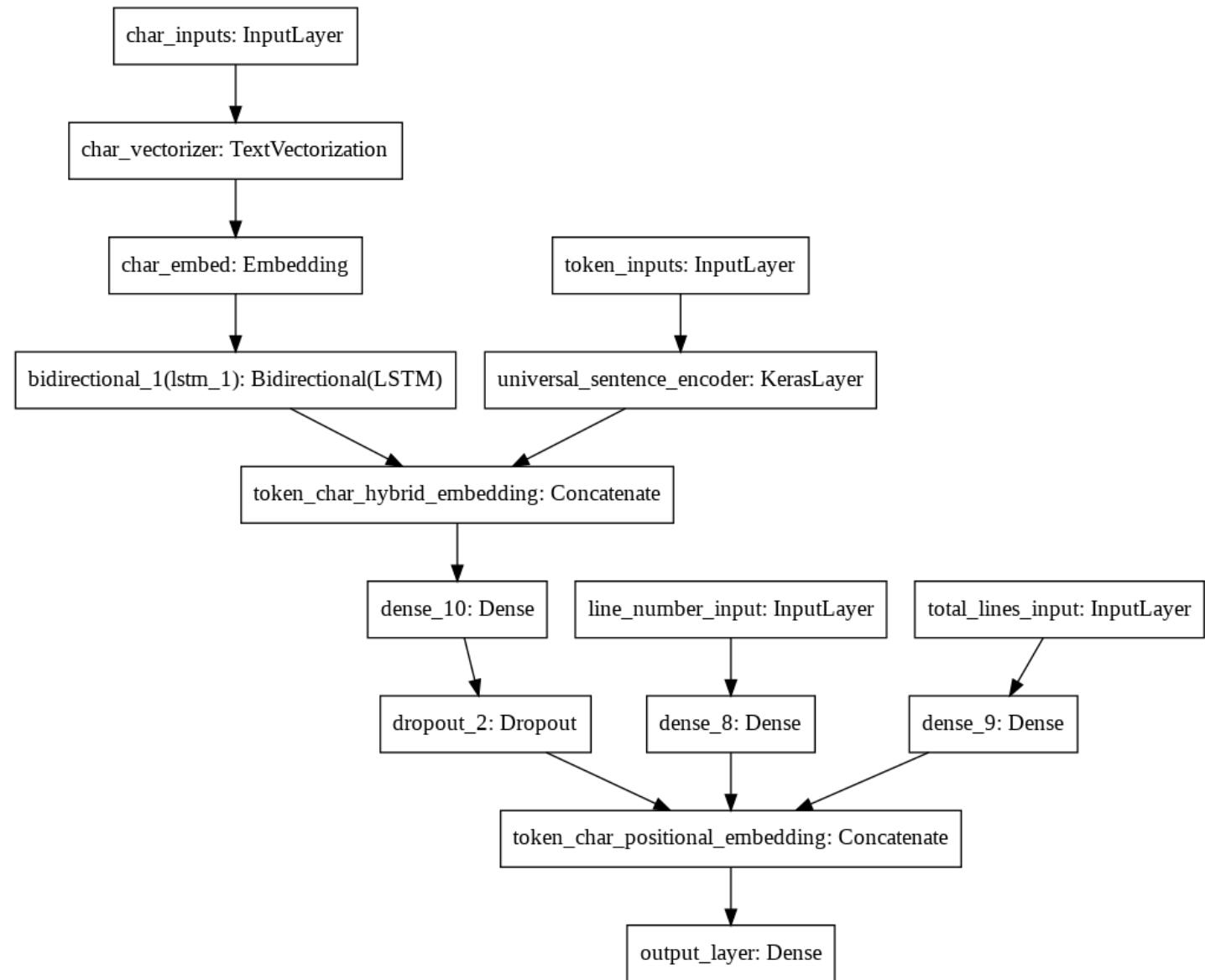


---

In [ ]:

```
# Plot the token, char, positional embedding model
from tensorflow.keras.utils import plot_model
plot_model(model_5)
```

Out [ ]:



**Visualizing the model makes it much easier to understand.**

**Essentially what we're doing is trying to encode as much information about our sequences as possible into various embeddings (the inputs to our model) so our model has the best chance to figure out what label belongs to a sequence (the outputs of our model).**

You'll notice our model is looking very similar to the model shown in Figure 1 of [Neural Networks for Joint Sentence Classification in Medical Paper Abstracts](#). However, a few differences still remain:

- We're using pretrained TensorFlow Hub token embeddings instead of GloVe emebddings.
- We're using a Dense layer on top of our token-character hybrid embeddings instead of a bi-LSTM layer.
- Section 3.1.3 of the paper mentions a label sequence optimization layer (which helps to make sure sequence labels come out in a respectable order) but it isn't shown in Figure 1. To makeup for the lack of this layer in

our model, we've created the positional embeddings layers.

- Section 4.2 of the paper mentions the token and character embeddings are updated during training, our pretrained TensorFlow Hub embeddings remain frozen.
- The paper uses the SGD optimizer, we're going to stick with Adam.

All of the differences above are potential extensions of this project.

In [ ]:

```
# Check which layers of our model are trainable or not
for layer in model_5.layers:
    print(layer, layer.trainable)

<keras.engine.input_layer.InputLayer object at 0x7f1fa3d725d0> True
<keras.layers.preprocessing.text_vectorization.TextVectorization object at 0x7f1ecec2d0d0>
> True
<keras.engine.input_layer.InputLayer object at 0x7f1fa3d22210> True
<keras.layers.embeddings.Embedding object at 0x7f1ecec8d0> True
<tensorflow_hub.keras_layer.KerasLayer object at 0x7f1fa586e5d0> False
<keras.layers.wrappers.Bidirectional object at 0x7f1fa3dff090> True
<keras.layers.merge.Concatenate object at 0x7f1fb14effd0> True
<keras.engine.input_layer.InputLayer object at 0x7f1fa3ec1890> True
<keras.engine.input_layer.InputLayer object at 0x7f1fa3ef9610> True
<keras.layers.core.Dense object at 0x7f1fb14efbd0> True
<keras.layers.core.Dense object at 0x7f1fa3d9b110> True
<keras.layers.core.Dense object at 0x7f1fb14e4f10> True
<keras.layers.core.Dropout object at 0x7f1fb14efed0> True
<keras.layers.merge.Concatenate object at 0x7f1fb14e49d0> True
<keras.layers.core.Dense object at 0x7f1fb14f3790> True
```

Now our model is constructed, let's compile it.

This time, we're going to introduce a new parameter to our loss function called `label_smoothing`. Label smoothing helps to regularize our model (prevent overfitting) by making sure it doesn't get too focused on applying one particular label to a sample.

For example, instead of having an output prediction of:

- `[0.0, 0.0, 1.0, 0.0, 0.0]` for a sample (the model is very confident the right label is index 2).

It's predictions will get smoothed to be something like:

- `[0.01, 0.01, 0.096, 0.01, 0.01]` giving a small activation to each of the other labels, in turn, hopefully improving generalization.

Resource: For more on label smoothing, see the great blog post by PyImageSearch, [Label smoothing with Keras, TensorFlow, and Deep Learning](#).

In [ ]:

```
# Compile token, char, positional embedding model
model_5.compile(loss=tf.keras.losses.CategoricalCrossentropy(label_smoothing=0.2), # add
label_smoothing (examples which are really confident get smoothed a little)
optimizer=tf.keras.optimizers.Adam(),
metrics=["accuracy"])
```

## Create tribrid embedding datasets and fit tribrid model

Model compiled!

Again, to keep our experiments swift, let's fit on 20,000 examples for 3 epochs.

This time our model requires four feature inputs:

1. Train line numbers one-hot tensor (`train_line_numbers_one_hot`)
2. Train total lines one-hot tensor (`train_total_lines_one_hot`)

### 3. Token-level sequences tensor (`train_sentences`)

### 4. Char-level sequences tensor (`train_chars`)

We can pass these as tuples to our `tf.data.Dataset.from_tensor_slices()` method to create appropriately shaped and batched `PrefetchedDataset`'s.

In [ ]:

```
# Create training and validation datasets (all four kinds of inputs)
train_pos_char_token_data = tf.data.Dataset.from_tensor_slices((train_line_numbers_one_hot, # line numbers
                                                               train_total_lines_one_h
ot, # total lines
                                                               train_sentences, # tra
n tokens
                                                               train_chars)) # train c
hars
train_pos_char_token_labels = tf.data.Dataset.from_tensor_slices(train_labels_one_hot) # train labels
train_pos_char_token_dataset = tf.data.Dataset.zip((train_pos_char_token_data, train_pos_
char_token_labels)) # combine data and labels
train_pos_char_token_dataset = train_pos_char_token_dataset.batch(32).prefetch(tf.data.AU
TOTUNE) # turn into batches and prefetch appropriately

# Validation dataset
val_pos_char_token_data = tf.data.Dataset.from_tensor_slices((val_line_numbers_one_hot,
                                                               val_total_lines_one_hot,
                                                               val_sentences,
                                                               val_chars))
val_pos_char_token_labels = tf.data.Dataset.from_tensor_slices(val_labels_one_hot)
val_pos_char_token_dataset = tf.data.Dataset.zip((val_pos_char_token_data, val_pos_char_t
oken_labels))
val_pos_char_token_dataset = val_pos_char_token_dataset.batch(32).prefetch(tf.data.AUTOT
UNE) # turn into batches and prefetch appropriately

# Check input shapes
train_pos_char_token_dataset, val_pos_char_token_dataset
```

Out[ ]:

```
(<PrefetchDataset shapes: (((None, 15), (None, 20), (None,), (None,)), (None, 5)), types:
((tf.float32, tf.float32, tf.string, tf.string), tf.float64)>,
 <PrefetchDataset shapes: (((None, 15), (None, 20), (None,), (None,)), (None, 5)), types:
((tf.float32, tf.float32, tf.string, tf.string), tf.float64)>)
```

In [ ]:

```
# Fit the token, char and positional embedding model
history_model_5 = model_5.fit(train_pos_char_token_dataset,
                               steps_per_epoch=int(0.1 * len(train_pos_char_token_dataset
)),
                               epochs=3,
                               validation_data=val_pos_char_token_dataset,
                               validation_steps=int(0.1 * len(val_pos_char_token_dataset
)))
```

```
Epoch 1/3
562/562 [=====] - 24s 36ms/step - loss: 1.1013 - accuracy: 0.726
0 - val_loss: 0.9930 - val_accuracy: 0.8002
Epoch 2/3
562/562 [=====] - 19s 34ms/step - loss: 0.9771 - accuracy: 0.811
4 - val_loss: 0.9606 - val_accuracy: 0.8268
Epoch 3/3
562/562 [=====] - 19s 34ms/step - loss: 0.9627 - accuracy: 0.818
0 - val_loss: 0.9493 - val_accuracy: 0.8271
```

**Tribrid model trained! Time to make some predictions with it and evaluate them just as we've done before.**

In [ ]:

```
# Make predictions with token-char-positional hybrid model
model_5_pred_probs = model_5.predict(val_pos_char_token_dataset, verbose=1)
model_5_pred_probs
```

```
945/945 [=====] - 20s 20ms/step
```

Out [ ]:

```
array([[0.51536554, 0.10340027, 0.01223736, 0.34324795, 0.02574881],
       [0.5037048 , 0.1263607 , 0.0476622 , 0.3120701 , 0.01020223],
       [0.31137902, 0.10944027, 0.11880615, 0.3917513 , 0.06862326],
       ...,
       [0.04232275, 0.09047632, 0.04658423, 0.02905692, 0.7915597 ],
       [0.03812133, 0.3116883 , 0.10215054, 0.02388792, 0.5241519 ],
       [0.18210074, 0.5038779 , 0.18253621, 0.03620264, 0.0952825 ]],
      dtype=float32)
```

In [ ]:

```
# Turn prediction probabilities into prediction classes
model_5_preds = tf.argmax(model_5_pred_probs, axis=1)
model_5_preds
```

Out [ ]:

```
<tf.Tensor: shape=(30212,), dtype=int64, numpy=array([0, 0, 3, ..., 4, 4, 1])>
```

In [ ]:

```
# Calculate results of token-char-positional hybrid model
model_5_results = calculate_results(y_true=val_labels_encoded,
                                      y_pred=model_5_preds)
model_5_results
```

Out [ ]:

```
{'accuracy': 82.6128690586522,
 'f1': 0.8250369638872138,
 'precision': 0.8244488224211757,
 'recall': 0.8261286905865219}
```

## Compare model results

Far out, we've come a long way. From a baseline model to training a model containing three different kinds of embeddings.

Now it's time to compare each model's performance against each other.

We'll also be able to compare our model's to the [PubMed 200k RCT: a Dataset for Sequential Sentence Classification in Medical Abstracts](#) paper.

Since all of our model results are in dictionaries, let's combine them into a pandas DataFrame to visualize them.

In [ ]:

```
# Combine model results into a DataFrame
all_model_results = pd.DataFrame({"baseline": baseline_results,
                                    "custom_token_embed_conv1d": model_1_results,
                                    "pretrained_token_embed": model_2_results,
                                    "custom_char_embed_conv1d": model_3_results,
                                    "hybrid_char_token_embed": model_4_results,
                                    "tribrid_pos_char_token_embed": model_5_results})
all_model_results = all_model_results.transpose()
all_model_results
```

Out [ ]:

	accuracy	precision	recall	f1
<b>baseline</b>	<b>72.183238</b>	<b>0.718647</b>	<b>0.721832</b>	<b>0.698925</b>
<b>custom token embed conv1d</b>	<b>78.448961</b>	<b>0.781494</b>	<b>0.784490</b>	<b>0.782299</b>

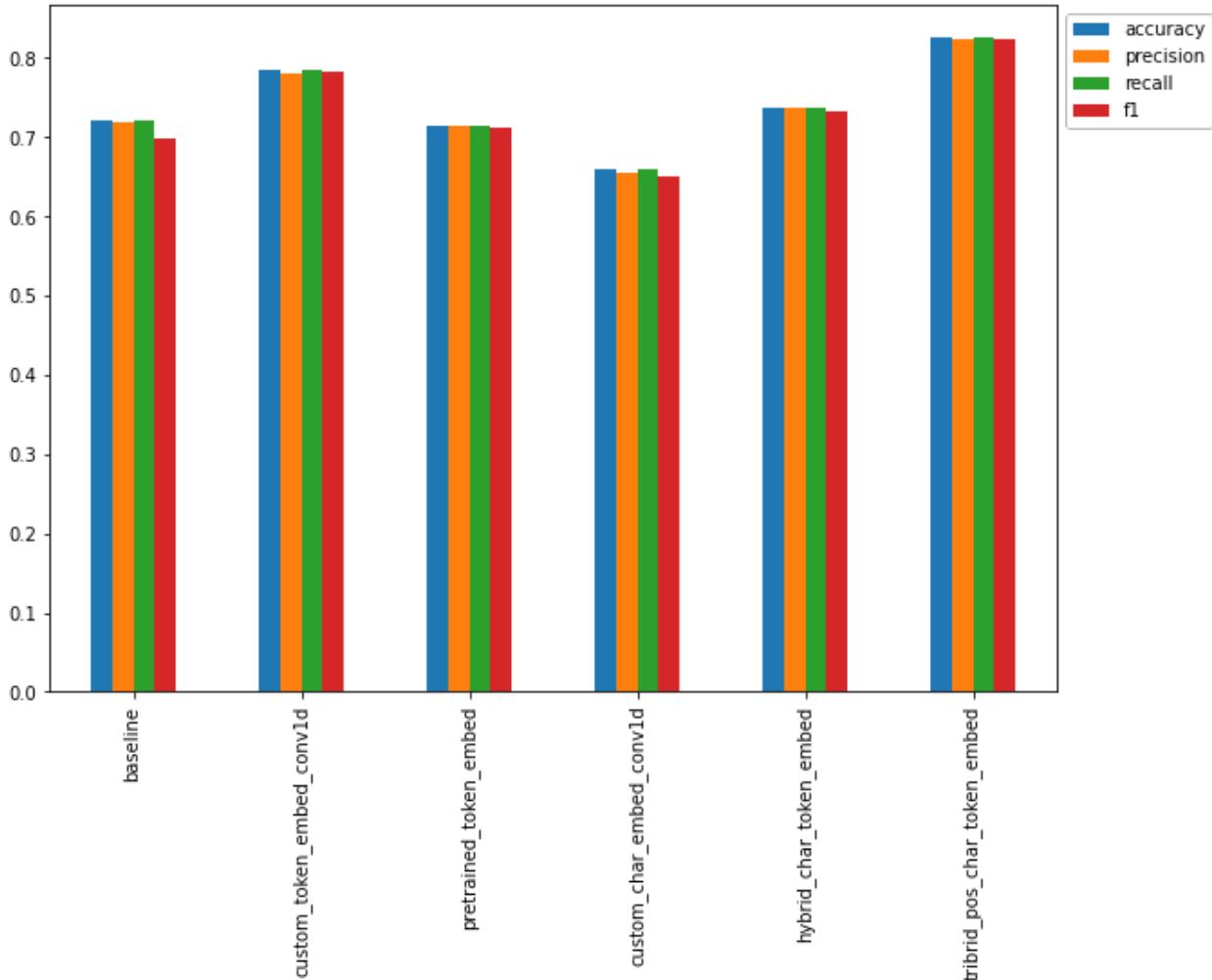
	accuracy	precision	recall	f1
pretrained_token_embed	71.42526	0.714881	0.714253	0.711455
custom_char_embed_conv1d	65.877797	0.654501	0.658778	0.651686
hybrid_char_token_embed	73.623064	0.737075	0.736231	0.733561
tribrid_pos_char_token_embed	82.612869	0.824449	0.826129	0.825037

In [ ]:

```
# Reduce the accuracy to same scale as other metrics
all_model_results["accuracy"] = all_model_results["accuracy"]/100
```

In [ ]:

```
# Plot and compare all of the model results
all_model_results.plot(kind="bar", figsize=(10, 7)).legend(bbox_to_anchor=(1.0, 1.0));
```



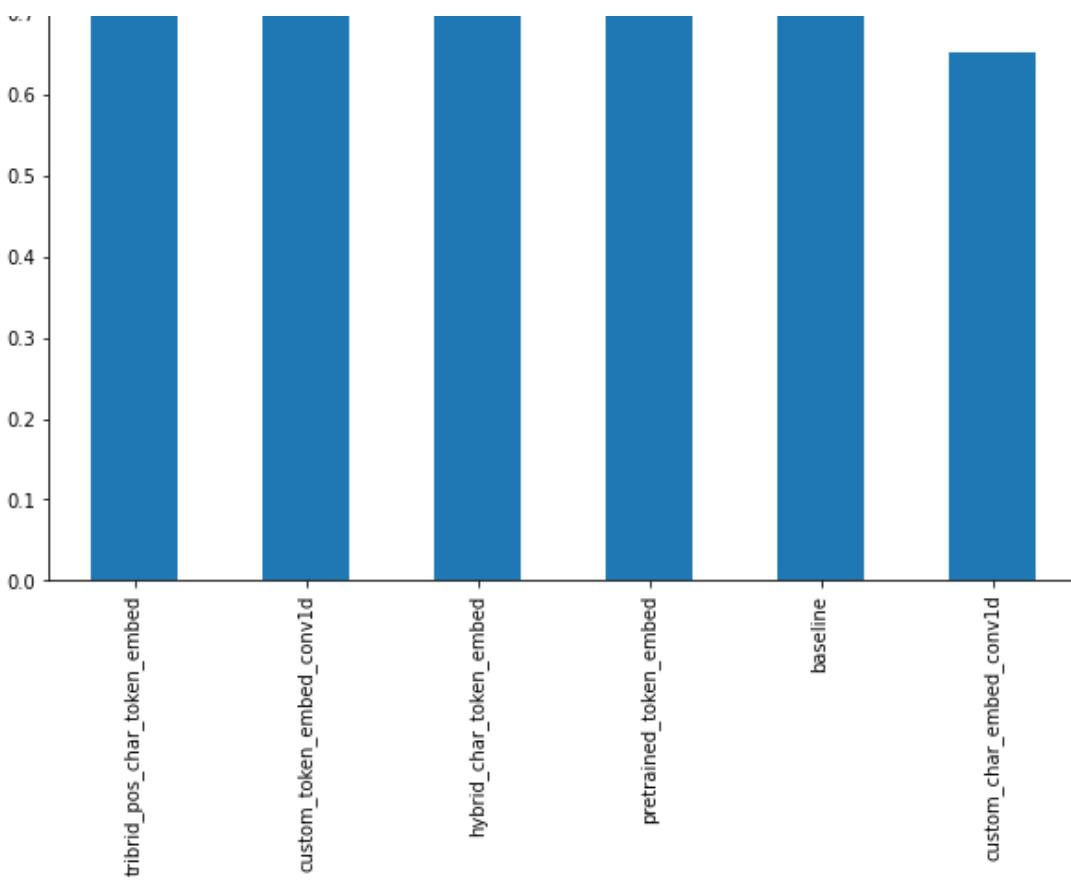
Since the [PubMed 200k RCT: a Dataset for Sequential Sentence Classification in Medical Abstracts](#) paper compares their tested model's F1-scores on the test dataset, let's take at our model's F1-scores.

▀ Note: We could've also made these comparisons in TensorBoard using the [TensorBoard](#) callback during training.

In [ ]:

```
# Sort model results by f1-score
all_model_results.sort_values("f1", ascending=False)[["f1"]].plot(kind="bar", figsize=(10, 7));
```





Nice! Based on F1-scores, it looks like our tribrid embedding model performs the best by a fair margin.

Though, in comparison to the results reported in Table 3 of the [PubMed 200k RCT: a Dataset for Sequential Sentence Classification in Medical Abstracts](#) paper, our model's F1-score is still underperforming (the authors model achieves an F1-score of 90.0 on the 20k RCT dataset versus our F1-score of ~82.6).

There are some things to note about this difference:

- Our models (with an exception for the baseline) have been trained on ~18,000 (10% of batches) samples of sequences and labels rather than the full ~180,000 in the 20k RCT dataset.
  - This is often the case in machine learning experiments though, make sure training works on a smaller number of samples, then upscale when needed (an extension to this project will be training a model on the full dataset).
- Our model's prediction performance levels have been evaluated on the validation dataset not the test dataset (we'll evaluate our best model on the test dataset shortly).

## Save and load best performing model

Since we've been through a fair few experiments, it's a good idea to save our best performing model so we can reuse it without having to retrain it.

We can save our best performing model by calling the `save()` method on it.

In [ ]:

```
# Save best performing model to SavedModel format (default)
model_5.save("skimlit_tribrid_model") # model will be saved to path specified by string
```

```
WARNING:absl:Found untraced functions such as lstm_cell_4_layer_call_and_return_condition
al_losses, lstm_cell_4_layer_call_fn, lstm_cell_5_layer_call_and_return_conditional_
losses, lstm_cell_5_layer_call_fn, lstm_cell_4_layer_call_fn while saving (showing 5 of 10). These
functions will not be directly callable after loading.
```

```
INFO:tensorflow:Assets written to: skimlit_tribrid_model/assets
```

```
INFO:tensorflow:Assets written to: skimlit_tribrid_model/assets
```

**Optional: If you're using Google Colab, you might want to copy your saved model to Google Drive (or [download](#))**

[it](#)) for more permanent storage (Google Colab files disappear after you disconnect).

In [ ] :

```
# Example of copying saved model from Google Colab to Drive (requires Google Drive to be mounted)
# !cp skim_lit_best_model -r /content/drive/MyDrive/tensorflow_course/skim_lit
```

Like all good cooking shows, we've got a pretrained model (exactly the same kind of model we built for model 5 [saved and stored on Google Storage](#)).

**So to make sure we're all using the same model for evaluation, we'll download it and load it in.**

And when loading in our model, since it uses a couple of `custom objects` (our TensorFlow Hub layer and `TextVectorization` `layer`), we'll have to load it in by specifying them in the `custom_objects` parameter of `tf.keras.models.load_model()`.

In [ ] :

```
# Download pretrained model from Google Storage
!wget https://storage.googleapis.com/ztm_tf_course/skimlit/skimlit_tribrid_model.zip
!mkdir skimlit_gs_model
!unzip skimlit_tribrid_model.zip -d skimlit_gs_model

--2021-08-25 00:03:10-- https://storage.googleapis.com/ztm_tf_course/skimlit/skimlit_tribrid_model.zip
Resolving storage.googleapis.com (storage.googleapis.com)... 142.250.141.128, 142.251.2.1
28, 74.125.137.128, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|142.250.141.128|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 962561955 (918M) [application/zip]
Saving to: 'skimlit_tribrid_model.zip'

skimlit_tribrid_mod 100%[=====] 917.97M 40.9MB/s    in 12s

2021-08-25 00:03:23 (74.5 MB/s) - 'skimlit_tribrid_model.zip' saved [962561955/962561955]

Archive: skimlit_tribrid_model.zip
  creating: skimlit_gs_model/skimlit_tribrid_model/
  creating: skimlit_gs_model/skimlit_tribrid_model/variables/
  inflating: skimlit_gs_model/skimlit_tribrid_model/variables/variables.index
  inflating: skimlit_gs_model/skimlit_tribrid_model/variables/variables.data-00000-of-000
01
  inflating: skimlit_gs_model/skimlit_tribrid_model/keras_metadata.pb
  inflating: skimlit_gs_model/skimlit_tribrid_model/saved_model.pb
  creating: skimlit_gs_model/skimlit_tribrid_model/assets/
```

In [ ] :

```
r}) # required for token embedding
```

## Make predictions and evalaute them against the truth labels

To make sure our model saved and loaded correctly, let's make predictions with it, evaluate them and then compare them to the prediction results we calculated earlier.

In [ ]:

```
# Make predictions with the loaded model on the validation set
loaded_pred_probs = loaded_model.predict(val_pos_char_token_dataset, verbose=1)
loaded_preds = tf.argmax(loaded_pred_probs, axis=1)
loaded_preds[:10]
```

945/945 [=====] - 132s 139ms/step

Out[ ]:

```
<tf.Tensor: shape=(10,), dtype=int64, numpy=array([0, 0, 3, 2, 2, 4, 4, 4, 4, 1])>
```

In [ ]:

```
# Evaluate loaded model's predictions
loaded_model_results = calculate_results(val_labels_encoded,
                                         loaded_preds)
loaded_model_results
```

Out[ ]:

```
{'accuracy': 82.74526678141136,
 'f1': 0.8264355957043299,
 'precision': 0.8258640600563426,
 'recall': 0.8274526678141136}
```

Now let's compare our loaded model's predictions with the prediction results we obtained before saving our model.

In [ ]:

```
# Compare loaded model results with original trained model results (should be quite close)
np.isclose(list(model_5_results.values()), list(loaded_model_results.values()), rtol=1e-02)
```

Out[ ]:

```
array([ True,  True,  True,  True])
```

It's worth noting that loading in a `SavedModel` unfreezes all layers (makes them all trainable). So if you want to freeze any layers, you'll have to set their `trainable` attribute to `False`.

In [ ]:

```
# Check loaded model summary (note the number of trainable parameters)
loaded_model.summary()
```

Model: "model\_8"

Layer (type)	Output Shape	Param #	Connected to
<hr/>			
char_inputs (InputLayer)	[ (None, 1) ]	0	
char_vectorizer (TextVectorizat	(None, None)	0	char_inputs[0][0]

token_inputs (InputLayer)	[ (None, ) ]	0	
char_embed (Embedding)	(None, None, 25)	1750	char_vectorizer[0][0]
universal_sentence_encoder (Ker	(None, 512)	256797824	token_inputs[0][0]
bidirectional_1 (Bidirectional)	(None, 64)	14848	char_embed[0][0]
token_char_hybrid_embedding (Co	(None, 576)	0	universal_sentence_enco
der[0][0]			bidirectional_1[0][0]
line_number_input (InputLayer)	[ (None, 15) ]	0	
total_lines_input (InputLayer)	[ (None, 20) ]	0	
dense_10 (Dense)	(None, 256)	147712	token_char_hybrid_embed
ding[0][0]			
dense_8 (Dense)	(None, 32)	512	line_number_input[0][0]
dense_9 (Dense)	(None, 32)	672	total_lines_input[0][0]
dropout_2 (Dropout)	(None, 256)	0	dense_10[0][0]
token_char_positional_embedding (None, 320)		0	dense_8[0][0]
			dense_9[0][0]
			dropout_2[0][0]
output_layer (Dense)	(None, 5)	1605	token_char_positional_e
mbedding[0]			
=====			
Total params: 256,964,923			
Trainable params: 167,099			
Non-trainable params: 256,797,824			

## Evaluate model on test dataset

To make our model's performance more comparable with the results reported in Table 3 of the [PubMed 200k RCT: a Dataset for Sequential Sentence Classification in Medical Abstracts](#) paper, let's make predictions on the test dataset and evaluate them.

In [ ]:

```
# Create test dataset batch and prefetched
test_pos_char_token_data = tf.data.Dataset.from_tensor_slices((test_line_numbers_one_hot,
                                                               test_total_lines_one_hot
                                                               ,
                                                               test_sentences,
                                                               test_chars))
test_pos_char_token_labels = tf.data.Dataset.from_tensor_slices(test_labels_one_hot)
test_pos_char_token_dataset = tf.data.Dataset.zip((test_pos_char_token_data, test_pos_char_token_labels))
test_pos_char_token_dataset = test_pos_char_token_dataset.batch(32).prefetch(tf.data.AUTOTUNE)

# Check shapes
test_pos_char_token_dataset
```

Out[ ]:

```
<PrefetchDataset shapes: (((None, 15), (None, 20), (None,), (None,)), (None, 5)), types: ((tf.float32, tf.float32, tf.string, tf.string), tf.float64)>
```

In [ ] :

```
# Make predictions on the test dataset
test_pred_probs = loaded_model.predict(test_pos_char_token_dataset,
                                         verbose=1)
test_preds = tf.argmax(test_pred_probs, axis=1)
test_preds[:10]
```

942/942 [=====] - 132s 140ms/step

Out[ ]:

```
<tf.Tensor: shape=(10,), dtype=int64, numpy=array([3, 3, 2, 2, 4, 4, 4, 4, 1, 4, 0])>
```

In [ ]:

```
# Evaluate loaded model test predictions
loaded_model_test_results = calculate_results(y_true=test_labels_encoded,
                                              y_pred=test_preds)
loaded_model_test_results
```

Out [ ]:

```
{'accuracy': 82.39588518334163,  
 'f1': 0.8229369808171064,  
 'precision': 0.8225726116113812,  
 'recall': 0.8239588518334163}
```

It seems our best model (so far) still has some ways to go to match the performance of the results in the paper (their model gets 90.0 F1-score on the test dataset, whereas ours gets ~82.1 F1-score).

However, as we discussed before our model has only been trained on 20,000 out of the total ~180,000 sequences in the RCT 20k dataset. We also haven't fine-tuned our pretrained embeddings (the paper fine-tunes GloVe embeddings). So there's a couple of extensions we could try to improve our results.

# Find most wrong

One of the best ways to investigate where your model is going wrong (or potentially where your data is wrong) is to visualize the "most wrong" predictions.

The most wrong predictions are samples where the model has made a prediction with a high probability but has gotten it wrong (the model's prediction disagrees with the ground truth label).

**Looking at the most wrong predictions can give us valuable information on how to improve further models or fix the labels in our data.**

Let's write some code to help us visualize the most wrong predictions from the test dataset.

First we'll convert all of our integer-based test predictions into their string-based class names.

In [ ]:

```
%time
# Get list of class names of test predictions
test_pred_classes = [label_encoder.classes_[pred] for pred in test_preds]
test_pred_classes
```

CPU times: user 10.2 s, sys: 856 ms, total: 11.1 s  
Wall time: 9.42 s

Now we'll enrich our test DataFame with a few values:

- A "prediction" (string) column containing our model's prediction for a given sample.
- A "pred\_prob" (float) column containing the model's maximum prediction probability for a given sample.
- A "correct" (bool) column to indicate whether or not the model's prediction matches the sample's target label.

In [ ]:

```
# Create prediction-enriched test dataframe
test_df["prediction"] = test_pred_classes # create column with test prediction class names
test_df["pred_prob"] = tf.reduce_max(test_pred_probs, axis=1).numpy() # get the maximum prediction probability
test_df["correct"] = test_df["prediction"] == test_df["target"] # create binary column for whether the prediction is right or not
test_df.head(20)
```

Out[ ]:

	target	text	line_number	total_lines	prediction	pred_prob	correct
0	BACKGROUND	this study analyzed liver function abnormalities...	0	8	OBJECTIVE	0.513077	False
1	RESULTS	a post hoc analysis was conducted with the use...	1	8	OBJECTIVE	0.310540	False
2	RESULTS	liver function tests ( lfts ) were measured at...	2	8	METHODS	0.801705	False
3	RESULTS	survival analyses were used to assess the assoc...	3	8	METHODS	0.627319	False
4	RESULTS	the percentage of patients with abnormal lfts ...	4	8	RESULTS	0.718288	True
5	RESULTS	when mean hemodynamic profiles were compared i...	5	8	RESULTS	0.879730	True
6	RESULTS	multivariable analyses revealed that patients ...	6	8	RESULTS	0.548948	True
7	CONCLUSIONS	abnormal lfts are common in the adhf populatio...	7	8	CONCLUSIONS	0.445276	True
8	CONCLUSIONS	elevated meld-xi scores are associated with po...	8	8	RESULTS	0.529703	False
9	BACKGROUND	minimally invasive endovascular aneurysm repai...	0	12	BACKGROUND	0.545452	True
10	BACKGROUND	the aim of this study was to analyse the cost...	1	12	OBJECTIVE	0.495984	False
11	METHODS	resource use was determined from the amsterdam...	2	12	METHODS	0.587782	True
12	METHODS	the analysis was performed from a provider per...	3	12	METHODS	0.852491	True
13	METHODS	all costs were calculated as if all patients	4	12	METHODS	0.573058	True

	target	text	line_number	total_lines	prediction	pred_prob	correct
14	RESULTS	a total of @ patients were randomized .	5	12	RESULTS	0.674374	True
15	RESULTS	the @-day mortality rate was @ per cent after ...	6	12	RESULTS	0.664036	True
16	RESULTS	at @months , the total mortality rate for evar...	7	12	RESULTS	0.897093	True
17	RESULTS	the mean cost difference between evar and or w...	8	12	RESULTS	0.828620	True
18	RESULTS	the incremental cost-effectiveness ratio per p...	9	12	RESULTS	0.803249	True
19	RESULTS	there was no significant difference in quality...	10	12	RESULTS	0.729450	True

Looking good! Having our data like this, makes it very easy to manipulate and view in different ways.

How about we sort our DataFrame to find the samples with the highest "pred\_prob" and where the prediction was wrong ("correct" == False)?

In [ ]:

```
# Find top 100 most wrong samples (note: 100 is an arbitrary number, you could go through all of them if you wanted)
top_100_wrong = test_df[test_df["correct"] == False].sort_values("pred_prob", ascending=False)[:100]
top_100_wrong
```

Out [ ]:

	target	text	line_number	total_lines	prediction	pred_prob	correct
16347	BACKGROUND	to evaluate the effects of the lactic acid bac...	0	12	OBJECTIVE	0.944838	False
13874	CONCLUSIONS	symptom outcomes will be assessed and estimate...	4	6	METHODS	0.941099	False
1221	RESULTS	data were collected prospectively for @ months...	3	13	METHODS	0.928523	False
13598	METHODS	-@ % vs. fish : -@ % vs. fish + s : -@ % ; p <...	6	9	RESULTS	0.918107	False
21382	OBJECTIVE	design , settings , participants , and interve...	3	13	METHODS	0.918088	False
...	...	...	...	...	...	...	...
12269	RESULTS	patients received oral se tablets (@ mcg ) or...	4	10	METHODS	0.821220	False
9881	RESULTS	the primary outcome was bp control , and secon...	4	11	METHODS	0.821166	False
1220	RESULTS	the group intervention consisted of @ weekly c...	2	13	METHODS	0.821033	False
22105	RESULTS	we randomised @ statin treated cvd patients an...	3	12	METHODS	0.820954	False
16840	RESULTS	the primary endpoint was a composite of cardio...	3	12	METHODS	0.820538	False

100 rows x 7 columns

Great (or not so great)! Now we've got a subset of our model's most wrong predictions, let's write some code to visualize them.

In [ ]:

```
# Investigate top wrong preds
for row in top_100_wrong[0:10].itertuples(): # adjust indexes to view different samples
```

```
_ , target, text, line_number, total_lines, prediction, pred_prob, _ = row
print(f"Target: {target}, Pred: {prediction}, Prob: {pred_prob}, Line number: {line_number}, Total lines: {total_lines}\n")
print(f"Text:\n{text}\n")
print("-----\n")
```

Target: BACKGROUND, Pred: OBJECTIVE, Prob: 0.9448384046554565, Line number: 0, Total lines: 12

Text:  
to evaluate the effects of the lactic acid bacterium lactobacillus salivarius on caries risk factors .

-----

Target: CONCLUSIONS, Pred: METHODS, Prob: 0.9410986304283142, Line number: 4, Total lines: 6

Text:  
symptom outcomes will be assessed and estimates of cost-effectiveness made .

-----

Target: RESULTS, Pred: METHODS, Prob: 0.9285234808921814, Line number: 3, Total lines: 13

Text:  
data were collected prospectively for @ months beginning after completion of the first @ group clinic appointments ( @ months post randomization ) .

-----

Target: METHODS, Pred: RESULTS, Prob: 0.9181066155433655, Line number: 6, Total lines: 9

Text:  
-@ % vs. fish : -@ % vs. fish + s : -@ % ; p < @ ) but there were no significant differences between groups .

-----

Target: OBJECTIVE, Pred: METHODS, Prob: 0.9180881381034851, Line number: 3, Total lines: 13

Text:  
design , settings , participants , and intervention : ten healthy , normal-weight men were studied in randomized , double-blind fashion , each receiving a @-minute intraduodenal infusion of l-trp at @ ( total @ kcal ) or @ ( total @ kcal ) kcal/min or saline ( control ) .

-----

Target: METHODS, Pred: RESULTS, Prob: 0.9168640971183777, Line number: 5, Total lines: 7

Text:  
at this time , an as@ response was achieved by @ ( @ % ) and @ ( @ % ) patients in groups @ and @ , respectively ( p < @ for all ) .

-----

Target: RESULTS, Pred: METHODS, Prob: 0.9164432883262634, Line number: 3, Total lines: 16

Text:  
a cluster randomised trial was implemented with @,@ children in @ government primary schools on the south coast of kenya in @-@ .

-----

Target: BACKGROUND, Pred: OBJECTIVE, Prob: 0.914499819278717, Line number: 0, Total lines: 9

Text:  
to compare the efficacy of the newcastle infant dialysis and ultrafiltration system ( nidus ) with peritoneal dialysis ( pd ) and conventional haemodialysis ( hd ) in infants wei

ghing < @ kg .

-----

Target: RESULTS, Pred: METHODS, Prob: 0.9141932725906372, Line number: 4, Total lines: 13

Text:

baseline measures included sociodemographics , standardized anthropometrics , asthma control test ( act ) , gerd symptom assessment scale , pittsburgh sleep quality index , and berlin questionnaire for sleep apnea .

-----

Target: BACKGROUND, Pred: OBJECTIVE, Prob: 0.9109073877334595, Line number: 0, Total lines: 11

Text:

to assess the temporal patterns of late gastrointestinal ( gi ) and genitourinary ( gu ) radiotherapy toxicity and resolution rates in a randomised controlled trial ( all-ireland cooperative oncology research group @-@ ) assessing duration of neo-adjuvant ( na ) hormone therapy for localised prostate cancer .

-----

**What do you notice about the most wrong predictions? Does the model make silly mistakes? Or are some of the labels incorrect/ambiguous (e.g. a line in an abstract could potentially be labelled OBJECTIVE or BACKGROUND and make sense).**

**A next step here would be if there are a fair few samples with inconsistent labels, you could go through your training dataset, update the labels and then retrain a model. The process of using a model to help improve/investigate your dataset's labels is often referred to as active learning.**

## Make example predictions

Okay, we've made some predictions on the test dataset, now's time to really test our model out.

To do so, we're going to get some data from the wild and see how our model performs.

In other words, were going to find an RCT abstract from PubMed, preprocess the text so it works with our model, then pass each sequence in the wild abstract through our model to see what label it predicts.

For an appropriate sample, we'll need to search PubMed for RCT's (randomized controlled trials) without abstracts which have been split up (on exploring PubMed you'll notice many of the abstracts are already preformatted into separate sections, this helps dramatically with readability).

Going through various PubMed studies, I managed to find the following unstructured abstract from [RCT of a manualized social treatment for high-functioning autism spectrum disorders](#):

This RCT examined the efficacy of a manualized social intervention for children with HFASDs. Participants were randomly assigned to treatment or wait-list conditions. Treatment included instruction and therapeutic activities targeting social skills, face-emotion recognition, interest expansion, and interpretation of non-literal language. A response-cost program was applied to reduce problem behaviors and foster skills acquisition. Significant treatment effects were found for five of seven primary outcome measures (parent ratings and direct child measures). Secondary measures based on staff ratings (treatment group only) corroborated gains reported by parents. High levels of parent, child and staff satisfaction were reported, along with high levels of treatment fidelity. Standardized effect size estimates were primarily in the medium and large ranges and favored the treatment group.

Looking at the large chunk of text can seem quite intimidating. Now imagine you're a medical researcher trying to skim through the literature to find a study relevant to your work.

Sounds like quite the challenge right?

## Enter SkimLit UI!

Let's see what our best model so far ( `model_5` ) makes of the above abstract.

But wait...

As you might've guessed the above abstract hasn't been formatted in the same structure as the data our model has been trained on. Therefore, before we can make a prediction on it, we need to preprocess it just as we have our other sequences.

More specifically, for each abstract, we'll need to:

1. Split it into sentences (lines).
2. Split it into characters.
3. Find the number of each line.
4. Find the total number of lines.

Starting with number 1, there are a couple of ways to split our abstracts into actual sentences. A simple one would be to use Python's in-built `split()` string method, splitting the abstract wherever a fullstop appears. However, can you imagine where this might go wrong?

Another more advanced option would be to leverage [spaCy's](#) (a very powerful NLP library) [sentencizer](#) class. Which is an easy to use sentence splitter based on spaCy's English language model.

I've prepared some abstracts from PubMed RCT papers to try our model on, we can download them [from GitHub](#).

In [ ]:

```
# Download and open example abstracts (copy and pasted from PubMed)
!wget https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/extras/skimlit_example_abstracts.json

with open("skimlit_example_abstracts.json", "r") as f:
    example_abstracts = json.load(f)

example_abstracts

--2021-08-25 00:08:37-- https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/extras/skimlit_example_abstracts.json
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.110.133, 185.199.108.133, 185.199.109.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.110.133|:443...
.. connected.
HTTP request sent, awaiting response... 200 OK
Length: 6737 (6.6K) [text/plain]
Saving to: 'skimlit_example_abstracts.json'

skimlit_example_abs 100%[=====] 6.58K --.-KB/s in 0s

2021-08-25 00:08:37 (82.5 MB/s) - 'skimlit_example_abstracts.json' saved [6737/6737]
```

```
-----
NameError Traceback (most recent call last)
<ipython-input-122-aa3c30151e9a> in <module>()
      3
      4     with open("skimlit_example_abstracts.json", "r") as f:
----> 5         example_abstracts = json.load(f)
      6
      7 example_abstracts
```

NameError: name 'json' is not defined

In [ ]:

```
# See what our example abstracts look like
abstracts = pd.DataFrame(example_abstracts)
abstracts
```

Now we've downloaded some example abstracts, let's see how one of them goes with our trained model.

First, we'll need to parse it using spaCy to turn it from a big chunk of text into sentences.

In [ ]:

```
# Create sentencizer - Source: https://spacy.io/usage/linguistic-features#sbd
from spacy.lang.en import English
nlp = English() # setup English sentence parser
sentencizer = nlp.create_pipe("sentencizer") # create sentence splitting pipeline object
nlp.add_pipe(sentencizer) # add sentence splitting pipeline object to sentence parser
doc = nlp(example_abstracts[0]["abstract"]) # create "doc" of parsed sequences, change index for a different abstract
abstract_lines = [str(sent) for sent in list(doc.sents)] # return detected sentences from doc in string type (not spaCy token type)
abstract_lines
```

Beautiful! It looks like spaCy has split the sentences in the abstract correctly. However, it should be noted, there may be more complex abstracts which don't get split perfectly into separate sentences (such as the example in [Baclofen promotes alcohol abstinence in alcohol dependent cirrhotic patients with hepatitis C virus \(HCV\) infection](#)), in this case, more custom splitting techniques would have to be investigated.

Now our abstract has been split into sentences, how about we write some code to count line numbers as well as total lines.

To do so, we can leverage some of the functionality of our `preprocess_text_with_line_numbers()` function.

In [ ]:

```
# Get total number of lines
total_lines_in_sample = len(abstract_lines)

# Go through each line in abstract and create a list of dictionaries containing features for each line
sample_lines = []
for i, line in enumerate(abstract_lines):
    sample_dict = {}
    sample_dict["text"] = str(line)
    sample_dict["line_number"] = i
    sample_dict["total_lines"] = total_lines_in_sample - 1
    sample_lines.append(sample_dict)
sample_lines
```

Now we've got "line\_number" and "total\_lines" values, we can one-hot encode them with `tf.one_hot` just like we did with our training dataset (using the same values for the `depth` parameter).

In [ ]:

```
# Get all line_number values from sample abstract
test_abstract_line_numbers = [line["line_number"] for line in sample_lines]
# One-hot encode to same depth as training data, so model accepts right input shape
test_abstract_line_numbers_one_hot = tf.one_hot(test_abstract_line_numbers, depth=15)
test_abstract_line_numbers_one_hot
```

In [ ]:

```
# Get all total_lines values from sample abstract
test_abstract_total_lines = [line["total_lines"] for line in sample_lines]
# One-hot encode to same depth as training data, so model accepts right input shape
test_abstract_total_lines_one_hot = tf.one_hot(test_abstract_total_lines, depth=20)
test_abstract_total_lines_one_hot
```

We can also use our `split_chars()` function to split our abstract lines into characters.

In [ ]:

```
# Split abstract lines into characters
abstract_chars = [split_chars(sentence) for sentence in abstract_lines]
abstract_chars
```

Alright, now we've preprocessed our wild RCT abstract into all of the same features our model was trained on, we can pass these features to our model and make sequence label predictions!

In [ ]:

```
# Make predictions on sample abstract features
%%time
test_abstract_pred_probs = loaded_model.predict(x=(test_abstract_line_numbers_one_hot,
                                                    test_abstract_total_lines_one_hot,
                                                    tf.constant(abstract_lines),
                                                    tf.constant(abstract_chars)))
test_abstract_pred_probs
```

In [ ]:

```
# Turn prediction probabilities into prediction classes
test_abstract_preds = tf.argmax(test_abstract_pred_probs, axis=1)
test_abstract_preds
```

Now we've got the predicted sequence label for each line in our sample abstract, let's write some code to visualize each sentence with its predicted label.

In [ ]:

```
# Turn prediction class integers into string class names
test_abstract_pred_classes = [label_encoder.classes_[i] for i in test_abstract_preds]
test_abstract_pred_classes
```

In [ ]:

```
# Visualize abstract lines and predicted sequence labels
for i, line in enumerate(abstract_lines):
    print(f'{test_abstract_pred_classes[i]}: {line}')
```

Nice! Isn't that much easier to read? I mean, it looks like our model's predictions could be improved, but how cool is that?

Imagine implementing our model to the backend of the PubMed website to format any unstructured RCT abstract on the site.

Or there could even be a browser extension, called "SkimLit" which would add structure (powered by our model) to any unstructured RCT abstract.

And if showed your medical researcher friend, and they thought the predictions weren't up to standard, there could be a button saying "is this label correct?... if not, what should it be?". That way the dataset, along with our model's future predictions, could be improved over time.

Of course, there are many more ways we could go to improve the model, the usability, the preprocessing functionality (e.g. functionizing our sample abstract preprocessing pipeline) but I'll leave these for the exercises/extensions.

QUESTION: How can we be sure the results of our test example from the wild are truly *wild*? Is there something we should check about the sample we're testing on?

## Exercises

1. Train `model_5` on all of the data in the training dataset for as many epochs until it stops improving. Since this might take a while, you might want to use:

- `tf.keras.callbacks.ModelCheckpoint` to save the model's best weights only.
- `tf.keras.callbacks.EarlyStopping` to stop the model from training once the validation loss has stopped improving for ~3 epochs.

2. Checkout the [Keras guide on using pretrained GloVe embeddings](#). Can you get this working with one of our models?

- Hint: You'll want to incorporate it with a custom token [Embedding](#) layer.
- It's up to you whether or not you fine-tune the GloVe embeddings or leave them frozen.

3. Try replacing the TensorFlow Hub Universal Sentence Encoder pretrained embedding for the [TensorFlow Hub BERT PubMed expert](#) (a language model pretrained on PubMed texts) pretrained embedding. Does this effect results?

- Note: Using the BERT PubMed expert pretrained embedding requires an extra preprocessing step for sequences (as detailed in the [TensorFlow Hub guide](#)).
- Does the BERT model beat the results mentioned in this paper? <https://arxiv.org/pdf/1710.06071.pdf>

4. What happens if you were to merge our `line_number` and `total_lines` features for each sequence? For example, created a `X_of_Y` feature instead? Does this effect model performance?

- Another example: `line_number=1` and `total_lines=11` turns into `line_of_X=1_of_11`.

5. Write a function (or series of functions) to take a sample abstract string, preprocess it (in the same way our model has been trained), make a prediction on each sequence in the abstract and return the abstract in the format:

- PREDICTED\_LABEL : SEQUENCE
- PREDICTED\_LABEL : SEQUENCE
- PREDICTED\_LABEL : SEQUENCE
- PREDICTED\_LABEL : SEQUENCE
- ...
- You can find your own unstructured RCT abstract from PubMed or try this one from: [Baclofen promotes alcohol abstinence in alcohol dependent cirrhotic patients with hepatitis C virus \(HCV\) infection](#).

## □ Extra-curriculum

- For more on working with text/spaCy, see [spaCy's advanced NLP course](#). If you're going to be working on production-level NLP problems, you'll probably end up using spaCy.
- For another look at how to approach a text classification problem like the one we've just gone through, I'd suggest going through [Google's Machine Learning Course for text classification](#).
- Since our dataset has imbalanced classes (as with many real-world datasets), so it might be worth looking into the [TensorFlow guide for different methods to training a model with imbalanced classes](#).

# 10. Milestone Project 3: Time series forecasting in TensorFlow (BitPredict 🤖)

The goal of this notebook is to get you familiar with working with time series data.

We're going to be building a series of models in an attempt to predict the price of Bitcoin.

Welcome to Milestone Project 3, BitPredict 🤖!

**Note:** ⚠️ This is not financial advice, as you'll see time series forecasting for stock market prices is actually quite terrible.

## What is a time series problem?

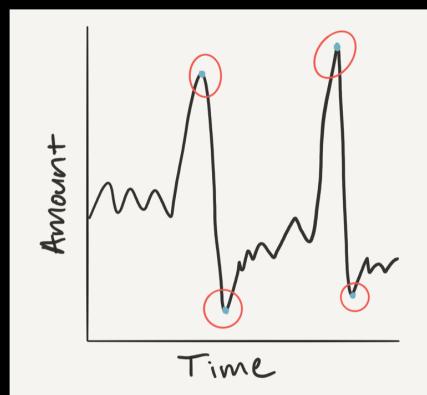
Time series problems deal with data over time.

Such as, the number of staff members in a company over 10-years, sales of computers for the past 5-years, electricity usage for the past 50-years.

The timeline can be short (seconds/minutes) or long (years/decades). And the problems you might investigate using can usually be broken down into two categories.

### Classification

“Which of these points is an anomaly?”



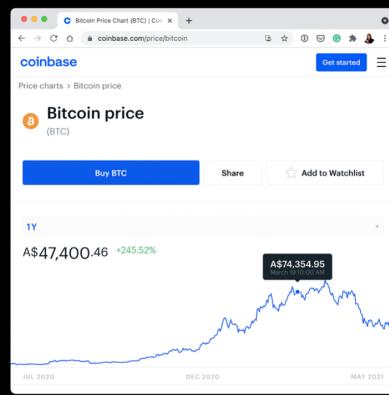
“What electronic device is this?”

“Are these heartbeats regular?”

**Output: discrete**

### Forecasting

“How much will the price of Bitcoin change tomorrow?”



“How many computers will we sell next year?”

“How many staff do we need for next week?”

**Output: continuous**

Problem Type	Examples	Output
Classification	Anomaly detection, time series identification (where did this time series come from?)	Discrete (a label)
Forecasting	Predicting stock market prices, forecasting future demand for a product, stocking inventory requirements	Continuous (a number)

In both cases above, a supervised learning approach is often used. Meaning, you'd have some example data and a label associated with that data.

For example, in forecasting the price of Bitcoin, your data could be the historical price of Bitcoin for the past month and the label could be today's price (the label can't be tomorrow's price because that's what we'd want to predict).

Can you guess what kind of problem BitPredict 🤖 is?

# What we're going to cover

Are you ready?

We've got a lot to go through.

- Get time series data (the historical price of Bitcoin)
  - Load in time series data using pandas/Python's CSV module
- Format data for a time series problem
  - Creating training and test sets (the wrong way)
  - Creating training and test sets (the right way)
  - Visualizing time series data
  - Turning time series data into a supervised learning problem (windowing)
  - Preparing univariate and multivariate (more than one variable) data
- Evaluating a time series forecasting model
- Setting up a series of deep learning modelling experiments
  - Dense (fully-connected) networks
  - Sequence models (LSTM and 1D CNN)
  - Ensembling (combining multiple models together)
  - Multivariate models
  - Replicating the N-BEATS algorithm using TensorFlow layer subclassing
- Creating a modelling checkpoint to save the best performing model during training
- Making predictions (forecasts) with a time series model
- Creating prediction intervals for time series model forecasts
- Discussing two different types of uncertainty in machine learning (data uncertainty and model uncertainty)
- Demonstrating why forecasting in an open system is BS (the turkey problem)

## How you can use this notebook

You can read through the descriptions and the code (it should all run), but there's a better option.

Write all of the code yourself.

Yes. I'm serious. Create a new notebook, and rewrite each line by yourself. Investigate it, see if you can break it, why does it break?

You don't have to write the text descriptions but writing the code yourself is a great way to get hands-on experience.

Don't worry if you make mistakes, we all do. The way to get better and make less mistakes is to [write more code](#).

 **Resource:** Get all of the materials you need for this notebook on the [course GitHub](#).

## Check for GPU

In order for our deep learning models to run as fast as possible, we'll need access to a GPU.

In Google Colab, you can set this up by going to Runtime -> Change runtime type -> Hardware accelerator -> GPU.

After selecting GPU, you may have to restart the runtime.

In [ ]:

```
# Check for GPU
!nvidia-smi -L
```

GPU 0: Tesla K80 (UUID: GPU-c7456639-4229-1150-8316-e4197bf2c93e)

# Get data

To build a time series forecasting model, the first thing we're going to need is data.

And since we're trying to predict the price of Bitcoin, we'll need Bitcoin data.

Specifically, we're going to get the prices of Bitcoin from 01 October 2013 to 18 May 2021.

Why these dates?

Because 01 October 2013 is when our data source ([Coindesk](#)) started recording the price of Bitcoin and 18 May 2021 is when this notebook was created.

If you're going through this notebook at a later date, you'll be able to use what you learn to predict on later dates of Bitcoin, you'll just have to adjust the data source.

Resource: To get the Bitcoin historical data, I went to the [Coindesk page for Bitcoin prices](#), clicked on "all" and then clicked on "Export data" and selected "CSV".

You can find the data we're going to use on [GitHub](#).

In [ ]:

```
# Download Bitcoin historical data from GitHub
# Note: you'll need to select "Raw" to download the data in the correct format
!wget https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/extras/BTC_USD_2013-10-01_2021-05-18-CoinDesk.csv

--2021-09-27 03:40:22-- https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/extras/BTC_USD_2013-10-01_2021-05-18-CoinDesk.csv
Resolving raw.githubusercontent.com (raw.githubusercontent.com) ... 185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443...
.. connected.
HTTP request sent, awaiting response... 200 OK
Length: 178509 (174K) [text/plain]
Saving to: 'BTC_USD_2013-10-01_2021-05-18-CoinDesk.csv'

BTC_USD_2013-10-01_ 100%[=====] 174.33K --.-KB/s   in 0.01s

2021-09-27 03:40:22 (16.5 MB/s) - 'BTC_USD_2013-10-01_2021-05-18-CoinDesk.csv' saved [178509/178509]
```

## Importing time series data with pandas

Now we've got some data to work with, let's import it using pandas so we can visualize it.

Because our data is in **CSV (comma separated values)** format (a very common data format for time series), we'll use the pandas `read_csv()` function.

And because our data has a date component, we'll tell pandas to parse the dates using the `parse_dates` parameter passing it the name of the date column ("Date").

In [ ]:

```
# Import with pandas
import pandas as pd
# Parse dates and set date column to index
df = pd.read_csv("/content/BTC_USD_2013-10-01_2021-05-18-CoinDesk.csv",
                 parse_dates=["Date"],
                 index_col=["Date"]) # parse the date column (tell pandas column 1 is a
# datetime
df.head()
```

Out [ ]:

Date	Currency	Closing Price (USD)	24h Open (USD)	24h High (USD)	24h Low (USD)
Date	Currency	Closing Price (USD)	24h Open (USD)	24h High (USD)	24h Low (USD)
2013-10-01	BTC	123.65499	124.30466	124.75166	122.56349
2013-10-02	BTC	125.45500	123.65499	125.75850	123.63383
2013-10-03	BTC	108.58483	125.45500	125.66566	83.32833
2013-10-04	BTC	118.67466	108.58483	118.67500	107.05816
2013-10-05	BTC	121.33866	118.67466	121.93633	118.00566

Looking good! Let's get some more info.

In [ ]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2787 entries, 2013-10-01 to 2021-05-18
Data columns (total 5 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   Currency        2787 non-null    object  
 1   Closing Price (USD) 2787 non-null    float64
 2   24h Open (USD)   2787 non-null    float64
 3   24h High (USD)  2787 non-null    float64
 4   24h Low (USD)   2787 non-null    float64
dtypes: float64(4), object(1)
memory usage: 130.6+ KB
```

Because we told pandas to parse the date column and set it as the index, its not in the list of columns.

You can also see there isn't many samples.

In [ ]:

```
# How many samples do we have?
len(df)
```

Out[ ]:

2787

We've collected the historical price of Bitcoin for the past ~8 years but there's only 2787 total samples.

This is something you'll run into with time series data problems. Often, the number of samples isn't as large as other kinds of data.

For example, collecting one sample at different time frames results in:

1 sample per timeframe	Number of samples per year	
	Second	31,536,000
Hour	8,760	
Day	365	
Week	52	
Month	12	

▀ Note: The frequency at which a time series value is collected is often referred to as **seasonality**. This is usually measured in number of samples per year. For example, collecting the price of Bitcoin once per day would result in a time series with a seasonality of 365. Time series data collected with different seasonality values often exhibit seasonal patterns (e.g. electricity demand being higher in Summer months for air conditioning than Winter months). For more on different time series patterns, see [Forecasting: Principles and Practice Chapter 2.3](#).

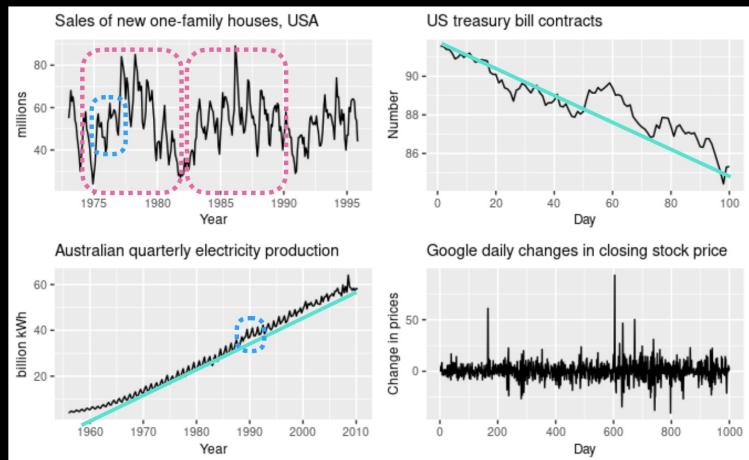
# Types of time series

(various patterns)

**Trend** — time series has a clear long-term increase or decrease (may or may not be linear)

**Seasonal** — time series affected by seasonal factors such as time of year (e.g. increased sales towards end of year) or day of week

**Cyclic** — time series shows rises and falls over an unfixed period, these tend to be longer/more variable than seasonal patterns



**Source:** Figure 2.3: Four examples of time series showing different patterns from [Forecasting: Principles and Practice 3rd Edition](#).

No patterns  
(random, likely not predictable)

*Example of different kinds of patterns you'll see in time series data. Notice the bottom right time series (Google stock price changes) has little to no patterns, making it difficult to predict. See [Forecasting: Principles and Practice Chapter 2.3](#) for full graphic.*

Deep learning algorithms usually flourish with lots of data, in the range of thousands to millions of samples.

In our case, we've got the daily prices of Bitcoin, a max of 365 samples per year.

But that doesn't we can't try them with our data.

To simplify, let's remove some of the columns from our data so we're only left with a date index and the closing price.

In [ ]:

```
# Only want closing price for each day
bitcoin_prices = pd.DataFrame(df["Closing Price (USD)"]).rename(columns={"Closing Price (USD)": "Price"})
bitcoin_prices.head()
```

Out [ ]:

Date	Price
2013-10-01	123.65499
2013-10-02	125.45500
2013-10-03	108.58483
2013-10-04	118.67466
2013-10-05	121.33866

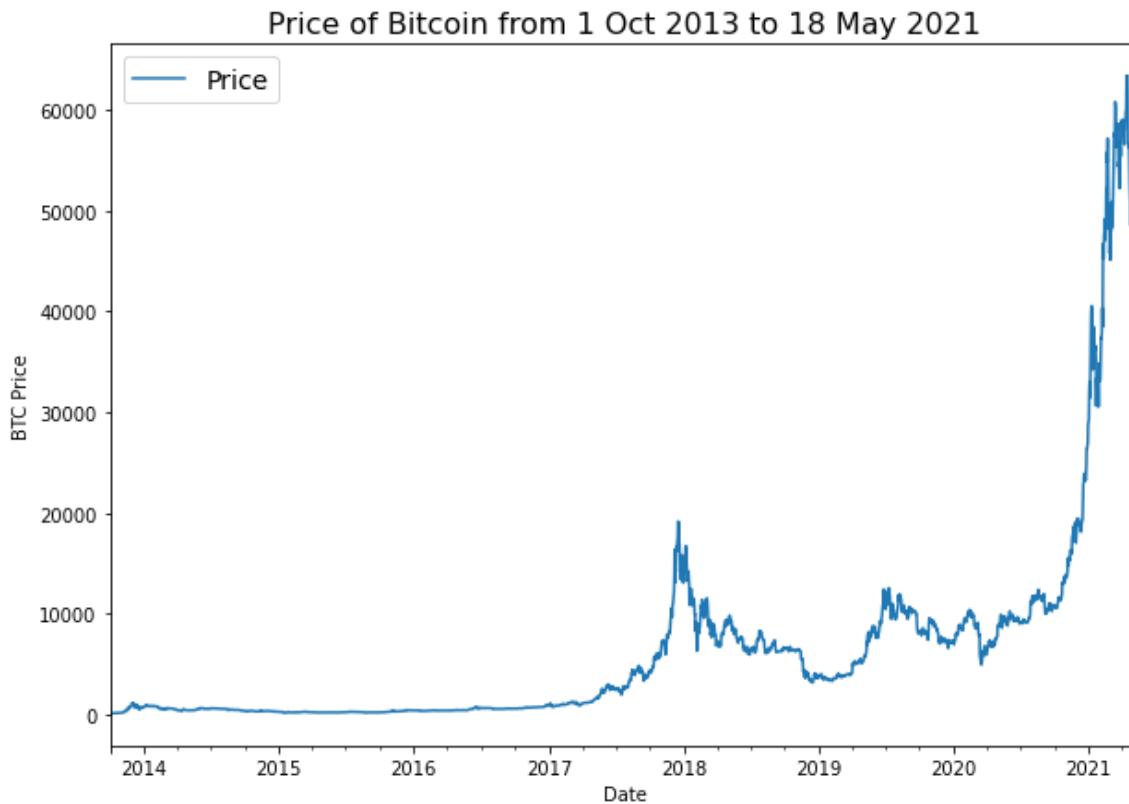
Much better!

But that's only five days worth of Bitcoin prices, let's plot everything we've got.

In [ ]:

```
import matplotlib.pyplot as plt
bitcoin_prices.plot(figsize=(10, 7))
plt.ylabel("BTC Price")
```

```
plt.title("Price of Bitcoin from 1 Oct 2013 to 18 May 2021", fontsize=16)
plt.legend(fontsize=14);
```



Woah, looks like it would've been a good idea to buy Bitcoin back in 2014.

## Importing time series data with Python's CSV module

If your time series data comes in CSV form you don't necessarily have to use pandas.

You can use Python's [in-built csv module](#). And if you're working with dates, you might also want to use Python's [datetime](#).

Let's see how we can replicate the plot we created before except this time using Python's `csv` and `datetime` modules.

**Resource:** For a great guide on using Python's `csv` module, check out Real Python's tutorial on [Reading and Writing CSV files in Python](#).

In [ ]:

```
# Importing and formatting historical Bitcoin data with Python
import csv
from datetime import datetime

timesteps = []
btc_price = []
with open("/content/BTC_USD_2013-10-01_2021-05-18-CoinDesk.csv", "r") as f:
    csv_reader = csv.reader(f, delimiter=",") # read in the target CSV
    next(csv_reader) # skip first line (this gets rid of the column titles)
    for line in csv_reader:
        timesteps.append(datetime.strptime(line[1], "%Y-%m-%d")) # get the dates as dates (not strings), strftime = string parse time
        btc_price.append(float(line[2])) # get the closing price as float

# View first 10 of each
timesteps[:10], btc_price[:10]
```

Out [ ]:

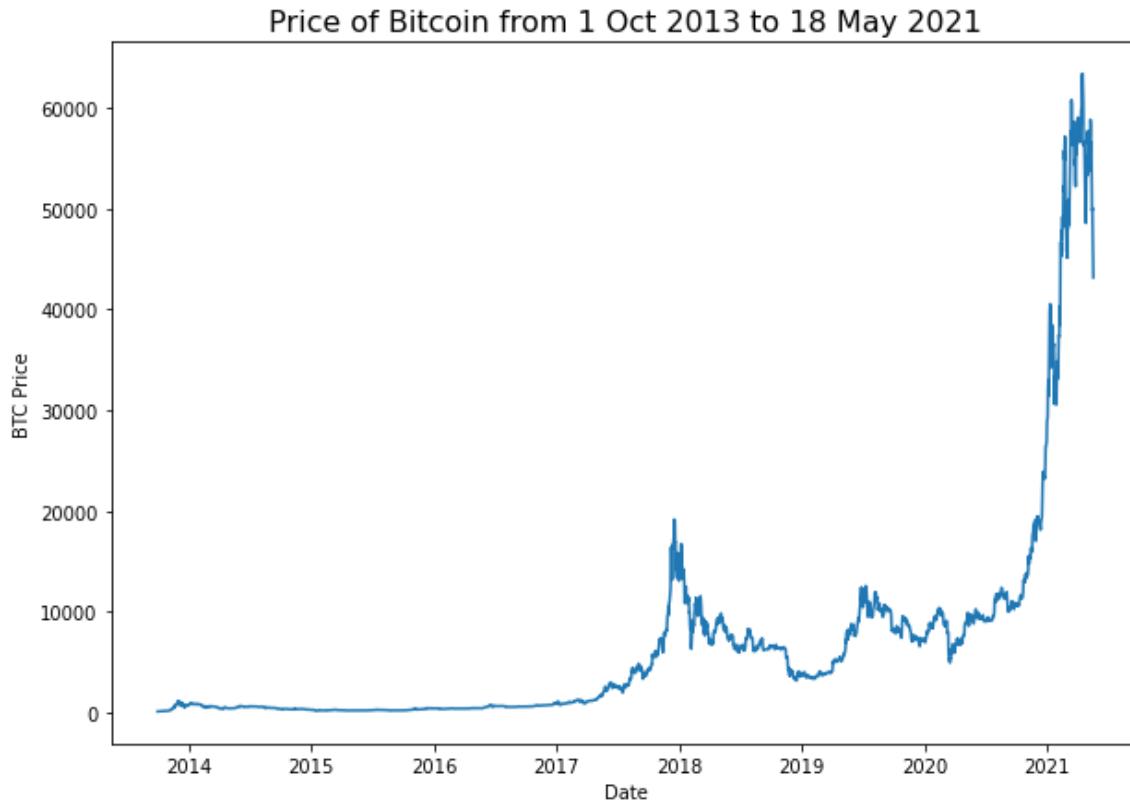
```
([datetime.datetime(2013, 10, 1, 0, 0),
```

```
datetime.datetime(2013, 10, 2, 0, 0),
datetime.datetime(2013, 10, 3, 0, 0),
datetime.datetime(2013, 10, 4, 0, 0),
datetime.datetime(2013, 10, 5, 0, 0),
datetime.datetime(2013, 10, 6, 0, 0),
datetime.datetime(2013, 10, 7, 0, 0),
datetime.datetime(2013, 10, 8, 0, 0),
datetime.datetime(2013, 10, 9, 0, 0),
datetime.datetime(2013, 10, 10, 0, 0)],
[123.65499,
125.455,
108.58483,
118.67466,
121.33866,
120.65533,
121.795,
123.033,
124.049,
125.96116])
```

**Beautiful! Now, let's see how things look.**

In [ ]:

```
# Plot from CSV
import matplotlib.pyplot as plt
import numpy as np
plt.figure(figsize=(10, 7))
plt.plot(timesteps, btc_price)
plt.title("Price of Bitcoin from 1 Oct 2013 to 18 May 2021", fontsize=16)
plt.xlabel("Date")
plt.ylabel("BTC Price");
```



**Ho ho! Would you look at that! Just like the pandas plot. And because we formatted the `timesteps` to be `datetime` objects, `matplotlib` displays a fantastic looking date axis.**

## Format Data Part 1: Creating train and test sets for time series data

Alrighty. What's next?

If you guessed preparing our data for a model, you'd be right.

What's the most important first step for preparing any machine learning dataset?

Scaling?

No...

Removing outliers?

No...

How about creating train and test splits?

Yes!

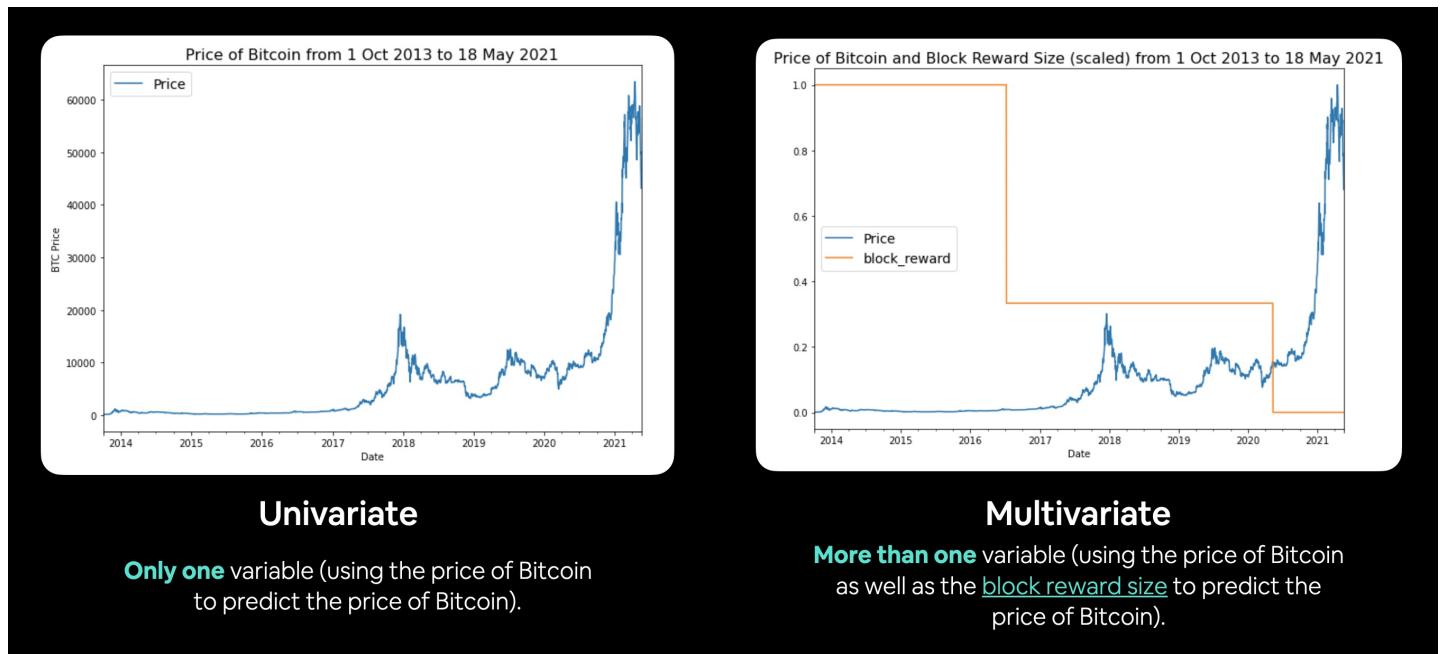
Usually, you could create a train and test split using a function like Scikit-Learn's outstanding `train_test_split()` but as we'll see in a moment, this doesn't really cut it for time series data.

But before we do create splits, it's worth talking about what *kind* of data we have.

In time series problems, you'll either have **univariate** or **multivariate** data.

Can you guess what our data is?

- **Univariate** time series data deals with *one* variable, for example, using the price of Bitcoin to predict the price of Bitcoin.
- **Multivariate** time series data deals with *more than one* variable, for example, predicting electricity demand using the day of week, time of year and number of houses in a region.



Example of univariate and multivariate time series data. Univariate involves using the target to predict the target. Multivariate involves using the target as well as another time series to predict the target.

## Create train & test sets for time series (the wrong way)

Okay, we've figured out we're dealing with a univariate time series, so we only have to make a split on one variable (for multivariate time series, you will have to split multiple variables).

How about we first see the *wrong way* for splitting time series data?

Let's turn our DataFrame index and column into NumPy arrays.

In [ ]:

```
# Get bitcoin date array
timesteps = bitcoin_prices.index.to_numpy()
prices = bitcoin_prices["Price"].to_numpy()
```

```
timesteps[:10], prices[:10]
```

Out [ ]:

```
(array(['2013-10-01T00:00:00.000000000', '2013-10-02T00:00:00.000000000',
       '2013-10-03T00:00:00.000000000', '2013-10-04T00:00:00.000000000',
       '2013-10-05T00:00:00.000000000', '2013-10-06T00:00:00.000000000',
       '2013-10-07T00:00:00.000000000', '2013-10-08T00:00:00.000000000',
       '2013-10-09T00:00:00.000000000', '2013-10-10T00:00:00.000000000'],
      dtype='datetime64[ns]'),
 array([123.65499, 125.455, 108.58483, 118.67466, 121.33866, 120.65533,
        121.795, 123.033, 124.049, 125.96116]))
```

And now we'll use the ever faithful `train_test_split` from Scikit-Learn to create our train and test sets.

In [ ]:

```
# Wrong way to make train/test sets for time series
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(timesteps, # dates
                                                    prices, # prices
                                                    test_size=0.2,
                                                    random_state=42)
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

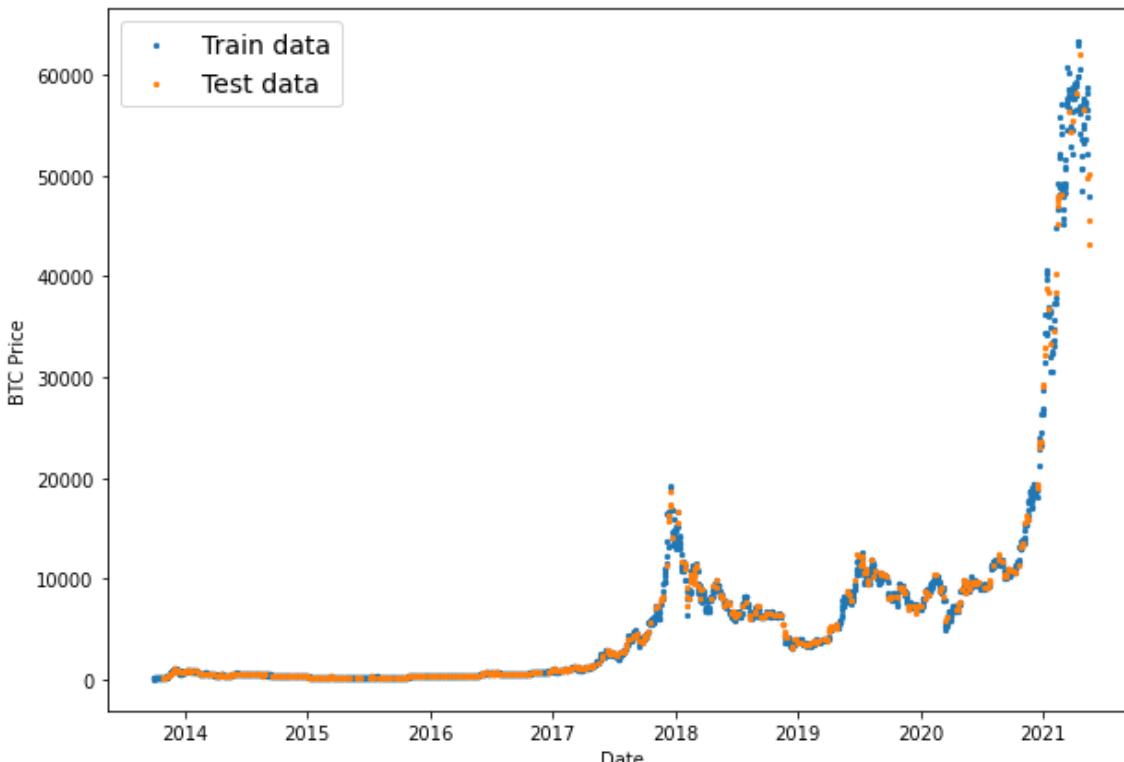
Out [ ]:

```
((2229,), (558,), (2229,), (558,))
```

Looks like the splits worked well, but let's not trust numbers on a page, let's visualize, visualize, visualize!

In [ ]:

```
# Let's plot wrong train and test splits
plt.figure(figsize=(10, 7))
plt.scatter(X_train, y_train, s=5, label="Train data")
plt.scatter(X_test, y_test, s=5, label="Test data")
plt.xlabel("Date")
plt.ylabel("BTC Price")
plt.legend(fontsize=14)
plt.show();
```



Hmmm... what's wrong with this plot?

Well, let's remind ourselves of what we're trying to do.

We're trying to use the historical price of Bitcoin to predict future prices of Bitcoin.

With this in mind, our seen data (training set) is what?

Prices of Bitcoin in the past.

And our unseen data (test set) is?

Prices of Bitcoin in the future.

Does the plot above reflect this?

No.

Our test data is scattered all throughout the training data.

This kind of random split is okay for datasets without a time component (such as images or passages of text for classification problems) but for time series, we've got to take the time factor into account.

To fix this, we've got to split our data in a way that reflects what we're actually trying to do.

We need to split our historical Bitcoin data to have a dataset that reflects the past (train set) and a dataset that reflects the future (test set).

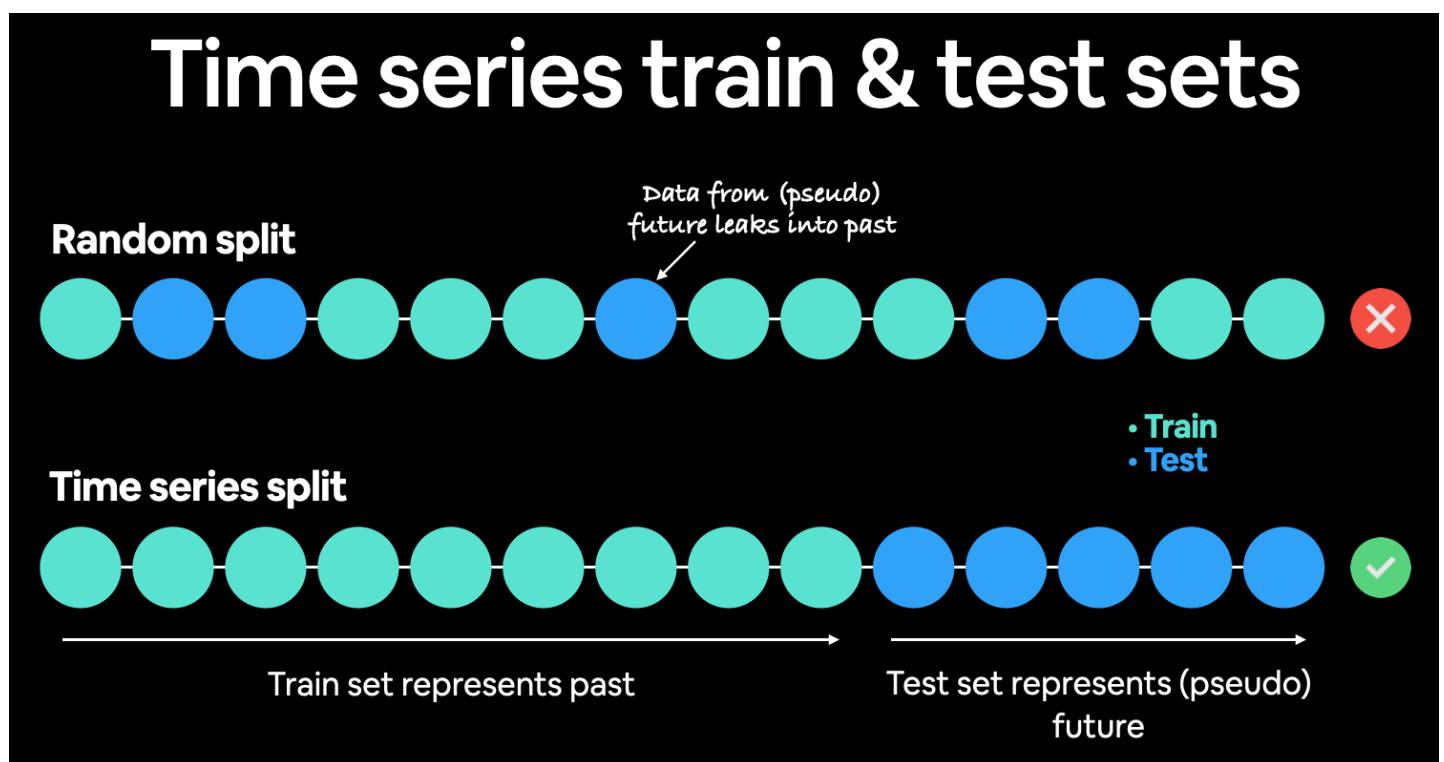
## Create train & test sets for time series (the right way)

Of course, there's no way we can actually access data from the future.

But we can engineer our test set to be in the future with respect to the training set.

To do this, we can create an arbitrary point in time to split our data.

Everything before the point in time can be considered the training set and everything after the point in time can be considered the test set.



*Demonstration of time series split. Rather than a traditionally random train/test split, it's best to split the time series data sequentially. Meaning, the test data should be data from the future when compared to the training data.*

In [ ]:

```
# Create train and test splits the right way for time series data
```

```

split_size = int(0.8 * len(prices)) # 80% train, 20% test

# Create train data splits (everything before the split)
X_train, y_train = timesteps[:split_size], prices[:split_size]

# Create test data splits (everything after the split)
X_test, y_test = timesteps[split_size:], prices[split_size:]

len(X_train), len(X_test), len(y_train), len(y_test)

```

Out [ ]:

(2229, 558, 2229, 558)

Okay, looks like our custom made splits are the same lengths as the splits we made with `train_test_split`.

But again, these are numbers on a page.

And you know how the saying goes, trust one eye more than two ears.

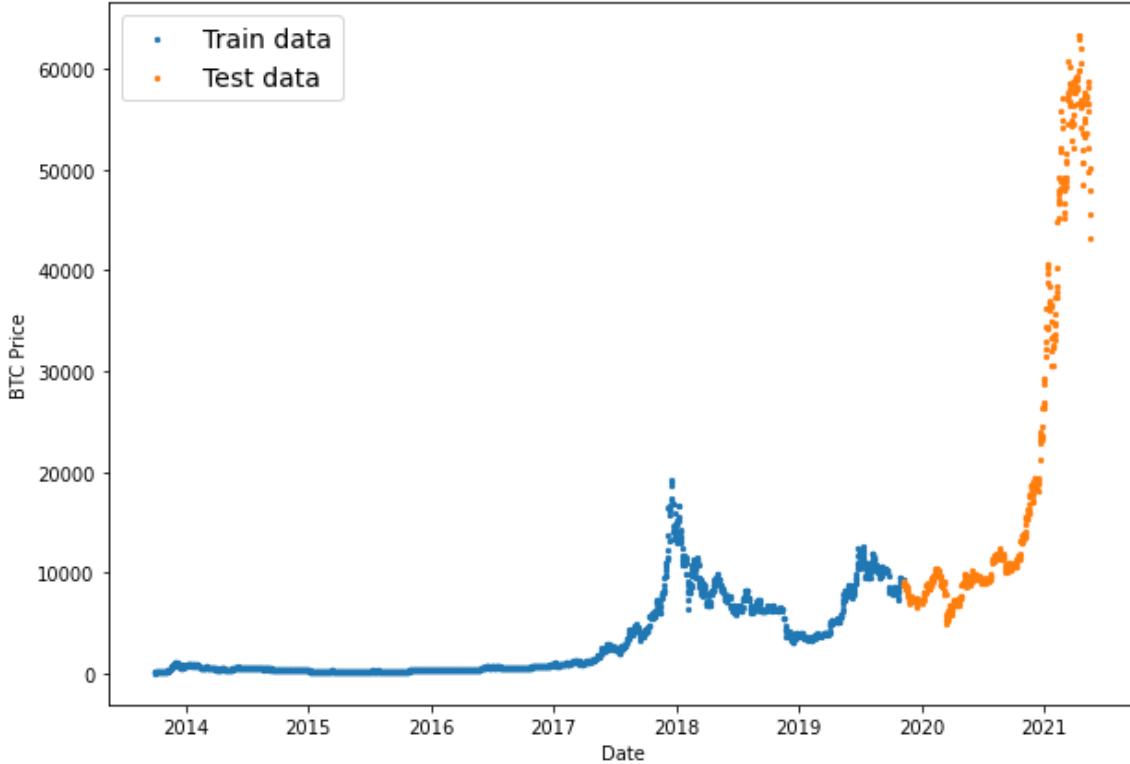
Let's visualize.

In [ ]:

```

# Plot correctly made splits
plt.figure(figsize=(10, 7))
plt.scatter(X_train, y_train, s=5, label="Train data")
plt.scatter(X_test, y_test, s=5, label="Test data")
plt.xlabel("Date")
plt.ylabel("BTC Price")
plt.legend(fontsize=14)
plt.show();

```



That looks much better!

Do you see what's happened here?

We're going to be using the training set (past) to train a model to try and predict values on the test set (future).

Because the test set is an *artificial* future, we can gauge how our model might perform on *actual*/future data.

**Note:** The amount of data you reserve for your test set is not set in stone. You could have 80/20, 90/10, 95/5 splits or in some cases, you might not even have enough data to split into train and test sets (see the resource below). The point is to remember the test set is a pseudofuture and not

the actual future, it is only meant to give you an indication of how the models you're building are performing.

Resource: Working with time series data can be tricky compared to other kinds of data. And there are a few pitfalls to watch out for, such as how much data to use for a test set. The article [3 facts about time series forecasting that surprise experienced machine learning practitioners](#) talks about different things to watch out for when working with time series data, I'd recommend reading it.

## Create a plotting function

Rather than retyping `matplotlib` commands to continuously plot data, let's make a plotting function we can reuse later.

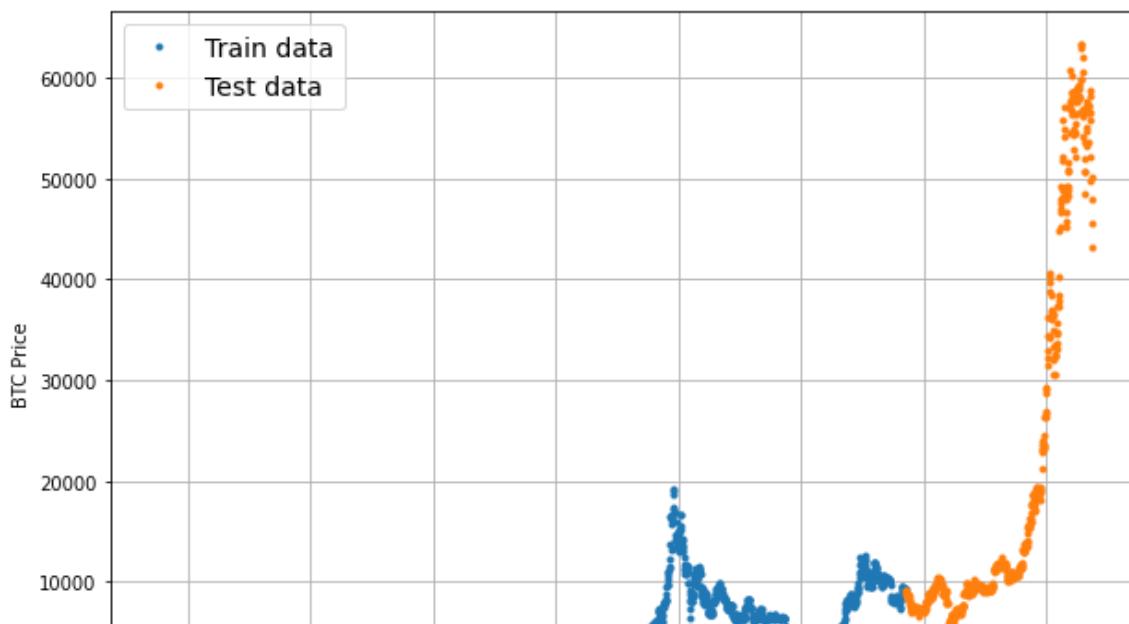
In [ ]:

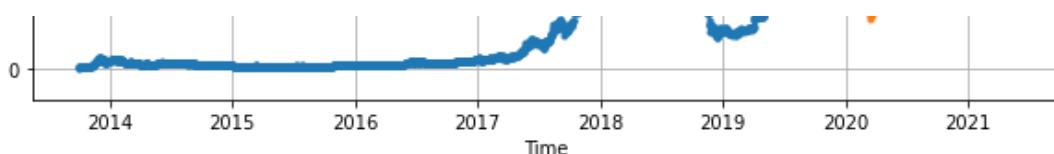
```
# Create a function to plot time series data
def plot_time_series(timesteps, values, format='.', start=0, end=None, label=None):
    """
    Plots a timesteps (a series of points in time) against values (a series of values across
    timesteps).

    Parameters
    -----
    timesteps : array of timesteps
    values : array of values across time
    format : style of plot, default "."
    start : where to start the plot (setting a value will index from start of timesteps & values)
    end : where to end the plot (setting a value will index from end of timesteps & values)
    label : label to show on plot of values
    """
    # Plot the series
    plt.plot(timesteps[start:end], values[start:end], format, label=label)
    plt.xlabel("Time")
    plt.ylabel("BTC Price")
    if label:
        plt.legend(fontsize=14) # make label bigger
    plt.grid(True)
```

In [ ]:

```
# Try out our plotting function
plt.figure(figsize=(10, 7))
plot_time_series(timesteps=X_train, values=y_train, label="Train data")
plot_time_series(timesteps=X_test, values=y_test, label="Test data")
```





Looking good!

Time for some modelling experiments.

## Modelling Experiments

We can build almost any kind of model for our problem as long as the data inputs and outputs are formatted correctly.

However, just because we *can* build *almost any* kind of model, doesn't mean it'll perform well/should be used in a production setting.

We'll see what this means as we build and evaluate models throughout.

Before we discuss what modelling experiments we're going to run, there are two terms you should be familiar with, **horizon** and **window**.

- **horizon** = number of timesteps to predict into future
- **window** = number of timesteps from past used to predict **horizon**

For example, if we wanted to predict the price of Bitcoin for tomorrow (1 day in the future) using the previous week's worth of Bitcoin prices (7 days in the past), the horizon would be 1 and the window would be 7.

Now, how about those modelling experiments?

Model Number	Model Type	Horizon size	Window size	Extra data
0	Naïve model (baseline)	NA	NA	NA
1	Dense model	1	7	NA
2	Same as 1	1	30	NA
3	Same as 1	7	30	NA
4	Conv1D	1	7	NA
5	LSTM	1	7	NA
6	Same as 1 (but with multivariate data)	1	7	Block reward size
7	<a href="#">N-BEATs Algorithm</a>	1	7	NA
8	Ensemble (multiple models optimized on different loss functions)	1	7	NA
9	Future prediction model (model to predict future values)	1	7	NA
10	Same as 1 (but with turkey 🦃 data introduced)	1	7	NA

💡 **Note:** To reiterate, as you can see, we can build many types of models for the data we're working with. But that doesn't mean that they'll perform well. Deep learning is a powerful technique but it doesn't always work. And as always, start with a simple model first and then add complexity as needed.

## Model 0: Naïve forecast (baseline)

As usual, let's start with a baseline.

One of the most common baseline models for time series forecasting, the naïve model (also called the [naïve forecast](#)), requires no training at all.

That's because all the naïve model does is use the previous timestep value to predict the next timestep value.

The formula looks like this:

$$\hat{y}_t = y_{t-1}$$

In English:

The prediction at timestep  $t$  ( $\hat{y}_t$ ) is equal to the value at timestep  $t-1$  (the previous timestep).

Sound simple?

Maybe not.

In an open system (like a stock market or crypto market), you'll often find beating the naïve forecast with *any* kind of model is quite hard.

▀ Note: For the sake of this notebook, an **open system** is a system where inputs and outputs can freely flow, such as a market (stock or crypto). Whereas, a **closed system** the inputs and outputs are contained within the system (like a poker game with your buddies, you know the buy in and you know how much the winner can get). Time series forecasting in **open systems** is generally quite poor.

In [ ]:

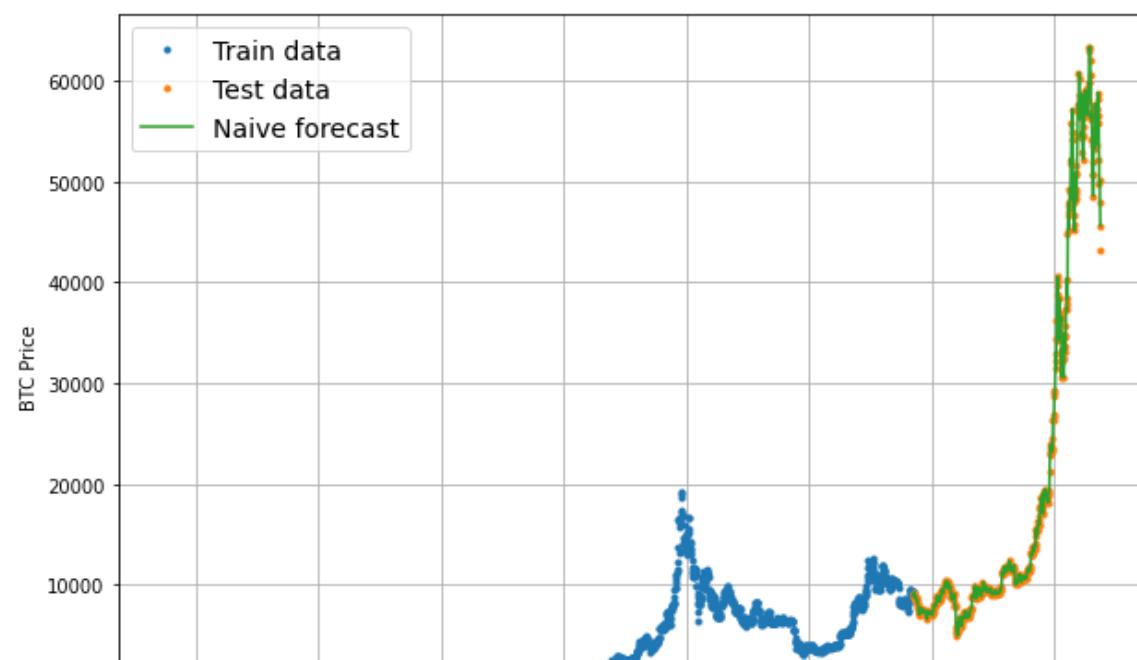
```
# Create a naïve forecast
naive_forecast = y_test[:-1] # Naïve forecast equals every value excluding the last value
naive_forecast[:10], naive_forecast[-10:] # View first 10 and last 10
```

Out[ ]:

```
(array([9226.48582088, 8794.35864452, 8798.04205463, 9081.18687849,
       8711.53433917, 8760.89271814, 8749.52059102, 8656.97092235,
       8500.64355816, 8469.2608989 ]),
 array([57107.12067189, 58788.20967893, 58102.19142623, 55715.54665129,
       56573.5554719 , 52147.82118698, 49764.1320816 , 50032.69313676,
       47885.62525472, 45604.61575361]))
```

In [ ]:

```
# Plot naïve forecast
plt.figure(figsize=(10, 7))
plot_time_series(timesteps=X_train, values=y_train, label="Train data")
plot_time_series(timesteps=X_test, values=y_test, label="Test data")
plot_time_series(timesteps=X_test[1:], values=naive_forecast, format="--", label="Naïve forecast");
```





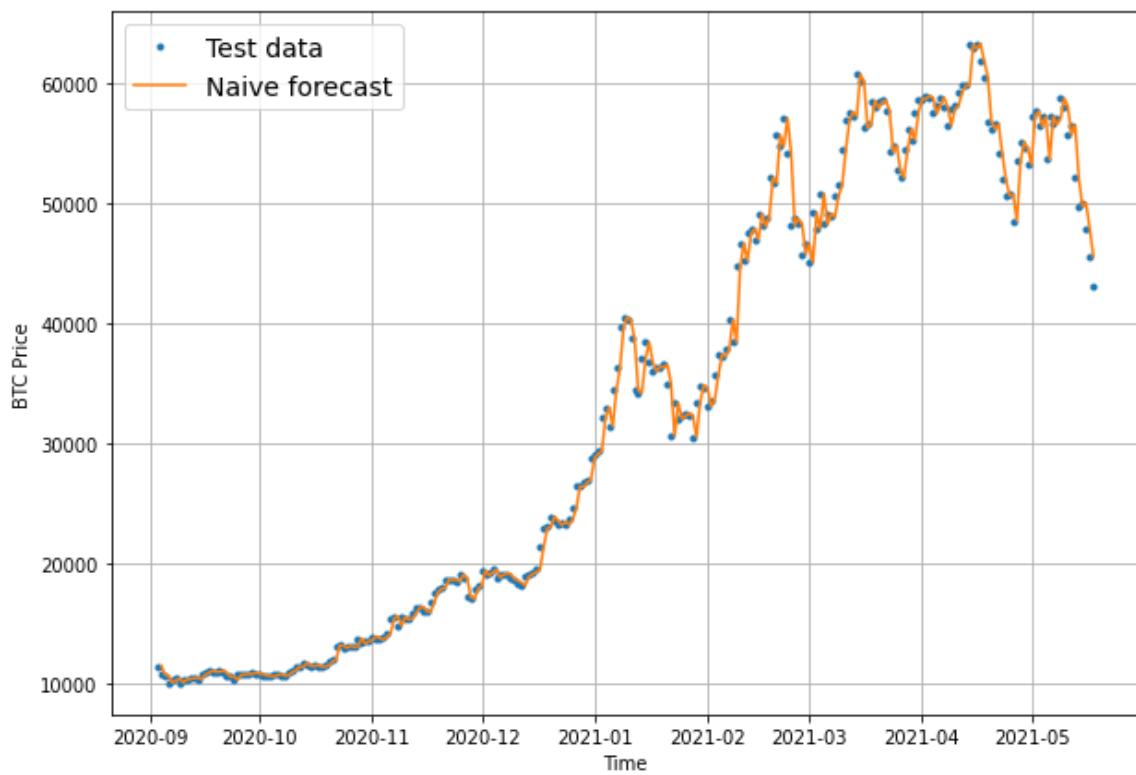
The naive forecast looks like it's following the data well.

Let's zoom in to take a better look.

We can do so by creating an offset value and passing it to the `start` parameter of our `plot_time_series()` function.

In [ ]:

```
plt.figure(figsize=(10, 7))
offset = 300 # offset the values by 300 timesteps
plot_time_series(timesteps=X_test, values=y_test, start=offset, label="Test data")
plot_time_series(timesteps=X_test[1:], values=naive_forecast, format="-", start=offset,
label="Naive forecast");
```



When we zoom in we see the naïve forecast comes slightly after the test data. This makes sense because the naive forecast uses the previous timestep value to predict the next timestep value.

Forecast made. Time to evaluate it.

## Evaluating a time series model

Time series forecasting often involves predicting a number (in our case, the price of Bitcoin).

And what kind of problem is predicting a number?

Ten points if you said regression.

With this known, we can use regression evaluation metrics to evaluate our time series forecasts.

The main thing we will be evaluating is: how do our model's predictions (`y_pred`) compare against the actual values (`y_true` or *ground truth values*)?

Resource: We're going to be using several metrics to evaluate our different model's time series forecast accuracy. Many of them are sourced and explained mathematically and conceptually in [Forecasting: Principles and Practice chapter 5.8](#), I'd recommend reading through here for a more

in-depth overview of what we're going to practice.

For all of the following metrics, **lower is better** (for example an MAE of 0 is better than an MAE 100).

## Scale-dependent errors

These are metrics which can be used to compare time series values and forecasts that are on the same scale.

For example, Bitcoin historical prices in USD versus Bitcoin forecast values in USD.

Metric	Details	Code
<b>MAE</b> (mean absolute error)	Easy to interpret (a forecast is X amount different from actual amount). Forecast methods which minimises the MAE will lead to forecasts of the median.	<code>tf.keras.metrics.mean_absolute_error()</code>
<b>RMSE</b> (root mean square error)	Forecasts which minimise the RMSE lead to forecasts of the mean.	<code>tf.sqrt( tf.keras.metrics.mean_squared_error() )</code>

## Percentage errors

Percentage errors do not have units, this means they can be used to compare forecasts across different datasets.

Metric	Details	Code
<b>MAPE</b> (mean absolute percentage error)	Most commonly used percentage error. May explode (not work) if $y=0$ .	<code>tf.keras.metrics.mean_absolute_percentage_error()</code>
<b>sMAPE</b> (symmetric mean absolute percentage error)	Recommended not to be used by <a href="#">Forecasting: Principles and Practice</a> , though it is used in forecasting competitions.	Custom implementation

## Scaled errors

Scaled errors are an alternative to percentage errors when comparing forecast performance across different time series.

Metric	Details	Code
<b>MASE</b> (mean absolute scaled error).	MASE equals one for the naive forecast (or very close to one). A forecast which performs better than the naïve should get <1 MASE.	See <a href="#">sktime's mase_loss()</a>

Question: There are so many metrics... which one should I pay most attention to? It's going to depend on your problem. However, since its ease of interpretation (you can explain it in a sentence to your grandma), MAE is often a very good place to start.

Since we're going to be evaluating a lot of models, let's write a function to help us calculate evaluation metrics on their forecasts.

First we'll need TensorFlow.

In [ ]:

```
# Let's get TensorFlow!
import tensorflow as tf
```

And since TensorFlow doesn't have a ready made version of MASE (mean absolute scaled error), how about we create our own?

We'll take inspiration from [sktime's \(Scikit-Learn for time series\)](#) `MeanAbsoluteScaledError` class which calculates the MASE.

In [ ]:

```
# MASE implemented courtesy of sktime - https://github.com/alan-turing-institute/sktime/blob/ee7a06843a44f4aaec7582d847e36073a9ab0566/sktime/performance_metrics/forecasting/_functions.py#L16
def mean_absolute_scaled_error(y_true, y_pred):
    """
    Implement MASE (assuming no seasonality of data).
    """
    mae = tf.reduce_mean(tf.abs(y_true - y_pred))

    # Find MAE of naive forecast (no seasonality)
    mae_naive_no_season = tf.reduce_mean(tf.abs(y_true[1:] - y_true[:-1])) # our seasonality is 1 day (hence the shifting of 1 day)

    return mae / mae_naive_no_season
```

You'll notice the version of MASE above doesn't take in the training values like sktime's `mae_loss()`. In our case, we're comparing the MAE of our predictions on the test to the MAE of the naïve forecast on the test set.

In practice, if we've created the function correctly, the naïve model should achieve an MASE of 1 (or very close to 1). Any model worse than the naïve forecast will achieve an MASE of >1 and any model better than the naïve forecast will achieve an MASE of <1.

Let's put each of our different evaluation metrics together into a function.

In [ ]:

```
def evaluate_preds(y_true, y_pred):
    # Make sure float32 (for metric calculations)
    y_true = tf.cast(y_true, dtype=tf.float32)
    y_pred = tf.cast(y_pred, dtype=tf.float32)

    # Calculate various metrics
    mae = tf.keras.metrics.mean_absolute_error(y_true, y_pred)
    mse = tf.keras.metrics.mean_squared_error(y_true, y_pred) # puts and emphasis on outliers (all errors get squared)
    rmse = tf.sqrt(mse)
    mape = tf.keras.metrics.mean_absolute_percentage_error(y_true, y_pred)
    mase = mean_absolute_scaled_error(y_true, y_pred)

    return {"mae": mae.numpy(),
            "mse": mse.numpy(),
            "rmse": rmse.numpy(),
            "mape": mape.numpy(),
            "mase": mase.numpy()}
```

Looking good! How about we test our function on the naive forecast?

In [ ]:

```
naive_results = evaluate_preds(y_true=y_test[1:],
                               y_pred=naive_forecast)
naive_results
```

Out [ ]:

```
{'mae': 567.9802,
 'mape': 2.516525,
 'mase': 0.99957,
 'mse': 1147547.0,
 'rmse': 1071.2362}
```

Alright, looks like we've got some baselines to beat.

Taking a look at the naïve forecast's MAE, it seems on average each forecast is ~\$567 different than the actual Bitcoin price.

How does this compare to the average price of Bitcoin in the test dataset?

In [ ]:

```
# Find average price of Bitcoin in test dataset
tf.reduce_mean(y_test).numpy()
```

Out[ ]:

20056.632963737226

Okay, looking at these two values is starting to give us an idea of how our model is performing:

- The average price of Bitcoin in the test dataset is: \$20,056 (note: average may not be the best measure here, since the highest price is over 3x this value and the lowest price is over 4x lower)
- Each prediction in naive forecast is on average off by: \$567

Is this enough to say it's a good model?

That's up your own interpretation. Personally, I'd prefer a model which was closer to the mark.

How about we try and build one?

## Other kinds of time series forecasting models which can be used for baselines and actual forecasts

Since we've got a naïve forecast baseline to work with, it's time we start building models to try and beat it.

And because this course is focused on TensorFlow and deep learning, we're going to be using TensorFlow to build deep learning models to try and improve on our naïve forecasting results.

That being said, there are many other kinds of models you may want to look into for building baselines/performing forecasts.

Some of them may even beat our best performing models in this notebook, however, I'll leave trying them out for extra-curriculum.

Model/Library Name	Resource
Moving average	<a href="https://machinelearningmastery.com/moving-average-smoothing-for-time-series-forecasting-python/">https://machinelearningmastery.com/moving-average-smoothing-for-time-series-forecasting-python/</a>
ARIMA (Autoregression Integrated Moving Average)	<a href="https://machinelearningmastery.com/arima-for-time-series-forecasting-with-python/">https://machinelearningmastery.com/arima-for-time-series-forecasting-with-python/</a>
sktime (Scikit-Learn for time series)	<a href="https://github.com/alain-turing-institute/sktime">https://github.com/alain-turing-institute/sktime</a>
TensorFlow Decision Forests (random forest, gradient boosting trees)	<a href="https://www.tensorflow.org/decision_forests">https://www.tensorflow.org/decision_forests</a>
Facebook Kats (purpose-built forecasting and time series analysis library by Facebook)	<a href="https://github.com/facebookresearch/Kats">https://github.com/facebookresearch/Kats</a>
LinkedIn Greykite (flexible, intuitive and fast forecasts)	<a href="https://github.com/linkedin/greykite">https://github.com/linkedin/greykite</a>

## Format Data Part 2: Windowing dataset

Surely we'd be ready to start building models by now?

We're so close! Only one more step (really two) to go.

We've got to window our time series.

Why do we window?

Windowing is a method to turn a time series dataset into **supervised learning problem**.

In other words, we want to use windows of the past to predict the future.

For example for a univariate time series, windowing for one week (`window=7`) to predict the next single value

(`horizon=1`) might look like:

Window for one week (univariate time series)

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

Or for the price of Bitcoin, it'd look like:

Window for one week with the target of predicting the next day (Bitcoin prices)

```
[123.654, 125.455, 108.584, 118.674, 121.338, 120.655, 121.795] -> [123.033]  
[125.455, 108.584, 118.674, 121.338, 120.655, 121.795, 123.033] -> [124.049]  
[108.584, 118.674, 121.338, 120.655, 121.795, 123.033, 124.049] -> [125.961]
```

```
# Window for one week with the target of predicting the next day (Bitcoin prices)  
[123.654, 125.455, 108.584, 118.674, 121.338, 120.655, 121.795] -> [123.033]  
[125.455, 108.584, 118.674, 121.338, 120.655, 121.795, 123.033] -> [124.049]  
[108.584, 118.674, 121.338, 120.655, 121.795, 123.033, 124.049] -> [125.961]
```

Window size = 7      Horizon = 1

(These values are tuneable hyperparameters)

“Can we predict the next \_\_\_\_\_ given the past \_\_\_\_\_?”

**Window size (input) —** number of time steps of historical data used to predict horizon      **Data**

**Horizon (output) —** number of time steps to predict into the future      **Label**

*Example of windows and horizons for Bitcoin data. Windowing can be used to turn time series data into a supervised learning problem.*

Let's build some functions which take in a univariate time series and turn it into windows and horizons of specified sizes.

We'll start with the default horizon size of 1 and a window size of 7 (these aren't necessarily the best values to use, I've just picked them).

In [ ]:

```
HORIZON = 1 # predict 1 step at a time  
WINDOW_SIZE = 7 # use a week worth of timesteps to predict the horizon
```

Now we'll write a function to take in an array and turn it into a window and horizon.

In [ ]:

```
# Create function to label windowed data  
def get_labelled_windows(x, horizon=1):  
    """  
    Creates labels for windowed dataset.  
  
    E.g. if horizon=1 (default)  
    Input: [1, 2, 3, 4, 5, 6] -> Output: ([1, 2, 3, 4, 5], [6])  
    """  
    return x[:, :-horizon], x[:, -horizon:]
```

In [ ]:

```
# Test out the window labelling function
test_window, test_label = get_labelled_windows(tf.expand_dims(tf.range(8)+1, axis=0), horizon=HORIZON)
print(f"Window: {tf.squeeze(test_window).numpy()} -> Label: {tf.squeeze(test_label).numpy()}"
```

Window: [1 2 3 4 5 6 7] -> Label: 8

**Oh yeah, that's what I'm talking about!**

**Now we need a way to make windows for an entire time series.**

**We could do this with Python for loops, however, for large time series, that'd be quite slow.**

**To speed things up, we'll leverage [NumPy's array indexing](#).**

**Let's write a function which:**

**1. Creates a window step of specific window size, for example: [[0, 1, 2, 3, 4, 5, 6, 7]]**

**2. Uses NumPy indexing to create a 2D of multiple window steps, for example:**

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

**3. Uses the 2D array of multiple window steps to index on a target series**

**4. Uses the `get_labelled_windows()` function we created above to turn the window steps into windows with a specified horizon**

**Resource: The function created below has been adapted from Syafiq Kamarul Azman's article [Fast and Robust Sliding Window Vectorization with NumPy](#).**

In [ ]:

```
# Create function to view NumPy arrays as windows
def make_windows(x, window_size=7, horizon=1):
    """
    Turns a 1D array into a 2D array of sequential windows of window_size.
    """
    # 1. Create a window of specific window_size (add the horizon on the end for later labelling)
    window_step = np.expand_dims(np.arange(window_size+horizon), axis=0)
    # print(f"Window step:\n {window_step}")

    # 2. Create a 2D array of multiple window steps (minus 1 to account for 0 indexing)
    window_indexes = window_step + np.expand_dims(np.arange(len(x)-(window_size+horizon-1)), axis=0).T # create 2D array of windows of size window_size
    # print(f"Window indexes:\n {window_indexes[:3]}, {window_indexes[-3:]}, {window_indexes.shape}")

    # 3. Index on the target array (time series) with 2D array of multiple window steps
    windowed_array = x[window_indexes]

    # 4. Get the labelled windows
    windows, labels = get_labelled_windows(windowed_array, horizon=horizon)

    return windows, labels
```

**Phew! A few steps there... let's see how it goes.**

In [ ]:

```
full_windows, full_labels = make_windows(prices, window_size=WINDOW_SIZE, horizon=HORIZON)
len(full_windows), len(full_labels)
```

Out [ ]:

```
(2780, 2780)
```

Of course we have to visualize, visualize, visualize!

In [ ]:

```
# View the first 3 windows/labels
for i in range(3):
    print(f"Window: {full_windows[i]} -> Label: {full_labels[i]}")
```

Window: [123.65499 125.455 108.58483 118.67466 121.33866 120.65533 121.795] -> Label: [123.033]  
Window: [125.455 108.58483 118.67466 121.33866 120.65533 121.795 123.033] -> Label: [124.049]  
Window: [108.58483 118.67466 121.33866 120.65533 121.795 123.033 124.049] -> Label: [125.96116]

In [ ]:

```
# View the last 3 windows/labels
for i in range(3):
    print(f"Window: {full_windows[i-3]} -> Label: {full_labels[i-3]}")
```

Window: [58788.20967893 58102.19142623 55715.54665129 56573.5554719 52147.82118698 49764.1320816 50032.69313676] -> Label: [47885.62525472]  
Window: [58102.19142623 55715.54665129 56573.5554719 52147.82118698 49764.1320816 50032.69313676 47885.62525472] -> Label: [45604.61575361]  
Window: [55715.54665129 56573.5554719 52147.82118698 49764.1320816 50032.69313676 47885.62525472 45604.61575361] -> Label: [43144.47129086]

▀ Note: You can find a function which achieves similar results to the ones we implemented above at [tf.keras.preprocessing.timeseries\\_dataset\\_from\\_array\(\)](#). Just like ours, it takes in an array and returns a windowed dataset. It has the benefit of returning data in the form of a `tf.data.Dataset` instance (we'll see how to do this with our own data later).

## Turning windows into training and test sets

Look how good those windows look! Almost like the stain glass windows on the Sistine Chapel, well, maybe not that good but still.

Time to turn our windows into training and test splits.

We could've windowed our existing training and test splits, however, with the nature of windowing (windowing often requires an offset at some point in the data), it usually works better to window the data first, then split it into training and test sets.

Let's write a function which takes in full sets of windows and their labels and splits them into train and test splits.

In [ ]:

```
# Make the train/test splits
def make_train_test_splits(windows, labels, test_split=0.2):
    """
    Splits matching pairs of windows and labels into train and test splits.
    """
    split_size = int(len(windows) * (1-test_split)) # this will default to 80% train/20% test
    train_windows = windows[:split_size]
    train_labels = labels[:split_size]
    test_windows = windows[split_size:]
    test_labels = labels[split_size:]
    return train_windows, test_windows, train_labels, test_labels
```

**Look at that amazing function, lets test it.**

In [ ]:

```
train_windows, test_windows, train_labels, test_labels = make_train_test_splits(full_windows, full_labels)
len(train_windows), len(test_windows), len(train_labels), len(test_labels)
```

Out [ ]:

```
(2224, 556, 2224, 556)
```

**Notice the default split of 80% training data and 20% testing data (this split can be adjusted if needed).**

**How do the first 5 samples of the training windows and labels looks?**

In [ ]:

```
train_windows[:5], train_labels[:5]
```

Out [ ]:

```
(array([[123.65499, 125.455 , 108.58483, 118.67466, 121.33866, 120.65533,
       121.795 ],
       [125.455 , 108.58483, 118.67466, 121.33866, 120.65533, 121.795 ,
       123.033 ],
       [108.58483, 118.67466, 121.33866, 120.65533, 121.795 , 123.033 ,
       124.049 ],
       [118.67466, 121.33866, 120.65533, 121.795 , 123.033 , 124.049 ,
       125.96116],
       [121.33866, 120.65533, 121.795 , 123.033 , 124.049 , 125.96116,
       125.27966]], array([[123.033 ],
       [124.049 ],
       [125.96116],
       [125.27966],
       [125.9275 ]]))
```

In [ ]:

```
# Check to see if same (accounting for horizon and window size)
np.array_equal(np.squeeze(train_labels[:-HORIZON-1]), y_train[WINDOW_SIZE:])
```

Out [ ]:

```
True
```

## Make a modelling checkpoint

We're so close to building models. So so so close.

Because our model's performance will fluctuate from experiment to experiment, we'll want to make sure we're comparing apples to apples.

What I mean by this is in order for a fair comparison, we want to compare each model's best performance against each model's best performance.

For example, if `model_1` performed incredibly well on epoch 55 but its performance fell off toward epoch 100, we want the version of the model from epoch 55 to compare to other models rather than the version of the model from epoch 100.

And the same goes for each of our other models: compare the best against the best.

To take of this, we'll implement a `ModelCheckpoint` callback.

The `ModelCheckpoint` callback will monitor our model's performance during training and save the best model to file by setting `save_best_only=True`.

That way when evaluating our model we could restore its best performing configuration from file.

**Note:** Because of the size of the dataset (smaller than usual), you'll notice our modelling experiment results fluctuate quite a bit during training (hence the implementation of the `ModelCheckpoint` callback to save the best model).

Because we're going to be running multiple experiments, it makes sense to keep track of them by saving models to file under different names.

To do this, we'll write a small function to create a `ModelCheckpoint` callback which saves a model to specified filename.

In [ ]:

```
import os

# Create a function to implement a ModelCheckpoint callback with a specific filename
def create_model_checkpoint(model_name, save_path="model_experiments"):
    return tf.keras.callbacks.ModelCheckpoint(filepath=os.path.join(save_path, model_name),
    # create filepath to save model
    verbose=0, # only output a limited amount of
    f text
    save_best_only=True) # save only the best mo
del to file
```

## Model 1: Dense model (window = 7, horizon = 1)

Finally!

Time to build one of our models.

If you think we've been through a fair bit of preprocessing before getting here, you're right.

Often, preparing data for a model is one of the largest parts of any machine learning project.

And once you've got a good model in place, you'll probably notice far more improvements from manipulating the data (e.g. collecting more, improving the quality) than manipulating the model.

We're going to start by keeping it simple, `model_1` will have:

- A single dense layer with 128 hidden units and ReLU (rectified linear unit) activation
- An output layer with linear activation (or no activation)
- Adam optimizer and MAE loss function
- Batch size of 128
- 100 epochs

Why these values?

I picked them out of experimentation.

A batch size of 32 works pretty well too and we could always train for less epochs but since the model runs so fast (you'll see in a second, it's because the number of samples we have isn't massive) we might as well train for more.

**Note:** As always, many of the values for machine learning problems are experimental. A reminder that the values you can set yourself in a machine learning algorithm (the hidden units, the batch size, horizon size, window size) are called **hyperparameters**. And experimenting to find the best values for hyperparameters is called **hyperparameter tuning**. Whereas parameters learned by a model itself (patterns in the data, formally called weights & biases) are referred to as **parameters**.

Let's import TensorFlow and build our first deep learning model for time series.

In [ ]:

```
import tensorflow as tf
```

```

from tensorflow.keras import layers

# Set random seed for as reproducible results as possible
tf.random.set_seed(42)

# Construct model
model_1 = tf.keras.Sequential([
    layers.Dense(128, activation="relu"),
    layers.Dense(HORIZON, activation="linear") # linear activation is the same as having no
activation
], name="model_1_dense") # give the model a name so we can save it

# Compile model
model_1.compile(loss="mae",
                 optimizer=tf.keras.optimizers.Adam(),
                 metrics=["mae"]) # we don't necessarily need this when the loss function
is already MAE

# Fit model
model_1.fit(x=train_windows, # train windows of 7 timesteps of Bitcoin prices
            y=train_labels, # horizon value of 1 (using the previous 7 timesteps to predict next day)
            epochs=100,
            verbose=1,
            batch_size=128,
            validation_data=(test_windows, test_labels),
            callbacks=[create_model_checkpoint(model_name=model_1.name)]) # create Model
Checkpoint callback to save best model

```

Epoch 1/100  
18/18 [=====] - 3s 12ms/step - loss: 780.3455 - mae: 780.3455 -  
val\_loss: 2279.6526 - val\_mae: 2279.6526  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 2/100  
18/18 [=====] - 0s 4ms/step - loss: 247.6756 - mae: 247.6756 - v  
al\_loss: 1005.9991 - val\_mae: 1005.9991  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 3/100  
18/18 [=====] - 0s 4ms/step - loss: 188.4116 - mae: 188.4116 - v  
al\_loss: 923.2862 - val\_mae: 923.2862  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 4/100  
18/18 [=====] - 0s 4ms/step - loss: 169.4340 - mae: 169.4340 - v  
al\_loss: 900.5872 - val\_mae: 900.5872  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 5/100  
18/18 [=====] - 0s 4ms/step - loss: 165.0895 - mae: 165.0895 - v  
al\_loss: 895.2238 - val\_mae: 895.2238  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 6/100  
18/18 [=====] - 0s 4ms/step - loss: 158.5210 - mae: 158.5210 - v  
al\_loss: 855.1982 - val\_mae: 855.1982  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 7/100  
18/18 [=====] - 0s 5ms/step - loss: 151.3566 - mae: 151.3566 - v  
al\_loss: 840.9168 - val\_mae: 840.9168  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 8/100  
18/18 [=====] - 0s 5ms/step - loss: 145.2560 - mae: 145.2560 - v  
al\_loss: 803.5956 - val\_mae: 803.5956  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 9/100  
18/18 [=====] - 0s 4ms/step - loss: 144.3546 - mae: 144.3546 - v  
al\_loss: 799.5455 - val\_mae: 799.5455  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 10/100  
18/18 [=====] - 0s 4ms/step - loss: 141.2943 - mae: 141.2943 - v  
al\_loss: 763.5010 - val\_mae: 763.5010  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 11/100  
18/18 [=====] - 0s 4ms/step - loss: 135.6595 - mae: 135.6595 - v  
al\_loss: 771.3357 - val\_mae: 771.3357

Epoch 12/100  
18/18 [=====] - 0s 5ms/step - loss: 134.1700 - mae: 134.1700 - val\_loss: 782.8079 - val\_mae: 782.8079  
Epoch 13/100  
18/18 [=====] - 0s 4ms/step - loss: 134.6015 - mae: 134.6015 - val\_loss: 784.4449 - val\_mae: 784.4449  
Epoch 14/100  
18/18 [=====] - 0s 4ms/step - loss: 130.6127 - mae: 130.6127 - val\_loss: 751.3234 - val\_mae: 751.3234  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 15/100  
18/18 [=====] - 0s 5ms/step - loss: 128.8347 - mae: 128.8347 - val\_loss: 696.5757 - val\_mae: 696.5757  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 16/100  
18/18 [=====] - 0s 5ms/step - loss: 124.7739 - mae: 124.7739 - val\_loss: 702.4698 - val\_mae: 702.4698  
Epoch 17/100  
18/18 [=====] - 0s 4ms/step - loss: 123.4474 - mae: 123.4474 - val\_loss: 704.9241 - val\_mae: 704.9241  
Epoch 18/100  
18/18 [=====] - 0s 5ms/step - loss: 122.2105 - mae: 122.2105 - val\_loss: 667.9725 - val\_mae: 667.9725  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 19/100  
18/18 [=====] - 0s 4ms/step - loss: 121.7263 - mae: 121.7263 - val\_loss: 718.8797 - val\_mae: 718.8797  
Epoch 20/100  
18/18 [=====] - 0s 4ms/step - loss: 119.2420 - mae: 119.2420 - val\_loss: 657.0667 - val\_mae: 657.0667  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 21/100  
18/18 [=====] - 0s 4ms/step - loss: 121.2275 - mae: 121.2275 - val\_loss: 637.0330 - val\_mae: 637.0330  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 22/100  
18/18 [=====] - 0s 4ms/step - loss: 119.9544 - mae: 119.9544 - val\_loss: 671.2490 - val\_mae: 671.2490  
Epoch 23/100  
18/18 [=====] - 0s 5ms/step - loss: 121.9248 - mae: 121.9248 - val\_loss: 633.3593 - val\_mae: 633.3593  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 24/100  
18/18 [=====] - 0s 5ms/step - loss: 116.3666 - mae: 116.3666 - val\_loss: 624.4852 - val\_mae: 624.4852  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 25/100  
18/18 [=====] - 0s 4ms/step - loss: 114.6816 - mae: 114.6816 - val\_loss: 619.7571 - val\_mae: 619.7571  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 26/100  
18/18 [=====] - 0s 4ms/step - loss: 116.4455 - mae: 116.4455 - val\_loss: 615.6364 - val\_mae: 615.6364  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 27/100  
18/18 [=====] - 0s 5ms/step - loss: 116.5868 - mae: 116.5868 - val\_loss: 615.9631 - val\_mae: 615.9631  
Epoch 28/100  
18/18 [=====] - 0s 4ms/step - loss: 113.4691 - mae: 113.4691 - val\_loss: 608.0920 - val\_mae: 608.0920  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 29/100  
18/18 [=====] - 0s 5ms/step - loss: 113.7598 - mae: 113.7598 - val\_loss: 621.9306 - val\_mae: 621.9306  
Epoch 30/100  
18/18 [=====] - 0s 4ms/step - loss: 116.8613 - mae: 116.8613 - val\_loss: 604.4056 - val\_mae: 604.4056  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 31/100  
18/18 [=====] - 0s 4ms/step - loss: 111.9375 - mae: 111.9375 - val\_loss: 609.3882 - val\_mae: 609.3882  
Epoch 32/100

```
18/18 [=====] - 0s 4ms/step - loss: 112.4175 - mae: 112.4175 - val_loss: 603.0588 - val_mae: 603.0588
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 33/100
18/18 [=====] - 0s 4ms/step - loss: 112.6697 - mae: 112.6697 - val_loss: 645.6975 - val_mae: 645.6975
Epoch 34/100
18/18 [=====] - 0s 4ms/step - loss: 111.9867 - mae: 111.9867 - val_loss: 604.7632 - val_mae: 604.7632
Epoch 35/100
18/18 [=====] - 0s 4ms/step - loss: 110.9451 - mae: 110.9451 - val_loss: 593.4648 - val_mae: 593.4648
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 36/100
18/18 [=====] - 0s 5ms/step - loss: 114.4816 - mae: 114.4816 - val_loss: 608.0073 - val_mae: 608.0073
Epoch 37/100
18/18 [=====] - 0s 4ms/step - loss: 110.2017 - mae: 110.2017 - val_loss: 597.2309 - val_mae: 597.2309
Epoch 38/100
18/18 [=====] - 0s 5ms/step - loss: 112.2372 - mae: 112.2372 - val_loss: 637.9797 - val_mae: 637.9797
Epoch 39/100
18/18 [=====] - 0s 4ms/step - loss: 115.1289 - mae: 115.1289 - val_loss: 587.4679 - val_mae: 587.4679
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 40/100
18/18 [=====] - 0s 5ms/step - loss: 110.0854 - mae: 110.0854 - val_loss: 592.7117 - val_mae: 592.7117
Epoch 41/100
18/18 [=====] - 0s 4ms/step - loss: 110.6343 - mae: 110.6343 - val_loss: 593.8997 - val_mae: 593.8997
Epoch 42/100
18/18 [=====] - 0s 4ms/step - loss: 113.5762 - mae: 113.5762 - val_loss: 636.3674 - val_mae: 636.3674
Epoch 43/100
18/18 [=====] - 0s 5ms/step - loss: 116.2286 - mae: 116.2286 - val_loss: 662.9264 - val_mae: 662.9264
Epoch 44/100
18/18 [=====] - 0s 4ms/step - loss: 120.0192 - mae: 120.0192 - val_loss: 635.6360 - val_mae: 635.6360
Epoch 45/100
18/18 [=====] - 0s 4ms/step - loss: 110.9675 - mae: 110.9675 - val_loss: 601.9926 - val_mae: 601.9926
Epoch 46/100
18/18 [=====] - 0s 4ms/step - loss: 111.6012 - mae: 111.6012 - val_loss: 593.3531 - val_mae: 593.3531
Epoch 47/100
18/18 [=====] - 0s 6ms/step - loss: 109.6161 - mae: 109.6161 - val_loss: 637.0014 - val_mae: 637.0014
Epoch 48/100
18/18 [=====] - 0s 5ms/step - loss: 109.1368 - mae: 109.1368 - val_loss: 598.4199 - val_mae: 598.4199
Epoch 49/100
18/18 [=====] - 0s 5ms/step - loss: 112.4355 - mae: 112.4355 - val_loss: 579.7040 - val_mae: 579.7040
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 50/100
18/18 [=====] - 0s 4ms/step - loss: 110.2108 - mae: 110.2108 - val_loss: 639.2326 - val_mae: 639.2326
Epoch 51/100
18/18 [=====] - 0s 5ms/step - loss: 111.0958 - mae: 111.0958 - val_loss: 597.3575 - val_mae: 597.3575
Epoch 52/100
18/18 [=====] - 0s 4ms/step - loss: 110.7351 - mae: 110.7351 - val_loss: 580.7227 - val_mae: 580.7227
Epoch 53/100
18/18 [=====] - 0s 5ms/step - loss: 111.1785 - mae: 111.1785 - val_loss: 648.3588 - val_mae: 648.3588
Epoch 54/100
18/18 [=====] - 0s 4ms/step - loss: 114.0832 - mae: 114.0832 - val_loss: 593.2007 - val_mae: 593.2007
```

Epoch 55/100  
18/18 [=====] - 0s 4ms/step - loss: 110.4910 - mae: 110.4910 - val\_loss: 579.5065 - val\_mae: 579.5065  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 56/100  
18/18 [=====] - 0s 5ms/step - loss: 108.0489 - mae: 108.0489 - val\_loss: 807.3851 - val\_mae: 807.3851  
Epoch 57/100  
18/18 [=====] - 0s 4ms/step - loss: 125.0614 - mae: 125.0614 - val\_loss: 674.1654 - val\_mae: 674.1654  
Epoch 58/100  
18/18 [=====] - 0s 4ms/step - loss: 115.4340 - mae: 115.4340 - val\_loss: 582.2698 - val\_mae: 582.2698  
Epoch 59/100  
18/18 [=====] - 0s 5ms/step - loss: 110.0881 - mae: 110.0881 - val\_loss: 606.7637 - val\_mae: 606.7637  
Epoch 60/100  
18/18 [=====] - 0s 4ms/step - loss: 108.7156 - mae: 108.7156 - val\_loss: 602.3102 - val\_mae: 602.3102  
Epoch 61/100  
18/18 [=====] - 0s 4ms/step - loss: 108.1525 - mae: 108.1525 - val\_loss: 573.9990 - val\_mae: 573.9990  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 62/100  
18/18 [=====] - 0s 4ms/step - loss: 107.3727 - mae: 107.3727 - val\_loss: 581.7012 - val\_mae: 581.7012  
Epoch 63/100  
18/18 [=====] - 0s 4ms/step - loss: 110.7667 - mae: 110.7667 - val\_loss: 637.5252 - val\_mae: 637.5252  
Epoch 64/100  
18/18 [=====] - 0s 4ms/step - loss: 110.1539 - mae: 110.1539 - val\_loss: 586.6601 - val\_mae: 586.6601  
Epoch 65/100  
18/18 [=====] - 0s 5ms/step - loss: 108.2325 - mae: 108.2325 - val\_loss: 573.5620 - val\_mae: 573.5620  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 66/100  
18/18 [=====] - 0s 5ms/step - loss: 108.6825 - mae: 108.6825 - val\_loss: 572.2206 - val\_mae: 572.2206  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 67/100  
18/18 [=====] - 0s 4ms/step - loss: 106.6371 - mae: 106.6371 - val\_loss: 646.6349 - val\_mae: 646.6349  
Epoch 68/100  
18/18 [=====] - 0s 4ms/step - loss: 114.1603 - mae: 114.1603 - val\_loss: 681.8561 - val\_mae: 681.8561  
Epoch 69/100  
18/18 [=====] - 0s 5ms/step - loss: 124.5514 - mae: 124.5514 - val\_loss: 655.9885 - val\_mae: 655.9885  
Epoch 70/100  
18/18 [=====] - 0s 4ms/step - loss: 125.0235 - mae: 125.0235 - val\_loss: 601.0032 - val\_mae: 601.0032  
Epoch 71/100  
18/18 [=====] - 0s 6ms/step - loss: 110.3652 - mae: 110.3652 - val\_loss: 595.3962 - val\_mae: 595.3962  
Epoch 72/100  
18/18 [=====] - 0s 5ms/step - loss: 107.9285 - mae: 107.9285 - val\_loss: 573.7085 - val\_mae: 573.7085  
Epoch 73/100  
18/18 [=====] - 0s 4ms/step - loss: 109.5085 - mae: 109.5085 - val\_loss: 580.4180 - val\_mae: 580.4180  
Epoch 74/100  
18/18 [=====] - 0s 4ms/step - loss: 108.7380 - mae: 108.7380 - val\_loss: 576.1211 - val\_mae: 576.1211  
Epoch 75/100  
18/18 [=====] - 0s 5ms/step - loss: 107.9404 - mae: 107.9404 - val\_loss: 591.1477 - val\_mae: 591.1477  
Epoch 76/100  
18/18 [=====] - 0s 5ms/step - loss: 109.4232 - mae: 109.4232 - val\_loss: 597.8605 - val\_mae: 597.8605  
Epoch 77/100  
18/18 [=====] - 0s 4ms/step - loss: 107.5879 - mae: 107.5879 - val\_loss: 597.8605 - val\_mae: 597.8605

al\_loss: 571.9299 - val\_mae: 571.9299  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 78/100  
18/18 [=====] - 0s 4ms/step - loss: 108.1598 - mae: 108.1598 - v  
al\_loss: 575.2383 - val\_mae: 575.2383  
Epoch 79/100  
18/18 [=====] - 0s 4ms/step - loss: 107.9175 - mae: 107.9175 - v  
al\_loss: 617.3071 - val\_mae: 617.3071  
Epoch 80/100  
18/18 [=====] - 0s 4ms/step - loss: 108.9510 - mae: 108.9510 - v  
al\_loss: 583.4847 - val\_mae: 583.4847  
Epoch 81/100  
18/18 [=====] - 0s 5ms/step - loss: 106.0505 - mae: 106.0505 - v  
al\_loss: 570.0802 - val\_mae: 570.0802  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 82/100  
18/18 [=====] - 0s 4ms/step - loss: 115.6827 - mae: 115.6827 - v  
al\_loss: 575.7382 - val\_mae: 575.7382  
Epoch 83/100  
18/18 [=====] - 0s 5ms/step - loss: 110.9379 - mae: 110.9379 - v  
al\_loss: 659.6570 - val\_mae: 659.6570  
Epoch 84/100  
18/18 [=====] - 0s 5ms/step - loss: 111.4836 - mae: 111.4836 - v  
al\_loss: 570.1959 - val\_mae: 570.1959  
Epoch 85/100  
18/18 [=====] - 0s 4ms/step - loss: 107.5948 - mae: 107.5948 - v  
al\_loss: 601.5945 - val\_mae: 601.5945  
Epoch 86/100  
18/18 [=====] - 0s 4ms/step - loss: 108.9426 - mae: 108.9426 - v  
al\_loss: 592.8107 - val\_mae: 592.8107  
Epoch 87/100  
18/18 [=====] - 0s 5ms/step - loss: 105.7717 - mae: 105.7717 - v  
al\_loss: 603.6169 - val\_mae: 603.6169  
Epoch 88/100  
18/18 [=====] - 0s 5ms/step - loss: 107.9217 - mae: 107.9217 - v  
al\_loss: 569.0500 - val\_mae: 569.0500  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 89/100  
18/18 [=====] - 0s 5ms/step - loss: 106.0344 - mae: 106.0344 - v  
al\_loss: 568.9512 - val\_mae: 568.9512  
INFO:tensorflow:Assets written to: model\_experiments/model\_1\_dense/assets  
Epoch 90/100  
18/18 [=====] - 0s 5ms/step - loss: 105.4977 - mae: 105.4977 - v  
al\_loss: 581.7681 - val\_mae: 581.7681  
Epoch 91/100  
18/18 [=====] - 0s 5ms/step - loss: 108.8468 - mae: 108.8468 - v  
al\_loss: 573.6023 - val\_mae: 573.6023  
Epoch 92/100  
18/18 [=====] - 0s 5ms/step - loss: 110.8884 - mae: 110.8884 - v  
al\_loss: 576.8247 - val\_mae: 576.8247  
Epoch 93/100  
18/18 [=====] - 0s 5ms/step - loss: 113.8781 - mae: 113.8781 - v  
al\_loss: 608.3018 - val\_mae: 608.3018  
Epoch 94/100  
18/18 [=====] - 0s 5ms/step - loss: 110.5763 - mae: 110.5763 - v  
al\_loss: 601.6047 - val\_mae: 601.6047  
Epoch 95/100  
18/18 [=====] - 0s 4ms/step - loss: 106.5906 - mae: 106.5906 - v  
al\_loss: 570.3652 - val\_mae: 570.3652  
Epoch 96/100  
18/18 [=====] - 0s 4ms/step - loss: 116.9515 - mae: 116.9515 - v  
al\_loss: 615.2581 - val\_mae: 615.2581  
Epoch 97/100  
18/18 [=====] - 0s 5ms/step - loss: 108.0739 - mae: 108.0739 - v  
al\_loss: 580.3073 - val\_mae: 580.3073  
Epoch 98/100  
18/18 [=====] - 0s 4ms/step - loss: 108.7102 - mae: 108.7102 - v  
al\_loss: 586.6512 - val\_mae: 586.6512  
Epoch 99/100  
18/18 [=====] - 0s 5ms/step - loss: 109.0488 - mae: 109.0488 - v  
al\_loss: 570.0629 - val\_mae: 570.0629  
Epoch 100/100

```
18/18 [=====] - 0s 4ms/step - loss: 106.1845 - mae: 106.1845 - val_loss: 585.9763 - val_mae: 585.9763
```

Out[ ]:

```
<keras.callbacks.History at 0x7fdcf48bd110>
```

**Because of the small size of our data (less than 3000 total samples), the model trains very fast.**

Let's evaluate it.

In [ ]:

```
# Evaluate model on test data
model_1.evaluate(test_windows, test_labels)
```

```
18/18 [=====] - 0s 2ms/step - loss: 585.9762 - mae: 585.9762
```

Out[ ]:

```
[585.9761962890625, 585.9761962890625]
```

You'll notice the model achieves the same `val_loss` (in this case, this is MAE) as the last epoch.

But if we load in the version of `model_1` which was saved to file using the `ModelCheckpoint` callback, we should see an improvement in results.

In [ ]:

```
# Load in saved best performing model_1 and evaluate on test data
model_1 = tf.keras.models.load_model("model_experiments/model_1_dense")
model_1.evaluate(test_windows, test_labels)
```

```
18/18 [=====] - 0s 3ms/step - loss: 568.9512 - mae: 568.9512
```

Out[ ]:

```
[568.951171875, 568.951171875]
```

Much better! Due to the fluctuating performance of the model during training, loading back in the best performing model see's a sizeable improvement in MAE.

## Making forecasts with a model (on the test dataset)

We've trained a model and evaluated the it on the test data, but the project we're working on is called BitPredict so how do you think we could use our model to make predictions?

Since we're going to be running more modelling experiments, let's write a function which:

1. Takes in a trained model (just like `model_1`)
2. Takes in some input data (just like the data the model was trained on)
3. Passes the input data to the model's `predict()` method
4. Returns the predictions

In [ ]:

```
def make_preds(model, input_data):
    """
    Uses model to make predictions on input_data.

    Parameters
    -----
    model: trained model
    input_data: windowed input data (same kind of data model was trained on)

    Returns model predictions on input_data.
    """

```

```
forecast = model.predict(input_data)
return tf.squeeze(forecast) # return 1D array of predictions
```

Nice!

Now let's use our `make_preds()` and see how it goes.

In [ ]:

```
# Make predictions using model_1 on the test dataset and view the results
model_1_preds = make_preds(model_1, test_windows)
len(model_1_preds), model_1_preds[:10]
```

Out [ ]:

```
(556, <tf.Tensor: shape=(10,), dtype=float32, numpy=
array([8861.711, 8769.886, 9015.71 , 8795.517, 8723.809, 8730.11 ,
       8691.95 , 8502.054, 8460.961, 8516.547], dtype=float32)>)
```

⚠ Note: With these outputs, our model isn't *forecasting* yet. It's only making predictions on the test dataset. Forecasting would involve a model making predictions into the future, however, the test dataset is only a pseudofuture.

Excellent! Now we've got some prediction values, let's use the `evaluate_preds()` we created before to compare them to the ground truth.

In [ ]:

```
# Evaluate preds
model_1_results = evaluate_preds(y_true=tf.squeeze(test_labels), # reduce to right shape
                                  y_pred=model_1_preds)
model_1_results
```

Out [ ]:

```
{'mae': 568.95123,
'mape': 2.5448983,
'mase': 0.9994897,
'mse': 1171744.0,
'rmse': 1082.4713}
```

How did our model go? Did it beat the naïve forecast?

In [ ]:

```
naive_results
```

Out [ ]:

```
{'mae': 567.9802,
'mape': 2.516525,
'mase': 0.99957,
'mse': 1147547.0,
'rmse': 1071.2362}
```

It looks like our naïve model beats our first deep model on nearly every metric.

That goes to show the power of the naïve model and the reason for having a baseline for any machine learning project.

And of course, no evaluation would be finished without visualizing the results.

Let's use the `plot_time_series()` function to plot `model_1_preds` against the test data.

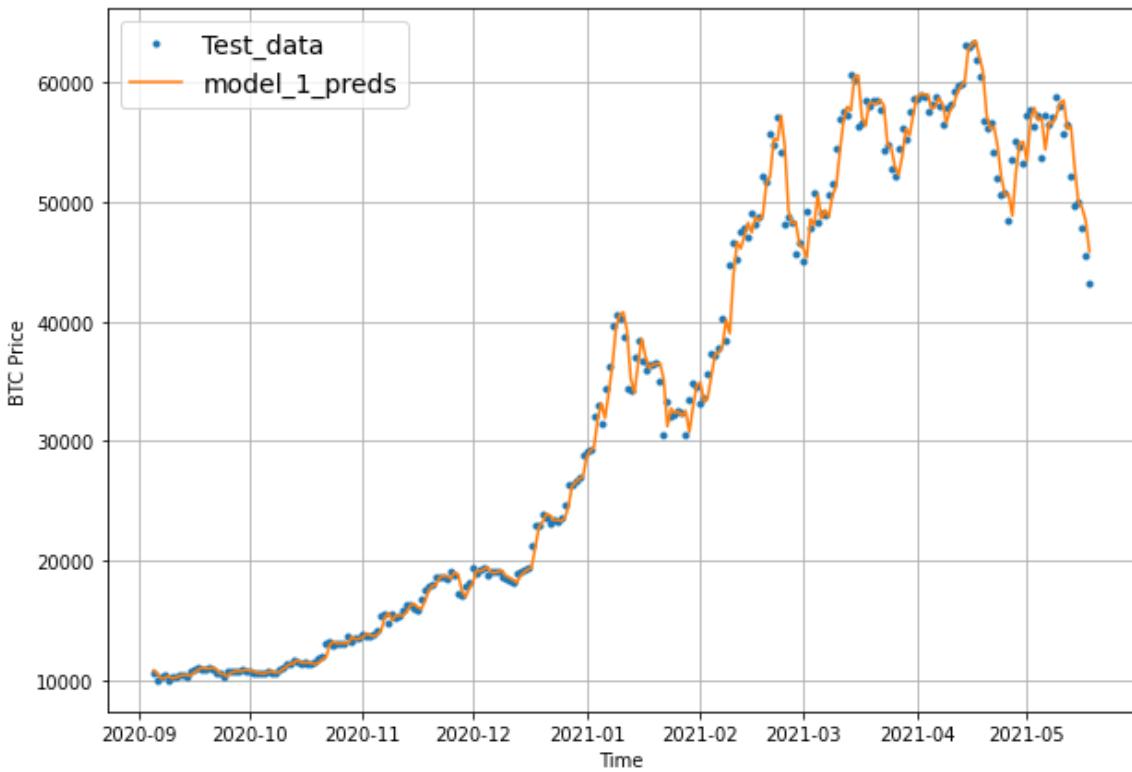
In [ ]:

```
offset = 300
```

```

plt.figure(figsize=(10, 7))
# Account for the test_window offset and index into test_labels to ensure correct plotting
plot_time_series(timesteps=X_test[-len(test_windows):], values=test_labels[:, 0], start=offset, label="Test_data")
plot_time_series(timesteps=X_test[-len(test_windows):], values=model_1_preds, start=offset, format="--", label="model_1_preds")

```



### What's wrong with these predictions?

As mentioned before, they're on the test dataset. So they're not actual forecasts.

With our current model setup, how do you think we'd make forecasts for the future?

Have a think about it for now, we'll cover this later on.

## Model 2: Dense (window = 30, horizon = 1)

A naïve model is currently beating our handcrafted deep learning model.

We can't let this happen.

Let's continue our modelling experiments.

We'll keep the previous model architecture but use a window size of 30.

In other words, we'll use the previous 30 days of Bitcoin prices to try and predict the next day price.

```

# Window for one month with the target of predicting the next day (Bitcoin prices)
[123.654, 125.455, 108.584, 118.674, 121.338, 120.655,
121.795, 123.033, 124.049, 125.961, 125.279, 125.927,
126.383, 135.241, 133.203, 142.763, 137.923, 142.951,
152.551, 160.338, 164.314, 177.633, 188.297, 200.701,
180.355, 175.031, 177.696, 187.159, 192.756, 197.400] -> [196.024]

[125.455, 108.584, 118.674, 121.338, 120.655, 121.795,
123.033, 124.049, 125.961, 125.279, 125.927, 126.383,
135.241, 133.203, 142.763, 137.923, 142.951, 152.551,
160.338, 164.314, 177.633, 188.297, 200.701, 180.355]

```

Window size = 30

Horizon = 1

**Data****Label**

(These values are tuneable hyperparameters)

**Example of Bitcoin prices windowed for 30 days to predict a horizon of 1.**

**Note:** Recall from before, the **window size** (how many timesteps to use to fuel a forecast) and the **horizon** (how many timesteps to predict into the future) are **hyperparameters**. This means you can tune them to try and find values will result in better performance.

We'll start our second modelling experiment by preparing datasets using the functions we created earlier.

In [ ]:

```
HORIZON = 1 # predict one step at a time
WINDOW_SIZE = 30 # use 30 timesteps in the past
```

In [ ]:

```
# Make windowed data with appropriate horizon and window sizes
full_windows, full_labels = make_windows(prices, window_size=WINDOW_SIZE, horizon=HORIZON)
len(full_windows), len(full_labels)
```

Out [ ]:

```
(2757, 2757)
```

In [ ]:

```
# Make train and testing windows
train_windows, test_windows, train_labels, test_labels = make_train_test_splits(windows=full_windows, labels=full_labels)
len(train_windows), len(test_windows), len(train_labels), len(test_labels)
```

Out [ ]:

```
(2205, 552, 2205, 552)
```

## Data prepared!

Now let's construct `model_2`, a model with the same architecture as `model_1` as well as the same training routine.

In [ ]:

```
tf.random.set_seed(42)

# Create model (same model as model 1 but data input will be different)
model_2 = tf.keras.Sequential([
    layers.Dense(128, activation="relu"),
    layers.Dense(HORIZON) # need to predict horizon number of steps into the future
], name="model_2_dense")

model_2.compile(loss="mae",
                 optimizer=tf.keras.optimizers.Adam())

model_2.fit(train_windows,
            train_labels,
            epochs=100,
            batch_size=128,
            verbose=0,
            validation_data=(test_windows, test_labels),
            callbacks=[create_model_checkpoint(model_name=model_2.name)])
```

INFO:tensorflow:Assets written to: model\_experiments/model\_2\_dense/assets  
INFO:tensorflow:Assets written to: model\_experiments/model\_2\_dense/assets

```
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
```

Out [ ]:

```
<keras.callbacks.History at 0x7fdcf1094290>
```

**Once again, training goes nice and fast.**

**Let's evaluate our model's performance.**

In [ ]:

```
# Evaluate model 2 preds
model_2.evaluate(test_windows, test_labels)
```

```
18/18 [=====] - 0s 2ms/step - loss: 608.9615
```

Out [ ]:

```
608.9614868164062
```

**Hmmm... is that the best it did?**

**How about we try loading in the best performing `model_2` which was saved to file thanks to our ModelCheckpoint callback.**

In [ ]:

```
# Load in best performing model
model_2 = tf.keras.models.load_model("model_experiments/model_2_dense/")
model_2.evaluate(test_windows, test_labels)
```

```
18/18 [=====] - 0s 2ms/step - loss: 608.9615
```

Out [ ]:

```
608.9614868164062
```

**Excellent! Loading back in the best performing model see's a performance boost.**

**But let's not stop there, let's make some predictions with `model_2` and then evaluate them just as we did before.**

In [ ]:

```
# Get forecast predictions
```

```
model_2_preds = make_preds(model_2,
                           input_data=test_windows)
```

In [ ]:

```
# Evaluate results for model 2 predictions
model_2_results = evaluate_preds(y_true=tf.squeeze(test_labels), # remove 1 dimension of
                                  test_labels
                                  y_pred=model_2_preds)
model_2_results
```

Out[ ]:

```
{'mae': 608.9615,
'mape': 2.7693386,
'mase': 1.0644706,
'mse': 1281438.8,
'rmse': 1132.0065}
```

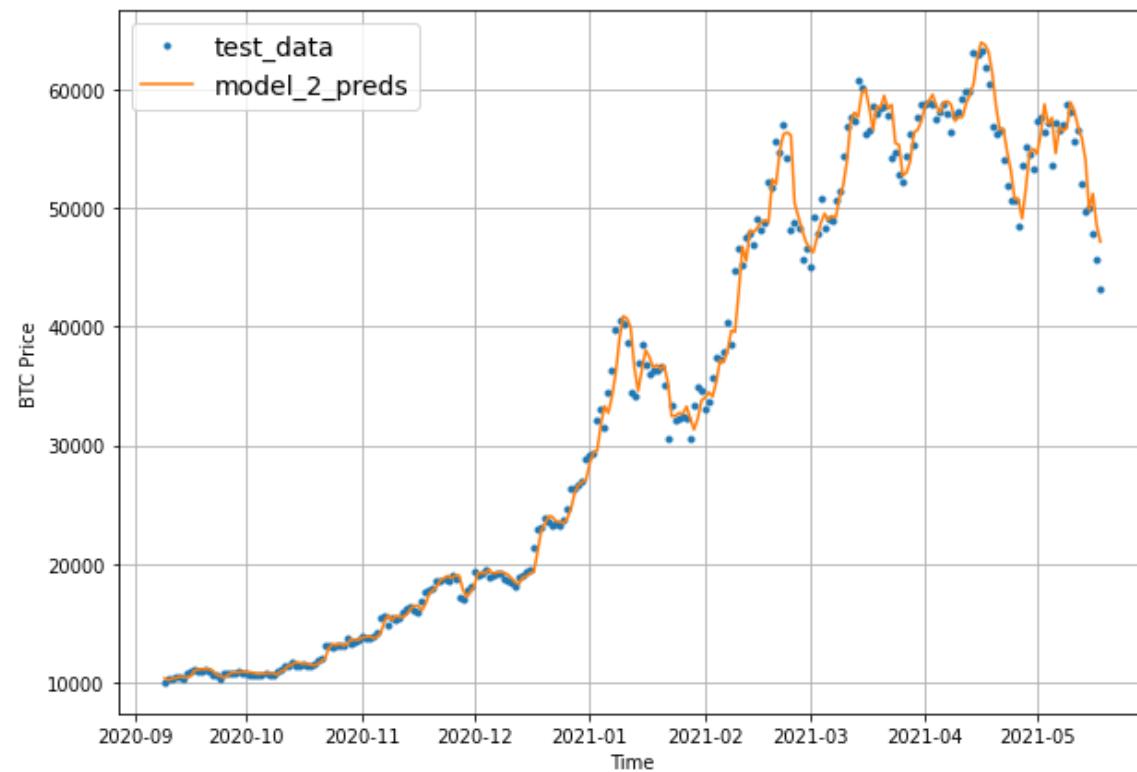
It looks like `model_2` performs worse than the naïve model as well as `model_1`!

Does this mean a smaller window size is better? (I'll leave this as a challenge you can experiment with)

How do the predictions look?

In [ ]:

```
offset = 300
plt.figure(figsize=(10, 7))
# Account for the test_window offset
plot_time_series(timesteps=X_test[-len(test_windows):], values=test_labels[:, 0], start=offset, label="test_data")
plot_time_series(timesteps=X_test[-len(test_windows):], values=model_2_preds, start=offset, format="--", label="model_2_preds")
```



## Model 3: Dense (window = 30, horizon = 7)

Let's try and predict 7 days ahead given the previous 30 days.

First, we'll update the `HORIZON` and `WINDOW_SIZE` variables and create windowed data.

In [ ]:

```
HORIZON = 1
WINDOW_SIZE = 30

full_windows, full_labels = make_windows(prices, window_size=WINDOW_SIZE, horizon=HORIZON)
len(full_windows), len(full_labels)
```

Out[ ]:

(2751, 2751)

And we'll split the full dataset windows into training and test sets

In [ ]:

```
train_windows, test_windows, train_labels, test_labels = make_train_test_splits(windows=full_windows, labels=full_labels, test_split=0.2)
len(train windows), len(test windows), len(train labels), len(test labels)
```

Out[ ]:

(2200, 551, 2200, 551)

**Now let's build, compile, fit and evaluate a model.**

In [ ]:

```
tf.random.set_seed(42)
```

# Create model (same as model 1 except with different data input size)

```
model_3 = tf.keras.Sequential([
```

```
model_3 = Sequential([
    layers.Dense(128, activation="relu"),
    layers.Dense(HORIZON),
    name="model 3 dense")
```

```
model_3.fit(train_windows,  
            train_labels,  
            batch_size=128,  
            epochs=100,  
            verbose=0,  
            validation_data=(test_windows, test_labels),  
            callbacks=[create model checkpoint(model name=model_3.name)])
```

Out[1]:

```
<keras.callbacks.History at 0x7fdcf1d36350>
```

In [ ]:

```
# How did our model with a larger window size and horizon go?
model_3.evaluate(test_windows, test_labels)
```

18/18 [=====] - 0s 2ms/step - loss: 1321.5201

Out[ ]:

1321.5201416015625

**To compare apples to apples (best performing model to best performing model), we've got to load in the best version of `model_3`.**

In [ ]:

```
# Load in best version of model_3 and evaluate
model_3 = tf.keras.models.load_model("model_experiments/model_3_dense/")
model_3.evaluate(test_windows, test_labels)
```

18/18 [=====] - 0s 2ms/step - loss: 1237.5063

Out[ ]:

1237.50634765625

**In this case, the error will be higher because we're predicting 7 steps at a time.**

**This makes sense though because the further you try and predict, the larger your error will be (think of trying to predict the weather 7 days in advance).**

**Let's make predictions with our model using the `make_preds()` function and evaluate them using the `evaluate_preds()` function.**

In [ ]:

```
# The predictions are going to be 7 steps at a time (this is the HORIZON size)
model_3_preds = make_preds(model_3,
                           input_data=test_windows)
model_3_preds[:5]
```

Out[ ]:

```
<tf.Tensor: shape=(5, 7), dtype=float32, numpy=
array([[9004.693, 9048.1, 9425.088, 9258.258, 9495.798, 9558.451,
       9357.354],
       [8735.507, 8840.304, 9247.793, 8885.6, 9097.188, 9174.328,
       9156.819],
       [8672.508, 8782.388, 9123.8545, 8770.37, 9007.13, 9003.87,
       9042.724],
       [8874.398, 8784.737, 9043.901, 8943.051, 9033.479, 9176.488,
       9039.676],
       [8825.891, 8777.4375, 8926.779, 8870.178, 9213.232, 9268.156,
       8942.485]], dtype=float32)>
```

In [ ]:

```
# Calculate model_3 results - these are going to be multi-dimensional because
# we're trying to predict more than one step at a time.
model_3_results = evaluate_preds(y_true=tf.squeeze(test_labels),
                                  y_pred=model_3_preds)
model_3_results
```

Out[ ]:

```
{'mae': array([ 513.60516, 355.08356, 327.17035, 358.50977, 420.53207,
      537.8539, 545.6606, 485.92307, 584.4969, 687.3814,
     836.22675, 755.1571, 731.4959, 775.3398, 567.9548,
    266.80865, 188.80217, 188.1077, 253.09521, 301.4336,
   151.10742, 196.81424, 191.46184, 231.65067, 143.6114,
   122.59033, 132.78844, 190.8116, 179.1598, 228.25949])}
```

314.44016	, 379.093	, 278.3254	, 295.34604	, 299.38525
248.64963	, 299.75635	, 259.6937	, 180.30559	, 206.72887
374.62906	, 144.85129	, 142.33607	, 131.1158	, 93.94057
54.825542	, 73.7943	, 103.59989	, 121.3337	, 168.67209
183.90945	, 152.25314	, 186.57137	, 146.91309	, 240.42955
351.00668	, 540.9516	, 549.15674	, 521.2422	, 526.8553
453.36313	, 257.98166	, 277.29492	, 301.82465	, 455.71756
458.96002	, 503.44427	, 522.3119	, 223.07687	, 250.09473
297.14468	, 400.56976	, 495.79785	, 364.33664	, 283.3654
325.59457	, 238.21178	, 318.9777	, 460.77246	, 651.0755
835.88074	, 669.9654	, 319.5452	, 261.99496	, 142.39217
136.62834	, 154.75252	, 221.32631	, 290.50446	, 503.8846
414.2602	, 434.35727	, 377.98926	, 251.7899	, 204.28418
388.22684	, 360.65945	, 493.80902	, 614.86035	, 754.7017
533.7708	, 378.98898	, 280.49484	, 339.48062	, 413.12875
452.87933	, 550.53906	, 634.57214	, 935.57227	, 931.6003
881.2804	, 426.40094	, 179.45885	, 121.66225	, 160.43806
372.07037	, 341.85776	, 476.52475	, 618.3239	, 1038.8976
1569.5022	, 2157.1196	, 1987.6074	, 2158.6108	, 2303.5603
2662.9421	, 1405.5017	, 728.30145	, 351.70047	, 322.03168
493.94742	, 435.48492	, 565.55615	, 350.1165	, 289.0203
251.21645	, 409.05566	, 342.2103	, 320.83594	, 330.88596
357.8457	, 335.9481	, 206.1031	, 544.3837	, 700.0093
468.91965	, 404.4963	, 172.80176	, 308.33664	, 210.47566
318.95444	, 486.1319	, 428.87982	, 533.91095	, 433.7813
396.89447	, 138.40346	, 189.96617	, 170.39133	, 181.54387
282.8902	, 264.3889	, 250.62172	, 240.33713	, 276.9541
326.0306	, 489.61572	, 686.2451	, 526.9798	, 603.0119
825.38275	, 871.04694	, 990.06903	, 1090.0593	, 560.2842
310.5219	, 371.39035	, 348.45996	, 355.73465	, 429.56473
581.2839	, 550.5218	, 635.4312	, 913.40375	, 840.71844
305.03488	, 493.93414	, 751.3177	, 410.63434	, 220.62459
282.25473	, 291.85352	, 422.50293	, 458.65375	, 637.2345
647.82367	, 417.24442	, 220.23717	, 246.72168	, 200.22935
455.14926	, 719.6058	, 696.3037	, 485.09836	, 294.9534
170.52371	, 211.82463	, 270.56488	, 189.89355	, 171.21721
366.4471	, 231.96303	, 318.78726	, 273.79352	, 358.55582
412.22797	, 512.31573	, 185.43848	, 196.38113	, 200.00586
224.58678	, 213.41965	, 186.23926	, 113.37096	, 172.23117
168.89885	, 236.09584	, 307.59683	, 328.93903	, 566.5961
285.04282	, 300.4495	, 125.45201	, 168.94322	, 137.25754
143.50404	, 145.27776	, 107.340126	, 77.16044	, 131.87096
134.01953	, 167.45871	, 137.92188	, 148.91281	, 204.95467
157.89732	, 196.95201	, 167.93861	, 156.45578	, 188.6808
161.44113	, 90.997765	, 136.84096	, 198.51799	, 230.13881
294.7839	, 594.24286	, 699.64856	, 815.137	, 905.33887
1127.2999	, 1342.6952	, 1317.0686	, 590.1699	, 296.34543
243.4368	, 256.3987	, 222.28488	, 323.8387	, 82.51339
120.14453	, 249.14857	, 205.73674	, 243.45451	, 250.64857
287.27567	, 224.3803	, 266.55972	, 221.50935	, 218.49539
272.42242	, 279.5964	, 252.58775	, 381.56473	, 417.97028
624.5562	, 368.0364	, 327.25473	, 263.4396	, 349.95242
398.62485	, 297.07465	, 147.04924	, 164.54367	, 313.36246
477.7486	, 675.042	, 897.269	, 1094.7098	, 1460.177
1398.5858	, 952.72375	, 645.9594	, 166.17912	, 144.66867
189.1822	, 304.81375	, 435.10742	, 449.64426	, 425.244
441.93637	, 407.29605	, 252.4036	, 248.36928	, 336.17062
482.8111	, 437.53043	, 533.0855	, 346.98047	, 127.68722
110.208565	, 301.75223	, 195.50697	, 174.65248	, 238.4707
302.3711	, 313.51703	, 310.1289	, 200.29701	, 172.47209
140.2302	, 252.48479	, 289.32407	, 343.62222	, 504.11816
635.1339	, 602.131	, 519.0612	, 214.90291	, 195.91197
265.03683	, 198.9008	, 345.51227	, 517.22235	, 631.02997
988.4249	, 1174.2136	, 1196.2849	, 1253.93	, 526.06573
210.31306	, 215.27205	, 169.86008	, 283.78543	, 269.76117
228.63477	, 186.64719	, 410.5434	, 601.2659	, 618.12164
768.7538	, 1158.5449	, 1232.8798	, 1254.429	, 423.02524
390.03613	, 367.23926	, 209.55803	, 530.2231	, 821.38293
812.37195	, 741.19464	, 953.48285	, 1258.9928	, 1844.9332
1605.2852	, 1112.7673	, 594.18945	, 549.7676	, 632.5438
829.2656	, 1103.2213	, 1130.1024	, 1033.3527	, 878.5851
595.4096	, 1115.6515	, 1371.1725	, 1385.25	, 387.52484

```

303.74945 , 495.12305 , 719.5804 , 648.9032 , 766.1624 ,
683.23157 , 596.4121 , 523.09235 , 577.34015 , 1337.6241 ,
2454.597 , 2759.2363 , 3135.8757 , 3407.3806 , 3602.6362 ,
2527.1584 , 1158.1016 , 679.84375 , 748.5586 , 1073.4896 ,
1217.7305 , 1459.5664 , 2502.7336 , 3075.8264 , 3090.2869 ,
2564.758 , 2660.5317 , 2928.9348 , 3690.5566 , 3525.1433 ,
4183.6704 , 5144.0693 , 4384.6533 , 4164.9614 , 4431.0967 ,
3335.121 , 3017.6858 , 2589.5403 , 4103.6313 , 5582.089 ,
5108.456 , 1382.9648 , 953.5435 , 1081.6842 , 2483.1992 ,
2992.056 , 3127.0942 , 2972.2717 , 3054.6477 , 3283.7107 ,
3617.3652 , 1170.2556 , 1074.1886 , 1059.3181 , 1044.5385 ,
1060.6202 , 2115.7234 , 3569.8901 , 3474.3647 , 2448.0173 ,
2641.4202 , 3188.6016 , 4899.913 , 5516.293 , 4635.4883 ,
4677.036 , 5732.8804 , 5866.7344 , 8083.341 , 5049.8237 ,
1476.2316 , 1873.3314 , 1219.1033 , 2324.0698 , 2467.0044 ,
3005.0864 , 2805.0486 , 3688.2556 , 2962.7131 , 3664.236 ,
5618.6836 , 6925.0913 , 9751.091 , 8790.822 , 5057.624 ,
2971.6863 , 1715.495 , 1125.7689 , 2201.432 , 3969.5005 ,
2988.5112 , 2911.9336 , 2796.869 , 3937.8755 , 5449.5913 ,
6395.02 , 6148.2524 , 5437.706 , 4291.8267 , 2362.3962 ,
1657.3444 , 1426.63 , 1647.7142 , 2730.3962 , 1813.716 ,
2061.149 , 3095.72 , 4312.8096 , 5500.7227 , 5258.029 ,
5152.029 , 1331.4045 , 1634.4017 , 2493.4336 , 3957.9548 ,
4499.2153 , 2215.8533 , 2587.4136 , 1622.4392 , 591.8778 ,
1297.4688 , 1284.5931 , 915.22656 , 940.4565 , 1084.6562 ,
875.89954 , 1041.3527 , 2299.2512 , 3152.6038 , 3518.3923 ,
2531.5212 , 2682.5737 , 3094.2947 , 3829.3286 , 5550.609 ,
7159.343 , 8820.526 , 8587.338 , 6777.814 , 5193.2866 ,
4793.332 , 2605.2517 , 1668.3544 , 2510.7244 , 4105.6865 ,
6278.3906 , 4109.759 , 1711.1211 , 2261.9878 , 2384.754 ,
1246.6602 , 1672.8494 , 1103.1322 , 1743.3594 , 1608.452 ,
1933.8995 , 2742.3455 , 3945.3633 , 5765.9414 , 6955.8384 ,
8107.707 ], dtype=float32),
'mape': array([ 5.8601456 , 4.087344 , 3.7758112 , 4.187648 , 4.9781313 ,
6.4483633 , 6.614449 , 6.0616755 , 7.547588 , 9.095916 ,
11.300213 , 10.396466 , 10.109091 , 10.697323 , 7.8628383 ,
3.6872823 , 2.585636 , 2.505 , 3.3504546 , 4.013595 ,
2.0036254 , 2.6397336 , 2.6087892 , 3.160931 , 1.9612147 ,
1.6679796 , 1.7952247 , 2.5901198 , 2.436256 , 3.1327312 ,
4.3595414 , 5.2856445 , 3.9403565 , 4.3142104 , 4.3733163 ,
3.6376827 , 4.362974 , 3.7757344 , 2.6192985 , 2.9007592 ,
5.1501136 , 2.014742 , 1.9706616 , 1.8104095 , 1.3007166 ,
0.75351447 , 1.0192169 , 1.4282547 , 1.6741827 , 2.3589606 ,
2.563099 , 2.1327207 , 2.5974777 , 2.017332 , 3.1387668 ,
4.502132 , 6.959625 , 6.9602156 , 6.5331526 , 6.5757833 ,
5.610295 , 3.0642097 , 3.2474015 , 3.4847279 , 5.222716 ,
5.229475 , 5.727943 , 5.9273834 , 2.5284538 , 2.8663971 ,
3.4309018 , 4.6874056 , 5.841506 , 4.30981 , 3.3552744 ,
3.8162014 , 2.7363582 , 3.4623704 , 4.971694 , 7.0260944 ,
8.974986 , 7.167823 , 3.401335 , 2.7761257 , 1.522323 ,
1.4535459 , 1.6354691 , 2.252781 , 2.9475648 , 5.073901 ,
4.1032834 , 4.295298 , 3.7052956 , 2.4620814 , 2.013232 ,
3.8428285 , 3.6306674 , 5.0162754 , 6.2794676 , 7.757465 ,
5.495671 , 3.8907118 , 2.8546908 , 3.531113 , 4.464891 ,
5.0162473 , 6.1257253 , 7.1294317 , 10.711446 , 10.66408 ,
10.087247 , 4.893642 , 2.071729 , 1.362841 , 1.8011061 ,
4.219873 , 4.055696 , 5.744903 , 7.492872 , 15.032968 ,
24.38846 , 35.42812 , 34.702423 , 39.86109 , 42.83657 ,
49.6046 , 26.081123 , 13.669959 , 6.360867 , 5.4986367 ,
7.9925113 , 6.8575478 , 8.82061 , 5.311197 , 4.5279207 ,
3.870413 , 6.2283797 , 5.2470355 , 5.1045485 , 5.329424 ,
5.7789664 , 5.358487 , 3.2211263 , 8.118441 , 10.231978 ,
6.726861 , 5.757309 , 2.4041245 , 4.2864537 , 2.977692 ,
4.454852 , 6.9261975 , 6.1590724 , 7.7638254 , 6.308201 ,
5.7371497 , 1.9719449 , 2.7191157 , 2.3998728 , 2.568591 ,
3.953004 , 3.6584775 , 3.453558 , 3.301869 , 3.7942226 ,
4.310324 , 6.4319835 , 8.567106 , 6.293662 , 7.0238814 ,
9.469167 , 9.915066 , 11.217036 , 12.28066 , 6.209284 ,
3.2802734 , 3.8508573 , 3.6124747 , 3.7172396 , 4.6127787 ,
6.2338514 , 6.037696 , 7.1359625 , 10.132258 , 9.337389 ,
3.379447 , 5.190343 , 7.822919 , 4.2644997 , 2.328503 ,
3.0320923 , 3.122556 , 4.5721846 , 5.013454 , 7.0593185 ,

```

7.180876	, 4.6288185	, 2.4294462	, 2.6893413	, 2.1346936
4.6703887	, 7.429678	, 7.177754	, 4.9274387	, 2.969763
1.7036035	, 2.1020818	, 2.7900844	, 1.9541025	, 1.7562655
3.8170404	, 2.4557195	, 3.335657	, 2.8912652	, 3.8006802
4.3685193	, 5.4371476	, 1.96905	, 2.086854	, 2.1329875
2.4014482	, 2.2829742	, 1.9878384	, 1.2127163	, 1.8306142
1.8156711	, 2.5207567	, 3.2923858	, 3.5837193	, 6.1877694
3.1183949	, 3.2884538	, 1.3740205	, 1.8472785	, 1.5017135
1.5671413	, 1.5889224	, 1.1630745	, 0.83710796	, 1.4223018
1.4407477	, 1.8030001	, 1.4797007	, 1.599079	, 2.2175813
1.7130904	, 2.1422603	, 1.8232663	, 1.7015574	, 2.0563016
1.7596645	, 0.98229814	, 1.4560648	, 2.1070251	, 2.4114208
3.0499995	, 5.7738442	, 6.618569	, 7.495159	, 8.227665
10.186895	, 11.928129	, 11.676125	, 5.199727	, 2.5929737
2.1249924	, 2.2335236	, 1.9233094	, 2.846623	, 0.7076666
1.0350616	, 2.132228	, 1.778991	, 2.1161108	, 2.1635604
2.4792838	, 1.9270526	, 2.2551262	, 1.8457135	, 1.8112589
2.262672	, 2.3515143	, 2.1173332	, 3.2111585	, 3.5736024
5.3715696	, 3.18694	, 2.858538	, 2.295805	, 3.0580947
3.4806054	, 2.5900245	, 1.2625037	, 1.4050376	, 2.7622259
4.3049536	, 6.2830725	, 8.513568	, 10.517086	, 14.161078
13.640608	, 9.320237	, 6.3080745	, 1.6300542	, 1.3983166
1.8191801	, 2.8875985	, 4.068832	, 4.1736484	, 3.92286
4.0472727	, 3.7387483	, 2.3069727	, 2.3063238	, 3.1941137
4.5895944	, 4.157942	, 5.0558047	, 3.3005743	, 1.2102847
1.037109	, 2.8088598	, 1.8151615	, 1.6353209	, 2.2353542
2.8412042	, 2.952003	, 2.9187465	, 1.8913074	, 1.6178632
1.2982845	, 2.3039308	, 2.6034784	, 3.017652	, 4.444261
5.5989385	, 5.26695	, 4.535601	, 1.8671715	, 1.7055075
2.3196962	, 1.7318225	, 2.7897227	, 4.062603	, 4.8944497
7.6816416	, 9.107798	, 9.207181	, 9.502263	, 3.9754786
1.5744586	, 1.5997065	, 1.242373	, 2.0770247	, 1.974714
1.6764196	, 1.3502706	, 2.7858412	, 3.9915662	, 4.0835853
5.016299	, 7.6388373	, 8.078356	, 8.1091385	, 2.6612654
2.4994845	, 2.2963924	, 1.304031	, 3.2585588	, 4.9145164
4.7377896	, 4.253874	, 5.353144	, 6.9800787	, 10.170477
8.772201	, 6.0037975	, 3.184015	, 3.0090022	, 3.637018
4.7740936	, 6.289234	, 6.4191875	, 5.904088	, 4.9963098
3.1906037	, 5.8562336	, 7.18103	, 7.211652	, 2.0219958
1.5912417	, 2.624319	, 3.8534112	, 3.5140653	, 4.1405296
3.6843975	, 3.215591	, 2.8228219	, 2.8898468	, 6.2849083
11.458009	, 12.342688	, 13.764669	, 14.824157	, 15.566204
10.829556	, 4.9278235	, 2.8345613	, 3.0226748	, 4.311801
4.7200484	, 5.516797	, 9.303045	, 11.212711	, 11.037065
8.631595	, 8.57773	, 9.389048	, 11.497772	, 10.463535
11.683612	, 13.823776	, 11.273857	, 10.805961	, 11.400717
8.4966135	, 7.8893795	, 7.169477	, 11.423848	, 15.540573
14.24638	, 3.8430922	, 2.545029	, 2.926236	, 7.288314
8.999365	, 9.586316	, 9.271836	, 9.423251	, 10.242945
11.396603	, 3.6557918	, 3.3454742	, 3.2290876	, 3.159098
3.1852577	, 6.1412444	, 10.163153	, 9.673946	, 6.657339
6.975828	, 8.377453	, 12.244584	, 13.00238	, 10.547294
10.269914	, 12.4054	, 12.598526	, 17.187769	, 10.61679
3.0717697	, 3.7662785	, 2.3915377	, 4.4263554	, 4.6217527
5.4785805	, 5.1152287	, 6.826434	, 5.61515	, 7.0909567
11.599258	, 14.51015	, 20.678543	, 18.670784	, 10.813841
6.395925	, 3.68131	, 2.323965	, 4.4593787	, 8.030951
5.9532943	, 5.6428704	, 5.1379447	, 7.1625338	, 9.831415
11.209798	, 10.556711	, 9.350725	, 7.3507624	, 3.9944
2.8233469	, 2.3929296	, 2.8540497	, 4.729086	, 3.236382
3.6810744	, 5.6685753	, 8.005048	, 10.247115	, 9.747856
9.5460415	, 2.4771707	, 2.9854116	, 4.3493814	, 6.8806467
7.758685	, 3.8011405	, 4.4251776	, 2.7610898	, 1.0093751
2.2422	, 2.2289147	, 1.58688	, 1.6239213	, 1.8814766
1.5162641	, 1.7166423	, 3.7594385	, 5.0978	, 5.664717
4.0754285	, 4.3486743	, 5.1419683	, 6.618967	, 9.823991
12.98137	, 16.388018	, 16.260675	, 13.249782	, 10.180592
9.383685	, 5.1218452	, 3.2893803	, 4.634893	, 7.4068375
11.284324	, 7.287351	, 3.0196326	, 4.006139	, 4.172643
2.233759	, 3.0054173	, 1.9702865	, 3.1210938	, 2.7774746
3.4924886	, 5.1419373	, 7.6572022	, 11.470654	, 14.311543
17.28017	1.	dtvpe=float32).		

```
'mase': 2.2020736,
'mse': array([3.15562156e+05, 1.69165547e+05, 1.44131812e+05, 1.76002922e+05,
 2.63519750e+05, 3.91517188e+05, 3.99524688e+05, 3.44422312e+05,
 4.93892156e+05, 6.88785625e+05, 9.18703562e+05, 8.00988125e+05,
 6.35508625e+05, 6.45535125e+05, 3.57740000e+05, 1.03007117e+05,
 5.33132578e+04, 4.65829805e+04, 1.10349188e+05, 1.11647695e+05,
 4.35359297e+04, 5.45387773e+04, 4.55849375e+04, 7.64886406e+04,
 4.03074453e+04, 2.71072188e+04, 1.90268887e+04, 4.55625000e+04,
 4.25345820e+04, 6.63246172e+04, 1.16741711e+05, 1.69440984e+05,
 1.10486977e+05, 1.59707406e+05, 1.61631141e+05, 1.23861828e+05,
 1.50757328e+05, 1.31196297e+05, 5.50281289e+04, 6.28155391e+04,
 1.77474391e+05, 2.68843926e+04, 2.53033750e+04, 2.58044258e+04,
 1.34592480e+04, 3.88203076e+03, 9.22772559e+03, 1.22399551e+04,
 2.20868379e+04, 3.60825352e+04, 4.87962109e+04, 3.74652812e+04,
 4.10560898e+04, 4.17418750e+04, 1.10490836e+05, 2.00762031e+05,
 3.48149000e+05, 3.58060812e+05, 3.28375281e+05, 2.96505844e+05,
 2.33354141e+05, 1.54047203e+05, 1.43171328e+05, 1.62586469e+05,
 2.96452719e+05, 2.66880719e+05, 2.94189062e+05, 3.10067656e+05,
 6.27178750e+04, 8.63868672e+04, 9.69545391e+04, 2.01814359e+05,
 2.97053594e+05, 2.00674891e+05, 1.27305148e+05, 1.32509578e+05,
 8.46020391e+04, 1.74817453e+05, 2.95115844e+05, 4.75888719e+05,
 7.21894500e+05, 4.62711281e+05, 1.33154750e+05, 8.13640156e+04,
 3.43108672e+04, 3.45204180e+04, 5.36851641e+04, 7.42413281e+04,
 1.09339375e+05, 2.79682844e+05, 2.19119422e+05, 1.99934359e+05,
 1.65903391e+05, 8.16738047e+04, 4.82395195e+04, 1.65301000e+05,
 1.81699391e+05, 3.30645000e+05, 4.63625750e+05, 6.75044000e+05,
 3.34911500e+05, 1.69343750e+05, 1.03219234e+05, 1.48347609e+05,
 2.98684844e+05, 3.71952656e+05, 4.43306688e+05, 5.27904688e+05,
 1.12945275e+06, 1.00652938e+06, 8.03246875e+05, 2.21277969e+05,
 6.64886406e+04, 2.66182090e+04, 3.29688867e+04, 1.68663250e+05,
 1.76257219e+05, 2.99365562e+05, 4.31793438e+05, 1.91346712e+06,
 3.90353825e+06, 6.29502500e+06, 5.49042400e+06, 6.25676850e+06,
 6.28618000e+06, 7.23876400e+06, 2.08143525e+06, 6.66786500e+05,
 1.44639844e+05, 1.83413594e+05, 3.41026562e+05, 3.02069719e+05,
 3.71339750e+05, 2.02418219e+05, 1.36151141e+05, 8.24758203e+04,
 1.96006719e+05, 1.61595656e+05, 1.71401391e+05, 2.00606359e+05,
 2.37198891e+05, 1.58720172e+05, 6.36030352e+04, 3.38904031e+05,
 5.23524156e+05, 2.86212188e+05, 1.92326125e+05, 3.73765547e+04,
 1.31962000e+05, 9.33163047e+04, 1.18443672e+05, 3.00598719e+05,
 2.28996984e+05, 3.57954281e+05, 2.36219078e+05, 1.71530359e+05,
 2.68482539e+04, 4.79163125e+04, 4.00419297e+04, 5.41031875e+04,
 1.07641391e+05, 9.80020000e+04, 7.08350078e+04, 6.55985312e+04,
 8.53740000e+04, 1.33837578e+05, 2.72516312e+05, 6.54133500e+05,
 4.97212750e+05, 6.11190062e+05, 9.92907688e+05, 9.56692312e+05,
 1.09541412e+06, 1.19283125e+06, 3.75002219e+05, 2.22053797e+05,
 2.60620031e+05, 1.88109625e+05, 1.87927500e+05, 2.49058609e+05,
 3.79192656e+05, 4.07706781e+05, 6.24783062e+05, 9.62079375e+05,
 7.95344562e+05, 1.40557531e+05, 2.65227406e+05, 6.26269500e+05,
 2.05804234e+05, 6.53089570e+04, 1.33562969e+05, 1.14692070e+05,
 2.26913016e+05, 2.61226500e+05, 5.08516250e+05, 4.90744656e+05,
 2.38721828e+05, 6.85555703e+04, 7.30576641e+04, 5.36113750e+04,
 3.28223562e+05, 6.26447375e+05, 5.17735438e+05, 3.32536062e+05,
 1.72657266e+05, 8.28138125e+04, 1.43392656e+05, 8.83169844e+04,
 5.15846680e+04, 4.20798125e+04, 1.78049141e+05, 1.17351570e+05,
 1.16870422e+05, 1.02416836e+05, 1.67488781e+05, 2.05509953e+05,
 2.73221469e+05, 4.73881641e+04, 4.60577500e+04, 5.33725703e+04,
 7.87071094e+04, 7.06486719e+04, 5.23131758e+04, 2.50280723e+04,
 4.52024609e+04, 4.14732461e+04, 7.00107344e+04, 1.24132148e+05,
 1.34168141e+05, 3.38401562e+05, 9.75599844e+04, 1.10023000e+05,
 1.99194824e+04, 3.32738164e+04, 2.43711914e+04, 2.76617012e+04,
 2.70386660e+04, 1.58115801e+04, 1.16166348e+04, 2.74852148e+04,
 2.95266426e+04, 3.95874922e+04, 2.81846035e+04, 3.19155312e+04,
 5.60368477e+04, 3.98338398e+04, 5.02201602e+04, 3.19759512e+04,
 2.96142285e+04, 4.48333711e+04, 4.32265664e+04, 1.52425576e+04,
 2.86077324e+04, 4.23488125e+04, 7.25032578e+04, 1.21924688e+05,
 6.41490688e+05, 8.50461562e+05, 1.11135362e+06, 1.22003100e+06,
 1.54110162e+06, 2.03452138e+06, 1.76440575e+06, 4.22100781e+05,
 1.30513797e+05, 7.30581250e+04, 8.56110391e+04, 7.42401094e+04,
 1.49466766e+05, 1.25623496e+04, 2.13100293e+04, 9.81795703e+04,
 7.28018203e+04, 1.26064742e+05, 1.01337961e+05, 1.24553164e+05,
 7.13400000e+04, 8.41950469e+04, 6.25289375e+04, 8.18789453e+04,
 1.27303195e+05, 8.94661250e+04, 1.04100523e+05, 1.66147031e+05,
```



208.65266	,	233.53539	,	213.50632	,	276.5658	,
200.76715	,	164.6427	,	137.93799	,	213.45375	,
206.23912	,	257.53568	,	341.67487	,	411.6321	,
332.3958	,	399.6341	,	402.03375	,	351.9401	,
388.27478	,	362.2103	,	234.58073	,	250.63026	,
421.2771	,	163.96461	,	159.07036	,	160.63757	,
116.014	,	62.305946	,	96.06105	,	110.63433	,
148.61641	,	189.95403	,	220.89862	,	193.55951	,
202.62302	,	204.30829	,	332.4016	,	448.06476	,
590.04156	,	598.38184	,	573.0404	,	544.5235	,
483.0674	,	392.4885	,	378.37988	,	403.22012	,
544.47473	,	516.605	,	542.39197	,	556.8372	,
250.43535	,	293.91644	,	311.3752	,	449.23752	,
545.02625	,	447.9675	,	356.79846	,	364.01865	,
290.8643	,	418.11176	,	543.24567	,	689.84686	,
849.64374	,	680.2288	,	364.9038	,	285.2438	,
185.23193	,	185.7967	,	231.7006	,	272.47263	,
330.66504	,	528.85046	,	468.10193	,	447.1402	,
407.3124	,	285.7863	,	219.63496	,	406.57227	,
426.26212	,	575.0174	,	680.9007	,	821.6106	,
578.7154	,	411.51398	,	321.27753	,	385.1592	,
546.5207	,	609.8793	,	665.8128	,	726.57056	,
1062.7572	,	1003.25934	,	896.2404	,	470.40195	,
257.8539	,	163.15088	,	181.57336	,	410.6863	,
419.83	,	547.14307	,	657.1099	,	1383.2812	,
1975.7374	,	2508.9888	,	2343.1653	,	2501.3535	,
2507.2256	,	2690.495	,	1442.718	,	816.56995	,
380.3155	,	428.26813	,	583.9748	,	549.6087	,
609.3766	,	449.90912	,	368.98663	,	287.18607	,
442.72644	,	401.98962	,	414.00653	,	447.891	,
487.03067	,	398.39697	,	252.1964	,	582.15466	,
723.54974	,	534.98804	,	438.55002	,	193.33018	,
363.26575	,	305.47717	,	344.15646	,	548.26886	,
478.53625	,	598.29285	,	486.0237	,	414.16226	,
163.85437	,	218.89795	,	200.1048	,	232.60092	,
328.0875	,	313.0527	,	266.14847	,	256.1221	,
292.1883	,	365.8382	,	522.03094	,	808.7852	,
705.1331	,	781.7865	,	996.4475	,	978.1065	,
1046.6204	,	1092.1682	,	612.37427	,	471.22586	,
510.50952	,	433.7161	,	433.50607	,	499.05774	,
615.78625	,	638.5192	,	790.4321	,	980.85645	,
891.8209	,	374.91006	,	515.0024	,	791.3719	,
453.65652	,	255.55615	,	365.4627	,	338.66217	,
476.35388	,	511.10318	,	713.1032	,	700.5317	,
488.5917	,	261.8312	,	270.2918	,	231.54132	,
572.908	,	791.48425	,	719.5384	,	576.6594	,
415.52045	,	287.7739	,	378.6722	,	297.18173	,
227.1226	,	205.13364	,	421.95868	,	342.56616	,
341.86316	,	320.02628	,	409.25394	,	453.33206	,
522.70593	,	217.68823	,	214.61069	,	231.02502	,
280.54785	,	265.7982	,	228.72073	,	158.20264	,
212.60869	,	203.64981	,	264.5954	,	352.3239	,
366.2897	,	581.72296	,	312.34595	,	331.69714	,
141.1364	,	182.4111	,	156.11276	,	166.31807	,
164.43439	,	125.7441	,	107.780495	,	165.78665	,
171.83319	,	198.96605	,	167.8827	,	178.64919	,
236.72102	,	199.58418	,	224.09854	,	178.8182	,
172.08786	,	211.73892	,	207.91	,	123.46075	,
169.1382	,	205.78828	,	269.2643	,	349.1772	,
800.93115	,	922.2048	,	1054.2076	,	1104.5502	,
1241.4113	,	1426.3665	,	1328.3093	,	649.6928	,
361.26694	,	270.29266	,	292.59366	,	272.4704	,
386.6093	,	112.081894	,	145.97955	,	313.3362	,
269.8181	,	355.05597	,	318.33624	,	352.9209	,
267.0955	,	290.16382	,	250.05786	,	286.145	,
356.79572	,	299.1089	,	322.64612	,	407.6114	,
481.7431	,	664.8045	,	419.33078	,	420.05396	,
302.0214	,	409.84808	,	433.44717	,	328.0807	,
180.71472	,	231.3898	,	371.67667	,	573.2698	,
799.3082	,	1040.5736	,	1248.2089	,	1556.075	,
1445.6144	,	991.4733	,	653.7855	,	241.664	,
200.71255	,	229.69164	,	365.6423	,	491.61737	,

```

504.54974 , 467.4137 , 489.7394 , 415.83694 ,
307.593 , 285.44888 , 428.70956 , 584.49255 ,
516.80835 , 595.6676 , 411.8345 , 162.18913 ,
145.37505 , 334.1946 , 246.74936 , 215.40991 ,
264.1329 , 341.95923 , 380.3039 , 338.56595 ,
281.5662 , 238.5704 , 174.35591 , 307.85596 ,
345.94073 , 464.11203 , 561.7188 , 646.22894 ,
625.9026 , 532.09955 , 285.625 , 223.7945 ,
318.74115 , 248.5351 , 562.78455 , 759.4393 ,
860.8736 , 1154.201 , 1251.0643 , 1247.2102 ,
1298.1444 , 567.3183 , 251.13608 , 278.9937 ,
226.39412 , 338.29608 , 336.92746 , 265.14423 ,
217.05034 , 672.6335 , 882.671 , 855.88336 ,
1019.35803 , 1249.835 , 1280.4559 , 1314.4961 ,
632.8708 , 484.36414 , 420.38657 , 260.38715 ,
624.8056 , 990.02106 , 1017.5319 , 919.0592 ,
1178.533 , 1408.3364 , 1904.8239 , 1633.771 ,
1192.1284 , 674.242 , 604.9521 , 972.9281 ,
1204.9263 , 1363.4519 , 1378.1527 , 1350.4559 ,
1101.2997 , 703.3076 , 1223.3164 , 1425.1029 ,
1502.1896 , 459.54623 , 348.41965 , 551.64197 ,
843.7437 , 787.5989 , 897.4333 , 791.77203 ,
752.5888 , 653.84814 , 885.1505 , 1905.3235 ,
2860.1357 , 3281.406 , 3557.5776 , 3653.7703 ,
3660.019 , 2554.4297 , 1267.6624 , 925.6123 ,
1075.2522 , 1301.0342 , 1473.5092 , 1828.1636 ,
2801.7761 , 3292.0957 , 3251.5688 , 3187.5154 ,
3379.4363 , 3408.5151 , 4165.453 , 4234.4043 ,
5202.1772 , 6104.635 , 5456.073 , 4632.28 ,
4940.305 , 3949.767 , 3221.3562 , 3152.0356 ,
4678.5625 , 5873.6855 , 5220.106 , 1586.0177 ,
1453.9965 , 1284.9175 , 2963.2163 , 3771.7898 ,
3876.5652 , 3776.622 , 3339.4758 , 3624.7173 ,
3798.29 , 1451.92 , 1540.3851 , 1437.0847 ,
1229.9243 , 1279.0306 , 2451.6096 , 3849.44 ,
3891.8213 , 2981.3203 , 3219.5703 , 3524.72 ,
5588.383 , 6664.252 , 5793.246 , 5805.394 ,
6580.4614 , 6361.1904 , 8173.865 , 5178.2466 ,
1836.379 , 2486.362 , 1826.4835 , 3124.1047 ,
3083.0798 , 3850.9783 , 3287.689 , 3909.1228 ,
3482.797 , 3835.5054 , 6679.7974 , 7869.091 ,
10282.95 , 8978.149 , 5480.48 , 3616.3398 ,
2135.3894 , 1295.878 , 2644.6782 , 4043.3542 ,
3485.6704 , 3489.9646 , 4110.7144 , 4892.719 ,
6107.91 , 7138.7007 , 6778.814 , 5656.8877 ,
4542.8135 , 2853.957 , 2036.6534 , 2057.9094 ,
1760.0701 , 2894.6804 , 2713.1316 , 2657.2437 ,
3966.2427 , 5279.841 , 6382.3184 , 5731.68 ,
5376.0254 , 1844.5288 , 1960.1613 , 2847.4656 ,
4180.656 , 4624.117 , 2403.1375 , 2703.1477 ,
1925.0488 , 783.9162 , 1388.3987 , 1515.1669 ,
1044.1838 , 1092.8105 , 1419.6317 , 1177.2158 ,
1615.2874 , 2735.192 , 3545.8242 , 3862.4895 ,
2765.3428 , 3083.6038 , 3413.7578 , 4640.995 ,
6578.3354 , 8076.225 , 9644.3 , 9219.384 ,
7661.638 , 5861.165 , 5316.8203 , 3050.3152 ,
2160.8616 , 2828.763 , 4538.453 , 6381.4897 ,
4394.529 , 2010.854 , 2434.1501 , 2715.4133 ,
1638.5604 , 2297.3062 , 1359.4976 , 2207.3335 ,
2208.8198 , 2403.1946 , 3488.5847 , 5198.071 ,
7033.5254 , 8308.436 , 9222.554 ], dtype=float32) }

```

## Make our evaluation function work for larger horizons

You'll notice the outputs for `model_3_results` are multi-dimensional.

This is because the predictions are getting evaluated across the HORIZON timesteps (7 predictions at a time).

To fix this, let's adjust our `evaluate_preds()` function to work with multiple shapes of data.

In [ ]:

```
def evaluate_preds(y_true, y_pred):  
    # Make sure float32 (for metric calculations)  
    y_true = tf.cast(y_true, dtype=tf.float32)  
    y_pred = tf.cast(y_pred, dtype=tf.float32)  
  
    # Calculate various metrics  
    mae = tf.keras.metrics.mean_absolute_error(y_true, y_pred)  
    mse = tf.keras.metrics.mean_squared_error(y_true, y_pred)  
    rmse = tf.sqrt(mse)  
    mape = tf.keras.metrics.mean_absolute_percentage_error(y_true, y_pred)  
    mase = mean_absolute_scaled_error(y_true, y_pred)  
  
    # Account for different sized metrics (for longer horizons, reduce to single number)  
    if mae.ndim > 0: # if mae isn't already a scalar, reduce it to one by aggregating tensors to mean  
        mae = tf.reduce_mean(mae)  
        mse = tf.reduce_mean(mse)  
        rmse = tf.reduce_mean(rmse)  
        mape = tf.reduce_mean(mape)  
        mase = tf.reduce_mean(mase)  
  
    return {"mae": mae.numpy(),  
            "mse": mse.numpy(),  
            "rmse": rmse.numpy(),  
            "mape": mape.numpy(),  
            "mase": mase.numpy() }
```

Now we've updated `evaluate_preds()` to work with multiple shapes, how does it look?

In [ ]:

```
# Get model_3 results aggregated to single values  
model_3_results = evaluate_preds(y_true=tf.squeeze(test_labels),  
                                  y_pred=model_3_preds)  
model_3_results
```

Out[ ]:

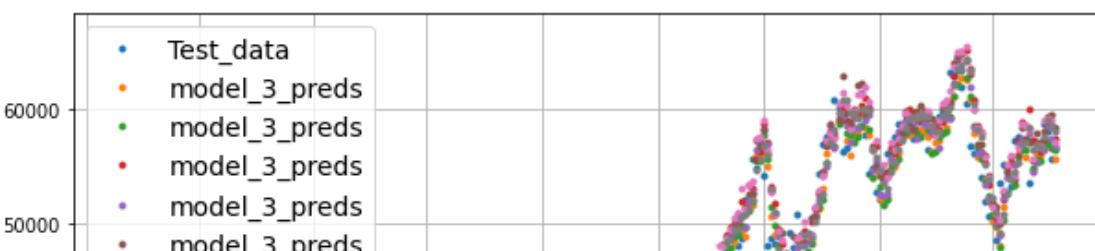
```
{'mae': 1237.5063,  
 'mape': 5.5588784,  
 'mase': 2.2020736,  
 'mse': 5405198.5,  
 'rmse': 1425.7477}
```

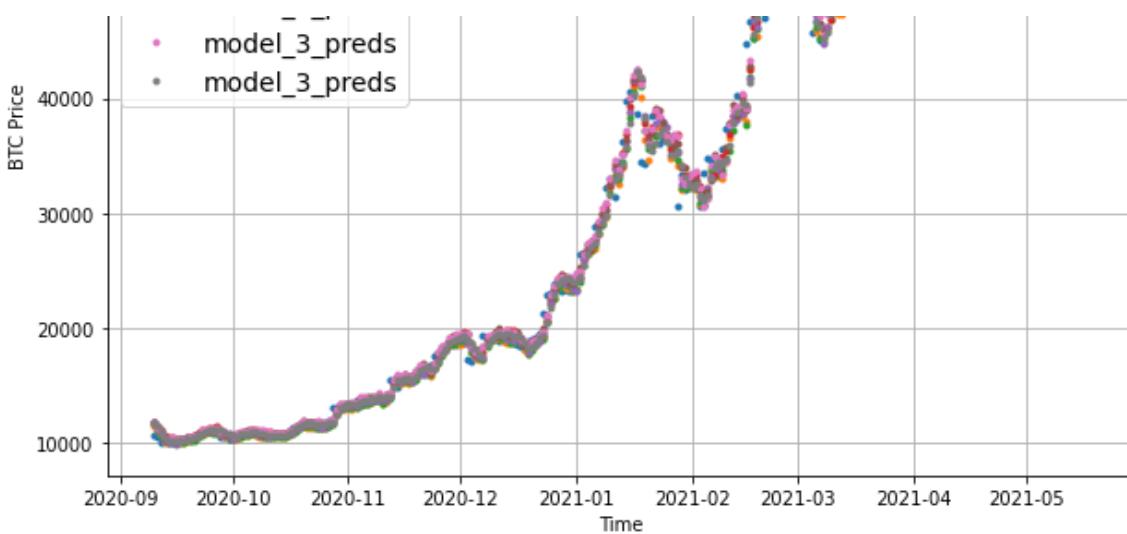
Time to visualize.

If our prediction evaluation metrics were multi-dimensional, how do you think the predictions will look like if we plot them?

In [ ]:

```
offset = 300  
plt.figure(figsize=(10, 7))  
plot_time_series(timesteps=X_test[-len(test_windows):], values=test_labels[:, 0], start=offset, label="Test_data")  
# Checking the shape of model_3_preds results in [n_test_samples, HORIZON] (this will screw up the plot)  
plot_time_series(timesteps=X_test[-len(test_windows):], values=model_3_preds, start=offset, label="model_3_preds")
```





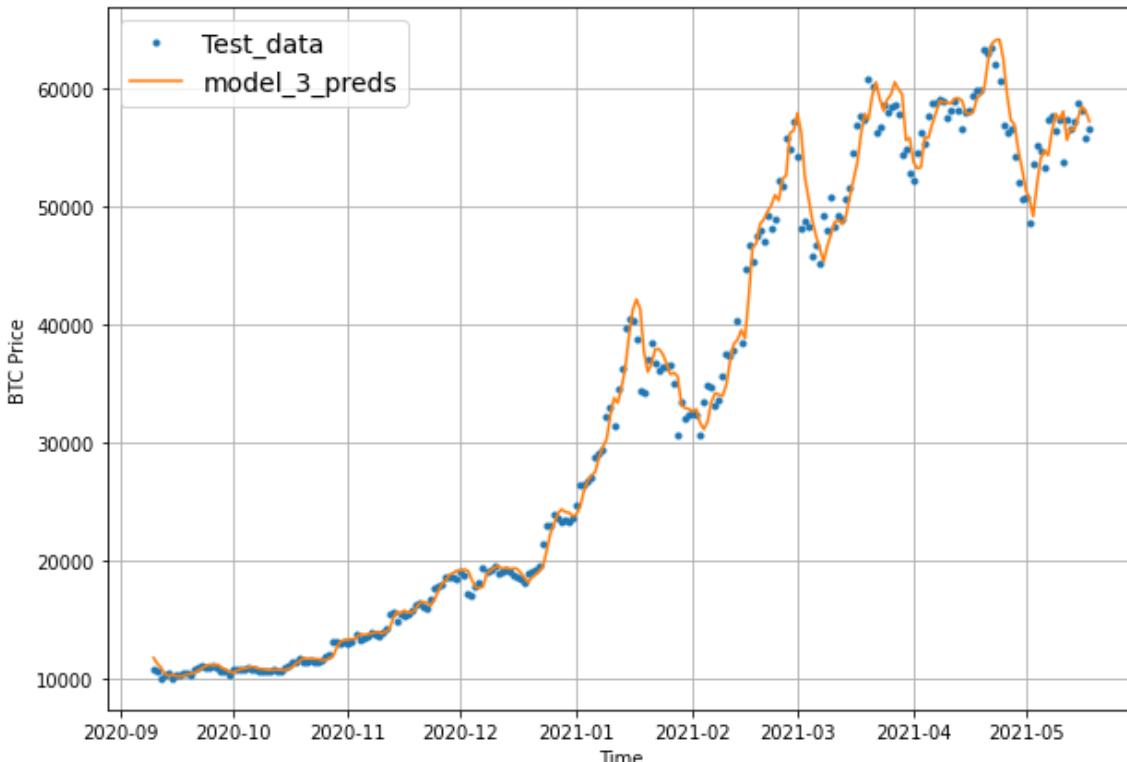
When we try to plot our multi-horizon predicts, we get a funky looking plot.

Again, we can fix this by aggregating our model's predictions.

**Note:** Aggregating the predictions (e.g. reducing a 7-day horizon to one value such as the mean) loses information from the original prediction. As in, the model predictions were trained to be made for 7-days but by reducing them to one, we gain the ability to plot them visually but we lose the extra information contained across multiple days.

In [ ]:

```
offset = 300
plt.figure(figsize=(10, 7))
# Plot model_3_preds by aggregating them (note: this condenses information so the preds will look further ahead than the test data)
plot_time_series(timesteps=X_test[-len(test_windows):],
                 values=test_labels[:, 0],
                 start=offset,
                 label="Test_data")
plot_time_series(timesteps=X_test[-len(test_windows):],
                 values=tf.reduce_mean(model_3_preds, axis=1),
                 format="-",
                 start=offset,
                 label="model_3_preds")
```



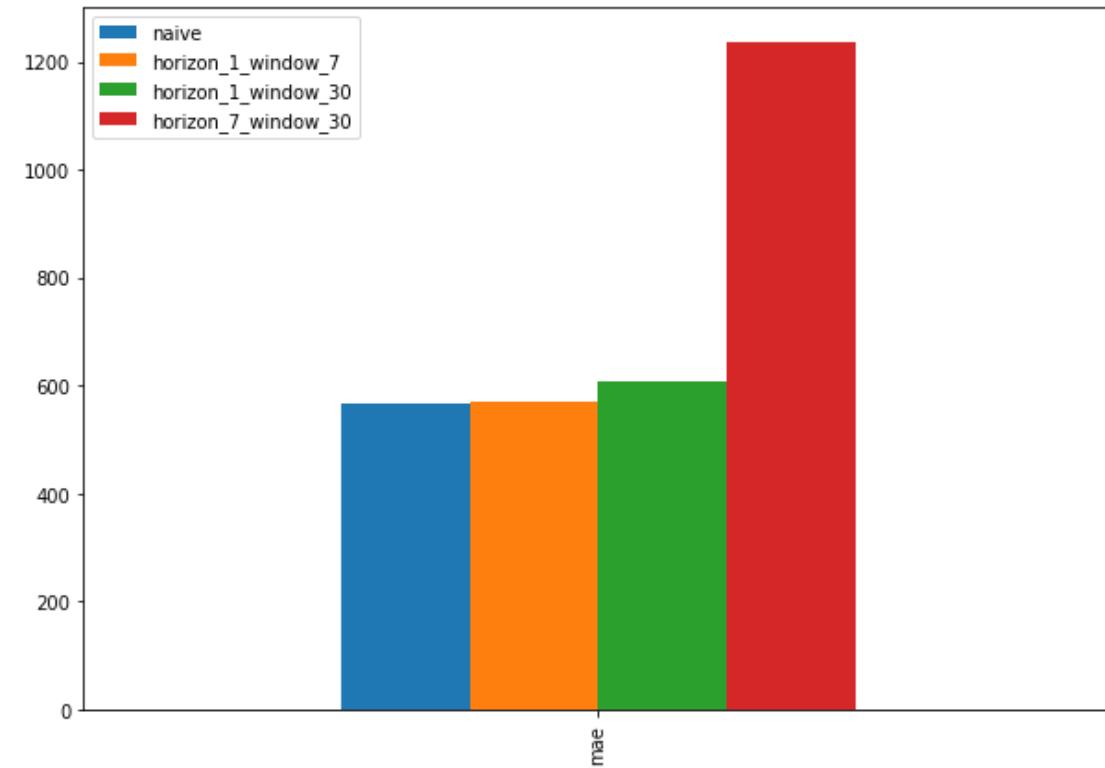
## Which of our models is performing best so far?

So far, we've trained 3 models which use the same architecture but use different data inputs.

Let's compare them with the naïve model to see which model is performing the best so far.

In [ ]:

```
pd.DataFrame({ "naive": naive_results["mae"],  
                "horizon_1_window_7": model_1_results["mae"],  
                "horizon_1_window_30": model_2_results["mae"],  
                "horizon_7_window_30": model_3_results["mae"] }, index=[ "mae" ]).plot(figsize=(10, 7), kind="bar");
```



Woah, our naïve model is performing best (it's very hard to beat a naïve model in open systems) but the dense model with a horizon of 1 and a window size of 7 looks to be performing closest.

Because of this, let's use `HORIZON=1` and `WINDOW_SIZE=7` for our next series of modelling experiments (in other words, we'll use the previous week of Bitcoin prices to try and predict the next day).

▀ **Note:** You might be wondering, why are the naïve results so good? One of the reasons could be due the presence of **autocorrelation** in the data. If a time series has **autocorrelation** it means the value at `t+1` (the next timestep) is typically close to the value at `t` (the current timestep). In other words, today's value is probably pretty close to yesterday's value. Of course, this isn't always the case but when it is, a naïve model will often get fairly good results.

▀ **Resource:** For more on how autocorrelation influences a model's predictions, see the article [How \(not\) to use Machine Learning for time series forecasting: Avoiding the pitfalls](#) by Vegard Fløvik

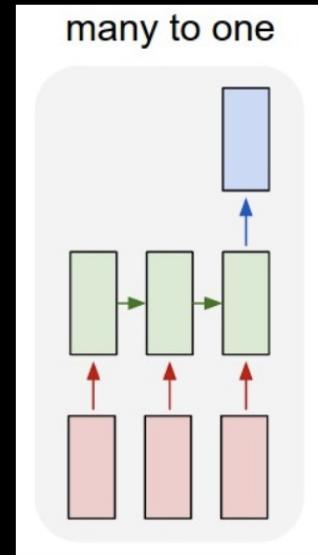
## Model 4: Conv1D

Onto the next modelling experiment!

This time, we'll be using a Conv1D model. Because as we saw in the sequence modelling notebook, Conv1D models can be used for seq2seq (sequence to sequence) problems.

In our case, the input sequence is the previous 7 days of Bitcoin price data and the output is the next day (in seq2seq terms this is called a many to one problem).

Output [123.033] Horizon = 1



Source: <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

Window size = 7

Input [123.654, 125.455, 108.584, 118.674, 121.338, 120.655, 121.795]

Framing Bitcoin forecasting in seq2seq (sequence to sequence) terms. Using a window size of 7 and a horizon of one results in a many to one problem. Using a window size of >1 and a horizon of >1 results in a many to many problem. The diagram comes from Andrei Karpathy's [The Unreasonable Effectiveness of Recurrent Neural Networks](#).

Before we build a Conv1D model, let's recreate our datasets.

In [ ]:

```
HORIZON = 1 # predict next day
WINDOW_SIZE = 7 # use previous week worth of data
```

In [ ]:

```
# Create windowed dataset
full_windows, full_labels = make_windows(prices, window_size=WINDOW_SIZE, horizon=HORIZON)
len(full_windows), len(full_labels)
```

Out[ ]:

```
(2780, 2780)
```

In [ ]:

```
# Create train/test splits
train_windows, test_windows, train_labels, test_labels = make_train_test_splits(full_windows, full_labels)
len(train_windows), len(test_windows), len(train_labels), len(test_labels)
```

Out[ ]:

```
(2224, 556, 2224, 556)
```

Data windowed!

Now, since we're going to be using [Conv1D layers](#), we need to make sure our input shapes are correct.

The Conv1D layer in TensorFlow takes an input of: (batch\_size, timesteps, input\_dim) .

In our case, the `batch_size` (by default this is 32 but we can change it) is handled for us but the other values will be:

- `timesteps = WINDOW_SIZE` - the `timesteps` is also often referred to as `features`, our features are the previous `WINDOW_SIZE` values of Bitcoin
- `input_dim = HORIZON` - our model views `WINDOW_SIZE` (one week) worth of data at a time to predict `HORIZON` (one day)

Right now, our data has the `timesteps` dimension ready but we'll have to adjust it to have the `input_dim` dimension.

In [ ]:

```
# Check data sample shapes
train_windows[0].shape # returns (WINDOW_SIZE, )
```

Out [ ]:

```
(7,)
```

To fix this, we could adjust the shape of all of our `train_windows` or we could use a `tf.keras.layers.Lambda` (called a Lambda layer) to do this for us in our model.

The Lambda layer wraps a function into a layer which can be used with a model.

Let's try it out.

In [ ]:

```
# Before we pass our data to the Conv1D layer, we have to reshape it in order to make sure it works
x = tf.constant(train_windows[0])
expand_dims_layer = layers.Lambda(lambda x: tf.expand_dims(x, axis=1)) # add an extra dimension for timesteps
print(f"Original shape: {x.shape}") # (WINDOW_SIZE)
print(f"Expanded shape: {expand_dims_layer(x).shape}") # (WINDOW_SIZE, input_dim)
print(f"Original values with expanded shape:\n {expand_dims_layer(x)}")
```

```
Original shape: (7,)
Expanded shape: (7, 1)
Original values with expanded shape:
[[123.65499]
 [125.455]
 [108.58483]
 [118.67466]
 [121.33866]
 [120.65533]
 [121.795]]
```

Looking good!

Now we've got a Lambda layer, let's build, compile, fit and evaluate a Conv1D model on our data.

**Note:** If you run the model below without the Lambda layer, you'll get an input shape error (one of the most common errors when building neural networks).

In [ ]:

```
tf.random.set_seed(42)

# Create model
model_4 = tf.keras.Sequential([
    # Create Lambda layer to reshape inputs, without this layer, the model will error
    layers.Lambda(lambda x: tf.expand_dims(x, axis=1)), # resize the inputs to adjust for
    window size / Conv1D 3D input requirements
```

```
layers.Conv1D(filters=128, kernel_size=5, padding="causal", activation="relu"),
    layers.Dense(HORIZON)
], name="model_4_conv1D")

# Compile model
model_4.compile(loss="mae",
                  optimizer=tf.keras.optimizers.Adam())

# Fit model
model_4.fit(train_windows,
            train_labels,
            batch_size=128,
            epochs=100,
            verbose=0,
            validation_data=(test_windows, test_labels),
            callbacks=[create_model_checkpoint(model_name=model_4.name)])
```

Out[ ]:

```
<keras.callbacks.History at 0x7fdcf1dfba50>
```

## What does the Lambda layer look like in a summary?

In [ ] :

```
model_4.summary()
```

Model: "model\_4\_conv1D"

Layer (type)	Output Shape	Param #
lambda_1 (Lambda)	(None, 1, 7)	0

conv1d (Conv1D)	(None, 1, 128)	4608
dense_6 (Dense)	(None, 1, 1)	129
<hr/>		
Total params: 4,737		
Trainable params: 4,737		
Non-trainable params: 0		

---

**The Lambda layer appears the same as any other regular layer.**

**Time to evaluate the Conv1D model.**

In [ ]:

```
# Load in best performing Conv1D model and evaluate it on the test data
model_4 = tf.keras.models.load_model("model_experiments/model_4_conv1D")
model_4.evaluate(test_windows, test_labels)
```

18/18 [=====] - 0s 3ms/step - loss: 570.8283

Out[ ]:

570.8283081054688

In [ ]:

```
# Make predictions
model_4_preds = make_preds(model_4, test_windows)
model_4_preds[:10]
```

Out[ ]:

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([8851.464, 8754.471, 8983.928, 8759.672, 8703.627, 8708.295,
       8661.667, 8494.839, 8435.317, 8492.115], dtype=float32)>
```

In [ ]:

```
# Evaluate predictions
model_4_results = evaluate_preds(y_true=tf.squeeze(test_labels),
                                  y_pred=model_4_preds)
model_4_results
```

Out[ ]:

```
{'mae': 570.8283,
'mape': 2.5593357,
'mase': 1.0027872,
'mse': 1176671.1,
'rmse': 1084.7448}
```

## Model 5: RNN (LSTM)

As you might've guessed, we can also use a recurrent neural network to model our sequential time series data.

**Resource:** For more on the different types of recurrent neural networks you can use for sequence problems, see the [Recurrent Neural Networks section of notebook 08](#).

Let's reuse the same data we used for the Conv1D model, except this time we'll create an [LSTM-cell](#) powered RNN to model our Bitcoin data.

Once again, one of the most important steps for the LSTM model will be getting our data into the right shape.

The `tf.keras.layers.LSTM()` layer takes a tensor with `[batch, timesteps, feature]` dimensions.

As mentioned earlier, the `batch` dimension gets taken care of for us but our data is currently only has the `feature` dimension (`WINDOW_SIZE`).

To fix this, just like we did with the `Conv1D` model, we can use a `tf.keras.layers.Lambda()` layer to adjust the shape of our input tensors to the LSTM layer.

In [ ]:

```
tf.random.set_seed(42)

# Let's build an LSTM model with the Functional API
inputs = layers.Input(shape=(WINDOW_SIZE))
x = layers.Lambda(lambda x: tf.expand_dims(x, axis=1))(inputs) # expand input dimension
# to be compatible with LSTM
# print(x.shape)
# x = layers.LSTM(128, activation="relu", return_sequences=True)(x) # this layer will err
# or if the inputs are not the right shape
x = layers.LSTM(128, activation="relu")(x) # using the tanh loss function results in a ma
ssive error
# print(x.shape)
# Add another optional dense layer (you could add more of these to see if they improve mo
del performance)
# x = layers.Dense(32, activation="relu")(x)
output = layers.Dense(HORIZON)(x)
model_5 = tf.keras.Model(inputs=inputs, outputs=output, name="model_5_lstm")

# Compile model
model_5.compile(loss="mae",
                  optimizer=tf.keras.optimizers.Adam())

# Seems when saving the model several warnings are appearing: https://github.com/tensorfl
ow/tensorflow/issues/47554
model_5.fit(train_windows,
            train_labels,
            epochs=100,
            verbose=0,
            batch_size=128,
            validation_data=(test_windows, test_labels),
            callbacks=[create_model_checkpoint(model_name=model_5.name)])
```

WARNING:tensorflow:Layer lstm will not use cuDNN kernels since it doesn't meet the criteria. It will use a generic GPU kernel as fallback when running on GPU.

```
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
```

Out [ ]:

```
<keras.callbacks.History at 0x7fdcf17b5190>
```

In [ ]:

```
# Load in best version of model 5 and evaluate on the test data
model_5 = tf.keras.models.load_model("model_experiments/model_5_lstm/")
model_5.evaluate(test_windows, test_labels)
```

```
WARNING:tensorflow:Layer lstm will not use cuDNN kernels since it doesn't meet the criteria. It will use a generic GPU kernel as fallback when running on GPU.
18/18 [=====] - 0s 2ms/step - loss: 596.6447
```

Out [ ]:

```
596.6446533203125
```

**Now we've got the best performing LSTM model loaded in, let's make predictions with it and evaluate them.**

In [ ]:

```
# Make predictions with our LSTM model
model_5_preds = make_preds(model_5, test_windows)
model_5_preds[:10]
```

Out [ ]:

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([8991.225, 8823.2 , 9009.359, 8847.859, 8742.254, 8788.655,
       8744.746, 8552.568, 8514.823, 8542.873], dtype=float32)>
```

In [ ]:

```
# Evaluate model 5 preds
model_5_results = evaluate_preds(y_true=tf.squeeze(test_labels),
                                   y_pred=model_5_preds)
model_5_results
```

Out [ ]:

```
{'mae': 596.64465,
 'mape': 2.6838453,
 'mase': 1.0481395,
 'mse': 1273486.9,
 'rmse': 1128.4888}
```

Hmmm... it seems even with an LSTM-powered RNN we weren't able to beat our naïve models results.

Perhaps adding another variable will help?

**▀ Note:** I'm putting this here again as a reminder that because neural networks are such powerful algorithms, they can be used for almost any problem, however, that doesn't mean they'll achieve performant or usable results. You're probably starting to clue onto this now.

## Make a multivariate time series

So far all of our models have barely kept up with the naïve forecast.

And so far all of them have been trained on a single variable (also called univariate time series): the historical price of Bitcoin.

If predicting the price of Bitcoin using the price of Bitcoin hasn't worked out very well, maybe giving our model more information may help.

More information is a vague term because we could actually feed almost anything to our model(s) and they would still try to find patterns.

For example, we could use the historical price of Bitcoin as well as anyone with the name [Daniel Bourke Tweeted](#) on that day to predict the future price of Bitcoin.

But would this help?

Porbably not.

What would be better is if we passed our model something related to Bitcoin (again, this is quite vauge, since in an open system like a market, you could argue everything is related).

This will be different for almost every time series you work on but in our case, we could try to see if the [Bitcoin block reward size](#) adds any predictive power to our model(s).

What is the Bitcoin block reward size?

The Bitcoin block reward size is the number of Bitcoin someone receives from mining a Bitcoin block.

At its inception, the Bitcoin block reward size was 50.

But every four years or so, the Bitcoin block reward halves.

For example, the block reward size went from 50 (starting January 2009) to 25 on November 28 2012.

Let's encode this information into our time series data and see if it helps a model's performance.

□ Note: Adding an extra feature to our dataset such as the Bitcoin block reward size will take our data from **univariate** (only the historical price of Bitcoin) to **multivariate** (the price of Bitcoin as well as the block reward size).

In [ ]:

```
# Let's make a multivariate time series
bitcoin_prices.head()
```

Out[ ]:

Date	Price
2013-10-01	123.65499
2013-10-02	125.45500
2013-10-03	108.58483
2013-10-04	118.67466
2013-10-05	121.33866

Alright, time to add another feature column, the block reward size.

First, we'll need to create variables for the different block reward sizes as well as the dates they came into play.

The following block rewards and dates were sourced from [cmcmarkets.com](#).

Block Reward	Start Date
50	3 January 2009 (2009-01-03)

Block Reward	Start Date
25	28 November 2012
12.5	9 July 2016
6.25	11 May 2020
3.125	TBA (expected 2024)
1.5625	TBA (expected 2028)

□ Note: Since our Bitcoin historical data starts from 01 October 2013, none of the timesteps in our multivariate time series will have a block reward of 50.

In [ ]:

```
# Block reward values
block_reward_1 = 50 # 3 January 2009 (2009-01-03) - this block reward isn't in our database
# (it starts from 01 October 2013)
block_reward_2 = 25 # 28 November 2012
block_reward_3 = 12.5 # 9 July 2016
block_reward_4 = 6.25 # 11 May 2020

# Block reward dates (datetime form of the above date stamps)
block_reward_2_datetime = np.datetime64("2012-11-28")
block_reward_3_datetime = np.datetime64("2016-07-09")
block_reward_4_datetime = np.datetime64("2020-05-11")
```

We're going to get the days (indexes) for different block reward values.

This is important because if we're going to use multiple variables for our time series, they have to be the same frequency as our original variable. For example, if our Bitcoin prices are daily, we need the block reward values to be daily as well.

□ Note: For using multiple variables, make sure they're the same frequency as each other. If your variables aren't at the same frequency (e.g. Bitcoin prices are daily but block rewards are weekly), you may need to transform them in a way that they can be used with your model.

In [ ]:

```
# Get date indexes for when to add in different block dates
block_reward_2_days = (block_reward_3_datetime - bitcoin_prices.index[0]).days
block_reward_3_days = (block_reward_4_datetime - bitcoin_prices.index[0]).days
block_reward_2_days, block_reward_3_days
```

Out [ ]:

(1012, 2414)

Now we can add another feature to our dataset `block_reward` (this gets lower over time so it may lead to increasing prices of Bitcoin).

In [ ]:

```
# Add block_reward column
bitcoin_prices_block = bitcoin_prices.copy()
bitcoin_prices_block["block_reward"] = None

# Set values of block_reward column (it's the last column hence -1 indexing on iloc)
bitcoin_prices_block.iloc[:block_reward_2_days, -1] = block_reward_2
bitcoin_prices_block.iloc[block_reward_2_days:block_reward_3_days, -1] = block_reward_3
bitcoin_prices_block.iloc[block_reward_3_days:, -1] = block_reward_4
bitcoin_prices_block.head()
```

Out [ ]:

Price	block_reward
1012	None
2414	None

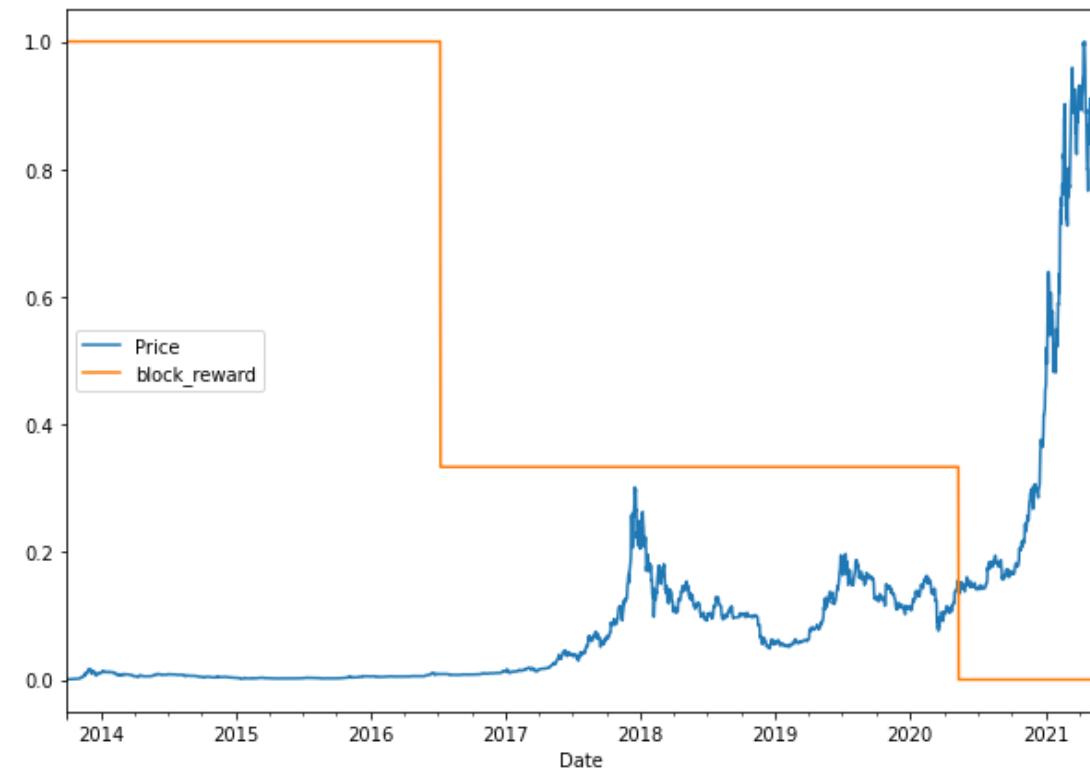
Date	Price	block_reward
2013-10-01	123.65499	25
2013-10-02	125.45500	25
2013-10-03	108.58483	25
2013-10-04	118.67466	25
2013-10-05	121.33866	25

Woohoo! We've officially added another variable to our time series data.

Let's see what it looks like.

In [ ]:

```
# Plot the block reward/price over time
# Note: Because of the different scales of our values we'll scale them to be between 0 and 1.
from sklearn.preprocessing import minmax_scale
scaled_price_block_df = pd.DataFrame(minmax_scale(bitcoin_prices_block[["Price", "block_reward"]]), # we need to scale the data first
                                      columns=bitcoin_prices_block.columns,
                                      index=bitcoin_prices_block.index)
scaled_price_block_df.plot(figsize=(10, 7));
```



When we scale the block reward and the Bitcoin price, we can see the price goes up as the block reward goes down, perhaps this information will be helpful to our model's performance.

## Making a windowed dataset with pandas

Previously, we used some custom made functions to window our univariate time series.

However, since we've just added another variable to our dataset, these functions won't work.

Not to worry though. Since our data is in a pandas DataFrame, we can leverage the `pandas.DataFrame.shift()` method to create a windowed multivariate time series.

The `shift()` method offsets an index by a specified number of periods.

Let's see it in action.

In [ ]:

```
# Setup dataset hyperparameters
HORIZON = 1
WINDOW_SIZE = 7
```

In [ ]:

```
# Make a copy of the Bitcoin historical data with block reward feature
bitcoin_prices_windowed = bitcoin_prices_block.copy()

# Add windowed columns
for i in range(WINDOW_SIZE): # Shift values for each step in WINDOW_SIZE
    bitcoin_prices_windowed[f"Price+{i+1}"] = bitcoin_prices_windowed["Price"].shift(periods=i+1)
bitcoin_prices_windowed.head(10)
```

Out[ ]:

Date	Price	block_reward	Price+1	Price+2	Price+3	Price+4	Price+5	Price+6	Price+7
2013-10-01	123.65499	25	NaN						
2013-10-02	125.45500	25	123.65499	NaN	NaN	NaN	NaN	NaN	NaN
2013-10-03	108.58483	25	125.45500	123.65499	NaN	NaN	NaN	NaN	NaN
2013-10-04	118.67466	25	108.58483	125.45500	123.65499	NaN	NaN	NaN	NaN
2013-10-05	121.33866	25	118.67466	108.58483	125.45500	123.65499	NaN	NaN	NaN
2013-10-06	120.65533	25	121.33866	118.67466	108.58483	125.45500	123.65499	NaN	NaN
2013-10-07	121.79500	25	120.65533	121.33866	118.67466	108.58483	125.45500	123.65499	NaN
2013-10-08	123.03300	25	121.79500	120.65533	121.33866	118.67466	108.58483	125.45500	123.65499
2013-10-09	124.04900	25	123.03300	121.79500	120.65533	121.33866	118.67466	108.58483	125.45500
2013-10-10	125.96116	25	124.04900	123.03300	121.79500	120.65533	121.33866	118.67466	108.58483

Now that we've got a windowed dataset, let's separate features ( `X` ) from labels ( `y` ).

Remember in our windowed dataset, we're trying to use the previous `WINDOW_SIZE` steps to predict `HORIZON` steps.

Window for a week (7) to predict a horizon of 1 (multivariate time series)  
`WINDOW_SIZE` & `block_reward` -> `HORIZON`

```
[0, 1, 2, 3, 4, 5, 6, block_reward] -> [7]
[1, 2, 3, 4, 5, 6, 7, block_reward] -> [8]
[2, 3, 4, 5, 6, 7, 8, block_reward] -> [9]
```

We'll also remove the `NaN` values using pandas `dropna()` method, this equivalent to starting our windowing function at sample 0 (the first sample) + `WINDOW_SIZE`.

In [ ]:

```
# Let's create X & y, remove the NaN's and convert to float32 to prevent TensorFlow errors
X = bitcoin_prices_windowed.dropna().drop("Price", axis=1).astype(np.float32)
y = bitcoin_prices_windowed.dropna()["Price"].astype(np.float32)
X.head()
```

Out[ ]:

Date	block_reward	Price+1	Price+2	Price+3	Price+4	Price+5	Price+6	Price+7
2013-10-08	25.0	121.794998	120.655327	121.338661	118.674660	108.584831	125.455002	123.654991

	block_reward	Price+1	Price+2	Price+3	Price+4	Price+5	Price+6	Price+7
Date	25.0	123.032997	121.794998	120.655327	121.338661	118.674660	108.584831	125.455002
2013-10-10	25.0	124.049004	123.032997	121.794998	120.655327	121.338661	118.674660	108.584831
2013-10-11	25.0	125.961159	124.049004	123.032997	121.794998	120.655327	121.338661	118.674660
2013-10-12	25.0	125.279663	125.961159	124.049004	123.032997	121.794998	120.655327	121.338661

In [ ]:

```
# View labels
y.head()
```

Out [ ]:

```
Date
2013-10-08    123.032997
2013-10-09    124.049004
2013-10-10    125.961159
2013-10-11    125.279663
2013-10-12    125.927498
Name: Price, dtype: float32
```

What a good looking dataset, let's split it into train and test sets using an 80/20 split just as we've done before.

In [ ]:

```
# Make train and test sets
split_size = int(len(X) * 0.8)
X_train, y_train = X[:split_size], y[:split_size]
X_test, y_test = X[split_size:], y[split_size:]
len(X_train), len(y_train), len(X_test), len(y_test)
```

Out [ ]:

```
(2224, 2224, 556, 556)
```

Training and test multivariate time series datasets made! Time to build a model.

## Model 6: Dense (multivariate time series)

To keep things simple, let's the `model_1` architecture and use it to train and make predictions on our multivariate time series data.

By replicating the `model_1` architecture we'll be able to see whether or not adding the block reward feature improves or detracts from model performance.

In [ ]:

```
tf.random.set_seed(42)

# Make multivariate time series model
model_6 = tf.keras.Sequential([
    layers.Dense(128, activation="relu"),
    # layers.Dense(128, activation="relu"), # adding an extra layer here should lead to beating the naive model
    layers.Dense(HORIZON)
], name="model_6_dense_multivariate")

# Compile
model_6.compile(loss="mae",
                  optimizer=tf.keras.optimizers.Adam())

# Fit
model_6.fit(X_train, y_train,
            epochs=100,
            batch_size=128,
            verbose=0, # only print 1 line per epoch
            validation_data=(X_test, y_test),
```

```
 callbacks=[create_model_checkpoint(model_name=model_6.name)])
```

Out [ ] :

```
<keras.callbacks.History at 0x7fdceed05590>
```

## Multivariate model fit!

You might've noticed that the model inferred the input shape of our data automatically (the data now has an extra feature). Often this will be the case, however, if you're running into shape issues, you can always explicitly define the input shape using `input_shape` parameter of the first layer in a model.

**Time to evaluate our multivariate model.**

In [ ]:

```
# Make sure best model is loaded and evaluate
model_6 = tf.keras.models.load_model("model_experiments/model_6_dense_multivariate")
model_6.evaluate(X_test, y_test)
```

18/18 [=====] - 0s 2ms/step - loss: 567.5873

Out[1]:

```
567.5873413085938

In [ ]:

# Make predictions on multivariate data
model_6_preds = tf.squeeze(model_6.predict(X_test))
```

model\_

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([8836.276, 8763.8 , 9040.486, 8741.225, 8719.326, 8765.071,
       8661.102, 8496.891, 8463.231, 8521.585], dtype=float32)>
```

In [ ]:

```
# Evaluate preds
model_6_results = evaluate_preds(y_true=y_test,
                                  y_pred=model_6_preds)
model_6_results
```

Out[ ]:

```
{'mae': 567.5874,
'mape': 2.541387,
'mase': 0.99709386,
'mse': 1161688.4,
'rmse': 1077.8165}
```

Hmmm... how do these results compare to `model_1` (same window size and horizon but without the block reward feature)?

In [ ]:

```
model_1_results
```

Out[ ]:

```
{'mae': 568.95123,
'mape': 2.5448983,
'mase': 0.9994897,
'mse': 1171744.0,
'rmse': 1082.4713}
```

It looks like the adding in the block reward may have helped our model slightly.

But there a few more things we could try.

□ **Resource:** For different ideas on how to improve a neural network model (from a model perspective), refer to the [Improving a model](#) section in notebook 02.

□ **Exercise(s):**

1. Try adding an extra `tf.keras.layers.Dense()` layer with 128 hidden units to `model_6`, how does this effect model performance?
2. Is there a better way to create this model? As in, should the `block_reward` feature be bundled in with the Bitcoin historical price feature? Perhaps you could test whether building a multi-input model (e.g. one model input for Bitcoin price history and one model input for `block_reward`) works better? See [Model 4: Hybrid embedding](#) section of notebook 09 for an idea on how to create a multi-input model.

## Model 7: N-BEATS algorithm

Time to step things up a notch.

So far we've tried a bunch of smaller models, models with only a couple of layers.

But one of the best ways to improve a model's performance is to increase the number of layers in it.

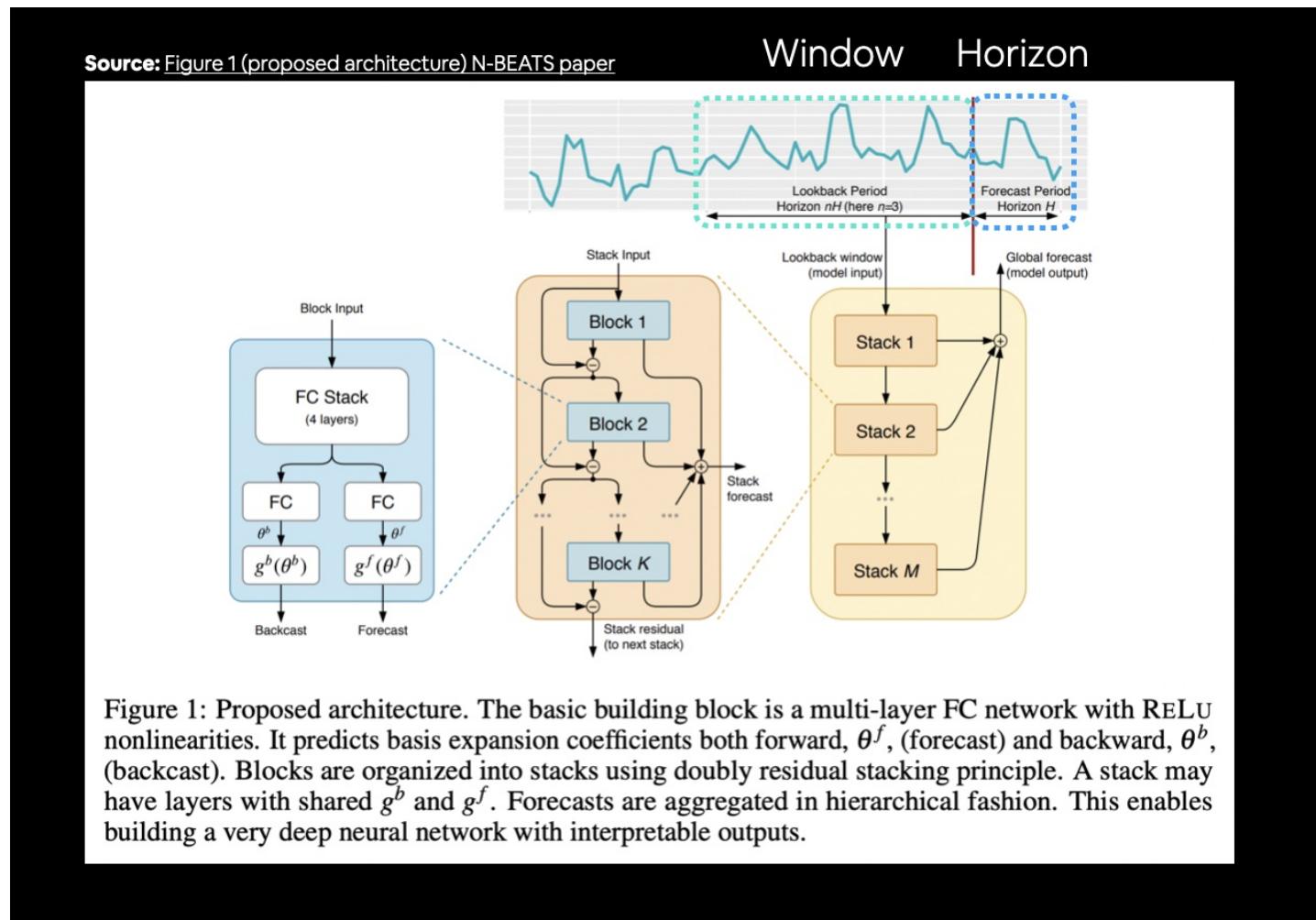
That's exactly what the [N-BEATS \(Neural Basis Expansion Analysis for Interpretable Time Series Forecasting\) algorithm](#) does.

The N-BEATS algorithm focuses on univariate time series problems and achieved state-of-the-art performance in the winner of the [M4 competition](#) (a forecasting competition).

For our next modelling experiment we're going to be replicating the **generic architecture of the N-BEATS algorithm** (see [section 3.3 of the N-BEATS paper](#)).

We're not going to go through all of the details in the paper, instead we're going to focus on:

### 1. Replicating the model architecture in [Figure 1 of the N-BEATS paper](#)



**N-BEATS algorithm** we're going to replicate with **TensorFlow** with window (input) and horizon (output) annotations.

### 1. Using the same hyperparameters as the paper which can be found in [Appendix D of the N-BEATS paper](#)

Doing this will give us an opportunity to practice:

- Creating a custom layer for the `NBeatsBlock` by subclassing `tf.keras.layers.Layer`
  - Creating a custom layer is helpful for when TensorFlow doesn't already have an existing implementation of a layer or if you'd like to make a layer configuration repeat a number of times (e.g. like a stack of N-BEATS blocks)
- Implementing a custom architecture using the Functional API
- Finding a paper related to our problem and seeing how it goes

□ Note: As you'll see in the paper, the authors state “N-BEATS is implemented and trained in TensorFlow”, that's what we'll be doing too!

## Building and testing an N-BEATS block layer

Let's start by building an N-BEATS block layer, we'll write the code first and then discuss what's going on.

In [ ]:

```
# Create NBeatsBlock custom layer
class NBeatsBlock(tf.keras.layers.Layer):
    def __init__(self, # the constructor takes all the hyperparameters for the layer
```

```

        input_size: int,
        theta_size: int,
        horizon: int,
        n_neurons: int,
        n_layers: int,
        **kwargs): # the **kwargs argument takes care of all of the arguments for
the parent class (input_shape, trainable, name)
super().__init__(**kwargs)
self.input_size = input_size
self.theta_size = theta_size
self.horizon = horizon
self.n_neurons = n_neurons
self.n_layers = n_layers

# Block contains stack of 4 fully connected layers each has ReLU activation
self.hidden = [tf.keras.layers.Dense(n_neurons, activation="relu") for _ in range(n_
layers)]
# Output of block is a theta layer with linear activation
self.theta_layer = tf.keras.layers.Dense(theta_size, activation="linear", name="theta"
a")

def call(self, inputs): # the call method is what runs when the layer is called
x = inputs
for layer in self.hidden: # pass inputs through each hidden layer
    x = layer(x)
theta = self.theta_layer(x)
# Output the backcast and forecast from theta
backcast, forecast = theta[:, :self.input_size], theta[:, -self.horizon:]
return backcast, forecast

```

## Setting up the `NBeatsBlock` custom layer we see:

- The class inherits from `tf.keras.layers.Layer` (this gives it all of the methods associated with `tf.keras.layers.Layer`)
- The constructor (`def __init__(...)`) takes all of the layer hyperparameters as well as the `**kwargs` argument
  - The `**kwargs` argument takes care of all of the hyperparameters which aren't mentioned in the constructor such as, `input_shape`, `trainable` and `name`
- In the constructor, the block architecture layers are created:
  - The hidden layers are created as a stack of fully connected with `n_neurons` hidden units layers with ReLU activation
  - The theta layer uses `theta_size` hidden units as well as linear activation
- The `call()` method is what is run when the layer is called:
  - It first passes the inputs (the historical Bitcoin data) through each of the hidden layers (a stack of fully connected layers with ReLU activation)
  - After the inputs have been through each of the fully connected layers, they get passed through the theta layer where the backcast (backwards predictions, shape: `input_size`) and forecast (forward predictions, shape: `horizon`) are returned

```

# Create NBeatsBlock custom layer
class NBeatsBlock(tf.keras.layers.Layer):
    def __init__(self, # the constructor takes all the hyperparameters for the layer
                 input_size: int,
                 theta_size: int,
                 horizon: int,
                 n_neurons: int,
                 n_layers: int,
                 **kwargs): # takes care of all of the arguments for the parent class (input_shape, trainable, name)
super().__init__(**kwargs)
self.input_size = input_size
self.theta_size = theta_size
self.horizon = horizon
self.n_neurons = n_neurons
self.n_layers = n_layers

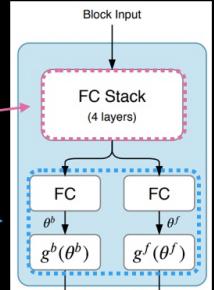
# Block contains stack of 4 fully connected layers each has ReLU activation
self.hidden = [tf.keras.layers.Dense(n_neurons, activation="relu") for _ in range(n_layers)]

# Output of block is a theta layer with linear activation
self.theta_layer = tf.keras.layers.Dense(theta_size, activation="linear", name="theta")

def call(self, inputs): # the call method is what runs when the layer is called
x = inputs
for layer in self.hidden: # pass inputs through each hidden layer

```

Source: Figure 1 (proposed architecture) N-BEATS paper



```

x = layer(x)
theta = self.theta_layer(x)

# Output the backcast and forecast from theta
backcast, forecast = theta[:, :self.input_size], theta[:, -self.horizon:]

return backcast, forecast

```



**Using TensorFlow layer subclassing to replicate the N-BEATS basic block.** See section 3.1 of N-BEATS paper for details.

Let's see our block replica in action by together by creating a toy version of `NBeatsBlock`.

**Resource:** Much of the creation of the time series materials (the ones you're going through now), including replicating the N-BEATS algorithm were streamed live on Twitch. If you'd like to see replays of how the algorithm was replicated, check out the [Time series research and TensorFlow course material creation playlist](#) on the Daniel Bourke arXiv YouTube channel.

In [ ]:

```

# Set up dummy NBeatsBlock layer to represent inputs and outputs
dummy_nbeats_block_layer = NBeatsBlock(input_size=WINDOW_SIZE,
                                         theta_size=WINDOW_SIZE+HORIZON, # backcast + forecast
                                         horizon=HORIZON,
                                         n_neurons=128,
                                         n_layers=4)

```

In [ ]:

```

# Create dummy inputs (have to be same size as input_size)
dummy_inputs = tf.expand_dims(tf.range(WINDOW_SIZE) + 1, axis=0) # input shape to the model has to reflect Dense layer input requirements (ndim=2)
dummy_inputs

```

Out[ ]:

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

In [ ]:

```

# Pass dummy inputs to dummy NBeatsBlock layer
backcast, forecast = dummy_nbeats_block_layer(dummy_inputs)
# These are the activation outputs of the theta layer (they'll be random due to no training of the model)
print(f"Backcast: {tf.squeeze(backcast.numpy())}")
print(f"Forecast: {tf.squeeze(forecast.numpy())}")

```

```

Backcast: [ 0.19014978  0.83798355 -0.32870018  0.25159916 -0.47540277 -0.77836645
           -0.5299447 ]
Forecast: -0.7554212808609009

```

## Preparing data for the N-BEATS algorithm using `tf.data`

We've got the basic building block for the N-BEATS architecture ready to go.

But before we use it to replicate the entire N-BEATS generic architecture, let's create some data.

This time, because we're going to be using a larger model architecture, to ensure our model training runs as fast as possible, we'll setup our datasets using the `tf.data` API.

And because the N-BEATS algorithm is focused on univariate time series, we'll start by making training and test windowed datasets of Bitcoin prices (just as we've done above).

In [ ]:

```

HORIZON = 1 # how far to predict forward
WINDOW_SIZE = 7 # how far to lookback

```

In [ ]:

```
# Create NBEATS data inputs (NBEATS works with univariate time series)
bitcoin_prices.head()
```

Out[ ]:

Date	Price
2013-10-01	123.65499
2013-10-02	125.45500
2013-10-03	108.58483
2013-10-04	118.67466
2013-10-05	121.33866

In [ ]:

```
# Add windowed columns
bitcoin_prices_nbeats = bitcoin_prices.copy()
for i in range(WINDOW_SIZE):
    bitcoin_prices_nbeats[f"Price+{i+1}"] = bitcoin_prices_nbeats["Price"].shift(periods=i+1)
bitcoin_prices_nbeats.dropna().head()
```

Out[ ]:

Date	Price	Price+1	Price+2	Price+3	Price+4	Price+5	Price+6	Price+7
2013-10-08	123.03300	121.79500	120.65533	121.33866	118.67466	108.58483	125.45500	123.65499
2013-10-09	124.04900	123.03300	121.79500	120.65533	121.33866	118.67466	108.58483	125.45500
2013-10-10	125.96116	124.04900	123.03300	121.79500	120.65533	121.33866	118.67466	108.58483
2013-10-11	125.27966	125.96116	124.04900	123.03300	121.79500	120.65533	121.33866	118.67466
2013-10-12	125.92750	125.27966	125.96116	124.04900	123.03300	121.79500	120.65533	121.33866

In [ ]:

```
# Make features and labels
X = bitcoin_prices_nbeats.dropna().drop("Price", axis=1)
y = bitcoin_prices_nbeats.dropna()["Price"]

# Make train and test sets
split_size = int(len(X) * 0.8)
X_train, y_train = X[:split_size], y[:split_size]
X_test, y_test = X[split_size:], y[split_size:]
len(X_train), len(y_train), len(X_test), len(y_test)
```

Out[ ]:

(2224, 2224, 556, 556)

**Train and test sets ready to go!**

Now let's convert them into TensorFlow `tf.data.Dataset`'s to ensure they run as fast as possible whilst training.

We'll do this by:

1. Turning the arrays in tensor Datasets using `tf.data.Dataset.from_tensor_slices()`

- Note: `from_tensor_slices()` works best when your data fits in memory, for extremely large datasets, you'll want to look into using the [TFRecord format](#)

2. Combining the labels and features into a Dataset using `tf.data.Dataset.zip()`

2. Combine the labels and features tensors into a Dataset using `tf.data.Dataset.zip()`

3. Batch and prefetch the Datasets using `batch()` and `prefetch()`

- Batching and prefetching ensures the loading time from CPU (preparing data) to GPU (computing on data) is as small as possible

Resource: For more on building highly performant TensorFlow data pipelines, I'd recommend reading through the [Better performance with the tf.data API](#) guide.

In [ ]:

```
# 1. Turn train and test arrays into tensor Datasets
train_features_dataset = tf.data.Dataset.from_tensor_slices(X_train)
train_labels_dataset = tf.data.Dataset.from_tensor_slices(y_train)

test_features_dataset = tf.data.Dataset.from_tensor_slices(X_test)
test_labels_dataset = tf.data.Dataset.from_tensor_slices(y_test)

# 2. Combine features & labels
train_dataset = tf.data.Dataset.zip((train_features_dataset, train_labels_dataset))
test_dataset = tf.data.Dataset.zip((test_features_dataset, test_labels_dataset))

# 3. Batch and prefetch for optimal performance
BATCH_SIZE = 1024 # taken from Appendix D in N-BEATS paper
train_dataset = train_dataset.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
test_dataset = test_dataset.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)

train_dataset, test_dataset
```

Out [ ]:

```
(<PrefetchDataset shapes: ((None, 7), (None,)), types: (tf.float64, tf.float64)>,
 <PrefetchDataset shapes: ((None, 7), (None,)), types: (tf.float64, tf.float64)>)
```

Data prepared! Notice the input shape for the features `(None, 7)`, the `None` leaves space for the batch size where as the `7` represents the `WINDOW_SIZE`.

Time to get create the N-BEATS architecture.

## Setting up hyperparameters for N-BEATS algorithm

Ho ho, would you look at that! Datasets ready, model building block ready, what'd you say we put things together?

Good idea.

Okay.

Let's go.

To begin, we'll create variables for each of the hyperparameters we'll be using for our N-BEATS replica.

Resource: The following hyperparameters are taken from Figure 1 and Table 18/Appendix D of the [N-BEATS paper](#).

Table 18: Settings of hyperparameters across subsets of M4, M3, TOURISM datasets.

Parameter	M4						M3				TOURISM		
	Yly	Qly	Mly	Wly	Dly	Hly	Yly	Qly	Mly	Other	Yly	Qly	Mly
N-BEATS-I													
$L_H$	1.5	1.5	1.5	10	10	10	20	5	5	20	20	10	20
Iterations	15K	15K	15K	5K	5K	5K	50	6K	6K	250	30	500	300
Losses	sMAPE/MAPE/MASE						sMAPE/MAPE/MASE				MAPE		

S-width	2048
S-blocks	3
S-block-layers	4
T-width	256
T-degree	2
T-blocks	3
T-block-layers	4
Sharing	STACK LEVEL
Lookback period	2H, 3H, 4H, 5H, 6H, 7H
Batch	1024

Parameter	N-BEATS-G												
	$L_H$	1.5	1.5	1.5	10	10	10	20	20	20	10	5	10
Iterations	15K	15K	15K	5K	5K	5K	20	250	10K	250	30	100	100
Losses	SMAPE/MAPE/MASE						SMAPE/MAPE/MASE						MAPE
Width							512						
Blocks							1						
Block-layers							4						
Stacks							30						
Sharing							NO						
Lookback period							2H, 3H, 4H, 5H, 6H, 7H						
Batch							1024						

Table 18 from [N-BEATS paper](#) describing the hyperparameters used for the different variants of N-BEATS. We're using N-BEATS-G which stands for the generic version of N-BEATS.

□ Note: If you see variables in a machine learning example in all caps, such as " `N_EPOCHS = 100` ", these variables are often hyperparameters which are used through the example. You'll usually see them instantiated towards the start of an experiment and then used throughout.

In [ ]:

```
# Values from N-BEATS paper Figure 1 and Table 18/Appendix D
N_EPOCHS = 5000 # called "Iterations" in Table 18
N_NEURONS = 512 # called "Width" in Table 18
N_LAYERS = 4
N_STACKS = 30

INPUT_SIZE = WINDOW_SIZE * HORIZON # called "Lookback" in Table 18
THETA_SIZE = INPUT_SIZE + HORIZON

INPUT_SIZE, THETA_SIZE
```

Out[ ]:

(7, 8)

## Getting ready for residual connections

Beautiful! Hyperparameters ready, now before we create the N-BEATS model, there are two layers to go through which play a large roll in the architecture.

They're what make N-BEATS double residual stacking (section 3.2 of the [N-BEATS paper](#)) possible:

- `tf.keras.layers.subtract(inputs)` - subtracts list of input tensors from each other
- `tf.keras.layers.add(inputs)` - adds list of input tensors to each other

Let's try them out.

In [ ]:

```
# Make tensors
tensor_1 = tf.range(10) + 10
tensor_2 = tf.range(10)

# Subtract
```

```

subtracted = layers.subtract([tensor_1, tensor_2])

# Add
added = layers.add([tensor_1, tensor_2])

print(f"Input tensors: {tensor_1.numpy()} & {tensor_2.numpy()}")
print(f"Subtracted: {subtracted.numpy()}")
print(f"Added: {added.numpy()}")

```

Input tensors: [10 11 12 13 14 15 16 17 18 19] & [0 1 2 3 4 5 6 7 8 9]  
Subtracted: [10 10 10 10 10 10 10 10 10 10]  
Added: [10 12 14 16 18 20 22 24 26 28]

**Both of these layer functions are straight-forward, subtract or add together their inputs.**

And as mentioned before, they're what powers N-BEATS double residual stacking.

The power of residual stacking or residual connections was revealed in [Deep Residual Learning for Image Recognition](#) where the authors were able to build a deeper but less complex neural network (this is what introduced the popular [ResNet architecture](#)) than previous attempts.

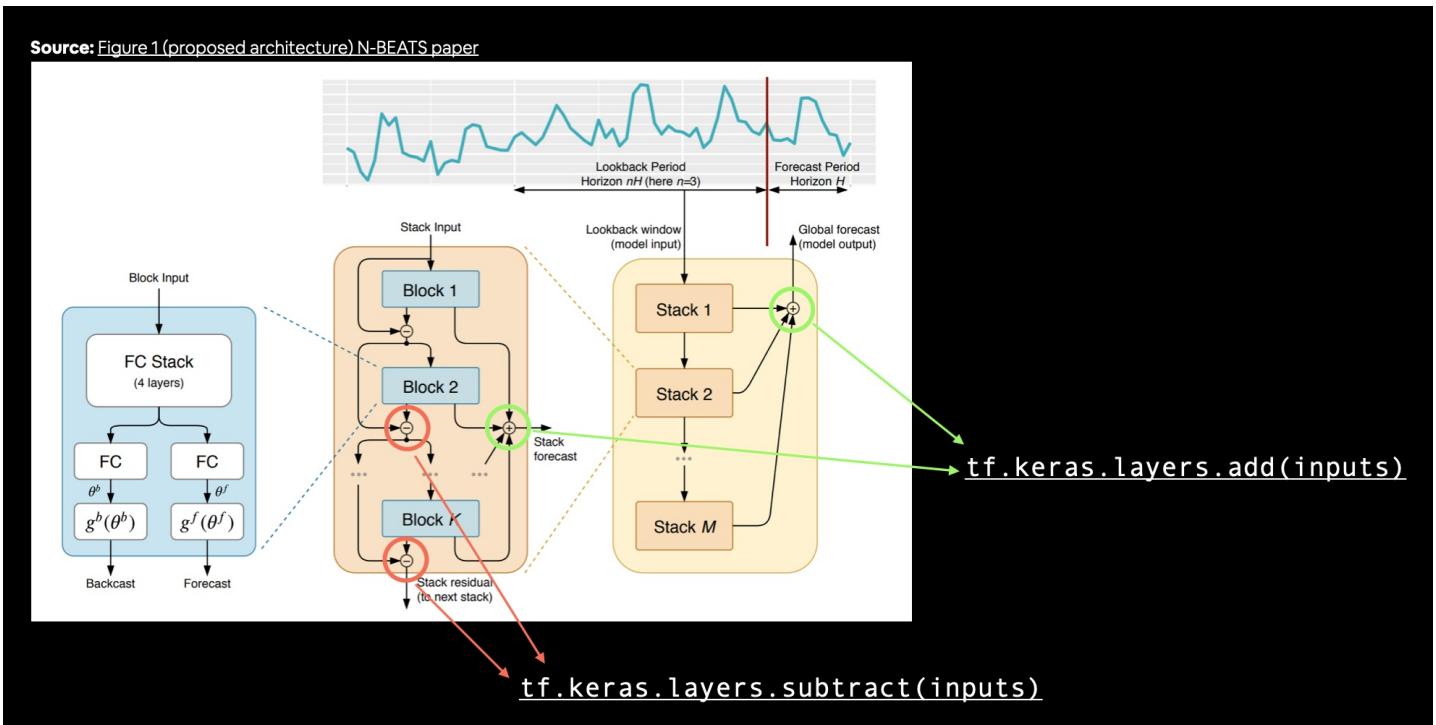
This deeper neural network led to state of the art results on the ImageNet challenge in 2015 and different versions of residual connections have been present in deep learning ever since.

### What is a residual connection?

A **residual connection** (also called skip connections) involves a deeper neural network layer receiving the outputs as well as the inputs of a shallower neural network layer.

In the case of N-BEATS, the architecture uses residual connections which:

- Subtract the backcast outputs from a previous block from the backcast inputs to the current block
- Add the forecast outputs from all blocks together in a stack



Annotated version of Figure 1 from the N-BEATS paper highlighting the double residual stacking (section 3.2) of the architecture. Backcast residuals of each block are subtracted from each other and used as the input to the next block where as the forecasts of each block are added together to become the stack forecast.

### What are the benefits of residual connections?

In practice, residual connections have been beneficial for training deeper models (N-BEATS reaches ~150 layers, also see "These approaches provide clear advantages in improving the trainability of deep architectures" in section 3.2 of the [N-BEATS paper](#)).

It's thought that they help avoid the problem of [vanishing gradients](#) (patterns learned by a neural network not being passed through to deeper layers).

## Building, compiling and fitting the N-BEATS algorithm

Okay, we've finally got all of the pieces of the puzzle ready for building and training the N-BEATS algorithm.

We'll do so by going through the following:

1. Setup an instance of the N-BEATS block layer using `NBeatsBlock` (this'll be the initial block used for the network, the rest will be created as part of stacks)
2. Create an input layer for the N-BEATS stack (we'll be using the [Keras Functional API](#) for this)
3. Make the initial backcast and forecasts for the model with the layer created in (1)
4. Use a for loop to create stacks of block layers
5. Use the `NBeatsBlock` class within the for loop created in (4) to create blocks which return backcasts and block-level forecasts
6. Create the double residual stacking using subtract and add layers
7. Put the model inputs and outputs together using `tf.keras.Model()`
8. Compile the model with MAE loss (the paper uses multiple losses but we'll use MAE to keep it inline with our other models) and Adam optimizer with default settings as per section 5.2 of [N-BEATS paper](#))
9. Fit the N-BEATS model for 5000 epochs and since it's fitting for so many epochs, we'll use a couple of callbacks:
  - `tf.keras.callbacks.EarlyStopping()` - stop the model from training if it doesn't improve validation loss for 200 epochs and restore the best performing weights using `restore_best_weights=True` (this'll prevent the model from training for loooongggggg period of time without improvement)
  - `tf.keras.callbacks.ReduceLROnPlateau()` - if the model's validation loss doesn't improve for 100 epochs, reduce the learning rate by 10x to try and help it make incremental improvements (the smaller the learning rate, the smaller updates a model tries to make)

Woah. A bunch of steps. But I'm sure you're up to it.

Let's do it!

In [ ]:

```
%%time

tf.random.set_seed(42)

# 1. Setup N-BEATS Block layer
nbeats_block_layer = NBeatsBlock(input_size=INPUT_SIZE,
                                  theta_size=THETA_SIZE,
                                  horizon=HORIZON,
                                  n_neurons=N_NEURONS,
                                  n_layers=N_LAYERS,
                                  name="InitialBlock")

# 2. Create input to stacks
stack_input = layers.Input(shape=(INPUT_SIZE), name="stack_input")

# 3. Create initial backcast and forecast input (backwards predictions are referred to as residuals in the paper)
backcast, forecast = nbeats_block_layer(stack_input)
# Add in subtraction residual link, thank you to: https://github.com/mrdbourke/tensorflow-deep-learning/discussions/174
residuals = layers.subtract([stack_input, backcast], name=f"subtract_00")

# 4. Create stacks of blocks
for i, _ in enumerate(range(N_STACKS-1)): # first stack is already created in (3)

    # 5. Use the NBeatsBlock to calculate the backcast as well as block forecast
    backcast, block_forecast = NBeatsBlock(
        input_size=INPUT_SIZE,
        theta_size=THETA_SIZE,
```

```

horizon=HORIZON,
n_neurons=N_NEURONS,
n_layers=N_LAYERS,
name=f"NBeatsBlock_{i}"
)(residuals) # pass it in residuals (the backcast)

# 6. Create the double residual stacking
residuals = layers.subtract([residuals, backcast], name=f"subtract_{i}")
forecast = layers.add([forecast, block_forecast], name=f"add_{i}")

# 7. Put the stack model together
model_7 = tf.keras.Model(inputs=stack_input,
                         outputs=forecast,
                         name="model_7_N-BEATS")

# 8. Compile with MAE loss and Adam optimizer
model_7.compile(loss="mae",
                  optimizer=tf.keras.optimizers.Adam(0.001),
                  metrics=["mae", "mse"])

# 9. Fit the model with EarlyStopping and ReduceLROnPlateau callbacks
model_7.fit(train_dataset,
            epochs=N_EPOCHS,
            validation_data=test_dataset,
            verbose=0, # prevent large amounts of training outputs
            # callbacks=[create_model_checkpoint(model_name=stack_model.name)] # saving
model every epoch consumes far too much time
            callbacks=[tf.keras.callbacks.EarlyStopping(monitor="val_loss", patience=200
, restore_best_weights=True),
                      tf.keras.callbacks.ReduceLROnPlateau(monitor="val_loss", patience=
100, verbose=1)])

```

Epoch 00328: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.

Epoch 00428: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.

CPU times: user 1min 23s, sys: 4.58 s, total: 1min 28s

Wall time: 3min 44s

And would you look at that! N-BEATS algorithm fit to our Bitcoin historical data.

## How did it perform?

In [ ]:

```
# Evaluate N-BEATS model on the test dataset  
model 7.evaluate(test dataset)
```

1/1 [=====] - 0s 46ms/step - loss: 585.4998 - mae: 585.4998 - ms  
e: 1179491.5000

Out[ ]:

[585.4998168945312, 585.4998168945312, 1179491.5]

In [ ]:

```
# Make predictions with N-BEATS model
model_7_preds = make_preds(model_7, test_dataset)
model_7_preds[:10]
```

Out [ ] :

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([8908.059, 8854.672, 8990.933, 8759.821, 8819.711, 8774.012,
       8604.187, 8547.038, 8495.928, 8489.514], dtype=float32)>
```

In [ ]:

Out [ ]:

```
{'mae': 585.4998,  
'mape': 2.7445195,  
'mase': 1.028561,  
'mse': 1179491.5,  
'rmse': 1086.044}
```

Woah... even with all of those special layers and hand-crafted network, it looks like the N-BEATS model doesn't perform as well as `model_1` or the original naive forecast.

This goes to show the power of smaller networks as well as the fact not all larger models are better suited for a certain type of data.

## Plotting the N-BEATS architecture we've created

You know what would be cool?

If we could plot the N-BEATS model we've crafted.

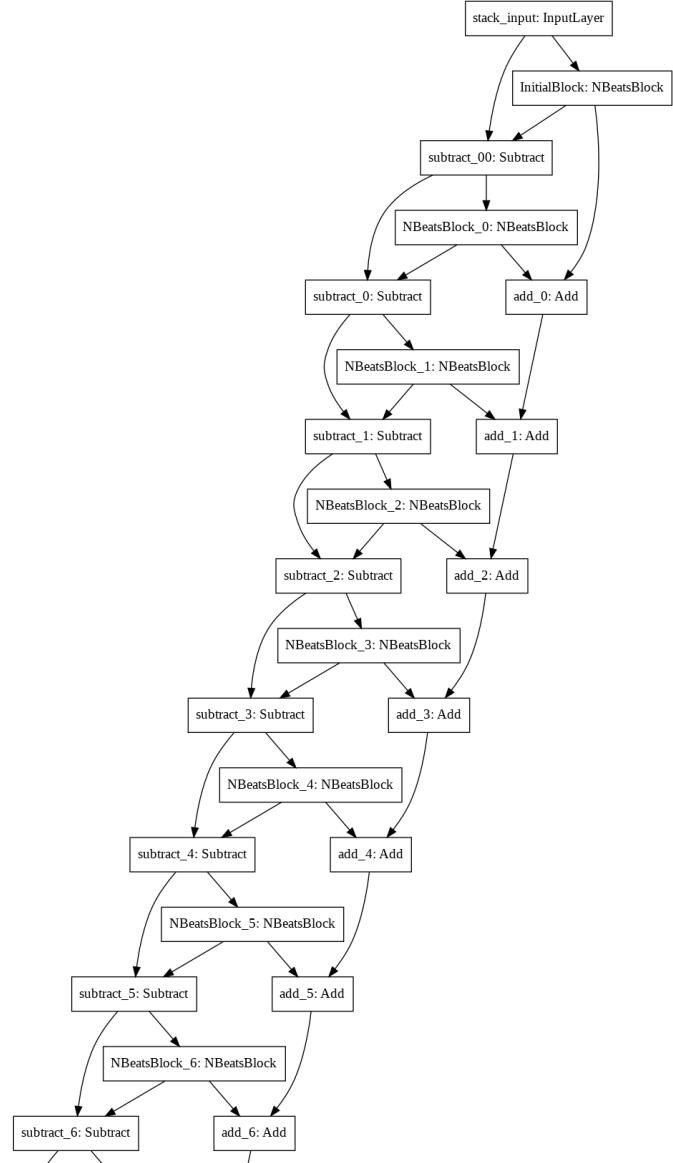
Well it turns out we can using `tensorflow.keras.utils.plot_model()`.

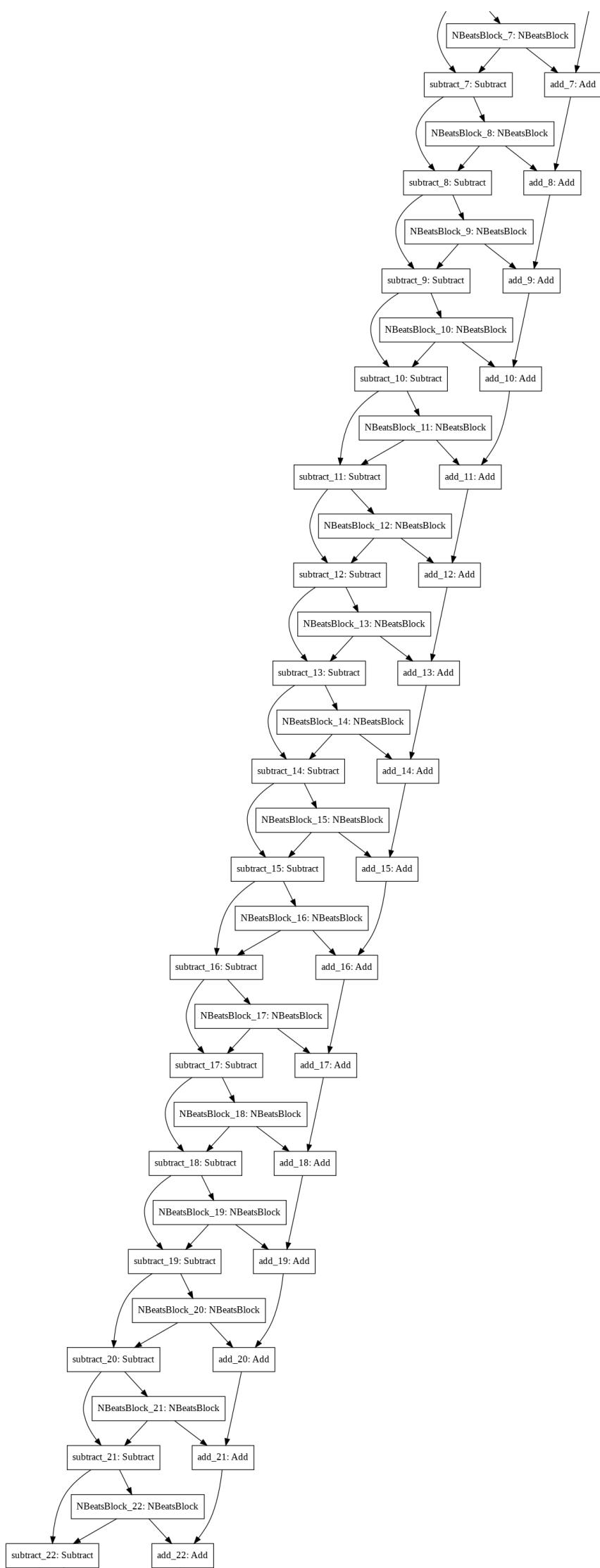
Let's see what it looks like.

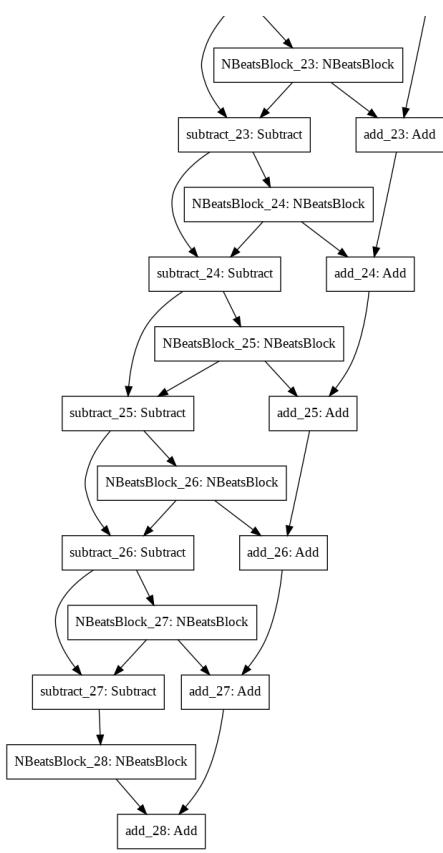
In [ ]:

```
# Plot the N-BEATS model and inspect the architecture  
from tensorflow.keras.utils import plot_model  
plot_model(model_7)
```

Out [ ]:

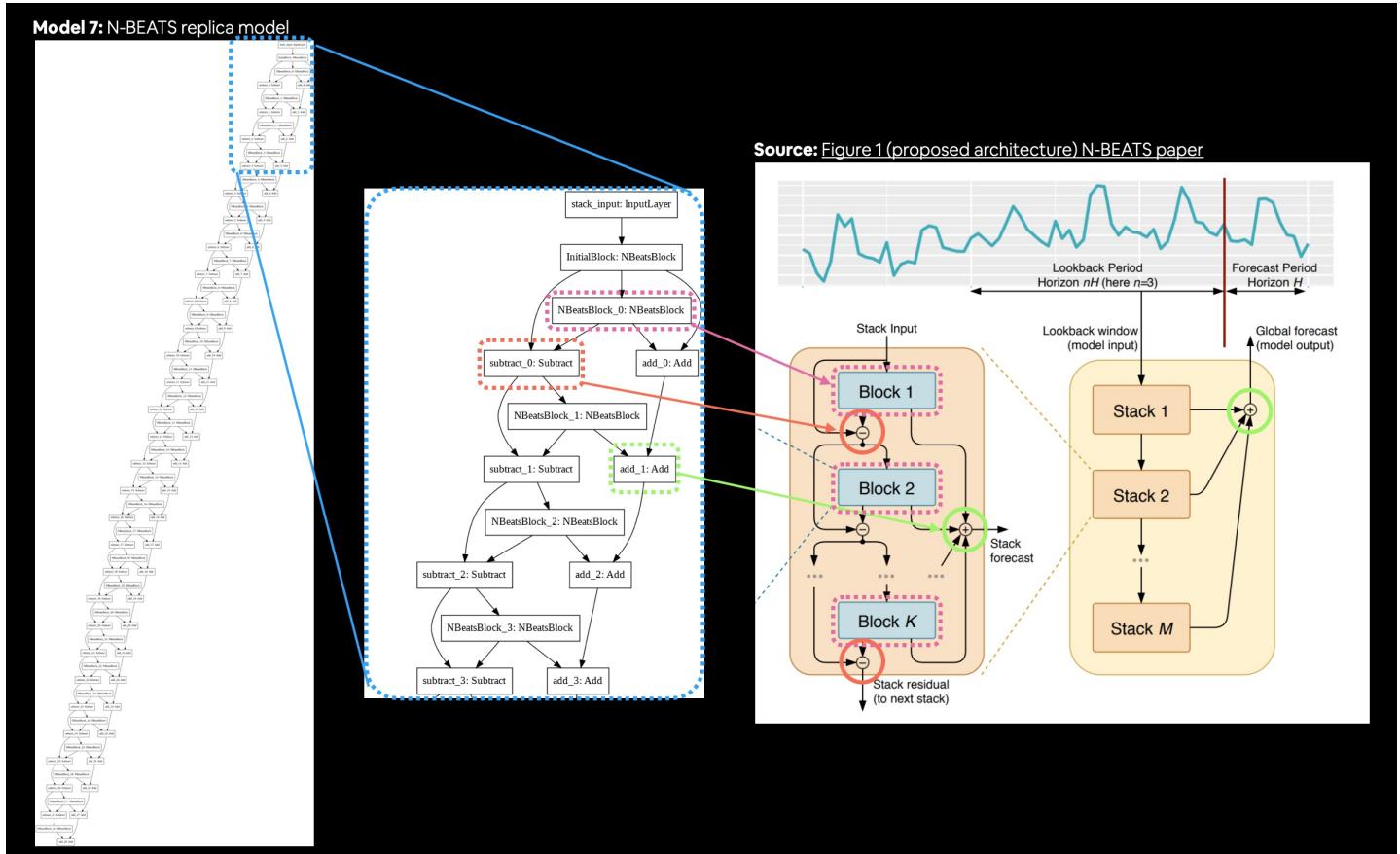






**Now that is one good looking model!**

It even looks similar to the model shown in Figure 1 of the N-BEATS paper.



**Comparison of `model_7` (N-BEATS replica model make with Keras Functional API) versus actual N-BEATS architecture diagram.**

Looks like our Functional API usage did the trick!

**Note:** Our N-BEATS model replicates the N-BEATS generic architecture, the training setups are largely the same, except for the N-BEATS paper used an ensemble of models to make predictions (multiple different loss functions and multiple different lookback windows), see Table 18 of the [N-BEATS paper](#) for more. An extension could be to setup this kind of training regime and see if it

improves performance.

## How about we try and save our version of the N-BEATS model?

In [ ]:

```
# This will error out unless a "get_config()" method is implemented - this could be extra
curriculum
model_7.save(model_7.name)
```

WARNING:absl:Found untraced functions such as theta\_layer\_call\_and\_return\_conditional\_losses, theta\_layer\_call\_fn, theta\_layer\_call\_and\_return\_conditional\_losses, theta\_layer\_call\_fn, theta\_layer\_call\_and\_return\_conditional\_losses while saving (showing 5 of 750). These functions will not be directly callable after loading.

INFO:tensorflow:Assets written to: model\_7\_N-BEATS/assets

INFO:tensorflow:Assets written to: model\_7\_N-BEATS/assets  
/usr/local/lib/python3.7/dist-packages/keras/utils/generic\_utils.py:497: CustomMaskWarning:  
Custom mask layers require a config and must override get\_config. When loading, the custom mask layer must be passed to the custom\_objects argument.  
category=CustomMaskWarning)

You'll notice a warning appears telling us to fully save our model correctly we need to implement a `get_config()` method in our custom layer class.

**Resource:** If you would like to save and load the N-BEATS model or any other custom or subclassed layer/model configuration, you should overwrite the `get_config()` and optionally `from_config()` methods. See the [TensorFlow Custom Objects documentation](#) for more.

## Model 8: Creating an ensemble (stacking different models together)

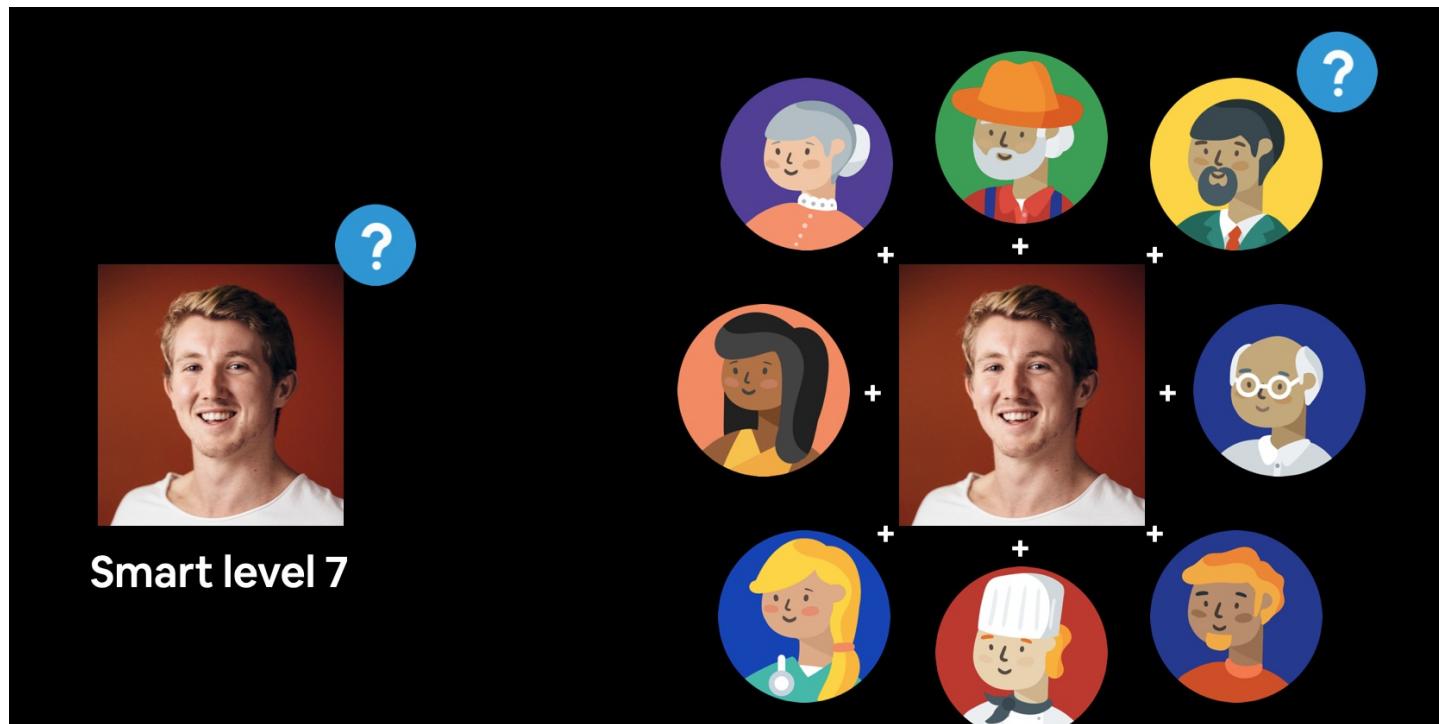
After all that effort, the N-BEATS algorithm's performance was underwhelming.

But again, this is part of the parcel of machine learning. Not everything will work.

That's when we refer back to the motto: experiment, experiment, experiment.

Our next experiment is creating an [ensemble of models](#).

An ensemble involves training and combining multiple different models on the same problem. Ensemble models are often the types of models you'll see winning data science competitions on websites like Kaggle.



*Example of the power of ensembling. One Daniel model makes a decision with a smart level of 7 but when a Daniel model teams up with multiple different people, together (ensembled) they make a decision with a smart level of 10. The key here is combining the decision power of people with different backgrounds, if you combined multiple Daniel models, you'd end up with an average smart level of 7. Note: smart level is not an actual measurement of decision making, it is for demonstration purposes only.*

For example, in the N-BEATS paper, they trained an ensemble of models (180 in total, see [section 3.4](#)) to achieve the results they did using a combination of:

- Different loss functions (sMAPE, MASE and MAPE)
- Different window sizes (2 x horizon, 3 x horizon, 4 x horizon...)

The benefit of ensembling models is you get the "decision of the crowd effect". Rather than relying on a single model's predictions, you can [take the average or median of many different models](#).

The keyword being: different.

It wouldn't make sense to train the same model 10 times on the same data and then average the predictions.

Fortunately, due to their random initialization, even deep learning models with the same architecture can produce different results.

What I mean by this is each time you create a deep learning model, it starts with random patterns (weights & biases) and then it adjusts these random patterns to better suit the dataset it's being trained on.

However, the process it adjusts these patterns is often a form of guided randomness as well (the SGD optimizer stands for stochastic or random gradient descent).

To create our ensemble models we're going to be using a combination of:

- Different loss functions (MAE, MSE, MAPE)
- Randomly initialized models

Essentially, we'll be creating a suite of different models all attempting to model the same data.

And hopefully the combined predictive power of each model is better than a single model on its own.

Let's find out!

We'll start by creating a function to produce a list of different models trained with different loss functions. Each layer in the ensemble models will be initialized with a random normal ([Gaussian distribution](#) using [He normal initialization](#), this'll help estimating the prediction intervals later on.

**Note:** In your machine learning experiments, you may have already dealt with examples of ensemble models. Algorithms such as the [random forest model](#) are a form of ensemble, it uses a number of randomly created decision trees where each individual tree may perform poorly but when combined gives great results.

## Constructing and fitting an ensemble of models (using different loss functions)

In [ ]:

```
def get_ensemble_models(horizon=HORIZON,
                        train_data=train_dataset,
                        test_data=test_dataset,
                        num_iter=10,
                        num_epochs=100,
                        loss_fns=["mae", "mse", "mape"]):
    """
    Returns a list of num_iter models each trained on MAE, MSE and MAPE loss.
    """
```

```

For example, if num_iter=10, a list of 30 trained models will be returned:
10 * len(["mae", "mse", "mape"]).
"""

# Make empty list for trained ensemble models
ensemble_models = []

# Create num_iter number of models per loss function
for i in range(num_iter):
    # Build and fit a new model with a different loss function
    for loss_function in loss_fns:
        print(f"Optimizing model by reducing: {loss_function} for {num_epochs} epochs, mode
l number: {i}")

    # Construct a simple model (similar to model_1)
    model = tf.keras.Sequential([
        # Initialize layers with normal (Gaussian) distribution so we can use the models
        for prediction
        # interval estimation later: https://www.tensorflow.org/api_docs/python/tf/keras/
        initializers/HeNormal
        layers.Dense(128, kernel_initializer="he_normal", activation="relu"),
        layers.Dense(128, kernel_initializer="he_normal", activation="relu"),
        layers.Dense(HORIZON)
    ])

    # Compile simple model with current loss function
    model.compile(loss=loss_function,
                  optimizer=tf.keras.optimizers.Adam(),
                  metrics=["mae", "mse"])

    # Fit model
    model.fit(train_data,
              epochs=num_epochs,
              verbose=0,
              validation_data=test_data,
              # Add callbacks to prevent training from going/stalling for too long
              callbacks=[tf.keras.callbacks.EarlyStopping(monitor="val_loss",
                                                          patience=200,
                                                          restore_best_weights=True),
                         tf.keras.callbacks.ReduceLROnPlateau(monitor="val_loss",
                                                               patience=100,
                                                               verbose=1)])
]

# Append fitted model to list of ensemble models
ensemble_models.append(model)

return ensemble_models # return list of trained models

```

## Ensemble model creator function created!

Let's try it out by running `num_iter=5` runs for 1000 epochs. This will result in 15 total models (5 for each different loss function).

Of course, these numbers could be tweaked to create more models trained for longer.

**Note:** With ensembles, you'll generally find more total models means better performance. However, this comes with the tradeoff of having to train more models (longer training time) and make predictions with more models (longer prediction time).

In [ ]:

```

%%time
# Get list of trained ensemble models
ensemble_models = get_ensemble_models(num_iter=5,
                                       num_epochs=1000)

```

Optimizing model by reducing: mae for 1000 epochs, model number: 0

Epoch 00794: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.

Epoch 00928: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.  
Optimizing model by reducing: mse for 1000 epochs, model number: 0

Epoch 00591: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.

Epoch 00707: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.

Epoch 00807: ReduceLROnPlateau reducing learning rate to 1.0000000656873453e-06.  
Optimizing model by reducing: mape for 1000 epochs, model number: 0

Epoch 00165: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.

Epoch 00282: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.

Epoch 00382: ReduceLROnPlateau reducing learning rate to 1.0000000656873453e-06.  
Optimizing model by reducing: mae for 1000 epochs, model number: 1  
Optimizing model by reducing: mse for 1000 epochs, model number: 1

Epoch 00409: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.

Epoch 00509: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.  
Optimizing model by reducing: mape for 1000 epochs, model number: 1

Epoch 00185: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.

Epoch 00726: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.

Epoch 00826: ReduceLROnPlateau reducing learning rate to 1.0000000656873453e-06.  
Optimizing model by reducing: mae for 1000 epochs, model number: 2  
Optimizing model by reducing: mse for 1000 epochs, model number: 2  
Optimizing model by reducing: mape for 1000 epochs, model number: 2

Epoch 00241: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.

Epoch 00341: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.  
Optimizing model by reducing: mae for 1000 epochs, model number: 3

Epoch 00572: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.  
Optimizing model by reducing: mse for 1000 epochs, model number: 3

Epoch 00304: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.

Epoch 00607: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.

Epoch 00707: ReduceLROnPlateau reducing learning rate to 1.0000000656873453e-06.  
Optimizing model by reducing: mape for 1000 epochs, model number: 3

Epoch 00301: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.

Epoch 00401: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.  
Optimizing model by reducing: mae for 1000 epochs, model number: 4  
Optimizing model by reducing: mse for 1000 epochs, model number: 4

Epoch 00640: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.

Epoch 00740: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.  
Optimizing model by reducing: mape for 1000 epochs, model number: 4

Epoch 00132: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.

Epoch 00609: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.

Epoch 00709: ReduceLROnPlateau reducing learning rate to 1.0000000656873453e-06.  
CPU times: user 6min 24s, sys: 38.6 s, total: 7min 3s  
Wall time: 7min 58s

**Look at all of those models!**

**How about we now write a function to use the list of trained ensemble models to make predictions and then return a list of predictions (one set of predictions per model)?**

## Making predictions with an ensemble model

In [ ]:

```
# Create a function which uses a list of trained models to make and return a list of predictions
def make_ensemble_preds(ensemble_models, data):
    ensemble_preds = []
    for model in ensemble_models:
        preds = model.predict(data) # make predictions with current ensemble model
        ensemble_preds.append(preds)
    return tf.constant(tf.squeeze(ensemble_preds))
```

In [ ]:

```
# Create a list of ensemble predictions
ensemble_preds = make_ensemble_preds(ensemble_models=ensemble_models,
                                      data=test_dataset)
ensemble_preds
```

WARNING:tensorflow:5 out of the last 22 calls to <function Model.make\_predict\_function.<locals>.predict\_function at 0x7fdcef255d40> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has experimental\_relax\_shapes=True option that relaxes argument shapes that can avoid unnecessary retracing. For (3), please refer to [https://www.tensorflow.org/guide/function#controlling\\_retracing](https://www.tensorflow.org/guide/function#controlling_retracing) and [https://www.tensorflow.org/api\\_docs/python/tf/function](https://www.tensorflow.org/api_docs/python/tf/function) for more details.

WARNING:tensorflow:5 out of the last 22 calls to <function Model.make\_predict\_function.<locals>.predict\_function at 0x7fdcef255d40> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has experimental\_relax\_shapes=True option that relaxes argument shapes that can avoid unnecessary retracing. For (3), please refer to [https://www.tensorflow.org/guide/function#controlling\\_retracing](https://www.tensorflow.org/guide/function#controlling_retracing) and [https://www.tensorflow.org/api\\_docs/python/tf/function](https://www.tensorflow.org/api_docs/python/tf/function) for more details.

WARNING:tensorflow:6 out of the last 23 calls to <function Model.make\_predict\_function.<locals>.predict\_function at 0x7fdc77fed9e0> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has experimental\_relax\_shapes=True option that relaxes argument shapes that can avoid unnecessary retracing. For (3), please refer to [https://www.tensorflow.org/guide/function#controlling\\_retracing](https://www.tensorflow.org/guide/function#controlling_retracing) and [https://www.tensorflow.org/api\\_docs/python/tf/function](https://www.tensorflow.org/api_docs/python/tf/function) for more details.

WARNING:tensorflow:6 out of the last 23 calls to <function Model.make\_predict\_function.<locals>.predict\_function at 0x7fdc77fed9e0> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has experimental\_relax\_shapes=True option that relaxes argument shapes that can avoid unnecessary retracing. For (3), please refer to [https://www.tensorflow.org/guide/function#controlling\\_retracing](https://www.tensorflow.org/guide/function#controlling_retracing) and [https://www.tensorflow.org/api\\_docs/python/tf/function](https://www.tensorflow.org/api_docs/python/tf/function) for more details.

Out[ ]:

```
<tf.Tensor: shape=(15, 556), dtype=float32, numpy=
array([[ 8805.756,   8773.019,   9028.609, ...,  50112.656,  49132.555,
       46455.695],
       [ 8764.092,   8740.744,   9051.838, ...,  49355.098,  48502.336,
       45333.934],
       [ 8732.57 ,   8719.407,   9093.386, ...,  49921.9 ,  47992.15 ,
       45316.45 ],
       ...,
       [ 8938.421,   8773.84 ,   9045.577, ...,  49488.133,  49741.4 ,
       46536.25 ],
```

```
[ 8724.761,  8805.311,  9094.972, ..., 49553.086, 48492.86 ,  
 45084.266],  
[ 8823.311,  8768.297,  9047.492, ..., 49759.902, 48090.945,  
 45874.336]], dtype=float32)>
```

Now we've got a set of ensemble predictions, we can evaluate them against the ground truth values.

However, since we've trained 15 models, there's going to be 15 sets of predictions. Rather than comparing every set of predictions to the ground truth, let's take the median (you could also take the mean too but [the median is usually more robust than the mean](#)).

In [ ]:

```
# Evaluate ensemble model(s) predictions  
ensemble_results = evaluate_preds(y_true=y_test,  
                                    y_pred=np.median(ensemble_preds, axis=0)) # take the m  
edian across all ensemble predictions  
ensemble_results
```

Out [ ]:

```
{'mae': 567.4423,  
'mape': 2.5843322,  
'mase': 0.996839,  
'mse': 1144512.9,  
'rmse': 1069.8191}
```

Nice! Looks like the ensemble model is the best performing model on the MAE metric so far.

## Plotting the prediction intervals (uncertainty estimates) of our ensemble

Right now all of our model's (prior to the ensemble model) are predicting single points.

Meaning, given a set of `WINDOW_SIZE=7` values, the model will predict `HORIZION=1`.

But what might be more helpful than a single value?

Perhaps a range of values?

For example, if a model is predicting the price of Bitcoin to be 50,000USD tomorrow, would it be helpful to know it's predicting the 50,000USD because it's predicting the price to be between 48,000 and 52,000USD? (note: "\$" has been omitted from the previous sentence due to formatting issues)

Knowing the range of values a model is predicting may help you make better decisions for your forecasts.

You'd know that although the model is predicting 50,000USD (a **point prediction**, or single value in time), the value could actually be within the range 48,000USD to 52,000USD (of course, the value could also be *outside* of this range as well, but we'll get to that later).

These kind of prediction ranges are called **prediction intervals** or **uncertainty estimates**. And they're often as important as the forecast itself.

Why?

Because **point predictions** are almost always going to be wrong. So having a range of values can help with decision making.

### Resource(s):

- The steps we're about to take have been inspired by the Machine Learning Mastery blog post [Prediction Intervals for Deep Learning Neural Networks](#). Check out the post for more options to measure uncertainty with neural networks.
- For an example of uncertainty estimates being used in the wild, I'd also refer to Uber's [Engineering Uncertainty Estimation in Neural Networks for Time Series Prediction at Uber](#) blog post.

## 95 percent prediction interval

Large periods of uncertainty around big events

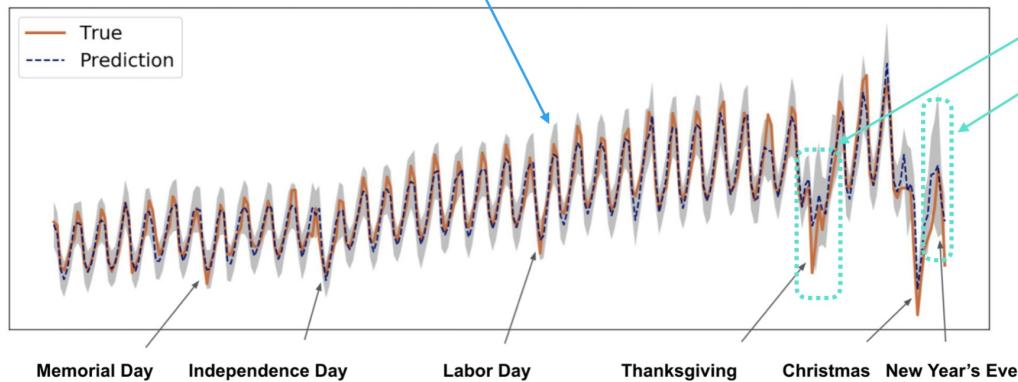


Figure 2: Daily completed trips in San Francisco during eight months of the testing set. True values are represented by the orange solid line, and predictions by the blue dashed line, where the 95 percent prediction band is shown as the grey area. (Note: exact values are anonymized.)

Source: Figure 2 from [Engineering Uncertainty Estimation in Neural Networks for Time Series Prediction at Uber](#)

**Example of how uncertainty estimates and predictions intervals can give an understanding of where point predictions (a single number) may not include all of useful information you'd like to know. For example, your model's point prediction for Uber trips on New Years Eve might be 100 (a made up number) but really, the prediction intervals are between 55 and 153 (both made up for the example). In this case, preparing 100 rides might end up being 53 short (it could even be more, like the point prediction, the prediction intervals are also estimates). The image comes from Uber's [blog post on uncertainty estimation in neural networks](#).**

One way of getting the 95% confidence prediction intervals for a deep learning model is the bootstrap method:

1. Take the predictions from a number of randomly initialized models (we've got this thanks to our ensemble model)
2. Measure the standard deviation of the predictions
3. Multiply standard deviation by [1.96](#) (assuming the distribution is Gaussian, 95% of observations fall within 1.96 standard deviations of the mean, this is why we initialized our neural networks with a normal distribution)
4. To get the prediction interval upper and lower bounds, add and subtract the value obtained in (3) to the mean/median of the predictions made in (1)

In [ ]:

```
# Find upper and lower bounds of ensemble predictions
def get_upper_lower(preds): # 1. Take the predictions of multiple randomly initialized deep learning neural networks

    # 2. Measure the standard deviation of the predictions
    std = tf.math.reduce_std(preds, axis=0)

    # 3. Multiply the standard deviation by 1.96
    interval = 1.96 * std # https://en.wikipedia.org/wiki/1.96

    # 4. Get the prediction interval upper and lower bounds
    preds_mean = tf.reduce_mean(preds, axis=0)
    lower, upper = preds_mean - interval, preds_mean + interval
    return lower, upper

# Get the upper and lower bounds of the 95%
lower, upper = get_upper_lower(preds=ensemble_preds)
```

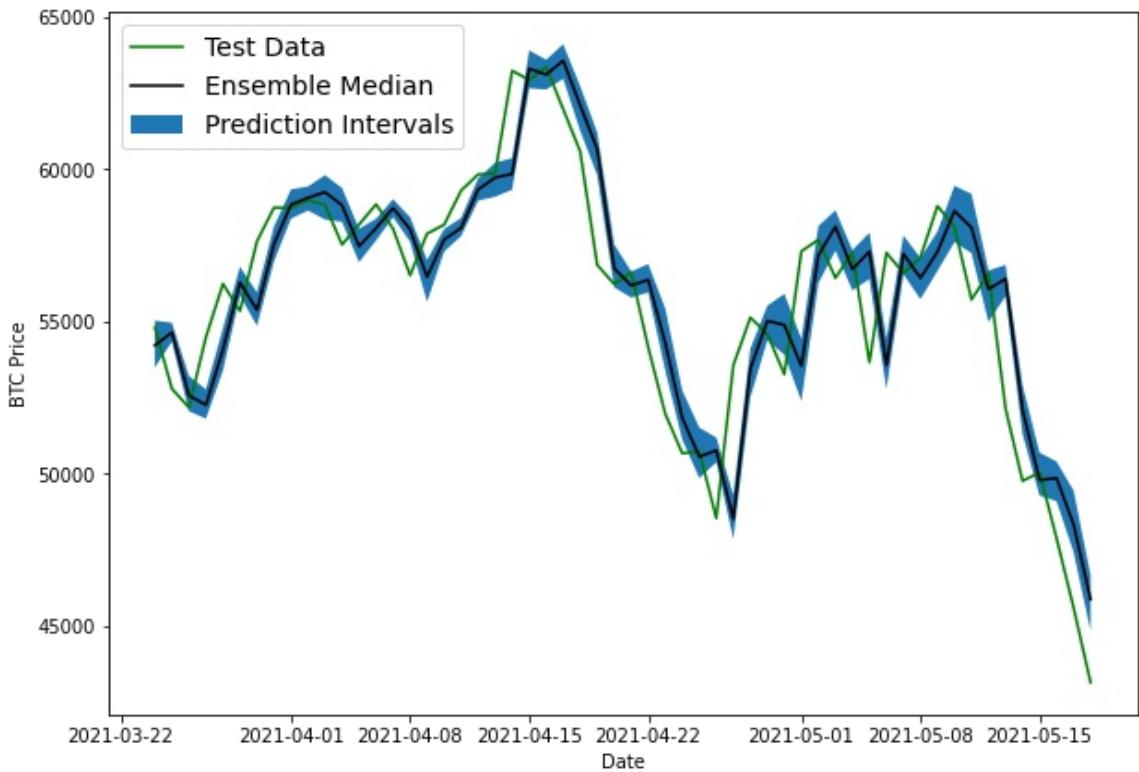
Wonderful, now we've got the upper and lower bounds for the the 95% prediction interval, let's plot them against our ensemble model's predictions.

To do so, we can use our plotting function as well as the [matplotlib.pyplot.fill\\_between\(\)](#) method to shade in the space between the upper and lower bounds.

In [ ]:

```
# Get the median values of our ensemble preds
ensemble_median = np.median(ensemble_preds, axis=0)

# Plot the median of our ensemble preds along with the prediction intervals (where the predictions fall between)
offset=500
plt.figure(figsize=(10, 7))
plt.plot(X_test.index[offset:], y_test[offset:], "g", label="Test Data")
plt.plot(X_test.index[offset:], ensemble_median[offset:], "k-", label="Ensemble Median")
plt.xlabel("Date")
plt.ylabel("BTC Price")
plt.fill_between(X_test.index[offset:],
                 (lower)[offset:],
                 (upper)[offset:], label="Prediction Intervals")
plt.legend(loc="upper left", fontsize=14);
```



We've just plotted:

- The test data (the ground truth Bitcoin prices)
- The median of the ensemble predictions
- The 95% prediction intervals (assuming the data is Gaussian/normal, the model is saying that 95% of the time, predicted value should fall between this range)

What can you tell about the ensemble model from the plot above?

It looks like the ensemble predictions are lagging slightly behind the actual data.

And the prediction intervals are fairly low throughout.

The combination of lagging predictions as well as low prediction intervals indicates that our ensemble model may be **overfitting** the data, meaning it's basically replicating what a naïve model would do and just predicting the previous timestep value for the next value.

This would explain why previous attempts to beat the naïve forecast have been futile.

We can test this hypothesis of overfitting by creating a model to make predictions into the future and seeing what they look like.

**Note:** Our prediction intervals assume that the data we're using come from a Gaussian/normal distribution (also called a bell curve), however, open systems rarely follow the Gaussian. We'll see

this later on with the turkey problem ☺. For further reading on this topic, I'd recommend reading [The Black Swan by Nassim Nicholas Taleb](#), especially Part 2 and Chapter 15.

## Aside: two types of uncertainty (coconut and subway)

Inheritly, you know you cannot predict the future.

That doesn't mean trying to isn't valuable.

For many things, future predictions are helpful. Such as knowing the bus you're trying to catch to the library leaves at 10:08am. The time 10:08am is a **point prediction**, if the bus left at a random time every day, how helpful would it be?

Just like saying the price of Bitcoin tomorrow will be 50,000USD is a point prediction.

However, as we've discussed knowing a **prediction interval** or **uncertainty estimate** can be as helpful or even more helpful than a point prediction itself.

Uncertainty estimates seek out to qualitatively and quantitatively answer the questions:

- What can my model know? (with perfect data, what's possible to learn?)
- What doesn't my model know? (what can a model never predict?)

There are two types of uncertainty in machine learning you should be aware of:

- **Aleatoric uncertainty** - this type of uncertainty cannot be reduced, it is also referred to as "data" or "subway" uncertainty.
  - Let's say your train is scheduled to arrive at 10:08am but very rarely does it arrive at *exactly* 10:08am. You know it's usually a minute or two either side and perhaps up to 10-minutes late if traffic is bad. Even with all the data you could imagine, this level of uncertainty is still going to be present (much of it being noise).
  - When we measured prediction intervals, we were measuring a form of subway uncertainty for Bitcoin price predictions (a little either side of the point prediction).
- **Epistemic uncertainty** - this type of uncertainty can be reduced, it is also referred to as "model" or "coconut" uncertainty, it is very hard to calculate.
  - The analogy for coconut uncertainty involves whether or not you'd get hit on the head by a coconut when going to a beach.
    - If you were at a beach with coconuts trees, as you could imagine, this would be very hard to calculate. How often does a coconut fall of a tree? Where are you standing?
    - But you could reduce this uncertainty to zero by going to a beach without coconuts (collect more data about your situation).
  - Model uncertainty can be reduced by collecting more data samples/building a model to capture different parameters about the data you're modelling.

The lines between these are blurred (one type of uncertainty can change forms into the other) and they can be confusing at first but are important to keep in mind for any kind of time series prediction.

If you ignore the uncertainties, are you really going to get a reliable prediction?

Perhaps another example might help.

## Uncertainty in dating

Let's say you're going on a First Date Feedback Radio Show to help improve your dating skills.

Where you go on a blind first date with a girl (feel free to replace girl with your own preference) and the radio hosts record the date and then playback snippets of where you could've improved.

And now let's add a twist.

Last week your friend went on the same show. They told you about the girl they met and how the conversation went.

Because you're now a machine learning engineer, you decide to build a machine learning model to help you with first date conversations.

What levels of uncertainty do we have here?

From an **aleatory uncertainty** (data) point of view, no matter how many conversations of first dates you collect, the conversation you end up having will likely be different to the rest (the best conversations have no subject and appear random).

From an **epistemic uncertainty** (model) point of view, if the date is truly blind and both parties don't know who they're seeing until they meet in person, the epistemic uncertainty would be high. Because now you have no idea who the person you're going to meet is nor what you might talk about.

However, the level of epistemic uncertainty would be reduced if your friend told about the girl they went on a date with last week on the show and it turns out you're going on a date with the same girl.

But even though you know a little bit about the girl, your **aleatory uncertainty** (or subway uncertainty) is still high because you're not sure where the conversation will go.

If you're wondering where above scenario came from, it happened to me this morning. Good timing right?

## Learning more on uncertainty

The field of quantifying uncertainty estimation in machine learning is a growing area of research.

If you'd like to learn more I'd recommend the following.

□ **Resources:** Places to learn more about uncertainty in machine learning/forecasting:

- □ [MIT 6.S191: Evidential Deep Learning and Uncertainty](#)
- [Uncertainty quantification on Wikipedia](#)
- [\*Why you should care about the Nate Silver vs. Nassim Taleb Twitter war\*](#) by Isaac Faber - a great insight into the role of uncertainty in the example of election prediction.
- [\*3 facts about time series forecasting that surprise experienced machine learning practitioners\*](#) by Skander Hannachi - fantastic outline of some of the main mistakes people make when building forecasting models, especially forgetting about uncertainty estimates.
- [\*Engineering Uncertainty Estimation in Neural Networks for Time Series Prediction at Uber\*](#) - a discussion on techniques Uber used to engineer uncertainty estimates into their time series neural networks.

## Model 9: Train a model on the full historical data to make predictions into future

What would a forecasting model be worth if we didn't use it to predict into the future?

It's time we created a model which is able to make future predictions on the price of Bitcoin.

To make predictions into the future, we'll train a model on the full dataset and then get to make predictions to some future horizon.

Why use the full dataset?

Previously, we split our data into training and test sets to evaluate how our model did on pseudo-future data (the test set).

But since the goal of a forecasting model is to predict values into the actual-future, we won't be using a test set.

□ **Note:** Forecasting models need to be retrained every time a forecast is made. Why? Because if Bitcoin prices are updated daily and you predict the price for tomorrow. Your model is only really valid for one day. When a new price comes out (e.g. the next day), you'll have to retrain your model to incorporate that new price to predict the next forecast.

Let's get some data ready.

In [ ]:

```
bitcoin_prices_windowed.head()
```

Out[ ]:

Date	Price	block_reward	Price+1	Price+2	Price+3	Price+4	Price+5	Price+6	Price+7
2013-10-01	123.65499	25	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2013-10-02	125.45500	25	123.65499	NaN	NaN	NaN	NaN	NaN	NaN
2013-10-03	108.58483	25	125.45500	123.65499	NaN	NaN	NaN	NaN	NaN
2013-10-04	118.67466	25	108.58483	125.45500	123.65499	NaN	NaN	NaN	NaN
2013-10-05	121.33866	25	118.67466	108.58483	125.45500	123.65499	NaN	NaN	NaN

In [ ]:

```
# Train model on entire data to make prediction for the next day
X_all = bitcoin_prices_windowed.drop(["Price", "block_reward"], axis=1).dropna().to_numpy()
# only want prices, our future model can be a univariate model
y_all = bitcoin_prices_windowed.dropna()["Price"].to_numpy()
```

**Windows and labels ready! Let's turn them into performance optimized TensorFlow Datasets by:**

1. Turning `X_all` and `y_all` into tensor Datasets using `tf.data.Dataset.from_tensor_slices()`
2. Combining the features and labels into a Dataset tuple using `tf.data.Dataset.zip()`
3. Batch and prefetch the data using `tf.data.Dataset.batch()` and `tf.data.Dataset.prefetch()` respectively

In [ ]:

```
# 1. Turn X and y into tensor Datasets
features_dataset_all = tf.data.Dataset.from_tensor_slices(X_all)
labels_dataset_all = tf.data.Dataset.from_tensor_slices(y_all)

# 2. Combine features & labels
dataset_all = tf.data.Dataset.zip((features_dataset_all, labels_dataset_all))

# 3. Batch and prefetch for optimal performance
BATCH_SIZE = 1024 # taken from Appendix D in N-BEATS paper
dataset_all = dataset_all.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)

dataset_all
```

Out[ ]:

```
<PrefetchDataset shapes: ((None, 7), (None,)), types: (tf.float64, tf.float64)>
```

And now let's create a model similar to `model_1` except with an extra layer, we'll also fit it to the entire dataset for 100 epochs (feel free to play around with the number of epochs or callbacks here, you've got the skills to now).

In [ ]:

```
tf.random.set_seed(42)

# Create model (nice and simple, just to test)
model_9 = tf.keras.Sequential([
    layers.Dense(128, activation="relu"),
    layers.Dense(128, activation="relu"),
    layers.Dense(HORIZON)
])
```

```
# Compile
model_9.compile(loss=tf.keras.losses.mae,
                 optimizer=tf.keras.optimizers.Adam())

# Fit model on all of the data to make future forecasts
model_9.fit(dataset_all,
             epochs=100,
             verbose=0) # don't print out anything, we've seen this all before
```

Out [ ]:

<keras.callbacks.History at 0x7fdc77ae0dd0>

## Make predictions on the future

Let's predict the future and get rich!

Well... maybe not.

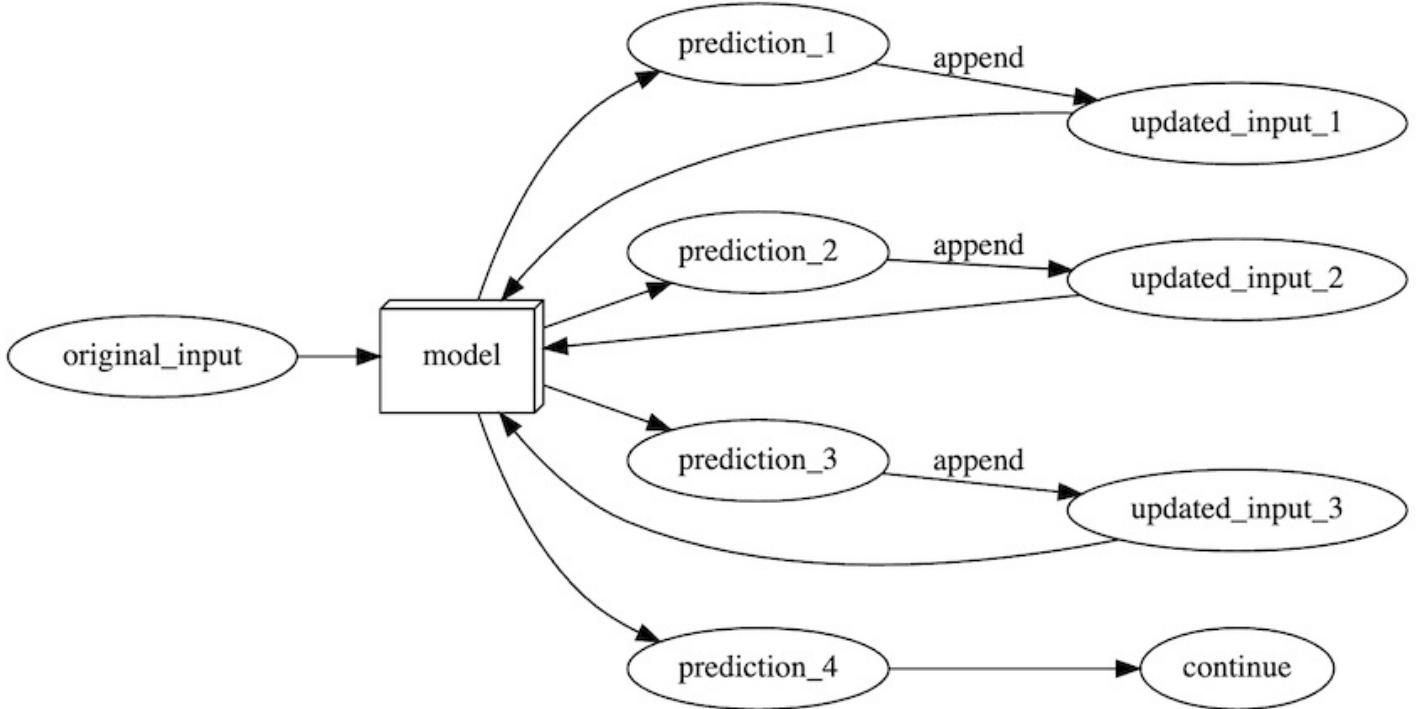
As you've seen so far, our machine learning models have performed quite poorly at predicting the price of Bitcoin (time series forecasting in open systems is typically a game of luck), often worse than the naive forecast.

That doesn't mean we can't use our models to *try* and predict into the future right?

To do so, let's start by defining a variable `INTO_FUTURE` which decides how many timesteps we'd like to predict into the future.

In [ ]:

```
# How many timesteps to predict into the future?
INTO_FUTURE = 14 # since our Bitcoin data is daily, this is for 14 days
```



*Example flow chart representing the loop we're about to create for making forecasts. Not pictured: retraining a forecasting model every time a forecast is made & new data is acquired. For example, if you're predicting the price of Bitcoin daily, you'd want to retrain your model every day, since each day you're going to have a new data point to work with.*

Alright, let's create a function which returns `INTO_FUTURE` forecasted values using a trained model.

To do so, we'll build the following steps:

1. Function which takes as input:

- a list of values (the Bitcoin historical data)
- a trained model (such as `model_9`)
- a window into the future to predict (our `INTO_FUTURE` variable)

- the window size a model was trained on (`WINDOW_SIZE`) - the model can only predict on the same kind of data it was trained on
  2. Creates an empty list for future forecasts (this will be returned at the end of the function) and extracts the last `WINDOW_SIZE` values from the input values (predictions will start from the last `WINDOW_SIZE` values of the training data)
  3. Loop `INTO_FUTURE` times making a prediction on `WINDOW_SIZE` datasets which update to remove the first the value and append the latest prediction
    - Eventually future predictions will be made using the model's own previous predictions as input

In [ ]:

```
# 1. Create function to make predictions into the future
def make_future_forecast(values, model, into_future, window_size=WINDOW_SIZE) -> list:
    """
    Makes future forecasts into_future steps after values ends.

    Returns future forecasts as list of floats.
    """
    # 2. Make an empty list for future forecasts/prepare data to forecast on
    future_forecast = []
    last_window = values[-WINDOW_SIZE:] # only want preds from the last window (this will
    get updated)

    # 3. Make INTO_FUTURE number of predictions, altering the data which gets predicted on
    each time
    for _ in range(into_future):
        # Predict on last window then append it again, again, again (model starts to make forecasts
        # on its own forecasts)
        future_pred = model.predict(tf.expand_dims(last_window, axis=0))
        print(f"Predicting on: \n {last_window} -> Prediction: {tf.squeeze(future_pred).numpy()}\n")

        # Append predictions to future_forecast
        future_forecast.append(tf.squeeze(future_pred).numpy())
        # print(future_forecast)

        # Update last window with new pred and get WINDOW_SIZE most recent preds (model was
        # trained on WINDOW_SIZE windows)
        last_window = np.append(last_window, future_pred)[-WINDOW_SIZE:]

    return future_forecast
```

**Nice! Time to bring BitPredict  to life and make future forecasts of the price of Bitcoin.**

**Exercise:** In terms of a forecasting model, what might another approach to our `make_future_forecasts()` function? Recall, that for making forecasts, you need to retrain a model each time you want to generate a new prediction.

**So perhaps you could try to: make a prediction (one timestep into the future), retrain a model with this new prediction appended to the data, make a prediction, append the prediction, retrain a model... etc.**

**As it is, the `make_future_forecasts()` function skips the retraining of a model part.**

In [ ] :

```
        into_future=INTO_FUTURE,  
        window_size=WINDOW_SIZE)
```

Predicting on:

```
[56573.5554719 52147.82118698 49764.1320816 50032.69313676  
47885.62525472 45604.61575361 43144.47129086] -> Prediction: 55764.46484375
```

Predicting on:

```
[52147.82118698 49764.1320816 50032.69313676 47885.62525472  
45604.61575361 43144.47129086 55764.46484375] -> Prediction: 50985.9453125
```

Predicting on:

```
[49764.1320816 50032.69313676 47885.62525472 45604.61575361  
43144.47129086 55764.46484375 50985.9453125] -> Prediction: 48522.96484375
```

Predicting on:

```
[50032.69313676 47885.62525472 45604.61575361 43144.47129086  
55764.46484375 50985.9453125 48522.96484375] -> Prediction: 48137.203125
```

Predicting on:

```
[47885.62525472 45604.61575361 43144.47129086 55764.46484375  
50985.9453125 48522.96484375 48137.203125] -> Prediction: 47880.63671875
```

Predicting on:

```
[45604.61575361 43144.47129086 55764.46484375 50985.9453125  
48522.96484375 48137.203125 47880.63671875] -> Prediction: 46879.71875
```

Predicting on:

```
[43144.47129086 55764.46484375 50985.9453125 48522.96484375  
48137.203125 47880.63671875 46879.71875] -> Prediction: 48227.6015625
```

Predicting on:

```
[55764.46484375 50985.9453125 48522.96484375 48137.203125  
47880.63671875 46879.71875 48227.6015625] -> Prediction: 53963.69140625
```

Predicting on:

```
[50985.9453125 48522.96484375 48137.203125 47880.63671875  
46879.71875 48227.6015625 53963.69140625] -> Prediction: 49685.55859375
```

Predicting on:

```
[48522.96484375 48137.203125 47880.63671875 46879.71875  
48227.6015625 53963.69140625 49685.55859375] -> Prediction: 47596.17578125
```

Predicting on:

```
[48137.203125 47880.63671875 46879.71875 48227.6015625  
53963.69140625 49685.55859375 47596.17578125] -> Prediction: 48114.4296875
```

Predicting on:

```
[47880.63671875 46879.71875 48227.6015625 53963.69140625  
49685.55859375 47596.17578125 48114.4296875] -> Prediction: 48808.0078125
```

Predicting on:

```
[46879.71875 48227.6015625 53963.69140625 49685.55859375  
47596.17578125 48114.4296875 48808.0078125] -> Prediction: 48623.85546875
```

Predicting on:

```
[48227.6015625 53963.69140625 49685.55859375 47596.17578125  
48114.4296875 48808.0078125 48623.85546875] -> Prediction: 50178.72265625
```

In [ ]:

```
future_forecast[:10]
```

Out[ ]:

```
[55764.465,  
 50985.945,  
 48522.965,  
 48137.203,  
 47880.637,  
 46879.72,  
 48227.6
```

```
[53963.69,
49685.56,
47596.176]
```

## Plot future forecasts

This is so exciting! Forecasts made!

But right now, they're just numbers on a page.

Let's bring them to life by adhering to the data explorer's motto: visualize, visualize, visualize!

To plot our model's future forecasts against the historical data of Bitcoin, we're going to need a series of future dates (future dates from the final date of where our dataset ends).

How about we create a function to return a date range from some specified start date to a specified number of days into the future (`INTO_FUTURE`).

To do so, we'll use a combination of NumPy's `datetime64 datatype` (our Bitcoin dates are already in this datatype) as well as NumPy's `timedelta64` method which helps to create date ranges.

In [ ]:

```
def get_future_dates(start_date, into_future, offset=1):
    """
    Returns array of datetime values from ranging from start_date to start_date+horizon.

    start_date: date to start range (np.datetime64)
    into_future: number of days to add onto start date for range (int)
    offset: number of days to offset start_date by (default 1)
    """
    start_date = start_date + np.timedelta64(offset, "D") # specify start date, "D" stands
    for day
    end_date = start_date + np.timedelta64(into_future, "D") # specify end date
    return np.arange(start_date, end_date, dtype="datetime64[D]") # return a date range between start date and end date
```

The start date of our forecasted dates will be the last date of our dataset.

In [ ]:

```
# Last timestep of timesteps (currently in np.datetime64 format)
last_timestep = bitcoin_prices.index[-1]
last_timestep
```

Out[ ]:

```
Timestamp('2021-05-18 00:00:00')
```

In [ ]:

```
# Get next two weeks of timesteps
next_time_steps = get_future_dates(start_date=last_timestep,
                                    into_future=INTO_FUTURE)
next_time_steps
```

Out[ ]:

```
array(['2021-05-19', '2021-05-20', '2021-05-21', '2021-05-22',
       '2021-05-23', '2021-05-24', '2021-05-25', '2021-05-26',
       '2021-05-27', '2021-05-28', '2021-05-29', '2021-05-30',
       '2021-05-31', '2021-06-01'], dtype='datetime64[D']')
```

Look at that! We've now got a list of dates we can use to visualize our future Bitcoin predictions.

But to make sure the lines of the plot connect (try not running the cell below and then plotting the data to see what I mean), let's insert the last timestep and Bitcoin price of our training data to the `next_time_steps` and `future_forecast arrays`.

In [ ]:

```
# Insert last timestep/final price so the graph doesn't look messed
next_time_steps = np.insert(next_time_steps, 0, last_timestep)
future_forecast = np.insert(future_forecast, 0, btc_price[-1])
next_time_steps, future_forecast
```

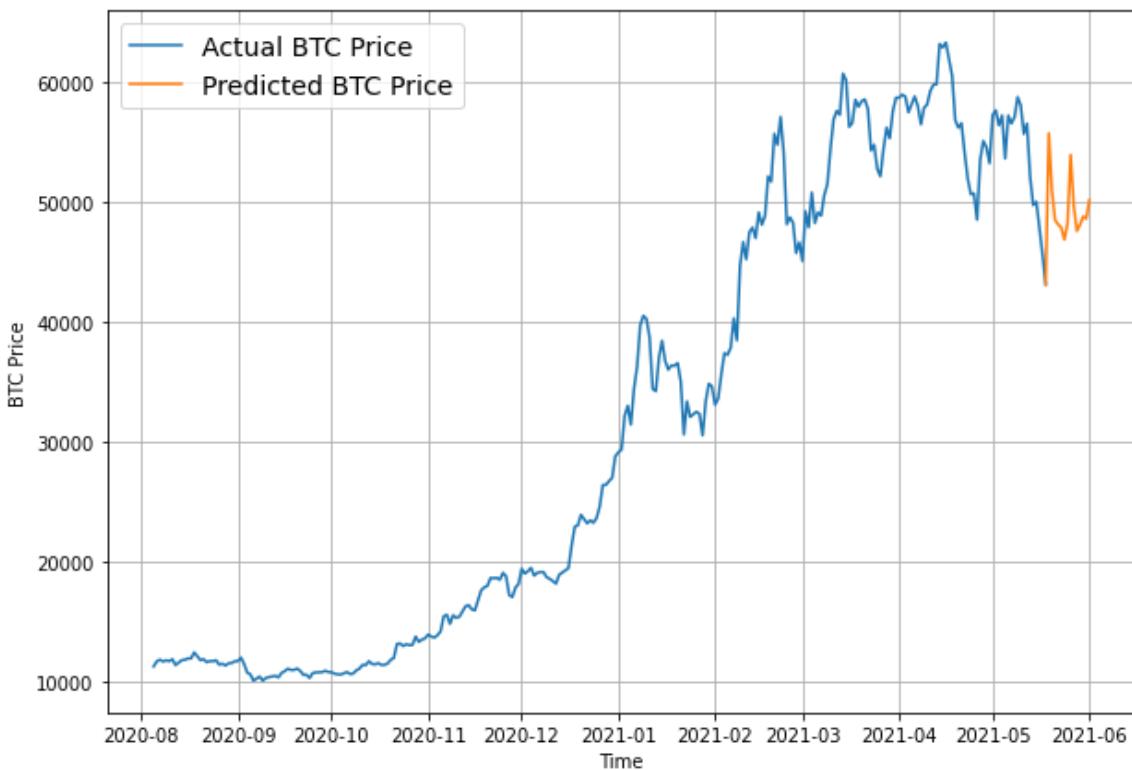
Out [ ]:

```
(array(['2021-05-18', '2021-05-19', '2021-05-20', '2021-05-21',
       '2021-05-22', '2021-05-23', '2021-05-24', '2021-05-25',
       '2021-05-26', '2021-05-27', '2021-05-28', '2021-05-29',
       '2021-05-30', '2021-05-31', '2021-06-01'], dtype='datetime64[D']'),
 array([43144.473, 55764.465, 50985.945, 48522.965, 48137.203, 47880.637,
        46879.72 , 48227.6 , 53963.69 , 49685.56 , 47596.176, 48114.43 ,
        48808.008, 48623.855, 50178.723], dtype=float32))
```

Time to plot!

In [ ]:

```
# Plot future price predictions of Bitcoin
plt.figure(figsize=(10, 7))
plot_time_series(bitcoin_prices.index, btc_price, start=2500, format="--", label="Actual BTC Price")
plot_time_series(next_time_steps, future_forecast, format="--", label="Predicted BTC Price")
```



Hmmm... how did our model go?

It looks like our predictions are starting to form a bit of a cyclic pattern (up and down in the same way).

Perhaps that's due to our model overfitting the training data and not generalizing well for future data. Also, as you could imagine, the further you predict into the future, the higher your chance for error (try seeing what happens when you predict 100 days into the future).

But of course, we can't measure these predictions as they are because after all, they're predictions into the actual-future (by the time you read this, the future might have already happened, if so, how did the model go?).

**Note:** A reminder, the predictions we've made here are not financial advice. And by now, you should be well aware of just how poor machine learning models can be at forecasting values in an open system - anyone promising you a model which can "beat the market" is likely trying to scam

you, oblivious to their errors or very lucky.

## Model 10: Why forecasting is BS (the turkey problem □)

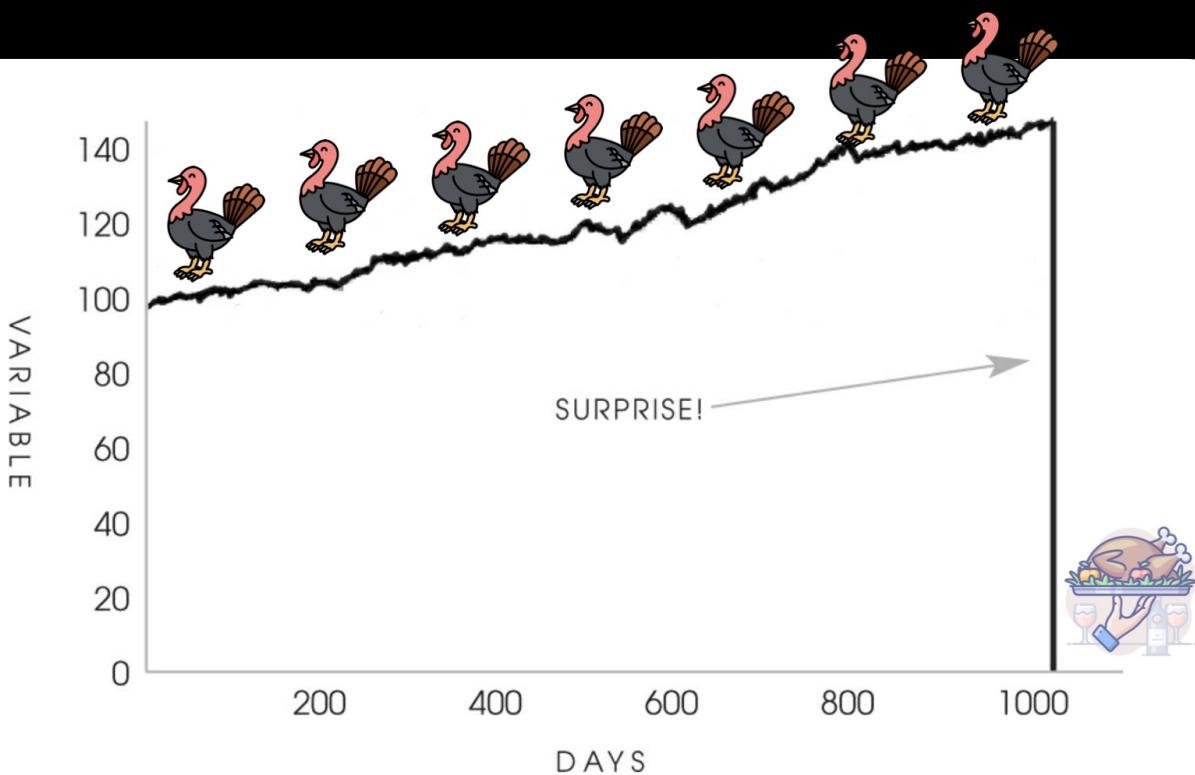
When creating any kind of forecast, you must keep the **turkey problem** in mind.

The **turkey problem** is an analogy for when your observational data (your historical data) fails to capture a future event which is catastrophic and could lead you to ruin.

The story goes, a turkey lives a good life for 1000 days, being fed every day and taken care of by its owners until the evening before Thanksgiving.

Based on the turkey's observational data, it has no reason to believe things shouldn't keep going the way they are.

In other words, how could a turkey possibly predict that on day 1001, after 1000 consecutive good days, it was about to have a far from ideal day.



**FIGURE 1: ONE THOUSAND AND ONE DAYS OF HISTORY**

A turkey before and after Thanksgiving. The history of a process over a thousand days tells you nothing about what is to happen next. This naïve projection of the future from the past can be applied to anything.

**Source:** Page 41 of [The Black Swan: The Impact of the Highly Improbable](#) by Nassim Nicholas Taleb  
(turkey graphics added by me)

**Example of the turkey problem.** A turkey might live 1000 good days and none of them would be a sign of what's to happen on day 1001. Similar with forecasting, your historical data may not have any indication of a change which is about to come. The graph image is from page 41 of [The Black Swan](#) by Nassim Taleb (I added in the turkey graphics).

How does this relate to predicting the price of Bitcoin (or the price of any stock or figure in an open market)?

You could have the historical data of Bitcoin for its entire existence and build a model which predicts it perfectly.

But then one day for some unknown and unpredictable reason, the price of Bitcoin plummets 100x in a single day.

Of course, this kind of scenario is unlikely.

But that doesn't take away from its significance.

Think about it in your own life, how many times have the most significant events happened seemingly out of the blue?

As in, you could go to a cafe and run into the love of your life, despite visiting the same cafe for 10-years straight and never running into this person before.

The same thing goes for predicting the price of Bitcoin, you could make money for 10-years straight and then lose it all in a single day.

It doesn't matter how many times you get paid, it matters the amount you get paid.

Resource: If you'd like to learn more about the turkey problem, I'd recommend the following:

- [Explaining both the XIV trade and why forecasting is BS](#) by Nassim Taleb
- [The Black Swan](#) by Nassim Taleb (especially Chapter 4 which outlines and discusses the turkey problem)

Let's get specific and see how the turkey problem effects us modelling the historical and future price of Bitcoin.

To do so, we're going to manufacture a highly unlikely data point into the historical price of Bitcoin, the price falling 100x in one day.

Note: A very unlikely and unpredictable event such as the price of Bitcoin falling 100x in a single day (note: the adjective "unlikely" is based on the historical price changes of Bitcoin) is also referred to a [Black Swan event](#). A Black Swan event is an unknown unknown, you have no way of predicting whether or not it will happen but these kind of events often have a large impact.

In [ ]:

```
# Let's introduce a Turkey problem to our BTC data (price BTC falls 100x in one day)
btc_price_turkey = btc_price.copy()
btc_price_turkey[-1] = btc_price_turkey[-1] / 100
```

In [ ]:

```
# Manufacture an extra price on the end (to showcase the Turkey problem)
btc_price_turkey[-10:]
```

Out[ ]:

```
[58788.2096789273,
 58102.1914262342,
 55715.5466512869,
 56573.5554719043,
 52147.8211869823,
 49764.1320815975,
 50032.6931367648,
 47885.6252547166,
 45604.6157536131,
 431.44471290860304]
```

Notice the last value is 100x lower than what it actually was (remember, this is not a real data point, its only to illustrate the effects of the turkey problem).

Now we've got Bitcoin prices including a turkey problem data point, let's get the timesteps.

In [ ]:

```
# Get the timesteps for the turkey problem
btc_timesteps_turkey = np.array(bitcoin_prices.index)
btc_timesteps_turkey[-10:]
```

Out[ ]:

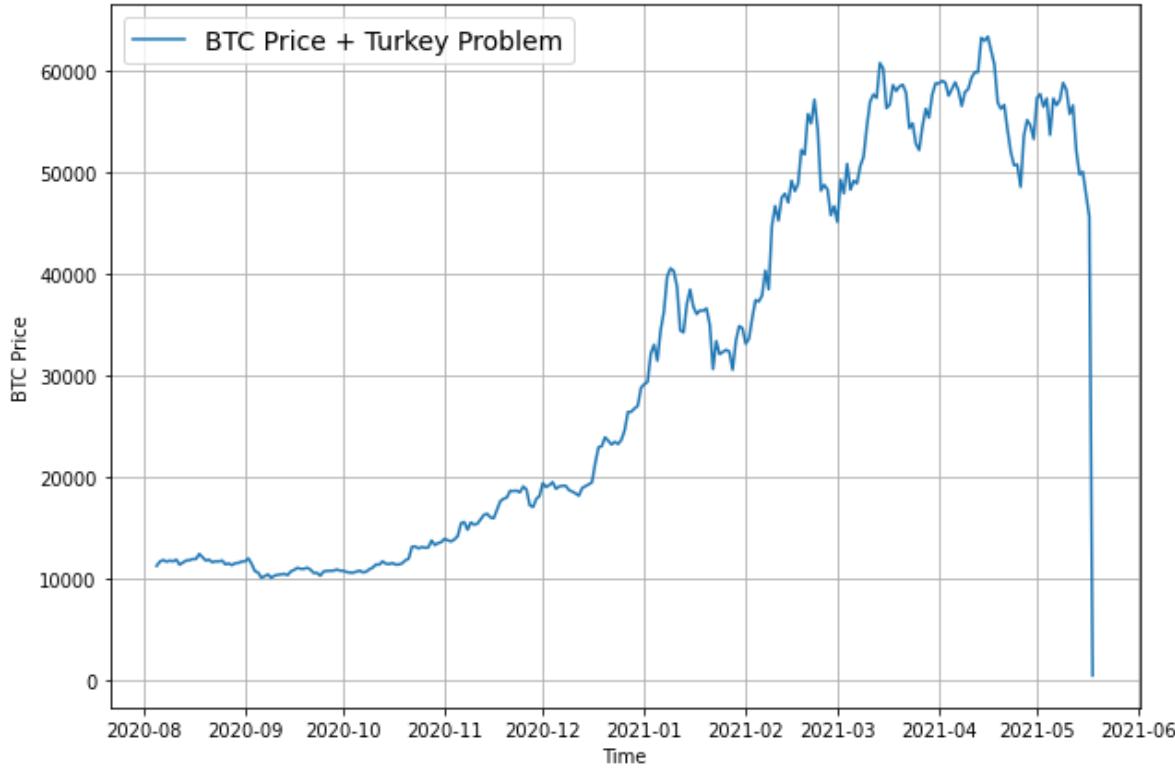
```
array(['2021-05-09T00:00:00.000000000', '2021-05-10T00:00:00.000000000',
       '2021-05-11T00:00:00.000000000', '2021-05-12T00:00:00.000000000']
```

```
'2021-05-13T00:00:00.000000000', '2021-05-14T00:00:00.000000000',
'2021-05-15T00:00:00.000000000', '2021-05-16T00:00:00.000000000',
'2021-05-17T00:00:00.000000000', '2021-05-18T00:00:00.000000000'],
dtype='datetime64[ns]')
```

**Beautiful! Let's see our artificially created turkey problem Bitcoin data.**

In [ ]:

```
plt.figure(figsize=(10, 7))
plot_time_series(timesteps=btc_timesteps_turkey,
                  values=btc_price_turkey,
                  format="--",
                  label="BTC Price + Turkey Problem",
                  start=2500)
```



**How do you think building a model on this data will go?**

**Remember, all we've changed is a single data point out of our entire dataset.**

**Before we build a model, let's create some windowed datasets with our turkey data.**

In [ ]:

```
# Create train and test sets for turkey problem data
full_windows, full_labels = make_windows(np.array(btc_price_turkey), window_size=WINDOW_SIZE, horizon=HORIZON)
len(full_windows), len(full_labels)

X_train, X_test, y_train, y_test = make_train_test_splits(full_windows, full_labels)
len(X_train), len(X_test), len(y_train), len(y_test)
```

Out [ ]:

```
(2224, 556, 2224, 556)
```

## **Building a turkey model (model to predict on turkey data)**

With our updated data, we only changed 1 value.

Let's see how it effects a model.

To keep things comparable to previous models, we'll create a `turkey_model` which is a clone of `model_1`

(same architecture, but different data).

That way, when we evaluate the `turkey_model` we can compare its results to `model_1_results` and see how much a single data point can influence a model's performance.

In [ ]:

```
# Clone model 1 architecture for turkey model and fit the turkey model on the turkey data
turkey_model = tf.keras.models.clone_model(model_1)
turkey_model._name = "Turkey_Model"
turkey_model.compile(loss="mae",
                      optimizer=tf.keras.optimizers.Adam())
turkey_model.fit(X_train, y_train,
                  epochs=100,
                  verbose=0,
                  validation_data=(X_test, y_test),
                  callbacks=[create_model_checkpoint(turkey_model.name)])
```

INFO:tensorflow:Assets written to: model\_experiments/Turkey\_Model/assets

```
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
```

```
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
```

Out[ ]:

```
<keras.callbacks.History at 0x7fdc7a4dd550>
```

In [ ]:

```
# Evaluate turkey model on test data
turkey_model.evaluate(X_test, y_test)
```

```
18/18 [=====] - 0s 2ms/step - loss: 696.1285
```

Out[ ]:

```
696.1284790039062
```

In [ ]:

```
# Load best model and evaluate on test data
turkey_model = tf.keras.models.load_model("model_experiments/Turkey_Model/")
turkey_model.evaluate(X_test, y_test)
```

```
18/18 [=====] - 0s 2ms/step - loss: 638.3047
```

Out[ ]:

```
638.3046875
```

**Alright, now let's make some predictions with our model and evaluate them on the test data.**

In [ ]:

```
# Make predictions with Turkey model
turkey_preds = make_preds(turkey_model, X_test)
turkey_preds[:10]
```

Out[ ]:

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([8858.391, 8803.98 , 9039.575, 8785.937, 8778.044, 8735.638,
       8684.118, 8558.659, 8461.373, 8542.206], dtype=float32)>
```

In [ ]:

```
# Evaluate turkey preds
turkey_results = evaluate_preds(y_true=y_test,
                                 y_pred=turkey_preds)
turkey_results
```

Out[ ]:

```
{'mae': 17144.766,
 'mape': 121.58286,
 'mase': 26.53158,
 'mse': 615487800.0,
 'rmse': 23743.305}
```

**And with just one value change, our error metrics go through the roof.**

**To make sure, let's remind ourselves of how model\_1 went on unmodified Bitcoin data (no turkey problem).**

In [ ]:

```
model_1_results
```

Out[ ]:

```
{'mae': 568.95123,
 'mape': 2.5448983,
 'mase': 0.9994897.}
```

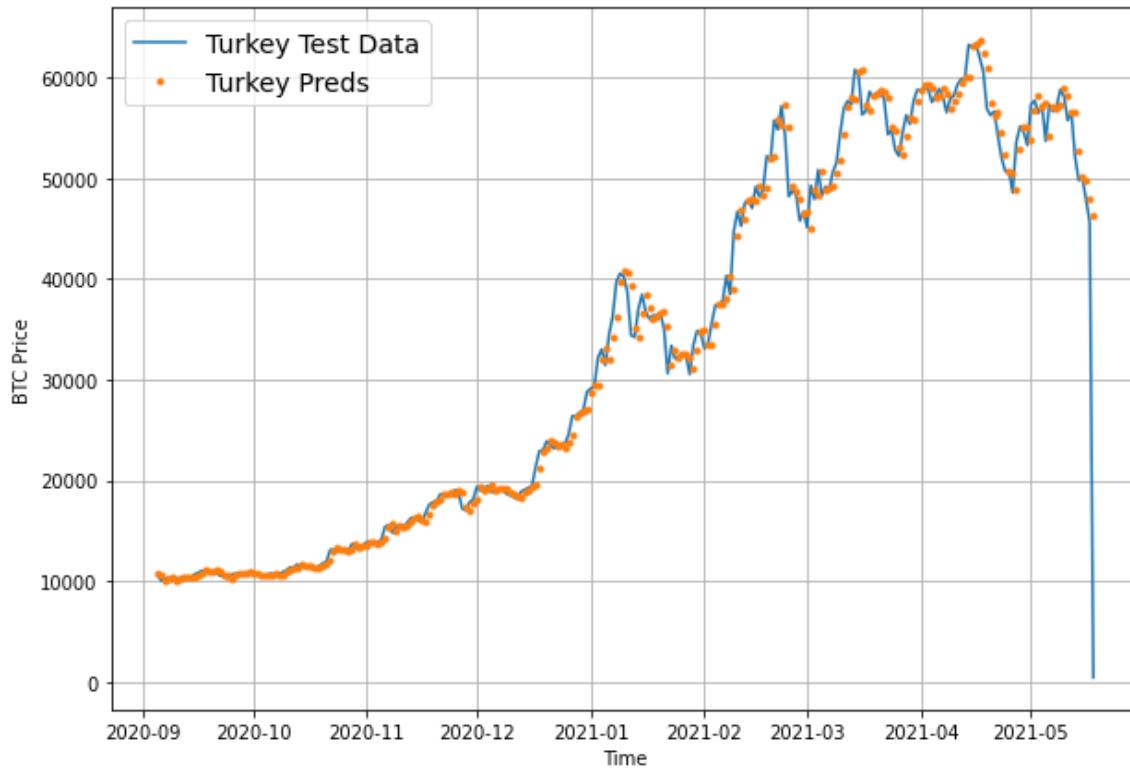
```
'mse': 1171744.0,  
'rmse': 1082.4713}
```

By changing just one value, the `turkey_model` MAE increases almost 30x over `model_1`.

Finally, we'll visualize the turkey predictions over the test turkey data.

In [ ]:

```
plt.figure(figsize=(10, 7))  
# plot_time_series(timesteps=btc_timesteps_turkey[:split_size], values=btc_price_turkey[:  
split_size], label="Train Data")  
offset=300  
plot_time_series(timesteps=btc_timesteps_turkey[-len(X_test):],  
values=btc_price_turkey[-len(y_test):],  
format="--",  
label="Turkey Test Data", start=offset)  
plot_time_series(timesteps=btc_timesteps_turkey[-len(X_test):],  
values=turkey_preds,  
label="Turkey Preds",  
start=offset);
```



Why does this happen?

Why does our model fail to capture the turkey problem data point?

Think about it like this, just like a turkey who lives 1000 joyful days, based on observation alone has no reason to believe day 1001 won't be as joyful as the last, a model which has been trained on historical data of Bitcoin which has no single event where the price decreased by 100x in a day, has no reason to predict it will in the future.

A model cannot predict anything in the future outside of the distribution it was trained on.

In turn, highly unlikely price movements (based on historical movements), upward or downward will likely never be part of a forecast.

However, as we've seen, despite their unlikeliness, these events can have huuuuuuuuuge impacts to the performance of our models.

Resource: For a great article which discusses Black Swan events and how they often get ignored due to the assumption that historical events come from a certain distribution and that future events will come from the same distribution see [Black Swans, Normal Distributions and Supply](#)

## Compare Models

We've trained a bunch of models.

And if anything, we've seen just how poorly machine learning and deep learning models are at forecasting the price of Bitcoin (or any kind of open market value).

To highlight this, let's compare the results of all of the modelling experiments we've performed so far.

In [ ]:

```
# Compare different model results (w = window, h = horizon, e.g. w=7 means a window size
# of 7)
model_results = pd.DataFrame({"naive_model": naive_results,
                               "model_1_dense_w7_h1": model_1_results,
                               "model_2_dense_w30_h1": model_2_results,
                               "model_3_dense_w30_h7": model_3_results,
                               "model_4_CONV1D": model_4_results,
                               "model_5_LSTM": model_5_results,
                               "model_6_multivariate": model_6_results,
                               "model_8_NBEATs": model_7_results,
                               "model_9_ensemble": ensemble_results,
                               "model_10_turkey": turkey_results}).T
model_results.head(10)
```

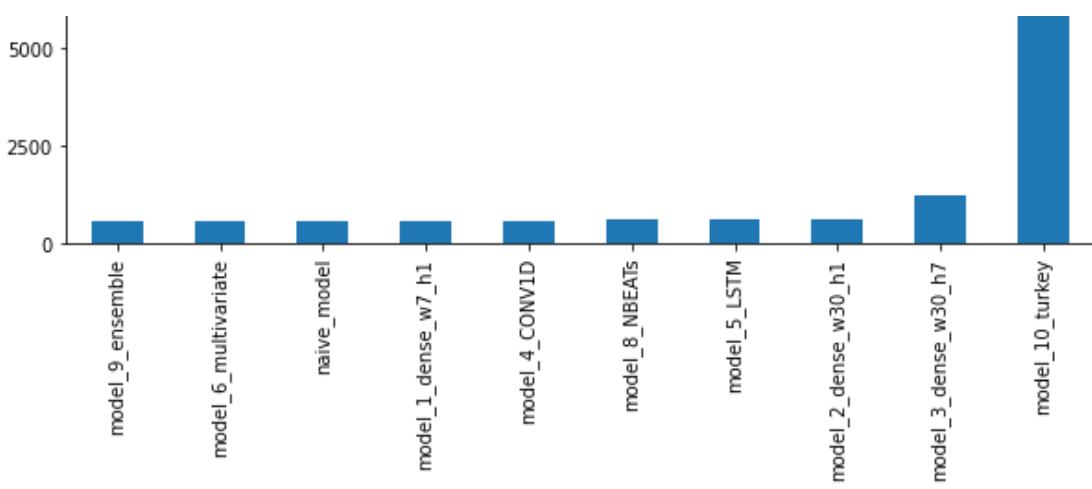
Out[ ]:

	mae	mse	rmse	mape	mase
naive_model	567.980225	1.147547e+06	1071.236206	2.516525	0.999570
model_1_dense_w7_h1	568.951233	1.171744e+06	1082.471313	2.544898	0.999490
model_2_dense_w30_h1	608.961487	1.281439e+06	1132.006470	2.769339	1.064471
model_3_dense_w30_h7	1237.506348	5.405198e+06	1425.747681	5.558878	2.202074
model_4_CONV1D	570.828308	1.176671e+06	1084.744751	2.559336	1.002787
model_5_LSTM	596.644653	1.273487e+06	1128.488770	2.683845	1.048139
model_6_multivariate	567.587402	1.161688e+06	1077.816528	2.541387	0.997094
model_8_NBEATs	585.499817	1.179492e+06	1086.043945	2.744519	1.028561
model_9_ensemble	567.442322	1.144513e+06	1069.819092	2.584332	0.996839
model_10_turkey	17144.765625	6.154878e+08	23743.304688	121.582863	26.531580

In [ ]:

```
# Sort model results by MAE and plot them
model_results[["mae"]].sort_values(by="mae").plot(figsize=(10, 7), kind="bar");
```





The majority of our deep learning models perform on par or only slightly better than the naive model. And for the turkey model, changing a single data point destroys its performance.

□ Note: Just because one type of model performs better here doesn't mean it'll perform the best elsewhere (and vice versa, just because one model performs poorly here, doesn't mean it'll perform poorly elsewhere).

As I said at the start, this is not financial advice.

After what we've gone through, you'll now have some of the skills required to callout BS for any future tutorial or blog post or investment sales guide claiming to have model which is able to predict the future.

[Mark Saroufim's Tweet](#) sums this up nicely (stock market forecasting with a machine learning model is just as reliable as palm reading).

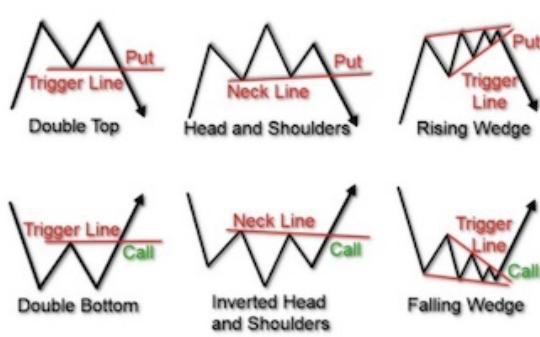


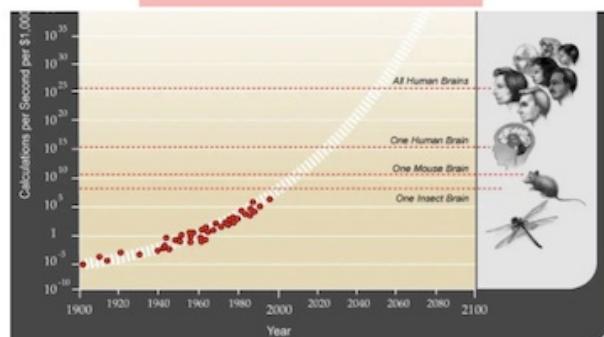
**Mark Saroufim** @marksaroufim · Mar 3  
Same vibe

...

### Time-Series Forecasting: Predicting Stock Prices Using An LSTM Model

In this post I show you how to predict stock prices using a forecasting LSTM model



*Beware the tutorials or trading courses which claim to use some kind of algorithm to beat the market (an open system), they're likely a scam or the creator is very lucky and hasn't yet come across a turkey problem.*

Don't let these results get you down though, forecasting in a closed system (such as predicting the demand of electricity) often yields quite usable results.

If anything, this module teaches anti-knowledge. Knowing that forecasting methods usually *don't* perform well in open systems.

## Exercises

1. Does scaling the data help for univariate/multivariate data? (e.g. getting all of the values between 0 & 1)
  - Try doing this for a univariate model (e.g. `model_1`) and a multivariate model (e.g. `model_6`) and see if it effects model training or evaluation results.
2. Get the most up to date data on Bitcoin, train a model & see how it goes (our data goes up to May 18 2021).
  - You can download the Bitcoin historical data for free from [coindesk.com/price/bitcoin](https://coindesk.com/price/bitcoin) and clicking "Export Data" -> "CSV".
3. For most of our models we used `WINDOW_SIZE=7`, but is there a better window size?
  - Setup a series of experiments to find whether or not there's a better window size.
  - For example, you might train 10 different models with `HORIZON=1` but with window sizes ranging from 2-12.
4. Create a windowed dataset just like the ones we used for `model_1` using  
`tf.keras.preprocessing.timeseries_dataset_from_array()` and retrain `model_1` using the recreated dataset.
5. For our multivariate modelling experiment, we added the Bitcoin block reward size as an extra feature to make our time series multivariate.
  - Are there any other features you think you could add?
  - If so, try it out, how do these affect the model?
6. Make prediction intervals for future forecasts. To do so, one way would be to train an ensemble model on all of the data, make future forecasts with it and calculate the prediction intervals of the ensemble just like we did for `model_8`.
7. For future predictions, try to make a prediction, retrain a model on the predictions, make a prediction, retrain a model, make a prediction, retrain a model, make a prediction (retrain a model each time a new prediction is made). Plot the results, how do they look compared to the future predictions where a model wasn't retrained for every forecast (`model_9`)?
8. Throughout this notebook, we've only tried algorithms we've handcrafted ourselves. But it's worth seeing how a purpose built forecasting algorithm goes.
  - Try out one of the extra algorithms listed in the modelling experiments part such as:
    - [Facebook's Kats library](#) - there are many models in here, remember the machine learning practitioner's motto: experiment, experiment, experiment.
    - [LinkedIn's Greykite library](#)

## Extra-curriculum

We've only really scratched the surface with time series forecasting and time series modelling in general. But the good news is, you've got plenty of hands-on coding experience with it already.

If you'd like to dig deeper in to the world of time series, I'd recommend the following:

- [Forecasting: Principles and Practice](#) is an outstanding online textbook which discusses at length many of the most important concepts in time series forecasting. I'd especially recommend reading at least Chapter 1 in full.
  - I'd definitely recommend at least checking out chapter 1 as well as the chapter on forecasting accuracy measures.
- [Introduction to machine learning and time series](#) by Markus Loning goes through different time series problems and how to approach them. It focuses on using the `sktime` library (Scikit-Learn for time series), though the principles are applicable elsewhere.
- [Why you should care about the Nate Silver vs. Nassim Taleb Twitter war](#) by Isaac Faber is an outstanding discussion insight into the role of uncertainty in the example of election prediction.
- [TensorFlow time series tutorial](#) - A tutorial on using TensorFlow to forecast weather time series data with TensorFlow.
- [The Black Swan](#) by Nassim Nicholas Taleb - Nassim Taleb was a pit trader (a trader who trades on their own behalf) for 25 years, this book compiles many of the lessons he learned from first-hand experience. It changed my whole perspective on our ability to predict.
- [3 facts about time series forecasting that surprise experienced machine learning practitioners](#) by Skander

Hannachi, Ph.D - time series data is different to other kinds of data, if you've worked on other kinds of machine learning problems before, getting into time series might require some adjustments, Hannachi outlines 3 of the most common.

- □ World-class lectures by Jordan Kern, watching these will take you from 0 to 1 with time series problems:
  - [Time Series Analysis](#) - how to analyse time series data.
  - [Time Series Modelling](#) - different techniques for modelling time series data (many of which aren't deep learning).