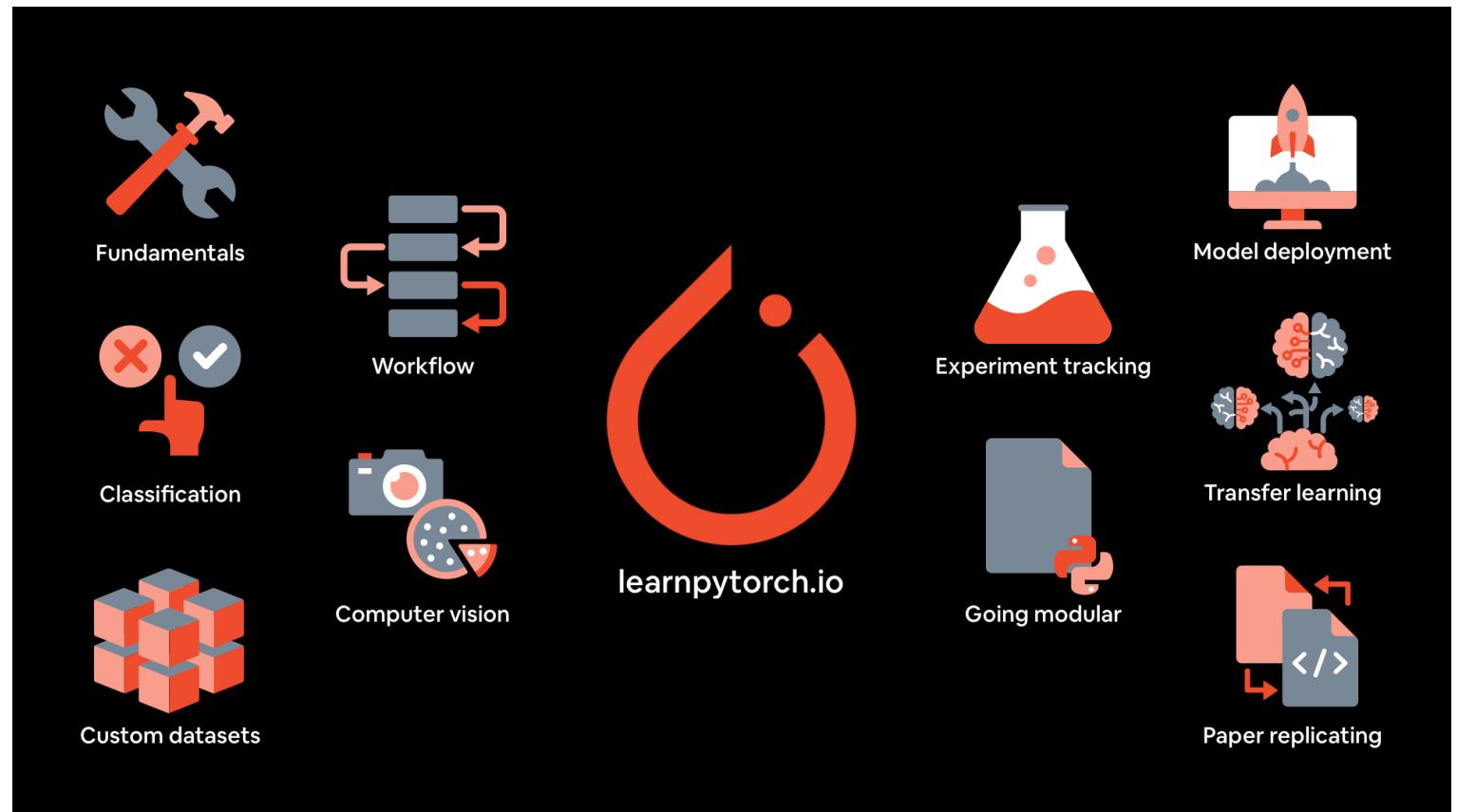


# Learn PyTorch for Deep Learning: Zero to Mastery book



Welcome to the second best place on the internet to learn PyTorch (the first being the PyTorch documentation).

This is the online book version of the Learn PyTorch for Deep Learning: Zero to Mastery course.

This course will teach you the foundations of machine learning and deep learning with PyTorch (a machine learning framework written in Python).

The course is video based. However, the videos are based on the contents of this online book.

For full code and resources see the course GitHub.

Otherwise, you can find more about the course below.

# Status

Course launched on ZTM Academy!

- Last update: August 31 2022
- Videos are done for chapters: 00, 01, 02, 03, 04, 05, 06, 07, 08, 09 (all chapters!)
- Currently working on: editing and uploading videos for 08 and 09
- See progress on the course GitHub Project.

**Get updates:** Follow the `pytorch-deep-learning` repo log or sign up for emails.

## Course materials/outline

- **Code on GitHub:** All of course materials are available open-source on GitHub.
- **First five sections on YouTube:** Learn Pytorch in a day by watching the first 25-hours of material.
- **Course focus:** code, code, experiment, experiment, experiment.
- ⚡ **Teaching style:** <https://sive.rs/kimo>.
- **Ask a question:** See the course GitHub Discussions page for existing questions/ask your own.

Section	What does it cover?	Exercises & Extra-curriculum	Slides
00 - PyTorch Fundamentals	Many fundamental PyTorch operations used for deep learning and neural networks.	Go to exercises & extra-curriculum	Go to slides
01 - PyTorch Workflow	Provides an outline for approaching deep learning problems and building neural networks with PyTorch.	Go to exercises & extra-curriculum	Go to slides
02 - PyTorch Neural Network Classification	Uses the PyTorch workflow from 01 to go through a neural network classification problem.	Go to exercises & extra-curriculum	Go to slides
03 - PyTorch Computer	Let's see how PyTorch can be used for computer vision problems using the same workflow from 01 & 02.	Go to exercises &	Go to slides

Vision

extra-curriculum

04 - PyTorch  
Custom  
Datasets

How do you load a custom dataset into PyTorch? Also we'll be laying the foundations in this notebook for our modular code (covered in 05).

Go to exercises & extra-curriculum

Go to slides

05 - PyTorch  
Going  
Modular

PyTorch is designed to be modular, let's turn what we've created into a series of Python scripts (this is how you'll often find PyTorch code in the wild).

Go to exercises & extra-curriculum

Go to slides

06 - PyTorch  
Transfer  
Learning

Let's take a well performing pre-trained model and adjust it to one of our own problems.

Go to exercises & extra-curriculum

Go to slides

07 - Milestone  
Project 1:  
PyTorch  
Experiment  
Tracking

We've built a bunch of models... wouldn't it be good to track how they're all going?

Go to exercises & extra-curriculum

Go to slides

08 - Milestone  
Project 2:  
PyTorch  
Paper  
Replicating

PyTorch is the most popular deep learning framework for machine learning research, let's see why by replicating a machine learning paper.

Go to exercises & extra-curriculum

Go to slides

09 - Milestone  
Project 3:  
Model  
Deployment

So we've built a working PyTorch model... how do we get it in the hands of others? Hint: deploy it to the internet.

Go to exercises & extra-curriculum

Go to slides

PyTorch  
Extra  
Resources

This course covers a large amount of PyTorch and deep learning but the field of machine learning is vast, inside here you'll find recommended books and resources for: PyTorch and deep learning, ML engineering, NLP (natural language processing), time series data, where to find datasets and more.

-

-

## About this course

### Who is this course for?

**You:** Are a beginner in the field of machine learning or deep learning and would like to learn PyTorch.

**This course:** Teaches you PyTorch and many machine learning concepts in a hands-on, code-first way.

If you already have 1-year+ experience in machine learning, this course may help but it is specifically designed to be beginner-friendly.

## What are the prerequisites?

1. 3-6 months coding Python.
2. At least one beginner machine learning course (however this might be able to be skipped, resources are linked for many different topics).
3. Experience using Jupyter Notebooks or Google Colab (though you can pick this up as we go along).
4. A willingness to learn (most important).

For 1 & 2, I'd recommend the Zero to Mastery Data Science and Machine Learning Bootcamp, it'll teach you the fundamentals of machine learning and Python (I'm biased though, I also teach that course).

## How is the course taught?

All of the course materials are available for free in an online book at [learnpytorch.io](https://learnpytorch.io). If you like to read, I'd recommend going through the resources there.

If you prefer to learn via video, the course is also taught in apprenticeship-style format, meaning I write PyTorch code, you write PyTorch code.

There's a reason the course motto's include *if in doubt, run the code* and *experiment, experiment, experiment!*.

My whole goal is to help you to do one thing: learn machine learning by writing PyTorch code.

The code is all written via Google Colab Notebooks (you could also use Jupyter Notebooks), an incredible free resource to experiment with machine learning.

## What will I get if I finish the course?

There's certificates and all that jazz if you go through the videos.

But certificates are meh.

You can consider this course a machine learning momentum builder.

By the end, you'll have written hundreds of lines of PyTorch code.

And will have been exposed to many of the most important concepts in machine learning.

So when you go to build your own machine learning projects or inspect a public machine learning project made with PyTorch, it'll feel familiar and if it doesn't, at least you'll know where to look.

## What will I build in the course?

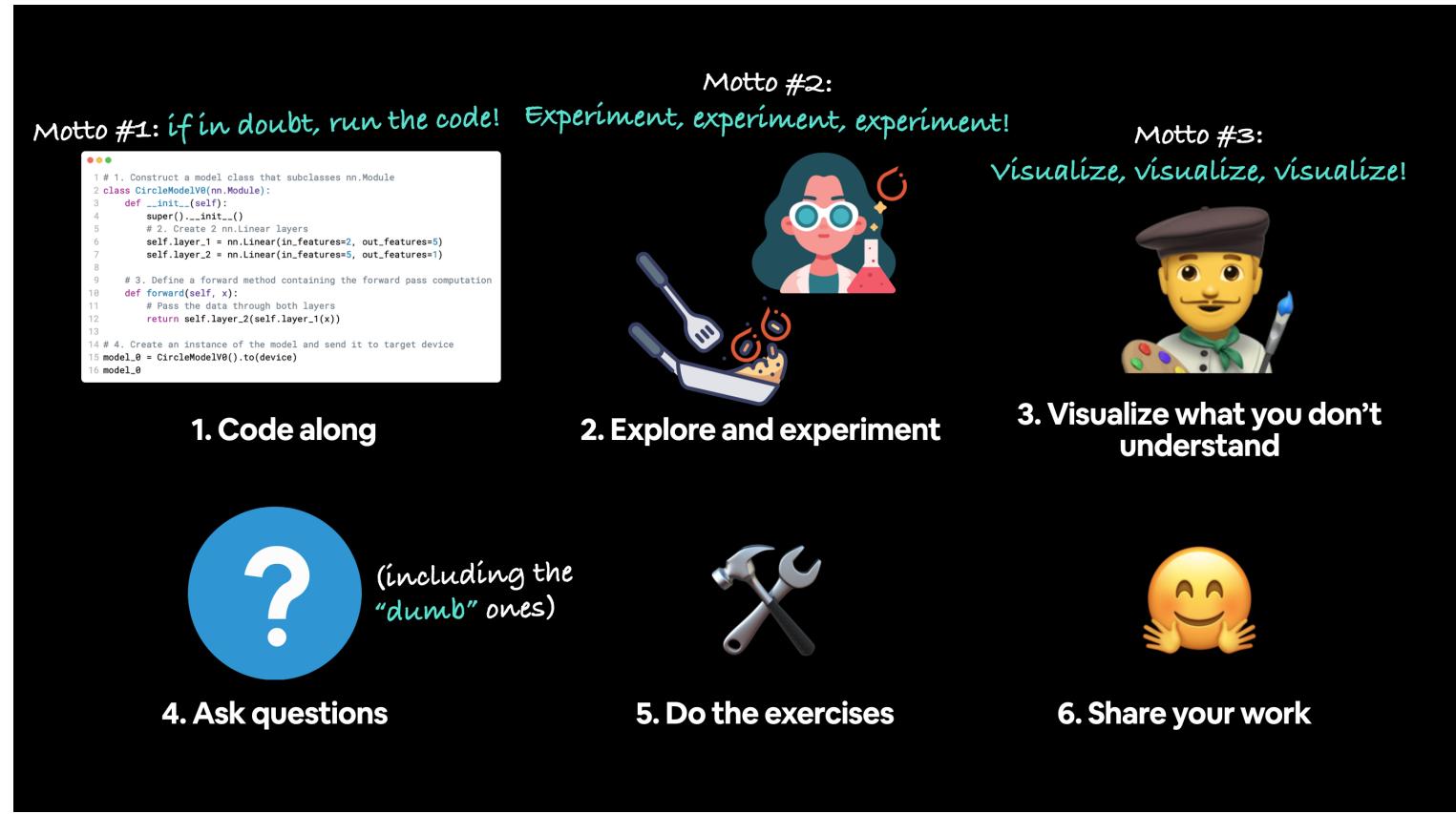
We start with the barebone fundamentals of PyTorch and machine learning, so even if you're new to machine learning you'll be caught up to speed.

Then we'll explore more advanced areas including PyTorch neural network classification, PyTorch workflows, computer vision, custom datasets, experiment tracking, model deployment, and my personal favourite: transfer learning, a powerful technique for taking what one machine learning model has learned on another problem and applying it to your own!

Along the way, you'll build three milestone projects surrounding an overarching project called FoodVision, a neural network computer vision model to classify images of food.

These milestone projects will help you practice using PyTorch to cover important machine learning concepts and create a portfolio you can show employers and say "here's what I've done".

## How do I approach this course?



As mentioned, the video version of the course is taught apprenticeship style.

Meaning I write PyTorch code, you write PyTorch code.

But here's what I recommend:

1. **Code along (*if in doubt, run the code!*)** - Follow along with code and try to write as much of it as you can yourself, keep doing so until you find yourself writing PyTorch code in your subconscious that's when you can stop writing the same code over and over again.
2. **Explore and experiment (*experiment, experiment, experiment!*)** - Machine learning (and deep learning) is very experimental. So if you find yourself wanting to try something on your own and ignoring the materials, do it.
3. **Visualize what you don't understand (*visualize, visualize, visualize!*)** - Numbers on a page can get confusing. So make things colourful, see what the inputs and outputs of your code looks like.
4. **Ask questions** - If you're stuck with something, ask a question, trying searching for it or if nothing comes up, the course GitHub Discussions page will be the place to go.
5. **Do the exercises** - Each module of the course comes with a dedicated exercices section. It's important to try these on your own. You will get stuck. But that's the nature of learning something new: everyone gets stuck.
6. **Share your work** - If you've learned something cool or even better, made something cool, share it. It could be with the course Discord group or on the course GitHub page or on your own website. The benefits of sharing your work is you get to practice communicating as well as others can help you out if you're not sure of

something.

## Do I need to take things in order?

The notebooks/chapters build upon each other sequentially but feel free to jump around.

## How do I get started?

You can read the materials on any device but this course is best viewed and coded along within a desktop browser.

The course uses a free tool called Google Colab. If you've got no experience with it, I'd go through the free Introduction to Google Colab tutorial and then come back here.

To start:

1. Click on one of the notebook or section links like "00. PyTorch Fundamentals".
2. Click the "Open in Colab" button up the top.
3. Press SHIFT+Enter a few times and see what happens.

Happy machine learning!

Next  
00. PyTorch Fundamentals

Made with Material for MkDocs Insiders



[View Source Code](#) | [View Slides](#) | [Watch Video Walkthrough](#)

## 00. PyTorch Fundamentals

### What is PyTorch?

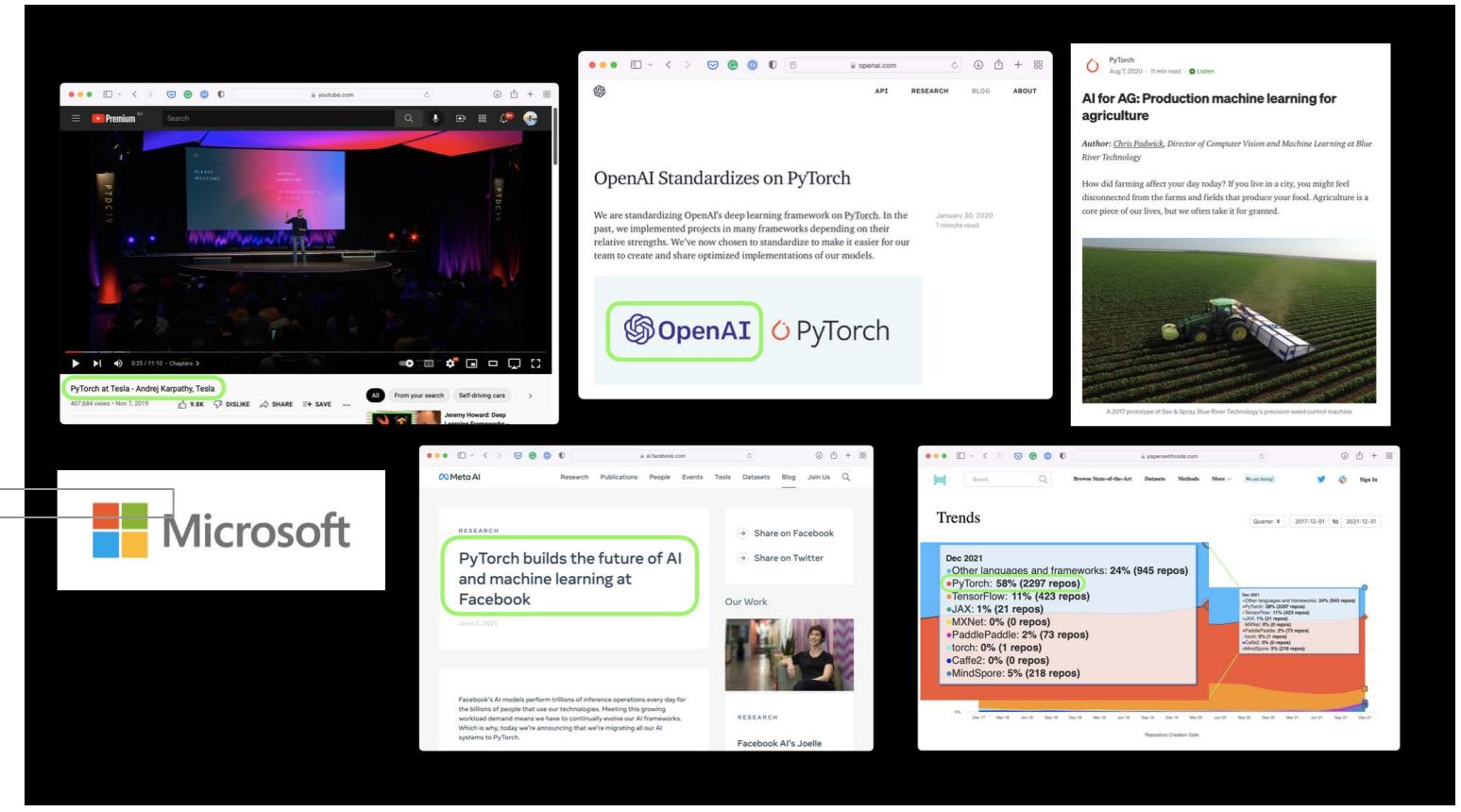
PyTorch is an open source machine learning and deep learning framework.

### What can PyTorch be used for?

PyTorch allows you to manipulate and process data and write machine learning algorithms using Python code.

### Who uses PyTorch?

Many of the worlds largest technology companies such as [Meta \(Facebook\)](#), Tesla and Microsoft as well as artificial intelligence research companies such as [OpenAI](#) use PyTorch to power research and bring machine learning to their products.



For example, Andrej Karpathy (head of AI at Tesla) has given several talks ([PyTorch DevCon 2019](#), [Tesla AI Day 2021](#)) about how Tesla use PyTorch to power their self-driving computer vision models.

PyTorch is also used in other industries such as agriculture to [power computer vision on tractors](#).

## Why use PyTorch?

Machine learning researchers love using PyTorch. And as of February 2022, PyTorch is the [most used deep learning framework on Papers With Code](#), a website for tracking machine learning research papers and the code repositories attached with them.

PyTorch also helps take care of many things such as GPU acceleration (making your code run faster) behind the scenes.

So you can focus on manipulating data and writing algorithms and PyTorch will make sure it runs fast.

And if companies such as Tesla and Meta (Facebook) use it to build models they deploy to power hundreds of applications, drive thousands of cars and deliver content to billions of people, it's clearly capable on the development front too.

## What we're going to cover in this module

This course is broken down into different sections (notebooks).

Each notebook covers important ideas and concepts within PyTorch.

Subsequent notebooks build upon knowledge from the previous one (numbering starts at 00, 01, 02 and goes to whatever it ends up going to).

This notebook deals with the basic building block of machine learning and deep learning, the tensor.

Specifically, we're going to cover:

Topic	Contents
<b>Introduction to tensors</b>	Tensors are the basic building block of all of machine learning and deep learning.
<b>Creating tensors</b>	Tensors can represent almost any kind of data (images, words, tables of numbers).
<b>Getting information from tensors</b>	If you can put information into a tensor, you'll want to get it out too.
<b>Manipulating tensors</b>	Machine learning algorithms (like neural networks) involve manipulating tensors in many different ways such as adding, multiplying, combining.
<b>Dealing with tensor shapes</b>	One of the most common issues in machine learning is dealing with shape mismatches (trying to mixed wrong shaped tensors with other tensors).
<b>Indexing on tensors</b>	If you've indexed on a Python list or NumPy array, it's very similar with tensors, except they can have far more dimensions.
<b>Mixing PyTorch tensors and NumPy</b>	PyTorch plays with tensors ( <code>torch.Tensor</code> ), NumPy likes arrays ( <code>np.ndarray</code> ) sometimes you'll want to mix and match these.
<b>Reproducibility</b>	Machine learning is very experimental and since it uses a lot of <i>randomness</i> to work, sometimes you'll want that <i>randomness</i> to not be so random.
<b>Running tensors on GPU</b>	GPUs (Graphics Processing Units) make your code faster, PyTorch makes it easy to run your code on GPUs.

## Where can you get help?

All of the materials for this course [live on GitHub](#).

And if you run into trouble, you can ask a question on the [Discussions page](#) there too.

There's also the [PyTorch developer forums](#), a very helpful place for all things PyTorch.

## Importing PyTorch

**Note:** Before running any of the code in this notebook, you should have gone through the [PyTorch setup steps](#).

However, **if you're running on Google Colab**, everything should work (Google Colab comes with PyTorch and other libraries installed).

Let's start by importing PyTorch and checking the version we're using.

In [1]:

```
'1.12.0+cu102'
```

Out[1]:

Wonderful, it looks like we've got PyTorch 1.10.0+.

This means if you're going through these materials, you'll see most compatibility with PyTorch 1.10.0+, however if your version number is far higher than that, you might notice some inconsistencies.

And if you do have any issues, please post on the course [GitHub Discussions page](#).

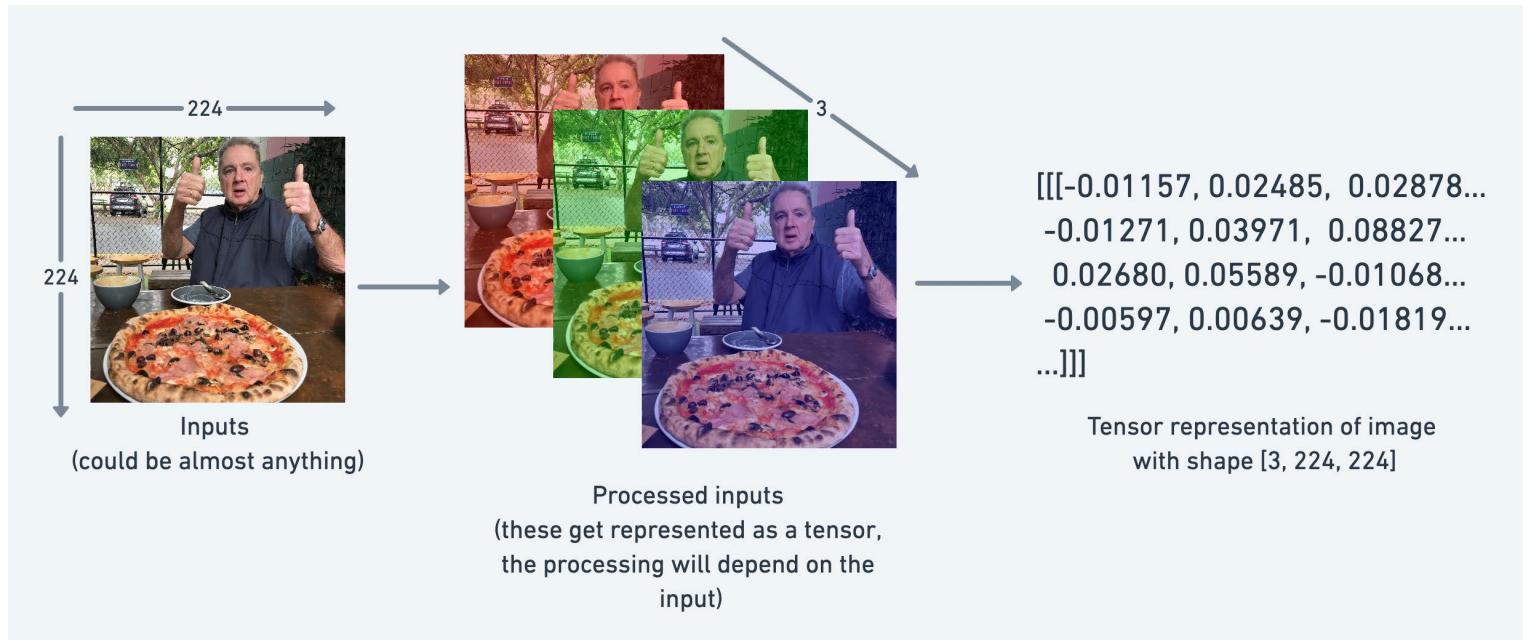
## Introduction to tensors

Now we've got PyTorch imported, it's time to learn about tensors.

Tensors are the fundamental building block of machine learning.

Their job is to represent data in a numerical way.

For example, you could represent an image as a tensor with shape `[3, 224, 224]` which would mean `[colour_channels, height, width]`, as in the image has 3 colour channels (red, green, blue), a height of 224 pixels and a width of 224 pixels.



In tensor-speak (the language used to describe tensors), the tensor would have three dimensions, one for `colour_channels`, `height` and `width`.

But we're getting ahead of ourselves.

Let's learn more about tensors by coding them.

## Creating tensors

PyTorch loves tensors. So much so there's a whole documentation page dedicated to the `torch.Tensor` class.

Your first piece of homework is to [read through the documentation on `torch.Tensor`](#) for 10-minutes. But you can get to that later.

Let's code.

The first thing we're going to create is a **scalar**.

A scalar is a single number and in tensor-speak it's a zero dimension tensor.

**Note:** That's a trend for this course. We'll focus on writing specific code. But often I'll set exercises which involve reading and getting familiar with the PyTorch documentation. Because after all, once you're finished this course, you'll no doubt want to learn more. And the documentation is somewhere you'll be finding yourself quite often.

In [2]:

Out[2]:

```
tensor(7)
```

See how the above printed out `tensor(7)` ?

That means although `scalar` is a single number, it's of type `torch.Tensor`.

We can check the dimensions of a tensor using the `ndim` attribute.

In [3]:

Out[3]:

```
0
```

What if we wanted to retrieve the number from the tensor?

As in, turn it from `torch.Tensor` to a Python integer?

To do we can use the `item()` method.

In [4]:

7

Okay, now let's see a **vector**.

A vector is a single dimension tensor but can contain many numbers.

As in, you could have a vector [3, 2] to describe [bedrooms, bathrooms] in your house. Or you could have [3, 2, 2] to describe [bedrooms, bathrooms, car\_parks] in your house.

The important trend here is that a vector is flexible in what it can represent (the same with tensors).

In [5]:

```
tensor([7, 7])
```

Wonderful, `vector` now contains two 7's, my favourite number.

How many dimensions do you think it'll have?

In [6]:

Out[6]:

1

Hmm, that's strange, `vector` contains two numbers but only has a single dimension.

I'll let you in on a trick.

You can tell the number of dimensions a tensor in PyTorch has by the number of square brackets on the outside ( `[ ]` ) and you only need to count one side.

How many square brackets does `vector` have?

Another important concept for tensors is their `shape` attribute. The shape tells you how the elements inside them are arranged.

Let's check out the shape of `vector`.

In [7]:

Out[7]:

```
torch.Size([2])
```

The above returns `torch.Size([2])` which means our vector has a shape of [2]. This is because of the two elements we placed inside the square brackets ([7, 7]).

Let's now see a **matrix**.

In [8]:

```
tensor([[ 7,  8],  
       [ 9, 10]])
```

Out[8]:

Wow! More numbers! Matrices are as flexible as vectors, except they've got an extra dimension.

In [9]:

2

`MATRIX` has two dimensions (did you count the number of square brackets on the outside of one side?).

What `shape` do you think it will have?

Out[9]:

In [10]:

`torch.Size([2, 2])`

Out[10]:

We get the output `torch.Size([2, 2])` because `MATRIX` is two elements deep and two elements wide.

How about we create a **tensor**?

In [11]:



Out[11]:

```
tensor([[ [1, 2, 3],
          [3, 6, 9],
          [2, 4, 5]]])
```

Woah! What a nice looking tensor.

I want to stress that tensors can represent almost anything.

The one we just created could be the sales numbers for a steak and almond butter store (two of my favourite foods).

	A	B	C	D	E	F	G	H	I
1	<b>Day of week</b>	1	2	3					
2	<b>Steak sales</b>	3	6	9					
3	<b>Almond butter sales</b>	2	4	5					
4									
5									
6									
7									

How many dimensions do you think it has? (hint: use the square bracket counting trick)

In [12]:

Out[12]:

3

And what about its shape?

In [13]:

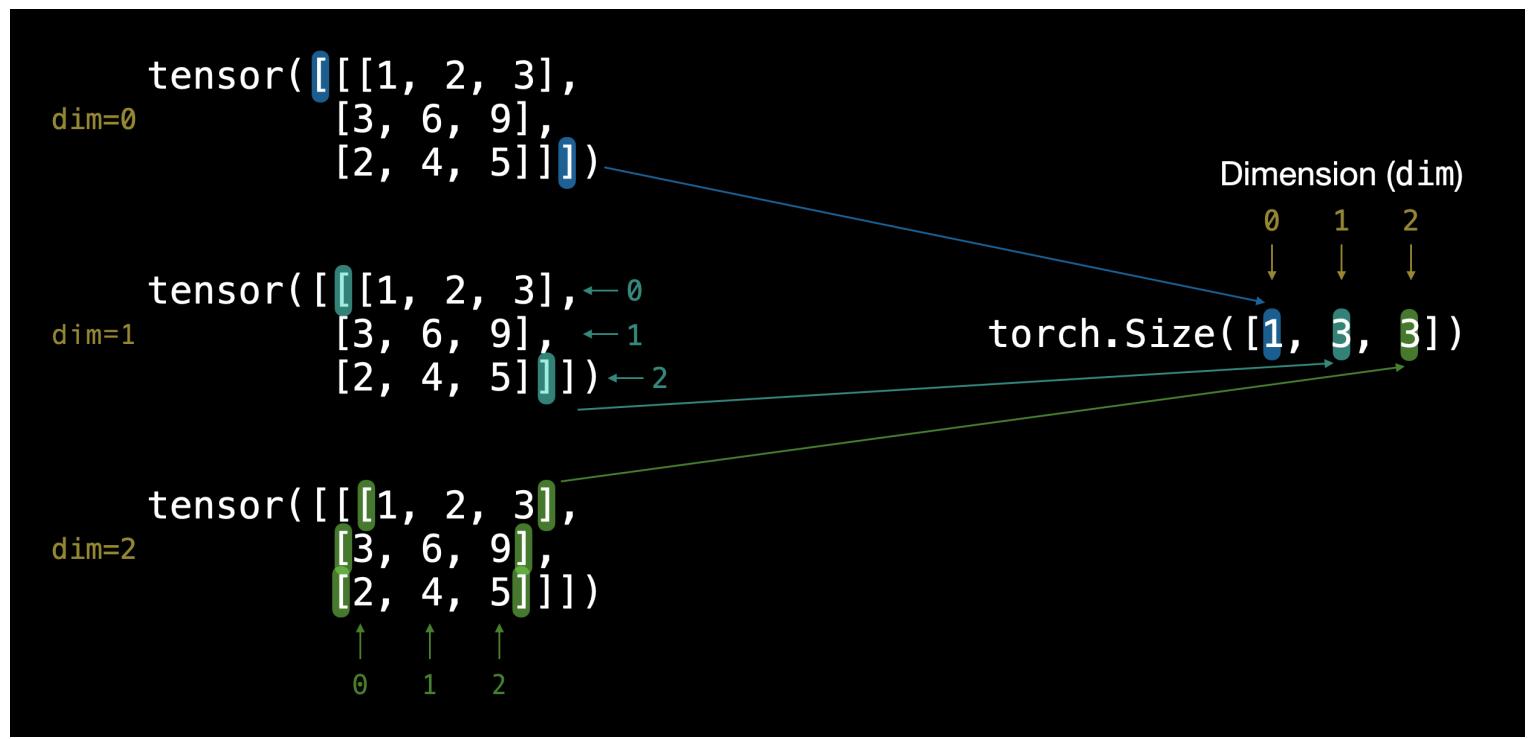
Out[13]:

```
torch.Size([1, 3, 3])
```

Alright, it outputs `torch.Size([1, 3, 3])`.

The dimensions go outer to inner.

That means there's 1 dimension of 3 by 3.



**Note:** You might've noticed me using lowercase letters for `scalar` and `vector` and uppercase letters for `MATRIX` and `TENSOR`. This was on purpose. In practice, you'll often see scalars and vectors denoted as lowercase letters such as `y` or `a`. And matrices and tensors denoted as uppercase letters such as `x` or `w`.

You also might notice the names martrix and tensor used interchangably. This is common. Since in PyTorch you're often dealing with `torch.Tensor`'s (hence the tensor name), however, the shape and dimensions of what's inside will dictate what it actually is.

Let's summarise.

Name	What is it?	Number of dimensions	Lower or upper (usually/example)
<code>scalar</code>	a single number	0	Lower ( <code>a</code> )
<code>vector</code>	a number with direction (e.g. wind speed with direction) but can also have many other numbers	1	Lower ( <code>y</code> )
<code>matrix</code>	a 2-dimensional array of numbers	2	Upper ( <code>Q</code> )
<code>tensor</code>	an n-dimensional array of numbers	can be any number, a 0-dimension tensor is a scalar, a 1-dimension tensor is a vector	Upper ( <code>x</code> )

## Scalar

7

## Vector

7
4

or

7	4
---	---

## Matrix

7	10
4	3
5	1

## Tensor

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

## Random tensors

We've established tensors represent some form of data.

And machine learning models such as neural networks manipulate and seek patterns within tensors.

But when building machine learning models with PyTorch, it's rare you'll create tensors by hand (like what we've been doing).

Instead, a machine learning model often starts out with large random tensors of numbers and adjusts these random numbers as it works through data to better represent it.

In essence:

```
Start with random numbers -> look at data -> update random numbers -> look at data -> update random numbers...
```

As a data scientist, you can define how the machine learning model starts (initialization), looks at data (representation) and updates (optimization) its random numbers.

We'll get hands on with these steps later on.

For now, let's see how to create a tensor of random numbers.

We can do so using `torch.rand()` and passing in the `size` parameter.

In [14]:

```
(tensor([[0.5650, 0.3879, 0.5377, 0.2371],
        [0.5277, 0.1666, 0.5494, 0.9758],
        [0.8932, 0.4900, 0.8934, 0.1187]]),
 torch.float32)
```

Out[14]:

The flexibility of `torch.rand()` is that we can adjust the `size` to be whatever we want.

For example, say you wanted a random tensor in the common image shape of [224, 224, 3] ([height, width, color\_channels]).

In [15]:

```
(torch.Size([224, 224, 3]), 3)
```

Out[15]:

## Zeros and ones

Sometimes you'll just want to fill tensors with zeros or ones.

This happens a lot with masking (like masking some of the values in one tensor with zeros to let a model know not to learn them).

Let's create a tensor full of zeros with `torch.zeros()`

Again, the `size` parameter comes into play.

In [16]:

```
(tensor([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]]),
 torch.float32)
```

Out[16]:

We can do the same to create a tensor of all ones except using `torch.ones()` instead.

In [17]:

Out[17]:

```
(tensor([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]]),
 torch.float32)
```

## Creating a range and tensors like

Sometimes you might want a range of numbers, such as 1 to 10 or 0 to 100.

You can use `torch.arange(start, end, step)` to do so.

Where:

- `start` = start of range (e.g. 0)
- `end` = end of range (e.g. 10)
- `step` = how many steps in between each value (e.g. 1)

**Note:** In Python, you can use `range()` to create a range. However in PyTorch, `torch.range()` is deprecated and may show an error in the future.

In [18]:



```
/tmp/ipykernel_1303467/193451495.py:2: UserWarning: torch.range is deprecated and will be removed  
in a future release because its behavior is inconsistent with Python's range builtin. Instead, u  
se torch.arange, which produces values in [start, end).  
zero_to_ten_DEPRECATED = torch.range(0, 10) # Note: this may return an error in the future  
Out[18]:  
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Sometimes you might want one tensor of a certain type with the same shape as another tensor.

For example, a tensor of all zeros with the same shape as a previous tensor.

To do so you can use `torch.zeros_like(input)` or `torch.ones_like(input)` which return a tensor filled with zeros or ones in the same shape as the `input` respectively.

In [19]:

Out[19]:

```
tensor([0, 0, 0, 0, 0, 0, 0, 0, 0])
```

## Tensor datatypes

There are many different [tensor datatypes available in PyTorch](#).

Some are specific for CPU and some are better for GPU.

Getting to know which is which can take some time.

Generally if you see `torch.cuda` anywhere, the tensor is being used for GPU (since Nvidia GPUs use a computing toolkit called CUDA).

The most common type (and generally the default) is `torch.float32` or `torch.float`.

This is referred to as "32-bit floating point".

But there's also 16-bit floating point (`torch.float16` or `torch.half`) and 64-bit floating point (`torch.float64` or `torch.double`).

And to confuse things even more there's also 8-bit, 16-bit, 32-bit and 64-bit integers.

Plus more!

**Note:** An integer is a flat round number like `7` whereas a float has a decimal `7.0`.

The reason for all of these is to do with **precision in computing**.

Precision is the amount of detail used to describe a number.

The higher the precision value (8, 16, 32), the more detail and hence data used to express a number.

This matters in deep learning and numerical computing because you're making so many operations, the more detail

you have to calculate on, the more compute you have to use.

So lower precision datatypes are generally faster to compute on but sacrifice some performance on evaluation metrics like accuracy (faster to compute but less accurate).

### Resources:

- See the [PyTorch documentation](#) for a list of all available tensor datatypes.
- Read the [Wikipedia page for an overview of what precision in computing](#)) is.

Let's see how to create some tensors with specific datatypes. We can do so using the `dtype` parameter.

In [20]:





Out[20]:

```
(torch.Size([3]), torch.float32, device(type='cpu'))
```

Aside from shape issues (tensor shapes don't match up), two of the other most common issues you'll come across in PyTorch are datatype and device issues.

For example, one of tensors is `torch.float32` and the other is `torch.float16` (PyTorch often likes tensors to be the same format).

Or one of your tensors is on the CPU and the other is on the GPU (PyTorch likes calculations between tensors to be on the same device).

We'll see more of this device talk later on.

For now let's create a tensor with `dtype=torch.float16`.

In [21]:

Out[21]:

```
torch.float16
```

## Getting information from tensors

Once you've created tensors (or someone else or a PyTorch module has created them for you), you might want to get some information from them.

We've seen these before but three of the most common attributes you'll want to find out about tensors are:

- `shape` - what shape is the tensor? (some operations require specific shape rules)
- `dtype` - what datatype are the elements within the tensor stored in?
- `device` - what device is the tensor stored on? (usually GPU or CPU)

Let's create a random tensor and find out details about it.

In [22]:



```

tensor([[0.4331, 0.8346, 0.7308, 0.9041],
       [0.4572, 0.2115, 0.6510, 0.1001],
       [0.0985, 0.9056, 0.8834, 0.8337]])
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu

```

**Note:** When you run into issues in PyTorch, it's very often one to do with one of the three attributes above. So when the error messages show up, sing yourself a little song called "what, what, where":

- "*what shape are my tensors? what datatype are they and where are they stored? what shape, what datatype, where where where*"

## Manipulating tensors (tensor operations)

In deep learning, data (images, text, video, audio, protein structures, etc) gets represented as tensors.

A model learns by investigating those tensors and performing a series of operations (could be 1,000,000s+) on tensors to create a representation of the patterns in the input data.

These operations are often a wonderful dance between:

- Addition
- Subtraction
- Multiplication (element-wise)
- Division
- Matrix multiplication

And that's it. Sure there are a few more here and there but these are the basic building blocks of neural networks.

Stacking these building blocks in the right way, you can create the most sophisticated of neural networks (just like lego!).

## Basic operations

Let's start with a few of the fundamental operations, addition (+), subtraction (-), multiplication (\*).

They work just as you think they would.

In [23]:

```
tensor([11, 12, 13])
```

Out[23]:

In [24]:

Out[24]:

```
tensor([10, 20, 30])
```

Notice how the tensor values above didn't end up being `tensor([110, 120, 130])`, this is because the values inside the tensor don't change unless they're reassigned.

In [25]:

Out[25]:

```
tensor([1, 2, 3])
```

Let's subtract a number and this time we'll reassign the `tensor` variable.

In [26]:

Out[26]:

```
tensor([-9, -8, -7])
```

In [27]:

Out[27]:

```
tensor([1, 2, 3])
```

PyTorch also has a bunch of built-in functions like `torch.mul()` (short for multiplication) and `torch.add()` to perform basic operations.

In [28]:

Out[28]:

```
tensor([10, 20, 30])
```

In [29]:

```
tensor([1, 2, 3])
```

Out[29]:

However, it's more common to use the operator symbols like `*` instead of `torch.mul()`

In [30]:

```
tensor([1, 2, 3]) * tensor([1, 2, 3])
Equals: tensor([1, 4, 9])
```

## Matrix multiplication (is all you need)

One of the most common operations in machine learning and deep learning algorithms (like neural networks) is [matrix multiplication](#).

PyTorch implements matrix multiplication functionality in the `torch.matmul()` method.

The main two rules for matrix multiplication to remember are:

### 1. The **inner dimensions** must match:

- $(3, 2) @ (3, 2)$  won't work
- $(2, 3) @ (3, 2)$  will work
- $(3, 2) @ (2, 3)$  will work

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

- $(2, 3) @ (3, 2) \rightarrow (2, 2)$
- $(3, 2) @ (2, 3) \rightarrow (3, 3)$

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

**Resource:** You can see all of the rules for matrix multiplication using `torch.matmul()` [in the PyTorch documentation](#).

Let's create a tensor and perform element-wise multiplication and matrix multiplication on it.

In [31]:

```
torch.Size([3])
```

Out[31]:

The difference between element-wise multiplication and matrix multiplication is the addition of values.

For our `tensor` variable with values `[1, 2, 3]`:

Operation	Calculation	Code
-----------	-------------	------

**Element-wise multiplication**       $[1*1, 2*2, 3*3] = [1, 4, 9]$       `tensor * tensor`

**Matrix multiplication**       $[1*1 + 2*2 + 3*3] = [14]$       `tensor.matmul(tensor)`

In [32]:

Out[32]:

```
tensor([1, 4, 9])
```

In [33]:

Out[33]:

```
tensor(14)
```

In [34]:

Out[34]:

```
tensor(14)
```

You can do matrix multiplication by hand but it's not recommended.

The in-built `torch.matmul()` method is faster.

In [35]:

```
CPU times: user 134 µs, sys: 13 µs, total: 147 µs
Wall time: 102 µs
```

```
tensor(14)
```

Out[35]:

```
CPU times: user 46 µs, sys: 0 ns, total: 46 µs
Wall time: 36.5 µs
```

```
tensor(14)
```

Out[36]:

## One of the most common errors in deep learning (shape errors)

Because much of deep learning is multiplying and performing operations on matrices and matrices have a strict rule about what shapes and sizes can be combined, one of the most common errors you'll run into in deep learning is shape mismatches.

In [37]:







```

-----  

RuntimeError                                Traceback (most recent call last)  

Input In [37], in <cell line: 10>()  

    2 tensor_A = torch.tensor([[1, 2],  

    3                 [3, 4],  

    4                 [5, 6]], dtype=torch.float32)  

    5 tensor_B = torch.tensor([[7, 10],  

    6                 [8, 11],  

    7                 [9, 12]], dtype=torch.float32)  

----> 10 torch.matmul(tensor_A, tensor_B)  

RuntimeError: mat1 and mat2 shapes cannot be multiplied (3x2 and 3x2)

```

We can make matrix multiplication work between `tensor_A` and `tensor_B` by making their inner dimensions match.

One of the ways to do this is with a **transpose** (switch the dimensions of a given tensor).

You can perform transposes in PyTorch using either:

- `torch.transpose(input, dim0, dim1)` - where `input` is the desired tensor to transpose and `dim0` and `dim1` are the dimensions to be swapped.
- `tensor.T` - where `tensor` is the desired tensor to transpose.

Let's try the latter.

In [38]:

```

tensor([[1., 2.],
       [3., 4.],
       [5., 6.]])

```

```
tensor([[ 7., 10.],  
       [ 8., 11.],  
       [ 9., 12.]])
```

In [39]:

```
tensor([[1., 2.],  
       [3., 4.],  
       [5., 6.]])  
tensor([[ 7.,  8.,  9.],  
       [10., 11., 12.]])
```

In [40]:



```
Original shapes: tensor_A = torch.Size([3, 2]), tensor_B = torch.Size([3, 2])

New shapes: tensor_A = torch.Size([3, 2]) (same as above), tensor_B.T = torch.Size([2, 3])

Multiplying: torch.Size([3, 2]) * torch.Size([2, 3]) <- inner dimensions match

Output:

tensor([[ 27.,  30.,  33.],
       [ 61.,  68.,  75.],
       [ 95., 106., 117.]])
```

Output shape: torch.Size([3, 3])

You can also use `torch.mm()` which is a short for `torch.matmul()`.

In [41]:

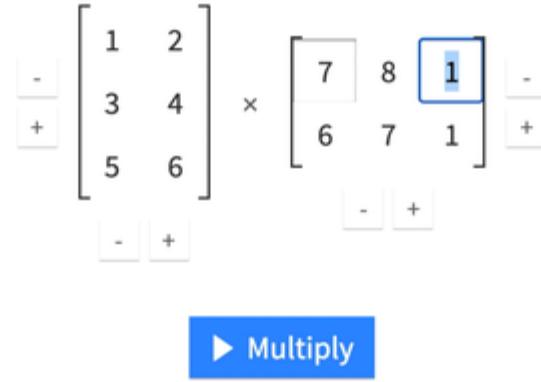
```
Out[41]:
```

```
tensor([[ 27.,  30.,  33.],
       [ 61.,  68.,  75.],
       [ 95., 106., 117.]])
```

Without the transpose, the rules of matrix multiplication aren't fulfilled and we get an error like above.

How about a visual?

## Matrix Multiplication



You can create your own matrix multiplication visuals like this at <http://matrixmultiplication.xyz/>.

**Note:** A matrix multiplication like this is also referred to as the **dot product** of two matrices.

Neural networks are full of matrix multiplications and dot products.

The `torch.nn.Linear()` module (we'll see this in action later on), also known as a feed-forward layer or fully connected layer, implements a matrix multiplication between an input `x` and a weights matrix `A`.

$$y = x \cdot A^T + b$$

Where:

- `x` is the input to the layer (deep learning is a stack of layers like `torch.nn.Linear()` and others on top of each other).
- `A` is the weights matrix created by the layer, this starts out as random numbers that get adjusted as a neural network learns to better represent patterns in the data (notice the "`T`", that's because the weights matrix gets transposed).
  - **Note:** You might also often see `w` or another letter like `x` used to showcase the weights matrix.
- `b` is the bias term used to slightly offset the weights and inputs.
- `y` is the output (a manipulation of the input in the hopes to discover patterns in it).

This is a linear function (you may have seen something like  $y = mx + b$  in high school or elsewhere), and can be used to draw a straight line!

Let's play around with a linear layer.

Try changing the values of `in_features` and `out_features` below and see what happens.

Do you notice anything to do with the shapes?

In [42]:



```
Input shape: torch.Size([3, 2])
```

Output:

```
tensor([[2.2368, 1.2292, 0.4714, 0.3864, 0.1309, 0.9838],  
       [4.4919, 2.1970, 0.4469, 0.5285, 0.3401, 2.4777],  
       [6.7469, 3.1648, 0.4224, 0.6705, 0.5493, 3.9716]],  
      grad_fn=<AddmmBackward0>)
```

```
Output shape: torch.Size([3, 6])
```

**Question:** What happens if you change `in_features` from 2 to 3 above? Does it error? How could you change the shape of the input (`x`) to accomodate to the error? Hint: what did we have to do to `tensor_B` above?

If you've never done it before, matrix multiplication can be a confusing topic at first.

But after you've played around with it a few times and even cracked open a few neural networks, you'll notice it's everywhere.

Remember, matrix multiplication is all you need.



*When you start digging into neural network layers and building your own, you'll find matrix multiplications everywhere. Source: <https://marksaroufim.substack.com/p/working-class-deep-learner>*

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

Now we've seen a few ways to manipulate tensors, let's run through a few ways to aggregate them (go from more values to less values).

First we'll create a tensor and then find the max, min, mean and sum of it.

In [43]:

Out[43]:

```
tensor([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

Now let's perform some aggregation.

In [44]:

```
Minimum: 0
Maximum: 90
Mean: 45.0
Sum: 450
```

**Note:** You may find some methods such as `torch.mean()` require tensors to be in `torch.float32` (the most common) or another specific datatype, otherwise the operation will fail.

You can also do the same as above with `torch` methods.

In [45]:

```
(tensor(90), tensor(0), tensor(45.), tensor(450))
```

Out[45]:

## Positional min/max

You can also find the index of a tensor where the max or minimum occurs with `torch.argmax()` and `torch.argmin()` respectively.

This is helpful incase you just want the position where the highest (or lowest) value is and not the actual value itself (we'll see this in a later section when using the [softmax activation function](#)).

In [46]:



```
Tensor: tensor([10, 20, 30, 40, 50, 60, 70, 80, 90])
Index where max value occurs: 8
Index where min value occurs: 0
```

## Change tensor datatype

As mentioned, a common issue with deep learning operations is having your tensors in different datatypes.

If one tensor is in `torch.float64` and another is in `torch.float32`, you might run into some errors.

But there's a fix.

You can change the datatypes of tensors using `torch.Tensor.type(dtype=None)` where the `dtype` parameter is the datatype you'd like to use.

First we'll create a tensor and check it's datatype (the default is `torch.float32`).

In [47]:

Out[47]:

```
torch.float32
```

Now we'll create another tensor the same as before but change its datatype to `torch.float16`.

In [48]:

Out[48]:

```
tensor([10., 20., 30., 40., 50., 60., 70., 80., 90.], dtype=torch.float16)
```

And we can do something similar to make a `torch.int8` tensor.

In [49]:

Out[49]:

```
tensor([10, 20, 30, 40, 50, 60, 70, 80, 90], dtype=torch.int8)
```

**Note:** Different datatypes can be confusing to begin with. But think of it like this, the lower the number (e.g. 32, 16, 8), the less precise a computer stores the value. And with a lower amount of storage, this generally results in faster computation and a smaller overall model. Mobile-based neural networks often operate with 8-bit integers, smaller and faster to run but less accurate than their float32 counterparts. For more on this, I'd read up about [precision in computing](#)).

**Exercise:** So far we've covered a fair few tensor methods but there's a bunch more in the [torch.Tensor documentation](#), I'd recommend spending 10-minutes scrolling through and looking into any that catch your eye. Click on them and then write them out in code yourself to see what happens.

## Reshaping, stacking, squeezing and unsqueezing

Often times you'll want to reshape or change the dimensions of your tensors without actually changing the values inside them.

To do so, some popular methods are:

Method	One-line description
<code>torch.reshape(input, shape)</code>	Reshapes <code>input</code> to <code>shape</code> (if compatible), can also use <code>torch.Tensor.reshape()</code> .
<code>torch.Tensor.view(shape)</code>	Returns a view of the original tensor in a different <code>shape</code> but shares the same data as the original tensor.
<code>torch.stack(tensors, dim=0)</code>	Concatenates a sequence of <code>tensors</code> along a new dimension ( <code>dim</code> ), all <code>tensors</code> must be same size.
<code>torch.squeeze(input)</code>	Squeezes <code>input</code> to remove all the dimensions with value <code>1</code> .
<code>torch.unsqueeze(input, dim)</code>	Returns <code>input</code> with a dimension value of <code>1</code> added at <code>dim</code> .
<code>torch.permute(input, dims)</code>	Returns a <i>view</i> of the original <code>input</code> with its dimensions permuted (rearranged) to <code>dims</code> .

Why do any of these?

Because deep learning models (neural networks) are all about manipulating tensors in some way. And because of the rules of matrix multiplication, if you've got shape mismatches, you'll run into errors. These methods help you make the right elements of your tensors are mixing with the right elements of other tensors.

Let's try them out.

First, we'll create a tensor.

In [50]:

```
(tensor([1., 2., 3., 4., 5., 6., 7.]), torch.Size([7]))
```

Now let's add an extra dimension with `torch.reshape()`.

Out[50]:

In [51]:

Out[51]:

```
(tensor([[1., 2., 3., 4., 5., 6., 7.]]), torch.Size([1, 7]))
```

We can also change the view with `torch.view()`.

In [52]:

Out[52]:

```
(tensor([[1., 2., 3., 4., 5., 6., 7.]]), torch.Size([1, 7]))
```

Remember though, changing the view of a tensor with `torch.view()` really only creates a new view of the *same* tensor.

So changing the view changes the original tensor too.

In [53]:

```
(tensor([[5., 2., 3., 4., 5., 6., 7.]]), tensor([5., 2., 3., 4., 5., 6., 7.]))
```

If we wanted to stack our new tensor on top of itself five times, we could do so with `torch.stack()`.

In [54]:

Out[54]:

```
tensor([[5., 2., 3., 4., 5., 6., 7.],  
       [5., 2., 3., 4., 5., 6., 7.],  
       [5., 2., 3., 4., 5., 6., 7.],  
       [5., 2., 3., 4., 5., 6., 7.]])
```

How about removing all single dimensions from a tensor?

To do so you can use `torch.squeeze()` (I remember this as *squeezing* the tensor to only have dimensions over 1).

In [55]:

```
Previous tensor: tensor([5., 2., 3., 4., 5., 6., 7.])
Previous shape: torch.Size([1, 7])
```

```
New tensor: tensor([5., 2., 3., 4., 5., 6., 7.])
New shape: torch.Size([7])
```

And to do the reverse of `torch.squeeze()` you can use `torch.unsqueeze()` to add a dimension value of 1 at a specific index.

In [56]:

```
Previous tensor: tensor([5., 2., 3., 4., 5., 6., 7.])
Previous shape: torch.Size([7])

New tensor: tensor([[5., 2., 3., 4., 5., 6., 7.]])
New shape: torch.Size([1, 7])
```

You can also rearrange the order of axes values with `torch.permute(input, dims)`, where the `input` gets turned into a `view` with new `dims`.

In [57]:

```
Previous shape: torch.Size([224, 224, 3])
New shape: torch.Size([3, 224, 224])
```

**Note:** Because permuting returns a *view* (shares the same data as the original), the values in the permuted tensor will be the same as the original tensor and if you change the values in the view, it will change the values of the original.

## Indexing (selecting data from tensors)

Sometimes you'll want to select specific data from tensors (for example, only the first column or second row).

To do so, you can use indexing.

If you've ever done indexing on Python lists or NumPy arrays, indexing in PyTorch with tensors is very similar.

In [58]:

Out[58]:

```
(tensor([[[1, 2, 3],  
        [4, 5, 6],  
        [7, 8, 9]]]),  
 torch.Size([1, 3, 3]))
```

Indexing values goes outer dimension -> inner dimension (check out the square brackets).

In [59]:

```
First square bracket:  
tensor([[1, 2, 3],  
       [4, 5, 6],  
       [7, 8, 9]])  
Second square bracket: tensor([1, 2, 3])  
Third square bracket: 1
```

You can also use : to specify "all values in this dimension" and then use a comma ( , ) to add another dimension.

In [60]:

Out[60]:

```
tensor([[1, 2, 3]])
```

In [61]:

Out[61]:

```
tensor([[2, 5, 8]])
```

In [62]:

```
tensor([5])
```

Out[62]:

In [63]:

Out[63]:

```
tensor([1, 2, 3])
```

Indexing can be quite confusing to begin with, especially with larger tensors (I still have to try indexing multiple times to get it right). But with a bit of practice and following the data explorer's motto (***visualize, visualize, visualize***),

you'll start to get the hang of it.

## PyTorch tensors & NumPy

Since NumPy is a popular Python numerical computing library, PyTorch has functionality to interact with it nicely.

The two main methods you'll want to use for NumPy to PyTorch (and back again) are:

- `torch.from_numpy(ndarray)` - NumPy array -> PyTorch tensor.
- `torch.Tensor.numpy()` - PyTorch tensor -> NumPy array.

Let's try them out.

In [64]:

Out[64]:

```
(array([1., 2., 3., 4., 5., 6., 7.]),  
 tensor([1., 2., 3., 4., 5., 6., 7.], dtype=torch.float64))
```

**Note:** By default, NumPy arrays are created with the datatype `float64` and if you convert it to a PyTorch tensor, it'll keep the same datatype (as above).

However, many PyTorch calculations default to using `float32`.

So if you want to convert your NumPy array (`float64`) -> PyTorch tensor (`float64`) -> PyTorch tensor (`float32`), you can use `tensor = torch.from_numpy(array).type(torch.float32)`.

Because we reassigned `tensor` above, if you change the tensor, the array stays the same.

In [65]:

Out[65]:

```
(array([2., 3., 4., 5., 6., 7., 8.]),  
 tensor([1., 2., 3., 4., 5., 6., 7.], dtype=torch.float64))
```

And if you want to go from PyTorch tensor to NumPy array, you can call `tensor.numpy()`.

In [66]:

Out[66]:

```
(tensor([1., 1., 1., 1., 1., 1.]),  
 array([1., 1., 1., 1., 1., 1.], dtype=float32))
```

And the same rule applies as above, if you change the original `tensor`, the new `numpy_tensor` stays the same.

In [67]:

Out[67]:

```
(tensor([2., 2., 2., 2., 2., 2.]),  
 array([1., 1., 1., 1., 1., 1.], dtype=float32))
```

## Reproducibility (trying to take the random out of random)

As you learn more about neural networks and machine learning, you'll start to discover how much randomness plays a part.

Well, pseudorandomness that is. Because after all, as they're designed, a computer is fundamentally deterministic (each step is predictable) so the randomness they create are simulated randomness (though there is debate on this too, but since I'm not a computer scientist, I'll let you find out more yourself).

How does this relate to neural networks and deep learning then?

We've discussed neural networks start with random numbers to describe patterns in data (these numbers are poor descriptions) and try to improve those random numbers using tensor operations (and a few other things we haven't discussed yet) to better describe patterns in data.

In short:

```
start with random numbers -> tensor operations -> try to make better (again and again and again)
```

Although randomness is nice and powerful, sometimes you'd like there to be a little less randomness.

Why?

So you can perform repeatable experiments.

For example, you create an algorithm capable of achieving X performance.

And then your friend tries it out to verify you're not crazy.

How could they do such a thing?

That's where **reproducibility** comes in.

In other words, can you get the same (or very similar) results on your computer running the same code as I get on mine?

Let's see a brief example of reproducibility in PyTorch.

We'll start by creating two random tensors, since they're random, you'd expect them to be different right?

In [68]:



```
Tensor A:  
tensor([[0.8016, 0.3649, 0.6286, 0.9663],  
       [0.7687, 0.4566, 0.5745, 0.9200],  
       [0.3230, 0.8613, 0.0919, 0.3102]])
```

```
Tensor B:  
tensor([[0.9536, 0.6002, 0.0351, 0.6826],  
       [0.3743, 0.5220, 0.1336, 0.9666],  
       [0.9754, 0.8474, 0.8988, 0.1105]])
```

Does Tensor A equal Tensor B? (anywhere)

```
tensor([[False, False, False, False],  
       [False, False, False, False],  
       [False, False, False, False]])
```

Just as you might've expected, the tensors come out with different values.

But what if you wanted to created two random tensors with the *same* values.

As in, the tensors would still contain random values but they would be of the same flavour.

That's where `torch.manual_seed(seed)` comes in, where `seed` is an integer (like `42` but it could be anything) that flavours the randomness.

Let's try it out by creating some more *flavoured* random tensors.

Out[68]:

In [69]:





```
Tensor C:  
tensor([[0.8823, 0.9150, 0.3829, 0.9593],  
       [0.3904, 0.6009, 0.2566, 0.7936],  
       [0.9408, 0.1332, 0.9346, 0.5936]])
```

```
Tensor D:  
tensor([[0.8823, 0.9150, 0.3829, 0.9593],  
       [0.3904, 0.6009, 0.2566, 0.7936],  
       [0.9408, 0.1332, 0.9346, 0.5936]])
```

```
Does Tensor C equal Tensor D? (anywhere)
```

Out[69]:

```
tensor([[True, True, True, True],  
       [True, True, True, True],  
       [True, True, True, True]])
```

Nice!

It looks like setting the seed worked.

**Resource:** What we've just covered only scratches the surface of reproducibility in PyTorch. For more, on reproducibility in general and random seeds, I'd checkout:

- [The PyTorch reproducibility documentation](#) (a good exercise would be to read through this for 10-minutes and even if you don't understand it now, being aware of it is important).
- [The Wikipedia random seed page](#) (this'll give a good overview of random seeds and pseudorandomness in general).

## Running tensors on GPUs (and making faster computations)

Deep learning algorithms require a lot of numerical operations.

And by default these operations are often done on a CPU (computer processing unit).

However, there's another common piece of hardware called a GPU (graphics processing unit), which is often much faster at performing the specific types of operations neural networks need (matrix multiplications) than CPUs.

Your computer might have one.

If so, you should look to use it whenever you can to train neural networks because chances are it'll speed up the training time dramatically.

There are a few ways to first get access to a GPU and secondly get PyTorch to use the GPU.

**Note:** When I reference "GPU" throughout this course, I'm referencing a [Nvidia GPU with CUDA enabled](#) (CUDA is a computing platform and API that helps allow GPUs be used for general purpose computing & not just graphics) unless otherwise specified.

### 1. Getting a GPU

You may already know what's going on when I say GPU. But if not, there are a few ways to get access to one.

Method	Difficulty to setup	Pros	Cons	How to setup
Google Colab	Easy	Free to use, almost zero setup required, can share work with others as easy as a link	Doesn't save your data outputs, limited compute, subject to timeouts	Follow the <a href="#">Google Colab Guide</a>
Use your own	Medium	Run everything locally on your own machine	GPUs aren't free, require upfront cost	Follow the <a href="#">PyTorch installation guidelines</a>
Cloud computing (AWS, GCP, Azure)	Medium-Hard	Small upfront cost, access to almost infinite compute	Can get expensive if running continually, takes some time to setup right	Follow the <a href="#">PyTorch installation guidelines</a>

There are more options for using GPUs but the above three will suffice for now.

Personally, I use a combination of Google Colab and my own personal computer for small scale experiments (and creating this course) and go to cloud resources when I need more compute power.

**Resource:** If you're looking to purchase a GPU of your own but not sure what to get, [Tim Dettmers has an excellent guide](#).

To check if you've got access to a Nvidia GPU, you can run `!nvidia-smi` where the `!` (also called bang) means "run this on the command line".

In [70]:

```
Fri Aug 12 14:50:11 2022
+-----+
| NVIDIA-SMI 515.48.07      Driver Version: 515.48.07      CUDA Version: 11.7      |
| -----+-----+-----+
| 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  NVIDIA TITAN RTX     On  | 00000000:01:00.0 Off |           N/A | | |
| 40%   24C     P8      7W / 280W |    1939MiB / 24576MiB |      0%     Default |
|          |          |             |              |                  N/A |
+-----+-----+-----+
+-----+
| Processes:
```

GPU	GI	CI	PID	Type	Process name	GPU Memory
	ID	ID				Usage
<hr/>						
0	N/A	N/A	1068	G	/usr/lib/xorg/Xorg	86MiB
0	N/A	N/A	1220	G	/usr/bin/gnome-shell	9MiB
0	N/A	N/A	1186254	C	...de/pytorch/env/bin/python	1839MiB

If you don't have a Nvidia GPU accessible, the above will output something like:

```
NVIDIA-SMI has failed because it couldn't communicate with the NVIDIA driver. Make sure that the latest NVIDIA
driver is installed and running.
```

In that case, go back up and follow the install steps.

If you do have a GPU, the line above will output something like:

Wed Jan 19 22:09:08 2022						
<hr/>						
NVIDIA-SMI 495.46      Driver Version: 460.32.03      CUDA Version: 11.2						
<hr/>						
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.						
<hr/>						
0 Tesla P100-PCIE... Off   00000000:00:04.0 Off        0						
N/A 35C P0 27W / 250W   0MiB / 16280MiB   0% Default						
N/A						
<hr/>						
<hr/>						
Processes:						
GPU GI CI      PID Type Process name      GPU Memory						
ID ID                               Usage						
<hr/>						
No running processes found						
<hr/>						

## 2. Getting PyTorch to run on the GPU

Once you've got a GPU ready to access, the next step is getting PyTorch to use for storing data (tensors) and computing on data (performing operations on tensors).

To do so, you can use the `torch.cuda` package.

Rather than talk about it, let's try it out.

You can test if PyTorch has access to a GPU using `torch.cuda.is_available()`.

In [71]:

Out[71]:

True

If the above outputs `True`, PyTorch can see and use the GPU, if it outputs `False`, it can't see the GPU and in that case, you'll have to go back through the installation steps.

Now, let's say you wanted to setup your code so it ran on CPU *or* the GPU if it was available.

That way, if you or someone decides to run your code, it'll work regardless of the computing device they're using.

Let's create a `device` variable to store what kind of device is available.

In [72]:

Out[72]:

'cuda'

If the above output "cuda" it means we can set all of our PyTorch code to use the available CUDA device (a GPU) and if it output "cpu", our PyTorch code will stick with the CPU.

**Note:** In PyTorch, it's best practice to write **device agnostic code**. This means code that'll run on CPU (always available) or GPU (if available).

If you want to do faster computing you can use a GPU but if you want to do *much* faster computing, you can use multiple GPUs.

You can count the number of GPUs PyTorch has access to using `torch.cuda.device_count()`.

In [73]:

Out[73]:

1

Knowing the number of GPUs PyTorch has access to is helpful incase you wanted to run a specific process on one GPU and another process on another (PyTorch also has features to let you run a process across *all* GPUs).

### 3. Putting tensors (and models) on the GPU

You can put tensors (and models, we'll see this later) on a specific device by calling `to(device)` on them. Where `device` is the target device you'd like the tensor (or model) to go to.

Why do this?

GPUs offer far faster numerical computing than CPUs do and if a GPU isn't available, because of our **device agnostic code** (see above), it'll run on the CPU.

**Note:** Putting a tensor on GPU using `to(device)` (e.g. `some_tensor.to(device)`) returns a copy of that tensor, e.g. the same tensor will be on CPU and GPU. To overwrite tensors, reassign them:

```
some_tensor = some_tensor.to(device)
```

Let's try creating a tensor and putting it on the GPU (if it's available).

In [74]:

```
tensor([1, 2, 3]) cpu  
tensor([1, 2, 3], device='cuda:0')
```

Out[74]:

If you have a GPU available, the above code will output something like:

```
tensor([1, 2, 3]) cpu  
tensor([1, 2, 3], device='cuda:0')
```

Notice the second tensor has `device='cuda:0'`, this means it's stored on the 0th GPU available (GPUs are 0 indexed, if two GPUs were available, they'd be '`'cuda:0'`' and '`'cuda:1'`' respectively, up to '`'cuda:n'`' ).

## 4. Moving tensors back to the CPU

What if we wanted to move the tensor back to CPU?

For example, you'll want to do this if you want to interact with your tensors with NumPy (NumPy does not leverage the GPU).

Let's try using the `torch.Tensor.numpy()` method on our `tensor_on_gpu`.

In [75]:

```
-----  
TypeError                                 Traceback (most recent call last)  
Input In [75], in <cell line: 2>()  
      1 # If tensor is on GPU, can't transform it to NumPy (this will error)  
----> 2 tensor_on_gpu.numpy()  
  
TypeError: can't convert cuda:0 device type tensor to numpy. Use Tensor.cpu() to copy the tensor  
to host memory first.
```

Instead, to get a tensor back to CPU and usable with NumPy we can use `Tensor.cpu()`.

This copies the tensor to CPU memory so it's usable with CPUs.

In [76]:

Out[76]:

```
array([1, 2, 3])
```

The above returns a copy of the GPU tensor in CPU memory so the original tensor is still on GPU.

In [77]:

Out[77]:

```
tensor([1, 2, 3], device='cuda:0')
```

## Exercises

All of the exercises are focused on practicing the code above.

You should be able to complete them by referencing each section or by following the resource(s) linked.

### Resources:

- [Exercise template notebook for 00](#).
  - [Example solutions notebook for 00](#) (try the exercises *before* looking at this).
1. Documentation reading - A big part of deep learning (and learning to code in general) is getting familiar with the documentation of a certain framework you're using. We'll be using the PyTorch documentation a lot throughout the rest of this course. So I'd recommend spending 10-minutes reading the following (it's okay if you don't get some things for now, the focus is not yet full understanding, it's awareness). See the documentation on `torch.Tensor` and for `torch.cuda`.
  2. Create a random tensor with shape `(7, 7)`.
  3. Perform a matrix multiplication on the tensor from 2 with another random tensor with shape `(1, 7)` (hint: you may have to transpose the second tensor).
  4. Set the random seed to `0` and do exercises 2 & 3 over again.
  5. Speaking of random seeds, we saw how to set it with `torch.manual_seed()` but is there a GPU equivalent? (hint: you'll need to look into the documentation for `torch.cuda` for this one). If there is, set the GPU random seed to `1234`.
  6. Create two random tensors of shape `(2, 3)` and send them both to the GPU (you'll need access to a GPU for this). Set `torch.manual_seed(1234)` when creating the tensors (this doesn't have to be the GPU random seed).

7. Perform a matrix multiplication on the tensors you created in 6 (again, you may have to adjust the shapes of one of the tensors).
8. Find the maximum and minimum values of the output of 7.
9. Find the maximum and minimum index values of the output of 7.
10. Make a random tensor with shape `(1, 1, 1, 10)` and then create a new tensor with all the `1` dimensions removed to be left with a tensor of shape `(10)`. Set the seed to `7` when you create it and print out the first tensor and its shape as well as the second tensor and its shape.

## Extra-curriculum

- Spend 1-hour going through the [PyTorch basics tutorial](#) (I'd recommend the [Quickstart](#) and [Tensors](#) sections).
- To learn more on how a tensor can represent data, see this video: [What's a tensor?](#)

Previous

[Home](#)

Next

[01. PyTorch Workflow Fundamentals](#)

Made with Material for MkDocs Insiders



[View Source Code](#) | [View Slides](#) | [Watch Video Walkthrough](#)

# 01. PyTorch Workflow Fundamentals

The essence of machine learning and deep learning is to take some data from the past, build an algorithm (like a neural network) to discover patterns in it and use the discovered patterns to predict the future.

There are many ways to do this and many new ways are being discovered all the time.

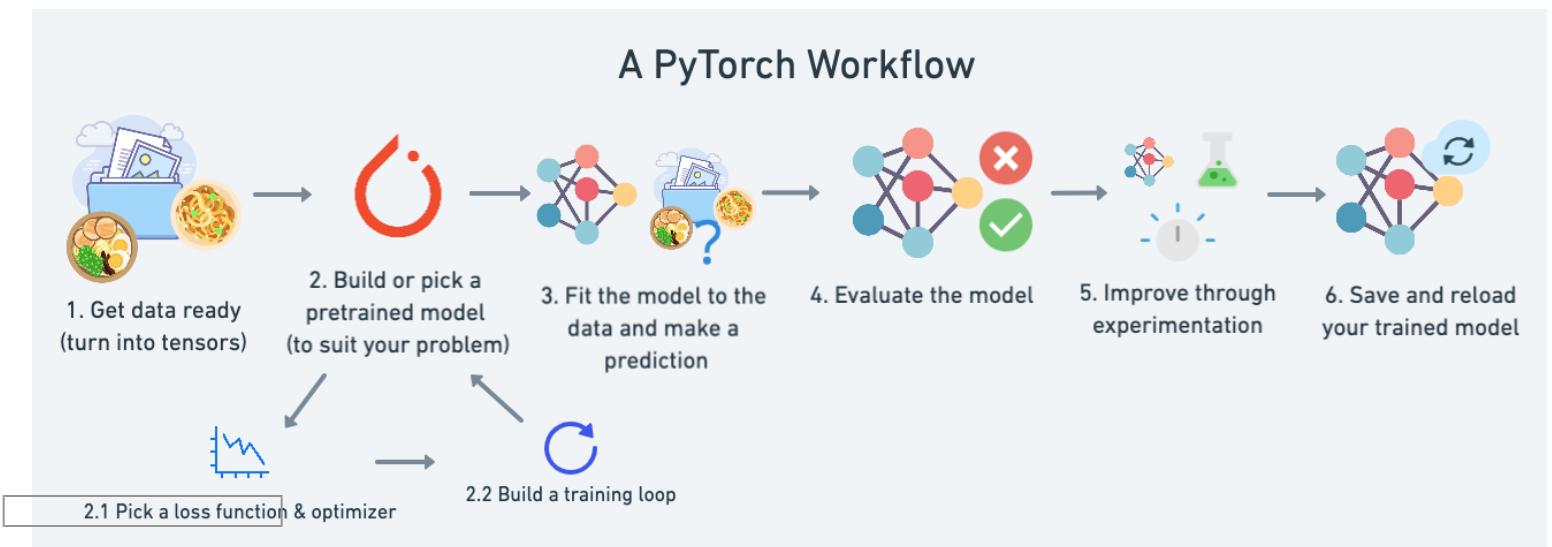
But let's start small.

How about we start with a straight line?

And we see if we can build a PyTorch model that learns the pattern of the straight line and matches it.

## What we're going to cover

In this module we're going to cover a standard PyTorch workflow (it can be chopped and changed as necessary but it covers the main outline of steps).



For now, we'll use this workflow to predict a simple straight line but the workflow steps can be repeated and changed depending on the problem you're working on.

Specifically, we're going to cover:

Topic	Contents
<b>1. Getting data ready</b>	Data can be almost anything but to get started we're going to create a simple straight line
<b>2. Building a model</b>	Here we'll create a model to learn patterns in the data, we'll also choose a <b>loss function</b> , <b>optimizer</b> and build a <b>training loop</b> .
<b>3. Fitting the model to data (training)</b>	We've got data and a model, now let's let the model (try to) find patterns in the ( <b>training</b> ) data.
<b>4. Making predictions and evaluating a model (inference)</b>	Our model's found patterns in the data, let's compare its findings to the actual ( <b>testing</b> ) data.
<b>5. Saving and loading a model</b>	You may want to use your model elsewhere, or come back to it later, here we'll cover that.
<b>6. Putting it all together</b>	Let's take all of the above and combine it.

## Where can you get help?

All of the materials for this course are [available on GitHub](#).

And if you run into trouble, you can ask a question on the [Discussions page](#) there too.

There's also the [PyTorch developer forums](#), a very helpful place for all things PyTorch.

Let's start by putting what we're covering into a dictionary to reference later.

In [1]:

And now let's import what we'll need for this module.

We're going to get `torch`, `torch.nn` (`nn` stands for neural network and this package contains the building blocks for creating neural networks in PyTorch) and `matplotlib`.

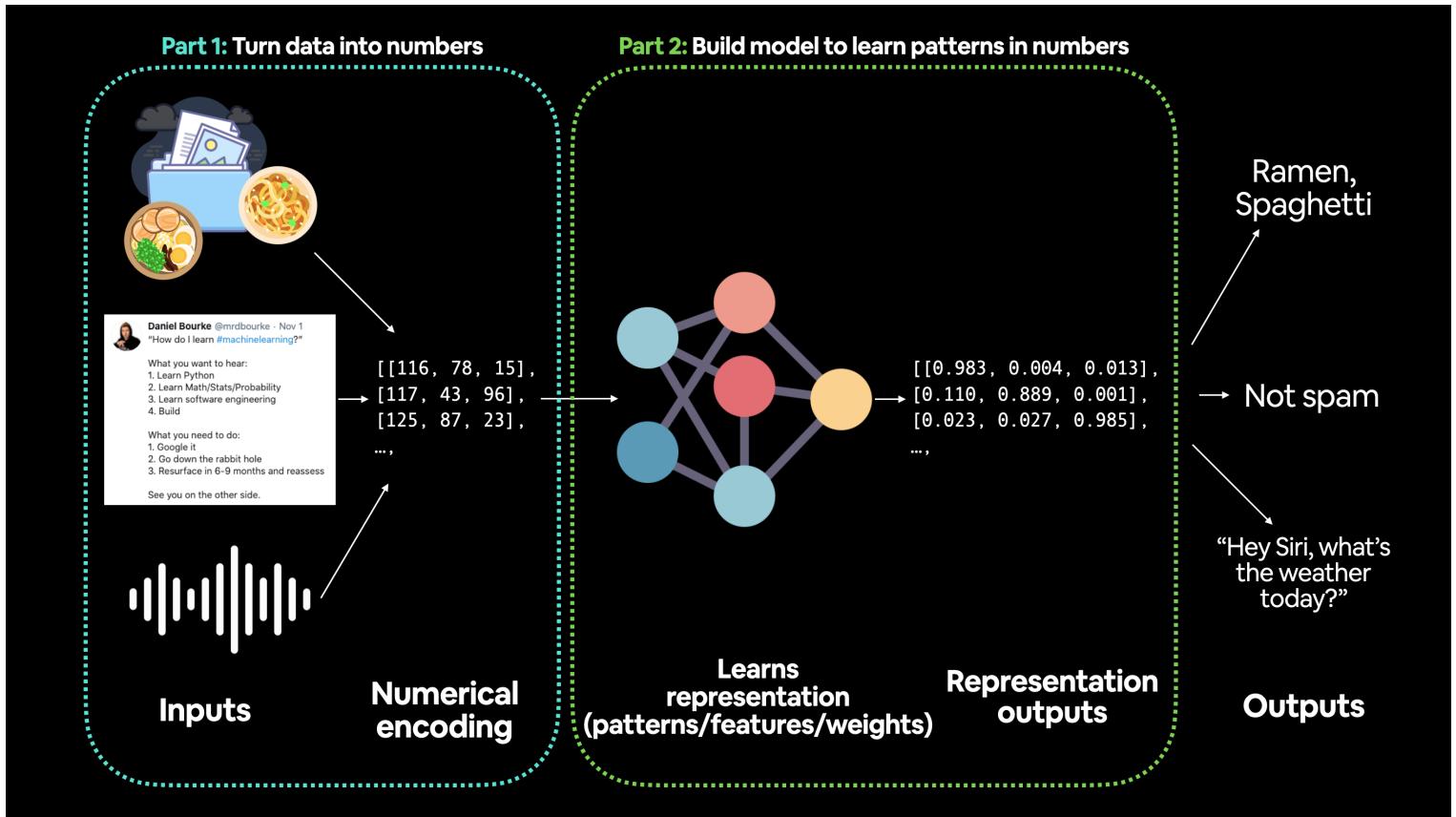
In [2]:

Out[2]:

'1.11.0'

## 1. Data (preparing and loading)

I want to stress that "data" in machine learning can be almost anything you can imagine. A table of numbers (like a big Excel spreadsheet), images of any kind, videos (YouTube has lots of data!), audio files like songs or podcasts, protein structures, text and more.



Machine learning is a game of two parts:

1. Turn your data, whatever it is, into numbers (a representation).
2. Pick or build a model to learn the representation as best as possible.

Sometimes one and two can be done at the same time.

But what if you don't have data?

Well, that's where we're at now.

No data.

But we can create some.

Let's create our data as a straight line.

We'll use [linear regression](#) to create the data with known **parameters** (things that can be learned by a model) and then we'll use PyTorch to see if we can build model to estimate these parameters using [gradient descent](#).

Don't worry if the terms above don't mean much now, we'll see them in action and I'll put extra resources below where you can learn more.

In [3]:

Out[3]:

```
(tensor([[0.0000,
         0.0200,
         0.0400,
         0.0600,
         0.0800,
         0.1000,
         0.1200,
         0.1400,
         0.1600,
         0.1800]]),
 tensor([[0.3000,
         0.3140,
         0.3280,
         0.3420,
         0.3560,
         0.3700,
         0.3840,
```

```
[0.3980],  
[0.4120],  
[0.4260]))
```

Beautiful! Now we're going to move towards building a model that can learn the relationship between `x` (**features**) and `y` (**labels**).

## Split data into training and test sets

We've got some data.

But before we build a model we need to split it up.

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

Each split of the dataset serves a specific purpose:

Split	Purpose	Amount of total data	How often is it used?
<b>Training set</b>	The model learns from this data (like the course materials you study during the semester).	~60-80%	Always
<b>Validation set</b>	The model gets tuned on this data (like the practice exam you take before the final exam).	~10-20%	Often but not always
<b>Testing set</b>	The model gets evaluated on this data to test what it has learned (like the final exam you take at the end of the semester).	~10-20%	Always

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` tensors.

**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 [4]:



Out[4]:

```
(40, 40, 10, 10)
```

Wonderful, we've got 40 samples for training (`x_train` & `y_train`) and 10 samples for testing (`x_test` & `y_test`).

The model we create is going to try and learn the relationship between `x_train` & `y_train` and then we will evaluate what it learns on `x_test` and `y_test`.

But right now our data is just numbers on a page.

Let's create a function to visualize it.

In [5]:

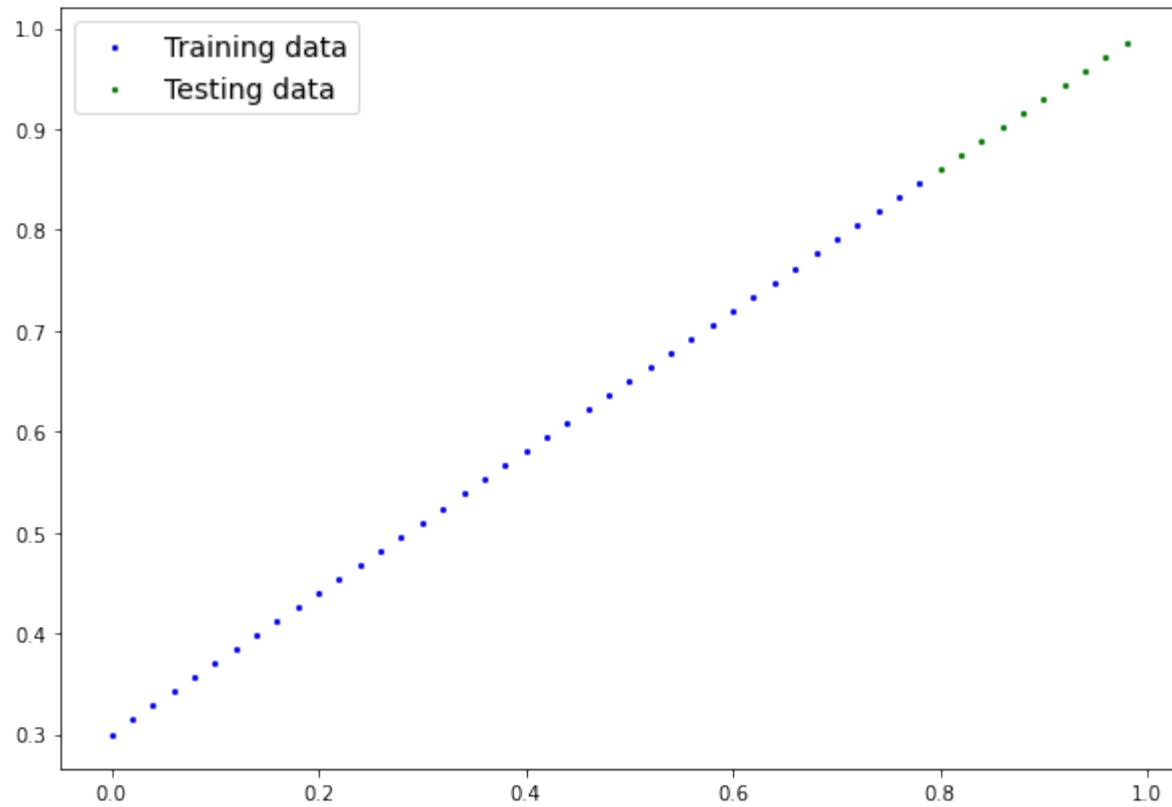








In [6]:



Epic!

Now instead of just being numbers on a page, our data is a straight line.

**Note:** Now's a good time to introduce you to the data explorer's motto... "visualize, visualize, visualize!"

Think of this whenever you're working with data and turning it into numbers, if you can visualize something, it can do wonders for understanding.

Machines love numbers and we humans like numbers too but we also like to look at things.

## 2. Build model

Now we've got some data, let's build a model to use the blue dots to predict the green dots.

We're going to jump right in.

We'll write the code first and then explain everything.

Let's replicate a standard linear regression model using pure PyTorch.

In [7]:















Alright there's a fair bit going on above but let's break it down bit by bit.

**Resource:** We'll be using Python classes to create bits and pieces for building neural networks. If you're unfamiliar with Python class notation, I'd recommend reading [Real Python's Object Orientating programming in Python 3 guide](#) a few times.

## PyTorch model building essentials

PyTorch has four (give or take) essential modules you can use to create almost any kind of neural network you can imagine.

They are `torch.nn`, `torch.optim`, `torch.utils.data.Dataset` and `torch.utils.data.DataLoader`. For now, we'll focus on the first two and get to the other two later (though you may be able to guess what they do).

PyTorch module	What does it do?
<code>torch.nn</code>	Contains all of the building blocks for computational graphs (essentially a series of computations executed in a particular way).
<code>torch.nn.Parameter</code>	Stores tensors that can be used with <code>nn.Module</code> . If <code>requires_grad=True</code> gradients (used for updating model parameters via <b>gradient descent</b> ) are calculated automatically, this is often referred to as "autograd".
<code>torch.nn.Module</code>	The base class for all neural network modules, all the building blocks for neural networks are subclasses. If you're building a neural network in PyTorch, your models should subclass <code>nn.Module</code> . Requires a <code>forward()</code> method be implemented.
<code>torch.optim</code>	Contains various optimization algorithms (these tell the model parameters stored in <code>nn.Parameter</code> how to best change to improve gradient descent and in turn reduce the loss).
<code>def forward()</code>	All <code>nn.Module</code> subclasses require a <code>forward()</code> method, this defines the computation that will take place on the data passed to the particular <code>nn.Module</code> (e.g. the linear regression formula above).

If the above sounds complex, think of like this, almost everything in a PyTorch neural network comes from `torch.nn`,

- `nn.Module` contains the larger building blocks (layers)
- `nn.Parameter` contains the smaller parameters like weights and biases (put these together to make `nn.Module(s)`)

- `forward()` tells the larger blocks how to make calculations on inputs (tensors full of data) within `nn.Module` (s)
- `torch.optim` contains optimization methods on how to improve the parameters within `nn.Parameter` to better represent input data

```

1 # Create a linear regression model in PyTorch
2 class LinearRegressionModel(nn.Module):
3     def __init__(self):
4         super().__init__()
5
6     # Initialize model parameters
7     self.weights = nn.Parameter(torch.randn(1,
8         requires_grad=True,
9         dtype=torch.float
10    ))
11
12    self.bias = nn.Parameter(torch.randn(1,
13        requires_grad=True,
14        dtype=torch.float
15    ))
16
17    # forward() defines the computation in the model
18    def forward(self, x: torch.Tensor) -> torch.Tensor:
19        return self.weights * x + self.bias
20

```

**Subclass `nn.Module`**  
(this contains all the building blocks for neural networks)

**Initialise model parameters** to be used in various computations (these could be different layers from `torch.nn`, single parameters, hard-coded values or functions)

`requires_grad=True` means PyTorch will track the gradients of this specific parameter for use with `torch.autograd` and gradient descent (for many `torch.nn` modules, `requires_grad=True` is set by default)

Any subclass of `nn.Module` needs to override `forward()` (this defines the forward computation of the model)

Basic building blocks of creating a PyTorch model by subclassing `nn.Module`. For objects that subclass `nn.Module`, the `forward()` method must be defined.

**Resource:** See more of these essential modules and their uses cases in the [PyTorch Cheat Sheet](#).

## Checking the contents of a PyTorch model

Now we've got these out of the way, let's create a model instance with the class we've made and check its parameters using `.parameters()`.

In [8]:



Out[8]:

```
[Parameter containing:  
tensor([0.3367], requires_grad=True),  
Parameter containing:  
tensor([0.1288], requires_grad=True)]
```

We can also get the state (what the model contains) of the model using `.state_dict()`.

In [9]:

Out[9]:

```
OrderedDict([('weights', tensor([0.3367])), ('bias', tensor([0.1288]))])
```

Notice how the values for `weights` and `bias` from `model_0.state_dict()` come out as random float tensors?

This is because we initialized them above using `torch.randn()`.

Essentially we want to start from random parameters and get the model to update them towards parameters that fit our data best (the hardcoded `weight` and `bias` values we set when creating our straight line data).

**Exercise:** Try changing the `torch.manual_seed()` value two cells above, see what happens to the weights and bias values.

Because our model starts with random values, right now it'll have poor predictive power.

## Making predictions using `torch.inference_mode()`

To check this we can pass it the test data `x_test` to see how closely it predicts `y_test`.

When we pass data to our model, it'll go through the model's `forward()` method and produce a result using the computation we've defined.

Let's make some predictions.

In [10]:

Hmm?

You probably noticed we used `torch.inference_mode()` as a **context manager** (that's what the `with torch.inference_mode():` is) to make the predictions.

As the name suggests, `torch.inference_mode()` is used when using a model for inference (making predictions).

`torch.inference_mode()` turns off a bunch of things (like gradient tracking, which is necessary for training but not for inference) to make **forward-passes** (data going through the `forward()` method) faster.

**Note:** In older PyTorch code, you may also see `torch.no_grad()` being used for inference. While `torch.inference_mode()` and `torch.no_grad()` do similar things, `torch.inference_mode()` is newer, potentially faster and preferred. See this [Tweet from PyTorch](#) for more.

We've made some predictions, let's see what they look like.

In [11]:

```
Number of testing samples: 10
Number of predictions made: 10
Predicted values:
tensor([[0.3982],
       [0.4049],
       [0.4116],
       [0.4184],
       [0.4251],
       [0.4318],
       [0.4386],
       [0.4453],
       [0.4520],
       [0.4588]])
```

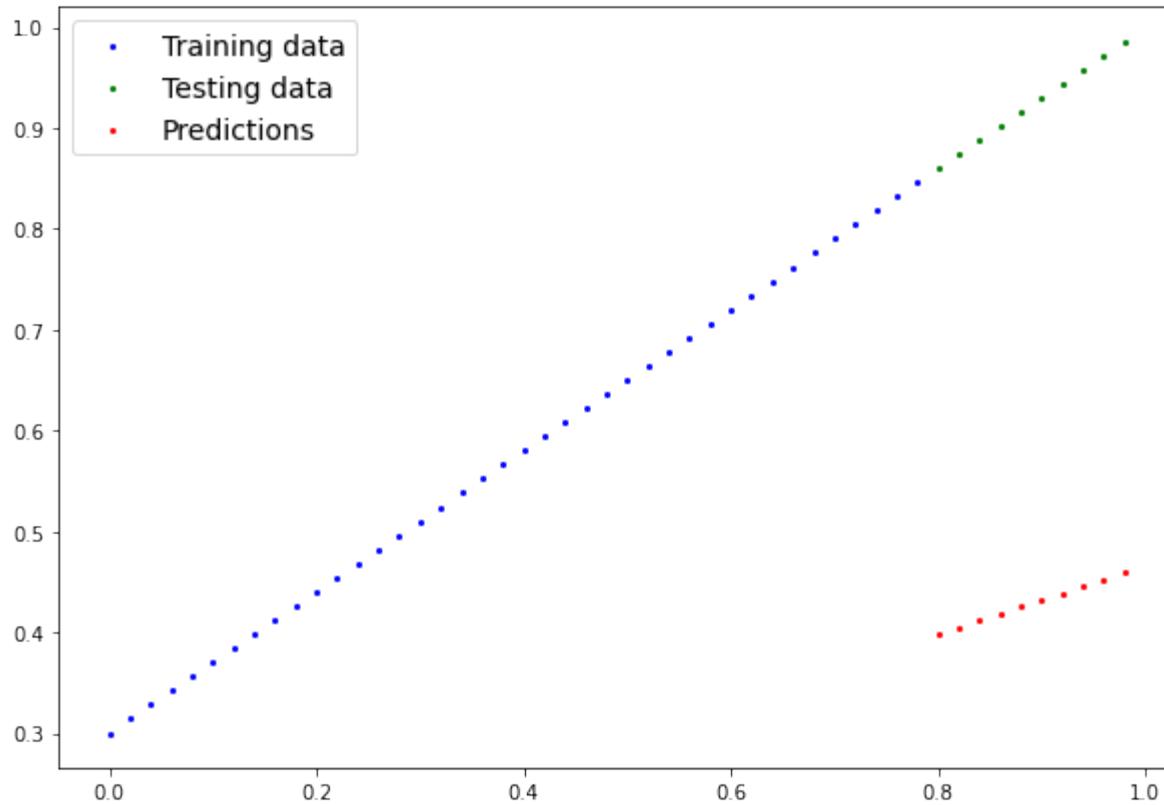
Notice how there's one prediction value per testing sample.

This is because of the kind of data we're using. For our straight line, one `x` value maps to one `y` value.

However, machine learning models are very flexible. You could have 100 `x` values mapping to one, two, three or 10 `y` values. It all depends on what you're working on.

Our predictions are still numbers on a page, let's visualize them with our `plot_predictions()` function we created above.

In [12]:



In [13]:

Out[13]:

```
tensor([[0.4618],  
       [0.4691],  
       [0.4764],  
       [0.4836],  
       [0.4909],  
       [0.4982],  
       [0.5054],  
       [0.5127],  
       [0.5200],  
       [0.5272]])
```

Woah! Those predictions look pretty bad...

This make sense though when you remember our model is just using random parameter values to make predictions.

It hasn't even looked at the blue dots to try to predict the green dots.

Time to change that.

### 3. Train model

Right now our model is making predictions using random parameters to make calculations, it's basically guessing (randomly).

To fix that, we can update its internal parameters (I also refer to *parameters* as patterns), the `weights` and `bias` values we set randomly using `nn.Parameter()` and `torch.randn()` to be something that better represents the data.

We could hard code this (since we know the default values `weight=0.7` and `bias=0.3`) but where's the fun in that?

Much of the time you won't know what the ideal parameters are for a model.

Instead, it's much more fun to write code to see if the model can try and figure them out itself.

### Creating a loss function and optimizer in PyTorch

For our model to update its parameters on its own, we'll need to add a few more things to our recipe.

And that's a **loss function** as well as an **optimizer**.

The roles of these are:

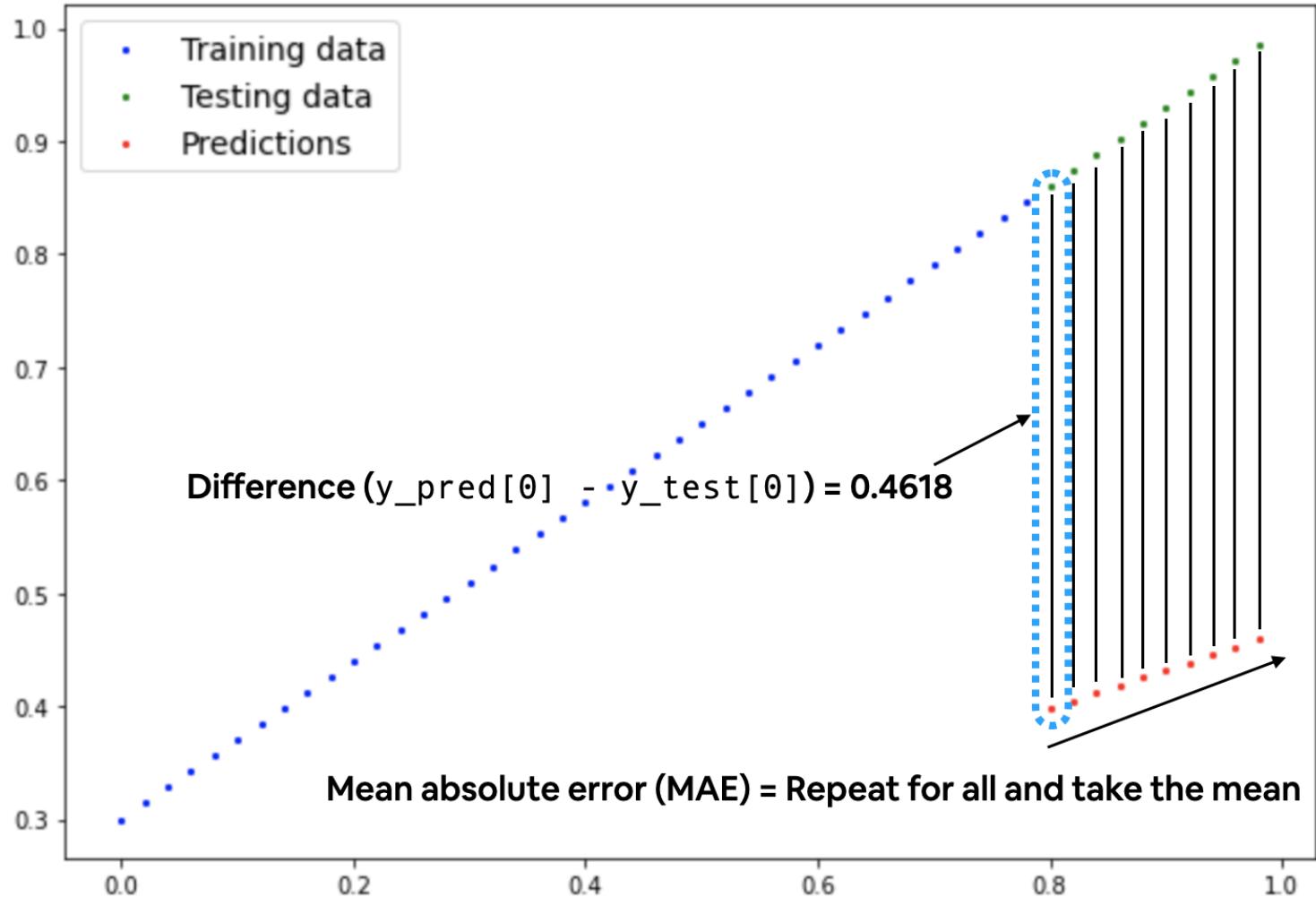
Function	What does it do?	Where does it live in PyTorch?	Common values
<b>Loss function</b>	Measures how wrong your models predictions (e.g. <code>y_preds</code> ) are compared to the truth labels (e.g. <code>y_test</code> ). Lower the better.	PyTorch has plenty of built-in loss functions in <code>torch.nn</code> .	Mean absolute error (MAE) for regression problems ( <code>torch.nn.L1Loss()</code> ). Binary cross entropy for binary classification problems ( <code>torch.nn.BCELoss()</code> ).
<b>Optimizer</b>	Tells your model how to update its internal parameters to best lower the loss.	You can find various optimization function implementations in <code>torch.optim</code> .	Stochastic gradient descent ( <code>torch.optim.SGD()</code> ). Adam optimizer ( <code>torch.optim.Adam()</code> ).

Let's create a loss function and an optimizer we can use to help improve our model.

Depending on what kind of problem you're working on will depend on what loss function and what optimizer you use.

However, there are some common values, that are known to work well such as the SGD (stochastic gradient descent) or Adam optimizer. And the MAE (mean absolute error) loss function for regression problems (predicting a number) or binary cross entropy loss function for classification problems (predicting one thing or another).

For our problem, since we're predicting a number, let's use MAE (which is under `torch.nn.L1Loss()`) in PyTorch as our loss function.



Mean absolute error (MAE, in PyTorch: `torch.nn.L1Loss`) measures the absolute difference between two points (predictions and labels) and then takes the mean across all examples.

And we'll use SGD, `torch.optim.SGD(params, lr)` where:

- `params` is the target model parameters you'd like to optimize (e.g. the `weights` and `bias` values we randomly set before).
- `lr` is the **learning rate** you'd like the optimizer to update the parameters at, higher means the optimizer will try larger updates (these can sometimes be too large and the optimizer will fail to work), lower means the optimizer will try smaller updates (these can sometimes be too small and the optimizer will take too long to find the ideal values). The learning rate is considered a **hyperparameter** (because it's set by a machine learning engineer).

Common starting values for the learning rate are `0.01`, `0.001`, `0.0001`, however, these can also be adjusted over time (this is called [learning rate scheduling](#)).

Woah, that's a lot, let's see it in code.

In [14]:



## Creating an optimization loop in PyTorch

Woohoo! Now we've got a loss function and an optimizer, it's now time to create a **training loop** (and **testing loop**).

The training loop involves the model going through the training data and learning the relationships between the features **and** labels.

The testing loop involves going through the testing data and evaluating how good the patterns are that the model learned on the training data (the model never sees the testing data during training).

Each of these is called a "loop" because we want our model to look (loop through) at each sample in each dataset.

To create these we're going to write a Python `for` loop in the theme of the [unofficial PyTorch optimization loop song](#) (there's a [video version too](#)).



**Daniel Bourke**  
@mrdbourke

Let's sing the [@PyTorch optimization loop song](#)

It's train time!  
do the forward pass,  
calculate the loss,  
optimizer zero grad,  
losssss backwards!

Optimizer step step step

Let's test now!  
with torch no grad:  
do the forward pass,  
calculate the loss,  
watch it go down down down!

*The unofficial PyTorch optimization loops song, a fun way to remember the steps in a PyTorch training (and testing) loop.*

There will be a fair bit of code but nothing we can't handle.

## PyTorch training loop

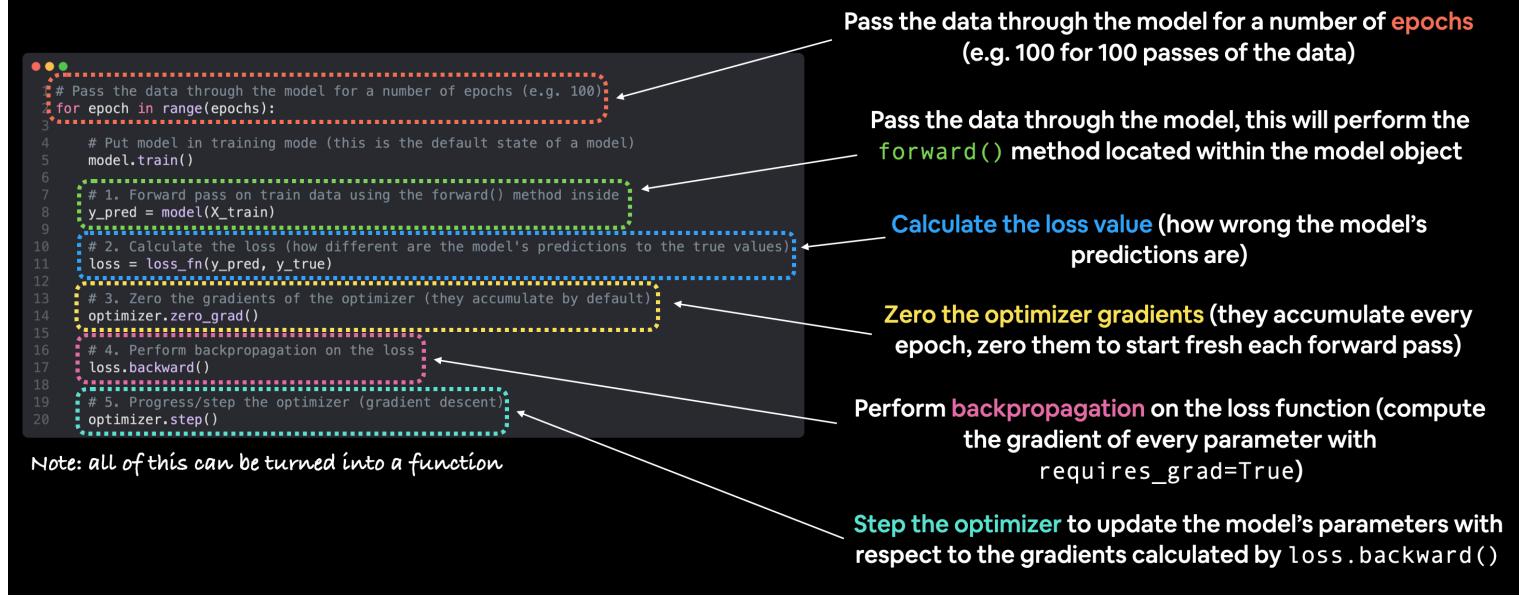
For the training loop, we'll build the following steps:

Number	Step name	What does it do?	Code example
1	Forward pass	The model goes through all of the training data once, performing its <code>forward()</code> function calculations.	<code>model(x_train)</code>
2	Calculate the loss	The model's outputs (predictions) are compared to the ground truth and evaluated to see how wrong they are.	<code>loss = loss_fn(y_pred, y_train)</code>
3	Zero gradients	The optimizers gradients are set to zero (they are accumulated by default) so they can be recalculated for the specific training step.	<code>optimizer.zero_grad()</code>
4	Perform	Computes the gradient of the loss with respect for every model parameter to	<code>loss.backward()</code>

backpropagation  
on the loss      be updated (each parameter with `requires_grad=True`). This is known as  
**backpropagation**, hence "backwards".  
d()

5	Update the optimizer <b>(gradient descent)</b>	Update the parameters with <code>requires_grad=True</code> with respect to the loss gradients in order to improve them. optimizer.st ep()
---	--	--

# PyTorch training loop



**Note:** The above is just one example of how the steps could be ordered or described. With experience you'll find making PyTorch training loops can be quite flexible.

And on the ordering of things, the above is a good default order but you may see slightly different orders. Some rules of thumb:

- Calculate the loss (`loss = ...`) *before* performing backpropagation on it (`loss.backward()`).
- Zero gradients (`optimizer.zero_grad()`) *before* stepping them (`optimizer.step()`).
- Step the optimizer (`optimizer.step()`) *after* performing backpropagation on the loss (`loss.backward()`).

For resources to help understand what's happening behind the scenes with backpropagation and gradient descent, see the extra-curriculum section.

## PyTorch testing loop

As for the testing loop (evaluating our model), the typical steps include:

Number	Step name	What does it do?	Code example
1	Forward pass	The model goes through all of the training data once, performing its <code>forward()</code> function calculations.	<code>model(x_test)</code>
2	Calculate the loss	The model's outputs (predictions) are compared to the ground truth and evaluated to see how wrong they are.	<code>loss = loss_fn(y_pred, y_test)</code>
3	Calulate evaluation metrics (optional)	Alongside the loss value you may want to calculate other evaluation metrics such as accuracy on the test set.	Custom functions

Notice the testing loop doesn't contain performing backpropagation (`loss.backward()`) or stepping the optimizer (`optimizer.step()`), this is because no parameters in the model are being changed during testing, they've already been calculated. For testing, we're only interested in the output of the forward pass through the model.

# PyTorch testing loop

```

1 # Setup empty lists to keep track of model progress
2 epoch_count = []
3 train_loss_values = []
4 test_loss_values = []

6 # Pass the data through the model for a number of epochs (e.g. 100 epochs):
7 for epoch in range(epochs):
8
9     ### Training loop code here #####
10
11    ### Testing starts #####
12
13    # Put the model in evaluation mode
14    model.eval()
15
16    # Turn on inference mode context manager
17    with torch.inference_mode():
18        # 1. Forward pass on test data
19        test_pred = model(X_test)
20
21        # 2. Calculate loss on test data
22        test_loss = loss_fn(test_pred, y_test)
23
24    # Print out what's happening every 10 epochs
25    if epoch % 10 == 0:
26        epoch_count.append(epoch)
27        train_loss_values.append(loss)
28        test_loss_values.append(test_loss)
29        print(f"Epoch: {epoch} | MAE Train Loss: {loss} | MAE Test Loss: {test_loss}")

```

Note: all of this can be turned into a function

Create empty lists for storing useful values (helpful for tracking model progress)

Tell the model we want to evaluate rather than train (this turns off functionality used for training but not evaluation)

Turn on `torch.inference_mode()` context manager to disable functionality such as gradient tracking for inference (gradient tracking not needed for inference)

Pass the test data through the model (this will call the model's implemented `forward()` method)

Calculate the test loss value (how wrong the model's predictions are on the test dataset, lower is better)

Display information outputs for how the model is doing during training/testing every ~10 epochs (note: what gets printed out here can be adjusted for specific problems)

Let's put all of the above together and train our model for 100 **epochs** (forward passes through the data) and we'll evaluate it every 10 epochs.

In [15]:

















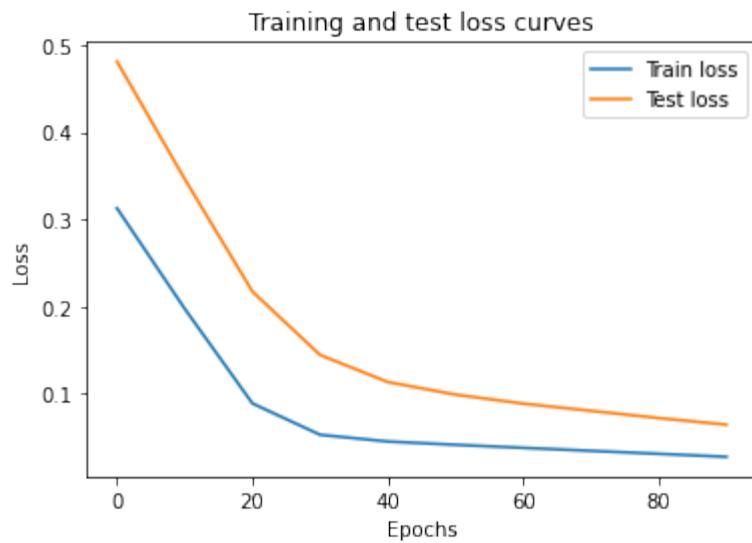




```
Epoch: 0 | MAE Train Loss: 0.31288138031959534 | MAE Test Loss: 0.48106518387794495
Epoch: 10 | MAE Train Loss: 0.1976713240146637 | MAE Test Loss: 0.3463551998138428
Epoch: 20 | MAE Train Loss: 0.08908725529909134 | MAE Test Loss: 0.21729660034179688
Epoch: 30 | MAE Train Loss: 0.053148526698350906 | MAE Test Loss: 0.14464017748832703
Epoch: 40 | MAE Train Loss: 0.04543796554207802 | MAE Test Loss: 0.11360953003168106
Epoch: 50 | MAE Train Loss: 0.04167863354086876 | MAE Test Loss: 0.09919948130846024
Epoch: 60 | MAE Train Loss: 0.03818932920694351 | MAE Test Loss: 0.08886633068323135
Epoch: 70 | MAE Train Loss: 0.03476089984178543 | MAE Test Loss: 0.0805937647819519
Epoch: 80 | MAE Train Loss: 0.03132382780313492 | MAE Test Loss: 0.07232122868299484
Epoch: 90 | MAE Train Loss: 0.02788739837706089 | MAE Test Loss: 0.06473556160926819
```

Oh would you look at that! Looks like our loss is going down with every epoch, let's plot it to find out.

In [16]:



Nice! The **loss curves** show the loss going down over time. Remember, loss is the measure of how *wrong* your model is, so the lower the better.

But why did the loss go down?

Well, thanks to our loss function and optimizer, the model's internal parameters (`weights` and `bias`) were updated to better reflect the underlying patterns in the data.

Let's inspect our model's `.state_dict()` to see how close our model gets to the original values we set for weights and bias.

In [17]:



The model learned the following values for weights and bias:

```
OrderedDict([('weights', tensor([0.5784])), ('bias', tensor([0.3513]))])
```

And the original values for weights and bias are:

```
weights: 0.7, bias: 0.3
```

Wow! How cool is that?

Our model got very close to calculate the exact original values for `weight` and `bias` (and it would probably get even closer if we trained it for longer).

**Exercise:** Try changing the `epochs` value above to 200, what happens to the loss curves and the weights and bias parameter values of the model?

It'd likely never guess them *perfectly* (especially when using more complicated datasets) but that's okay, often you can do very cool things with a close approximation.

This is the whole idea of machine learning and deep learning, **there are some ideal values that describe our data** and rather than figuring them out by hand, **we can train a model to figure them out programmatically**.

## 4. Making predictions with a trained PyTorch model (inference)

Once you've trained a model, you'll likely want to make predictions with it.

We've already seen a glimpse of this in the training and testing code above, the steps to do it outside of the training/testing loop are similar.

There are three things to remember when making predictions (also called performing inference) with a PyTorch model:

1. Set the model in evaluation mode (`model.eval()`).
2. Make the predictions using the inference mode context manager (`with torch.inference_mode(): ...`).
3. All predictions should be made with objects on the same device (e.g. data and model on GPU only or data and model on CPU only).

The first two items make sure all helpful calculations and settings PyTorch uses behind the scenes during training but aren't necessary for inference are turned off (this results in faster computation). And the third ensures that you won't run into cross-device errors.

In [18]:





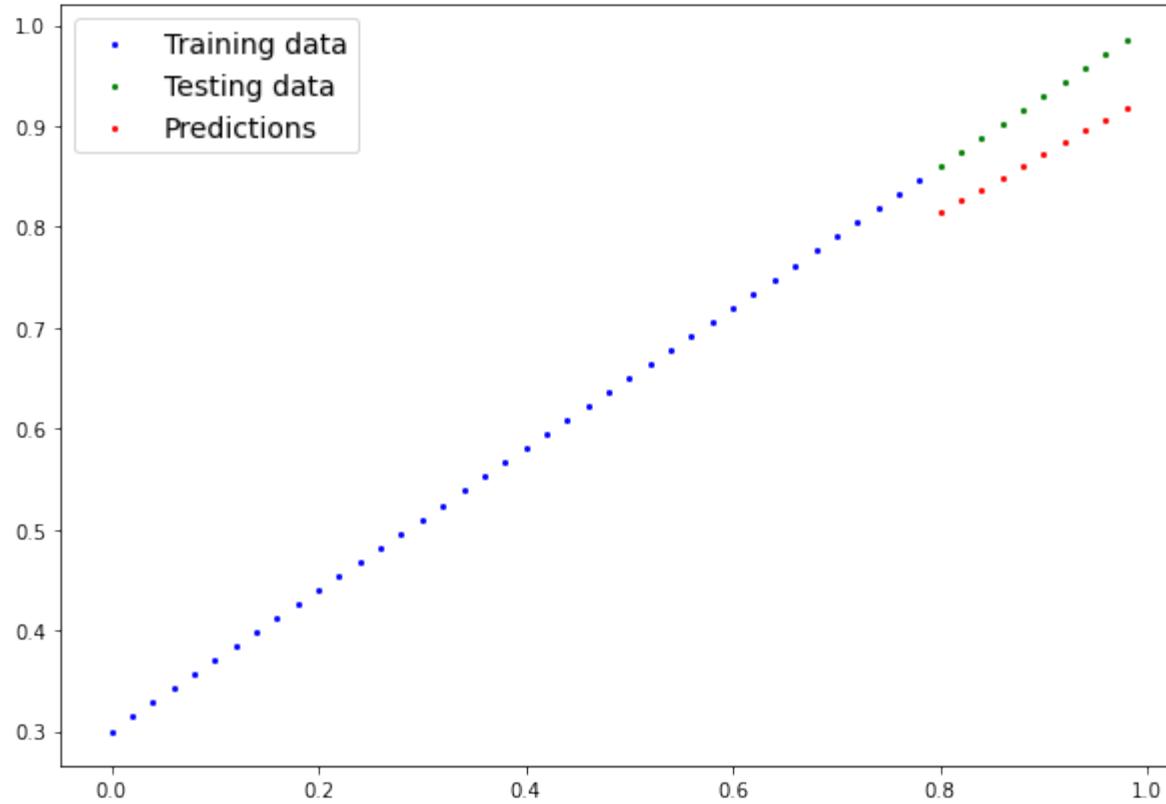
Out[18]:

```
tensor([[0.8141],  
       [0.8256],  
       [0.8372],  
       [0.8488],  
       [0.8603],  
       [0.8719],  
       [0.8835],  
       [0.8950],  
       [0.9066],
```

```
[0.9182]])
```

Nice! We've made some predictions with our trained model, now how do they look?

In [19]:



Woohoo! Those red dots are looking far closer than they were before!

Let's get onto saving and reloading a model in PyTorch.

## 5. Saving and loading a PyTorch model

If you've trained a PyTorch model, chances are you'll want to save it and export it somewhere.

As in, you might train it on Google Colab or your local machine with a GPU but you'd like to now export it to some sort of application where others can use it.

Or maybe you'd like to save your progress on a model and come back and load it back later.

For saving and loading models in PyTorch, there are three main methods you should be aware of (all of below have been taken from the [PyTorch saving and loading models guide](#)):

## PyTorch method

### What does it do?

`torch.save`

Saves a serialized object to disk using Python's `pickle` utility. Models, tensors and various other Python objects like dictionaries can be saved using `torch.save`.

`torch.load`

Uses `pickle`'s unpickling features to deserialize and load pickled Python object files (like models, tensors or dictionaries) into memory. You can also set which device to load the object to (CPU, GPU etc).

```
torch.nn.Module
.load_state_dict
t
```

Loads a model's parameter dictionary (`model.state_dict()`) using a saved `state_dict()` object.

**Note:** As stated in [Python's `pickle` documentation](#), the `pickle` module **is not secure**. That means you should only ever unpickle (load) data you trust. That goes for loading PyTorch models as well. Only ever use saved PyTorch models from sources you trust.

## Saving a PyTorch model's `state_dict()`

The [recommended way](#) for saving and loading a model for inference (making predictions) is by saving and loading a model's `state_dict()`.

Let's see how we can do that in a few steps:

1. We'll create a directory for saving models to called `models` using Python's `pathlib` module.
2. We'll create a file path to save the model to.
3. We'll call `torch.save(obj, f)` where `obj` is the target model's `state_dict()` and `f` is the filename of where to save the model.

**Note:** It's common convention for PyTorch saved models or objects to end with `.pt` or `.pth`, like `saved_model_01.pth`.

In [20]:





```
Saving model to: models/01_pytorch_workflow_model_0.pth
```

In [21]:

```
-rw-rw-r-- 1 daniel daniel 1063 May 10 10:26 models/01_pytorch_workflow_model_0.pth
```

## Loading a saved PyTorch model's `state_dict()`

Since we've now got a saved model `state_dict()` at `models/01_pytorch_workflow_model_0.pth` we can now load it in using `torch.nn.Module.load_state_dict(torch.load(f))` where `f` is the filepath of our saved model `state_dict()`.

Why call `torch.load()` inside `torch.nn.Module.load_state_dict()`?

Because we only saved the model's `state_dict()` which is a dictionary of learned parameters and not the *entire* model, we first have to load the `state_dict()` with `torch.load()` and then pass that `state_dict()` to a new instance of our model (which is a subclass of `nn.Module`).

Why not save the entire model?

[Saving the entire model](#) rather than just the `state_dict()` is more intuitive, however, to quote the PyTorch documentation (italics mine):

The disadvantage of this approach (*saving the whole model*) is that the serialized data is bound to the specific classes and the exact directory structure used when the model is saved...

Because of this, your code can break in various ways when used in other projects or after refactors.

So instead, we're using the flexible method of saving and loading just the `state_dict()`, which again is basically a dictionary of model parameters.

Let's test it out by created another instance of `LinearRegressionModel()`, which is a subclass of `torch.nn.Module` and will hence have the in-built method `load_state_dict()`.

In [22]:

Out[22]:

```
<All keys matched successfully>
```

Excellent! It looks like things matched up.

Now to test our loaded model, let's perform inference with it (make predictions) on the test data.

Remember the rules for performing inference with PyTorch models?

If not, here's a refresher:

**PyTorch inference rules**



In [23]:



Now we've made some predictions with the loaded model, let's see if they're the same as the previous predictions.

In [24]:

Out[24]:

```
tensor([[True],  
       [True],  
       [True],  
       [True],  
       [True],  
       [True],  
       [True],  
       [True],  
       [True],  
       [True]]))
```

Nice!

It looks like the loaded model predictions are the same as the previous model predictions (predictions made prior to saving). This indicates our model is saving and loading as expected.

**Note:** There are more methods to save and load PyTorch models but I'll leave these for extra-curriculum and further reading. See the [PyTorch guide for saving and loading models](#) for more.

## 6. Putting it all together

We've covered a fair bit of ground so far.

But once you've had some practice, you'll be performing the above steps like dancing down the street.

Speaking of practice, let's put everything we've done so far together.

Except this time we'll make our code device agnostic (so if there's a GPU available, it'll use it and if not, it will default to the CPU).

There'll be far less commentary in this section than above since what we're going to go through has already been covered.

We'll start by importing the standard libraries we need.

**Note:** If you're using Google Colab, to setup a GPU, go to Runtime -> Change runtime type -> Hardware acceleration -> GPU. If you do this, it will reset the Colab runtime and you will lose saved variables.

In [25]:

Out[25]:

'1.11.0'

Now let's start making our code device agnostic by setting `device="cuda"` if it's available, otherwise it'll default to `device="cpu"`.

In [26]:

```
Using device: cuda
```

If you've got access to a GPU, the above should've printed out:

```
Using device: cuda
```

Otherwise, you'll be using a CPU for the following computations. This is fine for our small dataset but it will take longer for larger datasets.

## 6.1 Data

Let's create some data just like before.

First, we'll hard-code some `weight` and `bias` values.

Then we'll make a range of numbers between 0 and 1, these will be our `x` values.

Finally, we'll use the `x` values, as well as the `weight` and `bias` values to create `y` using the linear regression formula ( $y = \text{weight} * x + \text{bias}$ ).

In [27]:



Out[27]:

```
(tensor([[0.0000,
         [0.0200],
         [0.0400],
         [0.0600],
         [0.0800],
         [0.1000],
         [0.1200],
         [0.1400],
         [0.1600],
         [0.1800]]),
```

```
tensor([[0.3000],  
       [0.3140],  
       [0.3280],  
       [0.3420],  
       [0.3560],  
       [0.3700],  
       [0.3840],  
       [0.3980],  
       [0.4120],  
       [0.4260]]))
```

Wonderful!

Now we've got some data, let's split it into training and test sets.

We'll use an 80/20 split with 80% training data and 20% testing data.

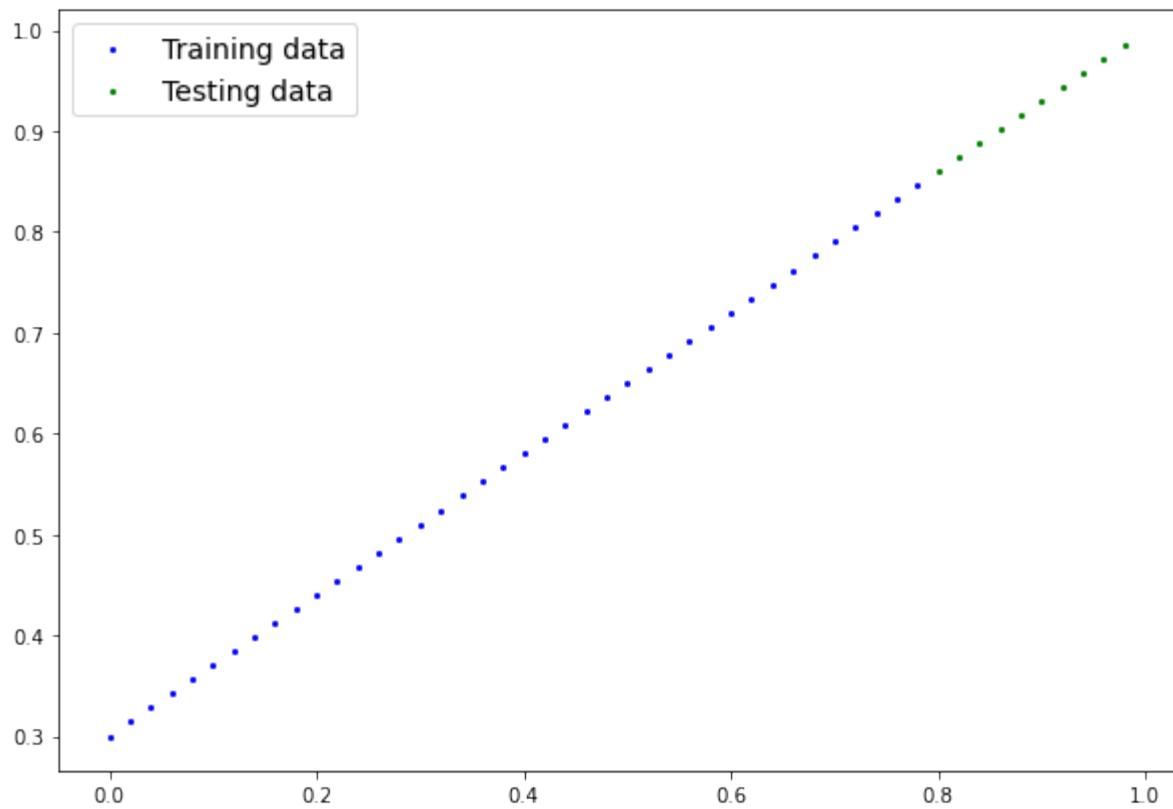
In [28]:

Out[28]:

```
(40, 40, 10, 10)
```

Excellent, let's visualize them to make sure they look okay.

In [29]:



## 6.2 Building a PyTorch linear model

We've got some data, now it's time to make a model.

We'll create the same style of model as before except this time, instead of defining the weight and bias parameters of our model manually using `nn.Parameter()`, we'll use `nn.Linear(in_features, out_features)` to do it for us.

Where `in_features` is the number of dimensions your input data has and `out_features` is the number of dimensions you'd like it to be output to.

In our case, both of these are `1` since our data has `1` input feature (`x`) per label (`y`).

```

1 # Create a linear regression model in PyTorch
2 class LinearRegressionModel(nn.Module):
3     def __init__(self):
4         super().__init__()
5
6         # Initialize model parameters
7         self.weights = nn.Parameter(torch.randn(1,
8             requires_grad=True,
9             dtype=torch.float
10        ))
11
12         self.bias = nn.Parameter(torch.randn(1,
13             requires_grad=True,
14             dtype=torch.float
15        ))
16
17     # forward() defines the computation in the model
18     def forward(self, x: torch.Tensor) -> torch.Tensor:
19         return self.weights * x + self.bias
20

```

```

1 # Create a linear regression model in PyTorch with nn.Linear
2 class LinearRegressionModel(nn.Module):
3     def __init__(self):
4         super().__init__()
5         # Use nn.Linear() for creating the model parameters
6         self.linear_layer = nn.Linear(in_features=1,
7                                       out_features=1)
8
9     # forward() defines the computation in the model
10    def forward(self, x: torch.Tensor) -> torch.Tensor:
11        return self.linear_layer(x)

```

**Linear regression model with nn.Parameter**

**Linear regression model with nn.Linear**

*Creating a linear regression model using `nn.Parameter` versus using `nn.Linear`. There are plenty more examples of where the `torch.nn` module has pre-built computations, including many popular and useful neural network layers.*

In [30]:









Out[30]:

```
(LinearRegressionModelV2(  
    (linear_layer): Linear(in_features=1, out_features=1, bias=True)  
)  
,  
OrderedDict([('linear_layer.weight', tensor([[0.7645]])),  
            ('linear_layer.bias', tensor([0.8300]))]))
```

Notice the outputs of `model_1.state_dict()`, the `nn.Linear()` layer created a random `weight` and `bias` parameter for us.

Now let's put our model on the GPU (if it's available).

We can change the device our PyTorch objects are on using `.to(device)`.

First let's check the model's current device.

In [31]:

Out[31]:

```
device(type='cpu')
```

Wonderful, looks like the model's on the CPU by default.

Let's change it to be on the GPU (if it's available).

In [32]:

Out[32]:

```
device(type='cuda', index=0)
```

Nice! Because of our device agnostic code, the above cell will work regardless of whether a GPU is available or not.

If you do have access to a CUDA-enabled GPU, you should see an output of something like:

```
device(type='cuda', index=0)
```

## 6.3 Training

Time to build a training and testing loop.

First we'll need a loss function and an optimizer.

Let's use the same functions we used earlier, `nn.L1Loss()` and `torch.optim.SGD()`.

We'll have to pass the new model's parameters (`model.parameters()`) to the optimizer for it to adjust them during training.

The learning rate of `0.1` worked well before too so let's use that again.

In [33]:

Beautiful, loss function and optimizer ready, now let's train and evaluate our model using a training and testing loop.

The only different thing we'll be doing in this step compared to the previous training loop is putting the data on the target `device`.

We've already put our model on the target `device` using `model_1.to(device)`.

And we can do the same with the data.

That way if the model is on the GPU, the data is on the GPU (and vice versa).

Let's step things up a notch this time and set `epochs=1000`.

If you need a reminder of the PyTorch training loop steps, see below.

### PyTorch training loop steps

In [34]:













```
Epoch: 0 | Train loss: 0.5551779866218567 | Test loss: 0.5739762187004089
Epoch: 100 | Train loss: 0.006215683650225401 | Test loss: 0.014086711220443249
Epoch: 200 | Train loss: 0.0012645035749301314 | Test loss: 0.013801801018416882
Epoch: 300 | Train loss: 0.0012645035749301314 | Test loss: 0.013801801018416882
Epoch: 400 | Train loss: 0.0012645035749301314 | Test loss: 0.013801801018416882
Epoch: 500 | Train loss: 0.0012645035749301314 | Test loss: 0.013801801018416882
Epoch: 600 | Train loss: 0.0012645035749301314 | Test loss: 0.013801801018416882
Epoch: 700 | Train loss: 0.0012645035749301314 | Test loss: 0.013801801018416882
Epoch: 800 | Train loss: 0.0012645035749301314 | Test loss: 0.013801801018416882
Epoch: 900 | Train loss: 0.0012645035749301314 | Test loss: 0.013801801018416882
```

**Note:** Due to the random nature of machine learning, you will likely get slightly different results (different loss and prediction values) depending on whether your model was trained on CPU or GPU. This is true even if you use the same random seed on either device. If the difference is large, you may want to look for errors, however, if it is small (ideally it is), you can ignore it.

Nice! That loss looks pretty low.

Let's check the parameters our model has learned and compare them to the original parameters we hard-coded.

In [35]:



The model learned the following values for weights and bias:  
OrderedDict([('linear\_layer.weight', tensor([[0.6968]]), device='cuda:0')),  
('linear\_layer.bias', tensor([0.3025], device='cuda:0'))])

And the original values for weights and bias are:  
weights: 0.7, bias: 0.3

Ho ho! Now that's pretty darn close to a perfect model.

Remember though, in practice, it's rare that you'll know the perfect parameters ahead of time.

And if you knew the parameters your model had to learn ahead of time, what would be the fun of machine learning?

Plus, in many real-world machine learning problems, the number of parameters can well exceed tens of millions.

I don't know about you but I'd rather write code for a computer to figure those out rather than doing it by hand.

## 6.4 Making predictions

Now we've got a trained model, let's turn on its evaluation mode and make some predictions.

In [36]:

Out[36]:

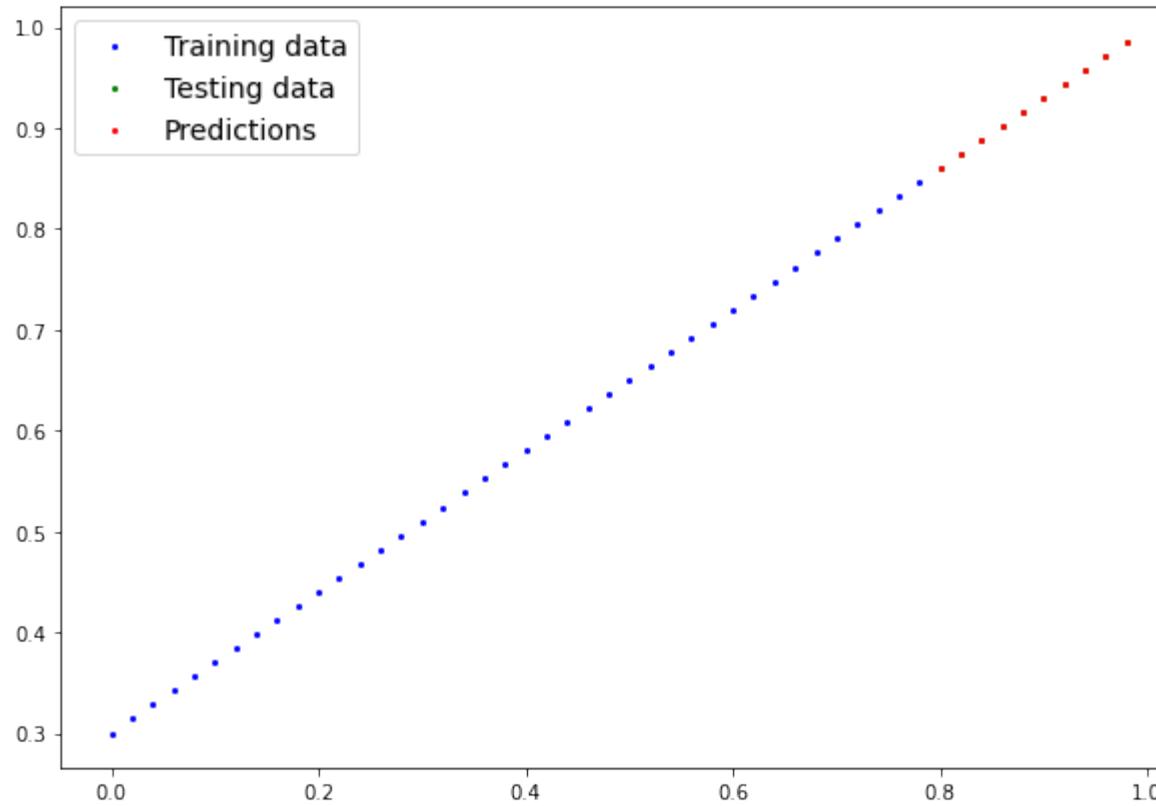
```
tensor([[0.8600],
       [0.8739],
       [0.8878],
       [0.9018],
       [0.9157],
       [0.9296],
       [0.9436],
       [0.9575],
       [0.9714],
       [0.9854]], device='cuda:0')
```

If you're making predictions with data on the GPU, you might notice the output of the above has `device='cuda:0'` towards the end. That means the data is on CUDA device 0 (the first GPU your system has access to due to zero-indexing), if you end up using multiple GPUs in the future, this number may be higher.

Now let's plot our model's predictions.

**Note:** Many data science libraries such as pandas, matplotlib and NumPy aren't capable of using data that is stored on GPU. So you might run into some issues when trying to use a function from one of these libraries with tensor data not stored on the CPU. To fix this, you can call `.cpu()` on your target tensor to return a copy of your target tensor on the CPU.

In [37]:



Woah! Look at those red dots, they line up almost perfectly with the green dots. I guess the extra epochs helped.

## 6.5 Saving and loading a model

We're happy with our models predictions, so let's save it to file so it can be used later.

In [38]:



```
Saving model to: models/01_pytorch_workflow_model_1.pth
```

And just to make sure everything worked well, let's load it back in.

We'll:

- Create a new instance of the `LinearRegressionModelV2()` class
- Load in the model state dict using `torch.nn.Module.load_state_dict()`
- Send the new instance of the model to the target device (to ensure our code is device-agnostic)

In [39]:

```
Loaded model:  
LinearRegressionModelV2(  
    (linear_layer): Linear(in_features=1, out_features=1, bias=True)  
)  
Model on device:  
cuda:0
```

Now we can evaluate the loaded model to see if its predictions line up with the predictions made prior to saving.

In [40]:

Out[40]:

```
tensor([[True],  
       [True],  
       [True],  
       [True],  
       [True],  
       [True],  
       [True],  
       [True],  
       [True],  
       [True]], device='cuda:0')
```

Everything adds up! Nice!

Well, we've come a long way. You've now built and trained your first two neural network models in PyTorch!

Time to practice your skills.

# Exercises

All exercises have been inspired from code throughout the notebook.

There is one exercise per major section.

You should be able to complete them by referencing their specific section.

**Note:** For all exercises, your code should be device agnostic (meaning it could run on CPU or GPU if it's available).

1. Create a straight line dataset using the linear regression formula (`weight * x + bias`).
  - Set `weight=0.3` and `bias=0.9` there should be at least 100 datapoints total.
  - Split the data into 80% training, 20% testing.
  - Plot the training and testing data so it becomes visual.
2. Build a PyTorch model by subclassing `nn.Module`.
  - Inside should be a randomly initialized `nn.Parameter()` with `requires_grad=True`, one for `weights` and one for `bias`.
  - Implement the `forward()` method to compute the linear regression function you used to create the dataset in 1.
  - Once you've constructed the model, make an instance of it and check its `state_dict()`.
  - **Note:** If you'd like to use `nn.Linear()` instead of `nn.Parameter()` you can.
3. Create a loss function and optimizer using `nn.L1Loss()` and `torch.optim.SGD(params, lr)` respectively.
  - Set the learning rate of the optimizer to be 0.01 and the parameters to optimize should be the model parameters from the model you created in 2.
  - Write a training loop to perform the appropriate training steps for 300 epochs.
  - The training loop should test the model on the test dataset every 20 epochs.
4. Make predictions with the trained model on the test data.
  - Visualize these predictions against the original training and testing data (**note:** you may need to make sure the predictions are *not* on the GPU if you want to use non-CUDA-enabled libraries such as matplotlib to plot).
5. Save your trained model's `state_dict()` to file.
  - Create a new instance of your model class you made in 2. and load in the `state_dict()` you just saved to it.
  - Perform predictions on your test data with the loaded model and confirm they match the original model

predictions from 4.

**Resource:** See the [exercises notebooks templates](#) and [solutions](#) on the course GitHub.

## Extra-curriculum

- Listen to [The Unofficial PyTorch Optimization Loop Song](#) (to help remember the steps in a PyTorch training/testing loop).
- Read [What is `torch.nn`, really?](#) by Jeremy Howard for a deeper understanding of how one of the most important modules in PyTorch works.
- Spend 10-minutes scrolling through and checking out the [PyTorch documentation cheatsheet](#) for all of the different PyTorch modules you might come across.
- Spend 10-minutes reading the [loading and saving documentation on the PyTorch website](#) to become more familiar with the different saving and loading options in PyTorch.
- Spend 1-2 hours read/watching the following for an overview of the internals of gradient descent and backpropagation, the two main algorithms that have been working in the background to help our model learn.
  - [Wikipedia page for gradient descent](#)
  - [Gradient Descent Algorithm — a deep dive](#) by Robert Kwiatkowski
  - [Gradient descent, how neural networks learn video](#) by 3Blue1Brown
  - [What is backpropagation really doing?](#) video by 3Blue1Brown
  - [Backpropagation Wikipedia Page](#)

Previous

00. PyTorch Fundamentals

Next

02. PyTorch Neural Network Classification



[View Source Code](#) | [View Slides](#) | [Watch Video Walkthrough](#)

## 02. PyTorch Neural Network Classification

### What is a classification problem?

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

For example, you might want to:

Problem type	What is it?	Example
<b>Binary classification</b>	Target can be one of two options, e.g. yes or no	Predict whether or not someone has heart disease based on their health parameters.
<b>Multi-class classification</b>	Target can be one of more than two options	Decide whether a photo is of food, a person or a dog.
<b>Multi-label classification</b>	Target can be assigned more than one option	Predict what categories should be assigned to a Wikipedia article (e.g. mathematics, science & philosophy).

**“Is this email spam or not spam?”**

To: daniel@mrdourke.com Hey Daniel,	To: daniel@mrdourke.com Hay daniel...
This deep learning course is incredible! I can't wait to use what I've learned!	C0ongratulations! U win \$1139239230
<b>Not spam</b>	<b>Spam</b>

**“Is this a photo of sushi, steak or pizza?”**



**Binary classification**  
*(one thing or another)*

**Multiclass classification**  
*(more than one thing or another)*

**“What tags should this article have?”**



**Multilabel classification**

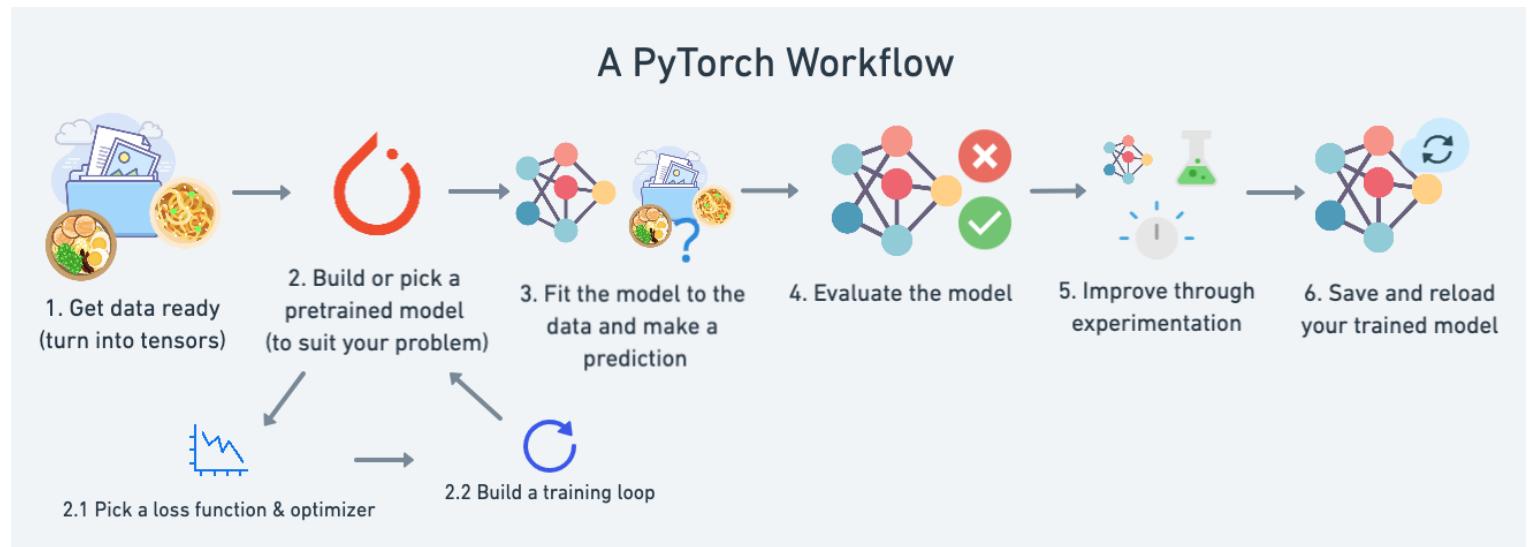
Classification, along with regression (predicting a number, covered in [notebook 01](#)) is one of the most common types of machine learning problems.

In this notebook, we're going to work through a couple of different classification problems with PyTorch.

In other words, taking a set of inputs and predicting what class those set of inputs belong to.

## What we're going to cover

In this notebook we're going to reiterate over the PyTorch workflow we covered in [01. PyTorch Workflow](#).



Except instead of trying to predict a straight line (predicting a number, also called a **regression problem**), we'll be working on a **classification problem**.

Specifically, we're going to cover:

Topic	Contents
<b>0. Architecture of a classification neural network</b>	Neural networks can come in almost any shape or size, but they typically follow a similar floor plan.
<b>1. Getting binary classification data ready</b>	Data can be almost anything but to get started we're going to create a simple binary classification dataset.
<b>2. Building a PyTorch classification model</b>	Here we'll create a model to learn patterns in the data, we'll also choose a <b>loss function</b> , <b>optimizer</b> and build a <b>training loop</b> specific to classification.
<b>3. Fitting the model to data (training)</b>	We've got data and a model, now let's let the model (try to) find patterns in the ( <b>training</b> ) data.
<b>4. Making predictions and evaluating a model (inference)</b>	Our model's found patterns in the data, let's compare its findings to the actual ( <b>testing</b> ) data.
<b>5. Improving a model (from a model perspective)</b>	We've trained and evaluated a model but it's not working, let's try a few things to improve it.
<b>6. Non-linearity</b>	So far our model has only had the ability to model straight lines, what about non-linear (non-straight) lines?
<b>7. Replicating non-linear functions</b>	We used <b>non-linear functions</b> to help model non-linear data, but what do these look like?

## 8. Putting it all together with multi-class classification

Let's put everything we've done so far for binary classification together with a multi-class classification problem.

# Where can you get help?

All of the materials for this course [live on GitHub](#).

And if you run into trouble, you can ask a question on the [Discussions page](#) there too.

There's also the [PyTorch developer forums](#), a very helpful place for all things PyTorch.

# 0. Architecture of a classification neural network

Before we get into writing code, let's look at the general architecture of a classification neural network.

Hyperparameter	Binary Classification	Multiclass classification
<b>Input layer shape</b> ( <code>in_features</code> )	Same as number of features (e.g. 5 for age, sex, height, weight, smoking status in heart disease prediction)	Same as binary classification
<b>Hidden layer(s)</b>	Problem specific, minimum = 1, maximum = unlimited	Same as binary classification
<b>Neurons per hidden layer</b>	Problem specific, generally 10 to 512	Same as binary classification
<b>Output layer shape</b> ( <code>out_features</code> )	1 (one class or the other)	1 per class (e.g. 3 for food, person or dog photo)
<b>Hidden layer activation</b>	Usually <a href="#">ReLU</a> (rectified linear unit) but <a href="#">can be many others</a>	Same as binary classification
<b>Output activation</b>	<a href="#">Sigmoid</a> ( <code>torch.sigmoid</code> in PyTorch)	<a href="#">Softmax</a> ( <code>torch.softmax</code> in PyTorch)
<b>Loss function</b>	<a href="#">Binary crossentropy</a> ( <code>torch.nn.BCELoss</code> in PyTorch)	Cross entropy ( <code>torch.nn.CrossEntropyLoss</code> in PyTorch)
<b>Optimizer</b>	<a href="#">SGD</a> (stochastic gradient descent), <a href="#">Adam</a> (see <code>torch.optim</code> for more options)	Same as binary classification

Of course, this ingredient list of classification neural network components will vary depending on the problem you're working on.

But it's more than enough to get started.

We're going to get hands-on with this setup throughout this notebook.

## 1. Make classification data and get it ready

Let's begin by making some data.

We'll use the `make_circles()` method from Scikit-Learn to generate two circles with different coloured dots.

In [1]:



Alright, now let's view the first 5 `x` and `y` values.

In [2]:

```
First 5 X features:
```

```
[[ 0.75424625  0.23148074]
 [-0.75615888  0.15325888]
 [-0.81539193  0.17328203]
 [-0.39373073  0.69288277]
 [ 0.44220765 -0.89672343]]
```

```
First 5 y labels:
```

```
[1 1 1 1 0]
```

Looks like there's two `x` values per one `y` value.

Let's keep following the data explorer's motto of *visualize, visualize, visualize* and put them into a pandas DataFrame.

In [3]:

Out[3]:

	X1	X2	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
4	0.442208	-0.896723	0
5	-0.479646	0.676435	1
6	-0.013648	0.803349	1
7	0.771513	0.147760	1
8	-0.169322	-0.793456	1
9	-0.121486	1.021509	0

It looks like each pair of  $x$  features ( $x_1$  and  $x_2$ ) has a label ( $y$ ) value of either 0 or 1.

This tells us that our problem is **binary classification** since there's only two options (0 or 1).

How many values of each class is there?

In [4]:

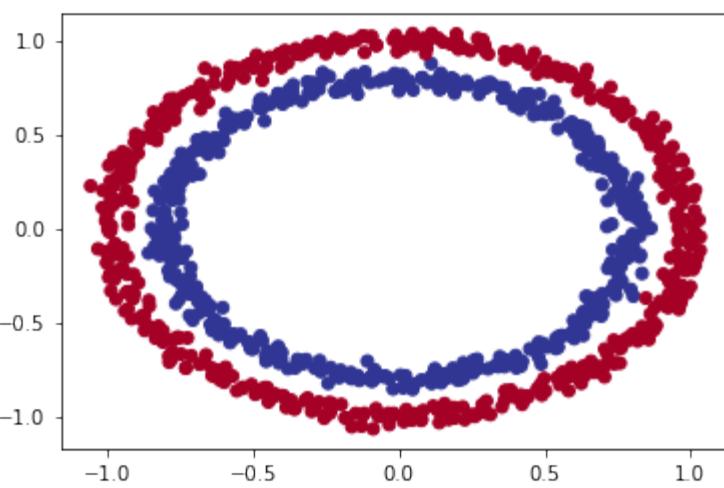
Out[4]:

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

500 each, nice and balanced.

Let's plot them.

In [5]:



Alrighty, looks like we've got a problem to solve.

Let's find out how we could build a PyTorch neural network to classify dots into red (0) or blue (1).

**Note:** This dataset is often what's considered a **toy problem** (a problem that's used to try and test things out on) in machine learning.

But it represents the major key of classification, you have some kind of data represented as numerical values and

you'd like to build a model that's able to classify it, in our case, separate it into red or blue dots.

## 1.1 Input and output shapes

One of the most common errors in deep learning is shape errors.

Mismatching the shapes of tensors and tensor operations will result in errors in your models.

We're going to see plenty of these throughout the course.

And there's no surefire way to make sure they won't happen, they will.

What you can do instead is continually familiarize yourself with the shape of the data you're working with.

I like referring to it as input and output shapes.

Ask yourself:

"What shapes are my inputs and what shapes are my outputs?"

Let's find out.

In [6]:

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

Out[6]:

Looks like we've got a match on the first dimension of each.

There's 1000  $x$  and 1000  $y$ .

But what's the second dimension on  $x$ ?

It often helps to view the values and shapes of a single sample (features and labels).

Doing so will help you understand what input and output shapes you'd be expecting from your model.

In [7]:

```
Values for one sample of X: [0.75424625 0.23148074] and the same for y: 1
Shapes for one sample of X: (2,) and the same for y: ()
```

This tells us the second dimension for `x` means it has two features (vector) where as `y` has a single feature (scalar).

We have two inputs for one output.

## 1.2 Turn data into tensors and create train and test splits

We've investigated the input and output shapes of our data, now let's prepare it for being used with PyTorch and for modelling.

Specifically, we'll need to:

1. Turn our data into tensors (right now our data is in NumPy arrays and PyTorch prefers to work with PyTorch tensors).
2. Split our data into training and test sets (we'll train a model on the training set to learn the patterns between  $x$  and  $y$  and then evaluate those learned patterns on the test dataset).

In [8]:

Out[8]:

```
(tensor([[ 0.7542,  0.2315],
       [-0.7562,  0.1533],
       [-0.8154,  0.1733],
       [-0.3937,  0.6929],
       [ 0.4422, -0.8967]]),
 tensor([1., 1., 1., 1., 0.])))
```

Now our data is in tensor format, let's split it into training and test sets.

To do so, let's use the helpful function `train_test_split()` from Scikit-Learn.

We'll use `test_size=0.2` (80% training, 20% testing) and because the split happens randomly across the data, let's use `random_state=42` so the split is reproducible.

In [9]:









Out[9]:

(800, 200, 800, 200)

Nice! Looks like we've now got 800 training samples and 200 testing samples.

## 2. Building a model

We've got some data ready, now it's time to build a model.

We'll break it down into a few parts.

1. Setting up device agnostic code (so our model can run on CPU or GPU if it's available).
2. Constructing a model by subclassing `nn.Module`.
3. Defining a loss function and optimizer.
4. Creating a training loop (this'll be in the next section).

The good news is we've been through all of the above steps before in notebook 01.

Except now we'll be adjusting them so they work with a classification dataset.

Let's start by importing PyTorch and `torch.nn` as well as setting up device agnostic code.

In [10]:

Out[10]:

```
'cuda'
```

Excellent, now `device` is setup, we can use it for any data or models we create and PyTorch will handle it on the CPU (default) or GPU if it's available.

How about we create a model?

We'll want a model capable of handling our `x` data as inputs and producing something in the shape of our `y` data as outputs.

In other words, given `x` (features) we want our model to predict `y` (label).

This setup where you have features and labels is referred to as **supervised learning**. Because your data is telling your model what the outputs should be given a certain input.

To create such a model it'll need to handle the input and output shapes of `x` and `y`.

Remember how I said input and output shapes are important? Here we'll see why.

Let's create a model class that:

1. Subclasses `nn.Module` (almost all PyTorch models are subclasses of `nn.Module`).
2. Creates 2 `nn.Linear` layers in the constructor capable of handling the input and output shapes of `x` and `y`.
3. Defines a `forward()` method containing the forward pass computation of the model.
4. Instantiates the model class and sends it to the target `device`.

In [11]:









Out[11]:

```
CircleModelV0(  
    layer_1): Linear(in_features=2, out_features=5, bias=True)  
    layer_2): Linear(in_features=5, out_features=1, bias=True)  
)
```

What's going on here?

We've seen a few of these steps before.

The only major change is what's happening between `self.layer_1` and `self.layer_2`.

`self.layer_1` takes 2 input features `in_features=2` and produces 5 output features `out_features=5`.

This is known as having 5 **hidden units or neurons**.

This layer turns the input data from having 2 features to 5 features.

Why do this?

This allows the model to learn patterns from 5 numbers rather than just 2 numbers, *potentially* leading to better outputs.

I say potentially because sometimes it doesn't work.

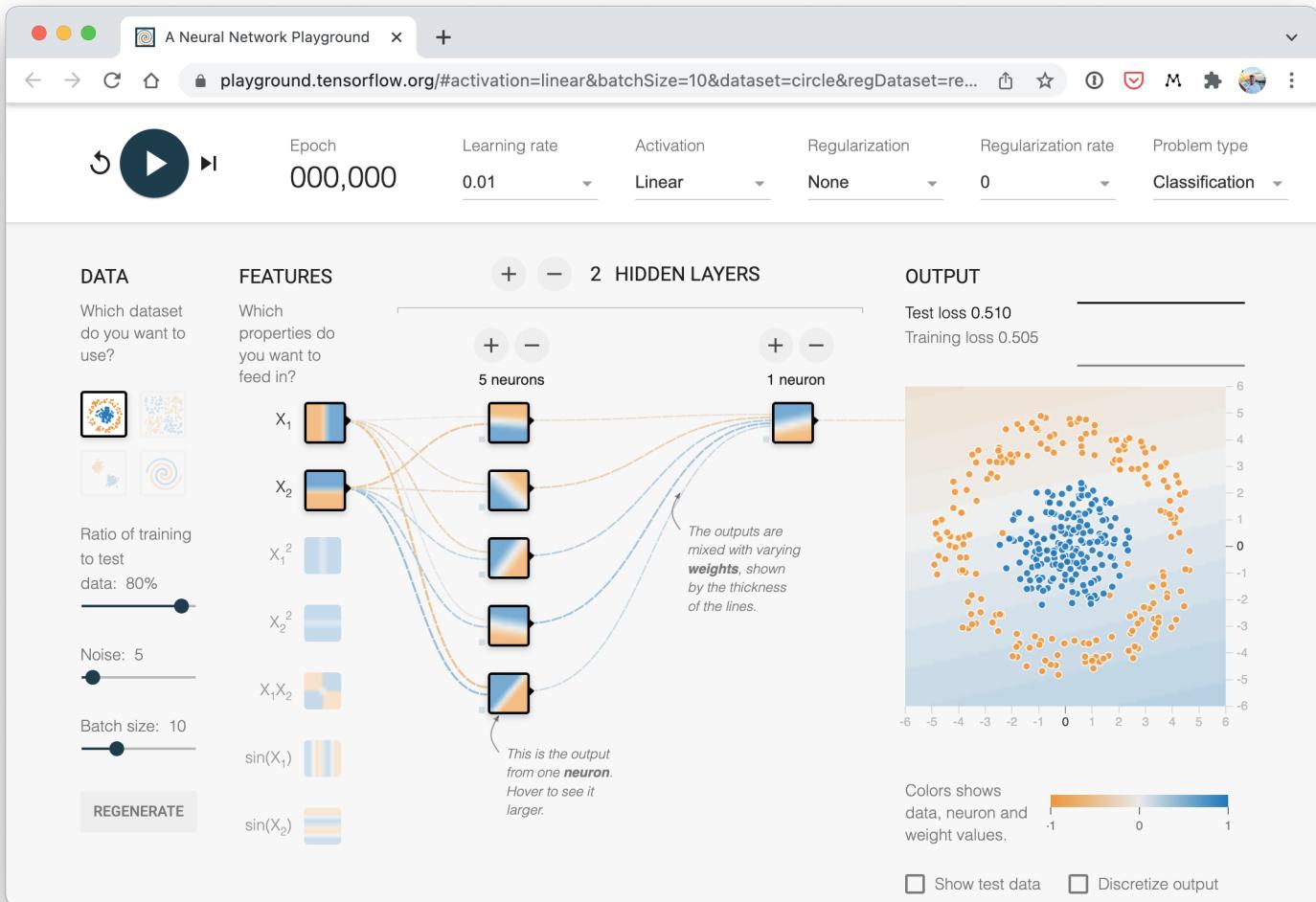
The number of hidden units you can use in neural network layers is a **hyperparameter** (a value you can set yourself) and there's no set in stone value you have to use.

Generally more is better but there's also such a thing as too much. The amount you choose will depend on your model type and dataset you're working with.

Since our dataset is small and simple, we'll keep it small.

The only rule with hidden units is that the next layer, in our case, `self.layer_2` has to take the same `in_features` as the previous layer `out_features`.

That's why `self.layer_2` has `in_features=5`, it takes the `out_features=5` from `self.layer_1` and performs a linear computation on them, turning them into `out_features=1` (the same shape as `y`).



A visual example of what a similar classification neural network to the one we've just built looks like. Try create one of your own on the [TensorFlow Playground website](#).

You can also do the same as above using `nn.Sequential`.

`nn.Sequential` performs a forward pass computation of the input data through the layers in the order they appear.

In [12]:

Out[12]:

```
Sequential(  
    (0): Linear(in_features=2, out_features=5, bias=True)  
    (1): Linear(in_features=5, out_features=1, bias=True)  
)
```

Woah, that looks much simpler than subclassing `nn.Module`, why not just always use `nn.Sequential`?

`nn.Sequential` is fantastic for straight-forward computations, however, as the namespace says, it *always* runs in sequential order.

So if you'd something else to happen (rather than just straight-forward sequential computation) you'll want to define your own custom `nn.Module` subclass.

Now we've got a model, let's see what happens when we pass some data through it.

In [13]:

```
Length of predictions: 200, Shape: torch.Size([200, 1])
Length of test samples: 200, Shape: torch.Size([200])
```

```
First 10 predictions:
tensor([[-0.2720],
```

```

[-0.3588],
[-0.0415],
[-0.3373],
[-0.0371],
[-0.0901],
[-0.3382],
[-0.2752],
[-0.0420],
[-0.3646]], device='cuda:0', grad_fn=<SliceBackward0>

```

First 10 test labels:

```
tensor([1., 0., 1., 0., 1., 0., 0., 1., 0.])
```

Hmm, it seems there's the same amount of predictions as there is test labels but the predictions don't look like they're in the same form or shape as the test labels.

We've got a couple steps we can do to fix this, we'll see these later on.

## 2.1 Setup loss function and optimizer

We've setup a loss (also called a criterion or cost function) and optimizer before in [notebook 01](#).

But different problem types require different loss functions.

For example, for a regression problem (predicting a number) you might used mean absolute error (MAE) loss.

And for a binary classification problem (like ours), you'll often use [binary cross entropy](#) as the loss function.

However, the same optimizer function can often be used across different problem spaces.

For example, the stochastic gradient descent optimizer ([SGD](#), `torch.optim.SGD()`) can be used for a range of problems, so can too the Adam optimizer (`torch.optim.Adam()`).

Loss function/Optimizer	Problem type	PyTorch Code
Stochastic Gradient Descent (SGD) optimizer	Classification, regression, many others.	<code>torch.optim.SGD()</code>
Adam Optimizer	Classification, regression, many others.	<code>torch.optim.Adam()</code>
Binary cross entropy loss	Binary classification	<code>torch.nn.BCELossWithLogits</code> or <code>torch.nn.BCELoss</code>
Cross entropy loss	Mutli-class classification	<code>torch.nn.CrossEntropyLoss</code>

Mean absolute error (MAE) or L1 Loss	Regression	<code>torch.nn.L1Loss</code>
Mean squared error (MSE) or L2 Loss	Regression	<code>torch.nn.MSELoss</code>

*Table of various loss functions and optimizers, there are more but these some common ones you'll see.*

Since we're working with a binary classification problem, let's use a binary cross entropy loss function.

**Note:** Recall a **loss function** is what measures how *wrong* your model predictions are, the higher the loss, the worse your model.

Also, PyTorch documentation often refers to loss functions as "loss criterion" or "criterion", these are all different ways of describing the same thing.

PyTorch has two binary cross entropy implementations:

1. `torch.nn.BCELoss()` - Creates a loss function that measures the binary cross entropy between the target (label) and input (features).
2. `torch.nn.BCEWithLogitsLoss()` - This is the same as above except it has a sigmoid layer (`nn.Sigmoid`) built-in (we'll see what this means soon).

Which one should you use?

The documentation for `torch.nn.BCEWithLogitsLoss()` states that it's more numerically stable than using `torch.nn.BCELoss()` after a `nn.Sigmoid` layer.

So generally, implementation 2 is a better option. However for advanced usage, you may want to separate the combination of `nn.Sigmoid` and `torch.nn.BCELoss()` but that is beyond the scope of this notebook.

Knowing this, let's create a loss function and an optimizer.

For the optimizer we'll use `torch.optim.SGD()` to optimize the model parameters with learning rate 0.1.

**Note:** There's a [discussion on the PyTorch forums about the use of `nn.BCELoss` vs. `nn.BCEWithLogitsLoss`](#). It can be confusing at first but as with many things, it becomes easier with practice.

In [14]:



Now let's also create an **evaluation metric**.

An evaluation metric can be used to offer another perspective on how your model is going.

If a loss function measures how *wrong* your model is, I like to think of evaluation metrics as measuring how *right* it is.

Of course, you could argue both of these are doing the same thing but evaluation metrics offer a different perspective.

After all, when evaluating your models it's good to look at things from multiple points of view.

There are several evaluation metrics that can be used for classification problems but let's start out with **accuracy**.

Accuracy can be measured by dividing the total number of correct predictions over the total number of predictions.

For example, a model that makes 99 correct predictions out of 100 will have an accuracy of 99%.

Let's write a function to do so.

In [15]:

Excellent! We can now use this function whilst training our model to measure it's performance alongside the loss.

### 3. Train model

Okay, now we've got a loss function and optimizer ready to go, let's train a model.

Do you remember the steps in a PyTorch training loop?

If not, here's a reminder.

Steps in training:

**PyTorch training loop steps**

#### 3.1 Going from raw model outputs to predicted labels (logits -> prediction probabilities -> prediction labels)

Before we the training loop steps, let's see what comes out of our model during the forward pass (the forward pass is defined by the `forward()` method).

To do so, let's pass the model some data.

In [16]:

Out[16]:

```
tensor([[-0.2720],  
       [-0.3588],  
       [-0.0415],  
       [-0.3373],  
       [-0.0371]], device='cuda:0', grad_fn=<SliceBackward0>)
```

Since our model hasn't been trained, these outputs are basically random.

But *what* are they?

They're the output of our `forward()` method.

Which implements two layers of `nn.Linear()` which internally calls the following equation:

$$\mathbf{y} = \mathbf{x} \cdot \mathbf{Weights}^T + \mathbf{bias}$$

The *raw outputs* (unmodified) of this equation ( $\mathbf{y}$ ) and in turn, the raw outputs of our model are often referred to as **logits**.

That's what our model is outputting above when it takes in the input data ( $\mathbf{x}$  in the equation or `x_test` in the code), logits.

However, these numbers are hard to interpret.

We'd like some numbers that are comparable to our truth labels.

To get our model's raw outputs (logits) into such a form, we can use the [sigmoid activation function](#).

Let's try it out.

In [17]:

```
tensor([[0.4324],
       [0.4113],
       [0.4896],
       [0.4165],
       [0.4907]], device='cuda:0', grad_fn=<SigmoidBackward0>)
```

Okay, it seems like the outputs now have some kind of consistency (even though they're still random).

Out[17]:

They're now in the form of **prediction probabilities** (I usually refer to these as `y_pred_probs`), in other words, the values are now how much the model thinks the data point belongs to one class or another.

In our case, since we're dealing with binary classification, our ideal outputs are 0 or 1.

So these values can be viewed as a decision boundary.

The closer to 0, the more the model thinks the sample belongs to class 0, the closer to 1, the more the model thinks the sample belongs to class 1.

More specifically:

- If  $y_{pred\_probs} \geq 0.5$ ,  $y=1$  (class 1)
- If  $y_{pred\_probs} < 0.5$ ,  $y=0$  (class 0)

To turn our prediction probabilities in prediction labels, we can round the outputs of the sigmoid activation function.

In [18]:

```
tensor([True, True, True, True, True], device='cuda:0')
```

Out[18]:

```
tensor([0., 0., 0., 0., 0.], device='cuda:0', grad_fn=<SqueezeBackward0>)
```

Excellent! Now it looks like our model's predictions are in the same form as our truth labels (`y_test`).

In [19]:

```
tensor([1., 0., 1., 0., 1.])
```

Out[19]:

This means we'll be able to compare our models predictions to the test labels to see how well it's going.

To recap, we converted our model's raw outputs (logits) to prediction probabilities using a sigmoid activation

function.

And then converted the prediction probabilities to prediction labels by rounding them.

**Note:** The use of the sigmoid activation function is often only for binary classification logits. For multi-class classification, we'll be looking at using the [softmax activation function](#) (this will come later on).

And the use of the sigmoid activation function is not required when passing our model's raw outputs to the `nn.BCEWithLogitsLoss` (the "logits" in logits loss is because it works on the model's raw logits output), this is because it has a sigmoid function built-in.

## 3.2 Building a training and testing loop

Alright, we've discussed how to take our raw model outputs and convert them to prediction labels, now let's build a training loop.

Let's start by training for 100 epochs and outputting the model's progress every 10 epochs.

In [20]:























```
Epoch: 0 | Loss: 0.69989, Accuracy: 44.25% | Test loss: 0.69391, Test acc: 48.50%
Epoch: 10 | Loss: 0.69678, Accuracy: 47.25% | Test loss: 0.69181, Test acc: 52.00%
Epoch: 20 | Loss: 0.69560, Accuracy: 48.25% | Test loss: 0.69138, Test acc: 51.00%
Epoch: 30 | Loss: 0.69502, Accuracy: 48.88% | Test loss: 0.69141, Test acc: 50.50%
Epoch: 40 | Loss: 0.69465, Accuracy: 49.00% | Test loss: 0.69158, Test acc: 53.00%
Epoch: 50 | Loss: 0.69438, Accuracy: 49.25% | Test loss: 0.69177, Test acc: 52.50%
Epoch: 60 | Loss: 0.69417, Accuracy: 49.12% | Test loss: 0.69196, Test acc: 52.50%
Epoch: 70 | Loss: 0.69400, Accuracy: 49.12% | Test loss: 0.69214, Test acc: 51.00%
Epoch: 80 | Loss: 0.69386, Accuracy: 49.62% | Test loss: 0.69231, Test acc: 48.50%
Epoch: 90 | Loss: 0.69374, Accuracy: 49.50% | Test loss: 0.69247, Test acc: 50.00%
```

Hmm, what do you notice about the performance of our model?

It looks like it went through the training and testing steps fine but the results don't seem to have moved too much.

The accuracy barely moves above 50% on each data split.

And because we're working with a balanced binary classification problem, it means our model is performing as good as random guessing (with 500 samples of class 0 and class 1 a model predicting class 1 every single time would achieve 50% accuracy).

## 4. Make predictions and evaluate the model

From the metrics it looks like our model is random guessing.

How could we investigate this further?

I've got an idea.

The data explorer's motto!

"Visualize, visualize, visualize!"

Let's make a plot of our model's predictions, the data it's trying to predict on and the decision boundary it's creating for whether something is class 0 or class 1.

To do so, we'll write some code to download and import the `helper_functions.py` script from the [Learn PyTorch for Deep Learning repo](#).

It contains a helpful function called `plot_decision_boundary()` which creates a NumPy meshgrid to visually plot the different points where our model is predicting certain classes.

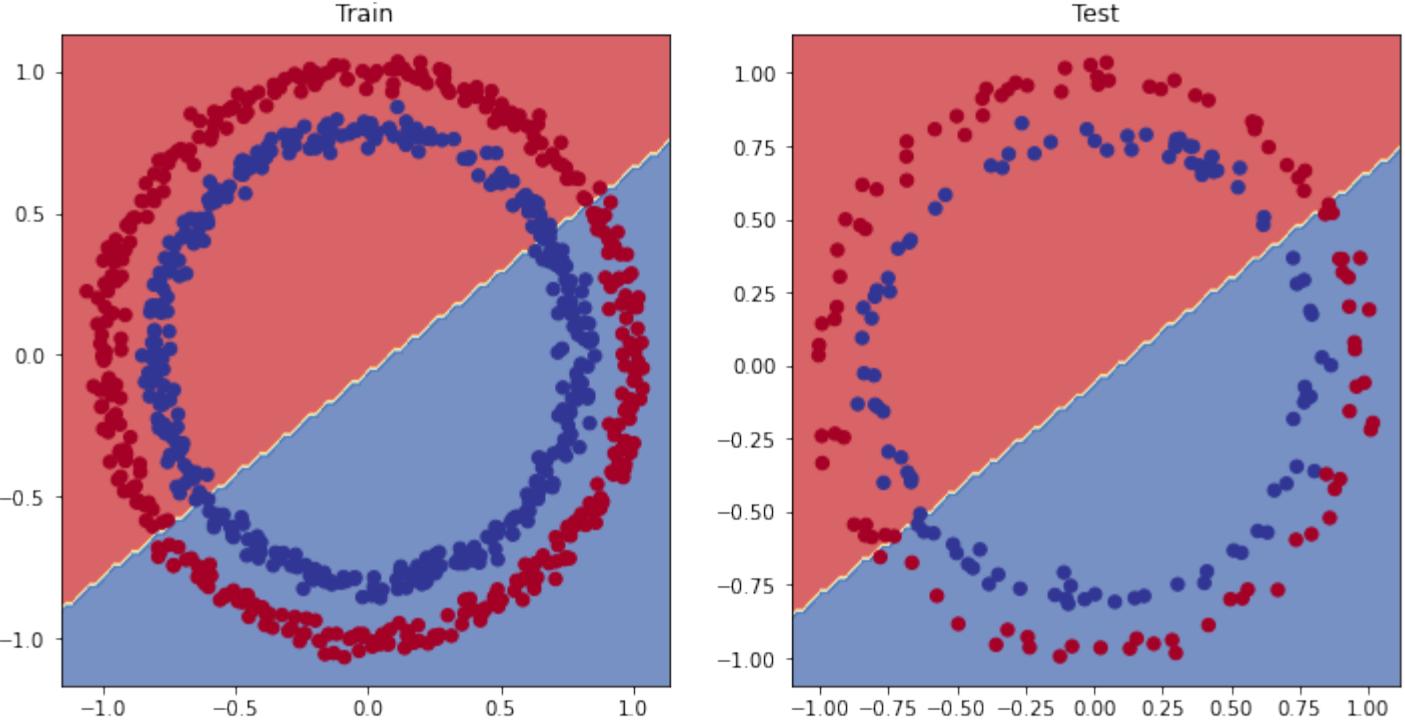
We'll also import `plot_predictions()` which we wrote in notebook 01 to use later.

In [21]:



```
helper_functions.py already exists, skipping download
```

In [22]:



Oh wow, it seems like we've found the cause of model's performance issue.

It's currently trying to split the red and blue dots using a straight line...

That explains the 50% accuracy. Since our data is circular, drawing a straight line can at best cut it down the middle.

In machine learning terms, our model is **underfitting**, meaning it's not learning predictive patterns from the data.

How could we improve this?

## 5. Improving a model (from a model perspective)

Let's try to fix our model's underfitting problem.

Focusing specifically on the model (not the data), there are a few ways we could do this.

Model improvement technique*	What does it do?
<b>Add more layers</b>	Each layer <i>potentially</i> increases the learning capabilities of the model with each layer being able to learn some kind of new pattern in the data, more layers is often referred to as making your neural network <i>deeper</i> .
<b>Add more hidden units</b>	Similar to the above, more hidden units per layer means a <i>potential</i> increase in learning capabilities of the model, more hidden units is often referred to as making your neural network <i>wider</i> .
<b>Fitting for longer (more epochs)</b>	Your model might learn more if it had more opportunities to look at the data.
<b>Changing the activation functions</b>	Some data just can't be fit with only straight lines (like what we've seen), using non-linear activation functions can help with this (hint, hint).
<b>Change the learning rate</b>	Less model specific, but still related, the learning rate of the optimizer decides how much a model should change its parameters each step, too much and the model overcorrects, too little and it doesn't learn enough.
<b>Change the loss function</b>	Again, less model specific but still important, different problems require different loss functions. For example, a binary cross entropy loss function won't work with a multi-class classification problem.
<b>Use transfer learning</b>	Take a pretrained model from a problem domain similar to yours and adjust it to your own problem. We cover transfer learning in <a href="#">notebook 06</a> .

**Note:** \*because you can adjust all of these by hand, they're referred to as **hyperparameters**.

And this is also where machine learning's half art half science comes in, there's no real way to know here what the best combination of values is for your project, best to follow the data scientist's motto of "experiment, experiment, experiment".

Let's see what happens if we add an extra layer to our model, fit for longer (`epochs=1000` instead of `epochs=100`) and increase the number of hidden units from `5` to `10`.

We'll follow the same steps we did above but with a few changed hyperparameters.

In [23]:









Out[23]:

```
CircleModelV1(  
    layer_1): Linear(in_features=2, out_features=10, bias=True)  
    layer_2): Linear(in_features=10, out_features=10, bias=True)  
    layer_3): Linear(in_features=10, out_features=1, bias=True)  
)
```

Now we've got a model, we'll recreate a loss function and optimizer instance, using the same settings as before.

In [24]:

Beautiful, model, optimizer and loss function ready, let's make a training loop.

This time we'll train for longer (`epochs=1000` vs `epochs=100`) and see if it improves our model.

In [25]:















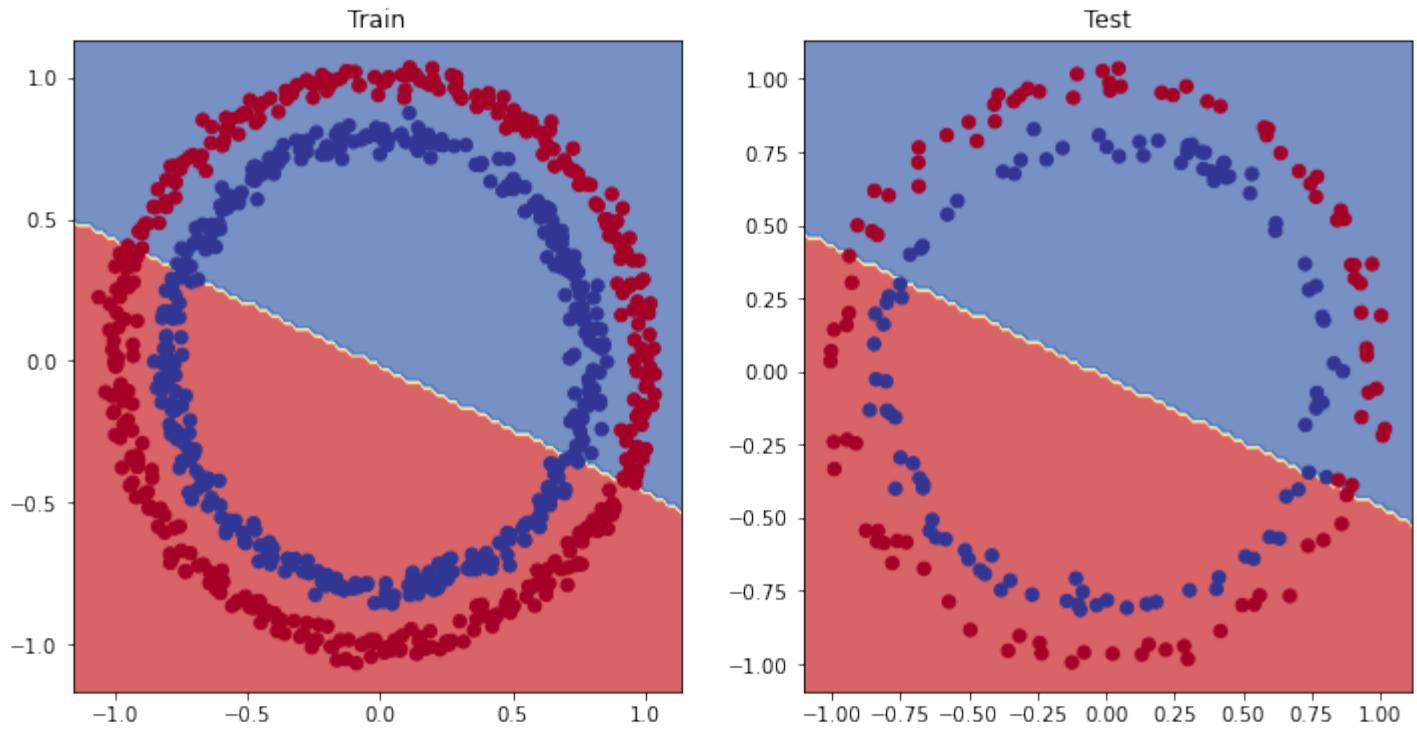


```
Epoch: 0 | Loss: 0.69396, Accuracy: 50.88% | Test loss: 0.69261, Test acc: 51.00%
Epoch: 100 | Loss: 0.69305, Accuracy: 50.38% | Test loss: 0.69379, Test acc: 48.00%
Epoch: 200 | Loss: 0.69299, Accuracy: 51.12% | Test loss: 0.69437, Test acc: 46.00%
Epoch: 300 | Loss: 0.69298, Accuracy: 51.62% | Test loss: 0.69458, Test acc: 45.00%
Epoch: 400 | Loss: 0.69298, Accuracy: 51.12% | Test loss: 0.69465, Test acc: 46.00%
Epoch: 500 | Loss: 0.69298, Accuracy: 51.00% | Test loss: 0.69467, Test acc: 46.00%
Epoch: 600 | Loss: 0.69298, Accuracy: 51.00% | Test loss: 0.69468, Test acc: 46.00%
Epoch: 700 | Loss: 0.69298, Accuracy: 51.00% | Test loss: 0.69468, Test acc: 46.00%
Epoch: 800 | Loss: 0.69298, Accuracy: 51.00% | Test loss: 0.69468, Test acc: 46.00%
Epoch: 900 | Loss: 0.69298, Accuracy: 51.00% | Test loss: 0.69468, Test acc: 46.00%
```

What? Our model trained for longer and with an extra layer but it still looks like it didn't learn any patterns better than random guessing.

Let's visualize.

In [26]:



Hmmm.

Our model is still drawing a straight line between the red and blue dots.

If our model is drawing a straight line, could it model linear data? Like we did in [notebook 01](#)?

## 5.1 Preparing data to see if our model can model a straight line

Let's create some linear data to see if our model's able to model it and we're not just using a model that can't learn anything.

In [27]:



100

Out[27]:

```
(tensor([[0.0000],
       [0.0100],
       [0.0200],
       [0.0300],
       [0.0400]]),
 tensor([[0.3000],
       [0.3070],
       [0.3140],
       [0.3210],
       [0.3280]]))
```

Wonderful, now let's split our data into training and test sets.

In [28]:



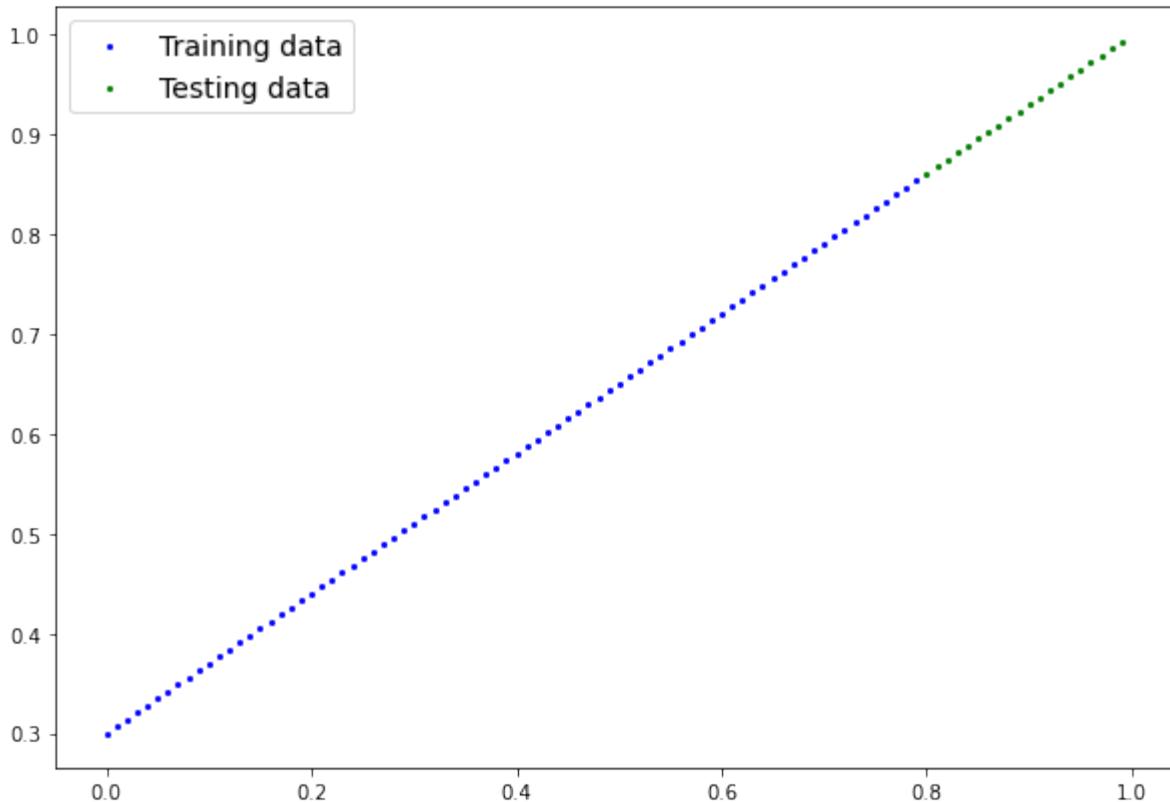
```
80 80 20 20
```

Beautiful, let's see how the data looks.

To do so, we'll use the `plot_predictions()` function we created in notebook 01.

It's contained within the `helper_functions.py` script on the Learn PyTorch for Deep Learning repo which we downloaded above.

In [29]:



## 5.2 Adjusting `model_1` to fit a straight line

Now we've got some data, let's recreate `model_1` but with a loss function suited to our regression data.

In [30]:

Out[30]:

```
Sequential(  
    (0): Linear(in_features=1, out_features=10, bias=True)  
    (1): Linear(in_features=10, out_features=10, bias=True)  
    (2): Linear(in_features=10, out_features=1, bias=True)  
)
```

We'll setup the loss function to be `nn.L1Loss()` (the same as mean absolute error) and the optimizer to be

```
torch.optim.SGD().
```

In [31]:

Now let's train the model using the regular training loop steps for `epochs=1000` (just like `model_1`).

**Note:** We've been writing similar training loop code over and over again. I've made it that way on purpose though, to keep practicing. However, do you have ideas how we could functionize this? That would save a fair bit of coding in the future. Potentially there could be a function for training and a function for testing.

In [32]:













```
Epoch: 0 | Train loss: 0.75986, Test loss: 0.54143
Epoch: 100 | Train loss: 0.09309, Test loss: 0.02901
Epoch: 200 | Train loss: 0.07376, Test loss: 0.02850
Epoch: 300 | Train loss: 0.06745, Test loss: 0.00615
Epoch: 400 | Train loss: 0.06107, Test loss: 0.02004
Epoch: 500 | Train loss: 0.05698, Test loss: 0.01061
Epoch: 600 | Train loss: 0.04857, Test loss: 0.01326
Epoch: 700 | Train loss: 0.06109, Test loss: 0.02127
Epoch: 800 | Train loss: 0.05599, Test loss: 0.01426
Epoch: 900 | Train loss: 0.05571, Test loss: 0.00603
```

Okay, unlike `model_1` on the classification data, it looks like `model_2`'s loss is actually going down.

Let's plot its predictions to see if that's so.

And remember, since our model and data are using the target `device`, and this device may be a GPU, however, our plotting function uses `matplotlib` and `matplotlib` can't handle data on the GPU.

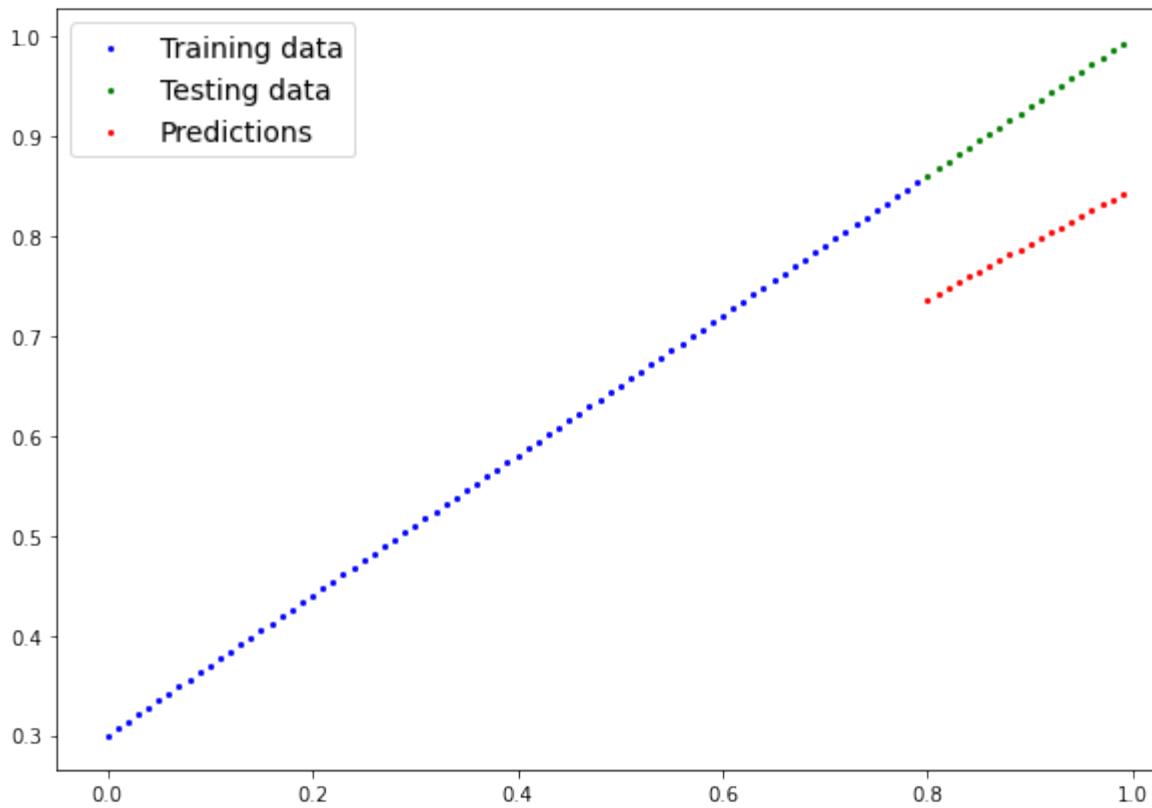
To handle that, we'll send all of our data to the CPU using `.cpu()` when we pass it to `plot_predictions()`.

In [33]:









Alright, it looks like our model is able to do far better than random guessing on straight lines.

This is a good thing.

It means our model at least has *some* capacity to learn.

**Note:** A helpful troubleshooting step when building deep learning models is to start as small as possible to see if the model works before scaling it up.

This could mean starting with a simple neural network (not many layers, not many hidden neurons) and a small dataset (like the one we've made) and then **overfitting** (making the model perform too well) on that small example before increasing the amount data or the model size/design to *reduce* overfitting.

So what could it be?

Let's find out.

## 6. The missing piece: non-linearity

We've seen our model can draw straight (linear) lines, thanks to its linear layers.

But how about we give it the capacity to draw non-straight (non-linear) lines?

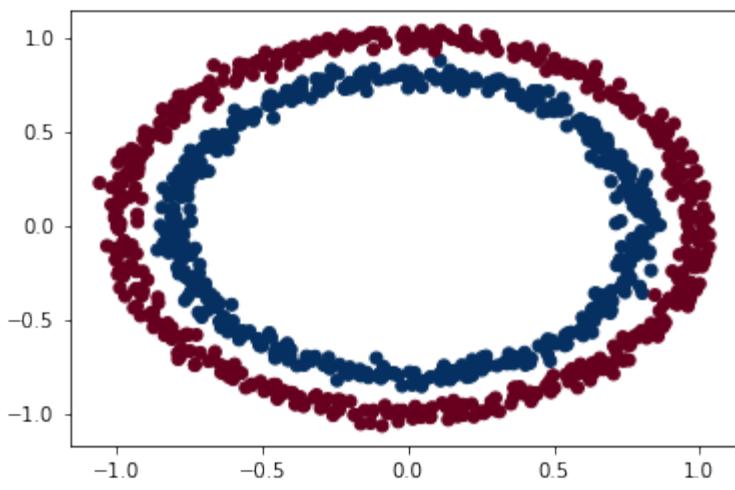
How?

Let's find out.

## 6.1 Recreating non-linear data (red and blue circles)

First, let's recreate the data to start off fresh. We'll use the same setup as before.

In [34]:



Nice! Now let's split it into training and test sets using 80% of the data for training and 20% for testing.

In [35]:









Out[35]:

```
(tensor([[ 0.6579, -0.4651],
       [ 0.6319, -0.7347],
       [-1.0086, -0.1240],
       [-0.9666, -0.2256],
       [-0.1666,  0.7994]]),
 tensor([1., 0., 0., 0., 1.])))
```

## 6.2 Building a model with non-linearity

Now here comes the fun part.

What kind of pattern do you think you could draw with unlimited straight (linear) and non-straight (non-linear) lines?

I bet you could get pretty creative.

So far our neural networks have only been using linear (straight) line functions.

But the data we've been working with is non-linear (circles).

What do you think will happen when we introduce the capability for our model to use **non-linear activation functions?**

Well let's see.

PyTorch has a bunch of [ready-made non-linear activation functions](#) that do similiar but different things.

One of the most common and best performing is [ReLU](#) (rectified linear-unit, `torch.nn.ReLU()` ).

Rather than talk about it, let's put it in our neural network between the hidden layers in the forward pass and see what happens.

In [36]:

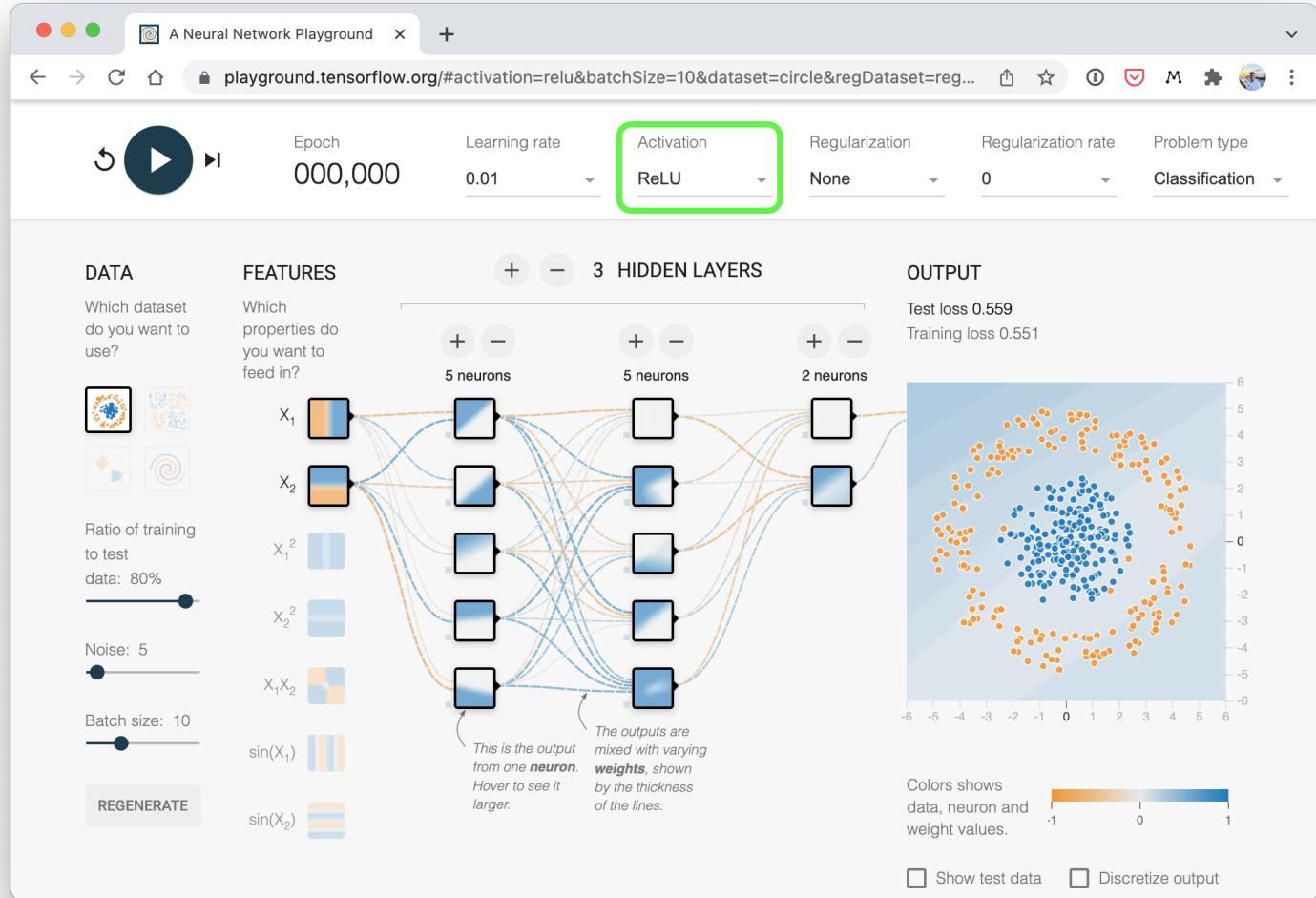








```
CircleModelV2 (
    (layer_1): Linear(in_features=2, out_features=10, bias=True)
    (layer_2): Linear(in_features=10, out_features=10, bias=True)
    (layer_3): Linear(in_features=10, out_features=1, bias=True)
    (relu): ReLU()
)
```



A visual example of what a similar classification neural network to the one we've just built (using ReLU activation) looks like. Try create one of your own on the [TensorFlow Playground website](#).

**Question:** Where should I put the non-linear activation functions when constructing a neural network?

A rule of thumb is to put them in between hidden layers and just after the output layer, however, there is no set in stone option. As you learn more about neural networks and deep learning you'll find a bunch of different ways of putting things together. In the meantime, best to experiment, experiment, experiment.

Now we've got a model ready to go, let's create a binary classification loss function as well as an optimizer.

In [37]:

Wonderful!

### 6.3 Training a model with non-linearity

You know the drill, model, loss function, optimizer ready to go, let's create a training and testing loop.

In [38]:

















```
Epoch: 0 | Loss: 0.69295, Accuracy: 50.00% | Test Loss: 0.69319, Test Accuracy: 50.00%
Epoch: 100 | Loss: 0.69115, Accuracy: 52.88% | Test Loss: 0.69102, Test Accuracy: 52.50%
Epoch: 200 | Loss: 0.68977, Accuracy: 53.37% | Test Loss: 0.68940, Test Accuracy: 55.00%
Epoch: 300 | Loss: 0.68795, Accuracy: 53.00% | Test Loss: 0.68723, Test Accuracy: 56.00%
Epoch: 400 | Loss: 0.68517, Accuracy: 52.75% | Test Loss: 0.68411, Test Accuracy: 56.50%
Epoch: 500 | Loss: 0.68102, Accuracy: 52.75% | Test Loss: 0.67941, Test Accuracy: 56.50%
Epoch: 600 | Loss: 0.67515, Accuracy: 54.50% | Test Loss: 0.67285, Test Accuracy: 56.00%
Epoch: 700 | Loss: 0.66659, Accuracy: 58.38% | Test Loss: 0.66322, Test Accuracy: 59.00%
Epoch: 800 | Loss: 0.65160, Accuracy: 64.00% | Test Loss: 0.64757, Test Accuracy: 67.50%
Epoch: 900 | Loss: 0.62362, Accuracy: 74.00% | Test Loss: 0.62145, Test Accuracy: 79.00%
```

Ho ho! That's looking far better!

## 6.4 Evaluating a model trained with non-linear activation functions

Remember how our circle data is non-linear? Well, let's see how our models predictions look now the model's been trained with non-linear activation functions.

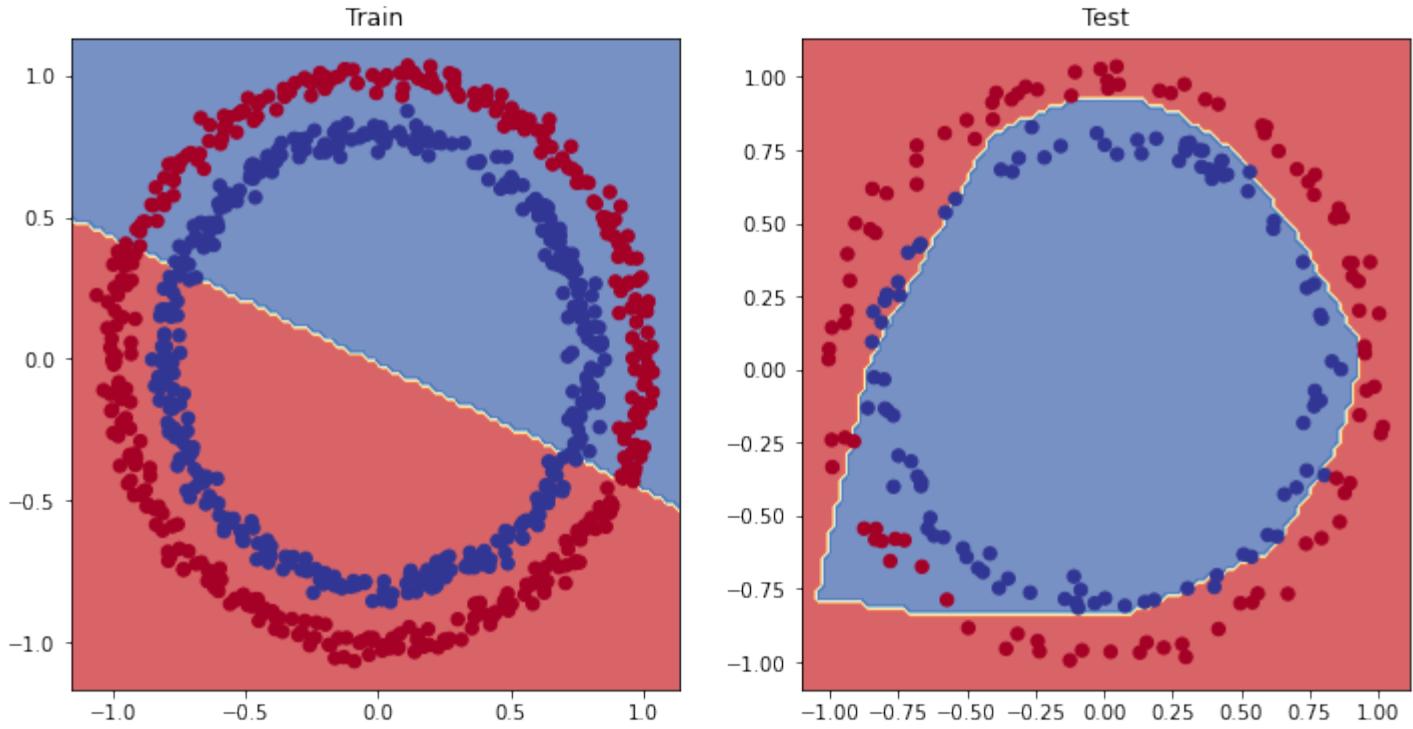
In [39]:

Out[39]:

```
(tensor([1., 0., 1., 0., 0., 1., 0., 0., 1., 0.], device='cuda:0'),  
 tensor([1., 1., 1., 1., 0., 1., 1., 1., 1., 0.]))
```

In [40]:





Nice! Not perfect but still far better than before.

Potentially you could try a few tricks to improve the test accuracy of the model? (hint: head back to section 5 for tips on improving the model)

## 7. Replicating non-linear activation functions

We saw before how adding non-linear activation functions to our model can help it to model non-linear data.

**Note:** Much of the data you'll encounter in the wild is non-linear (or a combination of linear and non-linear). Right now we've been working with dots on a 2D plot. But imagine if you had images of plants you'd like to classify, there's a lot of different plant shapes. Or text from Wikipedia you'd like to summarize, there's lots of different ways words can be put together (linear and non-linear patterns).

But what does a non-linear activation *look* like?

How about we replicate some and what they do?

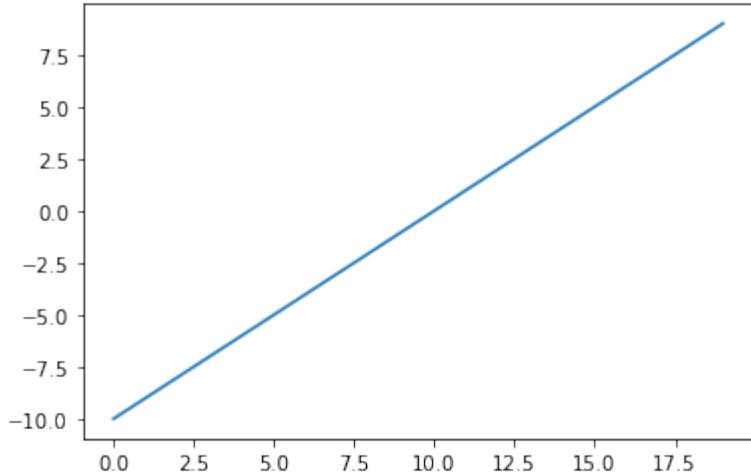
Let's start by creating a small amount of data.

In [41]:

```
tensor([-10., -9., -8., -7., -6., -5., -4., -3., -2., -1., 0., 1.,
       2., 3., 4., 5., 6., 7., 8., 9.])
```

Wonderful, now let's plot it.

In [42]:



A straight line, nice.

Now let's see how the ReLU activation function influences it.

And instead of using PyTorch's ReLU (`torch.nn.ReLU`), we'll recreate it ourselves.

The ReLU function turns all negatives to 0 and leaves the positive values as they are.

In [43]:

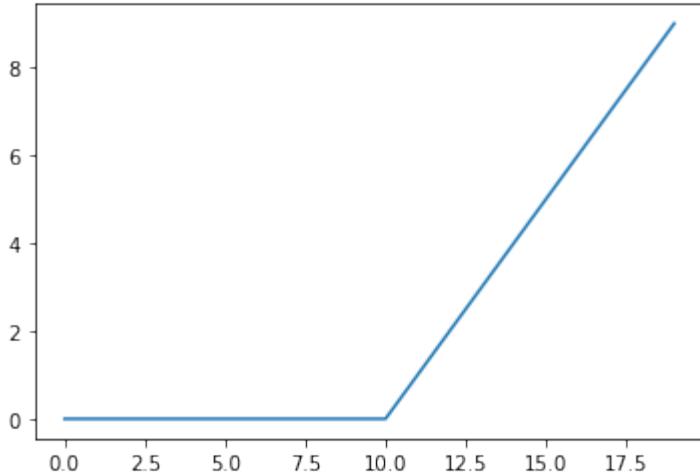
Out[43]:

```
tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 2., 3., 4., 5., 6., 7.,
       8., 9.])
```

It looks like our ReLU function worked, all of the negative values are zeros.

Let's plot them.

In [44]:



Nice! That looks exactly like the shape of the ReLU function on the [Wikipedia page for ReLU](#).

How about we try the **sigmoid function** we've been using?

The sigmoid function formula goes like so:

$$\text{out}_i = \frac{1}{1+e^{-\text{input}_i}}$$

Or using  $x$  as input:

$$S(x) = \frac{1}{1+e^{-x}}$$

Where  $S$  stands for sigmoid,  $e$  stands for [exponential](#) (`torch.exp()`) and  $i$  stands for a particular element in a tensor.

Let's build a function to replicate the sigmoid function with PyTorch.

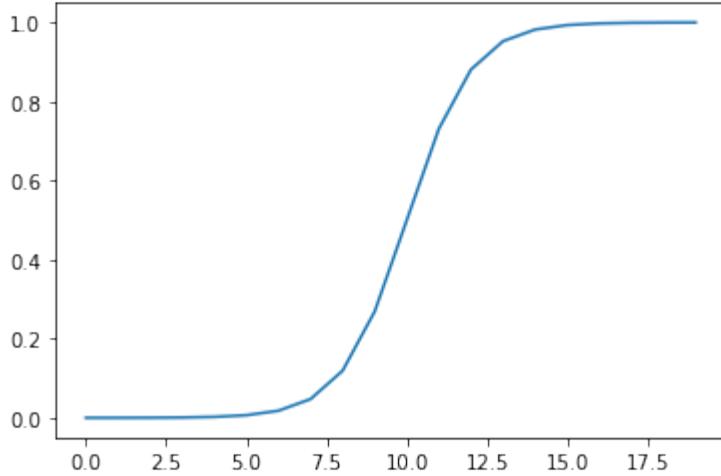
In [45]:

**Out[45]:**

```
tensor([4.5398e-05, 1.2339e-04, 3.3535e-04, 9.1105e-04, 2.4726e-03, 6.6929e-03,
       1.7986e-02, 4.7426e-02, 1.1920e-01, 2.6894e-01, 5.0000e-01, 7.3106e-01,
       8.8080e-01, 9.5257e-01, 9.8201e-01, 9.9331e-01, 9.9753e-01, 9.9909e-01,
       9.9966e-01, 9.9988e-01])
```

Woah, those values look a lot like prediction probabilities we've seen earlier, let's see what they look like visualized.

**In [46]:**



Looking good! We've gone from a straight line to a curved line.

Now there's plenty more [non-linear activation functions](#) that exist in PyTorch that we haven't tried.

But these two are two of the most common.

And the point remains, what patterns could you draw using an unlimited amount of linear (straight) and non-linear (not straight) lines?

Almost anything right?

That's exactly what our model is doing when we combine linear and non-linear functions.

Instead of telling our model what to do, we give it tools to figure out how to best discover patterns in the data.

And those tools are linear and non-linear functions.

## 8. Putting things together by building a multi-class PyTorch model

We've covered a fair bit.

But now let's put it all together using a multi-class classification problem.

Recall a **binary classification** problem deals with classifying something as one of two options (e.g. a photo as a cat photo or a dog photo) whereas a **multi-class classification** problem deals with classifying something from a list of *more than* two options (e.g. classifying a photo as a cat a dog or a chicken).



## Binary classification

(one thing or another)

## Multiclass classification

(more than one thing or another)

*Example of binary vs. multi-class classification. Binary deals with two classes (one thing or another), whereas multi-class classification can deal with any number of classes over two, for example, the popular [ImageNet-1k dataset](#) is used as a computer vision benchmark and has 1000 classes.*

## 8.1 Creating multi-class classification data

To begin a multi-class classification problem, let's create some multi-class data.

To do so, we can leverage Scikit-Learn's `make_blobs()` method.

This method will create however many classes (using the `centers` parameter) we want.

Specifically, let's do the following:

1. Create some multi-class data with `make_blobs()`.
2. Turn the data into tensors (the default of `make_blobs()` is to use NumPy arrays).
3. Split the data into training and test sets using `train_test_split()`.
4. Visualize the data.

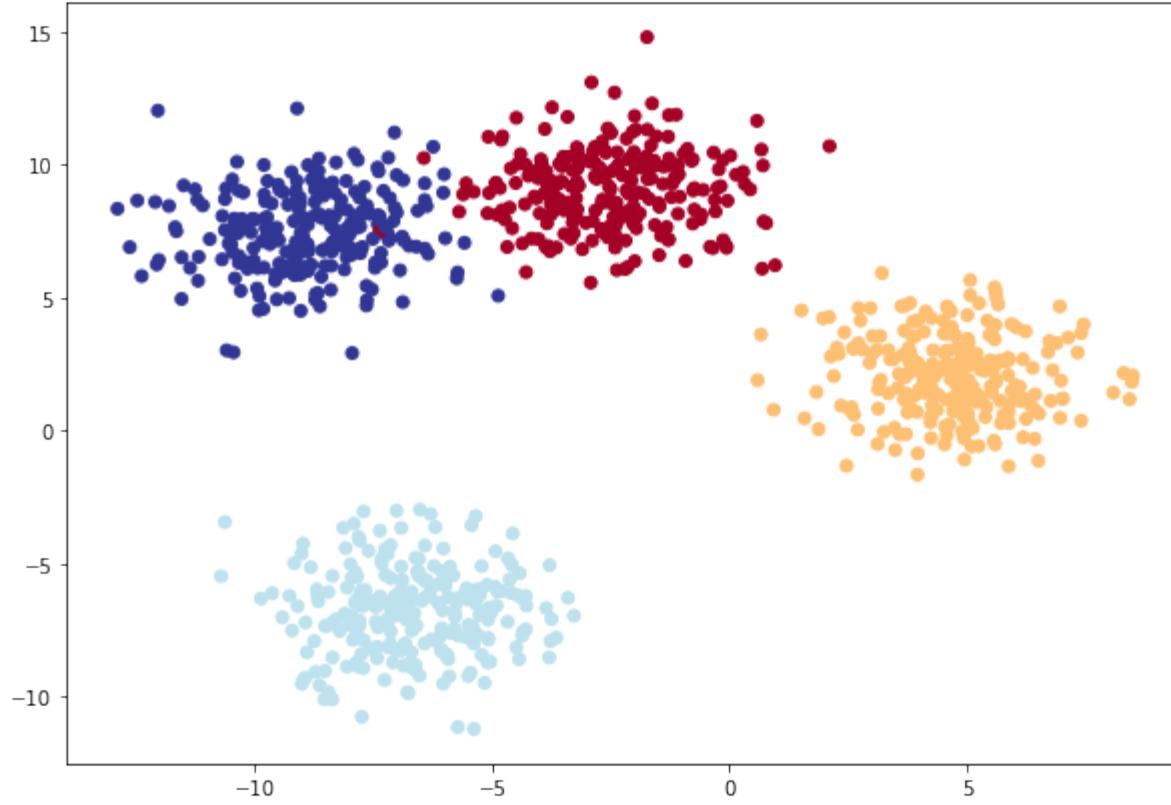
In [47]:







```
tensor([[-8.4134,   6.9352],  
       [-5.7665,  -6.4312],  
       [-6.0421,  -6.7661],  
       [ 3.9508,   0.6984],  
       [ 4.2505,  -0.2815]]) tensor([3, 2, 2, 1, 1])
```



Nice! Looks like we've got some multi-class data ready to go.

Let's build a model to separate the coloured blobs.

**Question:** Does this dataset need non-linearity? Or could you draw a succession of straight lines to separate it?

## 8.2 Building a multi-class classification model in PyTorch

We've created a few models in PyTorch so far.

You might also be starting to get an idea of how flexible neural networks are.

How about we build one similar to `model_3` but this still capable of handling multi-class data?

To do so, let's create a subclass of `nn.Module` that takes in three hyperparameters:

- `input_features` - the number of `x` features coming into the model.
- `output_features` - the ideal numbers of output features we'd like (this will be equivalent to `NUM_CLASSES` or the number of classes in your multi-class classification problem).
- `hidden_units` - the number of hidden neurons we'd like each hidden layer to use.

Since we're putting things together, let's setup some device agnostic code (we don't have to do this again in the same

notebook, it's only a reminder).

Then we'll create the model class using the hyperparameters above.

In [48]:

Out[48]:

```
'cuda'
```

In [49]:

















Out[49]:

```
BlobModel(  
    (linear_layer_stack): Sequential(  
        (0): Linear(in_features=2, out_features=8, bias=True)
```

```
(1): Linear(in_features=8, out_features=8, bias=True)
(2): Linear(in_features=8, out_features=4, bias=True)
)
)
```

Excellent! Our multi-class model is ready to go, let's create a loss function and optimizer for it.

## 8.3 Creating a loss function and optimizer for a multi-class PyTorch model

Since we're working on a multi-class classification problem, we'll use the `nn.CrossEntropyLoss()` method as our loss function.

And we'll stick with using SGD with a learning rate of 0.1 for optimizing our `model_4` parameters.

In [50]:

## 8.4 Getting prediction probabilities for a multi-class PyTorch model

Alright, we've got a loss function and optimizer ready, and we're ready to train our model but before we do let's do a single forward pass with our model to see if it works.

In [51]:

Out[51]:

```
tensor([[-1.2711, -0.6494, -1.4740, -0.7044],  
       [ 0.2210, -1.5439,  0.0420,  1.1531],  
       [ 2.8698,  0.9143,  3.3169,  1.4027],  
       [ 1.9576,  0.3125,  2.2244,  1.1324],  
       [ 0.5458, -1.2381,  0.4441,  1.1804]], device='cuda:0',  
grad_fn=<SliceBackward0>)
```

What's coming out here?

It looks like we get one value per feature of each sample.

Let's check the shape to confirm.

In [52]:

```
(torch.Size([4]), 4)
```

Wonderful, our model is predicting one value for each class that we have.

Do you remember what the raw outputs of our model are called?

Hint: it rhymes with "frog splits" (no animals were harmed in the creation of these materials).

If you guessed *logits*, you'd be correct.

So right now our model is outputting logits but what if we wanted to figure out exactly which label is was giving the sample?

Out[52]:

As in, how do we go from `logits -> prediction probabilities -> prediction labels` just like we did with the binary classification problem?

That's where the **softmax activation function** comes into play.

The softmax function calculates the probability of each prediction class being the actual predicted class compared to all other possible classes.

If this doesn't make sense, let's see in code.

In [53]:

```
tensor([[[-1.2549, -0.8112, -1.4795, -0.5696],
        [ 1.7168, -1.2270,  1.7367,  2.1010],
        [ 2.2400,  0.7714,  2.6020,  1.0107],
        [-0.7993, -0.3723, -0.9138, -0.5388],
        [-0.4332, -1.6117, -0.6891,  0.6852]], device='cuda:0',
       grad_fn=<SliceBackward0>)
tensor([[0.1872, 0.2918, 0.1495, 0.3715],
        [0.2824, 0.0149, 0.2881, 0.4147],
        [0.3380, 0.0778, 0.4854, 0.0989],
        [0.2118, 0.3246, 0.1889, 0.2748],
        [0.1945, 0.0598, 0.1506, 0.5951]], device='cuda:0',
       grad_fn=<SliceBackward0>)
```

Hmm, what's happened here?

It may still look like the outputs of the softmax function are jumbled numbers (and they are, since our model hasn't been trained and is predicting using random patterns) but there's a very specific thing different about each sample.

After passing the logits through the softmax function, each individual sample now adds to 1 (or very close to).

Let's check.

In [54]:

Out[54]:

```
tensor(1., device='cuda:0', grad_fn=<SumBackward0>)
```

These prediction probabilities are essentially saying how much the model *thinks* the target  $\times$  sample (the input) maps to each class.

Since there's one value for each class in `y_pred_probs`, the index of the *highest* value is the class the model thinks the specific data sample *most* belongs to.

We can check which index has the highest value using `torch.argmax()`.

In [55]:

```
tensor([0.1872, 0.2918, 0.1495, 0.3715], device='cuda:0',
      grad_fn=<SelectBackward0>)
tensor(3, device='cuda:0')
```

You can see the output of `torch.argmax()` returns 3, so for the features (`x`) of the sample at index 0, the model is predicting that the most likely class value (`y`) is 3.

Of course, right now this is just random guessing so it's got a 25% chance of being right (since there's four classes). But we can improve those chances by training the model.

**Note:** To summarize the above, a model's raw output is referred to as **logits**.

For a multi-class classification problem, to turn the logits into **prediction probabilities**, you use the softmax activation function (`torch.softmax`).

The index of the value with the highest **prediction probability** is the class number the model thinks is *most* likely given the input features for that sample (although this is a prediction, it doesn't mean it will be correct).

## 8.5 Creating a training and testing loop for a multi-class PyTorch model

Alright, now we've got all of the preparation steps out of the way, let's write a training and testing loop to improve and evaluate our model.

We've done many of these steps before so much of this will be practice.

The only difference is that we'll be adjusting the steps to turn the model outputs (logits) to prediction probabilities (using the softmax activation function) and then to prediction labels (by taking the argmax of the output of the softmax activation function).

Let's train the model for `epochs=100` and evaluate it every 10 epochs.

In [56]:



















```
Epoch: 0 | Loss: 1.04324, Acc: 65.50% | Test Loss: 0.57861, Test Acc: 95.50%
Epoch: 10 | Loss: 0.14398, Acc: 99.12% | Test Loss: 0.13037, Test Acc: 99.00%
Epoch: 20 | Loss: 0.08062, Acc: 99.12% | Test Loss: 0.07216, Test Acc: 99.50%
Epoch: 30 | Loss: 0.05924, Acc: 99.12% | Test Loss: 0.05133, Test Acc: 99.50%
Epoch: 40 | Loss: 0.04892, Acc: 99.00% | Test Loss: 0.04098, Test Acc: 99.50%
Epoch: 50 | Loss: 0.04295, Acc: 99.00% | Test Loss: 0.03486, Test Acc: 99.50%
Epoch: 60 | Loss: 0.03910, Acc: 99.00% | Test Loss: 0.03083, Test Acc: 99.50%
Epoch: 70 | Loss: 0.03643, Acc: 99.00% | Test Loss: 0.02799, Test Acc: 99.50%
Epoch: 80 | Loss: 0.03448, Acc: 99.00% | Test Loss: 0.02587, Test Acc: 99.50%
Epoch: 90 | Loss: 0.03300, Acc: 99.12% | Test Loss: 0.02423, Test Acc: 99.50%
```

## 8.6 Making and evaluating predictions with a PyTorch multi-class model

It looks like our trained model is performing pretty well.

But to make sure of this, let's make some predictions and visualize them.

In [57]:

Out[57]:

```
tensor([[ 4.3377,  10.3539, -14.8948, -9.7642],
       [ 5.0142, -12.0371,   3.3860, 10.6699],
       [-5.5885, -13.3448,  20.9894, 12.7711],
       [ 1.8400,   7.5599,  -8.6016, -6.9942],
       [ 8.0726,   3.2906, -14.5998, -3.6186],
       [ 5.5844, -14.9521,   5.0168, 13.2890],
       [-5.9739, -10.1913,  18.8655,  9.9179],
       [ 7.0755,  -0.7601,  -9.5531,  0.1736],
       [-5.5918, -18.5990,  25.5309, 17.5799],
       [ 7.3142,   0.7197, -11.2017, -1.2011]], device='cuda:0')
```

Alright, looks like our model's predictions are still in logit form.

Though to evaluate them, they'll have to be in the same form as our labels (`y_blob_test`) which are in integer form.

Let's convert our model's prediction logits to prediction probabilities (using `torch.softmax()`) then to prediction labels (by taking the `argmax()` of each sample).

**Note:** It's possible to skip the `torch.softmax()` function and go straight from predicted logits  $\rightarrow$  predicted labels by calling `torch.argmax()` directly on the logits.

For example, `y_preds = torch.argmax(y_logits, dim=1)`, this saves a computation step (no `torch.softmax()`) but results in no prediction probabilities being available to use.

In [58]:

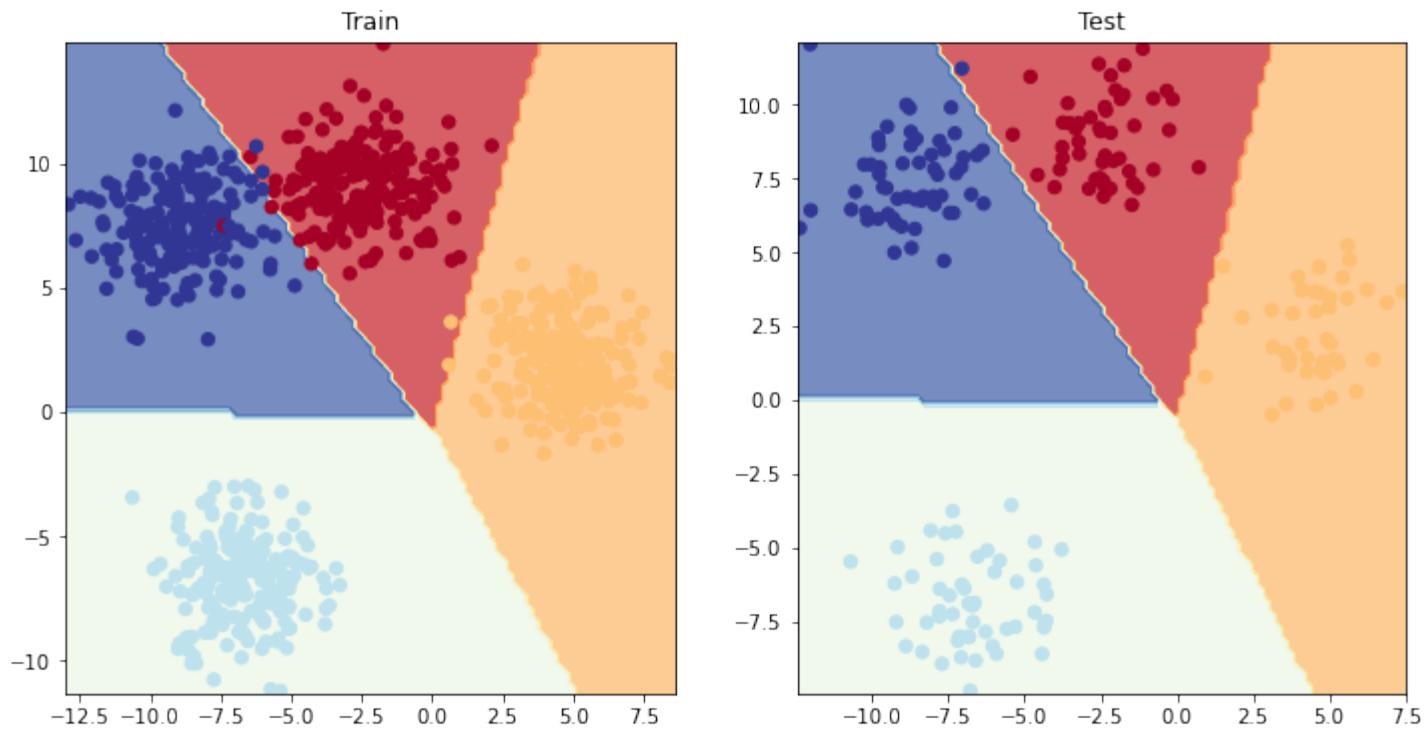


```
Predictions: tensor([1, 3, 2, 1, 0, 3, 2, 0, 2, 0], device='cuda:0')
Labels: tensor([1, 3, 2, 1, 0, 3, 2, 0, 2, 0], device='cuda:0')
Test accuracy: 99.5%
```

Nice! Our model predictions are now in the same form as our test labels.

Let's visualize them with `plot_decision_boundary()`, remember because our data is on the GPU, we'll have to move it to the CPU for use with `matplotlib` (`plot_decision_boundary()` does this automatically for us).

In [59]:



## 9. More classification evaluation metrics

So far we've only covered a couple of ways of evaluating a classification model (accuracy, loss and visualizing predictions).

These are some of the most common methods you'll come across and are a good starting point.

However, you may want to evaluate your classification model using more metrics such as the following:

Metric	Definition	Code
<b>name/Evaluation</b>		
<b>method</b>		
Accuracy	Out of 100 predictions, how many does your model get correct? E.g. 95% accuracy means it gets 95/100 predictions correct.	<code>torchmetrics.Accuracy()</code> or <code>sklearn.metrics.accuracy_score()</code>
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).	<code>torchmetrics.Precision()</code> or <code>sklearn.metrics.precision_score()</code>
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.	<code>torchmetrics.Recall()</code> or <code>sklearn.metrics.recall_score()</code>

F1-score	Combines precision and recall into one metric. 1 is best, 0 is worst.	<code>torchmetrics.F1Score()</code> or <code>sklearn.metrics.f1_score()</code>
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).	<code>torchmetrics.ConfusionMatrix</code> or <code>sklearn.metrics.plot_confusion_matrix()</code>
Classification report	Collection of some of the main classification metrics such as precision, recall and f1-score.	<code>sklearn.metrics.classification_report()</code>

Scikit-Learn (a popular and world-class machine learning library) has many implementations of the above metrics and you're looking for a PyTorch-like version, check out [TorchMetrics](#), especially the [TorchMetrics classification section](#).

Let's try the `torchmetrics.Accuracy` metric out.

In [60]:

Out[60]:

```
tensor(0.9950, device='cuda:0')
```

## Exercises

All of the exercises are focused on practicing the code in the sections above.

You should be able to complete them by referencing each section or by following the resource(s) linked.

All exercises should be completed using [device-agnostic code](#).

Resources:

- [Exercise template notebook for 02](#)
  - [Example solutions notebook for 02](#) (try the exercises *before* looking at this)
1. Make a binary classification dataset with Scikit-Learn's `make_moons()` function.
    - For consistency, the dataset should have 1000 samples and a `random_state=42`.
    - Turn the data into PyTorch tensors. Split the data into training and test sets using `train_test_split` with 80% training and 20% testing.

2. Build a model by subclassing `nn.Module` that incorporates non-linear activation functions and is capable of fitting the data you created in 1.
  - Feel free to use any combination of PyTorch layers (linear and non-linear) you want.
3. Setup a binary classification compatible loss function and optimizer to use when training the model.
4. Create a training and testing loop to fit the model you created in 2 to the data you created in 1.
  - To measure model accuracy, you can create your own accuracy function or use the accuracy function in [TorchMetrics](#).
  - Train the model for long enough for it to reach over 96% accuracy.
  - The training loop should output progress every 10 epochs of the model's training and test set loss and accuracy.
5. Make predictions with your trained model and plot them using the `plot_decision_boundary()` function created in this notebook.
6. Replicate the Tanh (hyperbolic tangent) activation function in pure PyTorch.
  - Feel free to reference the [ML cheatsheet website](#) for the formula.
7. Create a multi-class dataset using the [spirals data creation function from CS231n](#) (see below for the code).
  - Construct a model capable of fitting the data (you may need a combination of linear and non-linear layers).
  - Build a loss function and optimizer capable of handling multi-class data (optional extension: use the Adam optimizer instead of SGD, you may have to experiment with different values of the learning rate to get it working).
  - Make a training and testing loop for the multi-class data and train a model on it to reach over 95% testing accuracy (you can use any accuracy measuring function here that you like).
  - Plot the decision boundaries on the spirals dataset from your model predictions, the `plot_decision_boundary()` function should work for this dataset too.

```
# Code for creating a spiral dataset from CS231n
import numpy as np
N = 100 # number of points per class
D = 2 # dimensionality
K = 3 # number of classes
X = np.zeros((N*K,D)) # data matrix (each row = single example)
y = np.zeros(N*K, dtype='uint8') # class labels
for j in range(K):
    ix = range(N*j,N*(j+1))
    r = np.linspace(0.0,1,N) # radius
    t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
    X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
    y[ix] = j
# lets visualize the data
```

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)  
plt.show()
```

## Extra-curriculum

- Write down 3 problems where you think machine classification could be useful (these can be anything, get creative as you like, for example, classifying credit card transactions as fraud or not fraud based on the purchase amount and purchase location features).
- Research the concept of "momentum" in gradient-based optimizers (like SGD or Adam), what does it mean?
- Spend 10-minutes reading the [Wikipedia page for different activation functions](#), how many of these can you line up with [PyTorch's activation functions](#)?
- Research when accuracy might be a poor metric to use (hint: read "[Beyond Accuracy](#)" by [Will Koehrsen](#) for ideas).
- **Watch:** For an idea of what's happening within our neural networks and what they're doing to learn, watch [MIT's Introduction to Deep Learning video](#).

Previous

## 01. PyTorch Workflow Fundamentals

Next

## 03. PyTorch Computer Vision

Made with Material for MkDocs Insiders



[View Source Code](#) | [View Slides](#) | [Watch Video Walkthrough](#)

## 03. PyTorch Computer Vision

**Computer vision** is the art of teaching a computer to see.

For example, it could involve building a model to classify whether a photo is of a cat or a dog (**binary classification**).

Or whether a photo is of a cat, dog or chicken (**multi-class classification**).

Or identifying where a car appears in a video frame (**object detection**).

Or figuring out where different objects in an image can be separated (**panoptic segmentation**).

“Is this a photo of steak or pizza?”



**Binary classification**  
(one thing or another)

“Where’s the thing we’re looking for?”



**Object detection**

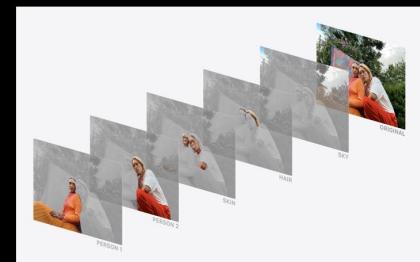
“Is this a photo of sushi, steak or pizza?”



**Multiclass classification**

(more than one thing or  
another)

“What are the different sections in this image?”



**Segmentation**

Example computer vision problems for binary classification, multiclass classification, object detection and segmentation.

## Where does computer vision get used?

If you use a smartphone, you've already used computer vision.

Camera and photo apps use **computer vision to enhance** and sort images.

Modern cars use **computer vision** to avoid other cars and stay within lane lines.

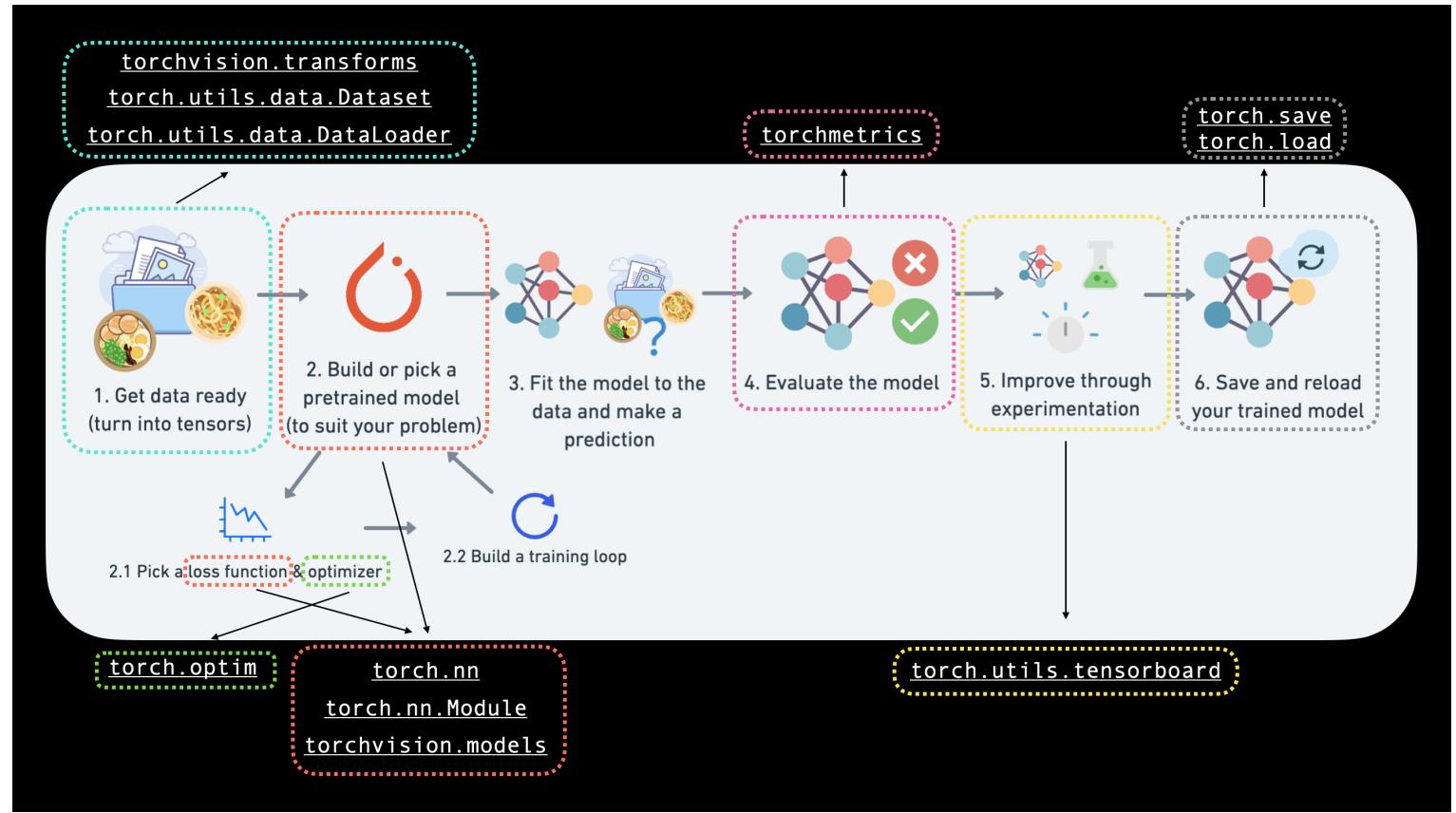
Manufacturers use computer vision to identify defects in various products.

Security cameras use computer vision to detect potential intruders.

In essence, anything that can be described in a visual sense can be a potential computer vision problem.

## What we're going to cover

We're going to apply the PyTorch Workflow we've been learning in the past couple of sections to computer vision.



Specifically, we're going to cover:

Topic	Contents
<b>0. Computer vision libraries in PyTorch</b>	PyTorch has a bunch of built-in helpful computer vision libraries, let's check them out.
<b>1. Load data</b>	To practice computer vision, we'll start with some images of different pieces of clothing from <a href="#">FashionMNIST</a> .
<b>2. Prepare data</b>	We've got some images, let's load them in with a PyTorch <code>DataLoader</code> so we can use them with our training loop.
<b>3. Model 0: Building a baseline model</b>	Here we'll create a multi-class classification model to learn patterns in the data, we'll also choose a <b>loss function</b> , <b>optimizer</b> and build a <b>training loop</b> .
<b>4. Making predictions and evaluating model 0</b>	Let's make some predictions with our baseline model and evaluate them.
<b>5. Setup device agnostic code for future models</b>	It's best practice to write device-agnostic code, so let's set it up.
<b>6. Model 1: Adding non-linearity</b>	Experimenting is a large part of machine learning, let's try and improve upon our baseline model by adding non-linear layers.
<b>7. Model 2: Convolutional Neural Network (CNN)</b>	Time to get computer vision specific and introduce the powerful convolutional neural network architecture.
<b>8. Comparing our models</b>	We've built three different models, let's compare them.
<b>9. Evaluating our best model</b>	Let's make some predictions on random images and evaluate our best model.
<b>10. Making a confusion matrix</b>	A confusion matrix is a great way to evaluate a classification model, let's see how we can make one.
<b>11. Saving and loading the best performing model</b>	Since we might want to use our model for later, let's save it and make sure it loads back in correctly.

## Where can you get help?

All of the materials for this course [live on GitHub](#).

If you run into trouble, you can ask a question on the course [GitHub Discussions page](#) there too.

And of course, there's the [PyTorch documentation](#) and [PyTorch developer forums](#), a very helpful place for all things PyTorch.

## 0. Computer vision libraries in PyTorch

Before we get started writing code, let's talk about some PyTorch computer vision libraries you should be aware of.

PyTorch module	What does it do?
<code>torchvision</code>	Contains datasets, model architectures and image transformations often used for computer vision problems.
<code>torchvision.datasets</code>	Here you'll find many example computer vision datasets for a range of problems from image classification, object detection, image captioning, video classification and more. It also contains <a href="#">a series of base classes for making custom datasets</a> .
<code>torchvision.models</code>	This module contains well-performing and commonly used computer vision model architectures implemented in PyTorch, you can use these with your own problems.
<code>torchvision.transforms</code>	Often images need to be transformed (turned into numbers/processed/augmented) before being used with a model, common image transformations are found here.
<code>torch.utils.data.Dataset</code>	Base dataset class for PyTorch.
<code>torch.utils.data.DataLoader</code>	Creates a Python iterable over a dataset (created with <code>torch.utils.data.Dataset</code> ).

**Note:** The `torch.utils.data.Dataset` and `torch.utils.data.DataLoader` classes aren't only for computer vision in PyTorch, they are capable of dealing with many different types of data.

Now we've covered some of the most important PyTorch computer vision libraries, let's import the relevant dependencies.

In [1]:





```
PyTorch version: 1.11.0
torchvision version: 0.12.0
```

## 1. Getting a dataset

To begin working on a computer vision problem, let's get a computer vision dataset.

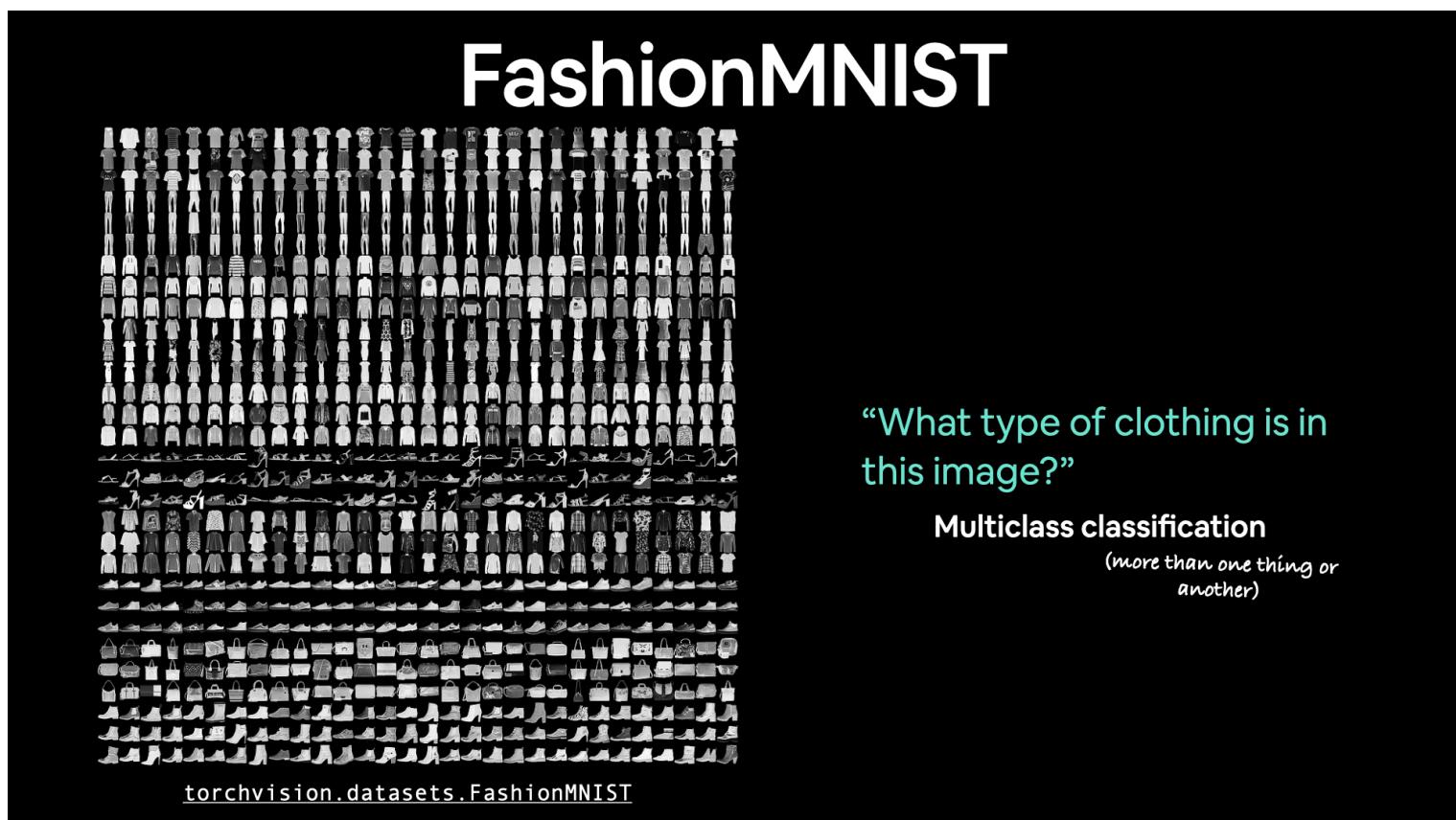
We're going to start with FashionMNIST.

MNIST stands for Modified National Institute of Standards and Technology.

The [original MNIST dataset](#) contains thousands of examples of handwritten digits (from 0 to 9) and was used to build computer vision models to identify numbers for postal services.

[FashionMNIST](#), made by Zalando Research, is a similar setup.

Except it contains grayscale images of 10 different kinds of clothing.



`torchvision.datasets` contains a lot of example datasets you can use to practice writing computer vision code on. *FashionMNIST* is one of those datasets. And since it has 10 different image classes (different types of clothing), it's a multi-class classification problem.

Later, we'll be building a computer vision neural network to identify the different styles of clothing in these images.

PyTorch has a bunch of common computer vision datasets stored in `torchvision.datasets`.

Including *FashionMNIST* in `torchvision.datasets.FashionMNIST()`.

To download it, we provide the following parameters:

- `root: str` - which folder do you want to download the data to?
- `train: Bool` - do you want the training or test split?
- `download: Bool` - should the data be downloaded?
- `transform: torchvision.transforms` - what transformations would you like to do on the data?
- `target_transform` - you can transform the targets (labels) if you like too.

Many other datasets in `torchvision` have these parameter options.

In [2]:





Let's check out the first sample of the training data.

In [3]:

Out[3]:

```
(tensor([[[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000], ,
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000], ,
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000], ,
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000], ,
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0039, 0.0000, 0.0000, 0.0510,
         0.2863, 0.0000, 0.0000, 0.0039, 0.0157, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0039, 0.0039, 0.0000], ,
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0118, 0.0000, 0.1412, 0.5333,
         0.4980, 0.2431, 0.2118, 0.0000, 0.0000, 0.0000, 0.0039, 0.0118,
         0.0157, 0.0000, 0.0000, 0.0118], ,
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0235, 0.0000, 0.4000, 0.8000,
         0.6902, 0.5255, 0.5647, 0.4824, 0.0902, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0471, 0.0392, 0.0000], ,
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.6078, 0.9255,
         0.8118, 0.6980, 0.4196, 0.6118, 0.6314, 0.4275, 0.2510, 0.0902,
         0.3020, 0.5098, 0.2824, 0.0588], ,
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0039, 0.0000, 0.2706, 0.8118, 0.8745,
         0.8549, 0.8471, 0.8471, 0.6392, 0.4980, 0.4745, 0.4784, 0.5725,
         0.5529, 0.3451, 0.6745, 0.2588], ,
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0039, 0.0039, 0.0000, 0.7843, 0.9098, 0.9098,
         0.9137, 0.8980, 0.8745, 0.8745, 0.8431, 0.8353, 0.6431, 0.4980,
         0.4824, 0.7686, 0.8980, 0.0000], ,
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0000, 0.0000, 0.0000, 0.7176, 0.8824, 0.8471,
         0.8745, 0.8941, 0.9216, 0.8902, 0.8784, 0.8706, 0.8784, 0.8667,
         0.8745, 0.9608, 0.6784, 0.0000], ]
```

```
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,  
0.0000, 0.0000, 0.0000, 0.0000, 0.7569, 0.8941, 0.8549,  
0.8353, 0.7765, 0.7059, 0.8314, 0.8235, 0.8275, 0.8353, 0.8745,  
0.8627, 0.9529, 0.7922, 0.0000],  
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,  
0.0000, 0.0039, 0.0118, 0.0000, 0.0471, 0.8588, 0.8627, 0.8314,  
0.8549, 0.7529, 0.6627, 0.8902, 0.8157, 0.8549, 0.8784, 0.8314,  
0.8863, 0.7725, 0.8196, 0.2039],  
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,  
0.0000, 0.0000, 0.0235, 0.0000, 0.3882, 0.9569, 0.8706, 0.8627,  
0.8549, 0.7961, 0.7765, 0.8667, 0.8431, 0.8353, 0.8706, 0.8627,  
0.9608, 0.4667, 0.6549, 0.2196],  
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,  
0.0000, 0.0157, 0.0000, 0.0000, 0.2157, 0.9255, 0.8941, 0.9020,  
0.8941, 0.9412, 0.9098, 0.8353, 0.8549, 0.8745, 0.9176, 0.8510,  
0.8510, 0.8196, 0.3608, 0.0000],  
[0.0000, 0.0000, 0.0039, 0.0157, 0.0235, 0.0275, 0.0078, 0.0000,  
0.0000, 0.0000, 0.0000, 0.9294, 0.8863, 0.8510, 0.8745,  
0.8706, 0.8588, 0.8706, 0.8667, 0.8471, 0.8745, 0.8980, 0.8431,  
0.8549, 1.0000, 0.3020, 0.0000],  
[0.0000, 0.0118, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,  
0.0000, 0.2431, 0.5686, 0.8000, 0.8941, 0.8118, 0.8353, 0.8667,  
0.8549, 0.8157, 0.8275, 0.8549, 0.8784, 0.8745, 0.8588, 0.8431,  
0.8784, 0.9569, 0.6235, 0.0000],  
[0.0000, 0.0000, 0.0000, 0.0000, 0.0706, 0.1725, 0.3216, 0.4196,  
0.7412, 0.8941, 0.8627, 0.8706, 0.8510, 0.8863, 0.7843, 0.8039,  
0.8275, 0.9020, 0.8784, 0.9176, 0.6902, 0.7373, 0.9804, 0.9725,  
0.9137, 0.9333, 0.8431, 0.0000],  
[0.0000, 0.2235, 0.7333, 0.8157, 0.8784, 0.8667, 0.8784, 0.8157,  
0.8000, 0.8392, 0.8157, 0.8196, 0.7843, 0.6235, 0.9608, 0.7569,  
0.8078, 0.8745, 1.0000, 1.0000, 0.8667, 0.9176, 0.8667, 0.8275,  
0.8627, 0.9098, 0.9647, 0.0000],  
[0.0118, 0.7922, 0.8941, 0.8784, 0.8667, 0.8275, 0.8275, 0.8392,  
0.8039, 0.8039, 0.8039, 0.8627, 0.9412, 0.3137, 0.5882, 1.0000,  
0.8980, 0.8667, 0.7373, 0.6039, 0.7490, 0.8235, 0.8000, 0.8196,  
0.8706, 0.8941, 0.8824, 0.0000],  
[0.3843, 0.9137, 0.7765, 0.8235, 0.8706, 0.8980, 0.8980, 0.9176,  
0.9765, 0.8627, 0.7608, 0.8431, 0.8510, 0.9451, 0.2549, 0.2863,  
0.4157, 0.4588, 0.6588, 0.8588, 0.8667, 0.8431, 0.8510, 0.8745,  
0.8745, 0.8784, 0.8980, 0.1137],  
[0.2941, 0.8000, 0.8314, 0.8000, 0.7569, 0.8039, 0.8275, 0.8824,  
0.8471, 0.7255, 0.7725, 0.8078, 0.7765, 0.8353, 0.9412, 0.7647,  
0.8902, 0.9608, 0.9373, 0.8745, 0.8549, 0.8314, 0.8196, 0.8706,  
0.8627, 0.8667, 0.9020, 0.2627],  
[0.1882, 0.7961, 0.7176, 0.7608, 0.8353, 0.7725, 0.7255, 0.7451,  
0.7608, 0.7529, 0.7922, 0.8392, 0.8588, 0.8667, 0.8627, 0.9255,  
0.8824, 0.8471, 0.7804, 0.8078, 0.7294, 0.7098, 0.6941, 0.6745,  
0.7098, 0.8039, 0.8078, 0.4510],  
[0.0000, 0.4784, 0.8588, 0.7569, 0.7020, 0.6706, 0.7176, 0.7686,  
0.8000, 0.8235, 0.8353, 0.8118, 0.8275, 0.8235, 0.7843, 0.7686,  
0.7608, 0.7490, 0.7647, 0.7490, 0.7765, 0.7529, 0.6902, 0.6118,  
0.6549, 0.6941, 0.8235, 0.3608],  
[0.0000, 0.0000, 0.2902, 0.7412, 0.8314, 0.7490, 0.6863, 0.6745,  
0.6863, 0.7098, 0.7255, 0.7373, 0.7412, 0.7373, 0.7569, 0.7765,
```

```

0.8000, 0.8196, 0.8235, 0.8235, 0.8275, 0.7373, 0.7373, 0.7608,
0.7529, 0.8471, 0.6667, 0.0000],
[0.0078, 0.0000, 0.0000, 0.0000, 0.2588, 0.7843, 0.8706, 0.9294,
0.9373, 0.9490, 0.9647, 0.9529, 0.9569, 0.8667, 0.8627, 0.7569,
0.7490, 0.7020, 0.7137, 0.7137, 0.7098, 0.6902, 0.6510, 0.6588,
0.3882, 0.2275, 0.0000, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.1569,
0.2392, 0.1725, 0.2824, 0.1608, 0.1373, 0.0000, 0.0000, 0.0000,
0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
0.0000, 0.0000, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
0.0000, 0.0000, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
0.0000, 0.0000, 0.0000]]),

```

9)

## 1.1 Input and output shapes of a computer vision model

We've got a big tensor of values (the image) leading to a single value for the target (the label).

Let's see the image shape.

In [4]:

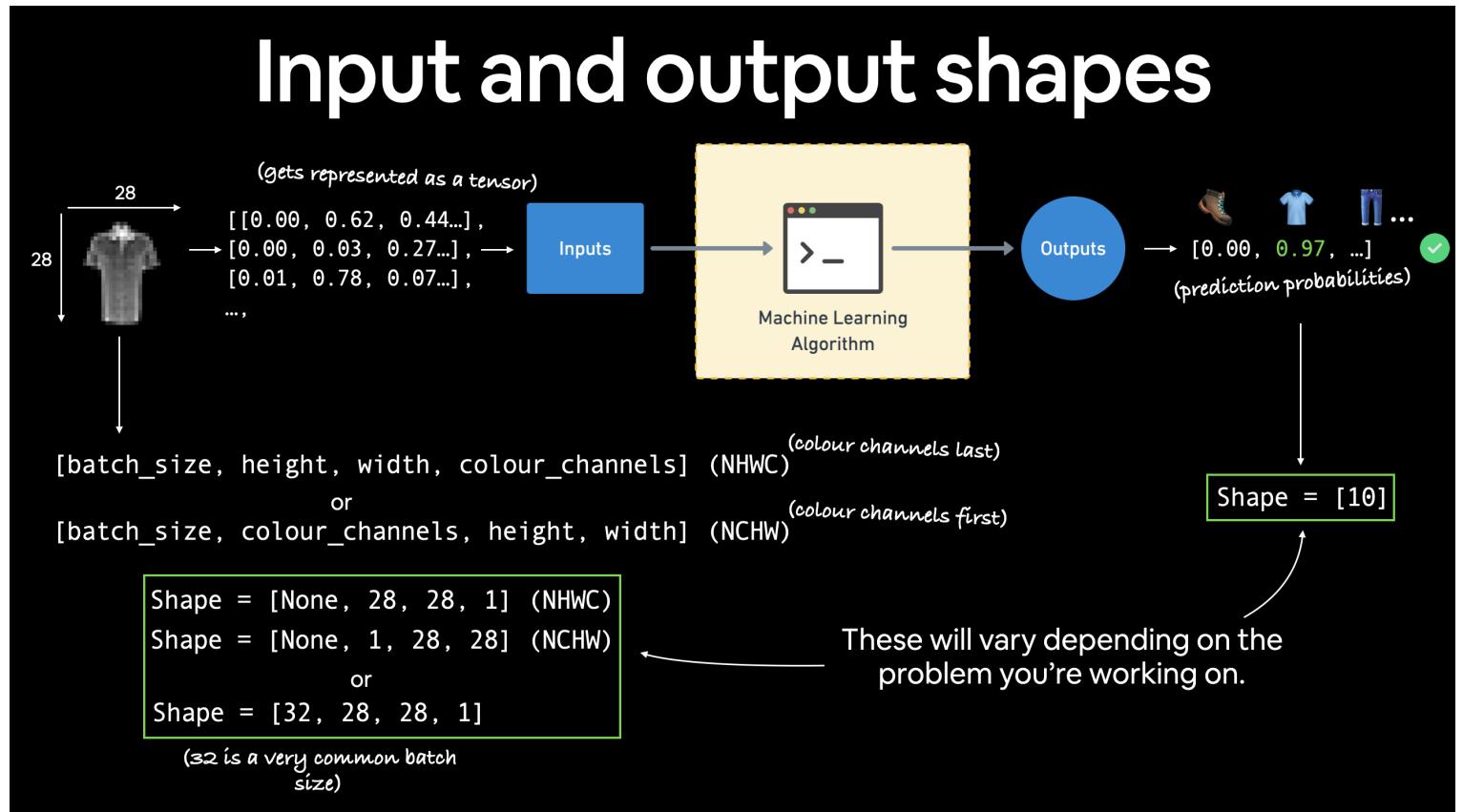
```
torch.Size([1, 28, 28])
```

The shape of the image tensor is [1, 28, 28] or more specifically:

```
[color_channels=1, height=28, width=28]
```

Out[4]:

Having `color_channels=1` means the image is grayscale.



Various problems will have various input and output shapes. But the premise remains: encode data into numbers, build a model to find patterns in those numbers, convert those patterns into something meaningful.

If `color_channels=3`, the image comes in pixel values for red, green and blue (this is also known as the [RGB color model](#)).

The order of our current tensor is often referred to as `CHW` (Color Channels, Height, Width).

There's debate on whether images should be represented as `CHW` (color channels first) or `HWC` (color channels last).

**Note:** You'll also see `NCHW` and `NHWC` formats where `N` stands for *number of images*. For example if you have a `batch_size=32`, your tensor shape may be `[32, 1, 28, 28]`. We'll cover batch sizes later.

PyTorch generally accepts `NCHW` (channels first) as the default for many operators.

However, PyTorch also explains that `NHWC` (channels last) performs better and is [considered best practice](#).

For now, since our dataset and models are relatively small, this won't make too much of a difference.

But keep it in mind for when you're working on larger image datasets and using convolutional neural networks (we'll see these later).

Let's check out more shapes of our data.

In [5]:

(60000, 60000, 10000, 10000)

So we've got 60,000 training samples and 10,000 testing samples.

What classes are there?

We can find these via the `.classes` attribute.

In [6]:

```
['T-shirt/top',  
 'Trouser',
```

Out[6]:

```
'Pullover',
'Dress',
'Coat',
'Sandal',
'Shirt',
'Sneaker',
'Bag',
'Ankle boot']
```

Sweet! It looks like we're dealing with 10 different kinds of clothes.

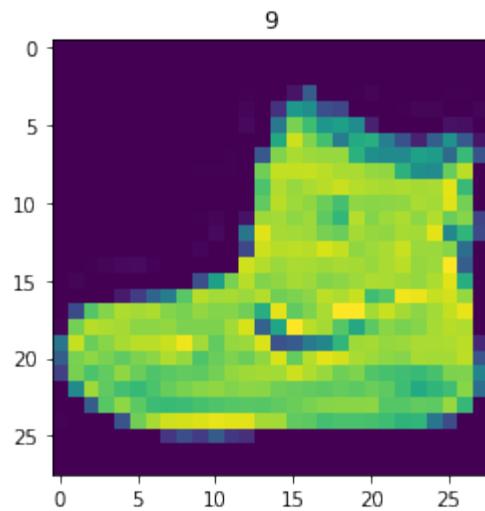
Because we're working with 10 different classes, it means our problem is **multi-class classification**.

Let's get visual.

## 1.2 Visualizing our data

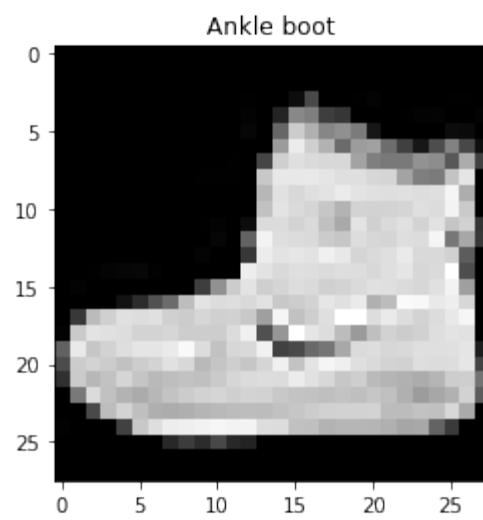
In [7]:

Image shape: `torch.Size([1, 28, 28])`



We can turn the image into grayscale using the `cmap` parameter of `plt.imshow()`.

In [8]:

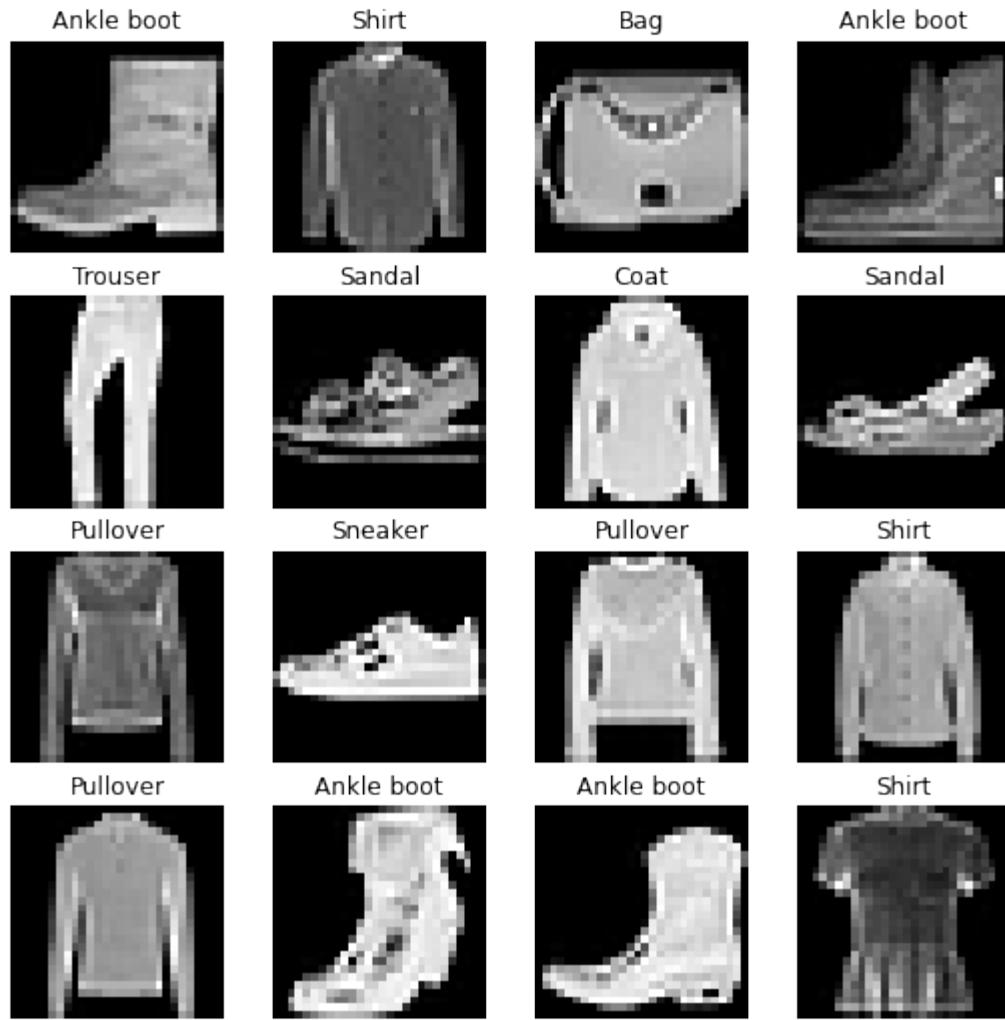


Beautiful, well as beautiful as a pixelated grayscale ankle boot can get.

Let's view a few more.

In [9]:





Hmmm, this dataset doesn't look too aesthetic.

But the principles we're going to learn on how to build a model for it will be similar across a wide range of computer vision problems.

In essence, taking pixel values and building a model to find patterns in them to use on future pixel values.

Plus, even for this small dataset (yes, even 60,000 images in deep learning is considered quite small), could you write a program to classify each one of them?

You probably could.

But I think coding a model in PyTorch would be faster.

**Question:** Do you think the above data can be model with only straight (linear) lines? Or do you think you'd also need non-straight (non-linear) lines?

## 2. Prepare DataLoader

Now we've got a dataset ready to go.

The next step is to prepare it with a `torch.utils.data.DataLoader` or `DataLoader` for short.

The `DataLoader` does what you think it might do.

It helps load data into a model.

For training and for inference.

It turns a large `Dataset` into a Python iterable of smaller chunks.

These smaller chunks are called **batches** or **mini-batches** and can be set by the `batch_size` parameter.

Why do this?

Because it's more computationally efficient.

In an ideal world you could do the forward pass and backward pass across all of your data at once.

But once you start using really large datasets, unless you've got infinite computing power, it's easier to break them up into batches.

It also gives your model more opportunities to improve.

With **mini-batches** (small portions of the data), gradient descent is performed more often per epoch (once per mini-batch rather than once per epoch).

What's a good batch size?

32 is a good place to start for a fair amount of problems.

But since this is a value you can set (a **hyperparameter**) you can try all different kinds of values, though generally powers of 2 are used most often (e.g. 32, 64, 128, 256, 512).



Batching FashionMNIST with a batch size of 32 and shuffle turned on. A similar batching process will occur for other datasets but will differ depending on the batch size.

Let's create `DataLoader`'s for our training and test sets.

In [10]:







```
Dataloaders: (<torch.utils.data.dataloader.DataLoader object at 0x7f9e193a8a90>, <torch.utils.dat  
a.dataloader.DataLoader object at 0x7f9e193b0700>)  
Length of train dataloader: 1875 batches of 32  
Length of test dataloader: 313 batches of 32
```

In [11]:

Out[11]:

```
(torch.Size([32, 1, 28, 28]), torch.Size([32]))
```

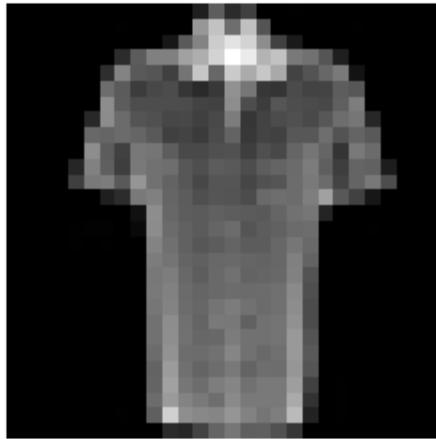
And we can see that the data remains unchanged by checking a single sample.

In [12]:

Image size: torch.Size([1, 28, 28])

Label: 6, label size: torch.Size([])

Shirt



### 3. Model 0: Build a baseline model

Data loaded and prepared!

Time to build a **baseline model** by subclassing `nn.Module`.

A **baseline model** is one of the simplest models you can imagine.

You use the baseline as a starting point and try to improve upon it with subsequent, more complicated models.

Our baseline will consist of two `nn.Linear()` layers.

We've done this in a previous section but there's going to be one slight difference.

Because we're working with image data, we're going to use a different layer to start things off.

And that's the `nn.Flatten()` layer.

`nn.Flatten()` compresses the dimensions of a tensor into a single vector.

This is easier to understand when you see it.

In [13]:





```
Shape before flattening: torch.Size([1, 28, 28]) -> [color_channels, height, width]
Shape after flattening: torch.Size([1, 784]) -> [color_channels, height*width]
```

The `nn.Flatten()` layer took our shape from `[color_channels, height, width]` to `[color_channels, height*width]`.

Why do this?

Because we've now turned our pixel data from height and width dimensions into one long **feature vector**.

And `nn.Linear()` layers like their inputs to be in the form of feature vectors.

Let's create our first model using `nn.Flatten()` as the first layer.

In [14]:







Wonderful!

We've got a baseline model class we can use, now let's instantiate a model.

We'll need to set the following parameters:

- `input_shape=784` - this is how many features you've got going in the model, in our case, it's one for every pixel in the target image (28 pixels high by 28 pixels wide = 784 features).
- `hidden_units=10` - number of units/neurons in the hidden layer(s), this number could be whatever you want but to keep the model small we'll start with `10`.
- `output_shape=len(class_names)` - since we're working with a multi-class classification problem, we need an output neuron per class in our dataset.

Let's create an instance of our model and send to the CPU for now (we'll run a small test for running `model_0` on CPU vs. a similar model on GPU soon).

In [15]:



Out[15]:

```
FashionMNISTModelV0(  
    (layer_stack): Sequential(  
        (0): Flatten(start_dim=1, end_dim=-1)  
        (1): Linear(in_features=784, out_features=10, bias=True)  
        (2): Linear(in_features=10, out_features=10, bias=True)  
    )  
)
```

### 3.1 Setup loss, optimizer and evaluation metrics

Since we're working on a classification problem, let's bring in our `helper_functions.py` script and subsequently the `accuracy_fn()` we defined in [notebook 02](#).

**Note:** Rather than importing and using our own accuracy function or evaluation metric(s), you could import various evaluation metrics from the [TorchMetrics package](#).

In [16]:



```
helper_functions.py already exists, skipping download
```

In [17]:

## 3.2 Creating a function to time our experiments

Loss function and optimizer ready!

It's time to start training a model.

But how about we do a little experiment while we train.

I mean, let's make a timing function to measure the time it takes our model to train on CPU versus using a GPU.

We'll train this model on the CPU but the next one on the GPU and see what happens.

Our timing function will import the `timeit.default_timer()` function from the Python `timeit` module.

In [18]:







### 3.3 Creating a training loop and training a model on batches of data

Beautiful!

Looks like we've got all of the pieces of the puzzle ready to go, a timer, a loss function, an optimizer, a model and

most importantly, some data.

Let's now create a training loop and a testing loop to train and evaluate our model.

We'll be using the same steps as the previous notebook(s), though since our data is now in batch form, we'll add another loop to loop through our data batches.

Our data batches are contained within our `DataLoader`s, `train_dataloader` and `test_dataloader` for the training and test data splits respectively.

A batch is `BATCH_SIZE` samples of `x` (features) and `y` (labels), since we're using `BATCH_SIZE=32`, our batches have 32 samples of images and targets.

And since we're computing on batches of data, our loss and evaluation metrics will be calculated **per batch** rather than across the whole dataset.

This means we'll have to divide our loss and accuracy values by the number of batches in each dataset's respective dataloader.

Let's step through it:

1. Loop through epochs.
2. Loop through training batches, perform training steps, calculate the train loss *per batch*.
3. Loop through testing batches, perform testing steps, calculate the test loss *per batch*.
4. Print out what's happening.
5. Time it all (for fun).

A fair few steps but...

...if in doubt, code it out.

In [19]:





































```
0% | 0/3 [00:00<?, ?it/s]
Epoch: 0
-----
Looked at 0/60000 samples
Looked at 12800/60000 samples
Looked at 25600/60000 samples
Looked at 38400/60000 samples
Looked at 51200/60000 samples

Train loss: 0.59039 | Test loss: 0.50954, Test acc: 82.04%

Epoch: 1
-----
Looked at 0/60000 samples
Looked at 12800/60000 samples
Looked at 25600/60000 samples
Looked at 38400/60000 samples
Looked at 51200/60000 samples

Train loss: 0.47633 | Test loss: 0.47989, Test acc: 83.20%

Epoch: 2
```

```
-----  
Looked at 0/60000 samples  
Looked at 12800/60000 samples  
Looked at 25600/60000 samples  
Looked at 38400/60000 samples  
Looked at 51200/60000 samples  
  
Train loss: 0.45503 | Test loss: 0.47664, Test acc: 83.43%  
  
Train time on cpu: 14.975 seconds
```

Nice! Looks like our baseline model did fairly well.

It didn't take too long to train either, even just on the CPU, I wonder if it'll speed up on the GPU?

Let's write some code to evaluate our model.

## 4. Make predictions and get Model 0 results

Since we're going to be building a few models, it's a good idea to write some code to evaluate them all in similar ways.

Namely, let's create a function that takes in a trained model, a `DataLoader`, a loss function and an accuracy function.

The function will use the model to make predictions on the data in the `DataLoader` and then we can evaluate those predictions using the loss function and accuracy function.

In [20]:























Out[20]:

```
{'model_name': 'FashionMNISTModelV0',
'model_loss': 0.47663894295692444,
'model_acc': 83.42651757188499}
```

Looking good!

We can use this dictionary to compare the baseline model results to other models later on.

## 5. Setup device agnostic-code (for using a GPU if there is one)

We've seen how long it takes to train ma PyTorch model on 60,000 samples on CPU.

**Note:** Model training time is dependent on hardware used. Generally, more processors means faster training and smaller models on smaller datasets will often train faster than large models and large datasets.

Now let's setup some [device-agnostic code](#) for our models and data to run on GPU if it's available.

If you're running this notebook on Google Colab, and you don't a GPU turned on yet, it's now time to turn one on via  
Runtime -> Change runtime type -> Hardware accelerator -> GPU. If you do this, your runtime will likely reset and  
you'll have to run all of the cells above by going Runtime -> Run before.

In [21]:

```
Out[21]:
```

```
'cuda'
```

Beautiful!

Let's build another model.

## 6. Model 1: Building a better model with non-linearity

We learned about [the power of non-linearity in notebook 02](#).

Seeing the data we've been working with, do you think it needs non-linear functions?

And remember, linear means straight and non-linear means non-straight.

Let's find out.

We'll do so by recreating a similar model to before, except this time we'll put non-linear functions (`nn.ReLU()`) in between each linear layer.

In [22]:





That looks good.

Now let's instantiate it with the same settings we used before.

We'll need `input_shape=784` (equal to the number of features of our image data), `hidden_units=10` (starting small and the same as our baseline model) and `output_shape=len(class_names)` (one output unit per class).

**Note:** Notice how we kept most of the settings of our model the same except for one change: adding non-linear layers. This is a standard practice for running a series of machine learning experiments, change one thing and see what happens, then do it again, again, again.

In [23]:

Out[23]:

```
device(type='cuda', index=0)
```

## 6.1 Setup loss, optimizer and evaluation metrics

As usual, we'll setup a loss function, an optimizer and an evaluation metric (we could do multiple evaluation metrics but we'll stick with accuracy for now).

In [24]:

## 6.2 Functionizing training and test loops

So far we've been writing train and test loops over and over.

Let's write them again but this time we'll put them in functions so they can be called again and again.

And because we're using device-agnostic code now, we'll be sure to call `.to(device)` on our feature (`x`) and target (`y`) tensors.

For the training loop we'll create a function called `train_step()` which takes in a model, a `DataLoader` a loss function

and an optimizer.

The testing loop will be similar but it'll be called `test_step()` and it'll take in a model, a `DataLoader`, a loss function and an evaluation function.

**Note:** Since these are functions, you can customize them in any way you like. What we're making here can be considered barebones training and testing functions for our specific classification use case.

In [25]:

































Woohoo!

Now we've got some functions for training and testing our model, let's run them.

We'll do so inside another loop for each epoch.

That way for each epoch we're going a training and a testing step.

**Note:** You can customize how often you do a testing step. Sometimes people do them every five epochs or 10 epochs or in our case, every epoch.

Let's also time things to see how long our code takes to run on the GPU.

In [26]:









```
0% | 0 / 3 [00:00<?, ?it/s]
Epoch: 0
-----
Train loss: 1.09199 | Train accuracy: 61.34%
Test loss: 0.95636 | Test accuracy: 65.00%

Epoch: 1
-----
Train loss: 0.78101 | Train accuracy: 71.93%
Test loss: 0.72227 | Test accuracy: 73.91%

Epoch: 2
-----
Train loss: 0.67027 | Train accuracy: 75.94%
Test loss: 0.68500 | Test accuracy: 75.02%

Train time on cuda: 16.943 seconds
```

Excellent!

Our model trained but the training time took longer?

**Note:** The training time on CUDA vs CPU will depend largely on the quality of the CPU/GPU you're using. Read on for a more explained answer.

**Question:** "I used a GPU but my model didn't train faster, why might that be?"

**Answer:** Well, one reason could be because your dataset and model are both so small (like the dataset and model we're working with) the benefits of using a GPU are outweighed by the time it actually takes to transfer the data there.

There's a small bottleneck between copying data from the CPU memory (default) to the GPU memory.

So for smaller models and datasets, the CPU might actually be the optimal place to compute on.

But for larger datasets and models, the speed of computing the GPU can offer usually far outweighs the cost of getting the data there.

However, this is largely dependant on the hardware you're using. With practice, you will get used to where the best place to train your models is.

Let's evaluate our trained `model_1` using our `eval_model()` function and see how it went.

In [27]:

```
-----  
RuntimeError                                     Traceback (most recent call last)  
/tmp/ipykernel_1084458/2906876561.py in <module>  
      2  
      3 # Note: This will error due to `eval_model()` not using device agnostic code  
----> 4 model_1_results = eval_model(model=model_1,  
      5         data_loader=test_dataloader,  
      6         loss_fn=loss_fn,  
  
/tmp/ipykernel_1084458/2300884397.py in eval_model(model, data_loader, loss_fn, accuracy_fn)  
     20         for X, y in data_loader:  
     21             # Make predictions with the model  
---> 22             y_pred = model(X)  
     23  
     24             # Accumulate the loss and accuracy values per batch  
  
~/code/pytorch/env/lib/python3.9/site-packages/torch/nn/modules/module.py in __call__(self, *input, **kwargs)  
    1108         if not (self._backward_hooks or self._forward_hooks or self._forward_pre_hooks or  
    _global_backward_hooks  
    1109             or _global_forward_hooks or _global_forward_pre_hooks):  
-> 1110             return forward_call(*input, **kwargs)  
    1111         # Do not call functions when jit is used  
    1112         full_backward_hooks, non_full_backward_hooks = [], []  
  
/tmp/ipykernel_1084458/3744982926.py in forward(self, x)  
    12  
    13     def forward(self, x: torch.Tensor):  
---> 14         return self.layer_stack(x)  
  
~/code/pytorch/env/lib/python3.9/site-packages/torch/nn/modules/module.py in __call__(self, *input, **kwargs)  
    1108         if not (self._backward_hooks or self._forward_hooks or self._forward_pre_hooks or  
    _global_backward_hooks  
    1109             or _global_forward_hooks or _global_forward_pre_hooks):  
-> 1110             return forward_call(*input, **kwargs)
```

```

1111      # Do not call functions when jit is used
1112      full_backward_hooks, non_full_backward_hooks = [], []
1113
~/code/pytorch/env/lib/python3.9/site-packages/torch/nn/modules/container.py in forward(self, input)
139      def forward(self, input):
140          for module in self:
--> 141              input = module(input)
142          return input
143
~/code/pytorch/env/lib/python3.9/site-packages/torch/nn/modules/module.py in __call__(self, *input, **kwargs)
1108          if not (self._backward_hooks or self._forward_hooks or self._forward_pre_hooks or
1109                  _global_backward_hooks
1110                  or _global_forward_hooks or _global_forward_pre_hooks):
-> 1110          return forward_call(*input, **kwargs)
1111          # Do not call functions when jit is used
1112          full_backward_hooks, non_full_backward_hooks = [], []
1113
~/code/pytorch/env/lib/python3.9/site-packages/torch/nn/modules/linear.py in forward(self, input)
101
102      def forward(self, input: Tensor) -> Tensor:
--> 103          return F.linear(input, self.weight, self.bias)
104
105      def extra_repr(self) -> str:

```

RuntimeError: Expected all tensors to be on the same device, but found at least two devices, cuda:0 and cpu! (when checking argument for argument mat1 in method wrapper\_addmm)

Oh no!

It looks like our `eval_model()` function errors out with:

```
RuntimeError: Expected all tensors to be on the same device, but found at least two devices, cuda:0 and cpu!
(when checking argument for argument mat1 in method wrapper_addmm)
```

It's because we've setup our data and model to use device-agnostic code but not our evaluation function.

How about we fix that by passing a target `device` parameter to our `eval_model()` function?

Then we'll try calculating the results again.

In [28]:























Out[28]:

```
{'model_name': 'FashionMNISTModelV1',
'model_loss': 0.6850008964538574,
'model_acc': 75.01996805111821}
```

In [29]:

Out[29]:

```
{'model_name': 'FashionMNISTModelV0',
'model_loss': 0.47663894295692444,
'model_acc': 83.42651757188499}
```

Woah, in this case, it looks like adding non-linearities to our model made it perform worse than the baseline.

That's a thing to note in machine learning, sometimes the thing you thought should work doesn't.

And then the thing you thought might not work does.

It's part science, part art.

From the looks of things, it seems like our model is **overfitting** on the training data.

Overfitting means our model is learning the training data well but those patterns aren't generalizing to the testing data.

Two of the main to fix overfitting include:

1. Using a smaller or different model (some models fit certain kinds of data better than others).
2. Using a larger dataset (the more data, the more chance a model has to learn generalizable patterns).

There are more, but I'm going to leave that as a challenge for you to explore.

Try searching online, "ways to prevent overfitting in machine learning" and see what comes up.

In the meantime, let's take a look at number 1: using a different model.

## 7. Model 2: Building a Convolutional Neural Network (CNN)

Alright, time to step things up a notch.

It's time to create a [Convolutional Neural Network](#) (CNN or ConvNet).

CNN's are known for their capabilities to find patterns in visual data.

And since we're dealing with visual data, let's see if using a CNN model can improve upon our baseline.

The CNN model we're going to be using is known as TinyVGG from the [CNN Explainer](#) website.

It follows the typical structure of a convolutional neural network:

```
Input layer -> [Convolutional layer -> activation layer -> pooling layer] -> Output layer
```

Where the contents of [Convolutional layer -> activation layer -> pooling layer] can be upscaled and repeated multiple times, depending on requirements.

### What model should I use?

**Question:** Wait, you say CNN's are good for images, are there any other model types I should be aware of?

Good question.

This table is a good general guide for which model to use (though there are exceptions).

Problem type	Model to use (generally)	Code example
Structured data (Excel spreadsheets, row and column data)	Gradient boosted models, Random Forest	<code>sklearn.ensemble</code> , <code>XGBoost library</code>

column data)

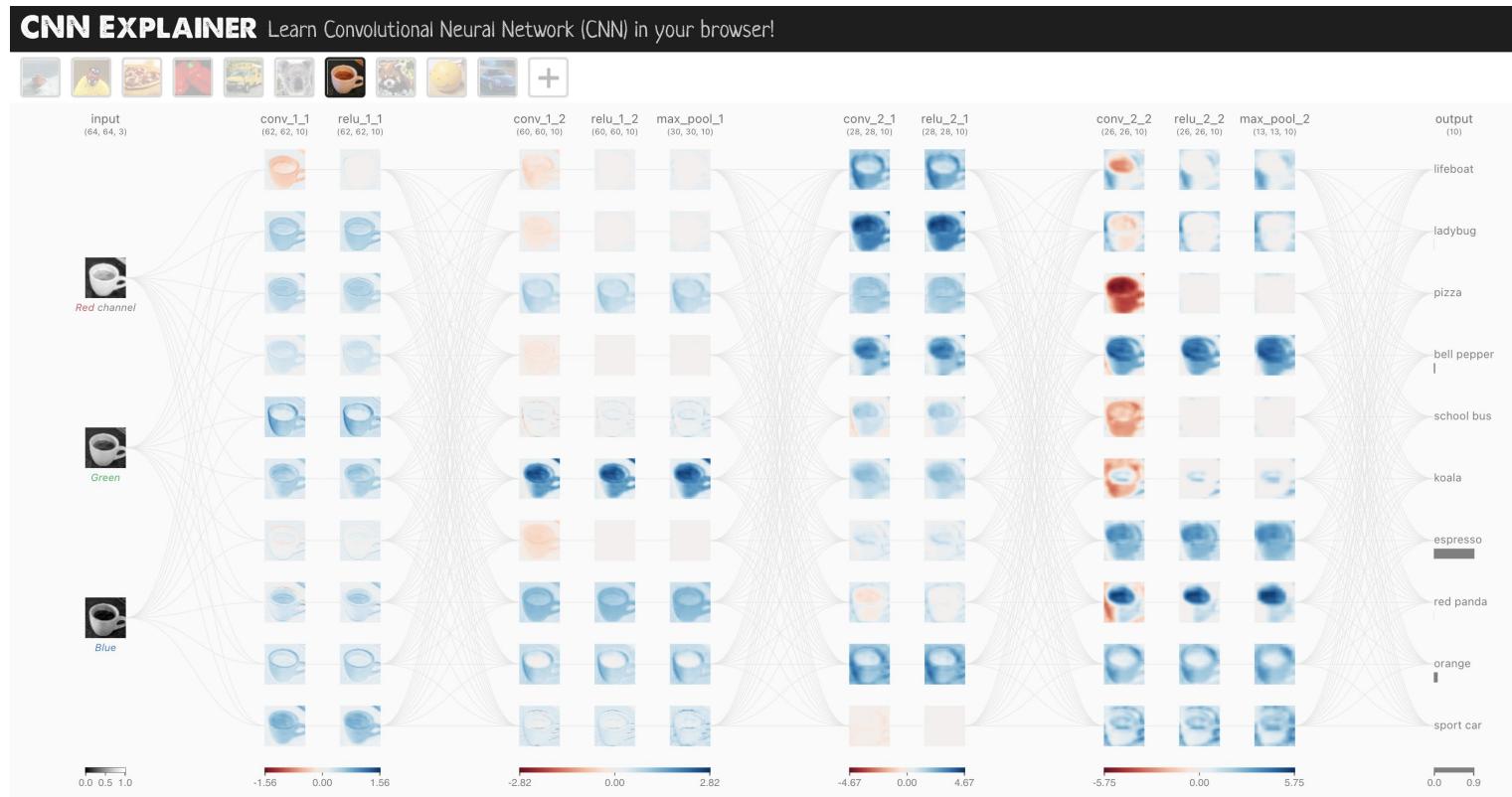
Forests, XGBoost

Unstructured data (images, audio, language)

Convolutional Neural Networks,  
Transformers`torchvision.models`, HuggingFace  
Transformers

**Note:** The table above is only for reference, the model you end up using will be highly dependant on the problem you're working on and the constraints you have (amount of data, latency requirements).

Enough talking about models, let's now build a CNN that replicates the model on the [CNN Explainer website](#).



To do so, we'll leverage the `nn.Conv2d()` and `nn.MaxPool2d()` layers from `torch.nn`.

In [34]:































Out[34]:

```
FashionMNISTModelV2(  
    (block_1): Sequential(  
        (0): Conv2d(1, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (1): ReLU()  
        (2): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (3): ReLU()  
        (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
```

```

)
(block_2): Sequential(
(0): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(1): ReLU()
(2): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(3): ReLU()
(4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
(classifier): Sequential(
(0): Flatten(start_dim=1, end_dim=-1)
(1): Linear(in_features=490, out_features=10, bias=True)
)
)
)

```

Nice!

Our biggest model yet!

What we've done is a common practice in machine learning.

Find a model architecture somewhere and replicate it with code.

## 7.1 Stepping through `nn.Conv2d()`

We could start using our model above and see what happens but let's first step through the two new layers we've added:

- `nn.Conv2d()`, also known as a convolutional layer.
- `nn.MaxPool2d()`, also known as a max pooling layer.

**Question:** What does the "2d" in `nn.Conv2d()` stand for?

The 2d is for 2-dimensional data. As in, our images have two dimensions: height and width. Yes, there's color channel dimension but each of the color channel dimensions have two dimensions too: height and width.

For other dimensional data (such as 1D for text or 3D for 3D objects) there's also `nn.Conv1d()` and `nn.Conv3d()`.

To test the layers out, let's create some toy data just like the data used on CNN Explainer.

In [35]:





```

Image batch shape: torch.Size([32, 3, 64, 64]) -> [batch_size, color_channels, height, width]
Single image shape: torch.Size([3, 64, 64]) -> [color_channels, height, width]
Single image pixel values:
tensor([[[ 1.9269,   1.4873,   0.9007,   ... ,  1.8446,  -1.1845,   1.3835],
        [ 1.4451,   0.8564,   2.2181,   ... ,  0.3399,   0.7200,   0.4114],
        [ 1.9312,   1.0119,  -1.4364,   ... , -0.5558,   0.7043,   0.7099],
        ... ,
        [-0.5610,  -0.4830,   0.4770,   ... , -0.2713,  -0.9537,  -0.6737],
        [ 0.3076,  -0.1277,   0.0366,   ... , -2.0060,   0.2824,  -0.8111],
        [-1.5486,   0.0485,  -0.7712,   ... , -0.1403,   0.9416,  -0.0118]],

       [[-0.5197,   1.8524,   1.8365,   ... ,  0.8935,  -1.5114,  -0.8515],
        [ 2.0818,   1.0677,  -1.4277,   ... ,  1.6612,  -2.6223,  -0.4319],
        [-0.1010,  -0.4388,  -1.9775,   ... ,  0.2106,   0.2536,  -0.7318],
        ... ,
        [ 0.2779,   0.7342,  -0.3736,   ... , -0.4601,   0.1815,   0.1850],
        [ 0.7205,  -0.2833,   0.0937,   ... , -0.1002,  -2.3609,   2.2465],
        [-1.3242,  -0.1973,   0.2920,   ... ,  0.5409,   0.6940,   1.8563]],

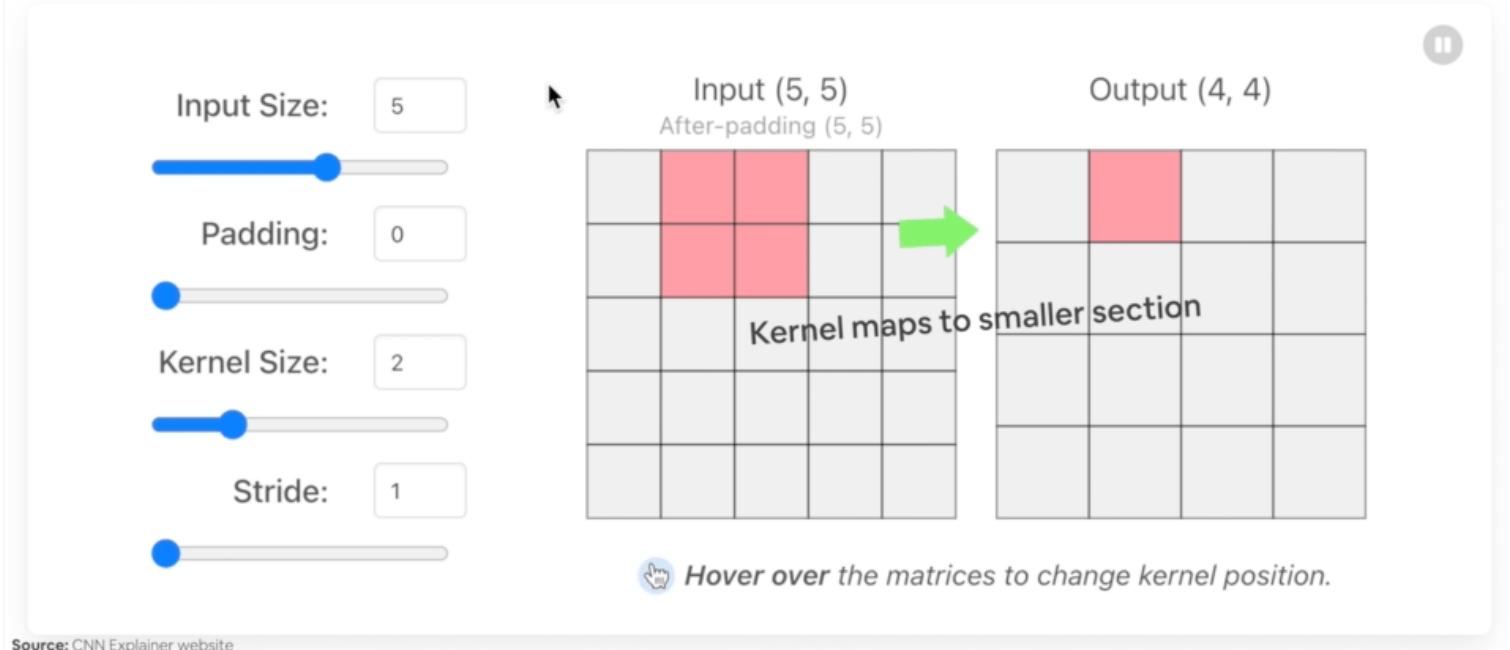
       [[-0.7978,   1.0261,   1.1465,   ... ,  1.2134,   0.9354,  -0.0780],
        [-1.4647,  -1.9571,   0.1017,   ... , -1.9986,  -0.7409,   0.7011],
        [-1.3938,   0.8466,  -1.7191,   ... , -1.1867,   0.1320,   0.3407],
        ... ,
        [ 0.8206,  -0.3745,   1.2499,   ... , -0.0676,   0.0385,   0.6335],
        [-0.5589,  -0.3393,   0.2347,   ... ,  2.1181,   2.4569,   1.3083],
        [-0.4092,   1.5199,   0.2401,   ... , -0.2558,   0.7870,   0.9924]]])

```

Let's create an example `nn.Conv2d()` with various parameters:

- `in_channels` (int) - Number of channels in the input image.
- `out_channels` (int) - Number of channels produced by the convolution.
- `kernel_size` (int or tuple) - Size of the convolving kernel/filter.
- `stride` (int or tuple, optional) - How big of a step the convolving kernel takes at a time. Default: 1.
- `padding` (int, tuple, str) - Padding added to all four sides of input. Default: 0.

## Understanding Hyperparameters of a Conv2d layer



*Example of what happens when you change the hyperparameters of a `nn.Conv2d()` layer.*

In [36]:







Out[36]:

```
tensor([[[ 1.5396,  0.0516,  0.6454,  ..., -0.3673,  0.8711,  0.4256],
       [ 0.3662,  1.0114, -0.5997,  ...,  0.8983,  0.2809, -0.2741],
       [ 1.2664, -1.4054,  0.3727,  ..., -0.3409,  1.2191, -0.0463],
       ...,
       [-0.1541,  0.5132, -0.3624,  ..., -0.2360, -0.4609, -0.0035],
       [ 0.2981, -0.2432,  1.5012,  ..., -0.6289, -0.7283, -0.5767],
       [-0.0386, -0.0781, -0.0388,  ...,  0.2842,  0.4228, -0.1802]],

      [[-0.2840, -0.0319, -0.4455,  ..., -0.7956,  1.5599, -1.2449],
       [ 0.2753, -0.1262, -0.6541,  ..., -0.2211,  0.1999, -0.8856],
       [-0.5404, -1.5489,  0.0249,  ..., -0.5932, -1.0913, -0.3849],
       ...,
       [ 0.3870, -0.4064, -0.8236,  ...,  0.1734, -0.4330, -0.4951],
```

```

[-0.1984, -0.6386,  1.0263,  ..., -0.9401, -0.0585, -0.7833],
[-0.6306, -0.2052, -0.3694,  ..., -1.3248,  0.2456, -0.7134]],

[[ 0.4414,  0.5100,  0.4846,  ..., -0.8484,  0.2638,  1.1258],
[ 0.8117,  0.3191, -0.0157,  ...,  1.2686,  0.2319,  0.5003],
[ 0.3212,  0.0485, -0.2581,  ...,  0.2258,  0.2587, -0.8804],
...,
[-0.1144, -0.1869,  0.0160,  ..., -0.8346,  0.0974,  0.8421],
[ 0.2941,  0.4417,  0.5866,  ..., -0.1224,  0.4814, -0.4799],
[ 0.6059, -0.0415, -0.2028,  ...,  0.1170,  0.2521, -0.4372]],

...,

[[[-0.2560, -0.0477,  0.6380,  ...,  0.6436,  0.7553, -0.7055],
[ 1.5595, -0.2209, -0.9486,  ..., -0.4876,  0.7754,  0.0750],
[-0.0797,  0.2471,  1.1300,  ...,  0.1505,  0.2354,  0.9576],
...,
[ 1.1065,  0.6839,  1.2183,  ...,  0.3015, -0.1910, -0.1902],
[-0.3486, -0.7173, -0.3582,  ...,  0.4917,  0.7219,  0.1513],
[ 0.0119,  0.1017,  0.7839,  ..., -0.3752, -0.8127, -0.1257]],

[[ 0.3841,  1.1322,  0.1620,  ...,  0.7010,  0.0109,  0.6058],
[ 0.1664,  0.1873,  1.5924,  ...,  0.3733,  0.9096, -0.5399],
[ 0.4094, -0.0861, -0.7935,  ..., -0.1285, -0.9932, -0.3013],
...,
[ 0.2688, -0.5630, -1.1902,  ...,  0.4493,  0.5404, -0.0103],
[ 0.0535,  0.4411,  0.5313,  ...,  0.0148, -1.0056,  0.3759],
[ 0.3031, -0.1590, -0.1316,  ..., -0.5384, -0.4271, -0.4876]],

[[[-1.1865, -0.7280, -1.2331,  ..., -0.9013, -0.0542, -1.5949],
[-0.6345, -0.5920,  0.5326,  ..., -1.0395, -0.7963, -0.0647],
[-0.1132,  0.5166,  0.2569,  ...,  0.5595, -1.6881,  0.9485],
...,
[-0.0254, -0.2669,  0.1927,  ..., -0.2917,  0.1088, -0.4807],
[-0.2609, -0.2328,  0.1404,  ..., -0.1325, -0.8436, -0.7524],
[-1.1399, -0.1751, -0.8705,  ...,  0.1589,  0.3377,  0.3493]]],  

grad_fn=<SqueezeBackward1>

```

If we try to pass a single image in, we get a shape mismatch error:

```

RuntimeError: Expected 4-dimensional input for 4-dimensional weight [10, 3, 3, 3], but got 3-dimensional input
of size [3, 64, 64] instead

```

**Note:** If you're running PyTorch 1.11.0+, this error won't occur.

This is because our `nn.Conv2d()` layer expects a 4-dimensional tensor as input with size `(N, C, H, W)` or `[batch_size, color_channels, height, width]`.

Right now our single image `test_image` only has a shape of `[color_channels, height, width]` or `[3, 64, 64]`.

We can fix this for a single image using `test_image.unsqueeze(dim=0)` to add an extra dimension for `N`.

In [37]:

```
torch.Size([1, 3, 64, 64])
```

Out[37]:

In [38]:

```
torch.Size([1, 10, 62, 62])
```

Out[38]:

Hmm, notice what happens to our shape (the same shape as the first layer of TinyVGG on [CNN Explainer](#)), we get different channel sizes as well as different pixel sizes.

What if we changed the values of `conv_layer`?

In [39]:









Out[39]:

```
torch.Size([1, 10, 30, 30])
```

Woah, we get another shape change.

Now our image is of shape [1, 10, 30, 30] (it will be different if you use different values) or [batch\_size=1, color\_channels=10, height=30, width=30].

What's going on here?

Behind the scenes, our `nn.Conv2d()` is compressing the information stored in the image.

It does this by performing operations on the input (our test image) against its internal parameters.

The goal of this is similar to all of the other neural networks we've been building.

Data goes in and the layers try to update their internal parameters (patterns) to lower the loss function thanks to some help of the optimizer.

The only difference is *how* the different layers calculate their parameter updates or in PyTorch terms, the operation present in the layer `forward()` method.

If we check out our `conv_layer_2.state_dict()` we'll find a similar weight and bias setup as we've seen before.

In [40]:

```
OrderedDict([('weight', tensor([[[[ 0.0883,  0.0958, -0.0271,  0.1061, -0.0253],
   [ 0.0233, -0.0562,  0.0678,  0.1018, -0.0847],
   [ 0.1004,  0.0216,  0.0853,  0.0156,  0.0557],
   [-0.0163,  0.0890,  0.0171, -0.0539,  0.0294],
   [-0.0532, -0.0135, -0.0469,  0.0766, -0.0911]],

   [[-0.0532, -0.0326, -0.0694,  0.0109, -0.1140],
   [ 0.1043, -0.0981,  0.0891,  0.0192, -0.0375],
   [ 0.0714,  0.0180,  0.0933,  0.0126, -0.0364],
   [ 0.0310, -0.0313,  0.0486,  0.1031,  0.0667],
   [-0.0505,  0.0667,  0.0207,  0.0586, -0.0704]],

   [[-0.1143, -0.0446, -0.0886,  0.0947,  0.0333],
   [ 0.0478,  0.0365, -0.0020,  0.0904, -0.0820],
   [ 0.0073, -0.0788,  0.0356, -0.0398,  0.0354],
   [-0.0241,  0.0958, -0.0684, -0.0689, -0.0689],
   [ 0.1039,  0.0385,  0.1111, -0.0953, -0.1145]]],

   [[[[-0.0903, -0.0777,  0.0468,  0.0413,  0.0959],
   [-0.0596, -0.0787,  0.0613, -0.0467,  0.0701],
   [-0.0274,  0.0661, -0.0897, -0.0583,  0.0352],
   [ 0.0244, -0.0294,  0.0688,  0.0785, -0.0837],
   [-0.0616,  0.1057, -0.0390, -0.0409, -0.1117]],

   [[-0.0661,  0.0288, -0.0152, -0.0838,  0.0027],
   [-0.0789, -0.0980, -0.0636, -0.1011, -0.0735],
   [ 0.1154,  0.0218,  0.0356, -0.1077, -0.0758],
   [-0.0384,  0.0181, -0.1016, -0.0498, -0.0691],
   [ 0.0003, -0.0430, -0.0080, -0.0782, -0.0793]],

   [[-0.0674, -0.0395, -0.0911,  0.0968, -0.0229],
   [ 0.0994,  0.0360, -0.0978,  0.0799, -0.0318],
   [-0.0443, -0.0958, -0.1148,  0.0330, -0.0252],
   [ 0.0450, -0.0948,  0.0857, -0.0848, -0.0199],
   [ 0.0241,  0.0596,  0.0932,  0.1052, -0.0916]]],

   [[[ 0.0291, -0.0497, -0.0127, -0.0864,  0.1052],
   [-0.0847,  0.0617,  0.0406,  0.0375, -0.0624],
   [ 0.1050,  0.0254,  0.0149, -0.1018,  0.0485],
   [-0.0173, -0.0529,  0.0992,  0.0257, -0.0639],
   [-0.0584, -0.0055,  0.0645, -0.0295, -0.0659]],

   [[-0.0395, -0.0863,  0.0412,  0.0894, -0.1087],
   [ 0.0268,  0.0597,  0.0209, -0.0411,  0.0603],
   [ 0.0607,  0.0432, -0.0203, -0.0306,  0.0124],
   [-0.0204, -0.0344,  0.0738,  0.0992, -0.0114],
   [-0.0259,  0.0017, -0.0069,  0.0278,  0.0324]],

   [[-0.1049, -0.0426,  0.0972,  0.0450, -0.0057],
   [-0.0696, -0.0706, -0.1034, -0.0376,  0.0390],
   [ 0.0736,  0.0533, -0.1021, -0.0694, -0.0182],
   [ 0.1117,  0.0167, -0.0299,  0.0478, -0.0440],
```

```

[-0.0747,  0.0843, -0.0525, -0.0231, -0.1149]],

[[[ 0.0773,  0.0875,  0.0421, -0.0805, -0.1140],
[-0.0938,  0.0861,  0.0554,  0.0972,  0.0605],
[ 0.0292, -0.0011, -0.0878, -0.0989, -0.1080],
[ 0.0473, -0.0567, -0.0232, -0.0665, -0.0210],
[-0.0813, -0.0754,  0.0383, -0.0343,  0.0713]],

[[-0.0370, -0.0847, -0.0204, -0.0560, -0.0353],
[-0.1099,  0.0646, -0.0804,  0.0580,  0.0524],
[ 0.0825, -0.0886,  0.0830, -0.0546,  0.0428],
[ 0.1084, -0.0163, -0.0009, -0.0266, -0.0964],
[ 0.0554, -0.1146,  0.0717,  0.0864,  0.1092]],

[[-0.0272, -0.0949,  0.0260,  0.0638, -0.1149],
[-0.0262, -0.0692, -0.0101, -0.0568, -0.0472],
[-0.0367, -0.1097,  0.0947,  0.0968, -0.0181],
[-0.0131, -0.0471, -0.1043, -0.1124,  0.0429],
[-0.0634, -0.0742, -0.0090, -0.0385, -0.0374]],

[[[ 0.0037, -0.0245, -0.0398, -0.0553, -0.0940],
[ 0.0968, -0.0462,  0.0306, -0.0401,  0.0094],
[ 0.1077,  0.0532, -0.1001,  0.0458,  0.1096],
[ 0.0304,  0.0774,  0.1138, -0.0177,  0.0240],
[-0.0803, -0.0238,  0.0855,  0.0592, -0.0731]],

[[-0.0926, -0.0789, -0.1140, -0.0891, -0.0286],
[ 0.0779,  0.0193, -0.0878, -0.0926,  0.0574],
[-0.0859, -0.0142,  0.0554, -0.0534, -0.0126],
[-0.0101, -0.0273, -0.0585, -0.1029, -0.0933],
[-0.0618,  0.1115, -0.0558, -0.0775,  0.0280]],

[[ 0.0318,  0.0633,  0.0878,  0.0643, -0.1145],
[ 0.0102,  0.0699, -0.0107, -0.0680,  0.1101],
[-0.0432, -0.0657, -0.1041,  0.0052,  0.0512],
[ 0.0256,  0.0228, -0.0876, -0.1078,  0.0020],
[ 0.1053,  0.0666, -0.0672, -0.0150, -0.0851]],

[[[-0.0557,  0.0209,  0.0629,  0.0957, -0.1060],
[ 0.0772, -0.0814,  0.0432,  0.0977,  0.0016],
[ 0.1051, -0.0984, -0.0441,  0.0673, -0.0252],
[-0.0236, -0.0481,  0.0796,  0.0566,  0.0370],
[-0.0649, -0.0937,  0.0125,  0.0342, -0.0533]],

[[-0.0323,  0.0780,  0.0092,  0.0052, -0.0284],
[-0.1046, -0.1086, -0.0552, -0.0587,  0.0360],
[-0.0336, -0.0452,  0.1101,  0.0402,  0.0823],
[-0.0559, -0.0472,  0.0424, -0.0769, -0.0755],
[-0.0056, -0.0422, -0.0866,  0.0685,  0.0929]],

[[ 0.0187, -0.0201, -0.1070, -0.0421,  0.0294],

```

```
[ 0.0544, -0.0146, -0.0457,  0.0643, -0.0920],  
[ 0.0730, -0.0448,  0.0018, -0.0228,  0.0140],  
[-0.0349,  0.0840, -0.0030,  0.0901,  0.1110],  
[-0.0563, -0.0842,  0.0926,  0.0905, -0.0882]]],  
  
[[[-0.0089, -0.1139, -0.0945,  0.0223,  0.0307],  
[ 0.0245, -0.0314,  0.1065,  0.0165, -0.0681],  
[-0.0065,  0.0277,  0.0404, -0.0816,  0.0433],  
[-0.0590, -0.0959, -0.0631,  0.1114,  0.0987],  
[ 0.1034,  0.0678,  0.0872, -0.0155, -0.0635]],  
  
[[ 0.0577, -0.0598, -0.0779, -0.0369,  0.0242],  
[ 0.0594, -0.0448, -0.0680,  0.0156, -0.0681],  
[-0.0752,  0.0602, -0.0194,  0.1055,  0.1123],  
[ 0.0345,  0.0397,  0.0266,  0.0018, -0.0084],  
[ 0.0016,  0.0431,  0.1074, -0.0299, -0.0488]],  
  
[[-0.0280, -0.0558,  0.0196,  0.0862,  0.0903],  
[ 0.0530, -0.0850, -0.0620, -0.0254, -0.0213],  
[ 0.0095, -0.1060,  0.0359, -0.0881, -0.0731],  
[-0.0960,  0.1006, -0.1093,  0.0871, -0.0039],  
[-0.0134,  0.0722, -0.0107,  0.0724,  0.0835]],  
  
[[[-0.1003,  0.0444,  0.0218,  0.0248,  0.0169],  
[ 0.0316, -0.0555, -0.0148,  0.1097,  0.0776],  
[-0.0043, -0.1086,  0.0051, -0.0786,  0.0939],  
[-0.0701, -0.0083, -0.0256,  0.0205,  0.1087],  
[ 0.0110,  0.0669,  0.0896,  0.0932, -0.0399]],  
  
[[ -0.0258,  0.0556, -0.0315,  0.0541, -0.0252],  
[ -0.0783,  0.0470,  0.0177,  0.0515,  0.1147],  
[ 0.0788,  0.1095,  0.0062, -0.0993, -0.0810],  
[ -0.0717, -0.1018, -0.0579, -0.1063, -0.1065],  
[ -0.0690, -0.1138, -0.0709,  0.0440,  0.0963]],  
  
[[ -0.0343, -0.0336,  0.0617, -0.0570, -0.0546],  
[ 0.0711, -0.1006,  0.0141,  0.1020,  0.0198],  
[ 0.0314, -0.0672, -0.0016,  0.0063,  0.0283],  
[ 0.0449,  0.1003, -0.0881,  0.0035, -0.0577],  
[ -0.0913, -0.0092, -0.1016,  0.0806,  0.0134]],  
  
[[[-0.0622,  0.0603, -0.1093, -0.0447, -0.0225],  
[-0.0981, -0.0734, -0.0188,  0.0876,  0.1115],  
[ 0.0735, -0.0689, -0.0755,  0.1008,  0.0408],  
[ 0.0031,  0.0156, -0.0928, -0.0386,  0.1112],  
[-0.0285, -0.0058, -0.0959, -0.0646, -0.0024]],  
  
[[ -0.0717, -0.0143,  0.0470, -0.1130,  0.0343],  
[ -0.0763, -0.0564,  0.0443,  0.0918, -0.0316],  
[ -0.0474, -0.1044, -0.0595, -0.1011, -0.0264],  
[ 0.0236, -0.1082,  0.1008,  0.0724, -0.1130],
```

```
[-0.0552,  0.0377, -0.0237, -0.0126, -0.0521]],

[[[ 0.0927, -0.0645,  0.0958,  0.0075,  0.0232],
   [ 0.0901, -0.0190, -0.0657, -0.0187,  0.0937],
   [-0.0857,  0.0262, -0.1135,  0.0605,  0.0427],
   [ 0.0049,  0.0496,  0.0001,  0.0639, -0.0914],
   [-0.0170,  0.0512,  0.1150,  0.0588, -0.0840]]],

[[[[ 0.0888, -0.0257, -0.0247, -0.1050, -0.0182],
    [ 0.0817,  0.0161, -0.0673,  0.0355, -0.0370],
    [ 0.1054, -0.1002, -0.0365, -0.1115, -0.0455],
    [ 0.0364,  0.1112,  0.0194,  0.1132,  0.0226],
    [ 0.0667,  0.0926,  0.0965, -0.0646,  0.1062]],

   [[ 0.0699, -0.0540, -0.0551, -0.0969,  0.0290],
    [-0.0936,  0.0488,  0.0365, -0.1003,  0.0315],
    [-0.0094,  0.0527,  0.0663, -0.1148,  0.1059],
    [ 0.0968,  0.0459, -0.1055, -0.0412, -0.0335],
    [-0.0297,  0.0651,  0.0420,  0.0915, -0.0432]],

   [[[ 0.0389,  0.0411, -0.0961, -0.1120, -0.0599],
     [ 0.0790, -0.1087, -0.1005,  0.0647,  0.0623],
     [ 0.0950, -0.0872, -0.0845,  0.0592,  0.1004],
     [ 0.0691,  0.0181,  0.0381,  0.1096, -0.0745],
     [-0.0524,  0.0808, -0.0790, -0.0637,  0.0843]]]]]), ('bias', tensor([ 0.0364,  0.0373,
-0.0489, -0.0016,  0.1057, -0.0693,  0.0009,  0.0549,
-0.0797,  0.1121]))])
```

Look at that! A bunch of random numbers for a weight and bias tensor.

The shapes of these are manipulated by the inputs we passed to `nn.Conv2d()` when we set it up.

Let's check them out.

In [41]:

```
conv_layer_2 weight shape:  
torch.Size([10, 3, 5, 5]) -> [out_channels=10, in_channels=3, kernel_size=5, kernel_size=5]  
  
conv_layer_2 bias shape:  
torch.Size([10]) -> [out_channels=10]
```

**Question:** What should we set the parameters of our `nn.Conv2d()` layers?

That's a good one. But similar to many other things in machine learning, the values of these aren't set in stone (and recall, because these values are ones we can set ourselves, they're referred to as "**hyperparameters**").

The best way to find out is to try out different values and see how they effect your model's performance.

Or even better, find a working example on a problem similar to yours (like we've done with TinyVGG) and copy it.

We're working with a different of layer here to what we've seen before.

But the premise remains the same: start with random numbers and update them to better represent the data.

## 7.2 Stepping through `nn.MaxPool2d()`

Now let's check out what happens when we move data through `nn.MaxPool2d()`.

In [42]:



```
Test image original shape: torch.Size([3, 64, 64])
Test image with unsqueezed dimension: torch.Size([1, 3, 64, 64])
Shape after going through conv_layer(): torch.Size([1, 10, 62, 62])
Shape after going through conv_layer() and max_pool_layer(): torch.Size([1, 10, 31, 31])
```

Notice the change in the shapes of what's happening in and out of a `nn.MaxPool2d()` layer.

The `kernel_size` of the `nn.MaxPool2d()` layer will effects the size of the output shape.

In our case, the shape halves from a `62x62` image to `31x31` image.

Let's see this work with a smaller tensor.

In [43]:





```
Random tensor:  
tensor([[[[0.3367, 0.1288],  
        [0.2345, 0.2303]]]])  
Random tensor shape: torch.Size([1, 1, 2, 2])  
  
Max pool tensor:  
tensor([[[[0.3367]]]]) <- this is the maximum value from random_tensor
```

```
Max pool tensor shape: torch.Size([1, 1, 1, 1])
```

Notice the final two dimensions between `random_tensor` and `max_pool_tensor`, they go from `[2, 2]` to `[1, 1]`.

In essence, they get halved.

And the change would be different for different values of `kernel_size` for `nn.MaxPool2d()`.

Also notice the value leftover in `max_pool_tensor` is the **maximum** value from `random_tensor`.

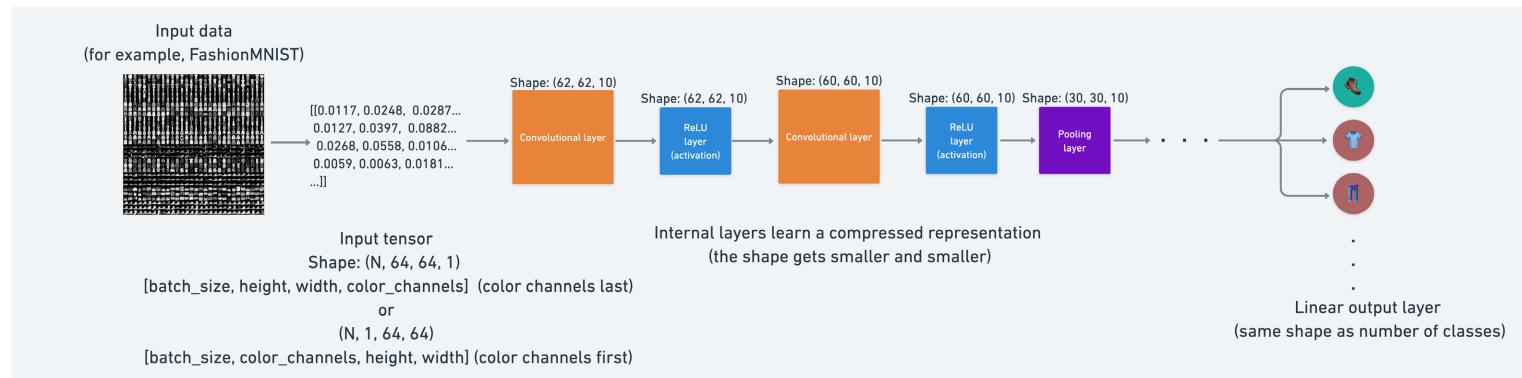
What's happening here?

This is another important piece of the puzzle of neural networks.

Essentially, **every layer in a neural network is trying to compress data from higher dimensional space to lower dimensional space.**

In other words, take a lot of numbers (raw data) and learn patterns in those numbers, patterns that are predictive whilst also being *smaller* in size than the original values.

From an artificial intelligence perspective, you could consider the whole goal of a neural network to *compress* information.



This means, that from the point of view of a neural network, intelligence is compression.

This is the idea of the use of a `nn.MaxPool2d()` layer: take the maximum value from a portion of a tensor and disregard the rest.

In essence, lowering the dimensionality of a tensor whilst still retaining a (hopefully) significant portion of the information.

It is the same story for a `nn.Conv2d()` layer.

Except instead of just taking the maximum, the `nn.Conv2d()` performs a convolutional operation on the data (see this in action on the [CNN Explainer webpage](#)).

**Exercise:** What do you think the `nn.AvgPool2d()` layer does? Try making a random tensor like we did above and passing it through. Check the input and output shapes as well as the input and output values.

**Extra-curriculum:** Lookup "most common convolutional neural networks", what architectures do you find? Are any of them contained within the `torchvision.models` library? What do you think you could do with these?

## 7.3 Setup a loss function and optimizer for `model_2`

We've stepped through the layers in our first CNN enough.

But remember, if something still isn't clear, try starting small.

Pick a single layer of a model, pass some data through it and see what happens.

Now it's time to move forward and get to training!

Let's setup a loss function and an optimizer.

We'll use the functions as before, `nn.CrossEntropyLoss()` as the loss function (since we're working with multi-class classification data).

And `torch.optim.SGD()` as the optimizer to optimize `model_2.parameters()` with a learning rate of `0.1`.

In [44]:

## 7.4 Training and testing `model_2` using our training and test functions

Loss and optimizer ready!

Time to train and test.

We'll use our `train_step()` and `test_step()` functions we created before.

We'll also measure the time to compare it to our other models.

In [45]:











```
0% | 0 / 3 [00:00<?, ?it/s]
Epoch: 0
-----
Train loss: 0.59411 | Train accuracy: 78.41%
Test loss: 0.39967 | Test accuracy: 85.70%

Epoch: 1
-----
Train loss: 0.36450 | Train accuracy: 86.81%
Test loss: 0.34607 | Test accuracy: 87.48%

Epoch: 2
-----
Train loss: 0.32553 | Train accuracy: 88.33%
Test loss: 0.32664 | Test accuracy: 88.23%
```

Train time on cuda: 21.099 seconds

Woah! Looks like the convolutional and max pooling layers helped improve performance a little.

Let's evaluate `model_2`'s results with our `eval_model()` function.

In [46]:

Out[46]:

```
{'model_name': 'FashionMNISTModelV2',
'model_loss': 0.32664385437965393,
'model_acc': 88.22883386581469}
```

## 8. Compare model results and training time

We've trained three different models.

1. `model_0` - our baseline model with two `nn.Linear()` layers.
2. `model_1` - the same setup as our baseline model except with `nn.ReLU()` layers in between the `nn.Linear()` layers.
3. `model_2` - our first CNN model that mimics the TinyVGG architecture on the CNN Explainer website.

This is a regular practice in machine learning.

Building multiple models and performing multiple training experiments to see which performs best.

Let's combine our model results dictionaries into a DataFrame and find out.

In [47]:

Out[47]:

	model_name	model_loss	model_acc
0	FashionMNISTModelV0	0.476639	83.426518
1	FashionMNISTModelV1	0.685001	75.019968
2	FashionMNISTModelV2	0.326644	88.228834

Nice!

We can add the training time values too.

In [48]:



Out[48]:

	model_name	model_loss	model_acc	training_time
0	FashionMNISTModelV0	0.476639	83.426518	14.974887
1	FashionMNISTModelV1	0.685001	75.019968	16.942553
2	FashionMNISTModelV2	0.326644	88.228834	21.098929

It looks like our CNN (`FashionMNISTModelV2`) model performed the best (lowest loss, highest accuracy) but had the longest training time.

And our baseline model (`FashionMNISTModelV0`) performed better than `model_1` (`FashionMNISTModelV1`) but took longer to train (this is likely because we used a CPU to train `model_0` but a GPU to train `model_1`).

The tradeoffs here are known as the **performance-speed** tradeoff.

Generally, you get better performance out of a larger, more complex model (like we did with `model_2`).

However, this performance increase often comes at a sacrifice of training speed and inference speed.

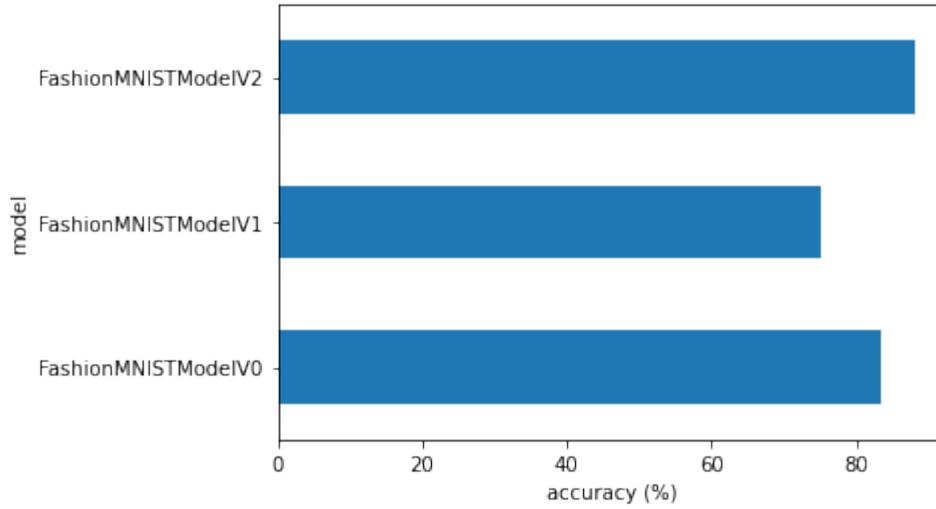
**Note:** The training times you get will be very dependant on the hardware you use.

Generally, the more CPU cores you have, the faster your models will train on CPU. And similar for GPUs.

Newer hardware (in terms of age) will also often train models faster due to incorporating technology advances.

How about we get visual?

In [49]:



## 9. Make and evaluate random predictions with best model

Alright, we've compared our models to each other, let's further evaluate our best performing model, `model_2`.

To do so, let's create a function `make_predictions()` where we can pass the model and some data for it to predict on.

In [50]:











In [51]:



```
Test sample image shape: torch.Size([1, 28, 28])
Test sample label: 5 (Sandal)
```

And now we can use our `make_predictions()` function to predict on `test_samples`.

In [52]:

Out[52]:

```
tensor([[2.3550e-07, 1.7185e-08, 4.6618e-07, 6.1371e-08, 5.1185e-08, 9.9957e-01,
        3.7702e-07, 1.5924e-05, 3.7681e-05, 3.7831e-04],
       [7.3275e-02, 6.7410e-01, 3.7231e-03, 8.8129e-02, 1.0114e-01, 6.9186e-05,
        5.8674e-02, 4.2595e-04, 3.8635e-04, 7.1354e-05]])
```

Excellent!

And now we can go from prediction probabilities to prediction labels by taking the `torch.argmax()` of the output of the `torch.softmax()` activation function.

In [53]:

Out[53]:

```
tensor([5, 1, 7, 4, 3, 0, 4, 7, 1])
```

In [54]:

Out[54]:

```
([5, 1, 7, 4, 3, 0, 4, 7, 1], tensor([5, 1, 7, 4, 3, 0, 4, 7, 1]))
```

Now our predicted classes are in the same format as our test labels, we can compare.

Since we're dealing with image data, let's stay true to the data explorer's motto.

"Visualize, visualize, visualize!"

In [55]:











Well, well, well, doesn't that look good!

Not bad for a couple dozen lines of PyTorch code!

## 10. Making a confusion matrix for further prediction evaluation

There are many [different evaluation metrics](#) we can use for classification problems.

One of the most visual is a [confusion matrix](#).

A confusion matrix shows you where your classification model got confused between predictions and true labels.

To make a confusion matrix, we'll go through three steps:

1. Make predictions with our trained model, `model_2` (a confusion matrix compares predictions to true labels).
2. Make a confusion matrix using `torch.ConfusionMatrix`.
3. Plot the confusion matrix using `mlxtend.plotting.plot_confusion_matrix()`.

Let's start by making predictions with our trained model.

In [56]:





Making predictions: 0% | 0/313 [00:00<?, ?it/s]

Wonderful!

Now we've got predictions, let's go through steps 2 & 3:

1. Make a confusion matrix using `torchmetrics.ConfusionMatrix`.
2. Plot the confusion matrix using `mlxtend.plotting.plot_confusion_matrix()`.

First we'll need to make sure we've got `torchmetrics` and `mlxtend` installed (these two libraries will help us make and visual a confusion matrix).

**Note:** If you're using Google Colab, the default version of `mlxtend` installed is 0.14.0 (as of March 2022), however, for the parameters of the `plot_confusion_matrix()` function we'd like use, we need 0.19.0 or higher.

In [57]:



```
mlxtend version: 0.19.0
```

To plot the confusion matrix, we need to make sure we've got and `mlxtend` version of 0.19.0 or higher.

In [58]:

0.19.0

`torchmetrics` and `mlxtend` installed, let's make a confusion matrix!

First we'll create a `torchmetrics.ConfusionMatrix` instance telling it how many classes we're dealing with by setting `num_classes=len(class_names)`.

Then we'll create a confusion matrix (in tensor format) by passing our instance our model's predictions (`preds=y_pred_tensor`) and targets (`target=test_data.targets`).

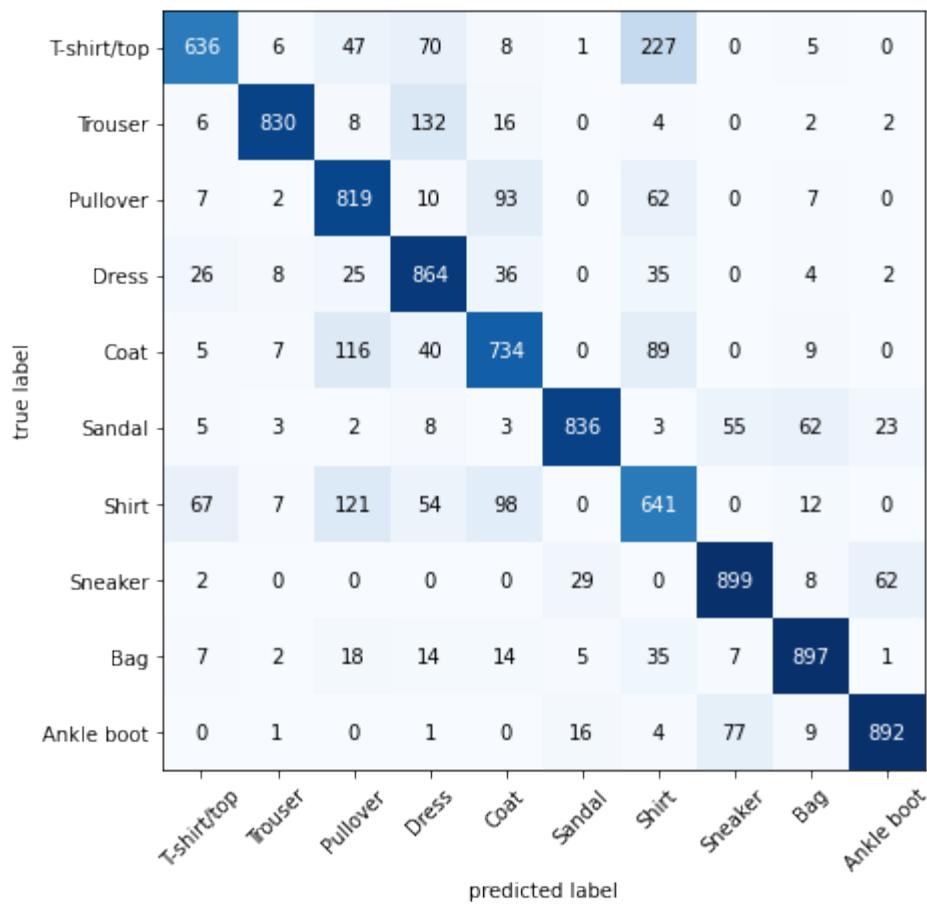
Finally we can plot our confision matrix using the `plot_confusion_matrix()` function from `mlxtend.plotting`.

In [59]:









Woah! Doesn't that look good?

We can see our model does fairly well since most of the dark squares are down the diagonal from top left to bottom right (and ideal model will have only values in these squares and 0 everywhere else).

The model gets most "confused" on classes that are similar, for example predicting "Pullover" for images that are actually labelled "Shirt".

And the same for predicting "Shirt" for classes that are actually labelled "T-shirt/top".

This kind of information is often more helpful than a single accuracy metric because it tells us where a model is getting things wrong.

It also hints at why the model may be getting certain things wrong.

It's understandable the model sometimes predicts "Shirt" for images labelled "T-shirt/top".

We can use this kind of information to further inspect our models and data to see how it could be improved.

**Exercise:** Use the trained `model_2` to make predictions on the test FashionMNIST dataset. Then plot some predictions where the model was wrong alongside what the label of the image should've been. After visualizing these predictions do you think it's more of a modelling error or a data error? As in, could the model do better or are the

labels of the data too close to each other (e.g. a "Shirt" label is too close to "T-shirt/top")?

## 11. Save and load best performing model

Let's finish this section off by saving and loading in our best performing model.

Recall from [notebook 01](#) we can save and load a PyTorch model using a combination of:

- `torch.save` - a function to save a whole PyTorch model or a model's `state_dict()`.
- `torch.load` - a function to load in a saved PyTorch object.
- `torch.nn.Module.load_state_dict()` - a function to load a saved `state_dict()` into an existing model instance.

You can see more of these three in the [PyTorch saving and loading models documentation](#).

For now, let's save our `model_2`'s `state_dict()` then load it back in and evaluate it to make sure the save and load went correctly.

In [60]:





```
Saving model to: models/03_pytorch_computer_vision_model_2.pth
```

Now we've got a saved model `state_dict()` we can load it back in using a combination of `load_state_dict()` and `torch.load()`.

Since we're using `load_state_dict()`, we'll need to create a new instance of `FashionMNISTModelV2()` with the same input parameters as our saved model `state_dict()`.

In [61]:







And now we've got a loaded model we can evaluate it with `eval_model()` to make sure its parameters work similarly to `model_2` prior to saving.

In [62]:

Out[62]:

```
{'model_name': 'FashionMNISTModelV2',
'model_loss': 0.32664385437965393,
'model_acc': 88.22883386581469}
```

Do these results look the same as `model_2_results`?

In [63]:

Out[63]:

```
{'model_name': 'FashionMNISTModelV2',
'model_loss': 0.32664385437965393,
'model_acc': 88.22883386581469}
```

We can find out if two tensors are close to each other using `torch.isclose()` and passing in a tolerance level of closeness via the parameters `atol` (absolute tolerance) and `rtol` (relative tolerance).

If our model's results are close, the output of `torch.isclose()` should be true.

In [64]:





Out[64]:

```
tensor(True)
```

## Exercises

All of the exercises are focused on practicing the code in the sections above.

You should be able to complete them by referencing each section or by following the resource(s) linked.

All exercises should be completed using [device-agnostic code](#).

### Resources:

- [Exercise template notebook for 03](#)
- [Example solutions notebook for 03](#) (try the exercises *before* looking at this)
  1. What are 3 areas in industry where computer vision is currently being used?
  2. Search "what is overfitting in machine learning" and write down a sentence about what you find.
  3. Search "ways to prevent overfitting in machine learning", write down 3 of the things you find and a sentence about each. **Note:** there are lots of these, so don't worry too much about all of them, just pick 3 and start with those.
  4. Spend 20-minutes reading and clicking through the [CNN Explainer website](#).
    - Upload your own example image using the "upload" button and see what happens in each layer of a CNN as your image passes through it.
  5. Load the `torchvision.datasets.MNIST()` train and test datasets.
  6. Visualize at least 5 different samples of the MNIST training dataset.
  7. Turn the MNIST train and test datasets into dataloaders using `torch.utils.data.DataLoader`, set the `batch_size=32`.
  8. Recreate `model_2` used in this notebook (the same model from the [CNN Explainer website](#), also known as TinyVGG) capable of fitting on the MNIST dataset.
  9. Train the model you built in exercise 8. on CPU and GPU and see how long it takes on each.
  10. Make predictions using your trained model and visualize at least 5 of them comparing the prediction to the target label.
  11. Plot a confusion matrix comparing your model's predictions to the truth labels.
  12. Create a random tensor of shape `[1, 3, 64, 64]` and pass it through a `nn.Conv2d()` layer with various

hyperparameter settings (these can be any settings you choose), what do you notice if the `kernel_size` parameter goes up and down?

13. Use a model similar to the trained `model_2` from this notebook to make predictions on the test `torchvision.datasets.FashionMNIST` dataset.

- Then plot some predictions where the model was wrong alongside what the label of the image should've been.
- After visualizing these predictions do you think it's more of a modelling error or a data error?
- As in, could the model do better or are the labels of the data too close to each other (e.g. a "Shirt" label is too close to "T-shirt/top")?

## Extra-curriculum

- **Watch:** [MIT's Introduction to Deep Computer Vision](#) lecture. This will give you a great intuition behind convolutional neural networks.
- Spend 10-minutes clicking through the different options of the [PyTorch vision library](#), what different modules are available?
- Lookup "most common convolutional neural networks", what architectures do you find? Are any of them contained within the `torchvision.models` library? What do you think you could do with these?
- For a large number of pretrained PyTorch computer vision models as well as many different extensions to PyTorch's computer vision functionalities check out the [PyTorch Image Models library](#) `timm` (Torch Image Models) by Ross Wightman.

Previous

[02. PyTorch Neural Network Classification](#)

Next

[04. PyTorch Custom Datasets](#)



[View Source Code](#) | [View Slides](#) | [Watch Video Walkthrough](#)

## 04. PyTorch Custom Datasets

In the last notebook, [notebook 03](#), we looked at how to build computer vision models on an in-built dataset in PyTorch (FashionMNIST).

The steps we took are similar across many different problems in machine learning.

Find a dataset, turn the dataset into numbers, build a model (or find an existing model) to find patterns in those numbers that can be used for prediction.

PyTorch has many built-in datasets used for a wide number of machine learning benchmarks, however, you'll often want to use your own **custom dataset**.

### What is a custom dataset?

A **custom dataset** is a collection of data relating to a specific problem you're working on.

In essence, a **custom dataset** can be comprised of almost anything.

For example, if we were building a food image classification app like [Nutrify](#), our custom dataset might be images of food.

Or if we were trying to build a model to classify whether or not a text-based review on a website was positive or negative, our custom dataset might be examples of existing customer reviews and their ratings.

Or if we were trying to build a sound classification app, our custom dataset might be sound samples alongside their sample labels.

Or if we were trying to build a recommendation system for customers purchasing things on our website, our custom

dataset might be examples of products other people have bought.

# PyTorch Domain Libraries

“Is this a photo of pizza, steak or sushi?”



**TorchVision**

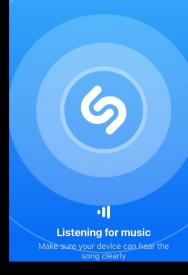
“Are these reviews positive or negative?”

 **martinhk** Today at 9:20 PM  
@mrdbourke Just started your tensorflow course a few days ago! I took quite a few ML/DL courses online and I would like to say it's by far the best deep learning course I have ever had! I like your way of teaching difficult topics and I learnt a lot more coding along with you!

 **iwatts** Yesterday at 10:20 PM  
Thanks TFM I've landed a job - in the UK working for a Global Marketing company as a Senior Analytics Developer. Thank you @Andrei Neagoie and @mrdbourke Couldn't have got there without you.

**TorchText**

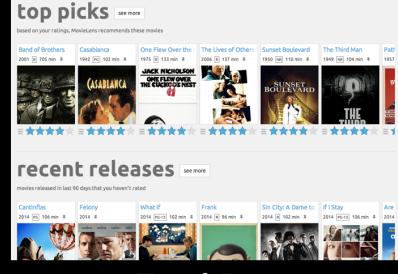
“What song is playing?”



Different domain libraries contain data loading functions for different data sources

**TorchAudio**

“How do we recommend similar products?”



**TorchRec**  
Source: [movielens.org](http://movielens.org)

PyTorch includes many existing functions to load in various custom datasets in the [TorchVision](#), [TorchText](#), [TorchAudio](#) and [TorchRec](#) domain libraries.

But sometimes these existing functions may not be enough.

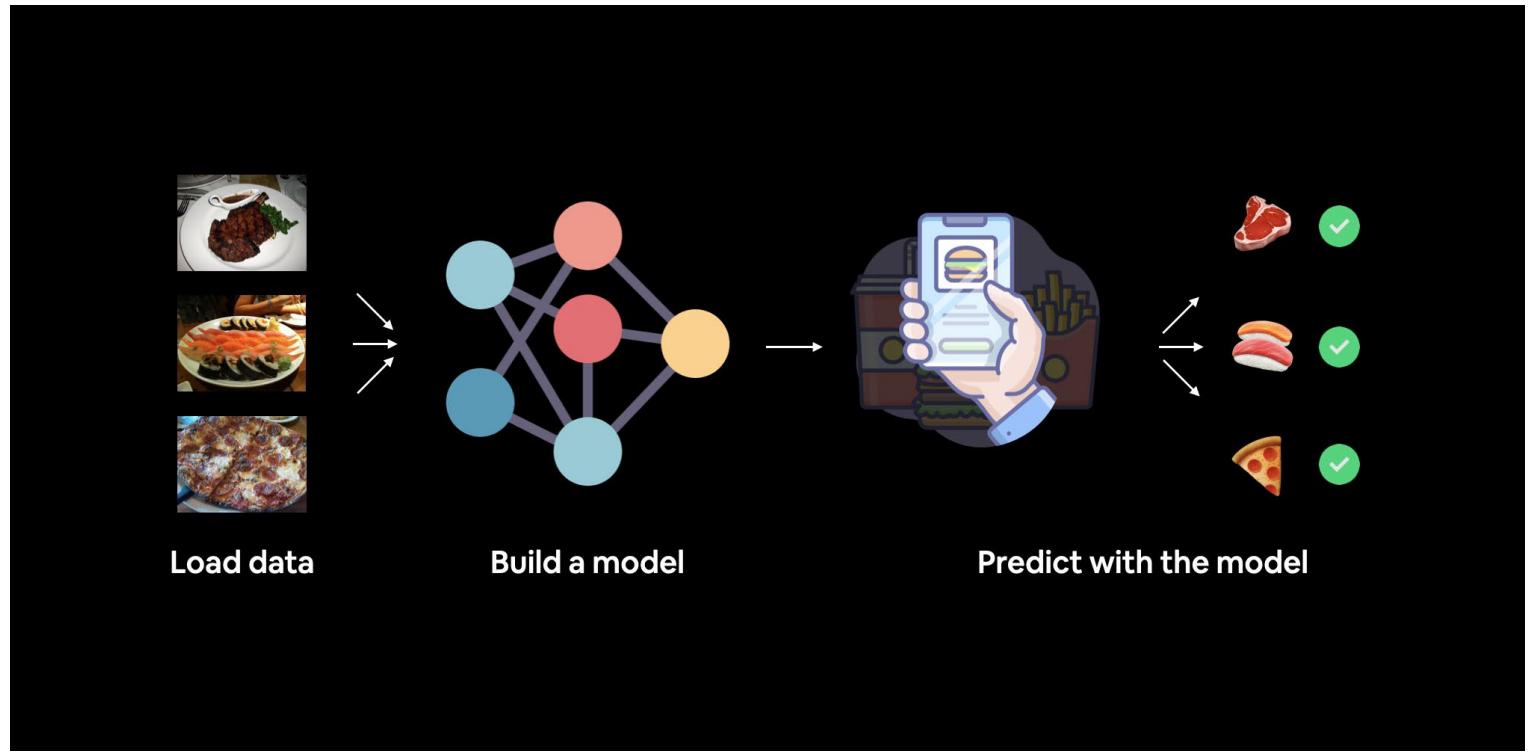
In that case, we can always subclass `torch.utils.data.Dataset` and customize it to our liking.

## What we're going to cover

We're going to be applying the PyTorch Workflow we covered in [notebook 01](#) and [notebook 02](#) to a computer vision problem.

But instead of using an in-built PyTorch dataset, we're going to be using our own dataset of pizza, steak and sushi images.

The goal will be to load these images and then build a model to train and predict on them.



What we're going to build. We'll use `torchvision.datasets` as well as our own custom `Dataset` class to load in images of food and then we'll build a PyTorch computer vision model to hopefully be able to classify them.

Specifically, we're going to cover:

Topic	Contents
<b>0. Importing PyTorch and setting up device-agnostic code</b>	Let's get PyTorch loaded and then follow best practice to setup our code to be device-agnostic.
<b>1. Get data</b>	We're going to be using our own <b>custom dataset</b> of pizza, steak and sushi images.
<b>2. Become one with the data (data preparation)</b>	At the beginning of any new machine learning problem, it's paramount to understand the data you're working with. Here we'll take some steps to figure out what data we have.
<b>3. Transforming data</b>	Often, the data you get won't be 100% ready to use with a machine learning model, here we'll look at some steps we can take to <i>transform</i> our images so they're ready to be used with a model.
<b>4. Loading data with <code>ImageFolder</code> (option 1)</b>	PyTorch has many in-built data loading functions for common types of data. <code>ImageFolder</code> is helpful if our images are in standard image classification format.
<b>5. Loading image data with a custom <code>Dataset</code></b>	What if PyTorch didn't have an in-built function to load data with? This is where we can build our own custom subclass of <code>torch.utils.data.Dataset</code> .
<b>6. Other forms of transforms</b>	Data augmentation is a common technique for expanding the diversity of your training data. Here

**(data augmentation)**

we'll explore some of `torchvision`'s in-built data augmentation functions.

**7. Model 0: TinyVGG without data augmentation**

By this stage, we'll have our data ready, let's build a model capable of fitting it. We'll also create some training and testing functions for training and evaluating our model.

**8. Exploring loss curves**

Loss curves are a great way to see how your model is training/improving over time. They're also a good way to see if your model is **underfitting** or **overfitting**.

**9. Model 1: TinyVGG with data augmentation**

By now, we've tried a model *without*, how about we try one *with* data augmentation?

**10. Compare model results**

Let's compare our different models' loss curves and see which performed better and discuss some options for improving performance.

**11. Making a prediction on a custom image**

Our model is trained to on a dataset of pizza, steak and sushi images. In this section we'll cover how to use our trained model to predict on an image *outside* of our existing dataset.

## Where can you get help?

All of the materials for this course [live on GitHub](#).

If you run into trouble, you can ask a question on the course [GitHub Discussions page](#) there too.

And of course, there's the [PyTorch documentation](#) and [PyTorch developer forums](#), a very helpful place for all things PyTorch.

## 0. Importing PyTorch and setting up device-agnostic code

In [2]:

Out[2]:

'1.11.0'

And now let's follow best practice and setup device-agnostic code.

**Note:** If you're using Google Colab, and you don't have a GPU turned on yet, it's now time to turn one on via Runtime -> Change runtime type -> Hardware accelerator -> GPU . If you do this, your runtime will likely reset and you'll have to run all of the cells above by going Runtime -> Run before .

In [3]:

Out[3]:

'cuda'

## 1. Get data

First thing's first we need some data.

And like any good cooking show, some data has already been prepared for us.

We're going to start small.

Because we're not looking to train the biggest model or use the biggest dataset yet.

Machine learning is an iterative process, start small, get something working and increase when necessary.

The data we're going to be using is a subset of the [Food101 dataset](#).

Food101 is popular computer vision benchmark as it contains 1000 images of 101 different kinds of foods, totaling 101,000 images (75,750 train and 25,250 test).

Can you think of 101 different foods?

Can you think of a computer program to classify 101 foods?

I can.

A machine learning model!

Specifically, a PyTorch computer vision model like we covered in [notebook 03](#).

Instead of 101 food classes though, we're going to start with 3: pizza, steak and sushi.

And instead of 1,000 images per class, we're going to start with a random 10% (start small, increase when necessary).

If you'd like to see where the data came from you see the following resources:

- [Original Food101 dataset and paper website](#).
- `torchvision.datasets.Food101` - the version of the data I downloaded for this notebook.
- `extras/04_custom_data_creation.ipynb` - a notebook I used to format the Food101 dataset to use for this notebook.
- `data/pizza_steak_sushi.zip` - the zip archive of pizza, steak and sushi images from Food101, created with the notebook linked above.

Let's write some code to download the formatted data from GitHub.

**Note:** The dataset we're about to use has been pre-formatted for what we'd like to use it for. However, you'll often have to format your own datasets for whatever problem you're working on. This is a regular practice in the machine learning world.

In [2]:









data/pizza\_steak\_sushi directory exists.

## 2. Become one with the data (data preparation)

Dataset downloaded!

Time to become one with it.

This is another important step before building a model.

As Abraham Lossfunction said...



Daniel Bourke  
@mrdbourke

...

**“If I had 8 hours to build a machine learning model, I’d spend the first 6 hours preparing my dataset.”**

**- Abraham Lossfunction**

12:35 PM · Nov 4, 2021 · Twitter Web App

*Data preparation is paramount. Before building a model, become one with the data. Ask: What am I trying to do here? Source: [@mrdbourke Twitter](#).*

What's inspecting the data and becoming one with it?

Before starting a project or building any kind of model, it's important to know what data you're working with.

In our case, we have images of pizza, steak and sushi in standard image classification format.

Image classification format contains separate classes of images in separate directories titled with a particular class name.

For example, all images of `pizza` are contained in the `pizza/` directory.

This format is popular across many different image classification benchmarks, including [ImageNet](#) (of the most popular computer vision benchmark datasets).

You can see an example of the storage format below, the images numbers are arbitrary.

```

pizza_steak_sushi/ <- overall dataset folder
  train/ <- training images
    pizza/ <- class name as folder name
      image01.jpeg
      image02.jpeg
      ...
    steak/
      image24.jpeg
      image25.jpeg
      ...
    sushi/
      image37.jpeg
      ...
  test/ <- testing images
    pizza/
      image101.jpeg
      image102.jpeg
      ...
    steak/
      image154.jpeg
      image155.jpeg
      ...
    sushi/
      image167.jpeg
      ...
  ...

```

The goal will be to **take this data storage structure and turn it into a dataset usable with PyTorch**.

**Note:** The structure of the data you work with will vary depending on the problem you're working on. But the premise still remains: become one with the data, then find a way to best turn it into a dataset compatible with PyTorch.

We can inspect what's in our data directory by writing a small helper function to walk through each of the subdirectories and count the files present.

To do so, we'll use Python's in-built `os.walk()`.

In [5]:





In [6]:

```
There are 2 directories and 0 images in 'data\pizza_steak_sushi'.
There are 3 directories and 0 images in 'data\pizza_steak_sushi\test'.
There are 0 directories and 25 images in 'data\pizza_steak_sushi\test\pizza'.
There are 0 directories and 19 images in 'data\pizza_steak_sushi\test\steak'.
There are 0 directories and 31 images in 'data\pizza_steak_sushi\test\sushi'.
There are 3 directories and 0 images in 'data\pizza_steak_sushi\train'.
There are 0 directories and 78 images in 'data\pizza_steak_sushi\train\pizza'.
There are 0 directories and 75 images in 'data\pizza_steak_sushi\train\steak'.
There are 0 directories and 72 images in 'data\pizza_steak_sushi\train\sushi'.
```

Excellent!

It looks like we've got about 75 images per training class and 25 images per testing class.

That should be enough to get started.

Remember, these images are subsets of the original Food101 dataset.

You can see how they were created in the [data creation notebook](#).

While we're at it, let's setup our training and testing paths.

In [7]:

Out[7]:

```
(WindowsPath('data/pizza_steak_sushi/train'),  
 WindowsPath('data/pizza_steak_sushi/test'))
```

## 2.1 Visualize an image

Okay, we've seen how our directory structure is formatted.

Now in the spirit of the data explorer, it's time to *visualize, visualize, visualize!*

Let's write some code to:

1. Get all of the image paths using `pathlib.Path.glob()` to find all of the files ending in `.jpg`.
2. Pick a random image path using Python's `random.choice()`.
3. Get the image class name using `pathlib.Path.parent.stem`.
4. And since we're working with images, we'll open the random image path using `PIL.Image.open()` (PIL stands for Python Image Library).
5. We'll then show the image and print some metadata.

In [8]:







```
Random image path: data\pizza_steak_sushi\test\sushi\2394442.jpg
Image class: sushi
Image height: 408
Image width: 512
```

Out[8]:



We can do the same with `matplotlib.pyplot.imshow()`, except we have to convert the image to a NumPy array first.

In [9]:



Image class: sushi | Image shape: (408, 512, 3) -> [height, width, color\_channels]



### 3. Transforming data

Now what if we wanted to load our image data into PyTorch?

Before we can use our image data with PyTorch we need to:

1. Turn it into tensors (numerical representations of our images).
2. Turn it into a `torch.utils.data.Dataset` and subsequently a `torch.utils.data.DataLoader`, we'll call these `Dataset` and `DataLoader` for short.

There are several different kinds of pre-built datasets and dataset loaders for PyTorch, depending on the problem you're working on.

**Problem space**

**Pre-built Datasets and Functions**

<b>Vision</b>	<code>torchvision.datasets</code>
<b>Audio</b>	<code>torchaudio.datasets</code>
<b>Text</b>	<code>torchtext.datasets</code>
<b>Recommendation system</b>	<code>torchrec.datasets</code>

Since we're working with a vision problem, we'll be looking at `torchvision.datasets` for our data loading functions as well as `torchvision.transforms` for preparing our data.

Let's import some base libraries.

In [10]:

### 3.1 Transforming data with `torchvision.transforms`

We've got folders of images but before we can use them with PyTorch, we need to convert them into tensors.

One of the ways we can do this is by using the `torchvision.transforms` module.

`torchvision.transforms` contains many pre-built methods for formatting images, turning them into tensors and even manipulating them for **data augmentation** (the practice of altering data to make it harder for a model to learn, we'll see this later on) purposes .

To get experience with `torchvision.transforms`, let's write a series of transform steps that:

1. Resize the images using `transforms.Resize()` (from about 512x512 to 64x64, the same shape as the images on the [CNN Explainer website](#)).
2. Flip our images randomly on the horizontal using `transforms.RandomHorizontalFlip()` (this could be considered a form of data augmentation because it will artificially change our image data).
3. Turn our images from a PIL image to a PyTorch tensor using `transforms.ToTensor()`.

We can compile all of these steps using `torchvision.transforms.Compose()`.

In [11]:





Now we've got a composition of transforms, let's write a function to try them out on various images.

In [12]:













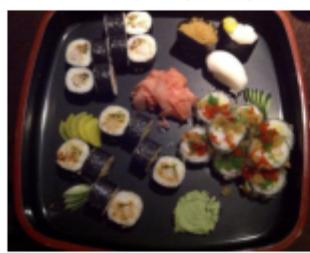




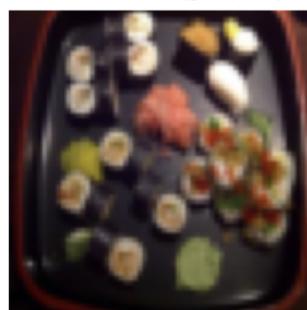


Class: sushi

Original  
Size: (512, 408)



Transformed  
Size: torch.Size([64, 64, 3])

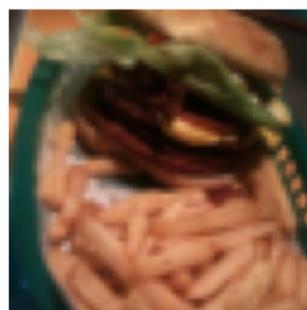


Class: pizza

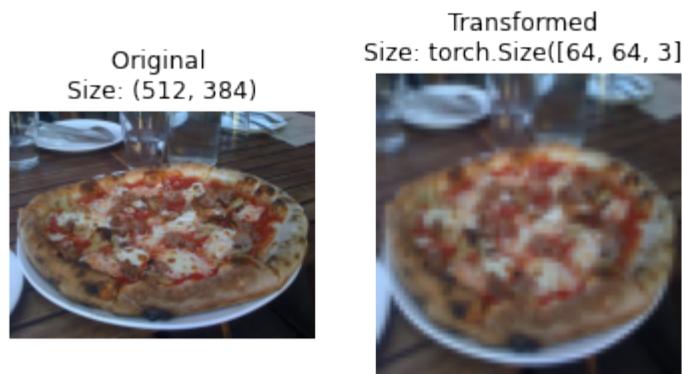
Original  
Size: (512, 512)



Transformed  
Size: torch.Size([64, 64, 3])



Class: pizza



Nice!

We've now got a way to convert our images to tensors using `torchvision.transforms`.

We also manipulate their size and orientation if needed (some models prefer images of different sizes and shapes).

Generally, the larger the shape of the image, the more information a model can recover.

For example, an image of size `[256, 256, 3]` will have 16x more pixels than an image of size `[64, 64, 3]`  
 $(256 \times 256 \times 3) / (64 \times 64 \times 3) = 16$ .

However, the tradeoff is that more pixels requires more computations.

**Exercise:** Try commenting out one of the transforms in `data_transform` and running the plotting function `plot_transformed_images()` again, what happens?

## 4. Option 1: Loading Image Data Using `ImageFolder`

Alright, time to turn our image data into a `Dataset` capable of being used with PyTorch.

Since our data is in standard image classification format, we can use the class `torchvision.datasets.ImageFolder`.

Where we can pass it the file path of a target image directory as well as a series of transforms we'd like to perform on our images.

Let's test it out on our data folders `train_dir` and `test_dir` passing in `transform=data_transform` to turn our images into tensors.

In [13]:







```
Train data:  
Dataset ImageFolder  
    Number of datapoints: 225  
    Root location: data\pizza_steak_sushi\train  
    StandardTransform  
Transform: Compose(  
    Resize(size=(64, 64), interpolation=bilinear, max_size=None, antialias=None)  
    RandomHorizontalFlip(p=0.5)  
    ToTensor()  
)  
Test data:  
Dataset ImageFolder  
    Number of datapoints: 75  
    Root location: data\pizza_steak_sushi\test  
    StandardTransform  
Transform: Compose(  
    Resize(size=(64, 64), interpolation=bilinear, max_size=None, antialias=None)  
    RandomHorizontalFlip(p=0.5)  
    ToTensor()  
)
```

Beautiful!

It looks like PyTorch has registered our `Dataset`'s.

Let's inspect them by checking out the `classes` and `class_to_idx` attributes as well as the lengths of our training and test sets.

In [14]:

```
['pizza', 'steak', 'sushi']
```

Out[14]:

In [15]:

Out[15]:

{'pizza': 0, 'steak': 1, 'sushi': 2}

In [16]:

(225, 75)

Out[16]:

Nice! Looks like we'll be able to use these to reference for later.

How about our images and labels?

How do they look?

We can index on our `train_data` and `test_data` `Dataset`'s to find samples and their target labels.

In [17]:

```

Image tensor:
tensor([[[0.1137, 0.1020, 0.0980, ..., 0.1255, 0.1216, 0.1176],
       [0.1059, 0.0980, 0.0980, ..., 0.1294, 0.1294, 0.1294],
       [0.1020, 0.0980, 0.0941, ..., 0.1333, 0.1333, 0.1333],
       ...,
       [0.1098, 0.1098, 0.1255, ..., 0.1686, 0.1647, 0.1686],
       [0.0863, 0.0941, 0.1098, ..., 0.1686, 0.1647, 0.1686],
       [0.0863, 0.0863, 0.0980, ..., 0.1686, 0.1647, 0.1647]],

      [[0.0745, 0.0706, 0.0745, ..., 0.0588, 0.0588, 0.0588],
       [0.0706, 0.0706, 0.0745, ..., 0.0627, 0.0627, 0.0627],
       [0.0706, 0.0745, 0.0745, ..., 0.0706, 0.0706, 0.0706],
       ...,
       [0.1255, 0.1333, 0.1373, ..., 0.2510, 0.2392, 0.2392],
       [0.1098, 0.1176, 0.1255, ..., 0.2510, 0.2392, 0.2314],
       [0.1020, 0.1059, 0.1137, ..., 0.2431, 0.2353, 0.2275]],

      [[0.0941, 0.0902, 0.0902, ..., 0.0196, 0.0196, 0.0196],
       [0.0902, 0.0863, 0.0902, ..., 0.0196, 0.0157, 0.0196],
       [0.0902, 0.0902, 0.0902, ..., 0.0157, 0.0157, 0.0196],
       ...,
       [0.1294, 0.1333, 0.1490, ..., 0.1961, 0.1882, 0.1804],
       [0.1098, 0.1137, 0.1255, ..., 0.1922, 0.1843, 0.1804],
       [0.1059, 0.1020, 0.1059, ..., 0.1843, 0.1804, 0.1765]]])

Image shape: torch.Size([3, 64, 64])
Image datatype: torch.float32
Image label: 0
Label datatype: <class 'int'>

```

Our images are now in the form of a tensor (with shape `[3, 64, 64]`) and the labels are in the form of an integer relating to a specific class (as referenced by the `class_to_idx` attribute).

How about we plot a single image tensor using `matplotlib`?

We'll first have to permute (rearrange the order of its dimensions) so it's compatible.

Right now our image dimensions are in the format `CHW` (color channels, height, width) but `matplotlib` prefers `HWC` (height, width, color channels).

In [18]:

```
Original shape: torch.Size([3, 64, 64]) -> [color_channels, height, width]
```

```
Image permute shape: torch.Size([64, 64, 3]) -> [height, width, color_channels]  
pizza
```



Notice the image is now more pixelated (less quality).

This is due to it being resized from `512x512` to `64x64` pixels.

The intuition here is that if you think the image is harder to recognize what's going on, chances are a model will find it harder to understand too.

## 4.1 Turn loaded images into `DataLoader`'s

We've got our images as PyTorch `Dataset`'s but now let's turn them into `DataLoader`'s.

We'll do so using `torch.utils.data.DataLoader`.

Turning our `Dataset`'s into `DataLoader`'s makes them iterable so a model can go through learn the relationships between samples and targets (features and labels).

To keep things simple, we'll use a `batch_size=1` and `num_workers=1`.

What's `num_workers`?

Good question.

It defines how many subprocesses will be created to load your data.

Think of it like this, the higher value `num_workers` is set to, the more compute power PyTorch will use to load your data.

Personally, I usually set it to the total number of CPUs on my machine via Python's `os.cpu_count()`.

This ensures the `DataLoader` recruits as many cores as possible to load data.

**Note:** There are more parameters you can get familiar with using `torch.utils.data.DataLoader` in the [PyTorch documentation](#).

In [19]:









Out[19]:

```
(<torch.utils.data.dataloader.DataLoader at 0x1fd2cf94fa0>,
 <torch.utils.data.dataloader.DataLoader at 0x1fd2cf94dc0>)
```

Wonderful!

Now our data is iterable.

Let's try it out and check the shapes.

In [20]:



```
Image shape: torch.Size([1, 3, 64, 64]) -> [batch_size, color_channels, height, width]
Label shape: torch.Size([1])
```

We could now use these `DataLoader`'s with a training and testing loop to train a model.

But before we do, let's look at another option to load images (or almost any other kind of data).

## 5. Option 2: Loading Image Data with a Custom `Dataset`

What if a pre-built `Dataset` creator like `torchvision.datasets.ImageFolder()` didn't exist?

Or one for your specific problem didn't exist?

Well, you could build your own.

But wait, what are the pros and cons of creating your own custom way to load `Dataset`'s?

### Pros of creating a custom `Dataset`

Can create a `Dataset` out of almost anything.

Not limited to PyTorch pre-built `Dataset` functions.

### Cons of creating a custom `Dataset`

Even though you *could* create a `Dataset` out of almost anything, it doesn't mean it will work.

Using a custom `Dataset` often results in writing more code, which could be prone to errors or performance issues.

To see this in action, let's work towards replicating `torchvision.datasets.ImageFolder()` by subclassing `torch.utils.data.Dataset` (the base class for all `Dataset`'s in PyTorch).

We'll start by importing the modules we need:

- Python's `os` for dealing with directories (our data is stored in directories).

Python's `pathlib` for dealing with filepaths (each of our images has a unique filepath).

- `torch` for all things PyTorch.
- PIL's `Image` class for loading images.
- `torch.utils.data.Dataset` to subclass and create our own custom `Dataset`.
- `torchvision.transforms` to turn our images into tensors.
- Various types from Python's `typing` module to add type hints to our code.

**Note:** You can customize the following steps for your own dataset. The premise remains: write code to load your data in the format you'd like it.

In [21]:

Remember how our instances of `torchvision.datasets.ImageFolder()` allowed us to use the `classes` and `class_to_idx` attributes?

In [22]:

Out[22]:

```
(['pizza', 'steak', 'sushi'], {'pizza': 0, 'steak': 1, 'sushi': 2})
```

## 5.1 Creating a helper function to get class names

Let's write a helper function capable of creating a list of class names and a dictionary of class names and their indexes given a directory path.

To do so, we'll:

1. Get the class names using `os.scandir()` to traverse a target directory (ideally the directory is in standard image classification format).
2. Raise an error if the class names aren't found (if this happens, there might be something wrong with the directory structure).
3. Turn the class names into a dictionary of numerical labels, one for each class.

Let's see a small example of step 1 before we write the full function.

In [23]:

```
Target directory: data\\pizza_steak_sushi\\train  
Class names found: ['pizza', 'steak', 'sushi']
```

Excellent!

How about we turn it into a full function?

In [24]:













Looking good!

Now let's test out our `find_classes()` function.

In [25]:

Out[25]:

```
(['pizza', 'steak', 'sushi'], {'pizza': 0, 'steak': 1, 'sushi': 2})
```

Woohoo! Looking good!

## 5.2 Create a custom `Dataset` to replicate `ImageFolder`

Now we're ready to build our own custom `Dataset`.

We'll build one to replicate the functionality of `torchvision.datasets.ImageFolder()`.

This will be good practice, plus, it'll reveal a few of the required steps to make your own custom `Dataset`.

It'll be a fair bit of a code... but nothing we can't handle!

Let's break it down:

1. Subclass `torch.utils.data.Dataset`.

2. Initialize our subclass with a `targ_dir` parameter (the target data directory) and `transform` parameter (so we have the option to transform our data if needed).
3. Create several attributes for `paths` (the paths of our target images), `transform` (the transforms we might like to use, this can be `None`), `classes` and `class_to_idx` (from our `find_classes()` function).
4. Create a function to load images from file and return them, this could be using `PIL` or `torchvision.io` (for input/output of vision data).
5. Overwrite the `__len__` method of `torch.utils.data.Dataset` to return the number of samples in the `Dataset`, this is recommended but not required. This is so you can call `len(Dataset)`.
6. Overwrite the `__getitem__` method of `torch.utils.data.Dataset` to return a single sample from the `Dataset`, this is required.

Let's do it!

In [26]:























Woah! A whole bunch of code to load in our images.

This is one of the downsides of creating your own custom `Dataset`'s.

However, now we've written it once, we could move it into a `.py` file such as `data_loader.py` along with some other helpful data functions and reuse it later on.

Before we test out our new `ImageFolderCustom` class, let's create some transforms to prepare our images.

In [27]:

Now comes the moment of truth!

Let's turn our training images (contained in `train_dir`) and our testing images (contained in `test_dir`) into `Dataset`'s using our own `ImageFolderCustom` class.

In [28]:



Out[28]:

```
(<__main__.ImageFolderCustom at 0x1fd2cf886d0>,
 <__main__.ImageFolderCustom at 0x1fd2cf5a0d0>)
```

Hmm... no errors, did it work?

Let's try calling `len()` on our new `Dataset`'s and find the `classes` and `class_to_idx` attributes.

In [29]:

Out[29]:

(225, 75)

In [30]:

['pizza', 'steak', 'sushi']

Out[30]:

In [31]:

{'pizza': 0, 'steak': 1, 'sushi': 2}

Out[31]:

len(test\_data\_custom) == len(test\_data) and len(test\_data\_custom) == len(test\_data) Yes!!!

It looks like it worked.

We could check for equality with the `Dataset`'s made by the `torchvision.datasets.ImageFolder()` class too.

In [32]:

True  
True  
True

Ho ho!

Look at us go!

Three `True`'s!

You can't get much better than that.

How about we take it up a notch and plot some random images to test our `__getitem__` override?

## 5.3 Create a function to display random images

You know what time it is!

Time to put on our data explorer's hat and *visualize, visualize, visualize!*

Let's create a helper function called `display_random_images()` that helps us visualize images in our `Dataset`'s.

Specifically, it'll:

1. Take in a `Dataset` and a number of other parameters such as `classes` (the names of our target classes), the number of images to display (`n`) and a random seed.

2. To prevent the display getting out of hand, we'll cap `n` at 10 images.
3. Set the random seed for reproducible plots (if `seed` is set).
4. Get a list of random sample indexes (we can use Python's `random.sample()` for this) to plot.
5. Setup a `matplotlib` plot.
6. Loop through the random sample indexes found in step 4 and plot them with `matplotlib`.
7. Make sure the sample images are of shape `HWC` (height, width, color channels) so we can plot them.

In [33]:





















What a good looking function!

Let's test it out first with the `Dataset` we created with `torchvision.datasets.ImageFolder()`.

In [34]:





And now with the `Dataset` we created with our own `ImageFolderCustom`.

In [35]:





Nice!!!

Looks like our `ImageFolderCustom` is working just as we'd like it to.

## 5.4 Turn custom loaded images into `DataLoader`'s

We've got a way to turn our raw images into `Dataset`'s (features mapped to labels or `x`'s mapped to `y`'s) through our `ImageFolderCustom` class.

Now how could we turn our custom `Dataset`'s into `DataLoader`'s?

If you guessed by using `torch.utils.data.DataLoader()`, you'd be right!

Because our custom `Dataset`'s subclass `torch.utils.data.Dataset`, we can use them directly with `torch.utils.data.DataLoader()`.

And we can do using very similar steps to before except this time we'll be using our custom created `Dataset`'s.

In [39]:













Out[39]:

```
(<torch.utils.data.dataloader.DataLoader at 0x1fd2cfabf10>,
 <torch.utils.data.dataloader.DataLoader at 0x1fd2cfabb80>)
```

Do the shapes of the samples look the same?

In [40]:

```
Image shape: torch.Size([1, 3, 64, 64]) -> [batch_size, color_channels, height, width]
Label shape: torch.Size([1])
```

They sure do!

Let's now take a look at some other forms of data transforms.

## 6. Other forms of transforms (data augmentation)

We've seen a couple of transforms on our data already but there's plenty more.

You can see them all in the [torchvision.transforms documentation](#).

The purpose of transforms is to alter your images in some way.

That may be turning your images into a tensor (as we've seen before).

Or cropping it or randomly erasing a portion or randomly rotating them.

Doing this kinds of transforms is often referred to as **data augmentation**.

**Data augmentation** is the process of altering your data in such a way that you *artificially* increase the diversity of your training set.

Training a model on this *artificially* altered dataset hopefully results in a model that is capable of better *generalization* (the patterns it learns are more robust to future unseen examples).

You can see many different examples of data augmentation performed on images using `torchvision.transforms` in PyTorch's [Illustration of Transforms example](#).

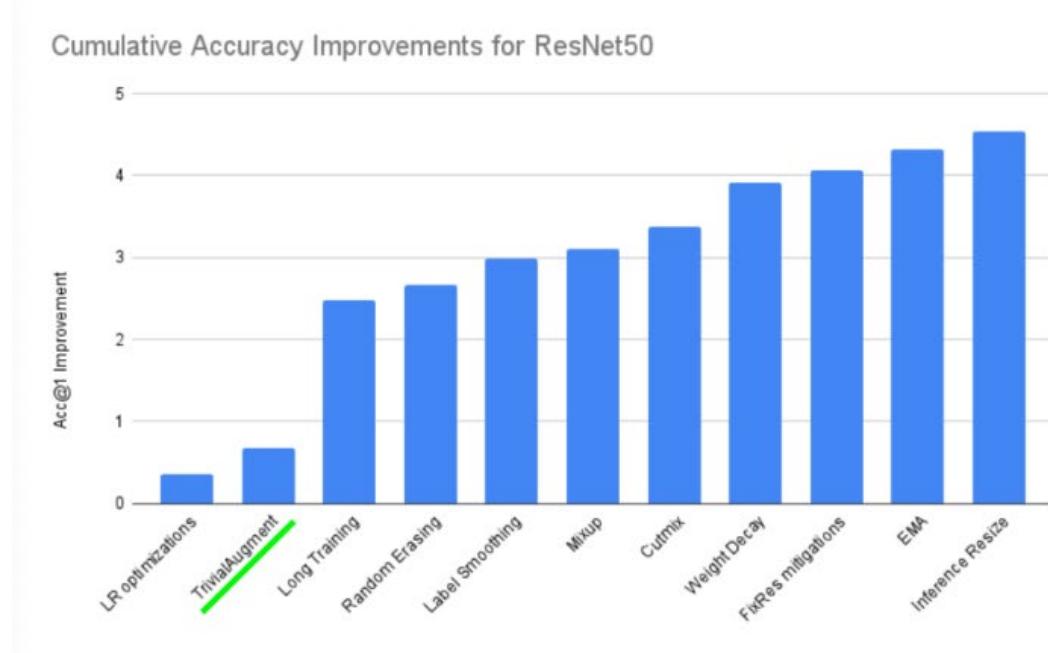
But let's try one out ourselves.

Machine learning is all about harnessing the power of randomness and research shows that random transforms (like `transforms.RandAugment()` and `transforms.TrivialAugmentWide()`) generally perform better than hand-picked transforms.

The idea behind `TrivialAugment` is... well, trivial.

You have a set of transforms and you randomly pick a number of them to perform on an image and at a random magnitude between a given range (a higher magnitude means more intense).

The PyTorch team even [used TrivialAugment](#) it to train their latest state-of-the-art vision models.



*TrivialAugment was one of the ingredients used in a recent state of the art training upgrade to various PyTorch vision models.*

How about we test it out on some of our own images?

The main parameter to pay attention to in `transforms.TrivialAugmentWide()` is `num_magnitude_bins=31`.

It defines how much of a range an intensity value will be picked to apply a certain transform, `0` being no range and `31` being maximum range (highest chance for highest intensity).

We can incorporate `transforms.TrivialAugmentWide()` into `transforms.Compose()`.

In [41]:



**Note:** You usually don't perform data augmentation on the test set. The idea of data augmentation is to to *artificially* increase the diversity of the training set to better predict on the testing set.

However, you do need to make sure your test set images are transformed to tensors. We size the test images to the same size as our training images too, however, inference can be done on different size images if necessary (though this may alter performance).

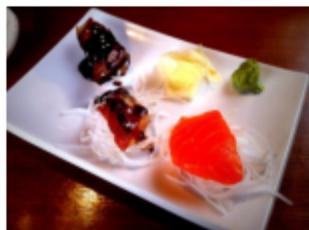
Beautiful, now we've got a training transform (with data augmentation) and test transform (without data augmentation).

Let's test our data augmentation out!

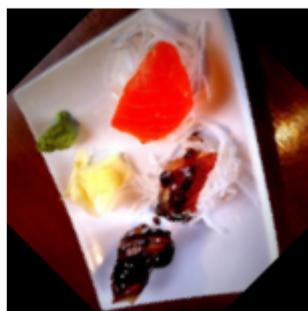
In [42]:

Class: sushi

Original  
Size: (512, 382)



Transformed  
Size: torch.Size([224, 224, 3])



Class: sushi

Original  
Size: (511, 512)



Transformed  
Size: torch.Size([224, 224, 3])



Class: pizza

Original  
Size: (512, 512)



Transformed  
Size: torch.Size([224, 224, 3])



Try running the cell above a few times and seeing how the original image changes as it goes through the transform.

## 7. Model 0: TinyVGG without data augmentation

Alright, we've seen how to turn our data from images in folders to transformed tensors.

Now let's construct a computer vision model to see if we can classify if an image is of pizza, steak or sushi.

To begin, we'll start with a simple transform, only resizing the images to  $(64, 64)$  and turning them into tensors.

## 7.1 Creating transforms and loading data for Model 0

In [43]:

Excellent, now we've got a simple transform, let's:

1. Load the data, turning each of our training and test folders first into a `Dataset` with  
`torchvision.datasets.ImageFolder()`
2. Then into a `DataLoader` using `torch.utils.data.DataLoader()`.

We'll set the `batch_size=32` and `num_workers` to as many CPUs on our machine (this will depend on what machine you're using).

In [44]:













Creating DataLoader's with batch size 32 and 16 workers.

Out[44]:

```
(<torch.utils.data.dataloader.DataLoader at 0x1fd2ce5c4f0>,
 <torch.utils.data.dataloader.DataLoader at 0x1fd1d0e10d0>)
```

DataLoader's created!

Let's build a model.

## 7.2 Create TinyVGG model class

In [notebook 03](#), we used the TinyVGG model from the [CNN Explainer website](#).

Let's recreate the same model, except this time we'll be using color images instead of grayscale (`in_channels=3` instead of `in_channels=1` for RGB pixels).

In [45]:

































Out[45]:

```
TinyVGG(  
    (conv_block_1): Sequential(  
        (0): Conv2d(3, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (1): ReLU()  
        (2): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (3): ReLU()  
        (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
    )  
    (conv_block_2): Sequential(  
        (0): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (1): ReLU()  
        (2): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (3): ReLU()  
)
```

```
(4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
(classifier): Sequential(
(0): Flatten(start_dim=1, end_dim=-1)
(1): Linear(in_features=2560, out_features=3, bias=True)
)
)
```

**Note:** One of the ways to speed up deep learning models computing on a GPU is to leverage **operator fusion**.

This means in the `forward()` method in our model above, instead of calling a layer block and reassigning `x` every time, we call each block in succession (see the final line of the `forward()` method in the model above for an example).

This saves the time spent reassigning `x` (memory heavy) and focuses on only computing on `x`.

See [\*Making Deep Learning Go Brrrr From First Principles\*](#) by Horace He for more ways on how to speed up machine learning models.

Now that's a nice looking model!

How about we test it out with a forward pass on a single image?

## 7.3 Try a forward pass on a single image (to test the model)

A good way to test a model is to do a forward pass on a single piece of data.

It's also handy way to test the input and output shapes of our different layers.

To do a forward pass on a single image, let's:

1. Get a batch of images and labels from the `DataLoader`.
2. Get a single image from the batch and `unsqueeze()` the image so it has a batch size of `1` (so its shape fits the model).
3. Perform inference on a single image (making sure to send the image to the target `device`).
4. Print out what's happening and convert the model's raw output logits to prediction probabilities with `torch.softmax()` (since we're working with multi-class data) and convert the prediction probabilities to prediction labels with `torch.argmax()`.

In [46]:







```
Single image shape: torch.Size([1, 3, 64, 64])

Output logits:
tensor([[0.0578, 0.0635, 0.0352]], device='cuda:0')

Output prediction probabilities:
tensor([[0.3352, 0.3371, 0.3277]], device='cuda:0')

Output prediction label:
tensor([1], device='cuda:0')

Actual label:
2
```

Wonderful, it looks like our model is outputting what we'd expect it to output.

You can run the cell above a few times and each time have a different image be predicted on.

And you'll probably notice the predictions are often wrong.

This is to be expected because the model hasn't been trained yet and it's essentially guessing using random weights.

## 7.4 Use `torchinfo` to get an idea of the shapes going through our model

Printing out our model with `print(model)` gives us an idea of what's going on with our model.

And we can print out the shapes of our data throughout the `forward()` method.

However, a helpful way to get information from our model is to use `torchinfo`.

`torchinfo` comes with a `summary()` method that takes a PyTorch model as well as an `input_shape` and returns what happens as a tensor moves through your model.

**Note:** If you're using Google Colab, you'll need to install `torchinfo`.

In [47]:



```
Collecting torchinfo
  Downloading torchinfo-1.6.5-py3-none-any.whl (21 kB)
Installing collected packages: torchinfo
Successfully installed torchinfo-1.6.5
```

Out[47]:

Layer (type:depth-idx)	Output Shape	Param #
TinyVGG	--	--
└ Sequential: 1-1	[1, 10, 32, 32]	--
└ Conv2d: 2-1	[1, 10, 64, 64]	280
└ ReLU: 2-2	[1, 10, 64, 64]	--
└ Conv2d: 2-3	[1, 10, 64, 64]	910
└ ReLU: 2-4	[1, 10, 64, 64]	--
└ MaxPool2d: 2-5	[1, 10, 32, 32]	--
└ Sequential: 1-2	[1, 10, 16, 16]	--
└ Conv2d: 2-6	[1, 10, 32, 32]	910
└ ReLU: 2-7	[1, 10, 32, 32]	--
└ Conv2d: 2-8	[1, 10, 32, 32]	910
└ ReLU: 2-9	[1, 10, 32, 32]	--
└ MaxPool2d: 2-10	[1, 10, 16, 16]	--
└ Sequential: 1-3	[1, 3]	--
└ Flatten: 2-11	[1, 2560]	--
└ Linear: 2-12	[1, 3]	7,683

Total params: 10,693

Trainable params: 10,693

Non-trainable params: 0

Total mult-adds (M): 6.75

Input size (MB): 0.05

Forward/backward pass size (MB): 0.82

Params size (MB): 0.04

Estimated Total Size (MB): 0.91

Nice!

The output of `torchinfo.summary()` gives us a whole bunch of information about our model.

Such as `Total params`, the total number of parameters in our model, the `Estimated Total Size (MB)` which is the size of our model.

You can also see the change in input and output shapes as data of a certain `input_size` moves through our model.

Right now, our parameter numbers and total model size is low.

This because we're starting with a small model.

And if we need to increase its size later, we can.

## 7.5 Create train & test loop functions

We've got data and we've got a model.

Now let's make some training and test loop functions to train our model on the training data and evaluate our model on the testing data.

And to make sure we can use these the training and testing loops again, we'll functionize them.

Specifically, we're going to make three functions:

1. `train_step()` - takes in a model, a `DataLoader`, a loss function and an optimizer and trains the model on the `DataLoader`.
2. `test_step()` - takes in a model, a `DataLoader` and a loss function and evaluates the model on the `DataLoader`.
3. `train()` - performs 1. and 2. together for a given number of epochs and returns a results dictionary.

**Note:** We covered the steps in a PyTorch optimization loop in [notebook 01](#), as well as the [Unofficial PyTorch Optimization Loop Song](#) and we've built similar functions in [notebook 03](#).

Let's start by building `train_step()`.

Because we're dealing with batches in the `DataLoader`'s, we'll accumulate the model loss and accuracy values during training (by adding them up for each batch) and then adjust them at the end before we return them.

In [48]:



















Woohoo! `train_step()` function done.

Now let's do the same for the `test_step()` function.

The main difference here will be the `test_step()` won't take in an optimizer and therefore won't perform gradient descent.

But since we'll be doing inference, we'll make sure to turn on the `torch.inference_mode()` context manager for making predictions.

In [49]:

















Excellent!

## 7.6 Creating a `train()` function to combine `train_step()` and `test_step()`

Now we need a way to put our `train_step()` and `test_step()` functions together.

To do so, we'll package them up in a `train()` function.

This function will train the model as well as evaluate it.

Specifically, it'll:

1. Take in a model, a `DataLoader` for training and test sets, an optimizer, a loss function and how many epochs to perform each train and test step for.
2. Create an empty results dictionary for `train_loss`, `train_acc`, `test_loss` and `test_acc` values (we can fill this up as training goes on).
3. Loop through the training and test step functions for a number of epochs.
4. Print out what's happening at the end of each epoch.
5. Update the empty results dictionary with the updated metrics each epoch.
6. Return the filled

To keep track of the number of epochs we've been through, let's import `tqdm` from `tqdm.auto` (`tqdm` is one of the most popular progress bar libraries for Python and `tqdm.auto` automatically decides what kind of progress bar is best for your computing environment, e.g. Jupyter Notebook vs. Python script).

In [50]:

























## 7.7 Train and Evaluate Model 0

Alright, alright, alright we've got all of the ingredients we need to train and evaluate our model.

Time to put our `TinyVGG` model, `DataLoader`'s and `train()` function together to see if we can build a model capable of discerning between pizza, steak and sushi!

Let's recreate `model_0` (we don't need to but we will for completeness) then call our `train()` function passing in the necessary parameters.

To keep our experiments quick, we'll train our model for **5 epochs** (though you could increase this if you want).

As for an **optimizer** and **loss function**, we'll use `torch.nn.CrossEntropyLoss()` (since we're working with multi-class classification data) and `torch.optim.Adam()` with a learning rate of `1e-3` respectively.

To see how long things take, we'll import Python's `timeit.default_timer()` method to calculate the training time.

In [51]:













```
0% | 0/5 [00:00<?, ?it/s]
Epoch: 1 | train_loss: 1.1078 | train_acc: 0.2578 | test_loss: 1.1360 | test_acc: 0.2604
Epoch: 2 | train_loss: 1.0847 | train_acc: 0.4258 | test_loss: 1.1620 | test_acc: 0.1979
Epoch: 3 | train_loss: 1.1157 | train_acc: 0.2930 | test_loss: 1.1695 | test_acc: 0.1979
Epoch: 4 | train_loss: 1.0955 | train_acc: 0.4141 | test_loss: 1.1380 | test_acc: 0.1979
Epoch: 5 | train_loss: 1.0985 | train_acc: 0.2930 | test_loss: 1.1420 | test_acc: 0.1979
Total training time: 65.122 seconds
```

Hmm...

It looks like our model performed pretty poorly.

But that's okay for now, we'll keep persevering.

What are some ways you could potentially improve it?

**Note:** Check out the *Improving a model (from a model perspective)* section in notebook 02 for ideas on improving our TinyVGG model.

## 7.8 Plot the loss curves of Model 0

From the print outs of our `model_0` training, it didn't look like it did too well.

But we can further evaluate it by plotting the model's **loss curves**.

**Loss curves** show the model's results over time.

And they're a great way to see how your model performs on different datasets (e.g. training and test).

Let's create a function to plot the values in our `model_0_results` dictionary.

In [52]:

```
dict_keys(['train_loss', 'train_acc', 'test_loss', 'test_acc'])
```

We'll need to extract each of these keys and turn them into a plot.

In [53]:







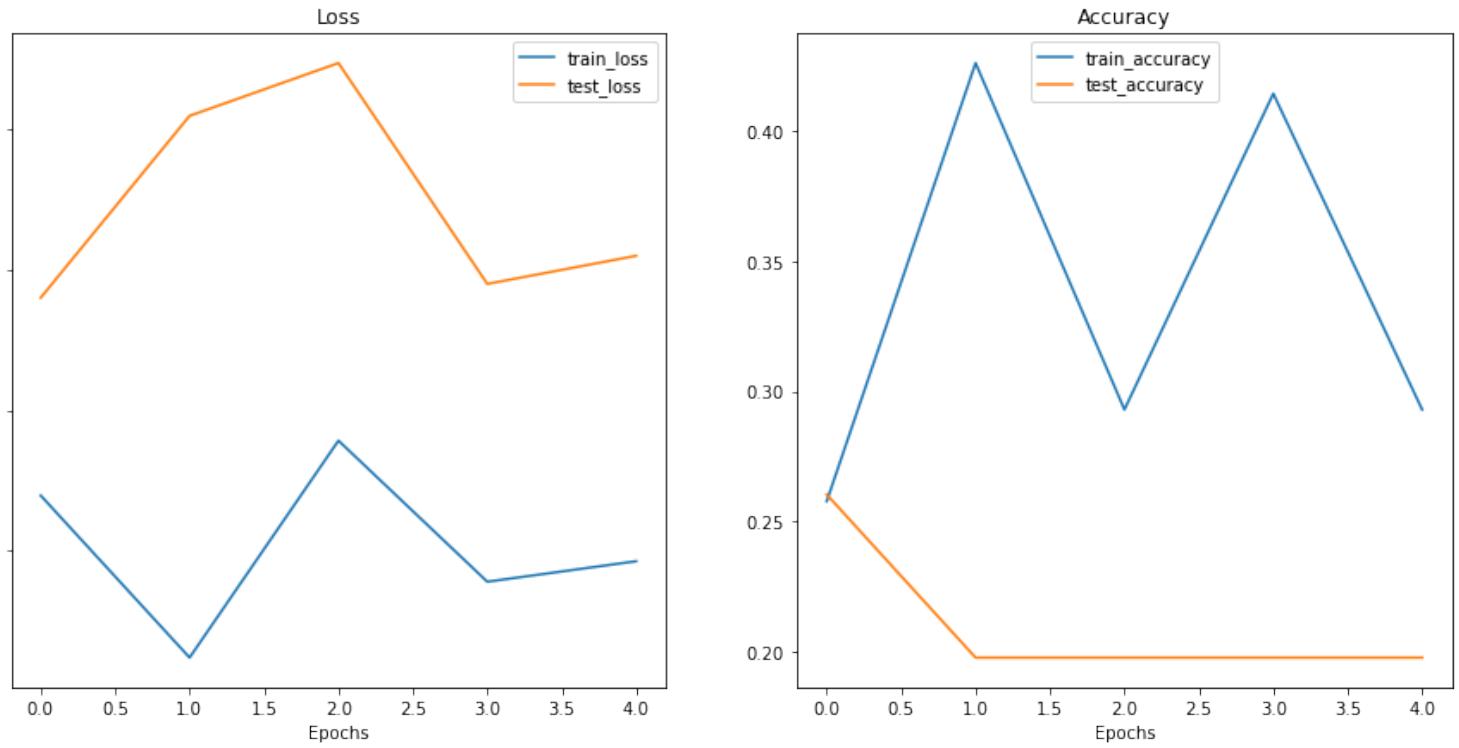






Okay, let's test our `plot_loss_curves()` function out.

In [54]:



Woah.

Looks like things are all over the place...

But we kind of knew that because our model's print out results during training didn't show much promise.

You could try training the model for longer and see what happens when you plot a loss curve over a longer time horizon.

## 8. What should an ideal loss curve look like?

Looking at training and test loss curves is a great way to see if your model is **overfitting**.

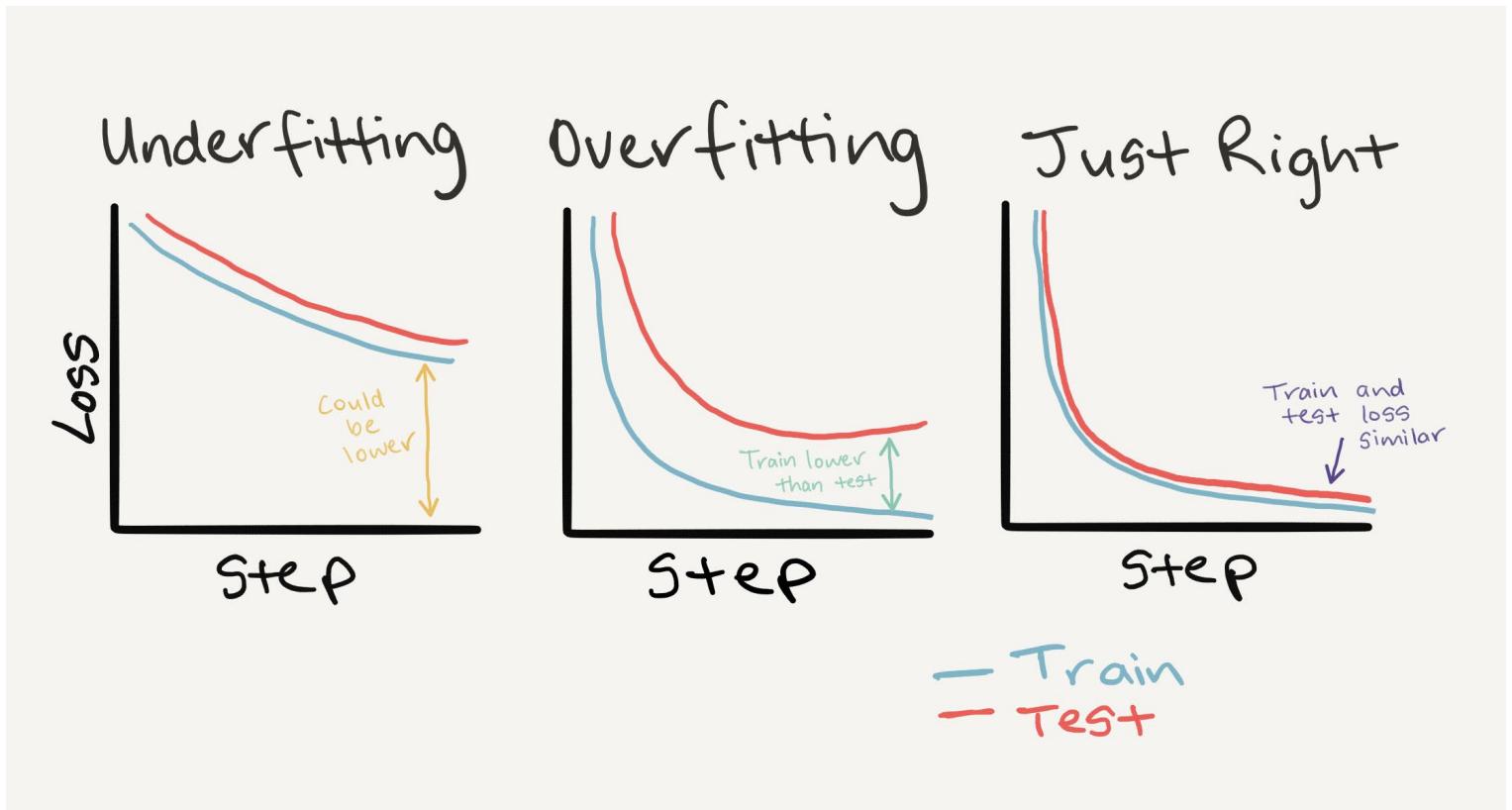
An overfitting model is one that performs better (often by a considerable margin) on the training set than the validation/test set.

If your training loss is far lower than your test loss, your model is **overfitting**.

As in, it's learning the patterns in the training too well and those patterns aren't generalizing to the test data.

The other side is when your training and test loss are not as low as you'd like, this is considered **underfitting**.

The ideal position for a training and test loss curve is for them to line up closely with each other.



Left: If your training and test loss curves aren't as low as you'd like, this is considered **underfitting**. Middle: When your test/validation loss is higher than your training loss this is considered **overfitting**. Right: The ideal scenario is when your training and test loss curves line up over time. This means your model is generalizing well. There are more combinations and different things loss curves can do, for more on these, see Google's [Interpreting Loss Curves guide](#).

## 8.1 How to deal with overfitting

Since the main problem with overfitting is that your model is fitting the training data *too well*, you'll want to use techniques to "reign it in".

A common technique of preventing overfitting is known as **regularization**.

I like to think of this as "making our models more regular", as in, capable of fitting *more* kinds of data.

Let's discuss a few methods to prevent overfitting.

<b>Method to prevent overfitting</b>	<b>What is it?</b>
--	--------------------

<b>Get more data</b>	Having more data gives the model more opportunities to learn patterns, patterns which may be more generalizable to new examples.
<b>Simplify your model</b>	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.
<b>Use data augmentation</b>	<b>Data augmentation</b> 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.
<b>Use transfer learning</b>	<b>Transfer learning</b> involves leveraging 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.
<b>Use dropout layers</b>	Dropout layers randomly remove connections between hidden layers in neural networks, effectively simplifying a model but also making the remaining connections better. See <code>torch.nn.Dropout()</code> for more.
<b>Use learning rate decay</b>	The idea here is to slowly decrease the learning rate as a model trains. This is akin to reaching for a coin at the back of a couch. The closer you get, the smaller your steps. The same with the learning rate, the closer you get to <b>convergence</b> , the smaller you'll want your weight updates to be.
<b>Use early stopping</b>	<b>Early stopping</b> stops model training <i>before</i> it begins to overfit. As in, say the model's loss has stopped decreasing for the past 10 epochs (this number is arbitrary), you may want to stop the model training here and go with the model weights that had the lowest loss (10 epochs prior).

There are more methods for dealing with overfitting but these are some of the main ones.

As you start to build more and more deep models, you'll find because deep learnings are *so good* at learning patterns in data, dealing with overfitting is one of the primary problems of deep learning.

## 8.2 How to deal with underfitting

When a model is **underfitting** it is considered to have poor predictive power on the training and test sets.

In essence, an underfitting model will fail to reduce the loss values to a desired level.

Right now, looking at our current loss curves, I'd considered our `TinyVGG` model, `model_0`, to be underfitting the data.

The main idea behind dealing with underfitting is to *increase* your model's predictive power.

There are several ways to do this.

<b>Method to prevent underfitting</b>	<b>What is it?</b>
<b>Add more layers/units to your model</b>	If your model is underfitting, it may not have enough capability to <i>learn</i> the required patterns/weights/representations of the data to be predictive. One way to add more predictive power to your model is to increase the number of hidden layers/units within those layers.
<b>Tweak the learning rate</b>	Perhaps your model's learning rate is too high to begin with. And it's trying to update its weights each epoch too much, in turn not learning anything. In this case, you might lower the learning rate and see what happens.
<b>Use transfer learning</b>	Transfer learning is capable of preventing overfitting and underfitting. It involves using the patterns from a previously working model and adjusting them to your own problem.
<b>Train for longer</b>	Sometimes a model just needs more time to learn representations of data. If you find in your smaller experiments your model isn't learning anything, perhaps leaving it train for a more epochs may result in better performance.
<b>Use less regularization</b>	Perhaps your model is underfitting because you're trying to prevent overfitting too much. Holding back on regularization techniques can help your model fit the data better.

## 8.3 The balance between overfitting and underfitting

None of the methods discussed above are silver bullets, meaning, they don't always work.

And preventing overfitting and underfitting is possibly the most active area of machine learning research.

Since everyone wants their models to fit better (less underfitting) but not so good they don't generalize well and perform in the real world (less overfitting).

There's a fine line between overfitting and underfitting.

Because too much of each can cause the other.

Transfer learning is perhaps one of the most powerful techniques when it comes to dealing with both overfitting and underfitting on your own problems.

Rather than handcraft different overfitting and underfitting techniques, transfer learning enables you to take an already working model in a similar problem space to yours (say one from [paperswithcode.com/sota](https://paperswithcode.com/sota) or [Hugging Face models](#)) and apply it to your own dataset.

We'll see the power of transfer learning in a later notebook.

## 9. Model 1: TinyVGG with Data Augmentation

Time to try out another model!

This time, let's load in the data and use **data augmentation** to see if it improves our results in anyway.

First, we'll compose a training transform to include `transforms.TrivialAugmentWide()` as well as resize and turn our images into tensors.

We'll do the same for a testing transform except without the data augmentation.

### 9.1 Create transform with data augmentation

In [55]:

Wonderful!

Now let's turn our images into `Dataset`'s using `torchvision.datasets.ImageFolder()` and then into `DataLoader`'s with `torch.utils.data.DataLoader()`.

## 9.2 Create train and test `Dataset`'s and `DataLoader`'s

We'll make sure the train Dataset uses the train\_transform\_trivial\_augment and the test Dataset uses the test\_transform.

In [56]:

Out[56]:

```
(Dataset ImageFolder
    Number of datapoints: 225
    Root location: data\pizza_steak_sushi\train
    StandardTransform
    Transform: Compose(
        Resize(size=(64, 64), interpolation=bilinear, max_size=None, antialias=None)
        TrivialAugmentWide(num_magnitude_bins=31, interpolation=InterpolationMode.NEAREST
, fill=None)
        ToTensor()
    ),
    Dataset ImageFolder
    Number of datapoints: 75
    Root location: data\pizza_steak_sushi\test
```

```
    StandardTransform  
    Transform: Compose(  
        Resize(size=(64, 64), interpolation=bilinear, max_size=None, antialias=None)  
        ToTensor()  
    )
```

And we'll make `DataLoader`'s with a `batch_size=32` and with `num_workers` set to the number of CPUs available on our machine (we can get this using Python's `os.cpu_count()`).

In [57]:









Out[57]:

```
(<torch.utils.data.DataLoader at 0x1fd2d0531c0>,
 <torch.utils.data.DataLoader at 0x1fd2cf94dc0>)
```

### 9.3 Construct and train Model 1

Data loaded!

Now to build our next model, `model_1`, we can reuse our `TinyVGG` class from before.

We'll make sure to send it to the target device.

In [58]:

Out[58]:

```
TinyVGG(  
    (conv_block_1): Sequential(  
        (0): Conv2d(3, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (1): ReLU()  
        (2): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (3): ReLU()  
        (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
```

```

)
(conv_block_2): Sequential(
(0): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(1): ReLU()
(2): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(3): ReLU()
(4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
(classifier): Sequential(
(0): Flatten(start_dim=1, end_dim=-1)
(1): Linear(in_features=2560, out_features=3, bias=True)
)
)

```

Model ready!

Time to train!

Since we've already got functions for the training loop (`train_step()`) and testing loop (`test_step()`) and a function to put them together in `train()`, let's reuse those.

We'll use the same setup as `model_0` with only the `train_dataloader` parameter varying:

- Train for 5 epochs.
- Use `train_dataloader=train_dataloader_augmented` as the training data in `train()`.
- Use `torch.nn.CrossEntropyLoss()` as the loss function (since we're working with multi-class classification).
- Use `torch.optim.Adam()` with `lr=0.001` as the learning rate as the optimizer.

In [59]:









```
0% | 0/5 [00:00<?, ?it/s]
Epoch: 1 | train_loss: 1.1074 | train_acc: 0.2461 | test_loss: 1.1058 | test_acc: 0.2604
Epoch: 2 | train_loss: 1.0791 | train_acc: 0.4258 | test_loss: 1.1382 | test_acc: 0.2604
Epoch: 3 | train_loss: 1.0804 | train_acc: 0.4258 | test_loss: 1.1683 | test_acc: 0.2604
Epoch: 4 | train_loss: 1.1287 | train_acc: 0.3047 | test_loss: 1.1626 | test_acc: 0.2604
Epoch: 5 | train_loss: 1.0884 | train_acc: 0.4258 | test_loss: 1.1481 | test_acc: 0.2604
Total training time: 67.543 seconds
```

Hmm...

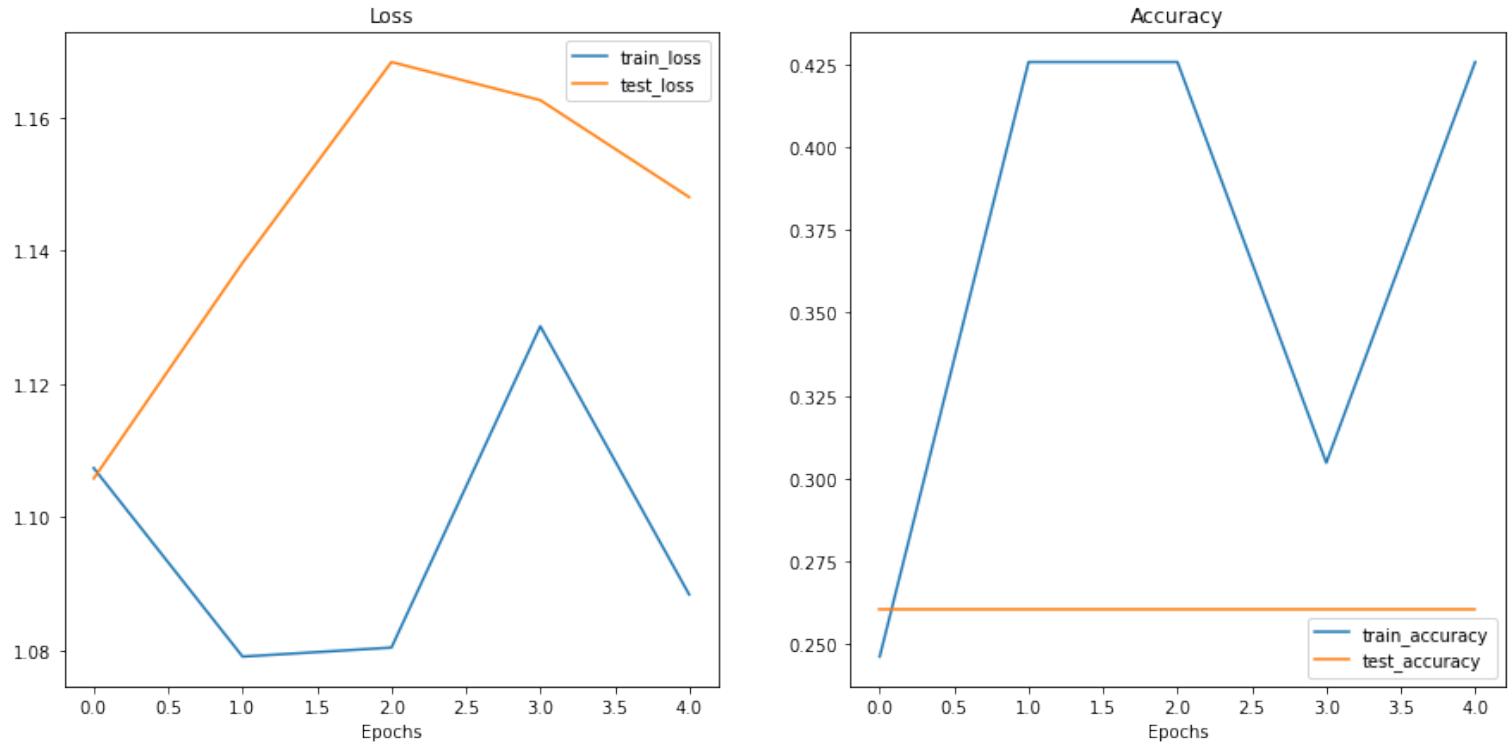
It doesn't look like our model performed very well again.

Let's check out its loss curves.

## 9.4 Plot the loss curves of Model 1

Since we've got the results of `model_1` saved in a results dictionary, `model_1_results`, we can plot them using `plot_loss_curves()`.

In [60]:



Wow...

These don't look very good either...

Is our model **underfitting** or **overfitting**?

Or both?

Ideally we'd like it have higher accuracy and lower loss right?

What are some methods you could try to use to achieve these?

## 10. Compare model results

Even though our models our performing quite poorly, we can still write code to compare them.

Let's first turn our model results in pandas DataFrames.

In [61]:

Out[61]:

	train_loss	train_acc	test_loss	test_acc
0	1.107832	0.257812	1.136025	0.260417
1	1.084726	0.425781	1.161953	0.197917
2	1.115656	0.292969	1.169479	0.197917
3	1.095543	0.414062	1.137993	0.197917
4	1.098464	0.292969	1.142002	0.197917

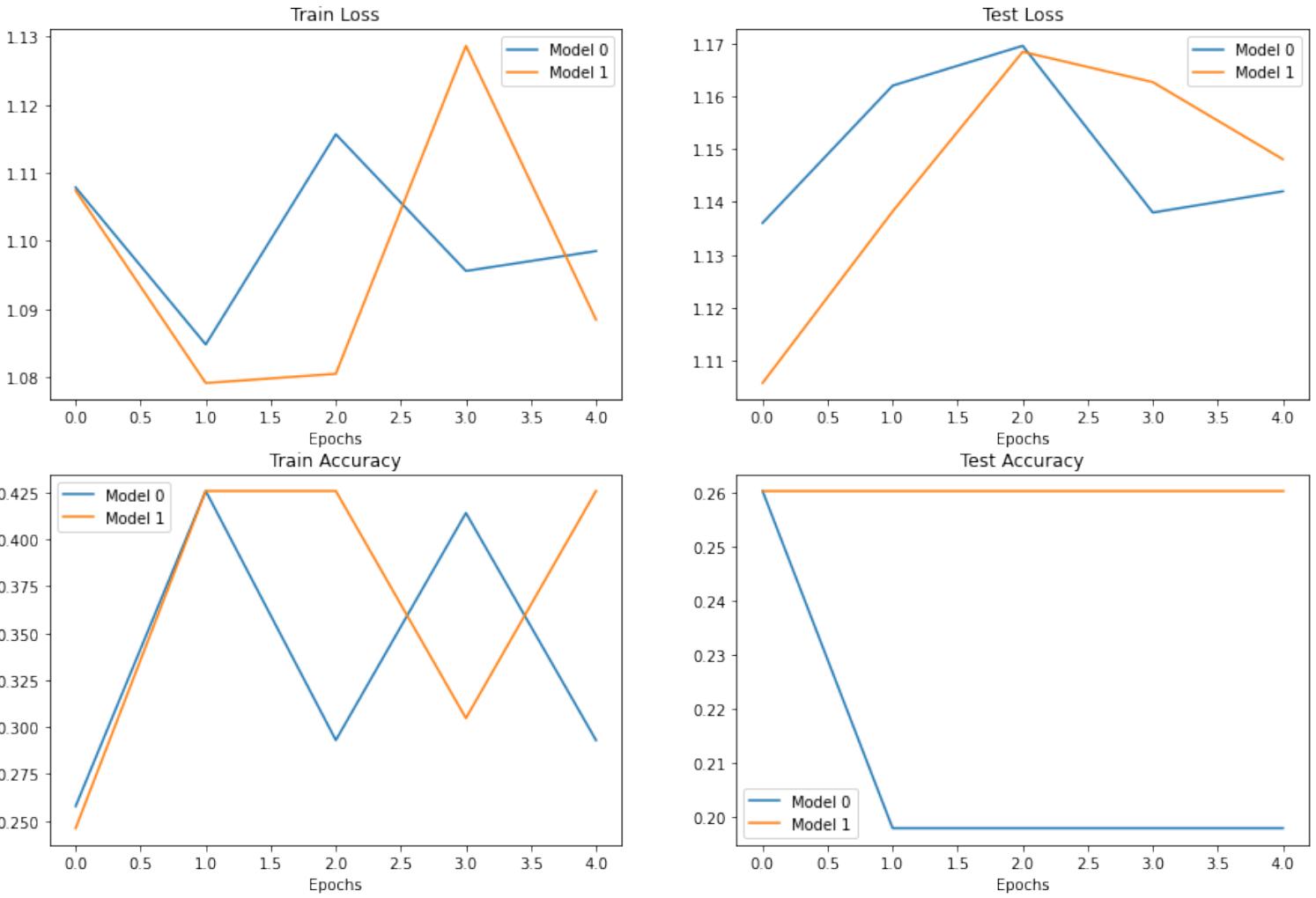
And now we can write some plotting code using `matplotlib` to visualize the results of `model_0` and `model_1` together.

In [62]:









It looks like our models both performed equally poorly and were kind of sporadic (the metrics go up and down sharply).

If you built `model_2`, what would you do differently to try and improve performance?

## 11. Make a prediction on a custom image

If you've trained a model on a certain dataset, chances are you'd like to make a prediction on your own custom data.

In our case, since we've trained a model on pizza, steak and sushi images, how could we use our model to make a prediction on one of our own images?

To do so, we can load an image and then **preprocess it in a way that matches the type of data our model was trained on.**

In other words, we'll have to convert our own custom image to a tensor and make sure it's in the right datatype before passing it to our model.

Let's start by downloading a custom image.

Since our model predicts whether an image contains pizza, steak or sushi, let's download a photo of [my Dad giving two thumbs up to a big pizza from the Learn PyTorch for Deep Learning GitHub](#).

We download the image using Python's `requests` module.

**Note:** If you're using Google Colab, you can also upload an image to the current session by going to the left hand side menu -> Files -> Upload to session storage. Beware though, this image will delete when your Google Colab session ends.

In [63]:





```
data\04-pizza-dad.jpeg already exists, skipping download.
```

## 11.1 Loading in a custom image with PyTorch

Excellent!

Looks like we've got a custom image downloaded and ready to go at `data/04-pizza-dad.jpeg`.

Time to load it in.

PyTorch's `torchvision` has several input and output ("IO" or "io" for short) methods for reading and writing images and video in `torchvision.io`.

Since we want to load in an image, we'll use `torchvision.io.read_image()`.

This method will read a JPEG or PNG image and turn it into a 3 dimensional RGB or grayscale `torch.Tensor` with values of datatype `uint8` in range `[0, 255]`.

Let's try it out.

In [64]:

```

Custom image tensor:
tensor([[[[154, 175, 181, ..., 21, 18, 14],
          [146, 167, 180, ..., 21, 18, 15],
          [124, 146, 171, ..., 18, 17, 15],
          ...,
          [ 72,  59,  45, ..., 152, 150, 148],
          [ 64,  55,  41, ..., 150, 147, 144],
          [ 64,  60,  46, ..., 149, 146, 143]],

         [[171, 189, 193, ..., 22, 19, 15],
          [163, 181, 194, ..., 22, 19, 16],
          [141, 163, 185, ..., 19, 18, 16],
          ...,
          [ 55,  42,  28, ..., 106, 104, 102],
          [ 47,  38,  24, ..., 108, 105, 102],
          [ 47,  43,  29, ..., 107, 104, 101]],

         [[117, 138, 145, ..., 17, 14, 10],
          [109, 130, 145, ..., 17, 14, 11],
          [ 87, 111, 136, ..., 14, 13, 11],
          ...,
          [ 35,  22,   8, ..., 54, 52, 50],
          [ 27,  18,   4, ..., 50, 47, 44],
          [ 27,  23,   9, ..., 49, 46, 43]]], dtype=torch.uint8)

```

Custom image shape: torch.Size([3, 4032, 3024])

Custom image dtype: torch.uint8

Nice! Looks like our image is in tensor format, however, is this image format compatible with our model?

Our `custom_image` tensor is of datatype `torch.uint8` and its values are between `[0, 255]`.

But our model takes image tensors of datatype `torch.float32` and with values between `[0, 1]`.

So before we use our custom image with our model, **we'll need to convert it to the same format as the data our model is trained on.**

If we don't do this, our model will error.

In [65]:

```
-----  
RuntimeError                                     Traceback (most recent call last)  
Input In [65], in <cell line: 3>()  
      2 model_1.eval()  
      3 with torch.inference_mode():  
-> 4     model_1(custom_image_uint8.to(device))  
  
File ~\mambaforge\envs\ds\lib\site-packages\torch\nn\modules\module.py:1110, in Module._call_impl  
(self, *input, **kwargs)  
    1106 # If we don't have any hooks, we want to skip the rest of the logic in  
    1107 # this function, and just call forward.  
    1108 if not (self._backward_hooks or self._forward_hooks or self._forward_pre_hooks or _global  
_backward_hooks  
    1109         or _global_forward_hooks or _global_forward_pre_hooks):  
-> 1110     return forward_call(*input, **kwargs)  
    1111 # Do not call functions when jit is used  
    1112 full_backward_hooks, non_full_backward_hooks = [], []  
  
Input In [45], in TinyVGG.forward(self, x)  
    39 def forward(self, x: torch.Tensor):  
-> 40     x = self.conv_block_1(x)
```

```

41      # print(x.shape)
42      x = self.conv_block_2(x)

File ~\mambaforge\envs\ds\lib\site-packages\torch\nn\modules\module.py:1110, in Module._call_impl
(self, *input, **kwargs)
    1106 # If we don't have any hooks, we want to skip the rest of the logic in
    1107 # this function, and just call forward.
    1108 if not (self._backward_hooks or self._forward_hooks or self._forward_pre_hooks or _global
_backward_hooks
    1109         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1110     return forward_call(*input, **kwargs)
    1111 # Do not call functions when jit is used
    1112 full_backward_hooks, non_full_backward_hooks = [], []

File ~\mambaforge\envs\ds\lib\site-packages\torch\nn\modules\container.py:141, in Sequential.forward(self, input)
    139 def forward(self, input):
    140     for module in self:
--> 141         input = module(input)
    142     return input

File ~\mambaforge\envs\ds\lib\site-packages\torch\nn\modules\module.py:1110, in Module._call_impl
(self, *input, **kwargs)
    1106 # If we don't have any hooks, we want to skip the rest of the logic in
    1107 # this function, and just call forward.
    1108 if not (self._backward_hooks or self._forward_hooks or self._forward_pre_hooks or _global
_backward_hooks
    1109         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1110     return forward_call(*input, **kwargs)
    1111 # Do not call functions when jit is used
    1112 full_backward_hooks, non_full_backward_hooks = [], []

File ~\mambaforge\envs\ds\lib\site-packages\torch\nn\modules\conv.py:447, in Conv2d.forward(self,
input)
    446 def forward(self, input: Tensor) -> Tensor:
--> 447     return self._conv_forward(input, self.weight, self.bias)

File ~\mambaforge\envs\ds\lib\site-packages\torch\nn\modules\conv.py:443, in Conv2d._conv_forward
(self, input, weight, bias)
    439 if self.padding_mode != 'zeros':
    440     return F.conv2d(F.pad(input, self._reversed_padding_repeated_twice, mode=self.padding
_mode),
    441                     weight, bias, self.stride,
    442                     _pair(0), self.dilation, self.groups)
--> 443 return F.conv2d(input, weight, bias, self.stride,
    444                     self.padding, self.dilation, self.groups)

```

RuntimeError: Input type (torch.cuda.ByteTensor) and weight type (torch.cuda.FloatTensor) should be the same

If we try to make a prediction on an image in a different datatype to what our model was trained on, we get an error like the following:

```
RuntimeError: Input type (torch.cuda.ByteTensor) and weight type (torch.cuda.FloatTensor) should be the same
```

Let's fix this by converting our custom image to the same datatype as what our model was trained on (`torch.float32`).

In [66]:



```

Custom image tensor:
tensor([[[0.6039, 0.6863, 0.7098, ..., 0.0824, 0.0706, 0.0549],
       [0.5725, 0.6549, 0.7059, ..., 0.0824, 0.0706, 0.0588],
       [0.4863, 0.5725, 0.6706, ..., 0.0706, 0.0667, 0.0588],
       ...,
       [0.2824, 0.2314, 0.1765, ..., 0.5961, 0.5882, 0.5804],
       [0.2510, 0.2157, 0.1608, ..., 0.5882, 0.5765, 0.5647],
       [0.2510, 0.2353, 0.1804, ..., 0.5843, 0.5725, 0.5608]],

      [[0.6706, 0.7412, 0.7569, ..., 0.0863, 0.0745, 0.0588],
       [0.6392, 0.7098, 0.7608, ..., 0.0863, 0.0745, 0.0627],
       [0.5529, 0.6392, 0.7255, ..., 0.0745, 0.0706, 0.0627],
       ...,
       [0.2157, 0.1647, 0.1098, ..., 0.4157, 0.4078, 0.4000],
       [0.1843, 0.1490, 0.0941, ..., 0.4235, 0.4118, 0.4000],
       [0.1843, 0.1686, 0.1137, ..., 0.4196, 0.4078, 0.3961]],

      [[0.4588, 0.5412, 0.5686, ..., 0.0667, 0.0549, 0.0392],
       [0.4275, 0.5098, 0.5686, ..., 0.0667, 0.0549, 0.0431],
       [0.3412, 0.4353, 0.5333, ..., 0.0549, 0.0510, 0.0431],
       ...,
       [0.1373, 0.0863, 0.0314, ..., 0.2118, 0.2039, 0.1961],
       [0.1059, 0.0706, 0.0157, ..., 0.1961, 0.1843, 0.1725],
       [0.1059, 0.0902, 0.0353, ..., 0.1922, 0.1804, 0.1686]]])

Custom image shape: torch.Size([3, 4032, 3024])

Custom image dtype: torch.float32

```

## 11.2 Predicting on custom images with a trained PyTorch model

Beautiful, it looks like our image data is now in the same format our model was trained on.

Except for one thing...

It's `shape`.

Our model was trained on images with shape `[3, 64, 64]`, whereas our custom image is currently `[3, 4032, 3024]`.

How could we make sure our custom image is the same shape as the images our model was trained on?

Are there any `torchvision.transforms` that could help?

Before we answer that question, let's plot the image with `matplotlib` to make sure it looks okay, remember we'll have to permute the dimensions from `CHW` to `HWC` to suit `matplotlib`'s requirements.

In [67]:



Image shape: `torch.Size([3, 4032, 3024])`



Two thumbs up!

Now how could we get our image to be the same size as the images our model was trained on?

One way to do so is with `torchvision.transforms.Resize()`.

Let's compose a transform pipeline to do so.

In [68]:

```
Original shape: torch.Size([3, 4032, 3024])
New shape: torch.Size([3, 64, 64])
```

Woohoo!

Let's finally make a prediction on our own custom image.

In [69]:

```
-----  
RuntimeError                                     Traceback (most recent call last)  
Input In [69], in <cell line: 2>()  
      1 model_1.eval()  
      2 with torch.inference_mode():  
----> 3     custom_image_pred = model_1(custom_image_transformed)  
  
File ~\mambaforge\envs\ds\lib\site-packages\torch\nn\modules\module.py:1110, in Module._call_impl(self, *input, **kwargs)  
    1106 # If we don't have any hooks, we want to skip the rest of the logic in  
    1107 # this function, and just call forward.  
    1108 if not (self._backward_hooks or self._forward_hooks or self._forward_pre_hooks or _global  
_backward_hooks  
        or _global_forward_hooks or _global_forward_pre_hooks):  
-> 1110     return forward_call(*input, **kwargs)  
    1111 # Do not call functions when jit is used  
    1112 full_backward_hooks, non_full_backward_hooks = [], []  
  
Input In [45], in TinyVGG.forward(self, x)  
    39 def forward(self, x: torch.Tensor):  
----> 40     x = self.conv_block_1(x)  
     41     # print(x.shape)  
     42     x = self.conv_block_2(x)  
  
File ~\mambaforge\envs\ds\lib\site-packages\torch\nn\modules\module.py:1110, in Module._call_impl(self, *input, **kwargs)  
    1106 # If we don't have any hooks, we want to skip the rest of the logic in  
    1107 # this function, and just call forward.  
    1108 if not (self._backward_hooks or self._forward_hooks or self._forward_pre_hooks or _global  
_backward_hooks  
        or _global_forward_hooks or _global_forward_pre_hooks):  
-> 1110     return forward_call(*input, **kwargs)  
    1111 # Do not call functions when jit is used  
    1112 full_backward_hooks, non_full_backward_hooks = [], []
```

```

File ~\mambaforge\envs\ds\lib\site-packages\torch\nn\modules\container.py:141, in Sequential.forward(self, input)
    139 def forward(self, input):
    140     for module in self:
--> 141         input = module(input)
    142     return input

File ~\mambaforge\envs\ds\lib\site-packages\torch\nn\modules\module.py:1110, in Module._call_impl(self, *input, **kwargs)
   1106 # If we don't have any hooks, we want to skip the rest of the logic in
   1107 # this function, and just call forward.
   1108 if not (self._backward_hooks or self._forward_hooks or self._forward_pre_hooks or _global_
_backward_hooks
   1109         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1110     return forward_call(*input, **kwargs)
   1111 # Do not call functions when jit is used
   1112 full_backward_hooks, non_full_backward_hooks = [], []

File ~\mambaforge\envs\ds\lib\site-packages\torch\nn\modules\conv.py:447, in Conv2d.forward(self, input)
   446 def forward(self, input: Tensor) -> Tensor:
--> 447     return self._conv_forward(input, self.weight, self.bias)

File ~\mambaforge\envs\ds\lib\site-packages\torch\nn\modules\conv.py:443, in Conv2d._conv_forward(self, input, weight, bias)
   439 if self.padding_mode != 'zeros':
   440     return F.conv2d(F.pad(input, self._reversed_padding_repeated_twice, mode=self.padding_
mode),
   441                     weight, bias, self.stride,
   442                     _pair(0), self.dilation, self.groups)
--> 443 return F.conv2d(input, weight, bias, self.stride,
   444                     self.padding, self.dilation, self.groups)

```

RuntimeError: Expected all tensors to be on the same device, but found at least two devices, cpu and cuda:0! (when checking argument for argument weight in method wrapper\_\_slow\_conv2d\_forward)

Oh my goodness...

Despite our preparations our custom image and model are on different devices.

And we get the error:

```

RuntimeError: Expected all tensors to be on the same device, but found at least two devices, cpu and cuda:0!
(when checking argument for argument weight in method wrapper__slow_conv2d_forward)

```

Let's fix that by putting our `custom_image_transformed` on the target device.

In [70]:

```

-----
RuntimeError                                     Traceback (most recent call last)
Input In [70], in <cell line: 2>()
      1 model_1.eval()
      2 with torch.inference_mode():
--> 3     custom_image_pred = model_1(custom_image_transformed.to(device))

File ~\mambaforge\envs\ds\lib\site-packages\torch\nn\modules\module.py:1110, in Module._call_impl
(self, *input, **kwargs)
    1106 # If we don't have any hooks, we want to skip the rest of the logic in
    1107 # this function, and just call forward.
    1108 if not (self._backward_hooks or self._forward_hooks or self._forward_pre_hooks or _global
_backward_hooks
    1109         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1110     return forward_call(*input, **kwargs)
    1111 # Do not call functions when jit is used
    1112 full_backward_hooks, non_full_backward_hooks = [], []

Input In [45], in TinyVGG.forward(self, x)
    42 x = self.conv_block_2(x)
    43 # print(x.shape)
--> 44 x = self.classifier(x)
    45 # print(x.shape)
    46 return x

File ~\mambaforge\envs\ds\lib\site-packages\torch\nn\modules\module.py:1110, in Module._call_impl
(self, *input, **kwargs)
    1106 # If we don't have any hooks, we want to skip the rest of the logic in
    1107 # this function, and just call forward.
    1108 if not (self._backward_hooks or self._forward_hooks or self._forward_pre_hooks or _global
_backward_hooks
    1109         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1110     return forward_call(*input, **kwargs)
    1111 # Do not call functions when jit is used
    1112 full_backward_hooks, non_full_backward_hooks = [], []

File ~\mambaforge\envs\ds\lib\site-packages\torch\nn\modules\container.py:141, in Sequential.forward(self, input)
    139 def forward(self, input):
    140     for module in self:
--> 141         input = module(input)

```

```
142     return input
```

```
File ~\mambaforge\envs\ds\lib\site-packages\torch\nn\modules\module.py:1110, in Module._call_impl
(self, *input, **kwargs)
 1106 # If we don't have any hooks, we want to skip the rest of the logic in
 1107 # this function, and just call forward.
 1108 if not (self._backward_hooks or self._forward_hooks or self._forward_pre_hooks or _global
_backward_hooks
 1109         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1110     return forward_call(*input, **kwargs)
 1111 # Do not call functions when jit is used
 1112 full_backward_hooks, non_full_backward_hooks = [], []
```

```
File ~\mambaforge\envs\ds\lib\site-packages\torch\nn\modules\linear.py:103, in Linear.forward(self, input)
 102 def forward(self, input: Tensor) -> Tensor:
--> 103     return F.linear(input, self.weight, self.bias)
```

RuntimeError: mat1 and mat2 shapes cannot be multiplied (10x256 and 2560x3)

What now?

It looks like we're getting a shape error.

Why might this be?

We converted our custom image to be the same size as the images our model was trained on...

Oh wait...

There's one dimension we forgot about.

The batch size.

Our model expects image tensors with a batch size dimension at the start (`NCHW` where `N` is the batch size).

Except our custom image is currently only `CHW`.

We can add a batch size dimension using `torch.unsqueeze(dim=0)` to add an extra dimension our image and *finally* make a prediction.

Essentially we'll be telling our model to predict on a single image (an image with a `batch_size` of 1).

In [71]:





```
Custom image transformed shape: torch.Size([3, 64, 64])
Unsqueezed custom image shape: torch.Size([1, 3, 64, 64])
```

Yes!!!

It looks like it worked!

**Note:** What we've just gone through are three of the classical and most common deep learning and PyTorch issues:

1. **Wrong datatypes** - our model expects `torch.float32` where our original custom image was `uint8`.
2. **Wrong device** - our model was on the target `device` (in our case, the GPU) whereas our target data hadn't been moved to the target `device` yet.
3. **Wrong shapes** - our model expected an input image of shape `[N, C, H, W]` or `[batch_size, color_channels, height, width]` whereas our custom image tensor was of shape `[color_channels, height, width]`.

Keep in mind, these errors aren't just for predicting on custom images.

They will be present with almost every kind of data type (text, audio, structured data) and problem you work with.

Now let's take a look at our model's predictions.

In [72]:

```
tensor([[ 0.1161,  0.0213, -0.1422]], device='cuda:0')
```

Out[72]:

Alright, these are still in *logit form* (the raw outputs of a model are called logits).

Let's convert them from logits -> prediction probabilities -> prediction labels.

In [73]:



```
Prediction logits: tensor([[ 0.1161,  0.0213, -0.1422]], device='cuda:0')
Prediction probabilities: tensor([[0.3729, 0.3392, 0.2880]], device='cuda:0')
Prediction label: tensor([0], device='cuda:0')
```

Alright!

Looking good.

But of course our prediction label is still in index/tensor form.

We can convert it to a string class name prediction by indexing on the `class_names` list.

In [74]:

Out[74]:

'pizza'

Wow.

It looks like the model gets the prediction right, even though it was performing poorly based on our evaluation metrics.

**Note:** The model in its current form will predict "pizza", "steak" or "sushi" no matter what image it's given. If you wanted your model to predict on a different class, you'd have to train it to do so.

But if we check the `custom_image_pred_probs`, we'll notice that the model gives almost equal weight (the values are similar) to every class.

In [75]:

Out[75]:

```
tensor([[0.3729, 0.3392, 0.2880]], device='cuda:0')
```

Having prediction probabilities this similar could mean a couple of things:

1. The model is trying to predict all three classes at the same time (there may be an image containing pizza, steak and sushi).
2. The model doesn't really know what it wants to predict and is in turn just assigning similar values to each of the classes.

Our case is number 2, since our model is poorly trained, it is basically *guessing* the prediction.

### 11.3 Putting custom image prediction together: building a function

Doing all of the above steps every time you'd like to make a prediction on a custom image would quickly become tedious.

So let's put them all together in a function we can easily use over and over again.

Specifically, let's make a function that:

1. Takes in a target image path and converts to the right datatype for our model (`torch.float32`).
2. Makes sure the target image pixel values are in the range `[0, 1]`.
3. Transforms the target image if necessary.
4. Makes sure the model is on the target device.
5. Makes a prediction on the target image with a trained model (ensuring the image is the right size and on the same device as the model).
6. Converts the model's output logits to prediction probabilities.
7. Converts the prediction probabilities to prediction labels.
8. Plots the target image alongside the model prediction and prediction probability.

A fair few steps but we've got this!

In [76]:























What a nice looking function, let's test it out.

In [77]:







Two thumbs up again!

Looks like our model got the prediction right just by guessing.

This won't always be the case with other images though...

The image is pixelated too because we resized it to `[64, 64]` using `custom_image_transform`.

**Exercise:** Try making a prediction with one of your own images of pizza, steak or sushi and see what happens.

## Main takeaways

We've covered a fair bit in this module.

Let's summarise it with a few dot points.

- PyTorch has many in-built functions to deal with all kinds of data, from vision to text to audio to recommendation systems.
- If PyTorch's built-in data loading functions don't suit your requirements, you can write code to create your own custom datasets by subclassing `torch.utils.data.Dataset`.
- `torch.utils.data.DataLoader`'s in PyTorch help turn your `Dataset`'s into iterables that can be used when training and testing a model.
- A lot of machine learning is dealing with the balance between **overfitting** and **underfitting** (we discussed different methods for each above, so a good exercise would be to research more and writing code to try out the different techniques).
- Predicting on your own custom data with a trained model is possible, as long as you format the data into a similar format to what the model was trained on. Make sure you take care of the three big PyTorch and deep learning

errors:

- a. **Wrong datatypes** - Your model expected `torch.float32` when your data is `torch.uint8`.
- b. **Wrong data shapes** - Your model expected `[batch_size, color_channels, height, width]` when your data is `[color_channels, height, width]`.
- c. **Wrong devices** - Your model is on the GPU but your data is on the CPU.

## Exercises

All of the exercises are focused on practicing the code in the sections above.

You should be able to complete them by referencing each section or by following the resource(s) linked.

All exercises should be completed using [device-agnostic code](#).

### Resources:

- [Exercise template notebook for 04](#)
  - [Example solutions notebook for 04](#) (try the exercises *before* looking at this)
1. Our models are underperforming (not fitting the data well). What are 3 methods for preventing underfitting? Write them down and explain each with a sentence.
  2. Recreate the data loading functions we built in sections 1, 2, 3 and 4. You should have train and test `DataLoader`'s ready to use.
  3. Recreate `model_0` we built in section 7.
  4. Create training and testing functions for `model_0`.
  5. Try training the model you made in exercise 3 for 5, 20 and 50 epochs, what happens to the results?
    - Use `torch.optim.Adam()` with a learning rate of 0.001 as the optimizer.
  6. Double the number of hidden units in your model and train it for 20 epochs, what happens to the results?
  7. Double the data you're using with your model and train it for 20 epochs, what happens to the results?
    - **Note:** You can use the [custom data creation notebook](#) to scale up your Food101 dataset.
    - You can also find the [already formatted double data \(20% instead of 10% subset\)](#) dataset on GitHub, you will need to write download code like in exercise 2 to get it into this notebook.
  8. Make a prediction on your own custom image of pizza/steak/sushi (you could even download one from the internet) and share your prediction.

Does the model you trained in exercise 7 get it right?

- If not, what do you think you could do to improve it?

## Extra-curriculum

- To practice your knowledge of PyTorch `Dataset`'s and `DataLoader`'s through PyTorch [datasets and dataloaders tutorial notebook](#).
- Spend 10-minutes reading the [PyTorch `torchvision.transforms` documentation](#).
  - You can see demos of transforms in action in the [illustrations of transforms tutorial](#).
- Spend 10-minutes reading the PyTorch `torchvision.datasets` documentation.
  - What are some datasets that stand out to you?
  - How could you try building a model on these?
- [TorchData is currently in beta](#) (as of April 2022), it'll be a future way of loading data in PyTorch, but you can start to check it out now.
- To speed up deep learning models, you can do a few tricks to improve compute, memory and overhead computations, for more read the post [\*Making Deep Learning Go Brrrr From First Principles\*](#) by Horace He.

Previous

03. PyTorch Computer Vision

Next

05. PyTorch Going Modular

[View Source Code](#) | [View Slides](#)

## 05. PyTorch Going Modular

This section answers the question, "how do I turn my notebook code into Python scripts?"

To do so, we're going to turn the most useful code cells in notebook 04. PyTorch Custom Datasets into a series of Python scripts saved to a directory called `going_modular`.

### What is going modular?

Going modular involves turning notebook code (from a Jupyter Notebook or Google Colab notebook) into a series of different Python scripts that offer similar functionality.

For example, we could turn our notebook code from a series of cells into the following Python files:

- `data_setup.py` - a file to prepare and download data if needed.
- `engine.py` - a file containing various training functions.
- `model_builder.py` or `model.py` - a file to create a PyTorch model.
- `train.py` - a file to leverage all other files and train a target PyTorch model.
- `utils.py` - a file dedicated to helpful utility functions.

**Note:** The naming and layout of the above files will depend on your use case and code requirements. Python scripts are as general as individual notebook cells, meaning, you could create one for almost any kind of functionality.

### Why would you want to go modular?

Notebooks are fantastic for iteratively exploring and running experiments quickly.

However, for larger scale projects you may find Python scripts more reproducible and easier to run.

Though this is a debated topic, as companies like Netflix have shown how they use notebooks for production code.

**Production code** is code that runs to offer a service to someone or something.

For example, if you have an app running online that other people can access and use, the code running that app is considered **production code**.

And libraries like fast.ai's `nb-dev` (short for notebook development) enable you to write whole Python libraries (including documentation) with Jupyter Notebooks.

## Pros and cons of notebooks vs Python scripts

There's arguments for both sides.

But this list sums up a few of the main topics.

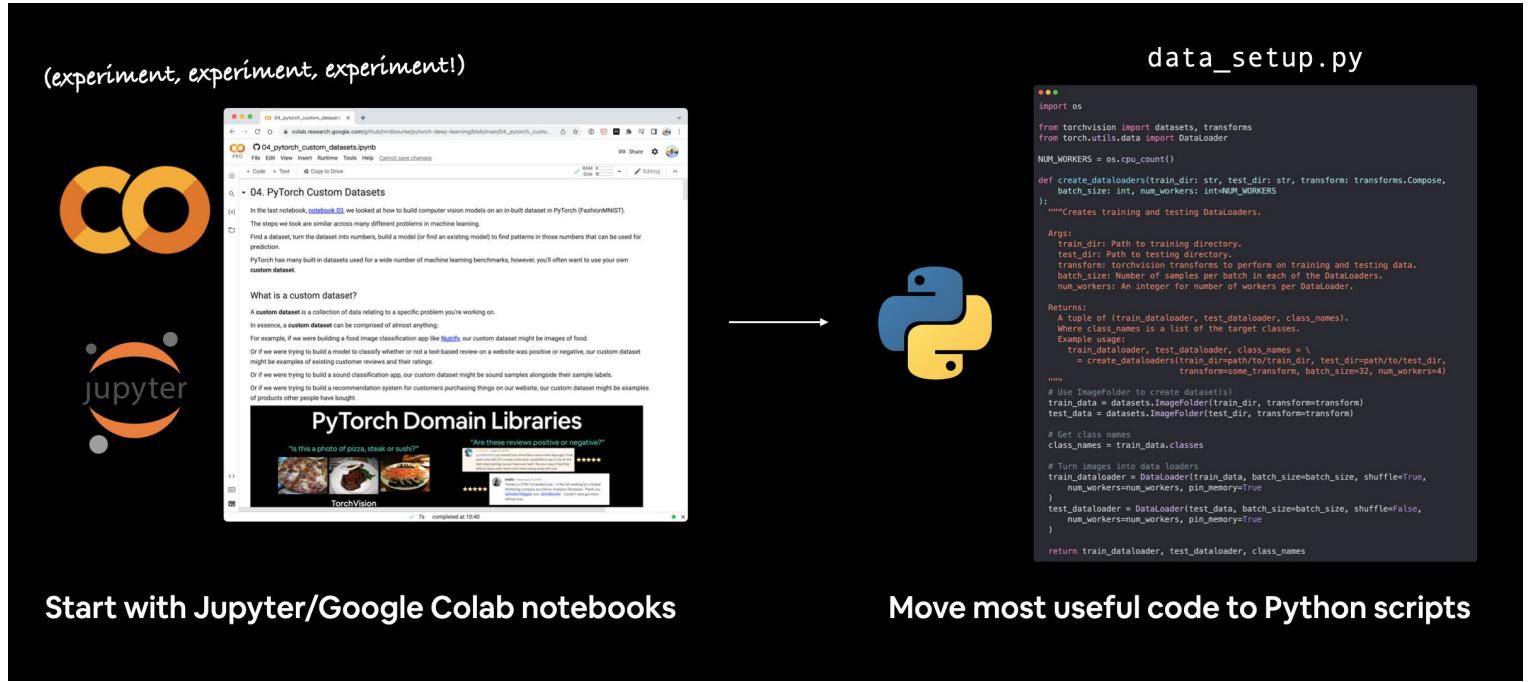
	<b>Pros</b>	<b>Cons</b>
<b>Notebooks</b>	<ul style="list-style-type: none"> <li>Easy to experiment/get started</li> <li>Easy to share (e.g. a link to a Google Colab notebook)</li> <li>Very visual</li> </ul>	<ul style="list-style-type: none"> <li>Versioning can be hard</li> <li>Hard to use only specific parts</li> <li>Text and graphics can get in the way of code</li> </ul>

	<b>Pros</b>	<b>Cons</b>
<b>Python scripts</b>	<ul style="list-style-type: none"> <li>Can package code together (saves rewriting similar code across different notebooks)</li> <li>Can use git for versioning</li> <li>Many open source projects use scripts</li> <li>Larger projects can be run on cloud vendors (not as much support for notebooks)</li> </ul>	<ul style="list-style-type: none"> <li>Experimenting isn't as visual (usually have to run the whole script rather than one cell)</li> </ul>

## My workflow

I usually start machine learning projects in Jupyter/Google Colab notebooks for quick experimentation and visualization.

Then when I've got something working, I move the most useful pieces of code to Python scripts.



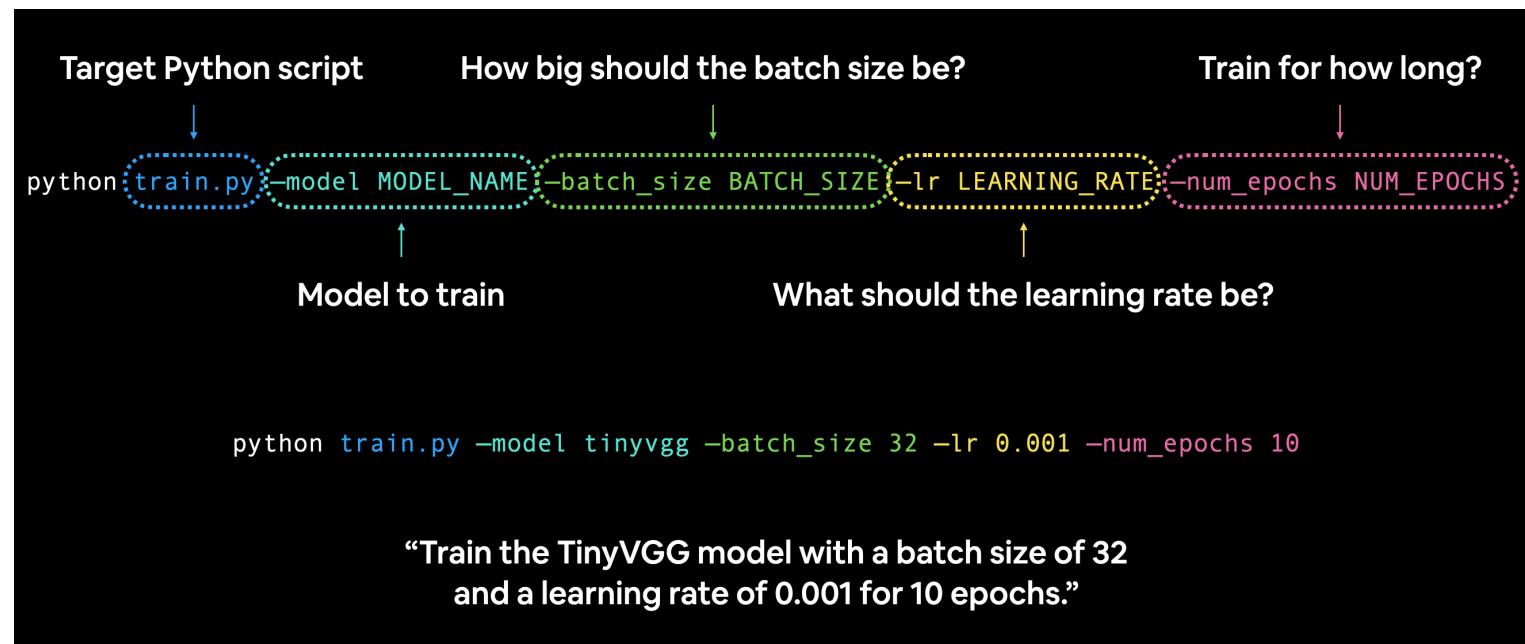
*There are many possible workflows for writing machine learning code. Some prefer to start with scripts, others (like me) prefer to start with notebooks and go to scripts later on.*

## PyTorch in the wild

In your travels, you'll see many code repositories for PyTorch-based ML projects have instructions on how to run the PyTorch code in the form of Python scripts.

For example, you might be instructed to run code like the following in a terminal/command line to train a model:

```
python train.py --model MODEL_NAME --batch_size BATCH_SIZE --lr LEARNING_RATE --num_epochs NUM_EPOCHS
```



*Running a PyTorch `train.py` script on the command line with various hyperparameter settings.*

In this case, `train.py` is the target Python script, it'll likely contain functions to train a PyTorch model.

And `--model`, `--batch_size`, `--lr` and `--num_epochs` are known as argument flags.

You can set these to whatever values you like and if they're compatible with `train.py`, they'll work, if not, they'll error.

For example, let's say we wanted to train our TinyVGG model from notebook 04 for 10 epochs with a batch size of 32 and a learning rate of 0.001:

```
python train.py --model tinyvgg --batch_size 32 --lr 0.001 --num_epochs 10
```

You could setup any number of these argument flags in your `train.py` script to suit your needs.

The PyTorch blog post for training state-of-the-art computer vision models uses this style.

Using our standard [training reference script](#), we can train a ResNet50 using the following command:

```
torchrun --nproc_per_node=8 train.py --model resnet50 --batch-size 128 --lr 0.5 \
--lr-scheduler cosineannealinglr --lr-warmup-epochs 5 --lr-warmup-method linear \
--auto-augment ta_wide --epochs 600 --random-erase 0.1 --weight-decay 0.00002 \
--norm-weight-decay 0.0 --label-smoothing 0.1 --mixup-alpha 0.2 --cutmix-alpha 1.0 \
--train-crop-size 176 --model-ema --val-resize-size 232 --ra-sampler --ra-reps 4
```

*PyTorch command line training script recipe for training state-of-the-art computer vision models with 8 GPUs.*

Source: PyTorch blog.

## What we're going to cover

The main concept of this section is: **turn useful notebook code cells into reusable Python files**.

Doing this will save us writing the same code over and over again.

There are two notebooks for this section:

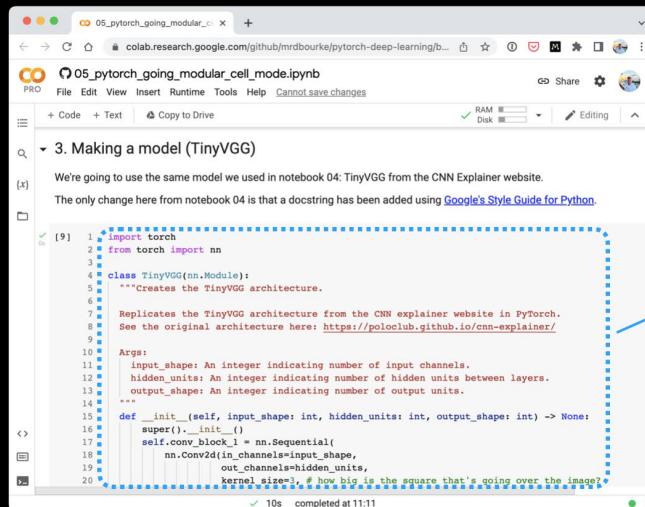
- 1. 05. Going Modular: Part 1 (cell mode)** - this notebook is run as a traditional Jupyter Notebook/Google Colab notebook and is a condensed version of notebook 04.
- 2. 05. Going Modular: Part 2 (script mode)** - this notebook is the same as number 1 but with added functionality to turn each of the major sections into Python scripts, such as, `data_setup.py` and `train.py`.

The text in this document focuses on the code cells 05. Going Modular: Part 2 (script mode), the ones with `%%writefile ...` at the top.

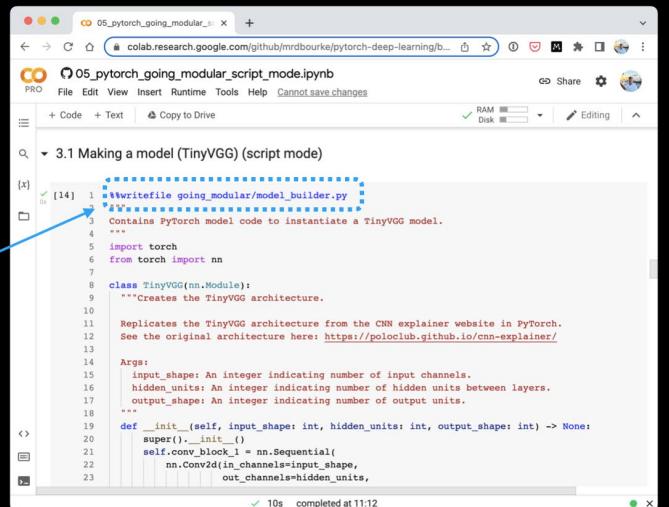
## Why two parts?

Because sometimes the best way to learn something is to see how it *differs* from something else.

If you run each notebook side-by-side you'll see how they differ and that's where the key learnings are.



Notebook 05: Cell mode



Notebook 05: Script mode  
(turns useful code into Python scripts)

Running the two notebooks for section 05 side-by-side. You'll notice that the script mode notebook has extra code

*cells to turn code from the cell mode notebook into Python scripts.*

## What we're working towards

By the end of this section we want to have two things:

1. The ability to train the model we built in notebook 04 (Food Vision Mini) with one line of code on the command line: `python train.py`.
2. A directory structure of reusable Python scripts, such as:

```
going_modular/
├── going_modular/
│   ├── data_setup.py
│   ├── engine.py
│   ├── model_builder.py
│   ├── train.py
│   └── utils.py
└── models/
    ├── 05_going_modular_cell_mode_tinyvgg_model.pth
    └── 05_going_modular_script_mode_tinyvgg_model.pth
└── data/
    └── pizza_steak_sushi/
        ├── train/
        │   ├── pizza/
        │   │   ├── image01.jpeg
        │   │   └── ...
        │   ├── steak/
        │   └── sushi/
        └── test/
            ├── pizza/
            ├── steak/
            └── sushi/
```

## Things to note

- **Docstrings** - Writing reproducible and understandable code is important. And with this in mind, each of the functions/classes we'll be putting into scripts has been created with Google's Python docstring style in mind.
- **Imports at the top of scripts** - Since all of the Python scripts we're going to create could be considered a small program on their own, all of the scripts require their input modules be imported at the start of the script for example:

```
# Import modules required for train.py
import os
import torch
import data_setup, engine, model_builder, utils

from torchvision import transforms
```

# Where can you get help?

All of the materials for this course are available on GitHub.

If you run into trouble, you can ask a question on the course GitHub Discussions page.

And of course, there's the PyTorch documentation and PyTorch developer forums, a very helpful place for all things PyTorch.

## 0. Cell mode vs. script mode

A cell mode notebook such as 05. Going Modular Part 1 (cell mode) is a notebook run normally, each cell in the notebook is either code or markdown.

A script mode notebook such as 05. Going Modular Part 2 (script mode) is very similar to a cell mode notebook, however, many of the code cells may be turned into Python scripts.

**Note:** You don't *need* to create Python scripts via a notebook, you can create them directly through an IDE (integrated developer environment) such as VS Code. Having the script mode notebook as part of this section is just to demonstrate one way of going from notebooks to Python scripts.

## 1. Get data

Getting the data in each of the 05 notebooks happens the same as in notebook 04.

A call is made to GitHub via Python's `requests` module to download a `.zip` file and unzip it.

```
import os
import requests
import zipfile
from pathlib import Path

# Setup path to data folder
data_path = Path("data/")
image_path = data_path / "pizza_steak_sushi"

# If the image folder doesn't exist, download it and prepare it...
if image_path.is_dir():
    print(f"{image_path} directory exists.")
else:
    print(f"Did not find {image_path} directory, creating one...")
    image_path.mkdir(parents=True, exist_ok=True)

# Download pizza, steak, sushi data
with open(data_path / "pizza_steak_sushi.zip", "wb") as f:
    request = requests.get("https://github.com/mrdbourke/pytorch-deep-
learning/raw/main/data/pizza_steak_sushi.zip")
    print("Downloading pizza, steak, sushi data...")
```

```

f.write(request.content)

# Unzip pizza, steak, sushi data
with zipfile.ZipFile(data_path / "pizza_steak_sushi.zip", "r") as zip_ref:
    print("Unzipping pizza, steak, sushi data...")
    zip_ref.extractall(image_path)

# Remove zip file
os.remove(data_path / "pizza_steak_sushi.zip")

```

This results in having a file called `data` that contains another directory called `pizza_steak_sushi` with images of pizza, steak and sushi in standard image classification format.

```

data/
└── pizza_steak_sushi/
    ├── train/
    │   ├── pizza/
    │   │   ├── train_image01.jpeg
    │   │   ├── test_image02.jpeg
    │   │   └── ...
    │   ├── steak/
    │   │   └── ...
    │   └── sushi/
    │       └── ...
    └── test/
        ├── pizza/
        │   ├── test_image01.jpeg
        │   └── test_image02.jpeg
        ├── steak/
        └── sushi/

```

## 2. Create Datasets and DataLoaders ( `data_setup.py` )

Once we've got data, we can then turn it into PyTorch `Dataset`'s and `DataLoader`'s (one for training data and one for testing data).

We convert the useful `Dataset` and `DataLoader` creation code into a function called `create_dataloaders()`.

And we write it to file using the line `%%writefile going_modular/data_setup.py`.

### `data_setup.py`

```

%%writefile going_modular/data_setup.py
"""
Contains functionality for creating PyTorch DataLoaders for
image classification data.
"""
import os

from torchvision import datasets, transforms
from torch.utils.data import DataLoader

NUM_WORKERS = os.cpu_count()

def create_dataloaders(

```

```

train_dir: str,
test_dir: str,
transform: transforms.Compose,
batch_size: int,
num_workers: int=NUM_WORKERS
):
    """Creates training and testing DataLoaders.

    Takes in a training directory and testing directory path and turns
    them into PyTorch Datasets and then into PyTorch DataLoaders.

    Args:
        train_dir: Path to training directory.
        test_dir: Path to testing directory.
        transform: torchvision transforms to perform on training and testing data.
        batch_size: Number of samples per batch in each of the DataLoaders.
        num_workers: An integer for number of workers per DataLoader.

    Returns:
        A tuple of (train_dataloader, test_dataloader, class_names).
        Where class_names is a list of the target classes.

    Example usage:
        train_dataloader, test_dataloader, class_names = \
            = create_dataloaders(train_dir=path/to/train_dir,
                                test_dir=path/to/test_dir,
                                transform=some_transform,
                                batch_size=32,
                                num_workers=4)
    """
    # Use ImageFolder to create dataset(s)
    train_data = datasets.ImageFolder(train_dir, transform=transform)
    test_data = datasets.ImageFolder(test_dir, transform=transform)

    # Get class names
    class_names = train_data.classes

    # Turn images into data loaders
    train_dataloader = DataLoader(
        train_data,
        batch_size=batch_size,
        shuffle=True,
        num_workers=num_workers,
        pin_memory=True,
    )
    test_dataloader = DataLoader(
        test_data,
        batch_size=batch_size,
        shuffle=True,
        num_workers=num_workers,
        pin_memory=True,
    )

    return train_dataloader, test_dataloader, class_names

```

If we'd like to make `DataLoader`'s we can now use the function within `data_setup.py` like so:

```

# Import data_setup.py
from going_modular import data_setup

# Create train/test dataloader and get class names as a list
train_dataloader, test_dataloader, class_names = data_setup.create_dataloaders(...)

```

### 3. Making a model ( `model_builder.py` )

Over the past few notebooks (notebook 03 and notebook 04), we've built the TinyVGG model a few times.

So it makes sense to put the model into its file so we can reuse it again and again.

Let's put our `TinyVGG()` model class into a script with the line `%%writefile going_modular/model_builder.py`:

#### `model_builder.py`

```
%%writefile going_modular/model_builder.py
"""
Contains PyTorch model code to instantiate a TinyVGG model.
"""

import torch
from torch import nn

class TinyVGG(nn.Module):
    """Creates the TinyVGG architecture.

    Replicates the TinyVGG architecture from the CNN explainer website in PyTorch.
    See the original architecture here: https://poloclub.github.io/cnn-explainer/

    Args:
        input_shape: An integer indicating number of input channels.
        hidden_units: An integer indicating number of hidden units between layers.
        output_shape: An integer indicating number of output units.
    """
    def __init__(self, input_shape: int, hidden_units: int, output_shape: int) -> None:
        super().__init__()
        self.conv_block_1 = nn.Sequential(
            nn.Conv2d(in_channels=input_shape,
                      out_channels=hidden_units,
                      kernel_size=3,
                      stride=1,
                      padding=0),
            nn.ReLU(),
            nn.Conv2d(in_channels=hidden_units,
                      out_channels=hidden_units,
                      kernel_size=3,
                      stride=1,
                      padding=0),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2,
                         stride=2)
        )
        self.conv_block_2 = nn.Sequential(
            nn.Conv2d(hidden_units, hidden_units, kernel_size=3, padding=0),
            nn.ReLU(),
            nn.Conv2d(hidden_units, hidden_units, kernel_size=3, padding=0),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            # Where did this in_features shape come from?
            # It's because each layer of our network compresses and changes the shape of our inputs data.
            nn.Linear(in_features=hidden_units*13*13,
                      out_features=output_shape)
    )

Args:
    input_shape: An integer indicating number of input channels.
    hidden_units: An integer indicating number of hidden units between layers.
    output_shape: An integer indicating number of output units.
"""

def __init__(self, input_shape: int, hidden_units: int, output_shape: int) -> None:
    super().__init__()
    self.conv_block_1 = nn.Sequential(
        nn.Conv2d(in_channels=input_shape,
                  out_channels=hidden_units,
                  kernel_size=3,
                  stride=1,
                  padding=0),
        nn.ReLU(),
        nn.Conv2d(in_channels=hidden_units,
                  out_channels=hidden_units,
                  kernel_size=3,
                  stride=1,
                  padding=0),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2,
                     stride=2)
    )
    self.conv_block_2 = nn.Sequential(
        nn.Conv2d(hidden_units, hidden_units, kernel_size=3, padding=0),
        nn.ReLU(),
        nn.Conv2d(hidden_units, hidden_units, kernel_size=3, padding=0),
        nn.ReLU(),
        nn.MaxPool2d(2)
    )
    self.classifier = nn.Sequential(
        nn.Flatten(),
        # Where did this in_features shape come from?
        # It's because each layer of our network compresses and changes the shape of our inputs data.
        nn.Linear(in_features=hidden_units*13*13,
                  out_features=output_shape)
    )
```

```

def forward(self, x: torch.Tensor):
    x = self.conv_block_1(x)
    x = self.conv_block_2(x)
    x = self.classifier(x)
    return x
    # return self.classifier(self.conv_block_2(self.conv_block_1(x))) # <- leverage the benefits of operator
fusion

```

Now instead of coding the TinyVGG model from scratch every time, we can import it using:

```

import torch
# Import model_builder.py
from going_modular import model_builder
device = "cuda" if torch.cuda.is_available() else "cpu"

# Instantiate an instance of the model from the "model_builder.py" script
torch.manual_seed(42)
model = model_builder.TinyVGG(input_shape=3,
                               hidden_units=10,
                               output_shape=len(class_names)).to(device)

```

## 4. Creating `train_step()` and `test_step()` functions and `train()` to combine them

We wrote several training functions in notebook 04:

1. `train_step()` - takes in a model, a `DataLoader`, a loss function and an optimizer and trains the model on the `DataLoader`.
2. `test_step()` - takes in a model, a `DataLoader` and a loss function and evaluates the model on the `DataLoader`.
3. `train()` - performs 1. and 2. together for a given number of epochs and returns a results dictionary.

Since these will be the *engine* of our model training, we can put them all into a Python script called `engine.py` with the line `%%writefile going_modular/engine.py`:

### **engine.py**

```

%%writefile going_modular/engine.py
"""
Contains functions for training and testing a PyTorch model.
"""
import torch

from tqdm.auto import tqdm
from typing import Dict, List, Tuple

def train_step(model: torch.nn.Module,
              dataloader: torch.utils.data.DataLoader,
              loss_fn: torch.nn.Module,
              optimizer: torch.optim.Optimizer,
              device: torch.device) -> Tuple[float, float]:
    """Trains a PyTorch model for a single epoch.

```

Turns a target PyTorch model to training mode and then runs through all of the required training steps (forward pass, loss calculation, optimizer step).

Args:

```
model: A PyTorch model to be trained.
dataloader: A DataLoader instance for the model to be trained on.
loss_fn: A PyTorch loss function to minimize.
optimizer: A PyTorch optimizer to help minimize the loss function.
device: A target device to compute on (e.g. "cuda" or "cpu").
```

Returns:

```
A tuple of training loss and training accuracy metrics.
In the form (train_loss, train_accuracy). For example:
```

```
(0.1112, 0.8743)
```

```
"""
```

```
# Put model in train mode
model.train()
```

```
# Setup train loss and train accuracy values
train_loss, train_acc = 0, 0
```

```
# Loop through data loader data batches
for batch, (X, y) in enumerate(dataloader):
    # Send data to target device
    X, y = X.to(device), y.to(device)
```

```
# 1. Forward pass
y_pred = model(X)
```

```
# 2. Calculate and accumulate loss
loss = loss_fn(y_pred, y)
train_loss += loss.item()
```

```
# 3. Optimizer zero grad
optimizer.zero_grad()
```

```
# 4. Loss backward
loss.backward()
```

```
# 5. Optimizer step
optimizer.step()
```

```
# Calculate and accumulate accuracy metric across all batches
y_pred_class = torch.argmax(torch.softmax(y_pred, dim=1), dim=1)
train_acc += (y_pred_class == y).sum().item() / len(y_pred)
```

```
# Adjust metrics to get average loss and accuracy per batch
train_loss = train_loss / len(dataloader)
train_acc = train_acc / len(dataloader)
return train_loss, train_acc
```

```
def test_step(model: torch.nn.Module,
             dataloader: torch.utils.data.DataLoader,
             loss_fn: torch.nn.Module,
             device: torch.device) -> Tuple[float, float]:
    """Tests a PyTorch model for a single epoch.
```

Turns a target PyTorch model to "eval" mode and then performs a forward pass on a testing dataset.

Args:

## 05. PyTorch Going Modular - Zero to Mastery Learn PyTorch for Deep Learning

```
model: A PyTorch model to be tested.  
dataloader: A DataLoader instance for the model to be tested on.  
loss_fn: A PyTorch loss function to calculate loss on the test data.  
device: A target device to compute on (e.g. "cuda" or "cpu").  
  
Returns:  
A tuple of testing loss and testing accuracy metrics.  
In the form (test_loss, test_accuracy). For example:  
  
(0.0223, 0.8985)  
"""  
# Put model in eval mode  
model.eval()  
  
# Setup test loss and test accuracy values  
test_loss, test_acc = 0, 0  
  
# Turn on inference context manager  
with torch.inference_mode():  
    # Loop through DataLoader batches  
    for batch, (X, y) in enumerate(dataloader):  
        # Send data to target device  
        X, y = X.to(device), y.to(device)  
  
        # 1. Forward pass  
        test_pred_logits = model(X)  
  
        # 2. Calculate and accumulate loss  
        loss = loss_fn(test_pred_logits, y)  
        test_loss += loss.item()  
  
        # Calculate and accumulate accuracy  
        test_pred_labels = test_pred_logits.argmax(dim=1)  
        test_acc += ((test_pred_labels == y).sum().item() / len(test_pred_labels))  
  
# Adjust metrics to get average loss and accuracy per batch  
test_loss = test_loss / len(dataloader)  
test_acc = test_acc / len(dataloader)  
return test_loss, test_acc  
  
def train(model: torch.nn.Module,  
         train_dataloader: torch.utils.data.DataLoader,  
         test_dataloader: torch.utils.data.DataLoader,  
         optimizer: torch.optim.Optimizer,  
         loss_fn: torch.nn.Module,  
         epochs: int,  
         device: torch.device) -> Dict[str, List]:  
    """Trains and tests a PyTorch model.  
  
    Passes a target PyTorch models through train_step() and test_step()  
    functions for a number of epochs, training and testing the model  
    in the same epoch loop.  
  
    Calculates, prints and stores evaluation metrics throughout.  
  
    Args:  
        model: A PyTorch model to be trained and tested.  
        train_dataloader: A DataLoader instance for the model to be trained on.  
        test_dataloader: A DataLoader instance for the model to be tested on.  
        optimizer: A PyTorch optimizer to help minimize the loss function.  
        loss_fn: A PyTorch loss function to calculate loss on both datasets.  
        epochs: An integer indicating how many epochs to train for.  
        device: A target device to compute on (e.g. "cuda" or "cpu").  
  
    Returns:
```

A dictionary of training and testing loss as well as training and testing accuracy metrics. Each metric has a value in a list for each epoch.

```

In the form: {train_loss: [...],
    train_acc: [...],
    test_loss: [...],
    test_acc: [...]}

For example if training for epochs=2:
    {train_loss: [2.0616, 1.0537],
    train_acc: [0.3945, 0.3945],
    test_loss: [1.2641, 1.5706],
    test_acc: [0.3400, 0.2973]}

"""

# Create empty results dictionary
results = {"train_loss": [],
    "train_acc": [],
    "test_loss": [],
    "test_acc": []
}

# Loop through training and testing steps for a number of epochs
for epoch in tqdm(range(epochs)):
    train_loss, train_acc = train_step(model=model,
        dataloader=train_dataloader,
        loss_fn=loss_fn,
        optimizer=optimizer,
        device=device)

    test_loss, test_acc = test_step(model=model,
        dataloader=test_dataloader,
        loss_fn=loss_fn,
        device=device)

    # Print out what's happening
    print(
        f"Epoch: {epoch+1} | "
        f"train_loss: {train_loss:.4f} | "
        f"train_acc: {train_acc:.4f} | "
        f"test_loss: {test_loss:.4f} | "
        f"test_acc: {test_acc:.4f}"
    )

    # Update results dictionary
    results["train_loss"].append(train_loss)
    results["train_acc"].append(train_acc)
    results["test_loss"].append(test_loss)
    results["test_acc"].append(test_acc)

# Return the filled results at the end of the epochs
return results

```

Now we've got the `engine.py` script, we can import functions from it via:

```

# Import engine.py
from going_modular import engine

# Use train() by calling it from engine.py
engine.train(...)
```

## 5. Creating a function to save the model (`utils.py`)

Often you'll want to save a model whilst it's training or after training.

Since we've written the code to save a model a few times now in previous notebooks, it makes sense to turn it into a function and save it to file.

It's common practice to store helper functions in a file called `utils.py` (short for utilities).

Let's save our `save_model()` function to a file called `utils.py` with the line `%%writefile going_modular/utils.py`:

### **utils.py**

```
%%writefile going_modular/utils.py
"""
Contains various utility functions for PyTorch model training and saving.
"""

import torch
from pathlib import Path

def save_model(model: torch.nn.Module,
              target_dir: str,
              model_name: str):
    """Saves a PyTorch model to a target directory.

Args:
    model: A target PyTorch model to save.
    target_dir: A directory for saving the model to.
    model_name: A filename for the saved model. Should include
        either ".pth" or ".pt" as the file extension.

Example usage:
    save_model(model=model_0,
               target_dir="models",
               model_name="05_going_modular_tingvgg_model.pth")
"""
# Create target directory
target_dir_path = Path(target_dir)
target_dir_path.mkdir(parents=True,
                      exist_ok=True)

# Create model save path
assert model_name.endswith(".pth") or model_name.endswith(".pt"), "model_name should end with '.pt' or \
'.pth'"
model_save_path = target_dir_path / model_name

# Save the model state_dict()
print(f"[INFO] Saving model to: {model_save_path}")
torch.save(obj=model.state_dict(),
           f=model_save_path)
```

Now if we wanted to use our `save_model()` function, instead of writing it all over again, we can import it and use it via:

```
# Import utils.py
from going_modular import utils

# Save a model to file
save_model(model=...
           target_dir=...)
```

```
model_name=...)
```

## 6. Train, evaluate and save the model ( `train.py` )

As previously discussed, you'll often come across PyTorch repositories that combine all of their functionality together in a `train.py` file.

This file is essentially saying "train the model using whatever data is available".

In our `train.py` file, we'll combine all of the functionality of the other Python scripts we've created and use it to train a model.

This way we can train a PyTorch model using a single line of code on the command line:

```
python train.py
```

To create `train.py` we'll go through the following steps:

1. Import the various dependencies, namely `torch`, `os`, `torchvision.transforms` and all of the scripts from the `going_modular` directory, `data_setup`, `engine`, `model_builder`, `utils`.
2. **Note:** Since `train.py` will be *inside* the `going_modular` directory, we can import the other modules via `import ...` rather than `from going_modular import ...`.
3. Setup various hyperparameters such as batch size, number of epochs, learning rate and number of hidden units (these could be set in the future via Python's `argparse`).
4. Setup the training and test directories.
5. Setup device-agnostic code.
6. Create the necessary data transforms.
7. Create the DataLoaders using `data_setup.py`.
8. Create the model using `model_builder.py`.
9. Setup the loss function and optimizer.
10. Train the model using `engine.py`.
11. Save the model using `utils.py`.

And we can create the file from a notebook cell using the line `%%writefile going_modular/train.py`:

### **train.py**

```
%%writefile going_modular/train.py
""""
```

## 05. PyTorch Going Modular - Zero to Mastery Learn PyTorch for Deep Learning

```
Trains a PyTorch image classification model using device-agnostic code.
"""

import os
import torch
import data_setup, engine, model_builder, utils

from torchvision import transforms

# Setup hyperparameters
NUM_EPOCHS = 5
BATCH_SIZE = 32
HIDDEN_UNITS = 10
LEARNING_RATE = 0.001

# Setup directories
train_dir = "data/pizza_steak_sushi/train"
test_dir = "data/pizza_steak_sushi/test"

# Setup target device
device = "cuda" if torch.cuda.is_available() else "cpu"

# Create transforms
data_transform = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.ToTensor()
])

# Create DataLoaders with help from data_setup.py
train_dataloader, test_dataloader, class_names = data_setup.create_dataloaders(
    train_dir=train_dir,
    test_dir=test_dir,
    transform=data_transform,
    batch_size=BATCH_SIZE
)

# Create model with help from model_builder.py
model = model_builder.TinyVGG(
    input_shape=3,
    hidden_units=HIDDEN_UNITS,
    output_shape=len(class_names)
).to(device)

# Set loss and optimizer
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(),
                             lr=LEARNING_RATE)

# Start training with help from engine.py
engine.train(model=model,
             train_dataloader=train_dataloader,
             test_dataloader=test_dataloader,
             loss_fn=loss_fn,
             optimizer=optimizer,
             epochs=NUM_EPOCHS,
             device=device)

# Save the model with help from utils.py
utils.save_model(model=model,
                 target_dir="models",
                 model_name="05_going_modular_script_mode_tinyvgg_model.pth")
```

Woohoo!

Now we can train a PyTorch model by running the following line on the command line:

```
python train.py
```

Doing this will leverage all of the other code scripts we've created.

And if we wanted to, we could adjust our `train.py` file to use argument flag inputs with Python's `argparse` module, this would allow us to provide different hyperparameter settings like previously discussed:

```
python train.py --model MODEL_NAME --batch_size BATCH_SIZE --lr LEARNING_RATE --num_epochs NUM_EPOCHS
```

## Exercises

### Resources:

- Exercise template notebook for 05
- Example solutions notebook for 05
  - Live coding run through of solutions notebook for 05 on YouTube

### Exercises:

1. Turn the code to get the data (from section 1. Get Data above) into a Python script, such as `get_data.py`.
  - When you run the script using `python get_data.py` it should check if the data already exists and skip downloading if it does.
  - If the data download is successful, you should be able to access the `pizza_steak_sushi` images from the `data` directory.
2. Use Python's `argparse` module to be able to send the `train.py` custom hyperparameter values for training procedures.
  - Add an argument for using a different:
    - Training/testing directory
    - Learning rate
    - Batch size
    - Number of epochs to train for
    - Number of hidden units in the TinyVGG model
  - Keep the default values for each of the above arguments as what they already are (as in notebook 05).
  - For example, you should be able to run something similar to the following line to train a TinyVGG model

with a learning rate of 0.003 and a batch size of 64 for 20 epochs: `python train.py --learning_rate 0.003 --batch_size 64 --num_epochs 20`.

- **Note:** Since `train.py` leverages the other scripts we created in section 05, such as, `model_builder.py`, `utils.py` and `engine.py`, you'll have to make sure they're available to use too. You can find these in the `going_modular` folder on the course GitHub.

### 3. Create a script to predict (such as `predict.py`) on a target image given a file path with a saved model.

- For example, you should be able to run the command `python predict.py some_image.jpeg` and have a trained PyTorch model predict on the image and return its prediction.
- To see example prediction code, check out the predicting on a custom image section in notebook 04.
- You may also have to write code to load in a trained model.

## Extra-curriculum

- To learn more about structuring a Python project, check out Real Python's guide on Python Application Layouts.
- For ideas on styling your PyTorch code, check out the PyTorch style guide by Igor Susmelj (much of styling in this chapter is based off this guide + various similar PyTorch repositories).
- For an example `train.py` script and various other PyTorch scripts written by the PyTorch team to train state-of-the-art image classification models, check out their `classification` repository on GitHub.

Previous

04. PyTorch Custom Datasets

Next

06. PyTorch Transfer Learning



[View Source Code](#) | [View Slides](#)

## 06. PyTorch Transfer Learning

**Note:** This notebook uses `torchvision`'s new [multi-weight support API \(available in `torchvision v0.13+`\)](#).

We've built a few models by hand so far.

But their performance has been poor.

You might be thinking, **is there a well-performing model that already exists for our problem?**

And in the world of deep learning, the answer is often *yes*.

We'll see how by using a powerful technique called **transfer learning**.

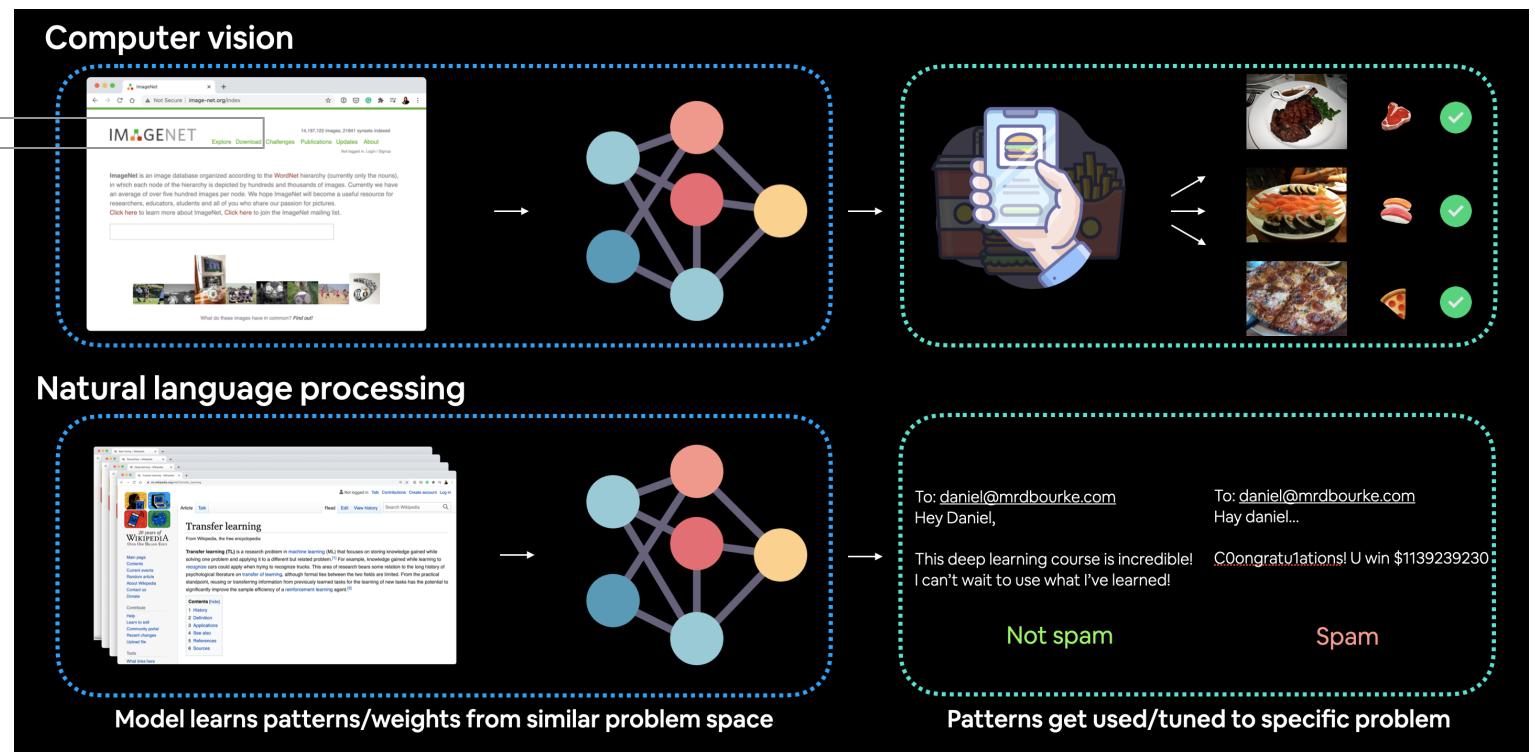
### What is transfer learning?

**Transfer learning** allows us to take the patterns (also called weights) another model has learned from another problem and use them for our own problem.

For example, we can take the patterns a computer vision model has learned from datasets such as [ImageNet](#) (millions of images of different objects) and use them to power our FoodVision Mini model.

Or we could take the patterns from a [language model](#) (a model that's been through large amounts of text to learn a representation of language) and use them as the basis of a model to classify different text samples.

The premise remains: find a well-performing existing model and apply it to your own problem.

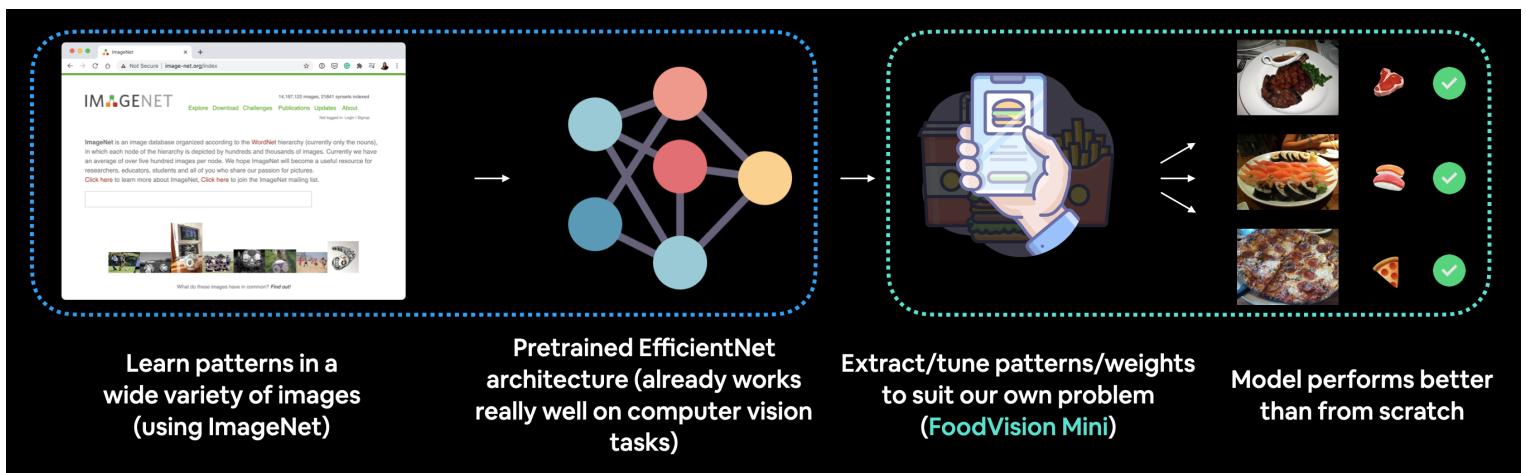


*Example of transfer learning being applied to computer vision and natural language processing (NLP). In the case of computer vision, a computer vision model might learn patterns on millions of images in ImageNet and then use those patterns to infer on another problem. And for NLP, a language model may learn the structure of language by reading all of Wikipedia (and perhaps more) and then apply that knowledge to a different problem.*

## Why use transfer learning?

There are two main benefits to using transfer learning:

1. Can leverage an existing model (usually a neural network architecture) proven to work on problems similar to our own.
2. Can leverage a working model which has **already learned** patterns on similar data to our own. This often results in achieving **great results with less custom data**.



*We'll be putting these to the test for our FoodVision Mini problem, we'll take a computer vision model pretrained on ImageNet and try to leverage its underlying learned representations for classifying images of pizza, steak and sushi.*

Both research and practice support the use of transfer learning too.

A finding from a recent machine learning research paper recommended practitioner's use transfer learning wherever possible.

We also perform an in-depth analysis of the transfer learning setting for Vision Transformers. We conclude that across a wide range of datasets, even if the downstream data of interest appears to only be weakly related to the data used for pre-training, transfer learning remains the best available option. Our analysis also suggests that among similarly performing pre-trained models, for transfer learning a model with more training data should likely be preferred over one with more data augmentation.

*A study into the effects of whether training from scratch or using transfer learning was better from a practitioner's point of view, found transfer learning to be far more beneficial in terms of cost and time. Source: [How to train your ViT? Data, Augmentation, and Regularization in Vision Transformers](#) paper section 6 (conclusion).*

And Jeremy Howard (founder of [fastai](#)) is a big proponent of transfer learning.

The things that really make a difference (transfer learning), if we can do better at transfer learning, it's this world changing thing. Suddenly lots more people can do world-class work with less resources and less data. — [Jeremy Howard on the Lex Fridman Podcast](#)

## Where to find pretrained models

The world of deep learning is an amazing place.

So amazing that many people around the world share their work.

Often, code and pretrained models for the latest state-of-the-art research is released within a few days of publishing.

And there are several places you can find pretrained models to use for your own problems.

Location	What's there?	Link(s)
<b>PyTorch domain libraries</b>	Each of the PyTorch domain libraries ( <code>torchvision</code> , <code>torchtext</code> ) come with pretrained models of some form. The models there work right within PyTorch.	<code>torchvision.models</code> , <code>torchtext.models</code> , <code>torchaudio.models</code> , <code>torchrec.models</code>
<b>HuggingFace Hub</b>	A series of pretrained models on many different domains (vision, text, audio and more) from organizations around the world. There's plenty of different datasets too.	<a href="https://huggingface.co/models">https://huggingface.co/models</a> , <a href="https://huggingface.co/datasets">https://huggingface.co/datasets</a>
<b>timm (PyTorch Image Models) library</b>	Almost all of the latest and greatest computer vision models in PyTorch code as well as plenty of other helpful computer vision features.	<a href="https://github.com/rwightman/pytorch-image-models">https://github.com/rwightman/pytorch-image-models</a>
<b>Paperswithcode</b>	A collection of the latest state-of-the-art machine learning papers with code implementations attached. You can also find benchmarks here of model performance on different tasks.	<a href="https://paperswithcode.com/">https://paperswithcode.com/</a>

The screenshot shows the PyTorch Models and pre-trained weights page. The 'Classification' section contains definitions for various model architectures like AlexNet, VGG, ResNet, etc. It includes a note about backward compatibility for loading serialized `state_dict`.

**PyTorch domains libraries (`torchvision`, `torchtext`, `torchaudio`, `torchrec`).**  
Source: <https://pytorch.org/vision/stable/models.html>

The screenshot shows the GitHub repository for `rwightman/pytorch-image-models`. It displays a list of commits, pull requests, and issues. The repository has 1,383 commits and 18 branches. Topics include efficientnet, resnet, and various image classification models.

**Torch Image Models (timm library).**  
Source: <https://github.com/rwightman/pytorch-image-models>

The screenshot shows the HuggingFace Hub's 'Models' section. It lists various pre-trained models for tasks like Image Classification, Translation, and Sentence Similarity. Examples include distilbert-base-uncased and bert-base-uncased.

**HuggingFace Hub.**  
Source: <https://huggingface.co/models>

The screenshot shows the Paperswithcode SOTA page for Computer Vision. It highlights five main areas: Semantic Segmentation, Image Classification, Object Detection, Image Generation, and Denoising, each with a brief description and a grid of related papers.

**Paperswithcode SOTA.**  
Source: <https://paperswithcode.com/sota>

*With access to such high-quality resources as above, it should be common practice at the start of every deep learning problem you take on to ask, "Does a pretrained model exist for my problem?"*

**Exercise:** Spend 5-minutes going through `torchvision.models` as well as the [HuggingFace Hub Models page](#), what do you find? (there's no right answers here, it's just to practice exploring)

## What we're going to cover

We're going to take a pretrained model from `torchvision.models` and customise it to work on (and hopefully improve) our FoodVision Mini problem.

Topic	Contents
<b>0. Getting setup</b>	We've written a fair bit of useful code over the past few sections, let's download it and make sure we can use it again.
<b>1. Get data</b>	Let's get the pizza, steak and sushi image classification dataset we've been using to try and improve our model's results.
<b>2. Create Datasets and DataLoaders</b>	We'll use the <code>data_setup.py</code> script we wrote in chapter 05. PyTorch Going Modular to setup our DataLoaders.
<b>3. Get and customise a pretrained model</b>	Here we'll download a pretrained model from <code>torchvision.models</code> and customise it to our own problem.
<b>4. Train model</b>	Let's see how the new pretrained model goes on our pizza, steak, sushi dataset. We'll use the training functions we created in the previous chapter.
<b>5. Evaluate the model by plotting loss curves</b>	How did our first transfer learning model go? Did it overfit or underfit?
<b>6. Make predictions on images from the test set</b>	It's one thing to check out a model's evaluation metrics but it's another thing to view its predictions on test samples, let's <i>visualize, visualize, visualize!</i>

## Where can you get help?

All of the materials for this course [are available on GitHub](#).

If you run into trouble, you can ask a question on the course [GitHub Discussions page](#).

And of course, there's the [PyTorch documentation](#) and [PyTorch developer forums](#), a very helpful place for all things PyTorch.

## 0. Getting setup

Let's get started by importing/downloading the required modules for this section.

To save us writing extra code, we're going to be leveraging some of the Python scripts (such as `data_setup.py` and `engine.py`) we created in the previous section, [05. PyTorch Going Modular](#).

Specifically, we're going to download the `going_modular` directory from the `pytorch-deep-learning` repository (if we don't already have it).

We'll also get the `torchinfo` package if it's not available.

`torchinfo` will help later on to give us a visual representation of our model.

**Note:** As of June 2022, this notebook uses the nightly versions of `torch` and `torchvision` as `torchvision v0.13+` is required for using the updated multi-weights API. You can install these using the command below.

In [1]:







```
torch version: 1.13.0.dev20220620+cu113
torchvision version: 0.14.0.dev20220620+cu113
```

In [2]:









Now let's setup device agnostic code.

**Note:** If you're using Google Colab, and you don't have a GPU turned on yet, it's now time to turn one on via

```
Runtime -> Change runtime type -> Hardware accelerator -> GPU .
```

In [3]:

```
'cuda'
```

Out[3]:

## 1. Get data

Before we can start to use **transfer learning**, we'll need a dataset.

To see how transfer learning compares to our previous attempts at model building, we'll download the same dataset we've been using for FoodVision Mini.

Let's write some code to download the `pizza_steak_sushi.zip` dataset from the course GitHub and then unzip it.

We can also make sure if we've already got the data, it doesn't redownload.

In [4]:









```
data/pizza_steak_sushi directory exists.
```

Excellent!

Now we've got the same dataset we've been using previously, a series of images of pizza, steak and sushi in standard image classification format.

Let's now create paths to our training and test directories.

In [5]:

## 2. Create Datasets and DataLoaders

Since we've downloaded the `going_modular` directory, we can use the `data_setup.py` script we created in section 05. [PyTorch Going Modular](#) to prepare and setup our DataLoaders.

But since we'll be using a pretrained model from `torchvision.models`, there's a specific transform we need to prepare our images first.

### 2.1 Creating a transform for `torchvision.models` (manual creation)

**Note:** As of `torchvision v0.13+`, there's an update to how data transforms can be created using `torchvision.models`. I've called the previous method "manual creation" and the new method "auto creation". This notebook showcases both.

When using a pretrained model, it's important that **your custom data going into the model is prepared in the same way as the original training data that went into the model**.

Prior to `torchvision v0.13+`, to create a transform for a pretrained model in `torchvision.models`, the documentation stated:

All pre-trained models expect input images normalized in the same way, i.e. mini-batches of 3-channel RGB images of shape (3 x H x W), where H and W are expected to be at least 224.

The images have to be loaded in to a range of `[0, 1]` and then normalized using `mean = [0.485, 0.456, 0.406]` and `std = [0.229, 0.224, 0.225]`.

You can use the following transform to normalize:

```
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                std=[0.229, 0.224, 0.225])
```

The good news is, we can achieve the above transformations with a combination of:

Transform	Transform required	Code to perform transform
-----------	--------------------	---------------------------

**number**

1	Mini-batches of size <code>[batch_size, 3, height, width]</code> where height and width are at least 224x224^.	<code>torchvision.transforms.Resize()</code> to resize images into <code>[3, 224, 224]</code> ^ and <code>torch.utils.data.DataLoader()</code> to create batches of images.
2	Values between 0 & 1.	<code>torchvision.transforms.ToTensor()</code>
3	A mean of <code>[0.485, 0.456, 0.406]</code> (values across each colour channel).	<code>torchvision.transforms.Normalize(mean=...)</code> to adjust the mean of our images.
4	A standard deviation of <code>[0.229, 0.224, 0.225]</code> (values across each colour channel).	<code>torchvision.transforms.Normalize(std=...)</code> to adjust the standard deviation of our images.

**Note:** ^some pretrained models from `torchvision.models` in different sizes to `[3, 224, 224]`, for example, some might take them in `[3, 240, 240]`. For specific input image sizes, see the documentation.

**Question:** *Where did the mean and standard deviation values come from? Why do we need to do this?*

These were calculated from the data. Specifically, the ImageNet dataset by taking the means and standard deviations across a subset of images.

We also don't *need* to do this. Neural networks are usually quite capable of figuring out appropriate data distributions (they'll calculate where the mean and standard deviations need to be on their own) but setting them at the start can help our networks achieve better performance quicker.

Let's compose a series of `torchvision.transforms` to perform the above steps.

In [6]:







Wonderful!

Now we've got a **manually created series of transforms** ready to prepare our images, let's create training and testing DataLoaders.

We can create these using the `create_dataloaders` function from the `data_setup.py` script we created in [05. PyTorch Going Modular Part 2](#).

We'll set `batch_size=32` so our model see's mini-batches of 32 samples at a time.

And we can transform our images using the transform pipeline we created above by setting

```
transform=simple_transform.
```

**Note:** I've included this manual creation of transforms in this notebook because you may come across resources that use this style. It's also important to note that because these transforms are manually created, they're also infinitely customizable. So if you wanted to included data augmentation techniques in your transforms pipeline, you could.

In [7]:











Out[7]:

```
(<torch.utils.data.dataloader.DataLoader at 0x7fa9429a3a60>,
 <torch.utils.data.dataloader.DataLoader at 0x7fa9429a37c0>,
 ['pizza', 'steak', 'sushi'])
```

## 2.2 Creating a transform for `torchvision.models` (auto creation)

As previously stated, when using a pretrained model, it's important that **your custom data going into the model is prepared in the same way as the original training data that went into the model.**

Above we saw how to manually create a transform for a pretrained model.

But as of `torchvision v0.13+`, an automatic transform creation feature has been added.

When you setup a model from `torchvision.models` and select the pretrained model weights you'd like to use, for example, say we'd like to use:

```
weights = torchvision.models.EfficientNet_B0_Weights.DEFAULT
```

Where,

- `EfficientNet_B0_Weights` is the model architecture weights we'd like to use (there are many different model architecture options in `torchvision.models`).
- `DEFAULT` means the *best available* weights (the best performance in ImageNet).
  - **Note:** Depending on the model architecture you choose, you may also see other options such as `IMAGENET_V1` and `IMAGENET_V2` where generally the higher version number the better. Though if you want the best available, `DEFAULT` is the easiest option. See the `torchvision.models` documentation for more.

Let's try it out.

In [8]:

Out[8]:

```
EfficientNet_B0_Weights.IMAGENET1K_V1
```

And now to access the transforms associated with our `weights`, we can use the `transforms()` method.

This is essentially saying "get the data transforms that were used to train the `EfficientNet_B0_Weights` on ImageNet".

In [9]:

Out[9]:

```
ImageClassification(  
    crop_size=[224]  
    resize_size=[256]  
    mean=[0.485, 0.456, 0.406]  
    std=[0.229, 0.224, 0.225]  
    interpolation=InterpolationMode.BICUBIC  
)
```

Notice how `auto_transforms` is very similar to `manual_transforms`, the only difference is that `auto_transforms` came with the model architecture we chose, where as we had to create `manual_transforms` by hand.

The benefit of automatically creating a transform through `weights.transforms()` is that you ensure you're using the same data transformation as the pretrained model used when it was trained.

However, the tradeoff of using automatically created transforms is a lack of customization.

We can use `auto_transforms` to create DataLoaders with `create_dataloaders()` just as before.

In [10]:











Out[10]:

```
(<torch.utils.data.dataloader.DataLoader at 0x7fa942951460>,
<torch.utils.data.dataloader.DataLoader at 0x7fa942951550>,
['pizza', 'steak', 'sushi'])
```

### 3. Getting a pretrained model

Alright, here comes the fun part!

Over the past few notebooks we've been building PyTorch neural networks from scratch.

And while that's a good skill to have, our models haven't been performing as well as we'd like.

That's where **transfer learning** comes in.

The whole idea of transfer learning is to **take an already well-performing model on a problem-space similar to yours and then customising it to your use case**.

Since we're working on a computer vision problem (image classification with FoodVision Mini), we can find pretrained classification models in `torchvision.models`.

Exploring the documentation, you'll find plenty of common computer vision architecture backbones such as:

**Architecuture backbone**

**Code**

[ResNet's](#)

```
torchvision.models.resnet18() , torchvision.models.resnet50() ...
```

**VGG** (similar to what we used for TinyVGG)`torchvision.models.vgg16()`**EfficientNet's**`torchvision.models.efficientnet_b0(), torchvision.models.efficientnet_b1() ...`**VisionTransformer** (ViT's)`torchvision.models.vit_b_16(), torchvision.models.vit_b_32() ...`**ConvNeXt**`torchvision.models.convnext_tiny(), torchvision.models.convnext_small() ...`More available in `torchvision.models``torchvision.models...`

### 3.1 Which pretrained model should you use?

It depends on your problem/the device you're working with.

Generally, the higher number in the model name (e.g. `efficientnet_b0()` → `efficientnet_b1()` → `efficientnet_b7()`) means *better performance* but a *larger* model.

You might think better performance is *always better*, right?

That's true but **some better performing models are too big for some devices**.

For example, say you'd like to run your model on a mobile-device, you'll have to take into account the limited compute resources on the device, thus you'd be looking for a smaller model.

But if you've got unlimited compute power, as *The Bitter Lesson* states, you'd likely take the biggest, most compute hungry model you can.

Understanding this **performance vs. speed vs. size tradeoff** will come with time and practice.

For me, I've found a nice balance in the `efficientnet_bx` models.

As of May 2022, **Nutrify** (the machine learning powered app I'm working on) is powered by an `efficientnet_b0`.

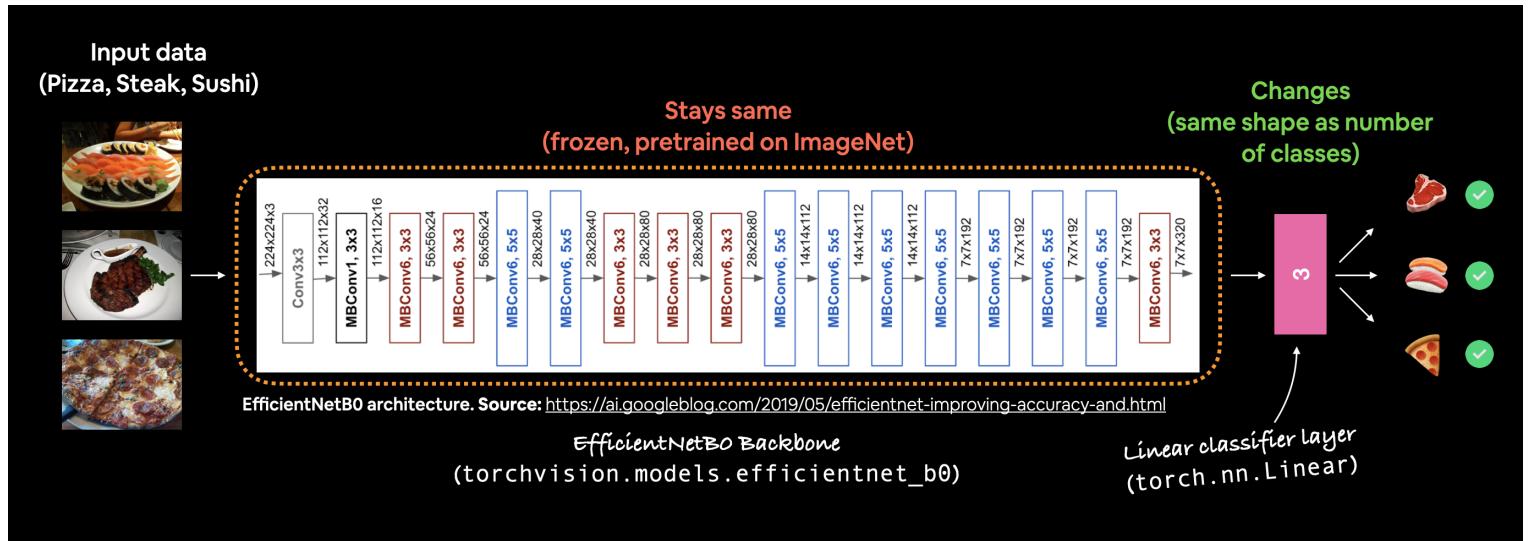
**Comma.ai** (a company that makes open source self-driving car software) **uses an `efficientnet_b2`** to learn a representation of the road.

**Note:** Even though we're using `efficientnet_bx`, it's important not to get too attached to any one architecture, as they are always changing as new research gets released. Best to experiment, experiment, experiment and see what works for your problem.

### 3.2 Setting up a pretrained model

The pretrained model we're going to be using is `torchvision.models.efficientnet_b0()`.

The architecture is from the paper *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*.



*Example of what we're going to create, a pretrained EfficientNet\_B0 model from torchvision.models with the output layer adjusted for our use case of classifying pizza, steak and sushi images.*

We can setup the `EfficientNet_B0` pretrained ImageNet weights using the same code as we used to create the transforms.

```
weights = torchvision.models.EfficientNet_B0_Weights.DEFAULT # .DEFAULT = best available weights  
for ImageNet
```

This means the model has already been trained on millions of images and has a good base representation of image data.

The PyTorch version of this pretrained model is capable of achieving ~77.7% accuracy across ImageNet's 1000 classes.

We'll also send it to the target device.

In [11]:





**Note:** In previous versions of `torchvision`, you'd create a pretrained model with code like:

```
model = torchvision.models.efficientnet_b0(pretrained=True).to(device)
```

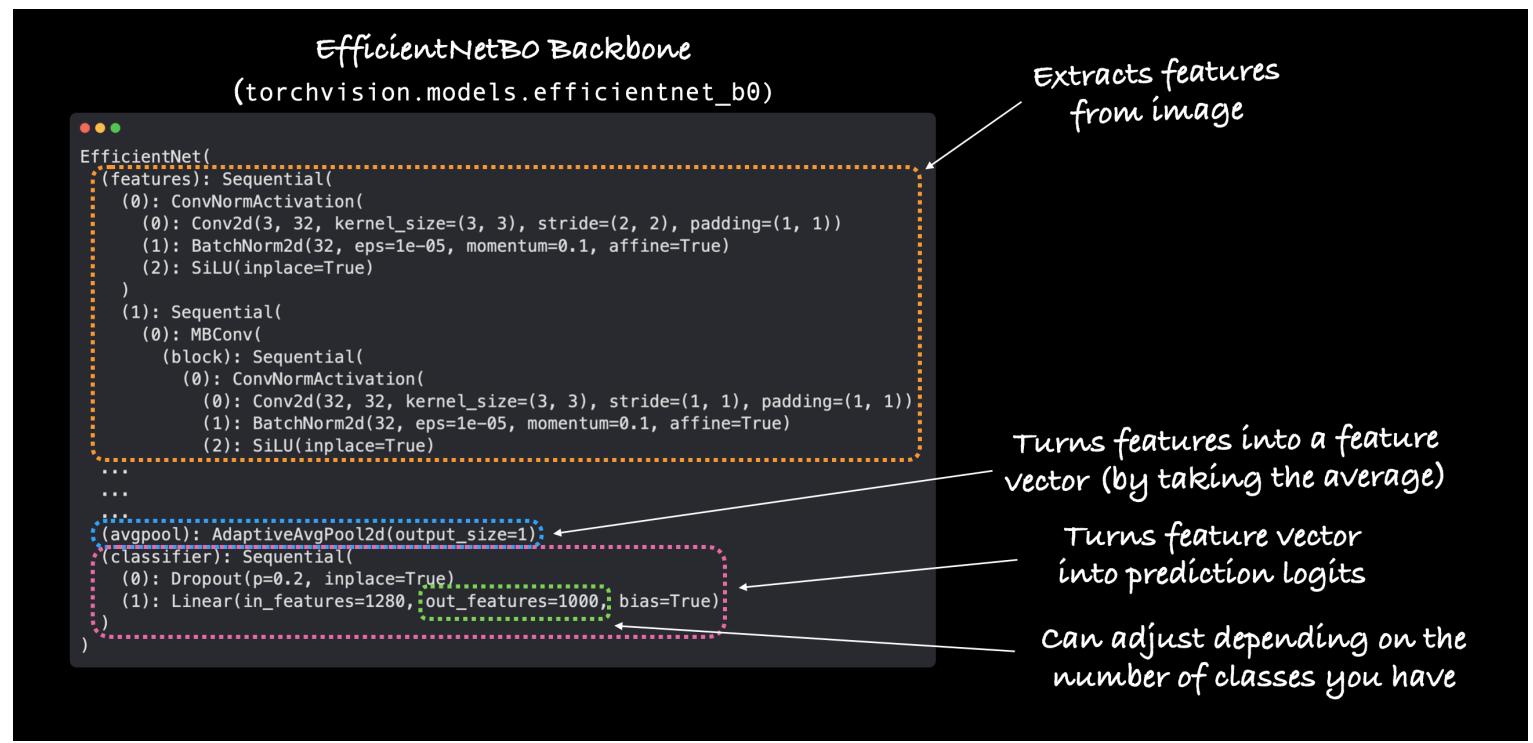
However, running this using `torchvision v0.13+` will result in errors such as the following:

```
UserWarning: The parameter 'pretrained' is deprecated since 0.13 and will be removed in 0.15, please use  
'weights' instead.
```

And...

```
UserWarning: Arguments other than a weight enum or None for weights are deprecated since 0.13 and will be  
removed in 0.15. The current behavior is equivalent to passing weights=EfficientNet_B0_Weights.IMAGENET1K_V1.  
You can also use weights=EfficientNet_B0_Weights.DEFAULT to get the most up-to-date weights.
```

If we print the model, we get something similar to the following:



Lots and lots and lots of layers.

This is one of the benefits of transfer learning, taking an existing model, that's been crafted by some of the best engineers in the world and applying to your own problem.

Our `efficientnet_b0` comes in three main parts:

1. `features` - A collection of convolutional layers and other various activation layers to learn a base representation of vision data (this base representation/collection of layers is often referred to as **features** or **feature extractor**, "the base layers of the model learn the different **features** of images").
2. `avgpool` - Takes the average of the output of the `features` layer(s) and turns it into a **feature vector**.
3. `classifier` - Turns the **feature vector** into a vector with the same dimensionality as the number of required output classes (since `efficientnet_b0` is pretrained on ImageNet and because ImageNet has 1000 classes, `out_features=1000` is the default).

### 3.3 Getting a summary of our model with `torchinfo.summary()`

To learn more about our model, let's use `torchinfo`'s `summary()` method.

To do so, we'll pass in:

- `model` - the model we'd like to get a summary of.
- `input_size` - the shape of the data we'd like to pass to our model, for the case of `efficientnet_b0`, the input size

is `(batch_size, 3, 224, 224)`, though other variants of `efficientnet_bx` have different input sizes.

- **Note:** Many modern models can handle input images of varying sizes thanks to `torch.nn.AdaptiveAvgPool2d()`, this layer adaptively adjusts the `output_size` of a given input as required. You can try this out by passing different size input images to `summary()` or your models.
- `col_names` - the various information columns we'd like to see about our model.
- `col_width` - how wide the columns should be for the summary.
- `row_settings` - what features to show in a row.

In [12]:



Out[12]:

Layer (type (var_name))	Input Shape	Output Shape	Param #	Trainable
EfficientNet (EfficientNet)	[32, 3, 224, 224]	[32, 1000]	--	True
└ Sequential (features)	[32, 3, 224, 224]	[32, 1280, 7, 7]	--	True
└ Conv2dNormActivation (0)	[32, 3, 224, 224]	[32, 32, 112, 112]	--	True
└ Conv2d (0)	[32, 3, 224, 224]	[32, 32, 112, 112]	864	True
└ BatchNorm2d (1)	[32, 32, 112, 112]	[32, 32, 112, 112]	64	True
└ SiLU (2)	[32, 32, 112, 112]	[32, 32, 112, 112]	--	--
└ Sequential (1)	[32, 32, 112, 112]	[32, 16, 112, 112]	--	True
└ MBCConv (0)	[32, 32, 112, 112]	[32, 16, 112, 112]	1,448	True
└ Sequential (2)	[32, 16, 112, 112]	[32, 24, 56, 56]	--	True
└ MBCConv (0)	[32, 16, 112, 112]	[32, 24, 56, 56]	6,004	True
└ MBCConv (1)	[32, 24, 56, 56]	[32, 24, 56, 56]	10,710	True
└ Sequential (3)	[32, 24, 56, 56]	[32, 40, 28, 28]	--	True
└ MBCConv (0)	[32, 24, 56, 56]	[32, 40, 28, 28]	15,350	True
└ MBCConv (1)	[32, 40, 28, 28]	[32, 40, 28, 28]	31,290	True
└ Sequential (4)	[32, 40, 28, 28]	[32, 80, 14, 14]	--	True
└ MBCConv (0)	[32, 40, 28, 28]	[32, 80, 14, 14]	37,130	True
└ MBCConv (1)	[32, 80, 14, 14]	[32, 80, 14, 14]	102,900	True
└ MBCConv (2)	[32, 80, 14, 14]	[32, 80, 14, 14]	102,900	True
└ Sequential (5)	[32, 80, 14, 14]	[32, 112, 14, 14]	--	True
└ MBCConv (0)	[32, 80, 14, 14]	[32, 112, 14, 14]	126,004	True
└ MBCConv (1)	[32, 112, 14, 14]	[32, 112, 14, 14]	208,572	True
└ MBCConv (2)	[32, 112, 14, 14]	[32, 112, 14, 14]	208,572	True
└ Sequential (6)	[32, 112, 14, 14]	[32, 192, 7, 7]	--	True
└ MBCConv (0)	[32, 112, 14, 14]	[32, 192, 7, 7]	262,492	True
└ MBCConv (1)	[32, 192, 7, 7]	[32, 192, 7, 7]	587,952	True

	└─MBCConv (2)	[32, 192, 7, 7]	[32, 192, 7, 7]	587,952	True
	└─MBCConv (3)	[32, 192, 7, 7]	[32, 192, 7, 7]	587,952	True
└─Sequential (7)		[32, 192, 7, 7]	[32, 320, 7, 7]	--	True
└─MBCConv (0)		[32, 192, 7, 7]	[32, 320, 7, 7]	717,232	True
└─Conv2dNormActivation (8)		[32, 320, 7, 7]	[32, 1280, 7, 7]	--	True
└─Conv2d (0)		[32, 320, 7, 7]	[32, 1280, 7, 7]	409,600	True
└─BatchNorm2d (1)		[32, 1280, 7, 7]	[32, 1280, 7, 7]	2,560	True
└─SiLU (2)		[32, 1280, 7, 7]	[32, 1280, 7, 7]	--	--
└─AdaptiveAvgPool2d (avgpool)		[32, 1280, 7, 7]	[32, 1280, 1, 1]	--	--
└─Sequential (classifier)		[32, 1280]	[32, 1000]	--	True
└─Dropout (0)		[32, 1280]	[32, 1280]	--	--
└─Linear (1)		[32, 1280]	[32, 1000]	1,281,000	True

Total params: 5,288,548

Trainable params: 5,288,548

Non-trainable params: 0

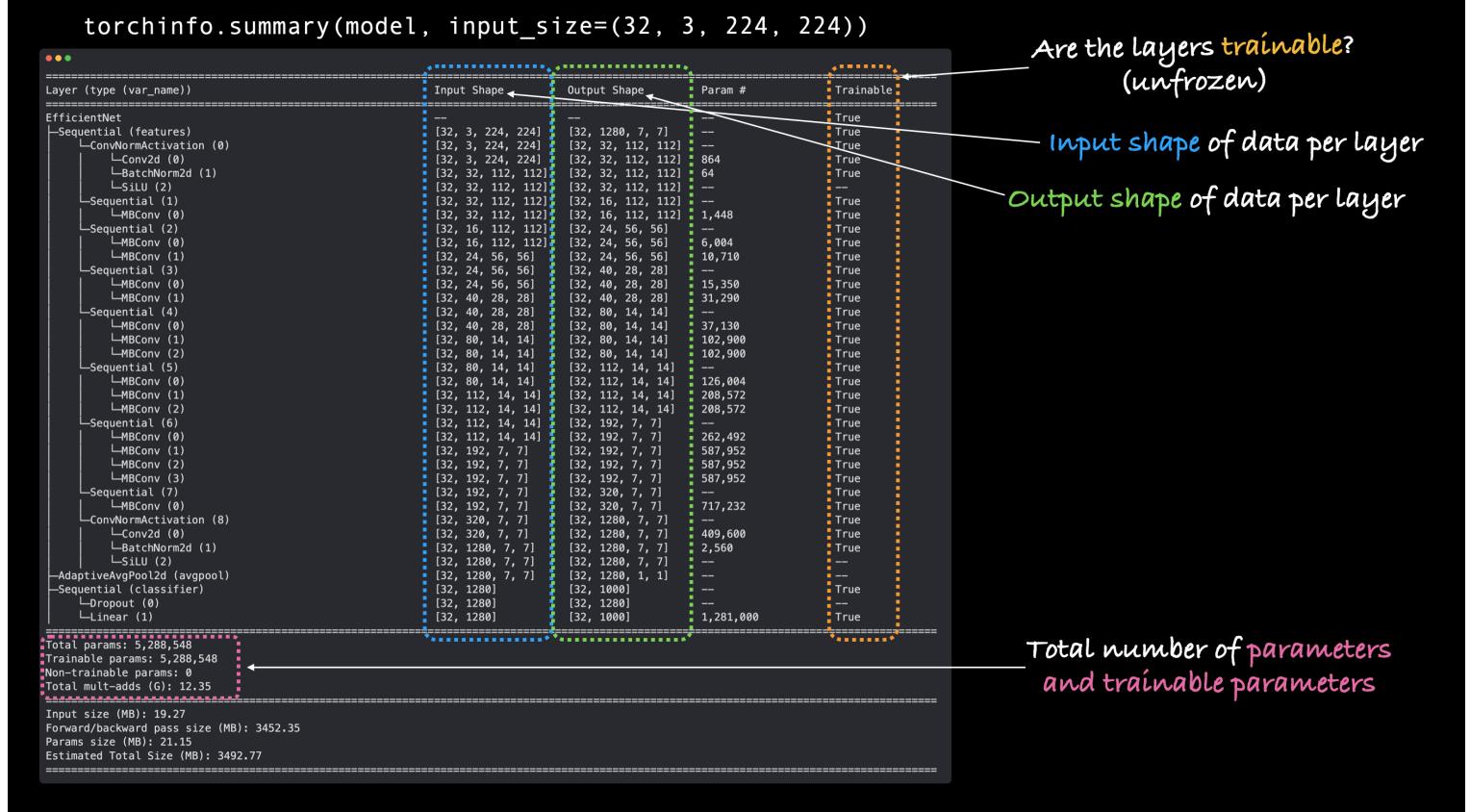
Total mult-adds (G): 12.35

Input size (MB): 19.27

Forward/backward pass size (MB): 3452.35

Params size (MB): 21.15

Estimated Total Size (MB): 3492.77



Woah!

Now that's a big model!

From the output of the summary, we can see all of the various input and output shape changes as our image data goes through the model.

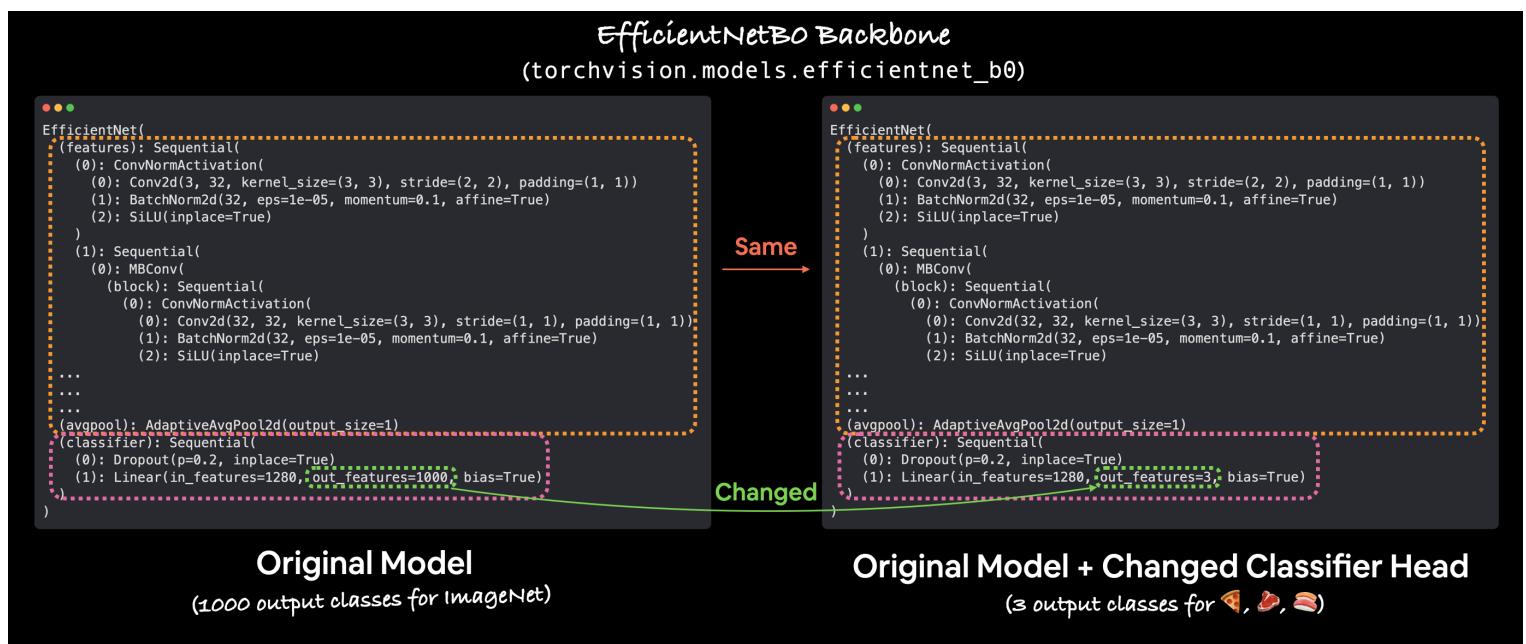
And there are a whole bunch more total parameters (pretrained weights) to recognize different patterns in our data.

For reference, our model from previous sections, **TinyVGG had 8,083 parameters vs. 5,288,548 parameters for efficientnet\_b0, an increase of ~654x!**

What do you think, will this mean better performance?

### 3.4 Freezing the base model and changing the output layer to suit our needs

The process of transfer learning usually goes: freeze some base layers of a pretrained model (typically the `features` section) and then adjust the output layers (also called head/classifier layers) to suit your needs.



You can customise the outputs of a pretrained model by changing the output layer(s) to suit your problem. The original `torchvision.models.efficientnet_b0()` comes with `out_features=1000` because there are 1000 classes in ImageNet, the dataset it was trained on. However, for our problem, classifying images of pizza, steak and sushi we only need `out_features=3`.

Let's freeze all of the layers/parameters in the `features` section of our `efficientnet_b0` model.

**Note:** To *freeze* layers means to keep them how they are during training. For example, if your model has pretrained layers, to *freeze* them would be to say, "don't change any of the patterns in these layers during training, keep them how they are." In essence, we'd like to keep the pretrained weights/patterns our model has learned from ImageNet

as a backbone and then only change the output layers.

We can freeze all of the layers/parameters in the `features` section by setting the attribute `requires_grad=False`.

For parameters with `requires_grad=False`, PyTorch doesn't track gradient updates and in turn, these parameters won't be changed by our optimizer during training.

In essence, a parameter with `requires_grad=False` is "untrainable" or "frozen" in place.

In [13]:

Feature extractor layers frozen!

Let's now adjust the output layer or the `classifier` portion of our pretrained model to our needs.

Right now our pretrained model has `out_features=1000` because there are 1000 classes in ImageNet.

However, we don't have 1000 classes, we only have three, pizza, steak and sushi.

We can change the `classifier` portion of our model by creating a new series of layers.

The current `classifier` consists of:

```
(classifier): Sequential(
    (0): Dropout(p=0.2, inplace=True)
    (1): Linear(in_features=1280, out_features=1000, bias=True)
```

We'll keep the `Dropout` layer the same using `torch.nn.Dropout(p=0.2, inplace=True)`.

**Note:** [Dropout layers](#) randomly remove connections between two neural network layers with a probability of `p`. For example, if `p=0.2`, 20% of connections between neural network layers will be removed at random each pass. This practice is meant to help regularize (prevent overfitting) a model by making sure the connections that remain learn features to compensate for the removal of the other connections (hopefully these remaining features are *more general*).

And we'll keep `in_features=1280` for our `Linear` output layer but we'll change the `out_features` value to the length of our `class_names` (`len(['pizza', 'steak', 'sushi']) = 3`).

Our new `classifier` layer should be on the same device as our `model`.

In [14]:





Nice!

Output layer updated, let's get another summary of our model and see what's changed.

In [15]:





Out[15]:

Layer (type (var_name))	Input Shape	Output Shape	Param #	Trainable
EfficientNet (EfficientNet)	[32, 3, 224, 224]	[32, 3]	--	Partial
└ Sequential (features)	[32, 3, 224, 224]	[32, 1280, 7, 7]	--	False
└ Conv2dNormActivation (0)	[32, 3, 224, 224]	[32, 32, 112, 112]	--	False
└ Conv2d (0)	[32, 3, 224, 224]	[32, 32, 112, 112]	(864)	False
└ BatchNorm2d (1)	[32, 32, 112, 112]	[32, 32, 112, 112]	(64)	False
└ SiLU (2)	[32, 32, 112, 112]	[32, 32, 112, 112]	--	--
└ Sequential (1)	[32, 32, 112, 112]	[32, 16, 112, 112]	--	False
└ MBCConv (0)	[32, 32, 112, 112]	[32, 16, 112, 112]	(1,448)	False
└ Sequential (2)	[32, 16, 112, 112]	[32, 24, 56, 56]	--	False
└ MBCConv (0)	[32, 16, 112, 112]	[32, 24, 56, 56]	(6,004)	False
└ MBCConv (1)	[32, 24, 56, 56]	[32, 24, 56, 56]	(10,710)	False
└ Sequential (3)	[32, 24, 56, 56]	[32, 40, 28, 28]	--	False
└ MBCConv (0)	[32, 24, 56, 56]	[32, 40, 28, 28]	(15,350)	False
└ MBCConv (1)	[32, 40, 28, 28]	[32, 40, 28, 28]	(31,290)	False
└ Sequential (4)	[32, 40, 28, 28]	[32, 80, 14, 14]	--	False
└ MBCConv (0)	[32, 40, 28, 28]	[32, 80, 14, 14]	(37,130)	False
└ MBCConv (1)	[32, 80, 14, 14]	[32, 80, 14, 14]	(102,900)	False
└ MBCConv (2)	[32, 80, 14, 14]	[32, 80, 14, 14]	(102,900)	False
└ Sequential (5)	[32, 80, 14, 14]	[32, 112, 14, 14]	--	False
└ MBCConv (0)	[32, 80, 14, 14]	[32, 112, 14, 14]	(126,004)	False
└ MBCConv (1)	[32, 112, 14, 14]	[32, 112, 14, 14]	(208,572)	False
└ MBCConv (2)	[32, 112, 14, 14]	[32, 112, 14, 14]	(208,572)	False
└ Sequential (6)	[32, 112, 14, 14]	[32, 192, 7, 7]	--	False
└ MBCConv (0)	[32, 112, 14, 14]	[32, 192, 7, 7]	(262,492)	False
└ MBCConv (1)	[32, 192, 7, 7]	[32, 192, 7, 7]	(587,952)	False
└ MBCConv (2)	[32, 192, 7, 7]	[32, 192, 7, 7]	(587,952)	False
└ MBCConv (3)	[32, 192, 7, 7]	[32, 192, 7, 7]	(587,952)	False

└ Sequential (7)	[32, 192, 7, 7]	[32, 320, 7, 7]	--	False
└ MBCConv (0)	[32, 192, 7, 7]	[32, 320, 7, 7]	(717,232)	False
└ Conv2dNormActivation (8)	[32, 320, 7, 7]	[32, 1280, 7, 7]	--	False
└ Conv2d (0)	[32, 320, 7, 7]	[32, 1280, 7, 7]	(409,600)	False
└ BatchNorm2d (1)	[32, 1280, 7, 7]	[32, 1280, 7, 7]	(2,560)	False
└ SiLU (2)	[32, 1280, 7, 7]	[32, 1280, 7, 7]	--	--
└ AdaptiveAvgPool2d (avgpool)	[32, 1280, 7, 7]	[32, 1280, 1, 1]	--	--
└ Sequential (classifier)	[32, 1280]	[32, 3]	--	True
└ Dropout (0)	[32, 1280]	[32, 1280]	--	--
└ Linear (1)	[32, 1280]	[32, 3]	3,843	True

Total params: 4,011,391

Trainable params: 3,843

Non-trainable params: 4,007,548

Total mult-adds (G): 12.31

Input size (MB): 19.27

Forward/backward pass size (MB): 3452.09

Params size (MB): 16.05

Estimated Total Size (MB): 3487.41

```
torchinfo.summary(model, input_size=(32, 3, 224, 224))

...  

Layer (type (var_name))           Input Shape        Output Shape       Param #      Trainable  

=====  

EfficientNet  

└ Sequential (features)  

  └ ConvNormActivation (0)  

    └ Conv2d (0)  

    └ BatchNorm2d (1)  

    └ SiLU (2)  

  └ Sequential (1)  

    └ MBCConv (0)  

  └ Sequential (2)  

    └ MBCConv (0)  

    └ MBCConv (1)  

  └ Sequential (3)  

    └ MBCConv (0)  

    └ MBCConv (1)  

  └ Sequential (4)  

    └ MBCConv (0)  

    └ MBCConv (1)  

    └ MBCConv (2)  

  └ Sequential (5)  

    └ MBCConv (0)  

    └ MBCConv (1)  

    └ MBCConv (2)  

  └ Sequential (6)  

    └ MBCConv (0)  

    └ MBCConv (1)  

    └ MBCConv (2)  

    └ MBCConv (3)  

  └ Sequential (7)  

    └ MBCConv (0)  

  └ ConvNormActivation (8)  

    └ Conv2d (0)  

    └ BatchNorm2d (1)  

    └ SiLU (2)  

└ AdaptiveAvgPool2d (avgpool)  

  └ Sequential (classifier)  

    └ Dropout (0)  

    └ Linear (1)  

Total params: 4,011,391  

Trainable params: 3,843  

Non-trainable params: 4,007,548  

Total mult-adds (G): 12.31  

Input size (MB): 19.27  

Forward/backward pass size (MB): 3452.09  

Params size (MB): 16.05  

Estimated Total Size (MB): 3487.41
```

Many layers untrainable (frozen)

Only last layers are trainable

Final layer output (same as number of classes)

Less trainable parameters because many layers are frozen

Ho, ho! There's a fair few changes here!

Let's go through them:

- **Trainable column** - You'll see that many of the base layers (the ones in the `features` portion) have their Trainable value as `False`. This is because we set their attribute `requires_grad=False`. Unless we change this, these layers won't be updated during future training.
- **Output shape of `classifier`** - The `classifier` portion of the model now has an Output Shape value of `[32, 3]` instead of `[32, 1000]`. Its Trainable value is also `True`. This means its parameters will be updated during training. In essence, we're using the `features` portion to feed our `classifier` portion a base representation of an image and then our `classifier` layer is going to learn how to base representation aligns with our problem.
- **Less trainable parameters** - Previously there was 5,288,548 trainable parameters. But since we froze many of the layers of the model and only left the `classifier` as trainable, there's now only 3,843 trainable parameters (even less than our TinyVGG model). Though there's also 4,007,548 non-trainable parameters, these will create a base representation of our input images to feed into our `classifier` layer.

**Note:** The more trainable parameters a model has, the more compute power/longer it takes to train. Freezing the base layers of our model and leaving it with less trainable parameters means our model should train quite quickly. This is one huge benefit of transfer learning, taking the already learned parameters of a model trained on a problem similar to yours and only tweaking the outputs slightly to suit your problem.

## 4. Train model

Now we've got a pre-trained model that's semi-frozen and has a customised `classifier`, how about we see transfer learning in action?

To begin training, let's create a loss function and an optimizer.

Because we're still working with multi-class classification, we'll use `nn.CrossEntropyLoss()` for the loss function.

And we'll stick with `torch.optim.Adam()` as our optimizer with `lr=0.001`.

In [16]:

Wonderful!

To train our model, we can use `train()` function we defined in the [05. PyTorch Going Modular section 04](#).

The `train()` function is in the `engine.py` script inside the `going_modular` directory.

Let's see how long it takes to train our model for 5 epochs.

**Note:** We're only going to be training the parameters `classifier` here as all of the other parameters in our model have been frozen.

In [17]:









```
0% |          | 0/5 [00:00<?, ?it/s]
Epoch: 1 | train_loss: 1.0924 | train_acc: 0.3984 | test_loss: 0.9133 | test_acc: 0.5398
Epoch: 2 | train_loss: 0.8717 | train_acc: 0.7773 | test_loss: 0.7912 | test_acc: 0.8153
Epoch: 3 | train_loss: 0.7648 | train_acc: 0.7930 | test_loss: 0.7463 | test_acc: 0.8561
Epoch: 4 | train_loss: 0.7108 | train_acc: 0.7539 | test_loss: 0.6372 | test_acc: 0.8655
Epoch: 5 | train_loss: 0.6254 | train_acc: 0.7852 | test_loss: 0.6260 | test_acc: 0.8561
[INFO] Total training time: 8.977 seconds
```

Wow!

Our model trained quite fast (~5 seconds on my local machine with a [NVIDIA TITAN RTX GPU](#)/about 15 seconds on Google Colab with a [NVIDIA P100 GPU](#)).

And it looks like it smashed our previous model results out of the park!

With an `efficientnet_b0` backbone, our model achieves almost 85%+ accuracy on the test dataset, almost *double* what we were able to achieve with TinyVGG.

Not bad for a model we downloaded with a few lines of code.

## 5. Evaluate model by plotting loss curves

Our model looks like it's performing pretty well.

Let's plot it's loss curves to see what the training looks like over time.

We can plot the loss curves using the function `plot_loss_curves()` we created in [04. PyTorch Custom Datasets section 7.8](#).

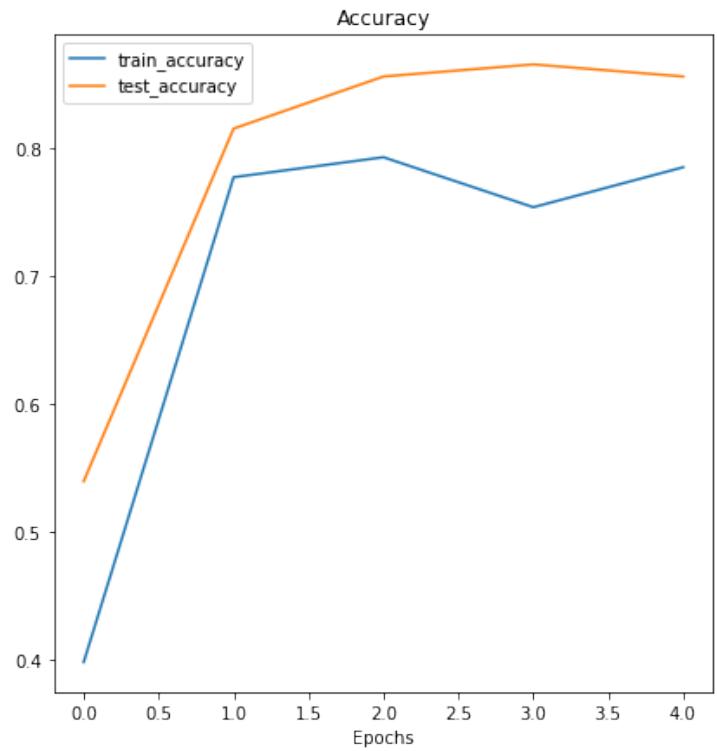
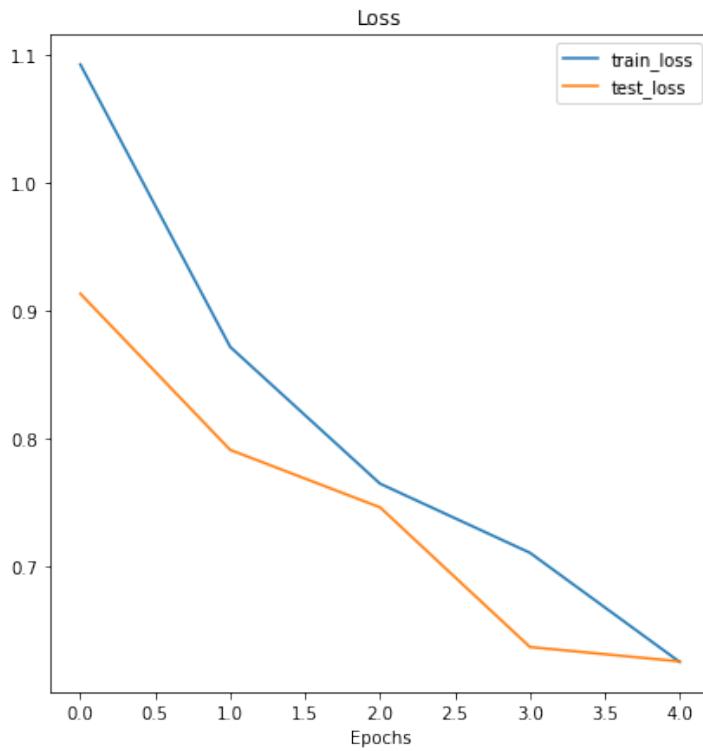
The function is stored in the `helper_functions.py` script so we'll try to import it and download the script if we don't have it.

In [18]:









Those are some excellent looking loss curves!

It looks like the loss for both datasets (train and test) is heading in the right direction.

The same with the accuracy values, trending upwards.

That goes to show the power of **transfer learning**. Using a pretrained model often leads to pretty good results with a small amount of data in less time.

I wonder what would happen if you tried to train the model for longer? Or if we added more data?

**Question:** Looking at the loss curves, does our model look like it's overfitting or underfitting? Or perhaps neither?

Hint: Check out notebook [04. PyTorch Custom Datasets part 8. What should an ideal loss curve look like?](#) for ideas.

## 6. Make predictions on images from the test set

It looks like our model performs well quantitatively but how about qualitatively?

Let's find out by making some predictions with our model on images from the test set (these aren't seen during training) and plotting them.

*Visualize, visualize, visualize!*

One thing we'll have to remember is that for our model to make predictions on an image, the image has to be in *same*

format as the images our model was trained on.

This means we'll need to make sure our images have:

- **Same shape** - If our images are different shapes to what our model was trained on, we'll get shape errors.
- **Same datatype** - If our images are a different datatype (e.g. `torch.int8` vs. `torch.float32`) we'll get datatype errors.
- **Same device** - If our images are on a different device to our model, we'll get device errors.
- **Same transformations** - If our model is trained on images that have been transformed in certain way (e.g. normalized with a specific mean and standard deviation) and we try and make predictions on images transformed in a different way, these predictions may be off.

**Note:** These requirements go for all kinds of data if you're trying to make predictions with a trained model. Data you'd like to predict on should be in the same format as your model was trained on.

To do all of this, we'll create a function `pred_and_plot_image()` to:

1. Take in a trained model, a list of class names, a filepath to a target image, an image size, a transform and a target device.
2. Open an image with `PIL.Image.open()`.
3. Create a transform for the image (this will default to the `manual_transforms` we created above or it could use a transform generated from `weights.transforms()`).
4. Make sure the model is on the target device.
5. Turn on model eval mode with `model.eval()` (this turns off layers like `nn.Dropout()`, so they aren't used for inference) and the inference mode context manager.
6. Transform the target image with the transform made in step 3 and add an extra batch dimension with `torch.unsqueeze(dim=0)` so our input image has shape `[batch_size, color_channels, height, width]`.
7. Make a prediction on the image by passing it to the model ensuring it's on the target device.
8. Convert the model's output logits to prediction probabilities with `torch.softmax()`.
9. Convert model's prediction probabilities to prediction labels with `torch.argmax()`.
10. Plot the image with `matplotlib` and set the title to the prediction label from step 9 and prediction probability from step 8.

**Note:** This is a similar function to [04. PyTorch Custom Datasets section 11.3's `pred\_and\_plot\_image\(\)`](#) with a few tweaked steps.

In [19]:



























What a good looking function!

Let's test it out by making predictions on a few random images from the test set.

We can get a list of all the test image paths using `list(Path(test_dir).glob("*//*.jpg"))`, the stars in the `glob()` method say "any file matching this pattern", in other words, any file ending in `.jpg` (all of our images).

And then we can randomly sample a number of these using Python's `random.sample(population, k)` where `population` is the sequence to sample and `k` is the number of samples to retrieve.

In [20]:











Pred: sushi | Prob: 0.507



Pred: sushi | Prob: 0.427



Pred: pizza | Prob: 0.655



Woohoo!

Those predictions look far better than the ones our TinyVGG model was previously making.

## 6.1 Making predictions on a custom image

It looks like our model does well qualitatively on data from the test set.

But how about on our own custom image?

That's where the real fun of machine learning is!

Predicting on your own custom data, outside of any training or test set.

To test our model on a custom image, let's import the old faithful `pizza-dad.jpeg` image (an image of my dad eating pizza).

We'll then pass it to the `pred_and_plot_image()` function we created above and see what happens.

In [21]:







data/04-pizza-dad.jpeg already exists, skipping download.

Pred: pizza | Prob: 0.499



Two thumbs up!

Looks like our model got it right again!

But this time the prediction probability is higher than the one from TinyVGG ( $0.373$ ) in [04. PyTorch Custom Datasets section 11.3](#).

This indicates our `efficientnet_b0` model is *more* confident in its prediction where as our TinyVGG model was *par* with just guessing.

## Main takeaways

- **Transfer learning** often allows you to get good results with a relatively small amount of custom data.

- Knowing the power of transfer learning, it's a good idea to ask at the start of every problem, "does an existing well-performing model exist for my problem?"
- When using a pretrained model, it's important that your custom data be formatted/preprocessed in the same way that the original model was trained on, otherwise you may get degraded performance.
- The same goes for predicting on custom data, ensure your custom data is in the same format as the data your model was trained on.
- There are [several different places to find pretrained models](#) from the PyTorch domain libraries, HuggingFace Hub and libraries such as `timm` (PyTorch Image Models).

## Exercises

All of the exercises are focused on practicing the code above.

You should be able to complete them by referencing each section or by following the resource(s) linked.

All exercises should be completed using [device-agnostic code](#).

### Resources:

- [Exercise template notebook for 06](#)
  - [Example solutions notebook for 06](#) (try the exercises *before* looking at this)
    - See a live [video walkthrough of the solutions on YouTube](#) (errors and all)
1. Make predictions on the entire test dataset and plot a confusion matrix for the results of our model compared to the truth labels. Check out [03. PyTorch Computer Vision section 10](#) for ideas.
  2. Get the "most wrong" of the predictions on the test dataset and plot the 5 "most wrong" images. You can do this by:
    - Predicting across all of the test dataset, storing the labels and predicted probabilities.
    - Sort the predictions by *wrong prediction* and then *descending predicted probabilities*, this will give you the wrong predictions with the *highest* prediction probabilities, in other words, the "most wrong".
    - Plot the top 5 "most wrong" images, why do you think the model got these wrong?
  3. Predict on your own image of pizza/steak/sushi - how does the model go? What happens if you predict on an image that isn't pizza/steak/sushi?
  4. Train the model from section 4 above for longer (10 epochs should do), what happens to the performance?
  5. Train the model from section 4 above with more data, say 20% of the images from Food101 of Pizza, Steak and

Sushi images.

- You can find the [20% Pizza, Steak, Sushi dataset](#) on the course GitHub. It was created with the notebook `extras/04_custom_data_creation.ipynb`.
6. Try a different model from `torchvision.models` on the Pizza, Steak, Sushi data, how does this model perform?
- You'll have to change the size of the classifier layer to suit our problem.
  - You may want to try an EfficientNet with a higher number than our B0, perhaps `torchvision.models.efficientnet_b2()`?

## Extra-curriculum

- Look up what "model fine-tuning" is and spend 30-minutes researching different methods to perform it with PyTorch. How would we change our code to fine-tine? Tip: fine-tuning usually works best if you have *lots* of custom data, whereas, feature extraction is typically better if you have less custom data.
- Check out the new/upcoming [PyTorch multi-weights API](#) (still in beta at time of writing, May 2022), it's a new way to perform transfer learning in PyTorch. What changes to our code would need to be made to use the new API?
- Try to create your own classifier on two classes of images, for example, you could collect 10 photos of your dog and your friends dog and train a model to classify the two dogs. This would be a good way to practice creating a dataset as well as building a model on that dataset.

Previous

[05. PyTorch Going Modular](#)

Next

[07. PyTorch Experiment Tracking](#)



[View Source Code](#) | [View Slides](#)

## 07. PyTorch Experiment Tracking

**Note:** This notebook uses `torchvision`'s new [multi-weight support API \(available in `torchvision v0.13+`\)](#).

We've trained a fair few models now on the journey to making FoodVision Mini (an image classification model to classify images of pizza, steak or sushi).

And so far we've keep track of them via Python dictionaries.

Or just comparing them by the metric print outs during training.

What if you wanted to run a dozen (or more) different models at once?

Surely there's a better way...

There is.

### Experiment tracking.

And since experiment tracking is so important and integral to machine learning, you can consider this notebook your first milestone project.

So welcome to Milestone Project 1: FoodVision Mini Experiment Tracking.

We're going to answer the question: **how do I track my machine learning experiments?**

### What is experiment tracking?

Machine learning and deep learning are very experimental.

You have to put on your artist's beret/chef's hat to cook up lots of different models.

And you have to put on your scientist's coat to track the results of various combinations of data, model architectures and training regimes.

That's where **experiment tracking** comes in.

If you're running lots of different experiments, **experiment tracking helps you figure out what works and what doesn't.**

## Why track experiments?

If you're only running a handful of models (like we've done so far), it might be okay just to track their results in print outs and a few dictionaries.

However, as the number of experiments you run starts to increase, this naive way of tracking could get out of hand.

So if you're following the machine learning practitioner's motto of *experiment, experiment, experiment!*, you'll want a way to track them.

### Experiment number/name

```
0 results
1 model_1_results
2 model_2_results
3 model_1_results_v2
4 model_2_results_v2
5 model_2_results_v3
6 model_2_results_v3
7 2022-07-07-model_3_results_v1
8 2022-07-07-model_3_results_v1_experiment_5
9 2022-07-07-model_3_results_v2_big_model_25_epochs
10 2022-07-07-model_4_results_v1_big_model_25_epochs_20_percent_data
11 2022-07-07-model_4_results_v1_big_model_25_epochs_20_percent_data_new_model
12 2022-07-07-model_4_results_v2_big_model_feature_extractor_30_epochs_20_percent_data_no_dro
13 2022-07-07-model_5_results_v1_bigger_model_feature_extractor_50_epochs_50_percent_data_no_dro
```

*After building a few models and tracking their results, you'll start to notice how quickly it can get out of hand.*

## Different ways to track machine learning experiments

There are as many different ways to track machine learning experiments as there are experiments to run.

This table covers a few.

Method	Setup	Pros	Cons	Cost
Python dictionaries, CSV files, print outs	None	Easy to setup, runs in pure Python	Hard to keep track of large numbers of experiments	Free
TensorBoard	Minimal, install <a href="#">tensorboard</a>	Extensions built into PyTorch, widely recognized and used, easily scales.	User-experience not as nice as other options.	Free
Weights & Biases Experiment Tracking	Minimal, install <a href="#">wandb</a> , make an account	Incredible user experience, makes experiments public, tracks almost anything.	Requires external resource outside of PyTorch.	Free for personal use
MLFlow	Minimal, install <a href="#">mlflow</a> and starting tracking	Fully open-source MLOps lifecycle management, many integrations.	Little bit harder to setup a remote tracking server than other services.	Free



Various places and techniques you can use to track your machine learning experiments. **Note:** There are various other options similar to Weights & Biases and open-source options similar to MLflow but I've left them out for brevity. You can find more by searching "machine learning experiment tracking".

## What we're going to cover

We're going to be running several different modelling experiments with various levels of data, model size and training time to try and improve on FoodVision Mini.

And due to its tight integration with PyTorch and widespread use, this notebook focuses on using TensorBoard to track our experiments.

However, the principles we're going to cover are similar across all of the other tools for experiment tracking.

### Topic

### Contents

#### 0. Getting setup

We've written a fair bit of useful code over the past few sections, let's download it and make sure we can use it again.

<b>1. Get data</b>	Let's get the pizza, steak and sushi image classification dataset we've been using to try and improve our FoodVision Mini model's results.
<b>2. Create Datasets and DataLoaders</b>	We'll use the <code>data_setup.py</code> script we wrote in chapter 05. PyTorch Going Modular to setup our DataLoaders.
<b>3. Get and customise a pretrained model</b>	Just like the last section, 06. PyTorch Transfer Learning we'll download a pretrained model from <code>torchvision.models</code> and customise it to our own problem.
<b>4. Train model and track results</b>	Let's see what it's like to train and track the training results of a single model using TensorBoard.
<b>5. View our model's results in TensorBoard</b>	Previously we visualized our model's loss curves with a helper function, now let's see what they look like in TensorBoard.
<b>6. Creating a helper function to track experiments</b>	If we're going to be adhering to the machine learner practitioner's motto of <i>experiment, experiment, experiment!</i> , we best create a function that will help us save our modelling experiment results.
<b>7. Setting up a series of modelling experiments</b>	Instead of running experiments one by one, how about we write some code to run several experiments at once, with different models, different amounts of data and different training times.
<b>8. View modelling experiments in TensorBoard</b>	By this stage we'll have run eight modelling experiments in one go, a fair bit to keep track of, let's what their results look like in TensorBoard.
<b>9. Load in the best model and make predictions with it</b>	The point of experiment tracking is to figure out which model performs the best, let's load in the best performing model and make some predictions with it to <i>visualize, visualize, visualize!</i> .

## Where can you get help?

All of the materials for this course [are available on GitHub](#).

If you run into trouble, you can ask a question on the course [GitHub Discussions page](#).

And of course, there's the [PyTorch documentation](#) and [PyTorch developer forums](#), a very helpful place for all things PyTorch.

## 0. Getting setup

Let's start by downloading all of the modules we'll need for this section.

To save us writing extra code, we're going to be leveraging some of the Python scripts (such as `data_setup.py` and `engine.py`) we created in section, [05. PyTorch Going Modular](#).

Specifically, we're going to download the `going_modular` directory from the `pytorch-deep-learning` repository (if we don't already have it).

We'll also get the `torchinfo` package if it's not available.

`torchinfo` will help later on to give us visual summaries of our model(s).

And since we're using a newer version of the `torchvision` package (v0.13 as of June 2022), we'll make sure we've got the latest versions.

In [1]:







```
torch version: 1.13.0.dev20220620+cu113
torchvision version: 0.14.0.dev20220620+cu113
```

**Note:** If you're using Google Colab, you may have to restart your runtime after running the above cell. After restarting, you can run the cell again and verify you've got the right versions of `torch` (0.12+) and `torchvision` (0.13+).

In [2]:







Now let's setup device agnostic code.

**Note:** If you're using Google Colab, and you don't have a GPU turned on yet, it's now time to turn one on via

Runtime -> Change runtime type -> Hardware accelerator -> GPU .

In [3]:

```
'cuda'
```

Out[3]:

## Create a helper function to set seeds

Since we've been setting random seeds a whole bunch throughout previous sections, how about we functionize it?

Let's create a function to "set the seeds" called `set_seeds()` .

**Note:** Recall a **random seed** is a way of flavouring the randomness generated by a computer. They aren't necessary to always set when running machine learning code, however, they help ensure there's an element of reproducibility (the numbers I get with my code are similar to the numbers you get with your code). Outside of an education or experimental setting, random seeds generally aren't required.

In [4]:





## 1. Get data

As always, before we can run machine learning experiments, we'll need a dataset.

We're going to continue trying to improve upon the results we've been getting on FoodVision Mini.

In the previous section, [06. PyTorch Transfer Learning](#), we saw how powerful using a pretrained model and transfer learning could be when classifying images of pizza, steak and sushi.

So how about we run some experiments and try to further improve our results?

To do so, we'll use similar code to the previous section to download the `pizza_steak_sushi.zip` (if the data doesn't already exist) except this time its been functionised.

This will allow us to use it again later.

In [5]:



























```
[INFO] data/pizza_steak_sushi directory exists, skipping download.  
Out[5]:  
PosixPath('data/pizza_steak_sushi')
```

Excellent! Looks like we've got our pizza, steak and sushi images in standard image classification format ready to go.

## 2. Create Datasets and DataLoaders

Now we've got some data, let's turn it into PyTorch DataLoaders.

We can do so using the `create_dataloaders()` function we created in [05. PyTorch Going Modular part 2](#).

And since we'll be using transfer learning and specifically pretrained models from `torchvision.models`, we'll create a transform to prepare our images correctly.

To transform our images in tensors, we can use:

1. Manually created transforms using `torchvision.transforms`.
2. Automatically created transforms using `torchvision.models.MODEL_NAME.MODEL_WEIGHTS.DEFAULT.transforms()`.
  - Where `MODEL_NAME` is a specific `torchvision.models` architecture, `MODEL_WEIGHTS` is a specific set of pretrained weights and `DEFAULT` means the "best available weights".

We saw an example of each of these in [06. PyTorch Transfer Learning section 2](#).

Let's see first an example of manually creating a `torchvision.transforms` pipeline (creating a transforms pipeline this way gives the most customization but can potentially result in performance degradation if the transforms don't match the pretrained model).

The main manual transformation we need to be sure of is that all of our images are normalized in ImageNet format (this is because pretrained `torchvision.models` are all pretrained on [ImageNet](#)).

We can do this with:

```
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],  
                               std=[0.229, 0.224, 0.225])
```

### 2.1 Create DataLoaders using manually created transforms

In [6]:









```
Manually created transforms: Compose(  
    Resize(size=(224, 224), interpolation=bilinear, max_size=None, antialias=None)  
    ToTensor()  
    Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  
)  
  
(<torch.utils.data.DataLoader at 0x7febf1d218e0>,  
<torch.utils.data.DataLoader at 0x7febf1d216a0>,  
 ['pizza', 'steak', 'sushi'])
```

Out[6]:

## 2.2 Create DataLoaders using automatically created transforms

Data transformed and DataLoaders created!

Let's now see what the same transformation pipeline looks like but this time by using automatic transforms.

We can do this by first instantiating a set of pretrained weights (for example `weights = torchvision.models.EfficientNet_B0_Weights.DEFAULT`) we'd like to use and calling the `transforms()` method on it.

In [7]:





```
Automatically created transforms: ImageClassification(  
    crop_size=[224]  
    resize_size=[256]  
    mean=[0.485, 0.456, 0.406]  
    std=[0.229, 0.224, 0.225]  
    interpolation=InterpolationMode.BICUBIC  
)  
(<torch.utils.data.dataloader.DataLoader at 0x7feb1d213a0>,  
<torch.utils.data.dataloader.DataLoader at 0x7feb1d21490>,  
 ['pizza', 'steak', 'sushi'])
```

Out[7]:

### 3. Getting a pretrained model, freezing the base layers and changing the classifier head

Before we run and track multiple modelling experiments, let's see what it's like to run and track a single one.

And since our data is ready, the next thing we'll need is a model.

Let's download the pretrained weights for a `torchvision.models.efficientnet_b0()` model and prepare it for use with our own data.

In [8]:





Wonderful!

Now we've got a pretrained model let's turn into a feature extractor model.

In essence, we'll freeze the base layers of the model (we'll use these to extract features from our input images) and we'll change the classifier head (output layer) to suit the number of classes we're working with (we've got 3 classes: pizza, steak, sushi).

**Note:** The idea of creating a feature extractor model (what we're doing here) was covered in more depth in [06. PyTorch Transfer Learning section 3.2: Setting up a pretrained model.](#)

In [9]:



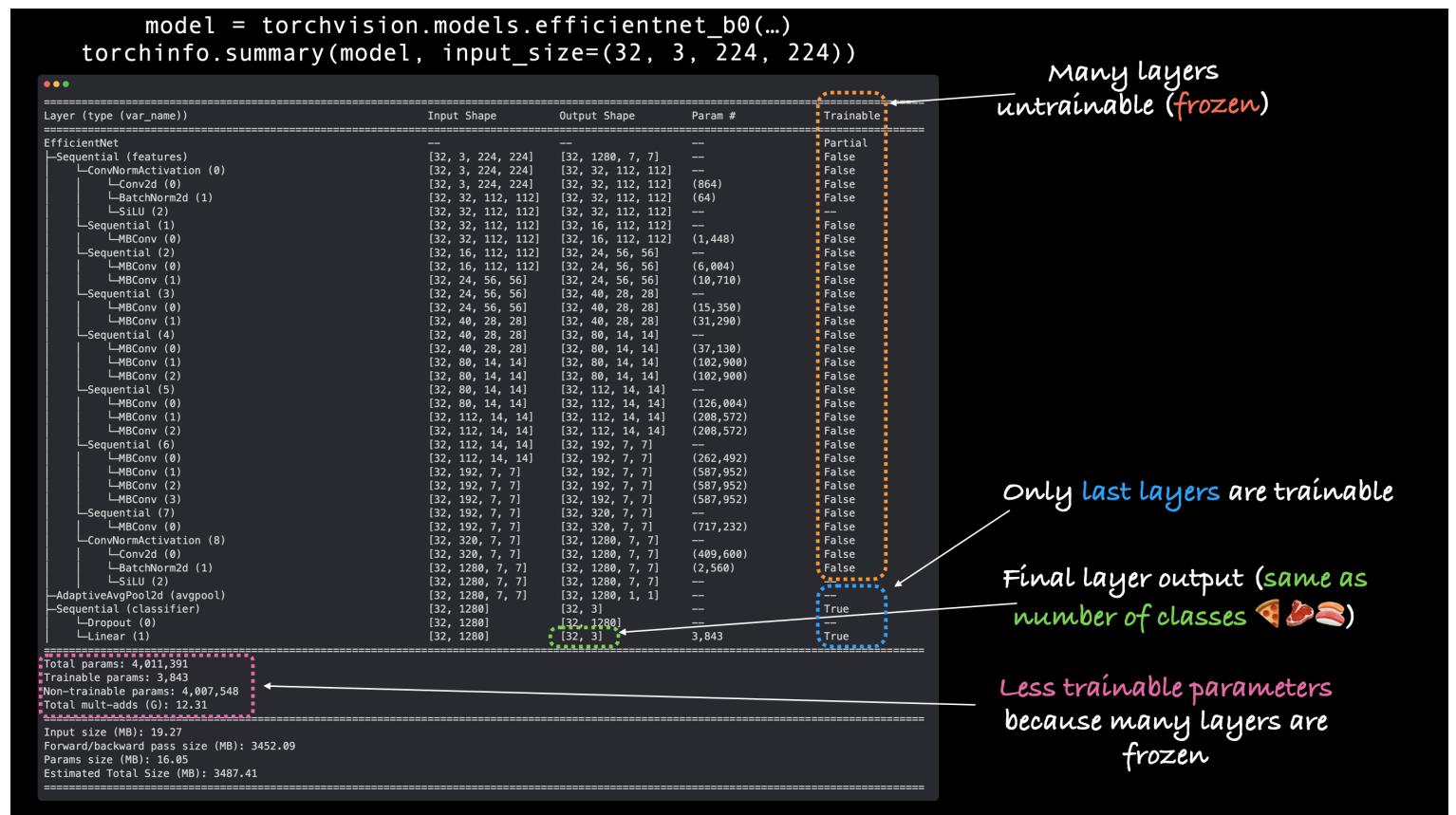


Base layers frozen, classifier head changed, let's get a summary of our model with `torchinfo.summary()`.

In [10]:







Output of `torchinfo.summary()` with our feature extractor EffNetB0 model, notice how the base layers are frozen (not trainable) and the output layers are customized to our own problem.

## 4. Train model and track results

Model ready to go!

Let's get ready to train it by creating a loss function and an optimizer.

Since we're working with multiple classes, we'll use `torch.nn.CrossEntropyLoss()` as the loss function.

And we'll stick with `torch.optim.Adam()` with learning rate of `0.001` for the optimizer.

In [11]:

## Adjust `train()` function to track results with `SummaryWriter()`

Beautiful!

All of the pieces of our training code are starting to come together.

Let's now add the final piece to track our experiments.

Previously, we've tracked our modelling experiments using multiple Python dictionaries (one for each model).

But you can imagine this could get out of hand if we were running anything more than a few experiments.

Not to worry, there's a better option!

We can use PyTorch's `torch.utils.tensorboard.SummaryWriter()` class to save various parts of our model's training progress to file.

By default, the `SummaryWriter()` class saves various information about our model to a file set by the `log_dir` parameter.

The default location for `log_dir` is under `runs/CURRENT_DATETIME_HOSTNAME`, where the `HOSTNAME` is the name of your computer.

But of course, you can change where your experiments are tracked (the filename is as customisable as you'd like).

The outputs of the `SummaryWriter()` are saved in **TensorBoard format**.

TensorBoard is a part of the TensorFlow deep learning library and is an excellent way to visualize different parts of your model.

To start tracking our modelling experiments, let's create a default `SummaryWriter()` instance.

In [12]:



Now to use the writer, we could write a new training loop or we could adjust the existing `train()` function we created in [05. PyTorch Going Modular section 4](#).

Let's take the latter option.

We'll get the `train()` function from `engine.py` and adjust it to use `writer`.

Specifically, we'll add the ability for our `train()` function to log our model's training and test loss and accuracy values.

We can do this with `writer.add_scalars(main_tag, tag_scalar_dict)`, where:

- `main_tag` (string) - the name for the scalars being tracked (e.g. "Accuracy")
- `tag_scalar_dict` (dict) - a dictionary of the values being tracked (e.g. `{"train_loss": 0.3454}`)
- **Note:** The method is called `add_scalars()` because our loss and accuracy values are generally scalars

(single values).

Once we've finished tracking values, we'll call `writer.close()` to tell the `writer` to stop looking for values to track.

To start modifying `train()` we'll also import `train_step()` and `test_step()` from `engine.py`.

**Note:** You can track information about your model almost anywhere in your code. But quite often experiments will be tracked *while* a model is training (inside a training/testing loop).

The `torch.utils.tensorboard.SummaryWriter()` class also has many different methods to track different things about your model/data, such as `add_graph()` which tracks the computation graph of your model. For more options, [check the `SummaryWriter\(\)` documentation](#).

In [13]:



































































Woohoo!

Our `train()` function is now updated to use a `SummaryWriter()` instance to track our model's results.

How about we try it out for 5 epochs?

In [14]:





```
0%|          | 0/5 [00:00<?, ?it/s]
Epoch: 1 | train_loss: 1.0924 | train_acc: 0.3984 | test_loss: 0.9133 | test_acc: 0.5398
Epoch: 2 | train_loss: 0.8975 | train_acc: 0.6562 | test_loss: 0.7838 | test_acc: 0.8561
Epoch: 3 | train_loss: 0.8037 | train_acc: 0.7461 | test_loss: 0.6723 | test_acc: 0.8864
Epoch: 4 | train_loss: 0.6769 | train_acc: 0.8516 | test_loss: 0.6698 | test_acc: 0.8049
Epoch: 5 | train_loss: 0.7065 | train_acc: 0.7188 | test_loss: 0.6746 | test_acc: 0.7737
```

**Note:** You might notice the results here are slightly different to what our model got in 06. PyTorch Transfer Learning. The difference comes from using the `engine.train()` and our modified `train()` function. Can you guess why? The [PyTorch documentation on randomness](#) may help more.

Running the cell above we get similar outputs we got in [06. PyTorch Transfer Learning section 4: Train model](#) but the difference is behind the scenes our `writer` instance has created a `runs/` directory storing our model's results.

For example, the save location might look like:

```
runs/Jun21_00-46-03_daniels_macbook_pro
```

Where the [default format](#) is `runs/CURRENT_DATETIME_HOSTNAME`.

We'll check these out in a second but just as a reminder, we were previously tracking our model's results in a dictionary.

In [15]:

Out[15]:

```
{'train_loss': [1.0923754647374153,
 0.8974628075957298,
 0.803724929690361,
 0.6769256368279457,
 0.7064960040152073],
'train_acc': [0.3984375, 0.65625, 0.74609375, 0.8515625, 0.71875],
'test_loss': [0.9132757981618246,
 0.7837507526079813,
 0.6722926497459412,
 0.6698453426361084,
 0.6746167540550232],
'test_acc': [0.5397727272727273,
 0.8560606060606061,
 0.8863636363636364,
 0.8049242424242425,
 0.7736742424242425] }
```

Hmmm, we could format this to be a nice plot but could you image keeping track of a bunch of these dictionaries?

There has to be a better way...

## 5. View our model's results in TensorBoard

The `SummaryWriter()` class stores our model's results in a directory called `runs/` in TensorBoard format by default.

TensorBoard is a visualization program created by the TensorFlow team to view and inspect information about models and data.

You know what that means?

It's time to follow the data visualizer's motto and *visualize, visualize, visualize!*

You can view TensorBoard in a number of ways:

Code environment	How to view TensorBoard	Resource
VS Code (notebooks or Python scripts)	Press <code>SHIFT + CMD + P</code> to open the Command Palette and search for the command "Python: Launch TensorBoard".	<a href="#">VS Code Guide on TensorBoard and PyTorch</a>
Jupyter and Colab Notebooks	Make sure <a href="#">TensorBoard is installed</a> , load it with <code>%load_ext tensorboard</code> and then view your results with <code>%tensorboard --logdir DIR_WITH_LOGS</code> .	<a href="#">torch.utils.tensorboard</a> and <a href="#">Get started with TensorBoard</a>

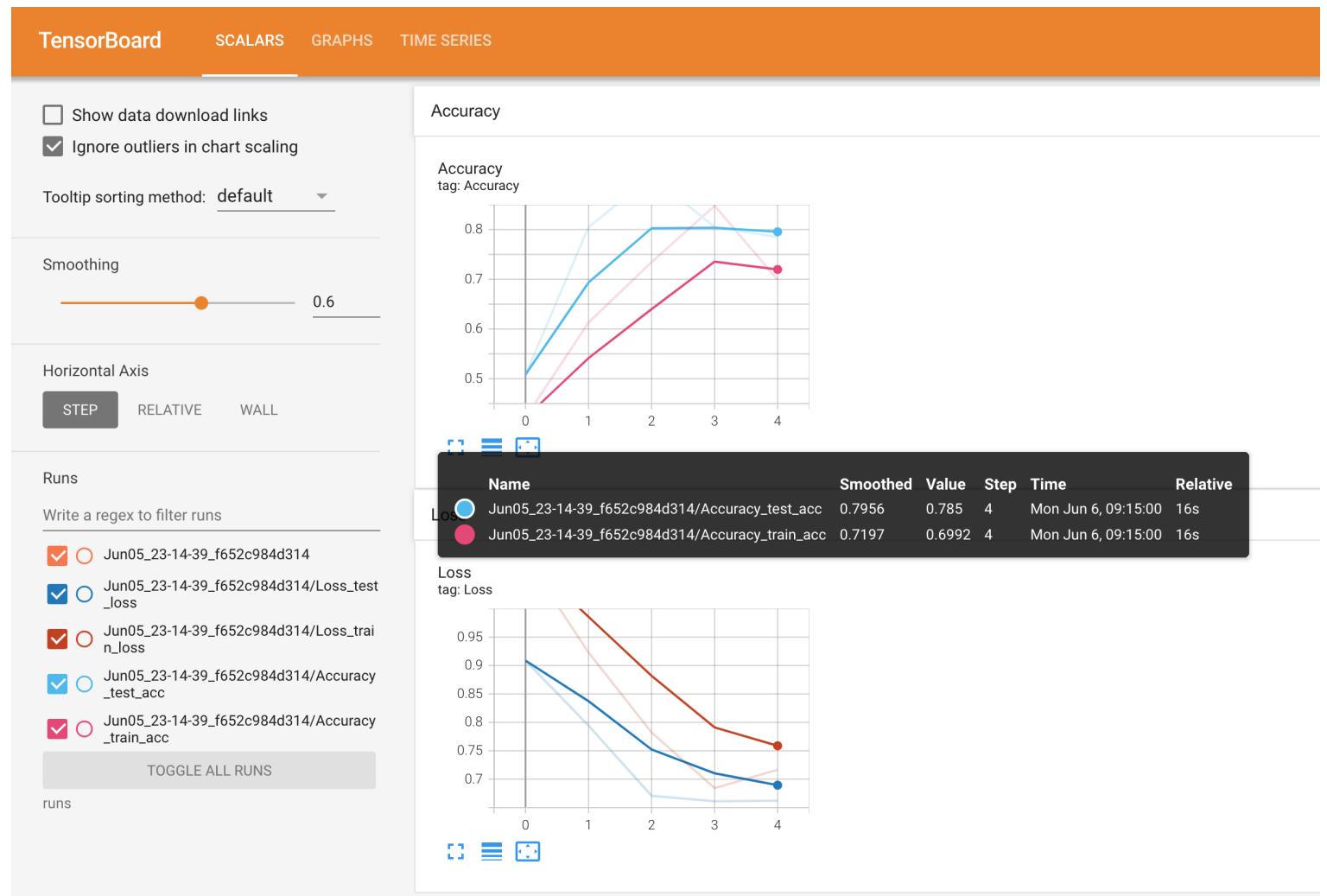
You can also upload your experiments to [tensorboard.dev](#) to share them publicly with others.

Running the following code in a Google Colab or Jupyter Notebook will start an interactive TensorBoard session to view TensorBoard files in the `runs/` directory.

```
%load_ext tensorboard # line magic to load TensorBoard
%tensorboard --logdir runs # run TensorBoard session with the "runs/" directory
```

In [16]:

If all went correctly, you should see something like the following:



*Viewing a single modelling experiment's results for accuracy and loss in TensorBoard.*

**Note:** For more information on running TensorBoard in notebooks or in other locations, see the following:

- [Using TensorBoard in Notebooks guide by TensorFlow](#)
- [Get started with TensorBoard.dev](#) (helpful for uploading your TensorBoard logs to a shareable link)

## 6. Create a helper function to build `SummaryWriter()` instances

The `SummaryWriter()` class logs various information to a directory specified by the `log_dir` parameter.

How about we make a helper function to create a custom directory per experiment?

In essence, each experiment gets its own logs directory.

For example, say we'd like to track things like:

- **Experiment date/timestamp** - when did the experiment take place?
- **Experiment name** - is there something we'd like to call the experiment?
- **Model name** - what model was used?
- **Extra** - should anything else be tracked?

You could track almost anything here and be as creative as you want but these should be enough to start.

Let's create a helper function called `create_writer()` that produces a `SummaryWriter()` instance tracking to a custom `log_dir`.

Ideally, we'd like the `log_dir` to be something like:

```
runs/YYYY-MM-DD/experiment_name/model_name/extra
```

Where `YYYY-MM-DD` is the date the experiment was run (you could add the time if you wanted to as well).

In [17]:





















Beautiful!

Now we've got a `create_writer()` function, let's try it out.

In [18]:



```
[INFO] Created SummaryWriter, saving to: runs/2022-06-23/data_10_percent/effnetb0/5_epochs...
```

Looking good, now we've got a way to log and trace back our various experiments.

## 6.1 Update the `train()` function to include a `writer` parameter

Our `create_writer()` function works fantastic.

How about we give our `train()` function the ability to take in a `writer` parameter so we actively update the `SummaryWriter()` instance we're using each time we call `train()`.

For example, say we're running a series of experiments, calling `train()` multiple times for multiple different models, it would be good if each experiment used a different `writer`.

One `writer` per experiment = one logs directory per experiment.

To adjust the `train()` function we'll add a `writer` parameter to the function and then we'll add some code to see if there's a `writer` and if so, we'll track our information there.

In [19]:



































































## 7. Setting up a series of modelling experiments

It's to step things up a notch.

Previously we've been running various experiments and inspecting the results one by one.

But what if we could run multiple experiments and then inspect the results all together?

You in?

C'mon, let's go.

### 7.1 What kind of experiments should you run?

That's the million dollar question in machine learning.

Because there's really no limit to the experiments you can run.

Such a freedom is why machine learning is so exciting and terrifying at the same time.

This is where you'll have to put on your scientist coat and remember the machine learning practitioner's motto: *experiment, experiment, experiment!*

Every hyperparameter stands as a starting point for a different experiment:

- Change the number of **epochs**.
- Change the number of **layers/hidden units**.
- Change the amount of **data**.
- Change the **learning rate**.
- Try different kinds of **data augmentation**.
- Choose a different **model architecture**.

With practice and running many different experiments, you'll start to build an intuition of what *might* help your

model.

I say *might* on purpose because there's no guarantees.

But generally, in light of *The Bitter Lesson* (I've mentioned this twice now because it's an important essay in the world of AI), generally the bigger your model (more learnable parameters) and the more data you have (more opportunities to learn), the better the performance.

However, when you're first approaching a machine learning problem: start small and if something works, scale it up.

Your first batch of experiments should take no longer than a few seconds to a few minutes to run.

The quicker you can experiment, the faster you can work out what *doesn't* work, in turn, the faster you can work out what *does* work.

## 7.2 What experiments are we going to run?

Our goal is to improve the model powering FoodVision Mini without it getting too big.

In essence, our ideal model achieves a high level of test set accuracy (90%+) but doesn't take too long to train/perform inference (make predictions).

We've got plenty of options but how about we keep things simple?

Let's try a combination of:

1. A different amount of data (10% of Pizza, Steak, Sushi vs. 20%)
2. A different model (`torchvision.models.efficientnet_b0` vs. `torchvision.models.efficientnet_b2`)
3. A different training time (5 epochs vs. 10 epochs)

Breaking these down we get:

Experiment number	Training Dataset	Model (pretrained on ImageNet)	Number of epochs
1	Pizza, Steak, Sushi 10% percent	EfficientNetB0	5
2	Pizza, Steak, Sushi 10% percent	EfficientNetB2	5
3	Pizza, Steak, Sushi 10% percent	EfficientNetB0	10
4	Pizza, Steak, Sushi 10% percent	EfficientNetB2	10

5	Pizza, Steak, Sushi 20% percent	EfficientNetB0	5
6	Pizza, Steak, Sushi 20% percent	EfficientNetB2	5
7	Pizza, Steak, Sushi 20% percent	EfficientNetB0	10
8	Pizza, Steak, Sushi 20% percent	EfficientNetB2	10

Notice how we're slowly scaling things up.

With each experiment we slowly increase the amount of data, the model size and the length of training.

By the end, experiment 8 will be using double the data, double the model size and double the length of training compared to experiment 1.

**Note:** I want to be clear that there truly is no limit to amount of experiments you can run. What we've designed here is only a very small subset of options. However, you can't test *everything* so best to try a few things to begin with and then follow the ones which work the best.

And as a reminder, the datasets we're using are a subset of the [Food101 dataset](#) (3 classes, pizza, steak, suhs, instead of 101) and 10% and 20% of the images rather than 100%. If our experiments work, we could start to run more on more data (though this will take longer to compute). You can see how the datasets were created via the [04\\_custom\\_data\\_creation.ipynb notebook](#).

## 7.3 Download different datasets

Before we start running our series of experiments, we need to make sure our datasets are ready.

We'll need two forms of a training set:

1. A training set with **10% of the data** of Food101 pizza, steak, sushi images (we've already created this above but we'll do it again for completeness).
2. A training set with **20% of the data** of Food101 pizza, steak, sushi images.

For consistency, all experiments will use the same testing dataset (the one from the 10% data split).

We'll start by downloading the various datasets we need using the `download_data()` function we created earlier.

Both datasets are available from the course GitHub:

1. [Pizza, steak, sushi 10% training data](#).

## 2. Pizza, steak, sushi 20% training data.

In [20]:



```
[INFO] data/pizza_steak_sushi directory exists, skipping download.  
[INFO] data/pizza_steak_sushi_20_percent directory exists, skipping download.
```

Data downloaded!

Now let's setup the filepaths to data we'll be using for the different experiments.

We'll create different training directory paths but we'll only need one testing directory path since all experiments will be using the same test dataset (the test dataset from pizza, steak, sushi 10%).

In [21]:



```
Training directory 10%: data/pizza_steak_sushi/train  
Training directory 20%: data/pizza_steak_sushi_20_percent/train  
Testing directory: data/pizza_steak_sushi/test
```

## 7.4 Transform Datasets and create DataLoaders

Next we'll create a series of transforms to prepare our images for our model(s).

To keep things consistent, we'll manually create a transform (just like we did above) and use the same transform across all of the datasets.

The transform will:

1. Resize all the images (we'll start with 224, 224 but this could be changed).
2. Turn them into tensors with values between 0 & 1.
3. Normalize them in way so their distributions are inline with the ImageNet dataset (we do this because our models from `torchvision.models` have been pretrained on ImageNet).

In [22]:









Transform ready!

Now let's create our DataLoaders using the `create_dataloaders()` function from `data_setup.py` we created in [05. PyTorch Going Modular section 2](#).

We'll create the DataLoaders with a batch size of 32.

For all of our experiments we'll be using the same `test_dataloader` (to keep comparisons consistent).

In [23]:







```
Number of batches of size 32 in 10 percent training data: 8
Number of batches of size 32 in 20 percent training data: 15
Number of batches of size 32 in testing data: 8 (all experiments will use the same test set)
Number of classes: 3, class names: ['pizza', 'steak', 'sushi']
```

## 7.5 Create feature extractor models

Time to start building our models.

We're going to create two feature extractor models:

1. `torchvision.models.efficientnet_b0()` pretrained backbone + custom classifier head (EffNetB0 for short).
2. `torchvision.models.efficientnet_b2()` pretrained backbone + custom classifier head (EffNetB2 for short).

To do this, we'll freeze the base layers (the feature layers) and update the model's classifier heads (output layers) to suit our problem just like we did in [06. PyTorch Transfer Learning section 3.4](#).

We saw in the previous chapter the `in_features` parameter to the classifier head of EffNetB0 is `1280` (the backbone turns the input image into a feature vector of size `1280`).

Since EffNetB2 has a different number of layers and parameters, we'll need to adapt it accordingly.

**Note:** Whenever you use a different model, one of the first things you should inspect is the input and output shapes. That way you'll know how you'll have to prepare your input data/update the model to have the correct output shape.

We can find the input and output shapes of EffNetB2 using `torchinfo.summary()` and passing in the `input_size=(32, 3, 224, 224)` parameter (`(32, 3, 224, 224)` is equivalent to `(batch_size, color_channels, height, width)`, i.e we pass in an example of what a single batch of data would be to our model).

**Note:** Many modern models can handle input images of varying sizes thanks to `torch.nn.AdaptiveAvgPool2d()` layer, this layer adaptively adjusts the `output_size` of a given input as required. You can try this out by passing different size input images to `torchinfo.summary()` or to your own models using the layer.

To find the required input shape to the final layer of EffNetB2, let's:

1. Create an instance of `torchvision.models.efficientnet_b2(pretrained=True)` .
2. See the various input and output shapes by running `torchinfo.summary()` .
3. Print out the number of `in_features` by inspecting `state_dict()` of the classifier portion of EffNetB2 and printing the length of the weight matrix.
  - **Note:** You could also just inspect the output of `effnetb2.classifier` .

In [24]:









Number of `in_features` to final layer of EfficientNetB2: 1408

Layer (type (var_name))	Input Shape	Output Shape	Param #	Trainable
EfficientNet (EfficientNet)	[32, 3, 224, 224]	[32, 1000]	--	True
Sequential (features)	[32, 3, 224, 224]	[32, 1408, 7, 7]	--	True
└Conv2dNormActivation (0)	[32, 3, 224, 224]	[32, 32, 112, 112]	--	True
└Conv2d (0)	[32, 3, 224, 224]	[32, 32, 112, 112]	864	True
└BatchNorm2d (1)	[32, 32, 112, 112]	[32, 32, 112, 112]	64	True
└SiLU (2)	[32, 32, 112, 112]	[32, 32, 112, 112]	--	--
└Sequential (1)	[32, 32, 112, 112]	[32, 16, 112, 112]	--	True
└MBConv (0)	[32, 32, 112, 112]	[32, 16, 112, 112]	1,448	True
└MBConv (1)	[32, 16, 112, 112]	[32, 16, 112, 112]	612	True
└Sequential (2)	[32, 16, 112, 112]	[32, 24, 56, 56]	--	True
└MBConv (0)	[32, 16, 112, 112]	[32, 24, 56, 56]	6,004	True
└MBConv (1)	[32, 24, 56, 56]	[32, 24, 56, 56]	10,710	True
└MBConv (2)	[32, 24, 56, 56]	[32, 24, 56, 56]	10,710	True
└Sequential (3)	[32, 24, 56, 56]	[32, 48, 28, 28]	--	True
└MBConv (0)	[32, 24, 56, 56]	[32, 48, 28, 28]	16,518	True
└MBConv (1)	[32, 48, 28, 28]	[32, 48, 28, 28]	43,308	True
└MBConv (2)	[32, 48, 28, 28]	[32, 48, 28, 28]	43,308	True
└Sequential (4)	[32, 48, 28, 28]	[32, 88, 14, 14]	--	True
└MBConv (0)	[32, 48, 28, 28]	[32, 88, 14, 14]	50,300	True
└MBConv (1)	[32, 88, 14, 14]	[32, 88, 14, 14]	123,750	True
└MBConv (2)	[32, 88, 14, 14]	[32, 88, 14, 14]	123,750	True
└MBConv (3)	[32, 88, 14, 14]	[32, 88, 14, 14]	123,750	True
└Sequential (5)	[32, 88, 14, 14]	[32, 120, 14, 14]	--	True
└MBConv (0)	[32, 88, 14, 14]	[32, 120, 14, 14]	149,158	True
└MBConv (1)	[32, 120, 14, 14]	[32, 120, 14, 14]	237,870	True
└MBConv (2)	[32, 120, 14, 14]	[32, 120, 14, 14]	237,870	True
└MBConv (3)	[32, 120, 14, 14]	[32, 120, 14, 14]	237,870	True
└Sequential (6)	[32, 120, 14, 14]	[32, 208, 7, 7]	--	True
└MBConv (0)	[32, 120, 14, 14]	[32, 208, 7, 7]	301,406	True
└MBConv (1)	[32, 208, 7, 7]	[32, 208, 7, 7]	686,868	True
└MBConv (2)	[32, 208, 7, 7]	[32, 208, 7, 7]	686,868	True
└MBConv (3)	[32, 208, 7, 7]	[32, 208, 7, 7]	686,868	True
└MBConv (4)	[32, 208, 7, 7]	[32, 208, 7, 7]	686,868	True
└Sequential (7)	[32, 208, 7, 7]	[32, 352, 7, 7]	--	True
└MBConv (0)	[32, 208, 7, 7]	[32, 352, 7, 7]	846,900	True
└MBConv (1)	[32, 352, 7, 7]	[32, 352, 7, 7]	1,888,920	True
└Conv2dNormActivation (8)	[32, 352, 7, 7]	[32, 1408, 7, 7]	--	True
└Conv2d (0)	[32, 352, 7, 7]	[32, 1408, 7, 7]	495,616	True
└BatchNorm2d (1)	[32, 1408, 7, 7]	[32, 1408, 7, 7]	2,816	True
└SiLU (2)	[32, 1408, 7, 7]	[32, 1408, 7, 7]	--	--
└AdaptiveAvgPool2d (avgpool)	[32, 1408, 7, 7]	[32, 1408, 1, 1]	--	--
└Sequential (classifier)	[32, 1408]	[32, 1000]	--	True
└Dropout (0)	[32, 1408]	[32, 1408]	--	--
└Linear (1)	[32, 1408]	[32, 1000]	1,409,000	True

Total params: 9,109,994  
 Trainable params: 9,109,994  
 Non-trainable params: 0  
 Total mult-adds (G): 21.09

Input size (MB): 19.27  
 Forward/backward pass size (MB): 5017.79  
 Params size (MB): 36.44  
 Estimated Total Size (MB): 5073.49

*Model summary of EffNetB2 feature extractor model with all layers unfrozen (trainable) and default classifier head from ImageNet pretraining.*

Now we know the required number of `in_features` for the EffNetB2 model, let's create a couple of helper functions to setup our EffNetB0 and EffNetB2 feature extractor models.

We want these functions to:

1. Get the base model from `torchvision.models`
2. Freeze the base layers in the model (set `requires_grad=False`)
3. Set the random seeds (we don't *need* to do this but since we're running a series of experiments and initializing a new layer with random weights, we want the randomness to be similar for each experiment)
4. Change the classifier head (to suit our problem)
5. Give the model a name (e.g. "effnetb0" for EffNetB0)

In [25]:



















Those are some nice looking functions!

Let's test them out by creating an instance of EffNetB0 and EffNetB2 and checking out their `summary()`.

In [26]:





[INFO] Created new effnetb0 model.

Layer (type (var_name))	Input Shape	Output Shape	Param #	Trainable
EfficientNet (EfficientNet)	[32, 3, 224, 224]	[32, 3]	--	Partial
Sequential (features)	[32, 3, 224, 224]	[32, 1280, 7, 7]	--	False
Conv2dNormActivation (0)	[32, 3, 224, 224]	[32, 32, 112, 112]	--	False
Conv2d (0)	[32, 3, 224, 224]	[32, 32, 112, 112]	(864)	False
BatchNorm2d (1)	[32, 32, 112, 112]	[32, 32, 112, 112]	(64)	False
SiLU (2)	[32, 32, 112, 112]	[32, 32, 112, 112]	--	--
Sequential (1)	[32, 32, 112, 112]	[32, 16, 112, 112]	--	False
MBConv (0)	[32, 32, 112, 112]	[32, 16, 112, 112]	(1,448)	False
Sequential (2)	[32, 16, 112, 112]	[32, 24, 56, 56]	--	False
MBConv (0)	[32, 16, 112, 112]	[32, 24, 56, 56]	(6,004)	False
MBConv (1)	[32, 24, 56, 56]	[32, 24, 56, 56]	(10,710)	False
Sequential (3)	[32, 24, 56, 56]	[32, 40, 28, 28]	--	False
MBConv (0)	[32, 24, 56, 56]	[32, 40, 28, 28]	(15,350)	False
MBConv (1)	[32, 40, 28, 28]	[32, 40, 28, 28]	(31,290)	False
Sequential (4)	[32, 40, 28, 28]	[32, 80, 14, 14]	--	False
MBConv (0)	[32, 40, 28, 28]	[32, 80, 14, 14]	(37,130)	False
MBConv (1)	[32, 80, 14, 14]	[32, 80, 14, 14]	(102,900)	False
MBConv (2)	[32, 80, 14, 14]	[32, 80, 14, 14]	(102,900)	False
Sequential (5)	[32, 80, 14, 14]	[32, 112, 14, 14]	--	False
MBConv (0)	[32, 80, 14, 14]	[32, 112, 14, 14]	(126,004)	False
MBConv (1)	[32, 112, 14, 14]	[32, 112, 14, 14]	(208,572)	False
MBConv (2)	[32, 112, 14, 14]	[32, 112, 14, 14]	(208,572)	False
Sequential (6)	[32, 112, 14, 14]	[32, 192, 7, 7]	--	False
MBConv (0)	[32, 112, 14, 14]	[32, 192, 7, 7]	(262,492)	False
MBConv (1)	[32, 192, 7, 7]	[32, 192, 7, 7]	(587,952)	False
MBConv (2)	[32, 192, 7, 7]	[32, 192, 7, 7]	(587,952)	False
MBConv (3)	[32, 192, 7, 7]	[32, 192, 7, 7]	(587,952)	False
Sequential (7)	[32, 192, 7, 7]	[32, 320, 7, 7]	--	False
MBConv (0)	[32, 192, 7, 7]	[32, 320, 7, 7]	(717,232)	False
Conv2dNormActivation (8)	[32, 320, 7, 7]	[32, 1280, 7, 7]	--	False
Conv2d (0)	[32, 320, 7, 7]	[32, 1280, 7, 7]	(409,600)	False
BatchNorm2d (1)	[32, 1280, 7, 7]	[32, 1280, 7, 7]	(2,560)	False
SiLU (2)	[32, 1280, 7, 7]	[32, 1280, 7, 7]	--	--
AdaptiveAvgPool2d (avgpool)	[32, 1280, 7, 7]	[32, 1280, 1, 1]	--	--
Sequential (classifier)	[32, 1280]	[32, 3]	--	True
Dropout (0)	[32, 1280]	[32, 1280]	--	--
Linear (1)	[32, 1280]	[32, 3]	3,843	True

Total params: 4,011,391  
 Trainable params: 3,843  
 Non-trainable params: 4,007,548  
 Total mult-adds (G): 12.31

Input size (MB): 19.27  
 Forward/backward pass size (MB): 3452.09  
 Params size (MB): 16.05  
 Estimated Total Size (MB): 3487.41

*Model summary of EffNetB0 model with base layers frozen (untrainable) and updated classifier head (suited for pizza, steak, sushi image classification).*

In [27]:





```
[INFO] Created new effnetb2 model.
```

Layer (type (var_name))	Input Shape	Output Shape	Param #	Trainable
EfficientNet (EfficientNet)	[32, 3, 224, 224]	[32, 3]	--	Partial
Sequential (features)	[32, 3, 224, 224]	[32, 1408, 7, 7]	--	False
Conv2dNormActivation (0)	[32, 3, 224, 224]	[32, 32, 112, 112]	--	False
Conv2d (0)	[32, 3, 224, 224]	[32, 32, 112, 112]	(864)	False
BatchNorm2d (1)	[32, 32, 112, 112]	[32, 32, 112, 112]	(64)	False
SiLU (2)	[32, 32, 112, 112]	[32, 32, 112, 112]	--	--
Sequential (1)	[32, 32, 112, 112]	[32, 16, 112, 112]	--	False
MBConv (0)	[32, 32, 112, 112]	[32, 16, 112, 112]	(1,448)	False
MBConv (1)	[32, 16, 112, 112]	[32, 16, 112, 112]	(612)	False
Sequential (2)	[32, 16, 112, 112]	[32, 24, 56, 56]	--	False
MBConv (0)	[32, 16, 112, 112]	[32, 24, 56, 56]	(6,004)	False
MBConv (1)	[32, 24, 56, 56]	[32, 24, 56, 56]	(10,710)	False
MBConv (2)	[32, 24, 56, 56]	[32, 24, 56, 56]	(10,710)	False
Sequential (3)	[32, 24, 56, 56]	[32, 48, 28, 28]	--	False
MBConv (0)	[32, 24, 56, 56]	[32, 48, 28, 28]	(16,518)	False
MBConv (1)	[32, 48, 28, 28]	[32, 48, 28, 28]	(43,308)	False
MBConv (2)	[32, 48, 28, 28]	[32, 48, 28, 28]	(43,308)	False
Sequential (4)	[32, 48, 28, 28]	[32, 88, 14, 14]	--	False
MBConv (0)	[32, 48, 28, 28]	[32, 88, 14, 14]	(50,300)	False
MBConv (1)	[32, 88, 14, 14]	[32, 88, 14, 14]	(123,750)	False
MBConv (2)	[32, 88, 14, 14]	[32, 88, 14, 14]	(123,750)	False
MBConv (3)	[32, 88, 14, 14]	[32, 88, 14, 14]	(123,750)	False
Sequential (5)	[32, 88, 14, 14]	[32, 120, 14, 14]	--	False
MBConv (0)	[32, 88, 14, 14]	[32, 120, 14, 14]	(149,158)	False
MBConv (1)	[32, 120, 14, 14]	[32, 120, 14, 14]	(237,870)	False
MBConv (2)	[32, 120, 14, 14]	[32, 120, 14, 14]	(237,870)	False
MBConv (3)	[32, 120, 14, 14]	[32, 120, 14, 14]	(237,870)	False
Sequential (6)	[32, 120, 14, 14]	[32, 208, 7, 7]	--	False
MBConv (0)	[32, 120, 14, 14]	[32, 208, 7, 7]	(301,406)	False
MBConv (1)	[32, 208, 7, 7]	[32, 208, 7, 7]	(686,868)	False
MBConv (2)	[32, 208, 7, 7]	[32, 208, 7, 7]	(686,868)	False
MBConv (3)	[32, 208, 7, 7]	[32, 208, 7, 7]	(686,868)	False
MBConv (4)	[32, 208, 7, 7]	[32, 208, 7, 7]	(686,868)	False
Sequential (7)	[32, 208, 7, 7]	[32, 352, 7, 7]	--	False
MBConv (0)	[32, 208, 7, 7]	[32, 352, 7, 7]	(846,900)	False
MBConv (1)	[32, 352, 7, 7]	[32, 352, 7, 7]	(1,888,920)	False
Conv2dNormActivation (8)	[32, 352, 7, 7]	[32, 1408, 7, 7]	--	False
Conv2d (0)	[32, 352, 7, 7]	[32, 1408, 7, 7]	(495,616)	False
BatchNorm2d (1)	[32, 1408, 7, 7]	[32, 1408, 7, 7]	(2,816)	False
SiLU (2)	[32, 1408, 7, 7]	[32, 1408, 7, 7]	--	--
AdaptiveAvgPool2d (avgpool)	[32, 1408]	[32, 3]	--	True
Sequential (classifier)	[32, 1408]	[32, 1408]	--	--
Dropout (0)	[32, 1408]	[32, 1408]	--	--
Linear (1)	[32, 1408]	[32, 3]	4,227	True

Total params: 7,705,221  
 Trainable params: 4,227  
 Non-trainable params: 7,700,994  
 Total mult-adds (G): 21.04

Input size (MB): 19.27  
 Forward/backward pass size (MB): 5017.53  
 Params size (MB): 30.82  
 Estimated Total Size (MB): 5067.62

Model summary of EffNetB2 model with base layers frozen (untrainable) and updated classifier head (suited for pizza, steak, sushi image classification).

Looking at the outputs of the summaries, it seems the EffNetB2 backbone has nearly double the amount of parameters as EffNetB0.

Model	Total parameters (before freezing/changing head)	Total parameters (after freezing/changing head)	Total trainable parameters (after freezing/changing head)
EfficientNetB0	5,288,548	4,011,391	3,843

EfficientNetB2	9,109,994	7,705,221	4,227
----------------	-----------	-----------	-------

This gives the backbone of the EffNetB2 model more opportunities to form a representation of our pizza, steak and sushi data.

However, the trainable parameters for each model (the classifier heads) aren't very different.

Will these extra parameters lead to better results?

We'll have to wait and see...

**Note:** In the spirit of experimenting, you really could try almost any model from `torchvision.models` in a similar fashion to what we're doing here. I've only chosen EffNetB0 and EffNetB2 as examples. Perhaps you might want to throw something like `torchvision.models.convnext_tiny()` or `torchvision.models.convnext_small()` into the mix.

## 7.6 Create experiments and set up training code

We've prepared our data and prepared our models, the time has come to setup some experiments!

We'll start by creating two lists and a dictionary:

1. A list of the number of epochs we'd like to test ( `[5, 10]` )
2. A list of the models we'd like to test ( `["effnetb0", "effnetb2"]` )
3. A dictionary of the different training DataLoaders

In [28]:



Lists and dictionary created!

Now we can write code to iterate through each of the different options and try out each of the different combinations.

We'll also save the model at the end of each experiment so later on we can load back in the best model and use it for making predictions.

Specifically, let's go through the following steps:

1. Set the random seeds (so our experiment results are reproducible, in practice, you might run the same experiment across ~3 different seeds and average the results).
2. Keep track of different experiment numbers (this is mostly for pretty print outs).
3. Loop through the `train_dataloaders` dictionary items for each of the different training DataLoaders.
4. Loop through the list of epoch numbers.

5. Loop through the list of different model names.
6. Create information print outs for the current running experiment (so we know what's happening).
7. Check which model is the target model and create a new EffNetB0 or EffNetB2 instance (we create a new model instance each experiment so all models start from the same standpoint).
8. Create a new loss function (`torch.nn.CrossEntropyLoss()`) and optimizer (`torch.optim.Adam(params=model.parameters(), lr=0.001)`) for each new experiment.
9. Train the model with the modified `train()` function passing the appropriate details to the `writer` parameter.
10. Save the trained model with an appropriate file name to file with `save_model()` from `utils.py`.

We can also use the `%%time` magic to see how long all of our experiments take together in a single Jupyter/Google Colab cell.

Let's do it!

In [29]:



































```
[INFO] Experiment number: 1
[INFO] Model: effnetb0
[INFO] DataLoader: data_10_percent
[INFO] Number of epochs: 5
[INFO] Created new effnetb0 model.
[INFO] Created SummaryWriter, saving to: runs/2022-06-23/data_10_percent/effnetb0/5_epochs...
    0% |          0/5 [00:00<?, ?it/s]
Epoch: 1 | train_loss: 1.0528 | train_acc: 0.4961 | test_loss: 0.9217 | test_acc: 0.4678
Epoch: 2 | train_loss: 0.8747 | train_acc: 0.6992 | test_loss: 0.8138 | test_acc: 0.6203
Epoch: 3 | train_loss: 0.8099 | train_acc: 0.6445 | test_loss: 0.7175 | test_acc: 0.8258
Epoch: 4 | train_loss: 0.7097 | train_acc: 0.7578 | test_loss: 0.5897 | test_acc: 0.8864
Epoch: 5 | train_loss: 0.5980 | train_acc: 0.9141 | test_loss: 0.5676 | test_acc: 0.8864
[INFO] Saving model to: models/07_effnetb0_data_10_percent_5_epochs.pth
-----
[INFO] Experiment number: 2
[INFO] Model: effnetb2
[INFO] DataLoader: data_10_percent
[INFO] Number of epochs: 5
[INFO] Created new effnetb2 model.
[INFO] Created SummaryWriter, saving to: runs/2022-06-23/data_10_percent/effnetb2/5_epochs...
    0% |          0/5 [00:00<?, ?it/s]
Epoch: 1 | train_loss: 1.0928 | train_acc: 0.3711 | test_loss: 0.9557 | test_acc: 0.6610
Epoch: 2 | train_loss: 0.9247 | train_acc: 0.6445 | test_loss: 0.8711 | test_acc: 0.8144
Epoch: 3 | train_loss: 0.8086 | train_acc: 0.7656 | test_loss: 0.7511 | test_acc: 0.9176
Epoch: 4 | train_loss: 0.7191 | train_acc: 0.8867 | test_loss: 0.7150 | test_acc: 0.9081
Epoch: 5 | train_loss: 0.6851 | train_acc: 0.7695 | test_loss: 0.7076 | test_acc: 0.8873
[INFO] Saving model to: models/07_effnetb2_data_10_percent_5_epochs.pth
-----
[INFO] Experiment number: 3
[INFO] Model: effnetb0
[INFO] DataLoader: data_10_percent
```

## 07. PyTorch Experiment Tracking - Zero to Mastery Learn PyTorch for Deep Learning

```
[INFO] Number of epochs: 10
[INFO] Created new effnetb0 model.
[INFO] Created SummaryWriter, saving to: runs/2022-06-23/data_10_percent/effnetb0/10_epochs...
 0% | 0/10 [00:00<?, ?it/s]
Epoch: 1 | train_loss: 1.0528 | train_acc: 0.4961 | test_loss: 0.9217 | test_acc: 0.4678
Epoch: 2 | train_loss: 0.8747 | train_acc: 0.6992 | test_loss: 0.8138 | test_acc: 0.6203
Epoch: 3 | train_loss: 0.8099 | train_acc: 0.6445 | test_loss: 0.7175 | test_acc: 0.8258
Epoch: 4 | train_loss: 0.7097 | train_acc: 0.7578 | test_loss: 0.5897 | test_acc: 0.8864
Epoch: 5 | train_loss: 0.5980 | train_acc: 0.9141 | test_loss: 0.5676 | test_acc: 0.8864
Epoch: 6 | train_loss: 0.5611 | train_acc: 0.8984 | test_loss: 0.5949 | test_acc: 0.8864
Epoch: 7 | train_loss: 0.5573 | train_acc: 0.7930 | test_loss: 0.5566 | test_acc: 0.8864
Epoch: 8 | train_loss: 0.4702 | train_acc: 0.9492 | test_loss: 0.5176 | test_acc: 0.8759
Epoch: 9 | train_loss: 0.5728 | train_acc: 0.7773 | test_loss: 0.5095 | test_acc: 0.8873
Epoch: 10 | train_loss: 0.4794 | train_acc: 0.8242 | test_loss: 0.4640 | test_acc: 0.9072
[INFO] Saving model to: models/07_effnetb0_data_10_percent_10_epochs.pth
-----
```

```
[INFO] Experiment number: 4
[INFO] Model: effnetb2
[INFO] DataLoader: data_10_percent
[INFO] Number of epochs: 10
[INFO] Created new effnetb2 model.
[INFO] Created SummaryWriter, saving to: runs/2022-06-23/data_10_percent/effnetb2/10_epochs...
 0% | 0/10 [00:00<?, ?it/s]
Epoch: 1 | train_loss: 1.0928 | train_acc: 0.3711 | test_loss: 0.9557 | test_acc: 0.6610
Epoch: 2 | train_loss: 0.9247 | train_acc: 0.6445 | test_loss: 0.8711 | test_acc: 0.8144
Epoch: 3 | train_loss: 0.8086 | train_acc: 0.7656 | test_loss: 0.7511 | test_acc: 0.9176
Epoch: 4 | train_loss: 0.7191 | train_acc: 0.8867 | test_loss: 0.7150 | test_acc: 0.9081
Epoch: 5 | train_loss: 0.6851 | train_acc: 0.7695 | test_loss: 0.7076 | test_acc: 0.8873
Epoch: 6 | train_loss: 0.6111 | train_acc: 0.7812 | test_loss: 0.6325 | test_acc: 0.9280
Epoch: 7 | train_loss: 0.6127 | train_acc: 0.8008 | test_loss: 0.6404 | test_acc: 0.8769
Epoch: 8 | train_loss: 0.5202 | train_acc: 0.9336 | test_loss: 0.6200 | test_acc: 0.8977
Epoch: 9 | train_loss: 0.5425 | train_acc: 0.8008 | test_loss: 0.6227 | test_acc: 0.8466
Epoch: 10 | train_loss: 0.4908 | train_acc: 0.8125 | test_loss: 0.5870 | test_acc: 0.8873
[INFO] Saving model to: models/07_effnetb2_data_10_percent_10_epochs.pth
-----
```

```
[INFO] Experiment number: 5
[INFO] Model: effnetb0
[INFO] DataLoader: data_20_percent
[INFO] Number of epochs: 5
[INFO] Created new effnetb0 model.
[INFO] Created SummaryWriter, saving to: runs/2022-06-23/data_20_percent/effnetb0/5_epochs...
 0% | 0/5 [00:00<?, ?it/s]
Epoch: 1 | train_loss: 0.9577 | train_acc: 0.6167 | test_loss: 0.6545 | test_acc: 0.8655
Epoch: 2 | train_loss: 0.6881 | train_acc: 0.8438 | test_loss: 0.5798 | test_acc: 0.9176
Epoch: 3 | train_loss: 0.5798 | train_acc: 0.8604 | test_loss: 0.4575 | test_acc: 0.9176
Epoch: 4 | train_loss: 0.4930 | train_acc: 0.8646 | test_loss: 0.4458 | test_acc: 0.9176
Epoch: 5 | train_loss: 0.4886 | train_acc: 0.8500 | test_loss: 0.3909 | test_acc: 0.9176
[INFO] Saving model to: models/07_effnetb0_data_20_percent_5_epochs.pth
-----
```

```
[INFO] Experiment number: 6
[INFO] Model: effnetb2
```

```
[INFO] DataLoader: data_20_percent
[INFO] Number of epochs: 5
[INFO] Created new effnetb2 model.
[INFO] Created SummaryWriter, saving to: runs/2022-06-23/data_20_percent/effnetb2/5_epochs...
 0%|          | 0/5 [00:00<?, ?it/s]
Epoch: 1 | train_loss: 0.9830 | train_acc: 0.5521 | test_loss: 0.7767 | test_acc: 0.8153
Epoch: 2 | train_loss: 0.7298 | train_acc: 0.7604 | test_loss: 0.6673 | test_acc: 0.8873
Epoch: 3 | train_loss: 0.6022 | train_acc: 0.8458 | test_loss: 0.5622 | test_acc: 0.9280
Epoch: 4 | train_loss: 0.5435 | train_acc: 0.8354 | test_loss: 0.5679 | test_acc: 0.9186
Epoch: 5 | train_loss: 0.4404 | train_acc: 0.9042 | test_loss: 0.4462 | test_acc: 0.9489
[INFO] Saving model to: models/07_effnetb2_data_20_percent_5_epochs.pth
-----
```

```
[INFO] Experiment number: 7
[INFO] Model: effnetb0
[INFO] DataLoader: data_20_percent
[INFO] Number of epochs: 10
[INFO] Created new effnetb0 model.
[INFO] Created SummaryWriter, saving to: runs/2022-06-23/data_20_percent/effnetb0/10_epochs...
 0%|          | 0/10 [00:00<?, ?it/s]
Epoch: 1 | train_loss: 0.9577 | train_acc: 0.6167 | test_loss: 0.6545 | test_acc: 0.8655
Epoch: 2 | train_loss: 0.6881 | train_acc: 0.8438 | test_loss: 0.5798 | test_acc: 0.9176
Epoch: 3 | train_loss: 0.5798 | train_acc: 0.8604 | test_loss: 0.4575 | test_acc: 0.9176
Epoch: 4 | train_loss: 0.4930 | train_acc: 0.8646 | test_loss: 0.4458 | test_acc: 0.9176
Epoch: 5 | train_loss: 0.4886 | train_acc: 0.8500 | test_loss: 0.3909 | test_acc: 0.9176
Epoch: 6 | train_loss: 0.3705 | train_acc: 0.8854 | test_loss: 0.3568 | test_acc: 0.9072
Epoch: 7 | train_loss: 0.3551 | train_acc: 0.9250 | test_loss: 0.3187 | test_acc: 0.9072
Epoch: 8 | train_loss: 0.3745 | train_acc: 0.8938 | test_loss: 0.3349 | test_acc: 0.8873
Epoch: 9 | train_loss: 0.2972 | train_acc: 0.9396 | test_loss: 0.3092 | test_acc: 0.9280
Epoch: 10 | train_loss: 0.3620 | train_acc: 0.8479 | test_loss: 0.2780 | test_acc: 0.9072
[INFO] Saving model to: models/07_effnetb0_data_20_percent_10_epochs.pth
-----
```

```
[INFO] Experiment number: 8
[INFO] Model: effnetb2
[INFO] DataLoader: data_20_percent
[INFO] Number of epochs: 10
[INFO] Created new effnetb2 model.
[INFO] Created SummaryWriter, saving to: runs/2022-06-23/data_20_percent/effnetb2/10_epochs...
 0%|          | 0/10 [00:00<?, ?it/s]
Epoch: 1 | train_loss: 0.9830 | train_acc: 0.5521 | test_loss: 0.7767 | test_acc: 0.8153
Epoch: 2 | train_loss: 0.7298 | train_acc: 0.7604 | test_loss: 0.6673 | test_acc: 0.8873
Epoch: 3 | train_loss: 0.6022 | train_acc: 0.8458 | test_loss: 0.5622 | test_acc: 0.9280
Epoch: 4 | train_loss: 0.5435 | train_acc: 0.8354 | test_loss: 0.5679 | test_acc: 0.9186
Epoch: 5 | train_loss: 0.4404 | train_acc: 0.9042 | test_loss: 0.4462 | test_acc: 0.9489
Epoch: 6 | train_loss: 0.3889 | train_acc: 0.9104 | test_loss: 0.4555 | test_acc: 0.8977
Epoch: 7 | train_loss: 0.3483 | train_acc: 0.9271 | test_loss: 0.4227 | test_acc: 0.9384
Epoch: 8 | train_loss: 0.3862 | train_acc: 0.8771 | test_loss: 0.4344 | test_acc: 0.9280
Epoch: 9 | train_loss: 0.3308 | train_acc: 0.8979 | test_loss: 0.4242 | test_acc: 0.9384
Epoch: 10 | train_loss: 0.3383 | train_acc: 0.8896 | test_loss: 0.3906 | test_acc: 0.9384
[INFO] Saving model to: models/07_effnetb2_data_20_percent_10_epochs.pth
-----
```

CPU times: user 29.5 s, sys: 1min 28s, total: 1min 58s

Wall time: 2min 33s

## 8. View experiments in TensorBoard

Ho, ho!

Look at us go!

Training eight models in one go?

Now that's living up to the motto!

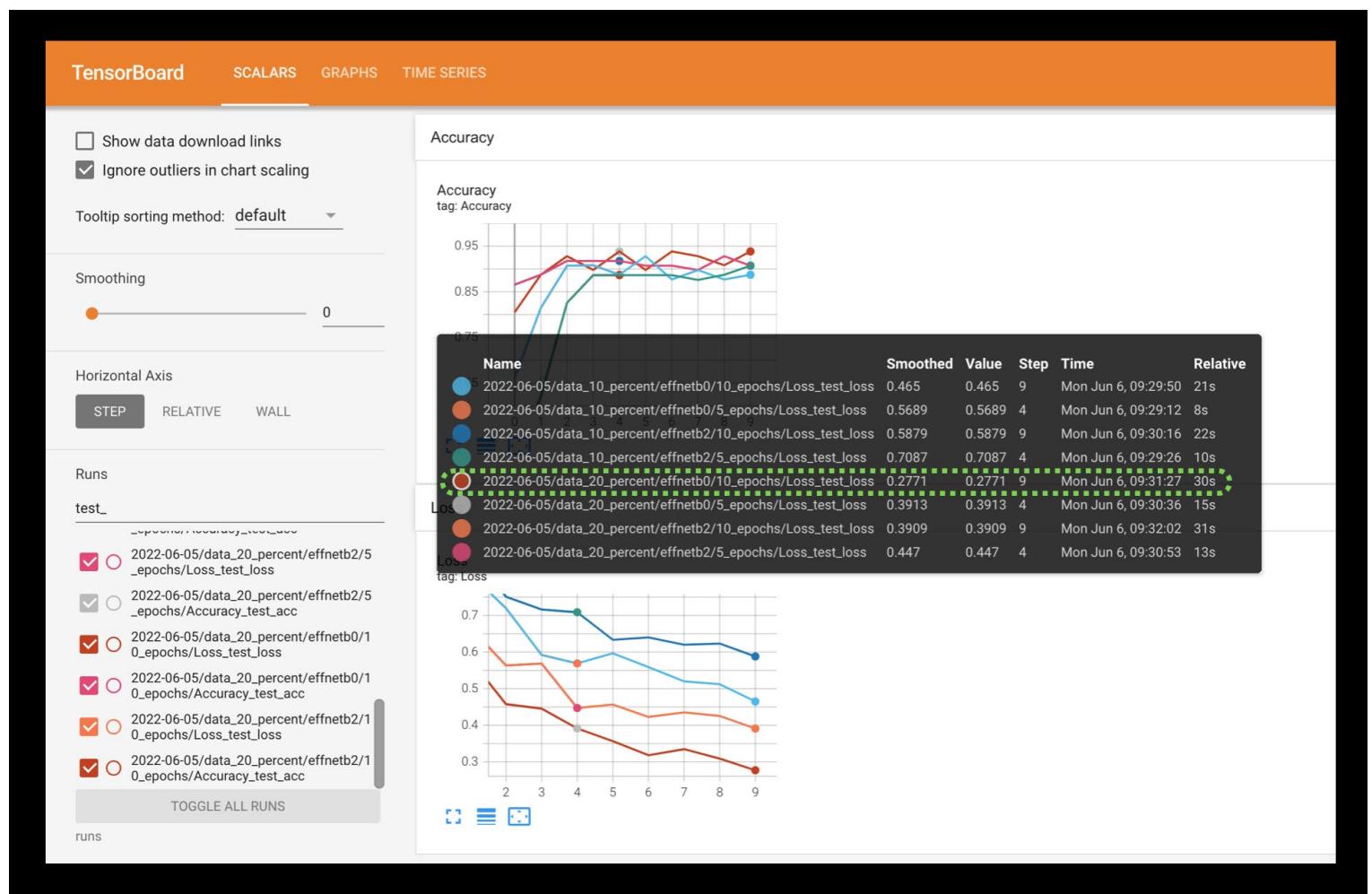
*Experiment, experiment, experiment!*

How about we check out the results in TensorBoard?

In [30]:

Running the cell above we should get an output similar to the following.

**Note:** Depending on the random seeds you used/hardware you used there's a chance your numbers aren't exactly the same as what's here. This is okay. It's due to the inherent randomness of deep learning. What matters most is the trend. Where your numbers are heading. If they're off by a large amount, perhaps there's something wrong and best to go back and check the code. But if they're off by a small amount (say a couple of decimal places or so), that's okay.



*Visualizing the test loss values for the different modelling experiments in TensorBoard, you can see that the EffNetB2 model trained for 10 epochs and with 20% of the data achieves the lowest loss. This sticks with the overall trend of the experiments that: more data, larger model and longer training time is generally better.*

You can also upload your TensorBoard experiment results to [tensorboard.dev](#) to host them publically for free.

For example, running code similiar to the following:

In [31]:



Running the cell above results in the experiments from this notebook being publically viewable at:

<https://tensorboard.dev/experiment/VySxUYY7Rje0xREYvCvZXA/>

**Note:** Beware that anything you upload to tensorboard.dev is publically available for anyone to see. So if you do upload your experiments, be careful they don't contain sensitive information.

## 9. Load in the best model and make predictions with it

Looking at the TensorBoard logs for our eight experiments, it seems experiment number eight achieved the best overall results (highest test accuracy, second lowest test loss).

This is the experiment that used:

- EffNetB2 (double the parameters of EffNetB0)
- 20% pizza, steak, sushi training data (double the original training data)
- 10 epochs (double the original training time)

In essence, our biggest model achieved the best results.

Though it wasn't as if these results were far better than the other models.

The same model on the same data achieved similar results in half the training time (experiment number 6).

This suggests that potentially the most influential parts of our experiments were the number of parameters and the amount of data.

Inspecting the results further it seems that generally a model with more parameters (EffNetB2) and more data (20% pizza, steak, sushi training data) performs better (lower test loss and higher test accuracy).

More experiments could be done to further test this but for now, let's import our best performing model from experiment eight (saved to: `models/07_effnetb2_data_20_percent_10_epochs.pth`, you can [download this model from the course GitHub](#)) and perform some qualitative evaluations.

In other words, let's *visualize, visualize, visualize!*

We can import the best saved model by creating a new instance of EffNetB2 using the `create_effnetb2()` function

and then load in the saved `state_dict()` with `torch.load()`.

In [32]:

```
[INFO] Created new effnetb2 model.
```

Out[32]:

```
<All keys matched successfully>
```

Best model loaded!

While we're here, let's check its filesize.

This is an important consideration later on when deploying the model (incorporating it in an app).

If the model is too large, it can be hard to deploy.

In [33]:

```
EfficientNetB2 feature extractor model size: 29 MB
```

Looks like our best model so far is 29 MB in size. We'll keep this in mind if we wanted to deploy it later on.

Time to make and visualize some predictions.

We created a `pred_and_plot_image()` function use a trained model to make predictions on an image in [06. PyTorch Transfer Learning section 6](#).

And we can reuse this function by importing it from `going_modular.going_modular.predictions.py` (I put the `pred_and_plot_image()` function in a script so we could reuse it).

So to make predictions on various images the model hasn't seen before, we'll first get a list of all the image filepaths from the 20% pizza, steak, sushi testing dataset and then we'll randomly select a subset of these filepaths to pass to our `pred_and_plot_image()` function.

In [34]:











Pred: steak | Prob: 0.837



Pred: pizza | Prob: 0.801



Pred: steak | Prob: 0.935



Nice!

Running the cell above a few times we can see our model performs quite well and often has higher prediction probabilities than previous models we've built.

This suggests the model is more confident in the decisions it's making.

## 9.1 Predict on a custom image with the best model

Making predictions on the test dataset is cool but the real magic of machine learning is making predictions on custom images of your own.

So let's import the trusty [pizza dad image](#) (a photo of my dad in front of a pizza) we've been using for the past couple of sections and see how our model performs on it.

In [35]:







data/04-pizza-dad.jpeg already exists, skipping download.

Pred: pizza | Prob: 0.978



Woah!

Two thumbs again!

Our best model predicts "pizza" correctly and this time with an even higher prediction probability (0.978) than the first feature extraction model we trained and used in [06. PyTorch Transfer Learning section 6.1](#).

This again suggests our current best model (EffNetB2 feature extractor trained on 20% of the pizza, steak, sushi training data and for 10 epochs) has learned patterns to make it more confident of its decision to predict pizza.

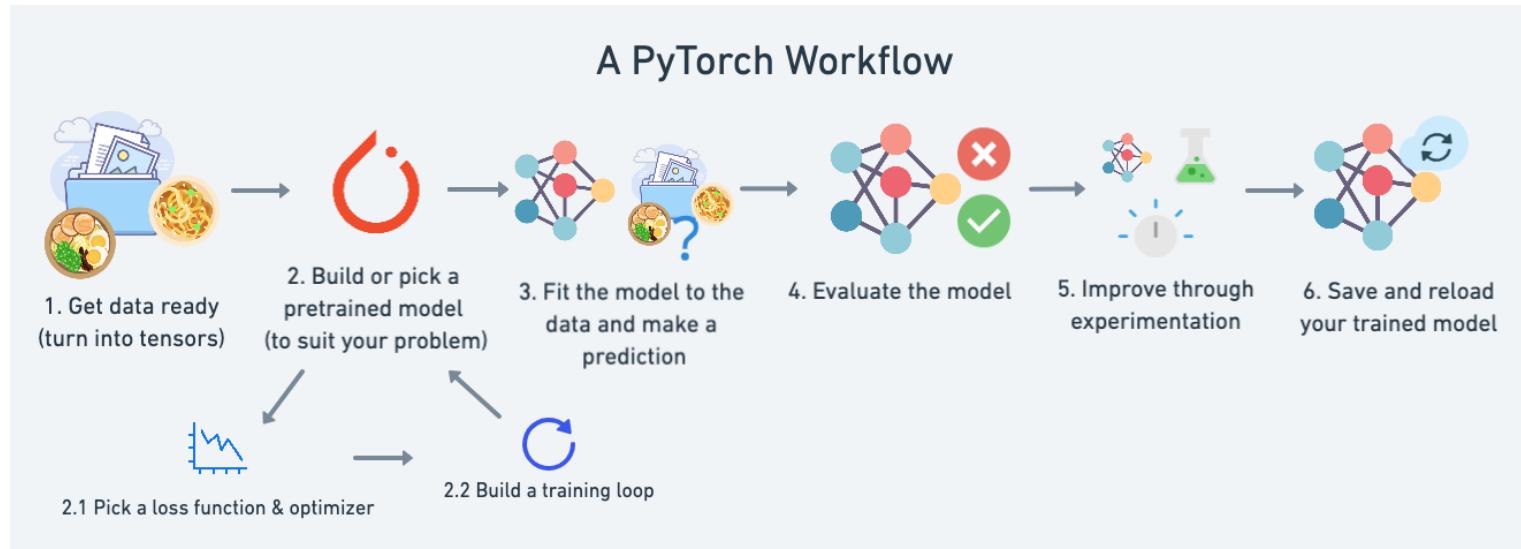
I wonder what could improve our model's performance even further?

I'll leave that as a challenge for you to investigate.

## Main takeaways

We've now gone full circle on the PyTorch workflow introduced in [01. PyTorch Workflow Fundamentals](#), we've gotten data ready, we've built and picked a pretrained model, we've used our various helper functions to train and evaluate the model and in this notebook we've improved our FoodVision Mini model by running and tracking a series

of experiments.



You should be proud of yourself, this is no small feat!

The main ideas you should take away from this Milestone Project 1 are:

- The machine learning practitioner's motto: *experiment, experiment, experiment!* (though we've been doing plenty of this already).
- In the beginning, keep your experiments small so you can work fast, your first few experiments shouldn't take more than a few seconds to a few minutes to run.
- The more experiments you do, the quicker you can figure out what *doesn't* work.
- Scale up when you find something that works. For example, since we've found a pretty good performing model with EffNetB2 as a feature extractor, perhaps you'd now like to see what happens when you scale it up to the whole [Food101 dataset](#) from `torchvision.datasets`.
- Programmatically tracking your experiments takes a few steps to set up but it's worth it in the long run so you can figure out what works and what doesn't.
  - There are many different machine learning experiment trackers out there so explore a few and try them out.

## Exercises

**Note:** These exercises expect the use of `torchvision v0.13+` (released July 2022), previous versions may work but will likely have errors.

All of the exercises are focused on practicing the code above.

You should be able to complete them by referencing each section or by following the resource(s) linked.

All exercises should be completed using [device-agnostic code](#).

## Resources:

- [Exercise template notebook for 07](#)
- [Example solutions notebook for 07](#) (try the exercises *before* looking at this)
  - See a live [video walkthrough of the solutions on YouTube](#) (errors and all)

1. Pick a larger model from `torchvision.models` to add to the list of experiments (for example, EffNetB3 or higher).
  - How does it perform compared to our existing models?
2. Introduce data augmentation to the list of experiments using the 20% pizza, steak, sushi training and test datasets, does this change anything?
  - For example, you could have one training DataLoader that uses data augmentation (e.g. `train_dataloader_20_percent_aug` and `train_dataloader_20_percent_no_aug`) and then compare the results of two of the same model types training on these two DataLoaders.
  - **Note:** You may need to alter the `create_dataloaders()` function to be able to take a transform for the training data and the testing data (because you don't need to perform data augmentation on the test data). See [04. PyTorch Custom Datasets section 6](#) for examples of using data augmentation or the script below for an example:

```
# Note: Data augmentation transform like this should only be performed on training data
train_transform_data_aug = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.TrivialAugmentWide(),
    transforms.ToTensor(),
    normalize
])

# Helper function to view images in a DataLoader (works with data augmentation transforms or not)
def view_dataloader_images(dataloader, n=10):
    if n > 10:
        print(f"Having n higher than 10 will create messy plots, lowering to 10.")
        n = 10
    imgs, labels = next(iter(dataloader))
    plt.figure(figsize=(16, 8))
    for i in range(n):
        # Min max scale the image for display purposes
        targ_image = imgs[i]
        sample_min, sample_max = targ_image.min(), targ_image.max()
        sample_scaled = (targ_image - sample_min)/(sample_max - sample_min)

        # Plot images with appropriate axes information
        plt.subplot(1, 10, i+1)
        plt.imshow(sample_scaled.permute(1, 2, 0)) # resize for Matplotlib requirements
        plt.title(class_names[labels[i]])
```

```

plt.axis(False)

# Have to update `create_dataloaders()` to handle different augmentations
import os
from torch.utils.data import DataLoader
from torchvision import datasets

NUM_WORKERS = os.cpu_count() # use maximum number of CPUs for workers to load data

# Note: this is an update version of data_setup.create_dataloaders to handle
# differnt train and test transforms.
def create_dataloaders(
    train_dir,
    test_dir,
    train_transform, # add parameter for train transform (transforms on train dataset)
    test_transform, # add parameter for test transform (transforms on test dataset)
    batch_size=32, num_workers=NUM_WORKERS
):
    # Use ImageFolder to create dataset(s)
    train_data = datasets.ImageFolder(train_dir, transform=train_transform)
    test_data = datasets.ImageFolder(test_dir, transform=test_transform)

    # Get class names
    class_names = train_data.classes

    # Turn images into data loaders
    train_dataloader = DataLoader(
        train_data,
        batch_size=batch_size,
        shuffle=True,
        num_workers=num_workers,
        pin_memory=True,
    )
    test_dataloader = DataLoader(
        test_data,
        batch_size=batch_size,
        shuffle=True,
        num_workers=num_workers,
        pin_memory=True,
    )

    return train_dataloader, test_dataloader, class_names

```

## 1. Scale up the dataset to turn FoodVision Mini into FoodVision Big using the entire [Food101](#) dataset from `torchvision.models`

- You could take the best performing model from your various experiments or even the EffNetB2 feature extractor we created in this notebook and see how it goes fitting for 5 epochs on all of Food101.
- If you try more than one model, it would be good to have the model's results tracked.
- If you load the Food101 dataset from `torchvision.models`, you'll have to create PyTorch DataLoaders to use it in training.

- **Note:** Due to the larger amount of data in Food101 compared to our pizza, steak, sushi dataset, this model will take longer to train.

## Extra-curriculum

- Read [The Bitter Lesson](#) blog post by Richard Sutton to get an idea of how many of the latest advancements in AI have come from increased scale (bigger datasets and bigger models) and more general (less meticulously crafted) methods.
- Go through the [PyTorch YouTube/code tutorial](#) for TensorBoard for 20-minutes and see how it compares to the code we've written in this notebook.
- Perhaps you may want to view and rearrange your model's TensorBoard logs with a DataFrame (so you can sort the results by lowest loss or highest accuracy), there's a guide for this [in the TensorBoard documentation](#).
- If you like to use VSCode for development using scripts or notebooks (VSCode can now use Jupyter Notebooks natively), you can setup TensorBoard right within VSCode using the [PyTorch Development in VSCode guide](#).
- To go further with experiment tracking and see how your PyTorch model is performing from a speed perspective (are there any bottlenecks that could be improved to speed up training?), see the [PyTorch documentation for the PyTorch profiler](#).
- Made With ML is an outstanding resource for all things machine learning by Goku Mohandas and their [guide on experiment tracking](#) contains a fantastic introduction to tracking machine learning experiments with MLflow.

Previous

06. PyTorch Transfer Learning

Next

08. PyTorch Paper Replicating



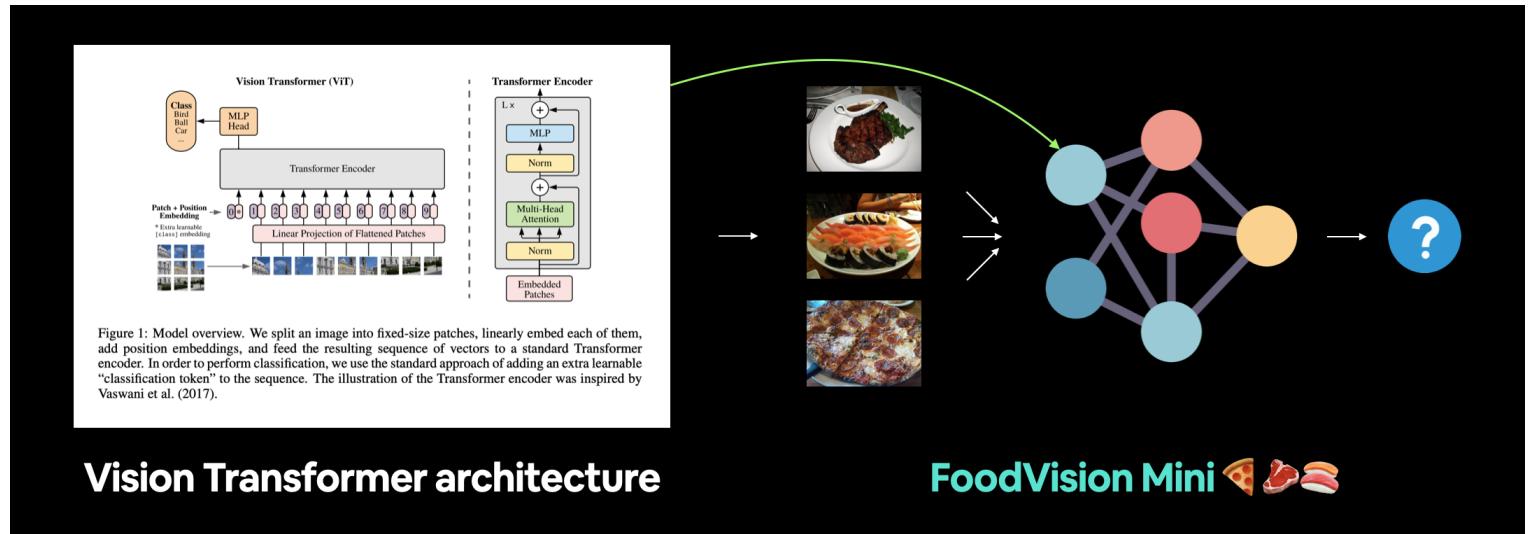
[View Source Code](#) | [View Slides](#)

## 08. PyTorch Paper Replicating

Welcome to Milestone Project 2: PyTorch Paper Replicating!

In this project, we're going to be **replicating a machine learning research paper** and creating a Vision Transformer (ViT) from scratch using PyTorch.

We'll then see how ViT, a state-of-the-art computer vision architecture, performs on our FoodVision Mini problem.



For Milestone Project 2 we're going to focus on recreating the Vision Transformer (ViT) computer vision architecture and applying it to our FoodVision Mini problem to classify different images of pizza, steak and sushi.

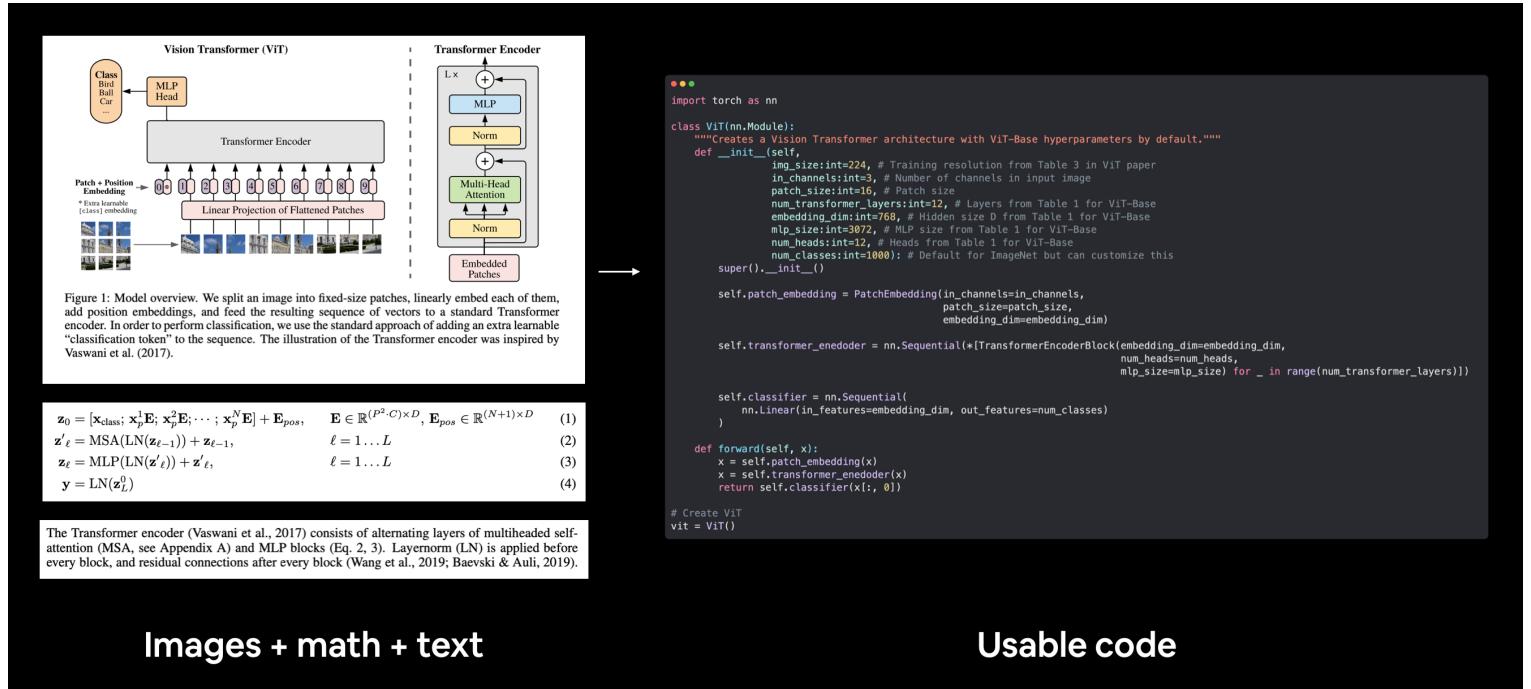
What is paper replicating?

It's no secret machine learning is advancing fast.

Many of these advances get published in machine learning research papers.

And the goal of **paper replicating** is to take replicate these advances with code so you can use the techniques for your own problem.

For example, let's say a new model architecture gets released that performs better than any other architecture before on various benchmarks, wouldn't it be nice to try that architecture on your own problems?



*Machine learning paper replicating involves turning a machine learning paper comprised of images/diagrams, math and text into usable code and in our case, usable PyTorch code. Diagram, math equations and text from the [ViT paper](#).*

## What is a machine learning research paper?

A machine learning research paper is a scientific paper that details findings of a research group on a specific area.

The contents of a machine learning research paper can vary from paper to paper but they generally follow the structure:

Section	Contents
<b>Abstract</b>	An overview/summary of the paper's main findings/contributions.
<b>Introduction</b>	What's the paper's main problem and details of previous methods used to try and solve it.

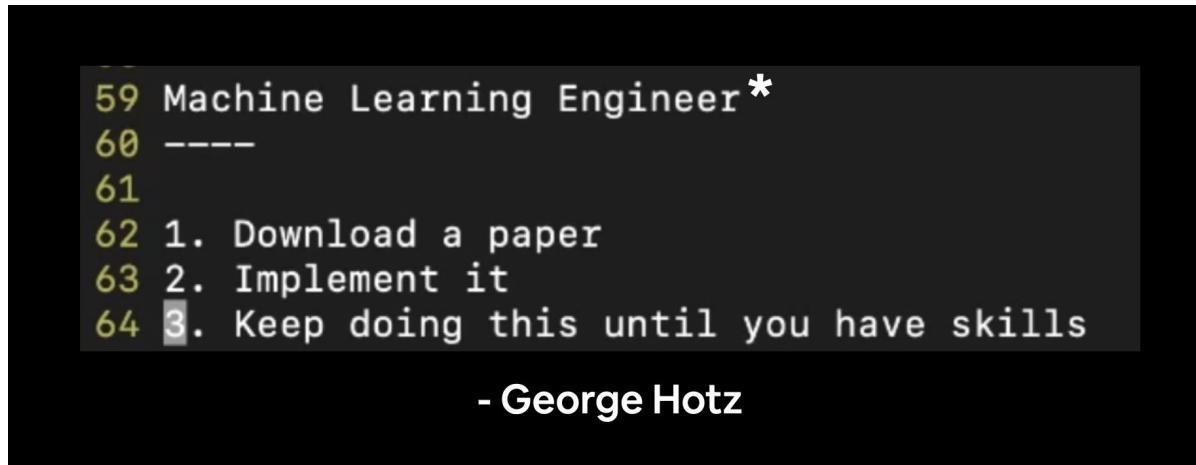
<b>Method</b>	How did the researchers go about conducting their research? For example, what model(s), data sources, training setups were used?
<b>Results</b>	What are the outcomes of the paper? If a new type of model or training setup was used, how did the results of findings compare to previous works? (this is where <b>experiment tracking</b> comes in handy)
<b>Conclusion</b>	What are the limitations of the suggested methods? What are some next steps for the research community?
<b>References</b>	What resources/other papers did the researchers look at to build their own body of work?
<b>Appendix</b>	Are there any extra resources/findings to look at that weren't included in any of the above sections?

## Why replicate a machine learning research paper?

A machine learning research paper is often a presentation of months of work and experiments done by some of the best machine learning teams in the world condensed into a few pages of text.

And if these experiments lead to better results in an area related to the problem you're working on, it'd be nice to them out.

Also, replicating the work of others is a fantastic way to practice your skills.



George Hotz is founder of [comma.ai](https://comma.ai), a self-driving car company and livestreams machine learning coding on [Twitch](https://twitch.tv) and those videos get posted in full to [YouTube](https://youtube.com). I pulled this quote from one of his livestreams. The "\*" is to note that machine learning engineering often involves the extra step(s) of preprocessing data and making your models available for others to use (deployment).

When you first start trying to replicate research papers, you'll likely be overwhelmed.

That's normal.

Research teams spend weeks, months and sometimes years creating these works so it makes sense if it takes you sometime to even read let alone reproduce the works.

Replicating research is such a tough problem, phenomenal machine learning libraries and tools such as, [HuggingFace](#), [PyTorch Image Models](#) (`timm` library) and [fast.ai](#) have been born out of making machine learning research more accessible.

## Where can you find code examples for machine learning research papers?

One of the first things you'll notice when it comes to machine learning research is: there's a lot of it.

So beware, trying to stay on top of it is like trying to outrun a hamster wheel.

Follow your interest, pick a few things that stand out to you.

In saying this, there are several places to find and read machine learning research papers (and code):

Resource	What is it?
<a href="#">arXiv</a>	Pronounced "archive", arXiv is a free and open resource for reading technical articles on everything from physics to computer science (including machine learning).
<a href="#">AK Twitter</a>	The AK Twitter account publishes machine learning research highlights, often with live demos almost every day. I don't understand 9/10 posts but I find it fun to explore every so often.
<a href="#">Papers with Code</a>	A curated collection of trending, active and greatest machine learning papers, many of which include code resources attached. Also includes a collection of common machine learning datasets, benchmarks and current state-of-the-art models.
<a href="#">lucidrains' vit-pytorch GitHub repository</a>	Less of a place to find research papers and more of an example of what paper replicating with code on a larger-scale and with a specific focus looks like. The <code>vit-pytorch</code> repository is a collection of Vision Transformer model architectures from various research papers replicated with PyTorch code (much of the inspiration for this notebook was gathered from this repository).

**Note:** This list is far from exhaustive. I only list a few places, the ones I use most frequently personally. So beware the bias. However, I've noticed that even this short list often fully satisfies my needs for knowing what's going on in the field. Anymore and I might go crazy.

# What we're going to cover

Rather than talk about replicating a paper, we're going to get hands-on and *actually* replicate a paper.

The process for replicating all papers will be slightly different but by seeing what it's like to do one, we'll get the momentum to do more.

More specifically, we're going to be replicating the machine learning research paper *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale* (ViT paper) with PyTorch.

The Transformer neural network architecture was originally introduced in the machine learning research paper *Attention is all you need*.

And the original Transformer architecture was designed to work on one-dimensional (1D) sequences of text.

A **Transformer architecture** is generally considered to be any neural network that uses the **attention mechanism**) as its primary learning layer. Similar to how a convolutional neural network (CNN) uses convolutions as its primary learning layer.

Like the name suggests, **the Vision Transformer (ViT) architecture was designed to adapt the original Transformer architecture to vision problem(s)** (classification being the first and since many others have followed).

The original Vision Transformer has been through several iterations over the past couple of years, however, we're going to focus on replicating the original, otherwise known as the "vanilla Vision Transformer". Because if you can recreate the original, you can adapt to the others.

We're going to be focusing on building the ViT architecture as per the original ViT paper and applying it to FoodVision Mini.

Topic	Contents
<b>0. Getting setup</b>	We've written a fair bit of useful code over the past few sections, let's download it and make sure we can use it again.
<b>1. Get data</b>	Let's get the pizza, steak and sushi image classification dataset we've been using and build a Vision Transformer to try and improve FoodVision Mini model's results.
<b>2. Create Datasets and DataLoaders</b>	We'll use the <code>data_setup.py</code> script we wrote in chapter 05. PyTorch Going Modular to setup our DataLoaders.
<b>3. Replicating the ViT paper: an overview</b>	Replicating a machine learning research paper can be a fair challenge, so before we jump in, let's break the ViT paper down into smaller chunks, so we can replicate the paper chunk by chunk.

**4. Equation 1: The Patch Embedding**

The ViT architecture is comprised of four main equations, the first being the patch and position embedding. Or turning an image into a sequence of learnable patches.

**5. Equation 2: Multi-Head Attention (MSA)**

The self-attention/multi-head self-attention (MSA) mechanism is at the heart of every Transformer architecture, including the ViT architecture, let's create an MSA block using PyTorch's in-built layers.

**6. Equation 3: Multilayer Perceptron (MLP)**

The ViT architecture uses a multilayer perceptron as part of its Transformer Encoder and for its output layer. Let's start by creating an MLP for the Transformer Encoder.

**7. Creating the Transformer Encoder**

A Transformer Encoder is typically comprised of alternating layers of MSA (equation 2) and MLP (equation 3) joined together via residual connections. Let's create one by stacking the layers we created in sections 5 & 6 on top of each other.

**8. Putting it all together to create ViT**

We've got all the pieces of the puzzle to create the ViT architecture, let's put them all together into a single class we can call as our model.

**9. Setting up training code for our ViT model**

Training our custom ViT implementation is similar to all of the other model's we've trained previously. And thanks to our `train()` function in `engine.py` we can start training with a few lines of code.

**10. Using a pretrained ViT from `torchvision.models`**

Training a large model like ViT usually takes a fair amount of data. Since we're only working with a small amount of pizza, steak and sushi images, let's see if we can leverage the power of transfer learning to improve our performance.

**11. Make predictions on a custom image**

The magic of machine learning is seeing it work on your own data, so let's take our best performing model and put FoodVision Mini to the test on the infamous *pizza-dad* image (a photo of my dad eating pizza).

**Note:** Despite the fact we're going to be focused on replicating the ViT paper, avoid getting too bogged down on a particular paper as newer better methods will often come along, quickly, so the skill should be to remain curious whilst building the fundamental skills of turning math and words on a page into working code.

## Terminology

There are going to be a fair few acronyms throughout this notebook.

In light of this, here are some definitions:

- **ViT** - Stands for Vision Transformer (the main neural network architecture we're going to be focused on replicating).

- **ViT paper** - Short hand for the original machine learning research paper that introduced the ViT architecture, [\*An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale\*](#), anytime *ViT paper* is mentioned, you can be assured it is referencing this paper.

## Where can you get help?

All of the materials for this course [are available on GitHub](#).

If you run into trouble, you can ask a question on the course [GitHub Discussions page](#).

And of course, there's the [PyTorch documentation](#) and [PyTorch developer forums](#), a very helpful place for all things PyTorch.

## 0. Getting setup

As we've done previously, let's make sure we've got all of the modules we'll need for this section.

We'll import the Python scripts (such as `data_setup.py` and `engine.py`) we created in [05. PyTorch Going Modular](#).

To do so, we'll download `going_modular` directory from the `pytorch-deep-learning` repository (if we don't already have it).

We'll also get the `torchinfo` package if it's not available.

`torchinfo` will help later on to give us a visual representation of our model.

And since later on we'll be using `torchvision` v0.13 package (available as of July 2022), we'll make sure we've got the latest versions.

In [1]:







```
torch version: 1.12.0+cu102
torchvision version: 0.13.0+cu102
```

**Note:** If you're using Google Colab and the cell above starts to install various software packages, you may have to restart your runtime after running the above cell. After restarting, you can run the cell again and verify you've got the right versions of `torch` and `torchvision`.

Now we'll continue with the regular imports, setting up device agnostic code and this time we'll also get the `helper_functions.py` script from GitHub.

The `helper_functions.py` script contains several functions we created in previous sections:

- `set_seeds()` to set the random seeds (created in [07. PyTorch Experiment Tracking section 0](#)).
- `download_data()` to download a data source given a link (created in [07. PyTorch Experiment Tracking section 1](#)).
- `plot_loss_curves()` to inspect our model's training results (created in [04. PyTorch Custom Datasets section 7.8](#))

**Note:** It may be a better idea for many of the functions in the `helper_functions.py` script to be merged into `going_modular/going_modular/utils.py`, perhaps that's an extension you'd like to try.

In [2]:











**Note:** If you're using Google Colab, and you don't have a GPU turned on yet, it's now time to turn one on via

Runtime -> Change runtime type -> Hardware accelerator -> GPU .

In [3]:

Out[3]:

'cuda'

## 1. Get Data

Since we're continuing on with FoodVision Mini, let's download the pizza, steak and sushi image dataset we've been using.

To do so we can use the `download_data()` function from `helper_functions.py` that we created in [07. PyTorch Experiment Tracking section 1](#).

We'll `source` to the raw GitHub link of the `pizza_steak_sushi.zip` `data` and the `destination` to `pizza_steak_sushi`.

In [4]:

```
[INFO] data/pizza_steak_sushi directory exists, skipping download.  
PosixPath('data/pizza_steak_sushi')
```

Beautiful! Data downloaded, let's setup the training and test directories.

Out[4]:

In [5]:

## 2. Create Datasets and DataLoaders

Now we've got some data, let's now turn it into `DataLoader`'s.

To do so we can use the `create_dataloaders()` function in `data_setup.py`.

First, we'll create a transform to prepare our images.

This where one of the first references to the ViT paper will come in.

In Table 3, the training resolution is mentioned as being 224 (height=224, width=224).

Models	Dataset	Epochs	Base LR	LR decay	Weight decay	Dropout
ViT-B/{16,32}	JFT-300M	7	$8 \cdot 10^{-4}$	linear	0.1	0.0
ViT-L/32	JFT-300M	7	$6 \cdot 10^{-4}$	linear	0.1	0.0
ViT-L/16	JFT-300M	7/14	$4 \cdot 10^{-4}$	linear	0.1	0.0
ViT-H/14	JFT-300M	14	$3 \cdot 10^{-4}$	linear	0.1	0.0
R50x{1,2}	JFT-300M	7	$10^{-3}$	linear	0.1	0.0
R101x1	JFT-300M	7	$8 \cdot 10^{-4}$	linear	0.1	0.0
R152x{1,2}	JFT-300M	7	$6 \cdot 10^{-4}$	linear	0.1	0.0
R50+ViT-B/{16,32}	JFT-300M	7	$8 \cdot 10^{-4}$	linear	0.1	0.0
R50+ViT-L/32	JFT-300M	7	$2 \cdot 10^{-4}$	linear	0.1	0.0
R50+ViT-L/16	JFT-300M	7/14	$4 \cdot 10^{-4}$	linear	0.1	0.0
ViT-B/{16,32}	ImageNet-21k	90	$10^{-3}$	linear	0.03	0.1
ViT-L/{16,32}	ImageNet-21k	30/90	$10^{-3}$	linear	0.03	0.1
ViT-*	ImageNet	300	$3 \cdot 10^{-3}$	cosine	0.3	0.1

Table 3: Hyperparameters for training. All models are trained with a batch size of 4096 and learning rate warmup of 10k steps. For ImageNet we found it beneficial to additionally apply gradient clipping at global norm 1. Training resolution is 224.

**Batch size = 4096**

Source: [ViT paper](#)

**Image size = 224x224 (height=224, width=224)**

You can often find various hyperparameter settings listed in a table. In this case we're still preparing our data, so we're mainly concerned with things like image size and batch size. Source: [Table 3 in ViT paper](#).

So we'll make sure our transform resizes our images appropriately.

And since we'll be training our model from scratch (no transfer learning to begin with), we won't provide a normalize transform like we did in [06. PyTorch Transfer Learning section 2.1](#).

## 2.1 Prepare transforms for images

In [6]:



```

Manually created transforms: Compose(
    Resize(size=(224, 224), interpolation=bilinear, max_size=None, antialias=None)
    ToTensor()
)

```

## 2.2 Turn images into `DataLoader`'s

Transforms created!

Let's now create our `DataLoader`'s.

The ViT paper states the use of a batch size of 4096 which is 128x the size of the batch size we've been using (32).

However, we're going to stick with a batch size of 32.

Why?

Because some hardware (including the free tier of Google Colab) may not be able to handle a batch size of 4096.

Having a batch size of 4096 means that 4096 images need to fit into the GPU memory at a time.

This works when you've got the hardware to handle it like a research team from Google often does but when you're running on a single GPU (such as using Google Colab), making sure things work with smaller batch size first is a good idea.

An extension of this project could be to try a higher batch size value and see what happens.

**Note:** We're using the `pin_memory=True` parameter in the `create_dataloaders()` function to speed up computation. `pin_memory=True` avoids unnecessary copying of memory between the CPU and GPU memory by "pinning" examples that have been seen before. Though the benefits of this will likely be seen with larger dataset sizes (our FoodVision Mini dataset is quite small). However, setting `pin_memory=True` doesn't *always* improve performance (this is another one of those we're scenarios in machine learning where some things work sometimes and don't other times), so best to *experiment, experiment, experiment*. See the PyTorch [torch.utils.data.DataLoader documentation](#) or [Making Deep Learning Go Brrrr from First Principles](#) by Horace He for more.

In [7]:



Out[7]:

```
(<torch.utils.data.DataLoader at 0x7f18845ff0d0>,
 <torch.utils.data.DataLoader at 0x7f17f3f5f520>,
 ['pizza', 'steak', 'sushi'])
```

## 2.3 Visualize a single image

Now we've loaded our data, let's *visualize, visualize, visualize!*

An important step in the ViT paper is preparing the images into patches.

We'll get to what this means in [section 4](#) but for now, let's view a single image and its label.

To do so, let's get a single image and label from a batch of data and inspect their shapes.

In [8]:

Out[8]:

```
(torch.Size([3, 224, 224]), tensor(2))
```

Wonderful!

Now let's plot the image and its label with `matplotlib`.

In [9]:



Nice!

Looks like our images are importing correctly, let's continue with the paper replication.

### 3. Replicating the ViT paper: an overview

Before we write anymore code, let's discuss what we're doing.

We'd like to replicate the ViT paper for our own problem, FoodVision Mini.

So our **model inputs** are: images of pizza, steak and sushi.

And our ideal **model outputs** are: predicted labels of pizza, steak or sushi.

No different to what we've been doing throughout the previous sections.

The question is: how do we go from our inputs to the desired outputs?

#### 3.1 Inputs and outputs, layers and blocks

ViT is a deep learning neural network architecture.

And any neural network architecture is generally comprised of **layers**.

And a collection of layers is often referred to as a **block**.

And stacking many blocks together is what gives us the whole architecture.

A **layer** takes an input (say an image tensor), performs some kind of function on it (for example what's in the layer's `forward()` method) and then returns an output.

So if a **single layer** takes an input and gives an output, then a collection of layers or a **block** also takes an input and gives an output.

Let's make this concrete:

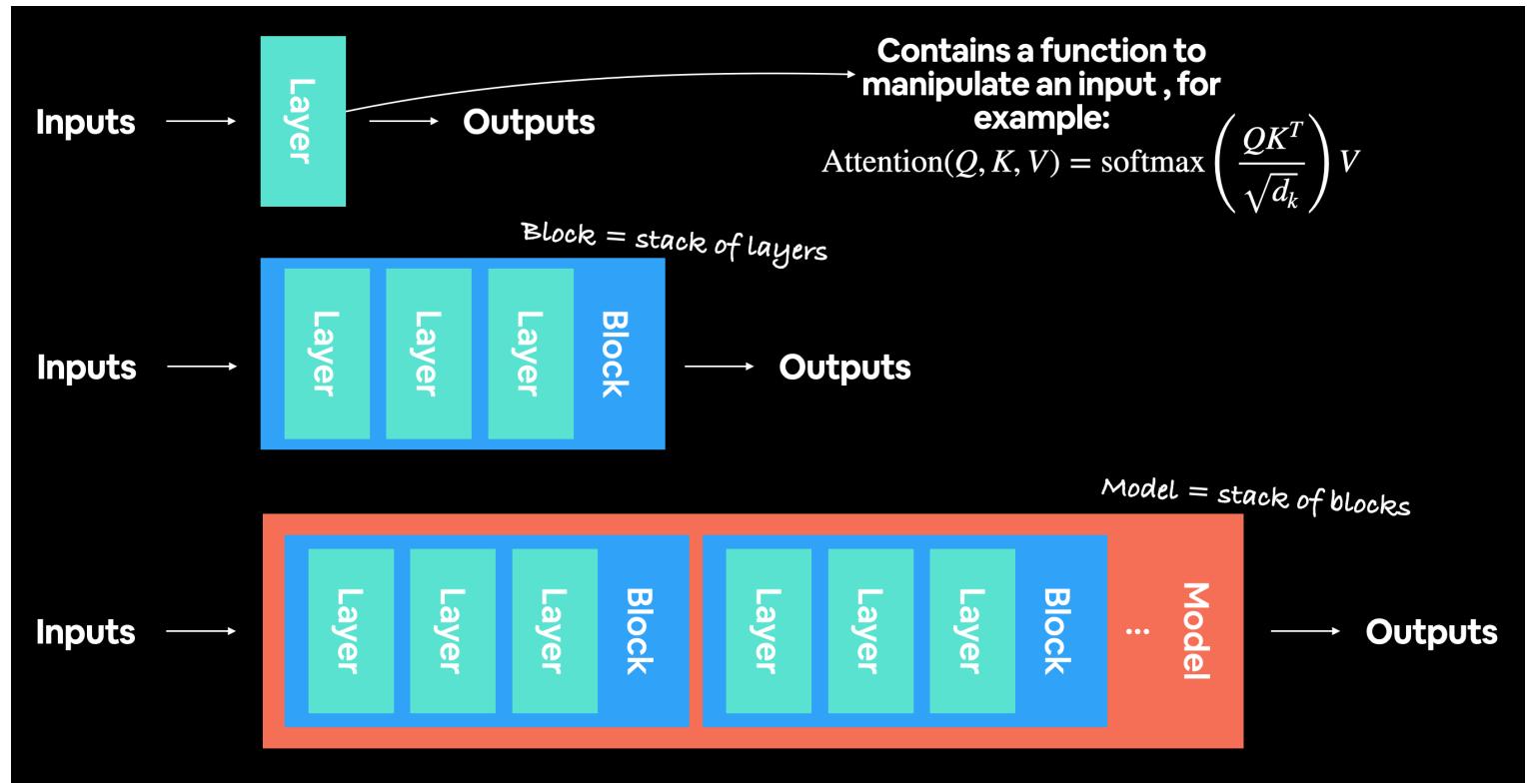
- **Layer** - takes an input, performs a function on it, returns an output.
- **Block** - a collection of layers, takes an input, performs a series of functions on it, returns an output.
- **Architecture (or model)** - a collection of blocks, takes an input, performs a series of functions on it, returns an output.

This ideology is what we're going to be using to replicate the ViT paper.

We're going to take it layer by layer, block by block, function by function putting the pieces of the puzzle together like Lego to get our desired overall architecture.

The reason we do this is because looking at a whole research paper can be intimidating.

So for a better understanding, we'll break it down, starting with the inputs and outputs of single layer and working up to the inputs and outputs of the whole model.



A modern deep learning architecture is usually collection of layers and blocks. Where layers take an input (data as a numerical representation) and manipulate it using some kind of function (for example, the self-attention formula pictured above, however, this function could be almost anything) and then output it. Blocks are generally stacks of layers on top of each other doing a similar thing to a single layer but multiple times.

### 3.2 Getting specific: What's ViT made of?

There are many little details about the ViT model sprinkled throughout the paper.

Finding them all is like one big treasure hunt!

Remember, a research paper is often months of work compressed into a few pages so it's understandable for it to take of practice to replicate.

However, the main three resources we'll be looking at for the architecture design are:

1. **Figure 1** - This gives an overview of the model in a graphical sense, you could *almost* recreate the architecture with this figure alone.
2. **Four equations in section 3.1** - These equations give a little bit more of a mathematical grounding to the

coloured blocks in Figure 1.

3. **Table 1** - This table shows the various hyperparameter settings (such as number of layers and number of hidden units) for different ViT model variants. We'll be focused on the smallest version, ViT-Base.

### 3.2.1 Exploring Figure 1

Let's start by going through Figure 1 of the ViT Paper.

The main things we'll be paying attention to are:

1. **Layers** - takes an **input**, performs an operation or function on the input, produces an **output**.
2. **Blocks** - a collection of layers, which in turn also takes an **input** and produces an **output**.

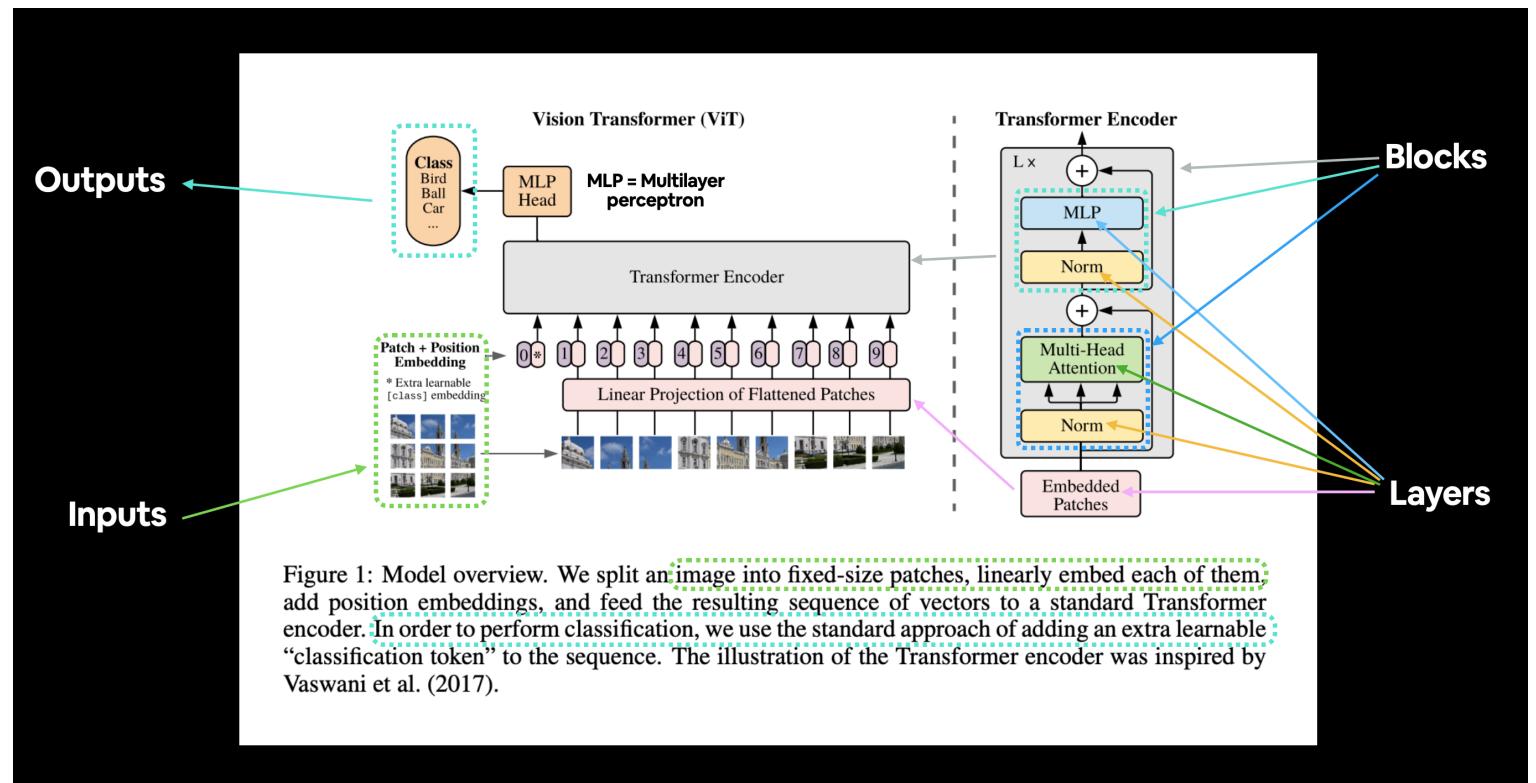


Figure 1 from the ViT Paper showcasing the different inputs, outputs, layers and blocks that create the architecture. Our goal will be to replicate each of these using PyTorch code.

The ViT architecture is comprised of several stages:

- **Patch + Position Embedding (inputs)** - Turns the input image into a sequence of image patches and add a position number what order the patch comes in.
- **Linear projection of flattened patches (Embedded Patches)** - The image patches get turned into an **embedding**, the benefit of using an embedding rather than just the image values is that an embedding is a *learnable* representation (typically in the form of a vector) of the image that can improve with training.

- **Norm** - This is short for "Layer Normalization" or "LayerNorm", a technique for regularizing (reducing overfitting) a neural network, you can use LayerNorm via the PyTorch layer `torch.nn.LayerNorm()`.
- **Multi-Head Attention** - This is a **Multi-Headed Self-Attention layer** or "MSA" for short. You can create an MSA layer via the PyTorch layer `torch.nn.MultiheadAttention()`.
- **MLP (or Multilayer perceptron)** - A MLP can often refer to any collection of feedforward layers (or in PyTorch's case, a collection of layers with a `forward()` method). In the ViT Paper, the authors refer to the MLP as "MLP block" and it contains two `torch.nn.Linear()` layers with a `torch.nn.GELU()` non-linearity activation in between them (section 3.1) and a `torch.nn.Dropout()` layer after each (Appendix B.1).
- **Transformer Encoder** - The Transformer Encoder, is a collection of the layers listed above. There are two skip connections inside the Transformer encoder (the "+" symbols) meaning the layer's inputs are fed directly to immediate layers as well as subsequent layers. The overall ViT architecture is comprised of a number of Transformer encoders stacked on top of eachother.
- **MLP Head** - This is the output layer of the architecture, it converts the learned features of an input to a class output. Since we're working on image classification, you could also call this the "classifier head". The structure of the MLP Head is similar to the MLP block.

You might notice that many of the pieces of the ViT architecture can be created with existing PyTorch layers.

This is because of how PyTorch is designed, it's one of the main purposes of PyTorch to create reusable neural network layers for both researchers and machine learning practitioners.

**Question:** Why not code everything from scratch?

You could definitely do that by reproducing all of the math equations from the paper with custom PyTorch layers and that would certainly be an educative exercise, however, using pre-existing PyTorch layers is usually favoured as pre-existing layers have often been extensively tested and performance checked to make sure they run correctly and fast.

**Note:** We're going to focused on write PyTorch code to create these layers, for the background on what each of these layers does, I'd suggest reading the ViT Paper in full or reading the linked resources for each layer.

Let's take Figure 1 and adapt it to our FoodVision Mini problem of classifying images of food into pizza, steak or sushi.

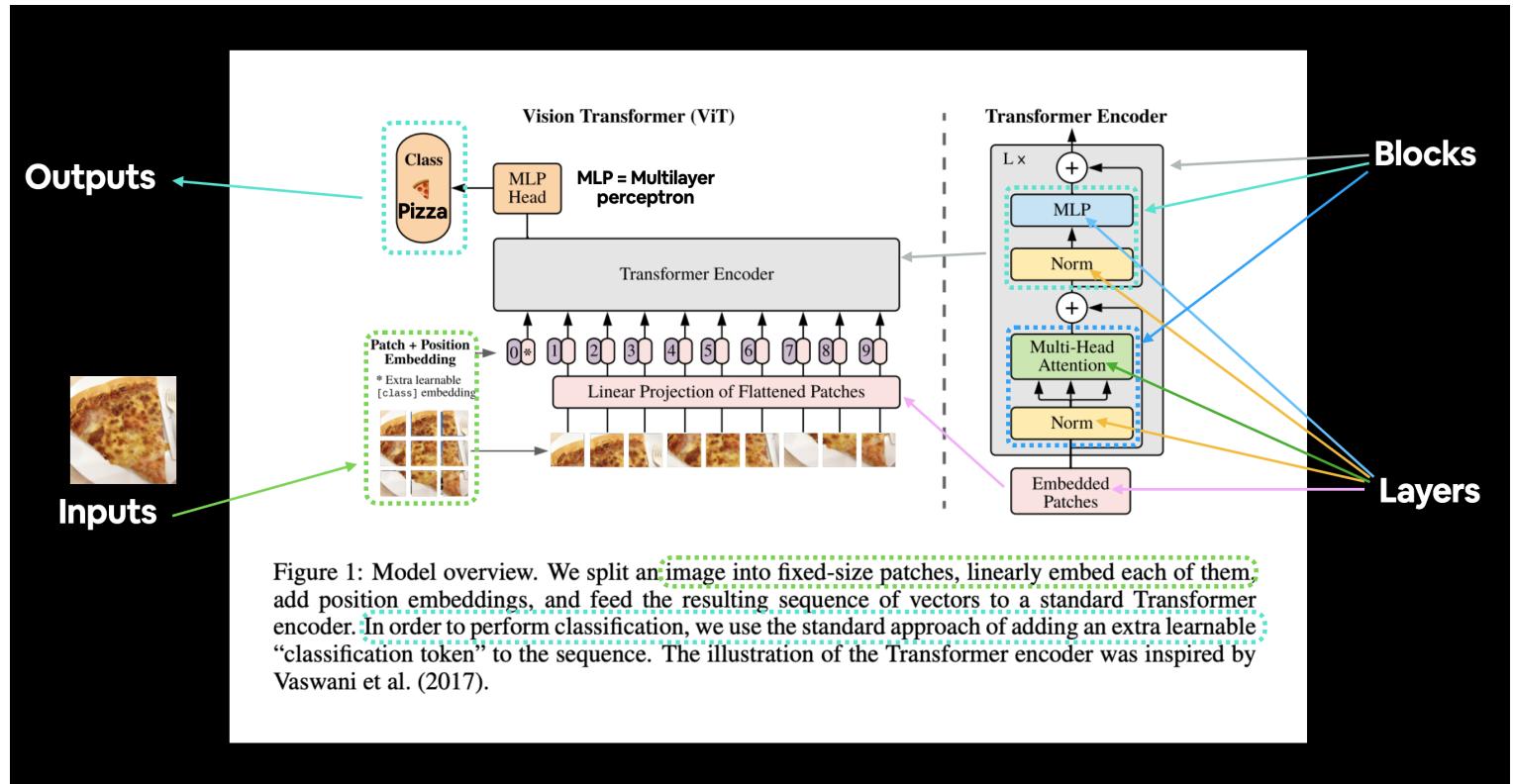


Figure 1 from the ViT Paper adapted for use with FoodVision Mini. An image of food goes in (pizza), the image gets turned into patches and then projected to an embedding. The embedding then travels through the various layers and blocks and (hopefully) the class "pizza" is returned.

### 3.2.2 Exploring the Four Equations

The next main part(s) of the ViT paper we're going to look at are the four equations in section 3.1.

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}}; \mathbf{x}_p^1 \mathbf{E}; \mathbf{x}_p^2 \mathbf{E}; \dots; \mathbf{x}_p^N \mathbf{E}] + \mathbf{E}_{pos}, \quad \mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D}, \mathbf{E}_{pos} \in \mathbb{R}^{(N+1) \times D} \quad (1)$$

$$\mathbf{z}'_\ell = \text{MSA}(\text{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \quad \ell = 1 \dots L \quad (2)$$

$$\mathbf{z}_\ell = \text{MLP}(\text{LN}(\mathbf{z}'_\ell)) + \mathbf{z}'_\ell, \quad \ell = 1 \dots L \quad (3)$$

$$\mathbf{y} = \text{LN}(\mathbf{z}_L^0) \quad (4)$$

These four equations represent the math behind the four major parts of the ViT architecture.

Section 3.1 describes each of these (some of the text has been omitted for brevity, bolded text is mine):

Equation number	Description from ViT paper section 3.1
-----------------	--

- 1 ...The Transformer uses constant latent vector size  $D$  through all of its layers, so we flatten the patches and map to  $D$  dimensions with a **trainable linear projection** (Eq. 1). We refer to the output of this projection as the **patch**

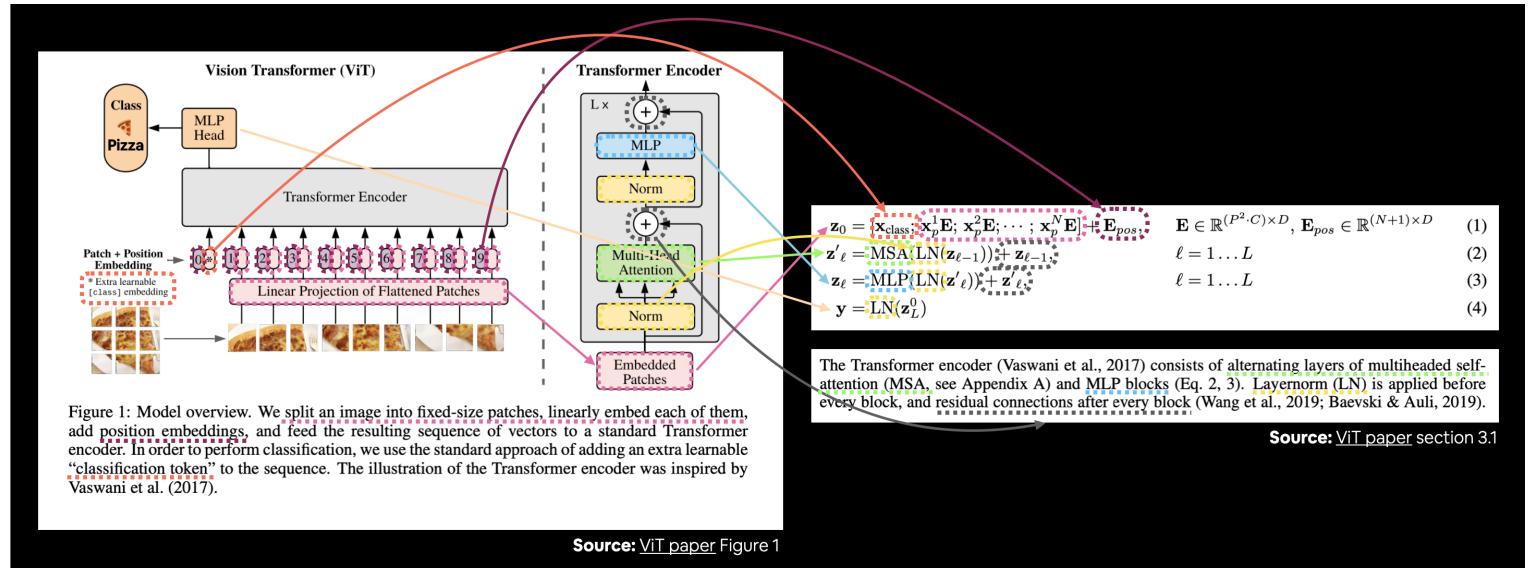
**embeddings... Position embeddings** are added to the patch embeddings to retain positional information. We use standard **learnable 1D position embeddings...**

2 The Transformer encoder (Vaswani et al., 2017) consists of alternating layers of multiheaded selfattention (MSA, see Appendix A) and MLP blocks (Eq. 2, 3). **Layernorm (LN)** is applied before every block, and **residual connections after every block** (Wang et al., 2019; Baevski & Auli, 2019).

3 Same as equation 2.

4 Similar to BERT's [ class ] token, we **prepend a learnable embedding to the sequence of embedded patches**  $\mathbf{z}_0 = \mathbf{x}_{\text{class}} + \mathbf{E}_0$ , whose state at the output of the Transformer encoder  $\mathbf{z}_L$  serves as the image representation  $\mathbf{y}$  (Eq. 4)...

Let's map these descriptions to the ViT architecture in Figure 1.



Connecting Figure 1 from the ViT paper to the four equations from section 3.1 describing the math behind each of the layers/blocks.

There's a lot happening in the image above but following the coloured lines and arrows reveals the main concepts of the ViT architecture.

How about we break down each equation further (it will be our goal to recreate these with code)?

In all equations (except equation 4), " $\mathbf{z}$ " is the raw output of a particular layer:

1.  $\mathbf{z}_0$  is "z zero" (this is the output of the initial patch embedding layer).
2.  $\mathbf{z}'_\ell$  is "z of a particular layer *prime*" (or an intermediary value of z).
3.  $\mathbf{z}_\ell$  is "z of a particular layer".

And  $\mathbf{y}$  is the overall output of the architecture.

### 3.2.3 Equation 1 overview

$$\begin{aligned} \mathbf{z}_0 &= \left[ \mathbf{x}_{\text{class}} ; \mathbf{x}_p^1 \mathbf{E} ; \mathbf{x}_p^2 \mathbf{E} ; \dots ; \mathbf{x}_p^N \mathbf{E} \right] + \mathbf{E}_{\text{pos}}, \quad \& \& \\ \mathbf{E} &\in \mathbb{R}^{(P^2) \times C \times D}, \mathbf{E}_{\text{pos}} \in \mathbb{R}^{(N+1) \times D} \end{aligned}$$

This equation deals with the class token, patch embedding and position embedding ( $\mathbf{E}$  is for embedding) of the input image.

In vector form, the embedding might look something like:

```
x_input = [class_token, image_patch_1, image_patch_2, image_patch_3...] + [class_token_position,
image_patch_1_position, image_patch_2_position, image_patch_3_position...]
```

Where each of the elements in the vector is learnable (their `requires_grad=True`).

### 3.2.4 Equation 2 overview

$$\begin{aligned} \mathbf{z}_{\ell} &= \left[ \operatorname{MSA} \left( \operatorname{LN} \left( \mathbf{z}_{\ell-1} \right) \right) + \mathbf{z}_{\ell-1} \right], \quad \& \& \\ \ell &= 1 \dots L \end{aligned}$$

This says that for every layer from  $1$  through to  $L$  (the total number of layers), there's a Multi-Head Attention layer (MSA) wrapping a LayerNorm layer (LN).

The addition on the end is the equivalent of adding the input to the output and forming a [skip/residual connection](#).

We'll call this layer the "MSA block".

In pseudocode, this might look like:

```
x_output_MSA_block = MSA_layer(LN_layer(x_input)) + x_input
```

Notice the skip connection on the end (adding the input of the layers to the output of the layers).

### 3.2.5 Equation 3 overview

$$\begin{aligned} \mathbf{z}_{\ell} &= \left[ \operatorname{MLP} \left( \operatorname{LN} \left( \mathbf{z}_{\ell}^{\prime} \right) \right) + \mathbf{z}_{\ell}^{\prime} \right], \quad \& \& \\ \ell &= 1 \dots L \end{aligned}$$

This says that for every layer from  $1$  through to  $L$  (the total number of layers), there's also a Multilayer Perceptron layer (MLP) wrapping a LayerNorm layer (LN).

The addition on the end is showing the presence of a skip/residual connection.

We'll call this layer the "MLP block".

In pseudocode, this might look like:

```
x_output_MLP_block = MLP_layer(LN_layer(x_output_MSA_block)) + x_output_MSA_block
```

Notice the skip connection on the end (adding the input of the layers to the output of the layers).

### 3.2.6 Equation 4 overview

$$\mathbf{y} = \text{LN}(\mathbf{z})$$

This says for the last layer  $L$ , the output  $y$  is the 0 index token of  $z$  wrapped in a LayerNorm layer (LN).

Or in our case, the 0 index of `x_output_MLP_block`:

```
y = LN_layer(Linear_layer(x_output_MLP_block[0]))
```

Of course there are some simplifications above but we'll take care of those when we start to write PyTorch code for each section.

**Note:** The above section covers a lot of information. But don't forget if something doesn't make sense, you can always research it further. By asking questions like "what is a residual connection?".

### 3.2.7 Exploring Table 1

The final piece of the ViT architecture puzzle we'll focus on (for now) is Table 1.

Model	Layers	Hidden size $D$	MLP size	Heads	Params
ViT-Base	12	768	3072	12	\$86M\$
ViT-Large	24	1024	4096	16	\$307M\$
ViT-Huge	32	1280	5120	16	\$632M\$

Table 1: Details of Vision Transformer model variants. Source: [ViT paper](#).

This table showcasing the various hyperparameters of each of the ViT architectures.

You can see the numbers gradually increase from ViT-Base to ViT-Huge.

We're going to focus on replicating ViT-Base (start small and scale up when necessary) but we'll be writing code that could easily scale up to the larger variants.

Breaking the hyperparameters down:

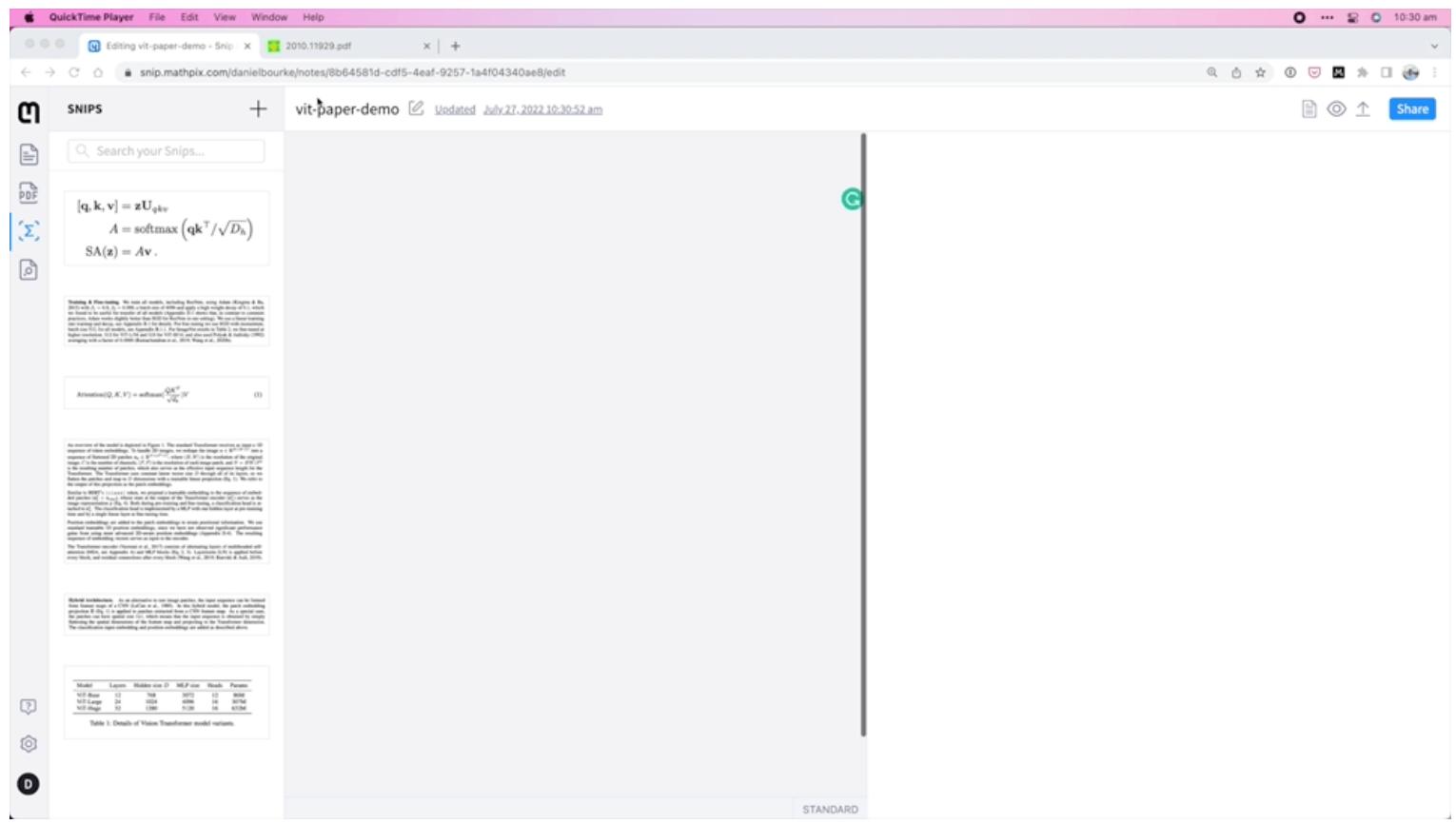
- **Layers** - How many Transformer Encoder blocks are there? (each of these will contain a MSA block and MLP block)
- **Hidden size  $D$**  - This is the embedding dimension throughout the architecture, this will be the size of the vector that our image gets turned into when it gets patched and embedded. Generally, the larger the embedding dimension, the more information can be captured, the better results. However, a larger embedding comes at the cost of more compute.
- **MLP size** - What are the number of hidden units in the MLP layers?
- **Heads** - How many heads are there in the Multi-Head Attention layers?
- **Params** - What are the total number of parameters of the model? Generally, more parameters leads to better performance but at the cost of more compute. You'll notice even ViT-Base has far more parameters than any other model we've used so far.

We'll use these values as the hyperparameter settings for our ViT architecture.

### 3.3 My workflow for replicating papers

When I start working on replicating a paper, I go through the following steps:

1. Read the whole paper end-to-end once (to get an idea of the main concepts).
2. Go back through each section and see how they line up with each other and start thinking about how they might be turned into code (just like above).
3. Repeat step 2 until I've got a fairly good outline.
4. Use [mathpix.com](https://mathpix.com) (a very handy tool) to turn any sections of the paper into markdown/LaTeX to put into notebooks.
5. Replicate the simplest version of the model possible.
6. If I get stuck, look up other examples.



Turning the four equations from the ViT paper into editable LaTeX/markdown using [mathpix.com](https://mathpix.com).

We've already gone through the first few steps above (and if you haven't read the full paper yet, I'd encourage you to give it a go) but what we'll be focusing on next is step 5: replicating the simplest version fo the model possible.

This is why we're starting with ViT-Base.

Replicating the smallest version of the architecture possible, get it working and then we can scale up if we wanted to.

**Note:** If you've never read a research paper before, many of the above steps can be intimidating. But don't worry, like anything, your skills at reading *and* replicating papers will improve with practice. Don't forget, a research paper is often *months* of work by many people compressed into a few pages. So trying to replicate it on your own is no small feat.

## 4. Equation 1: Split data into patches and creating the class, position and patch embedding

I remember one of my machine learning engineer friends used to say "it's all about the embedding."

As in, if you can represent your data in a good, learnable way (as **embeddings are learnable representations**), chances are, a learning algorithm will be able to perform well on them.

With that being said, let's start by creating the class, position and patch embeddings for the ViT architecture.

We'll start with the **patch embedding**.

This means we'll be turning our input images in a sequence of patches and then embedding those patches.

Recall that an **embedding** is a learnable representation of some form and is often a vector.

The term learnable is important because this means the numerical representation of an input image (that the model sees) can be improved over time.

We'll begin by following the opening paragraph of section 3.1 of the ViT paper (bold mine):

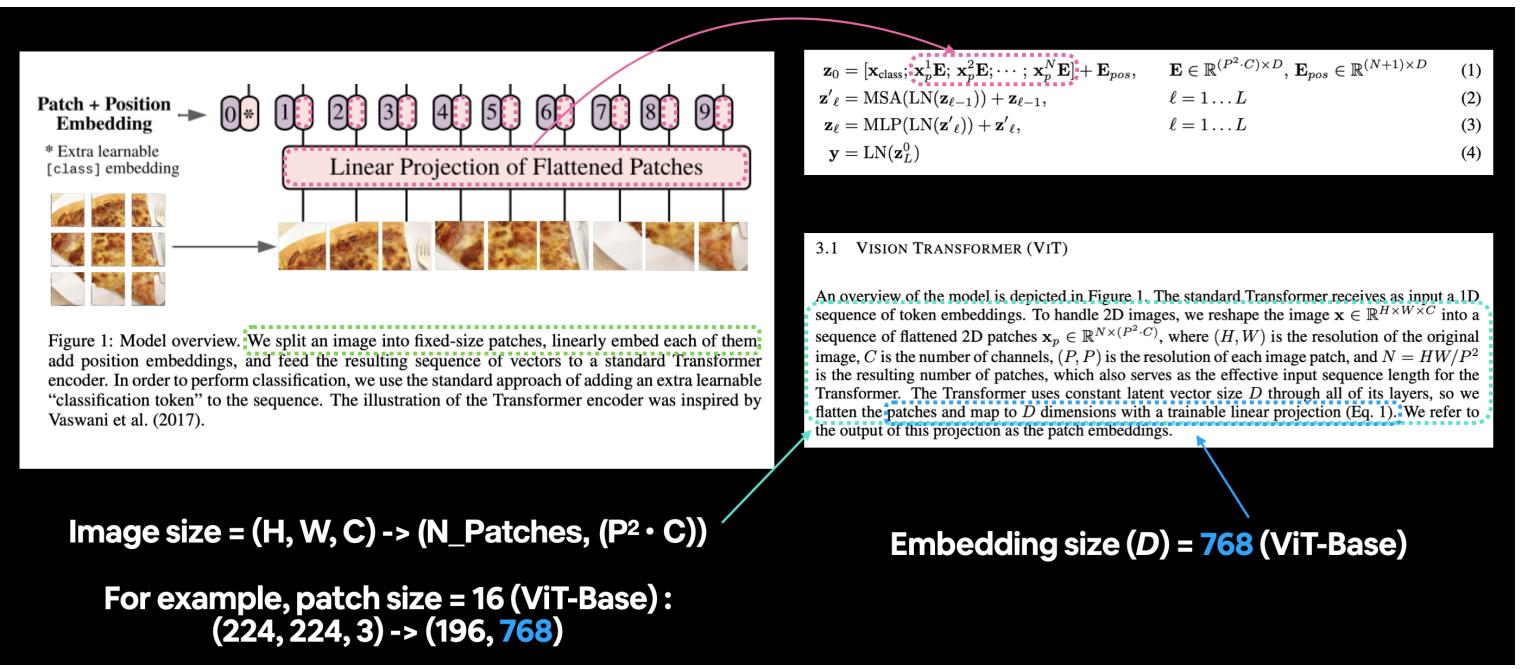
The standard Transformer receives as input a 1D sequence of token embeddings. To handle 2D images, we reshape the image  $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$  into a sequence of flattened 2D patches  $\mathbf{x}_p \in \mathbb{R}^{N \times (P^2 \cdot C)}$ , where  $(H, W)$  is the resolution of the original image,  $C$  is the number of channels,  $(P, P)$  is the resolution of each image patch, and  $N=H \cdot W / P^2$  is the resulting number of patches, which also serves as the effective input sequence length for the Transformer. The Transformer uses constant latent vector size  $D$  through all of its layers, so we flatten the patches and map to  $D$  dimensions with a trainable linear projection (Eq. 1). We refer to the output of this projection as the **patch embeddings**.

And since we're dealing with image shapes, let's keep in mind the line from Table 3 of the ViT paper:

Training resolution is **224**.

Let's break down the text above.

- $D$  is the size of the **patch embeddings**, different values for  $D$  for various sized ViT models can be found in Table 1.
- The image starts as 2D with size  $H \times W \times C$ .
  - $(H, W)$  is the resolution of the original image (height, width).
  - $C$  is the number of channels.
- The image gets converted to a sequence of flattened 2D patches with size  $N \times (P^2 \cdot C)$ .
  - $(P, P)$  is the resolution of each image patch (**patch size**).
  - $N=H \cdot W / P^2$  is the resulting number of patches, which also serves as the input sequence length for the Transformer.



Mapping the patch and position embedding portion of the ViT architecture from Figure 1 to Equation 1. The opening paragraph of section 3.1 describes the different input and output shapes of the patch embedding layer.

## 4.1 Calculating patch embedding input and output shapes by hand

How about we start by calculating these input and output shape values by hand?

To do so, let's create some variables to mimic each of the terms (such as \$H\$, \$W\$ etc) above.

We'll use a patch size (\$P\$) of 16 since it's the best performing version of ViT-Base uses (see column "ViT-B/16" of Table 5 in the ViT paper for more).

In [10]:



Number of patches (N) with image height (H=224), width (W=224) and patch size (P=16): 196

We've got the number of patches, how about we create the image output size as well?

Better yet, let's replicate the input and output shapes of the patch embedding layer.

Recall:

- **Input:** The image starts as 2D with size  $H \times W \times C$ .
- **Output:** The image gets converted to a sequence of flattened 2D patches with size  $N \times \left(P^2 \cdot C\right)$ .

In [11]:

```
Input shape (single 2D image): (224, 224, 3)
Output shape (single 2D image flattened into patches): (196, 768)
```

Input and output shapes acquired!

## 4.2 Turning a single image into patches

Now we know the ideal input and output shapes for our **patch embedding** layer, let's move towards making it.

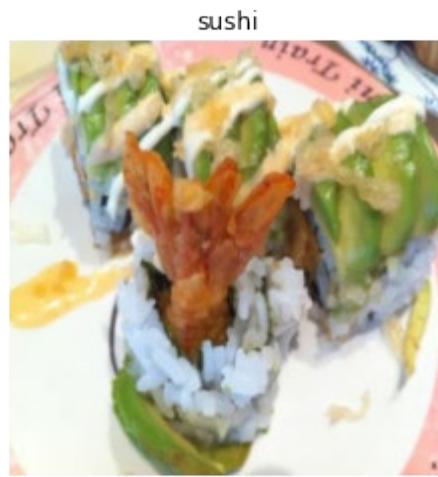
What we're doing is breaking down the overall architecture into smaller pieces, focusing on the inputs and outputs of individual layers.

So how do we create the patch embedding layer?

We'll get to that shortly, first, let's *visualize, visualize, visualize!* what it looks like to turn an image into patches.

Let's start with our single image.

In [12]:

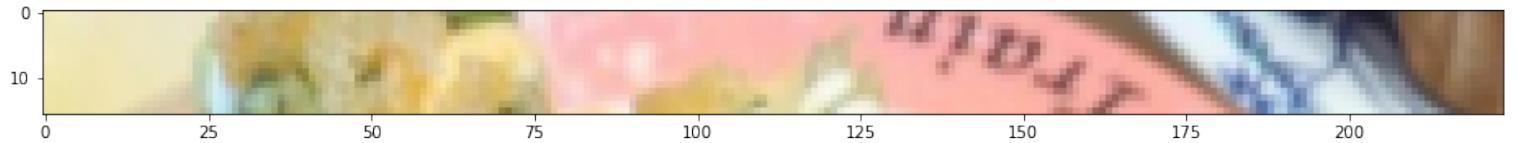


We want to turn this image into patches of itself inline with Figure 1 of the ViT paper.

How about we start by just visualizing the top row of patched pixels?

We can do this by indexing on the different image dimensions.

In [13]:



Now we've got the top row, let's turn it into patches.

We can do this by iterating through the number of patches there'd be in the top row.

In [14]:

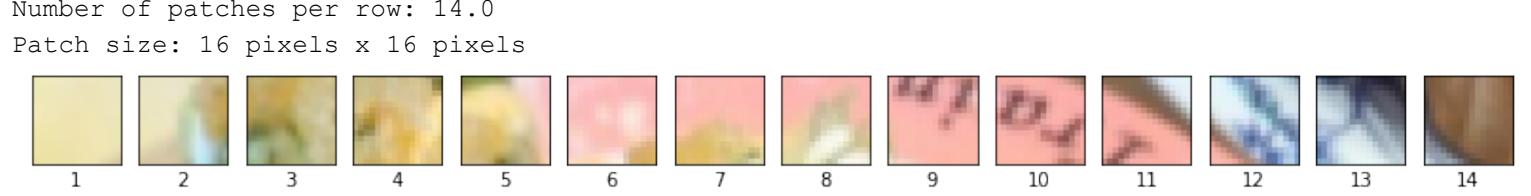












Those are some nice looking patches!

How about we do it for the whole image?

This time we'll iterate through the indexs for height and width and plot each patch as it's own subplot.

In [15]:

























```
Number of patches per row: 14.0
Number of patches per column: 14.0
Total patches: 196.0
Patch size: 16 pixels x 16 pixels
```

## sushi -&gt; Patchified

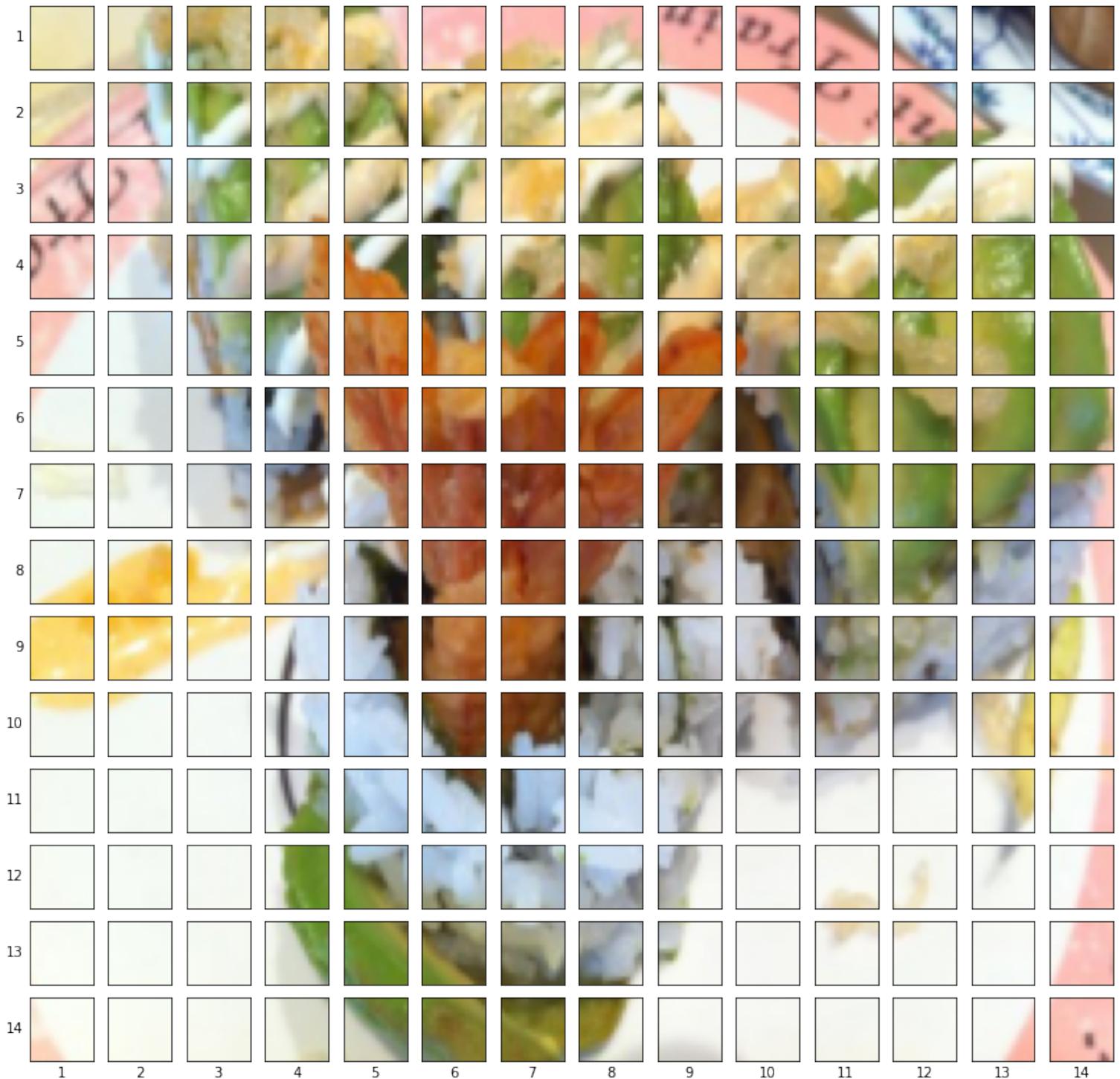


Image patchified!

Woah, that looks cool.

Now how do we turn each of these patches into an embedding and convert them into a sequence?

Hint: we can use PyTorch layers. Can you guess which?

### 4.3 Creating image patches with `torch.nn.Conv2d()`

We've seen what an image looks like when it gets turned into patches, now let's start moving towards replicating the patch embedding layers with PyTorch.

To visualize our single image we wrote code to loop through the different height and width dimensions of a single image and plot individual patches.

This operation is very similar to the convolutional operation we saw in [03. PyTorch Computer Vision section 7.1: Stepping through `nn.Conv2d\(\)`](#).

In fact, the authors of the ViT paper mention in section 3.1 that the patch embedding is achievable with a convolutional neural network (CNN):

**Hybrid Architecture.** As an alternative to raw image patches, the input sequence can be formed from feature maps of a CNN (LeCun et al., 1989). In this hybrid model, the patch embedding projection  $\mathbf{E}$  (Eq. 1) is applied to patches extracted from a **CNN feature map**. As a special case, the patches can have spatial size  $1 \times 1$ , which means that the **input sequence is obtained by simply flattening the spatial dimensions of the feature map and projecting to the Transformer dimension**. The classification input embedding and position embeddings are added as described above.

The "**feature map**" they're referring to are the weights/activations produced by a convolutional layer passing over a given image.

**Original image**

By setting the `kernel_size` and `stride` parameters of a `torch.nn.Conv2d()` layer equal to the `patch_size`, we can effectively get a layer that splits our image into patches and creates a learnable embedding (referred to as a "Linear Projection" in the ViT paper) of each patch.

Remember our ideal input and output shapes for the patch embedding layer?

- **Input:** The image starts as 2D with size  $\{H \times W \times C\}$ .
- **Output:** The image gets converted to a 1D sequence of flattened 2D patches with size  $\{N \times \left(P^2 \cdot C\right)\}$ .

Or for an image size of 224 and patch size of 16:

- **Input (2D image):** (224, 224, 3) -> (height, width, color channels)
- **Output (flattened 2D patches):** (196, 768) -> (number of patches, embedding dimension)

We can recreate these with:

- `torch.nn.Conv2d()` for turning our image into patches of CNN feature maps.
- `torch.nn.Flatten()` for flattening the spatial dimensions of the feature map.

Let's start with the `torch.nn.Conv2d()` layer.

We can replicate the creation of patches by setting the `kernel_size` and `stride` equal to `patch_size`.

This means each convolutional kernel will be of size `(patch_size x patch_size)` or if `patch_size=16`, `(16 x 16)` (the equivalent of one whole patch).

And each step or `stride` of the convolutional kernel will be `patch_size` pixels long or `16` pixels long (equivalent of stepping to the next patch).

We'll set `in_channels=3` for the number of color channels in our image and we'll set `out_channels=768`, the same as the `D$` value in Table 1 for ViT-Base (this is the embedding dimension, each image will be embedded into a learnable vector of size 768).

In [16]:





Now we've got a convolutional layer, let's see what happens when we pass a single image through it.

In [17]:



In [18]:

```
torch.Size([1, 768, 14, 14])
```

Passing our image through the convolutional layer turns it into a series of 768 (this is the embedding size or \$D\$) feature/activation maps.

So its output shape can be read as:

```
torch.Size([1, 768, 14, 14]) -> [batch_size, embedding_dim, feature_map_height,  
feature_map_width]
```

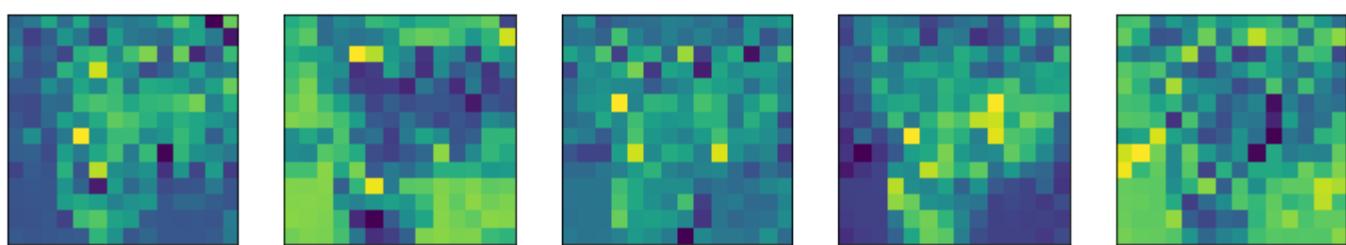
Let's visualize five random feature maps and see what they look like.

In [19]:





Showing random convolutional feature maps from indexes: [571, 727, 734, 380, 90]



Notice how the feature maps all kind of represent the original image, after visualizing a few more you can start to see the different major outlines and some major features.

The important thing to note is that these features may change over time as the neural network learns.

And because of these, these feature maps can be considered a **learnable embedding** of our image.

Let's check one out in numerical form.

In [20]:

Out[20]:

```
(tensor([[[ 0.4732,  0.3567,  0.3377,  0.3736,  0.3208,  0.3913,  0.3464,
          0.3702,  0.2541,  0.3594,  0.1984,  0.3982,  0.3741,  0.1251],
         [ 0.4178,  0.4771,  0.3374,  0.3353,  0.3159,  0.4008,  0.3448,
          0.3345,  0.5850,  0.4115,  0.2969,  0.2751,  0.6150,  0.4188],
         [ 0.3209,  0.3776,  0.4970,  0.4272,  0.3301,  0.4787,  0.2754,
          0.3726,  0.3298,  0.4631,  0.3087,  0.4915,  0.4129,  0.4592],
```

```
[ 0.4540,  0.4930,  0.5570,  0.2660,  0.2150,  0.2044,  0.2766,
  0.2076,  0.3278,  0.3727,  0.2637,  0.2493,  0.2782,  0.3664],
[ 0.4920,  0.5671,  0.3298,  0.2992,  0.1437,  0.1701,  0.1554,
  0.1375,  0.1377,  0.3141,  0.2694,  0.2771,  0.2412,  0.3700],
[ 0.5783,  0.5790,  0.4229,  0.5032,  0.1216,  0.1000,  0.0356,
  0.1258, -0.0023,  0.1640,  0.2809,  0.2418,  0.2606,  0.3787],
[ 0.5334,  0.5645,  0.4781,  0.3307,  0.2391,  0.0461,  0.0095,
  0.0542,  0.1012,  0.1331,  0.2446,  0.2526,  0.3323,  0.4120],
[ 0.5724,  0.2840,  0.5188,  0.3934,  0.1328,  0.0776,  0.0235,
  0.1366,  0.3149,  0.2200,  0.2793,  0.2351,  0.4722,  0.4785],
[ 0.4009,  0.4570,  0.4972,  0.5785,  0.2261,  0.1447, -0.0028,
  0.2772,  0.2697,  0.4008,  0.3606,  0.3372,  0.4535,  0.4492],
[ 0.5678,  0.5870,  0.5824,  0.3438,  0.5113,  0.0757,  0.1772,
  0.3677,  0.3572,  0.3742,  0.3820,  0.4868,  0.3781,  0.4694],
[ 0.5845,  0.5877,  0.5826,  0.3212,  0.5276,  0.4840,  0.4825,
  0.5523,  0.5308,  0.5085,  0.5606,  0.5720,  0.4928,  0.5581],
[ 0.5853,  0.5849,  0.5793,  0.3410,  0.4428,  0.4044,  0.3275,
  0.4958,  0.4366,  0.5750,  0.5494,  0.5868,  0.5557,  0.5069],
[ 0.5880,  0.5888,  0.5796,  0.3377,  0.2635,  0.2347,  0.3145,
  0.3486,  0.5158,  0.5722,  0.5347,  0.5753,  0.5816,  0.4378],
[ 0.5692,  0.5843,  0.5721,  0.5081,  0.2694,  0.2032,  0.1589,
  0.3464,  0.5349,  0.5768,  0.5739,  0.5764,  0.5394,  0.4482]]],
grad_fn=<SliceBackward0>),
True)
```

The `grad_fn` output of the `single_feature_map` and the `requires_grad=True` attribute means PyTorch is tracking the gradients of this feature map and it will be updated by gradient descent during training.

## 4.4 Flattening the patch embedding with `torch.nn.Flatten()`

We've turned our image into patch embeddings but they're still in 2D format.

How do we get them into the desired output shape of the patch embedding layer of the ViT model?

- **Desried output (1D sequence of flattened 2D patches):**  $(196, 768) \rightarrow (\text{number of patches, embedding dimension}) \rightarrow N \times (P^2 \times C)$

Let's check the current shape.

In [21]:

```
Current tensor shape: torch.Size([1, 768, 14, 14]) -> [batch, embedding_dim, feature_map_height, feature_map_width]
```

Well we've got the 768 part ( $(P^2) \cdot C$ ) but we still need the number of patches ( $N$ ).

Reading back through section 3.1 of the ViT paper it says (bold mine):

As a special case, the patches can have spatial size  $1 \times 1$ , which means that the **input sequence is obtained by simply flattening the spatial dimensions of the feature map and projecting to the Transformer dimension.**

Flattening the spatial dimensions of the feature map hey?

What layer do we have in PyTorch that can flatten?

How about `torch.nn.Flatten()` ?

But we don't want to flatten the whole tensor, we only want to flatten the "spatial dimensions of the feature map".

Which in our case is the `feature_map_height` and `feature_map_width` dimensions of `image_out_of_conv`.

So how about we create a `torch.nn.Flatten()` layer to only flatten those dimensions, we can use the `start_dim` and `end_dim` parameters to set that up?

In [22]:

Nice! Now let's put it all together!

We'll:

1. Take a single image.
2. Put in through the convolutional layer (`conv2d`) to turn the image into 2D feature maps (patch embeddings).
3. Flatten the 2D feature map into a single sequence.

In [23]:



Original image shape: `torch.Size([3, 224, 224])`  
 Image feature map shape: `torch.Size([1, 768, 14, 14])`  
 Flattened image feature map shape: `torch.Size([1, 768, 196])`

**sushi**

Woohoo! It looks like our `image_out_of_conv_flattened` shape is very close to our desired output shape:

- **Desried output (flattened 2D patches):**  $(196, 768) \rightarrow \$\{N \times (P^2 \cdot C)\}$
- **Current shape:**  $(1, 768, 196)$

The only difference is our current shape has a batch size and the dimensions are in a different order to the desired output.

How could we fix this?

Well, how about we rearrange the dimensions?

We can do so with `torch.Tensor.permute()` just like we do when rearranging image tensors to plot them with `matplotlib`.

Let's try.

In [24]:

```
Patch embedding sequence shape: torch.Size([1, 196, 768]) -> [batch_size, num_patches, embedding_size]
```

Yes!!!

We've now matched the desired input and output shapes for the patch embedding layer of the ViT architecture using a couple of PyTorch layers.

How about we visualize one of the flattened feature maps?

In [25]:



Flattened feature map shape: torch.Size([1, 196])

Hmm, the flattened feature map doesn't look like much visually, but that's not what we're concerned about, this is what will be the output of the patching embedding layer and the input to the rest of the ViT architecture.

**Note:** The original Transformer architecture was designed to work with text. The Vision Transformer architecture (ViT) had the goal of using the original Transformer for images. This is why the input to the ViT architecture is processed in the way it is. We're essentially taking a 2D image and formatting it so it appears as a 1D sequence of text.

How about we view the flattened feature map in tensor form?

In [26]:

Out[26]:

```
(tensor([[ 0.4732,  0.3567,  0.3377,  0.3736,  0.3208,  0.3913,  0.3464,  0.3702,
        0.2541,  0.3594,  0.1984,  0.3982,  0.3741,  0.1251,  0.4178,  0.4771,
        0.3374,  0.3353,  0.3159,  0.4008,  0.3448,  0.3345,  0.5850,  0.4115,
        0.2969,  0.2751,  0.6150,  0.4188,  0.3209,  0.3776,  0.4970,  0.4272,
        0.3301,  0.4787,  0.2754,  0.3726,  0.3298,  0.4631,  0.3087,  0.4915,
        0.4129,  0.4592,  0.4540,  0.4930,  0.5570,  0.2660,  0.2150,  0.2044,
        0.2766,  0.2076,  0.3278,  0.3727,  0.2637,  0.2493,  0.2782,  0.3664,
        0.4920,  0.5671,  0.3298,  0.2992,  0.1437,  0.1701,  0.1554,  0.1375,
        0.1377,  0.3141,  0.2694,  0.2771,  0.2412,  0.3700,  0.5783,  0.5790,
        0.4229,  0.5032,  0.1216,  0.1000,  0.0356,  0.1258, -0.0023,  0.1640,
        0.2809,  0.2418,  0.2606,  0.3787,  0.5334,  0.5645,  0.4781,  0.3307,
        0.2391,  0.0461,  0.0095,  0.0542,  0.1012,  0.1331,  0.2446,  0.2526,
        0.3323,  0.4120,  0.5724,  0.2840,  0.5188,  0.3934,  0.1328,  0.0776,
        0.0235,  0.1366,  0.3149,  0.2200,  0.2793,  0.2351,  0.4722,  0.4785,
        0.4009,  0.4570,  0.4972,  0.5785,  0.2261,  0.1447, -0.0028,  0.2772,
        0.2697,  0.4008,  0.3606,  0.3372,  0.4535,  0.4492,  0.5678,  0.5870,
        0.5824,  0.3438,  0.5113,  0.0757,  0.1772,  0.3677,  0.3572,  0.3742,
        0.3820,  0.4868,  0.3781,  0.4694,  0.5845,  0.5877,  0.5826,  0.3212,
        0.5276,  0.4840,  0.4825,  0.5523,  0.5308,  0.5085,  0.5606,  0.5720,
        0.4928,  0.5581,  0.5853,  0.5849,  0.5793,  0.3410,  0.4428,  0.4044,
        0.3275,  0.4958,  0.4366,  0.5750,  0.5494,  0.5868,  0.5557,  0.5069,
        0.5880,  0.5888,  0.5796,  0.3377,  0.2635,  0.2347,  0.3145,  0.3486,
        0.5158,  0.5722,  0.5347,  0.5753,  0.5816,  0.4378,  0.5692,  0.5843,
        0.5721,  0.5081,  0.2694,  0.2032,  0.1589,  0.3464,  0.5349,  0.5768,
```

```
0.5739, 0.5764, 0.5394, 0.4482]], grad_fn=<SelectBackward0>),  
True,  
torch.Size([1, 196]))
```

Beautiful!

We've turned our single 2D image into a 1D learnable embedding vector (or "Linear Projection of Flattened Patches" in Figure 1 of the ViT paper).

## 4.5 Turning the ViT patch embedding layer into a PyTorch module

Time to put everything we've done for creating the patch embedding into a single PyTorch layer.

We can do so by subclassing `nn.Module` and creating a small PyTorch "model" to do all of the steps above.

Specifically we'll:

1. Create a class called `PatchEmbedding` which subclasses `nn.Module` (so it can be used a PyTorch layer).
2. Initialize the class with the parameters `in_channels=3`, `patch_size=16` (for ViT-Base) and `embedding_dim=768` (this is \$D\$ for ViT-Base from Table 1).
3. Create a layer to turn an image into patches using `nn.Conv2d()` (just like in 4.3 above).
4. Create a layer to flatten the patch feature maps into a single dimension (just like in 4.4 above).
5. Define a `forward()` method to take an input and pass it through the layers created in 3 and 4.
6. Make sure the output shape reflects the required output shape of the ViT architecture ( $\{N \times (P^2 \times C)\}$ ).

Let's do it!

In [27]:































PatchEmbedding layer created!

Let's try it out on a single image.

In [28]:



```
Input image shape: torch.Size([1, 3, 224, 224])
Output patch embedding shape: torch.Size([1, 196, 768])
```

Beautiful!

The output shape matches the ideal input and output shapes we'd like to see from the patch embedding layer:

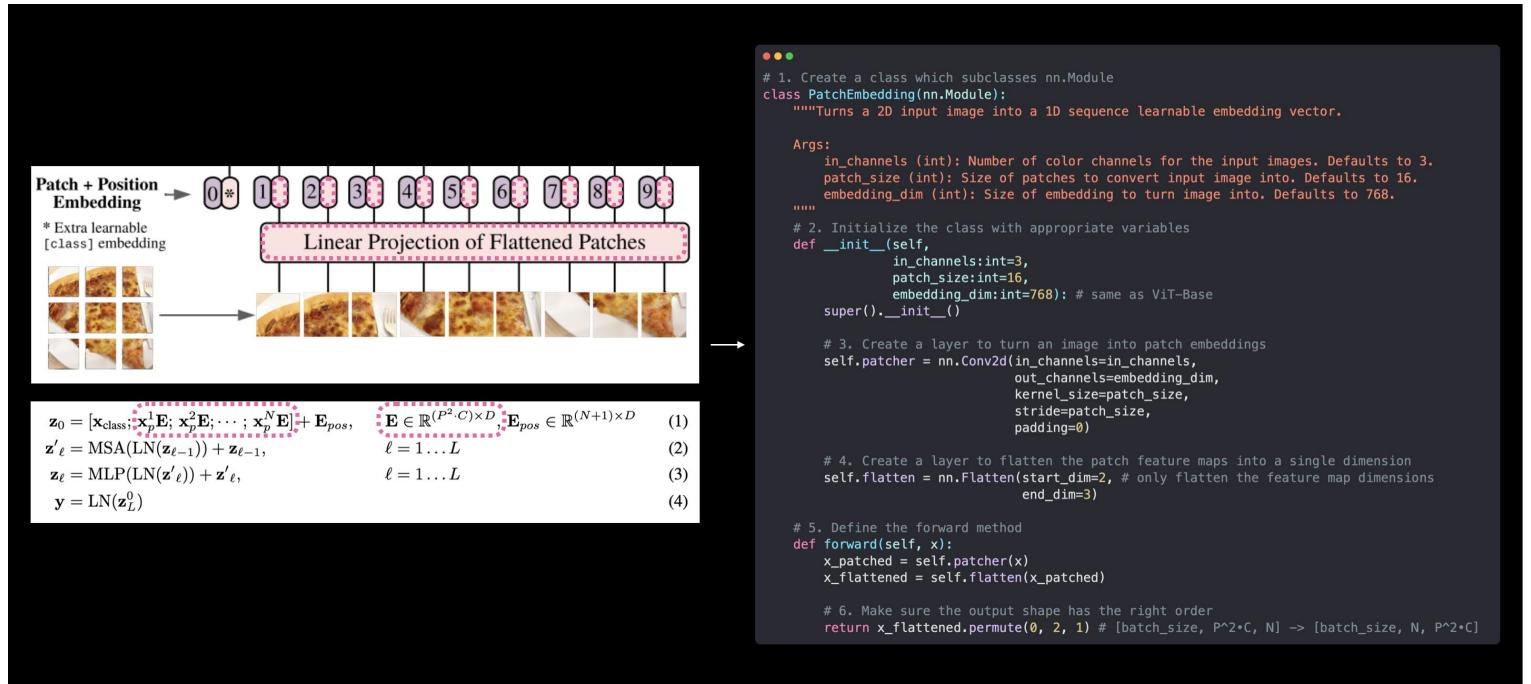
- **Input:** The image starts as 2D with size  $\{H \times W \times C\}$ .
- **Output:** The image gets converted to a 1D sequence of flattened 2D patches with size  $\{N \times \left(P^2 \cdot C\right)\}$ .

Where:

- $(H, W)$  is the resolution of the original image.
- $C$  is the number of channels.
- $(P, P)$  is the resolution of each image patch (**patch size**).
- $N=H \cdot W / P^2$  is the resulting number of patches, which also serves as the effective input sequence length for the Transformer.

We've now replicated the patch embedding for equation 1 but not the class token/position embedding.

We'll get to these later on.



Our `PatchEmbedding` class (right) replicates the patch embedding of the ViT architecture from Figure 1 and Equation 1 from the ViT paper (left). However, the learnable class embedding and position embeddings haven't been created yet. These will come soon.

Let's now get a summary of our `PatchEmbedding` layer.

In [29]:





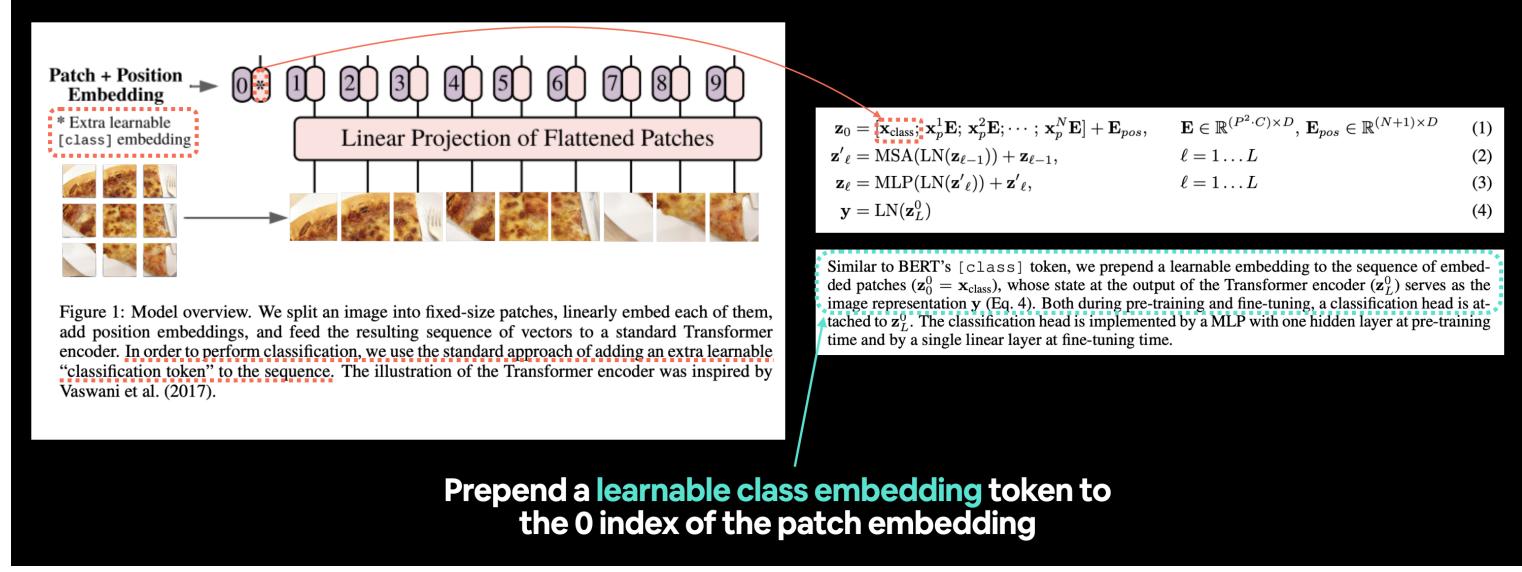


Layer (type (var_name))	Input Shape	Output Shape	Param #	Trainable
PatchEmbedding (PatchEmbedding)	[1, 3, 224, 224]	[1, 196, 768]	--	True
—Conv2d (patcher)	[1, 3, 224, 224]	[1, 768, 14, 14]	590,592	True
—Flatten (flatten)	[1, 768, 14, 14]	[1, 768, 196]	--	--
Total params: 590,592				
Trainable params: 590,592				
Non-trainable params: 0				
Total mult-adds (M): 115.76				
Input size (MB): 0.60				
Forward/backward pass size (MB): 1.20				
Params size (MB): 2.36				
Estimated Total Size (MB): 4.17				

## 4.6 Creating the class token embedding

Okay we've made the image patch embedding, time to get to work on the class token embedding.

Or  $\mathbf{x}_{\text{class}}$  from equation 1.



Left: Figure 1 from the ViT paper with the "classification token" or  $[class]$  embedding token we're going to recreate highlighted. Right: Equation 1 and section 3.1 of the ViT paper that relate to the learnable class embedding token.

Reading the second paragraph of section 3.1 from the ViT paper, we see the following description:

Similar to BERT's  $[class]$  token, we prepend a learnable embedding to the sequence of embedded patches  $\mathbf{z}_0^0 = \mathbf{x}_{\text{class}}$ , whose state at the output of the Transformer encoder  $\mathbf{z}_L^0$  serves as the image representation  $\mathbf{y}$  (Eq. 4).

**Note:** [BERT](#) (Bidirectional Encoder Representations from Transformers) is one of the original machine learning research papers to use the Transformer architecture to achieve outstanding results on natural language processing (NLP) tasks and is where the idea of having a `[ class ]` token at the start of a sequence originated, class being a description for the "classification" class the sequence belonged to.

So we need to "prepend a learnable embedding to the sequence of embedded patches".

Let's start by viewing our sequence of embedded patches tensor (created in section 4.5) and its shape.

In [30]:

```
tensor([[-0.9145,  0.2454, -0.2292,    ... ,  0.6768, -0.4515,  0.3496],
       [-0.7427,  0.1955, -0.3570,    ... ,  0.5823, -0.3458,  0.3261],
       [-0.7589,  0.2633, -0.1695,    ... ,  0.5897, -0.3980,  0.0761],
```

```

    ...,
    [-1.0072,  0.2795, -0.2804,  ...,  0.7624, -0.4584,  0.3581],
    [-0.9839,  0.1652, -0.1576,  ...,  0.7489, -0.5478,  0.3486],
    [-0.9260,  0.1383, -0.1157,  ...,  0.5847, -0.4717,  0.3112]]],
grad_fn=<PermuteBackward0>
Patch embedding shape: torch.Size([1, 196, 768]) -> [batch_size, number_of_patches, embedding_dimension]

```

To "prepend a learnable embedding to the sequence of embedded patches" we need to create a learnable embedding in the shape of the `embedding_dimension` (\$D\$) and then add it to the `number_of_patches` dimension.

Or in pseudocode:

```

patch_embedding = [image_patch_1, image_patch_2, image_patch_3...]
class_token = learnable_embedding
patch_embedding_with_class_token = torch.cat((class_token, patch_embedding), dim=1)

```

Notice the concatenation (`torch.cat()`) happens on `dim=1` (the `number_of_patches` dimension).

Let's create a learnable embedding for the class token.

To do so, we'll get the batch size and embedding dimension shape and then we'll create a `torch.ones()` tensor in the shape `[batch_size, 1, embedding_dimension]`.

And we'll make the tensor learnable by passing it to `nn.Parameter()` with `requires_grad=True`.

In [31]:





```
tensor([[[1., 1., 1., 1., 1., 1., 1., 1., 1.]]], grad_fn=<SliceBackward0>)
Class token shape: torch.Size([1, 1, 768]) -> [batch_size, number_of_tokens, embedding_dimension]
```

**Note:** Here we're only creating the class token embedding as `torch.ones()` for demonstration purposes, in reality, you'd likely create the class token embedding with `torch.randn()` (since machine learning is all about harnessing the power of controlled randomness, you generally start with a random number and improve it over time).

See how the `number_of_tokens` dimension of `class_token` is `1` since we only want to prepend one class token value to the start of the patch embedding sequence.

Now we've got the class token embedding, let's prepend it to our sequence of image patches, `patch_embedded_image`.

We can do so using `torch.cat()` and set `dim=1` (so `class_token`'s `number_of_tokens` dimension is prepended to `patch_embedded_image`'s `number_of_patches` dimension).

In [32]:





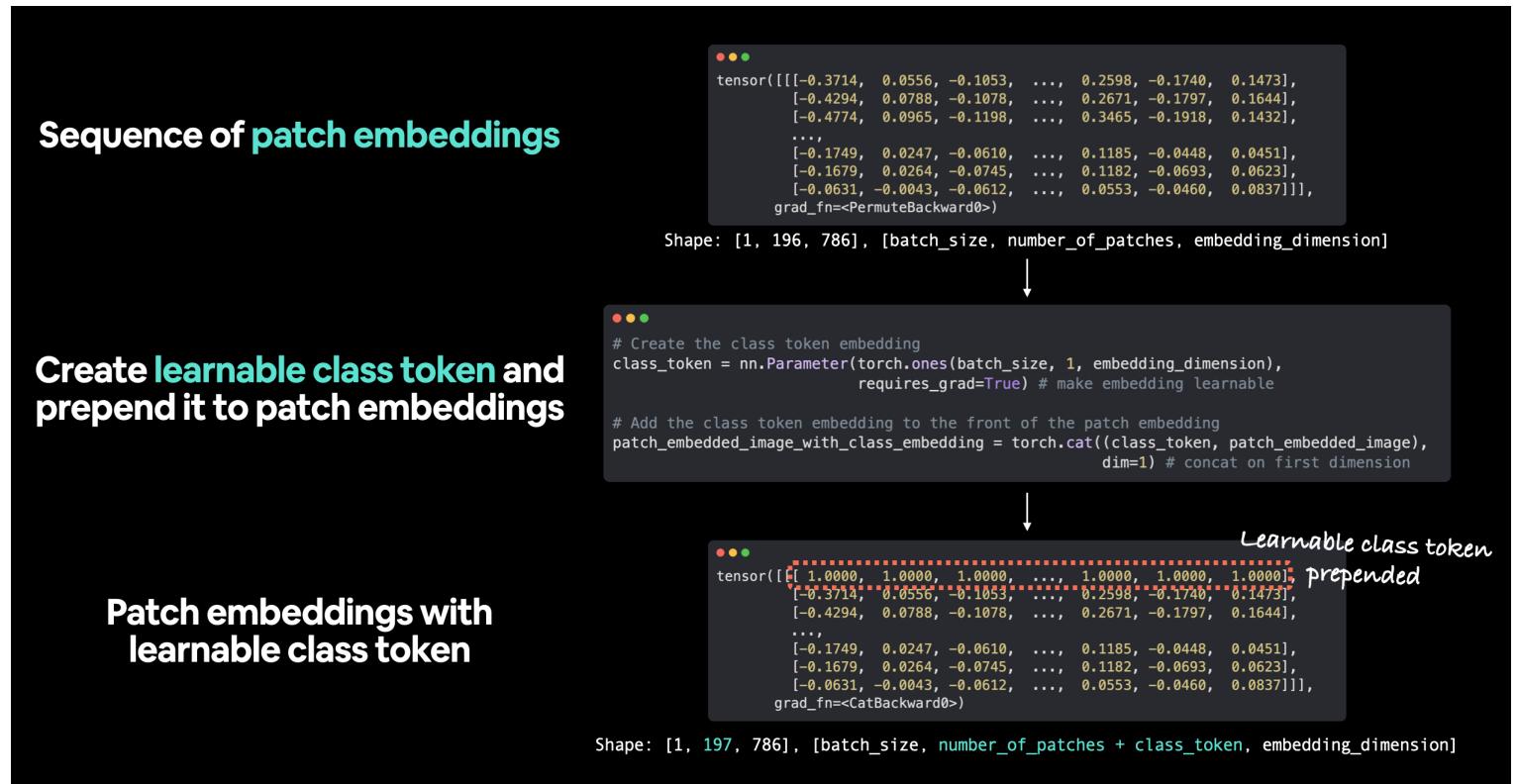
```

tensor([[[ 1.0000,  1.0000,  1.0000, ... ,  1.0000,  1.0000,  1.0000],
        [-0.9145,  0.2454, -0.2292, ... ,  0.6768, -0.4515,  0.3496],
        [-0.7427,  0.1955, -0.3570, ... ,  0.5823, -0.3458,  0.3261],
        ... ,
        [-1.0072,  0.2795, -0.2804, ... ,  0.7624, -0.4584,  0.3581],
        [-0.9839,  0.1652, -0.1576, ... ,  0.7489, -0.5478,  0.3486],
        [-0.9260,  0.1383, -0.1157, ... ,  0.5847, -0.4717,  0.3112]]],
grad_fn=<CatBackward0>)

```

Sequence of patch embeddings with class token prepended shape: torch.Size([1, 197, 768]) -> [batch\_size, number\_of\_patches, embedding\_dimension]

Nice! Learnable class token prepended!



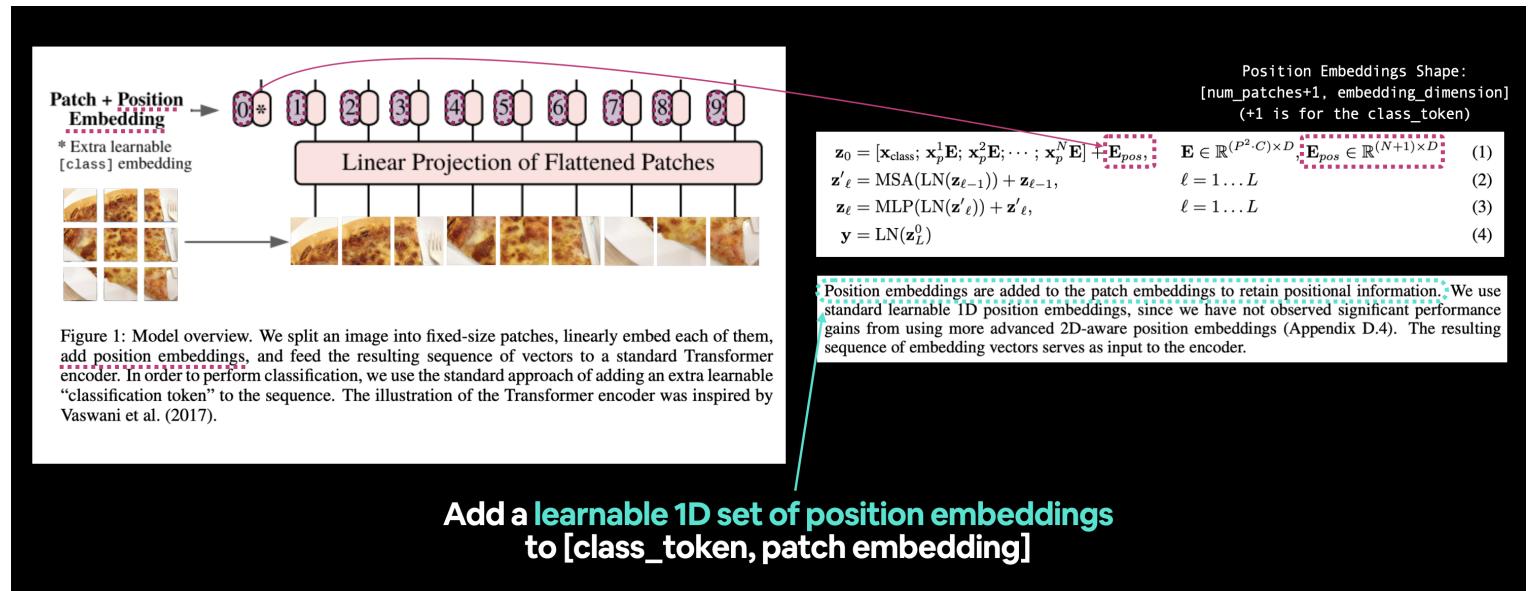
Reviewing what we've done to create the learnable class token, we start with a sequence of image patch embeddings created by `PatchEmbedding()` on single image, we then created a learnable class token with one value for each of the

*embedding dimensions and then prepended it to the original sequence of patch embeddings. Note: Using `torch.ones()` to create the learnable class token is mostly for demonstration purposes only, in practice, you'd like create it with `torch.randn()`.*

## 4.7 Creating the position embedding

Well, we've got the class token embedding and the patch embedding, now how might we create the position embedding?

Or  $\mathbf{E}_{\text{pos}}$  from equation 1 where  $\mathbf{E}$  stands for "embedding".



Left: Figure 1 from the ViT paper with the position embedding we're going to recreate highlighted. Right: Equation 1 and section 3.1 of the ViT paper that relate to the position embedding.

Let's find out more by reading section 3.1 of the ViT paper (bold mine):

Position embeddings are added to the patch embeddings to retain positional information. We use **standard learnable 1D position embeddings**, since we have not observed significant performance gains from using more advanced 2D-aware position embeddings (Appendix D.4). The resulting sequence of embedding vectors serves as input to the encoder.

By "retain positional information" the authors mean they want the architecture to know what "order" the patches come in. As in, patch two comes after patch one and patch three comes after patch two and on and on.

This positional information can be important when considering what's in an image (without positional information a flattened sequence could be seen as having no order and thus no patch relates to any other patch).

To start creating the position embeddings, let's view our current embeddings.

In [33]:

```
(tensor([[[ 1.0000,  1.0000,  1.0000, ...,  1.0000,  1.0000,  1.0000],
       [-0.9145,  0.2454, -0.2292, ...,  0.6768, -0.4515,  0.3496],
       [-0.7427,  0.1955, -0.3570, ...,  0.5823, -0.3458,  0.3261],
       ...,
       [-1.0072,  0.2795, -0.2804, ...,  0.7624, -0.4584,  0.3581],
       [-0.9839,  0.1652, -0.1576, ...,  0.7489, -0.5478,  0.3486],
       [-0.9260,  0.1383, -0.1157, ...,  0.5847, -0.4717,  0.3112]]],
      grad_fn=<CatBackward0>),
      torch.Size([1, 197, 768]))
```

Equation 1 states that the position embeddings ( $\mathbf{E}_{\text{pos}}$ ) should have the shape  $(N + 1) \times D$ :

$$\mathbf{E}_{\text{pos}} \in \mathbb{R}^{(N+1) \times D}$$

Where:

- $N = H \cdot W / P^2$  is the resulting number of patches, which also serves as the effective input sequence length for the Transformer (number of patches).

- $D$  is the size of the **patch embeddings**, different values for  $D$  can be found in Table 1 (embedding dimension).

Luckily we've got both of these values already.

So let's make a learnable 1D embedding with `torch.ones()` to create  $\mathbf{E}_{\text{pos}}$ .

In [34]:









```
tensor([[1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1.]]], grad_fn=<SliceBackward0>)
Position embedding shape: torch.Size([1, 197, 768]) -> [batch_size, number_of_patches, embedding_dimension]
```

**Note:** Only creating the position embedding as `torch.ones()` for demonstration purposes, in reality, you'd likely create the position embedding with `torch.randn()` (start with a random number and improve via gradient descent).

Position embeddings created!

Let's add them to our sequence of patch embeddings with a prepended class token.

In [35]:

```
tensor([[[ 2.0000,  2.0000,  2.0000,  ...,  2.0000,  2.0000,  2.0000],
        [ 0.0855,  1.2454,  0.7708,  ...,  1.6768,  0.5485,  1.3496],
        [ 0.2573,  1.1955,  0.6430,  ...,  1.5823,  0.6542,  1.3261],
        ...,
        [-0.0072,  1.2795,  0.7196,  ...,  1.7624,  0.5416,  1.3581],
        [ 0.0161,  1.1652,  0.8424,  ...,  1.7489,  0.4522,  1.3486],
        [ 0.0740,  1.1383,  0.8843,  ...,  1.5847,  0.5283,  1.3112]]],  
grad_fn=<AddBackward0>)  
Patch embeddings, class token prepended and positional embeddings added shape: torch.Size([1, 197  
, 768]) -> [batch_size, number_of_patches, embedding_dimension]
```

Notice how the values of each of the elements in the embedding tensor increases by 1 (this is because of the position

embeddings being created with `torch.ones()`).

**Note:** We could put both the class token embedding and position embedding into their own layer if we wanted to. But we'll see later on in section 8 how they can be incorporated into the overall ViT architecture's `forward()` method.



The workflow we've used for adding the position embeddings to the sequence of patch embeddings and class token.

**Note:** `torch.ones()` only used to create embeddings for illustration purposes, in practice, you'd likely use `torch.randn()` to start with a random number.

## 4.8 Putting it all together: from image to embedding

Alright, we've come a long way in terms of turning our input images into an embedding and replicating equation 1 from section 3.1 of the ViT paper:

```
$$ \begin{aligned} \mathbf{z}_0 &= \left[ \mathbf{x}_{\text{class}} ; \mathbf{x}_p^1 \mathbf{E} ; \mathbf{x}_p^2 \mathbf{E} ; \dots ; \mathbf{x}_p^N \mathbf{E} \right] + \mathbf{E}_{\text{pos}}, \\ \mathbf{E} &\in \mathbb{R}^{(N+1) \times D}, \mathbf{E}_{\text{pos}} \in \mathbb{R}^D \end{aligned} $$
```

Let's now put everything together in a single code cell and go from input image ( $\mathbf{x}$ ) to output embedding ( $\mathbf{z}_0$ ).

We can do so by:

1. Setting the patch size (we'll use `16` as it's widely used throughout the paper and for ViT-Base).
2. Getting a single image, printing its shape and storing its height and width.
3. Adding a batch dimension to the single image so it's compatible with our `PatchEmbedding` layer.
4. Creating a `PatchEmbedding` layer (the one we made in section 4.5) with a `patch_size=16` and `embedding_dim=768` (from Table 1 for ViT-Base).
5. Passing the single image through the `PatchEmbedding` layer in 4 to create a sequence of patch embeddings.
6. Creating a class token embedding like in section 4.6.
7. Prepending the class token embedding to the patch embeddings created in step 5.
8. Creating a position embedding like in section 4.7.
9. Adding the position embedding to the class token and patch embeddings created in step 7.

We'll also make sure to set the random seeds with `set_seeds()` and print out the shapes of different tensors along the way.

In [36]:

















```
Image tensor shape: torch.Size([3, 224, 224])
Input image with batch dimension shape: torch.Size([1, 3, 224, 224])
Patching embedding shape: torch.Size([1, 196, 768])
Class token embedding shape: torch.Size([1, 1, 768])
Patch embedding with class token shape: torch.Size([1, 197, 768])
Patch and position embedding shape: torch.Size([1, 197, 768])
```

Woohoo!

From a single image to patch and position embeddings in a single cell of code.

```

# 1. Set patch size
patch_size = 16

# 2. Print shape of original image tensor and get the image dimensions
print(f"Image tensor shape: {image.shape}")
height, width = image.shape[1], image.shape[2]

# 3. Get image tensor and add batch dimension
x = image.unsqueeze(0)
print(f"Input image with batch dimension shape: {x.shape}")

# 4. Create patch embedding layer
patch_embedding_layer = PatchEmbedding(in_channels=3, # number of color channels in image
                                         patch_size=patch_size,
                                         embedding_dim=768) # from Table 1 for ViT-Base

# 5. Pass image through patch embedding layer
patch_embedding = patch_embedding_layer(x)
print(f"Patch embedding shape: {patch_embedding.shape}")

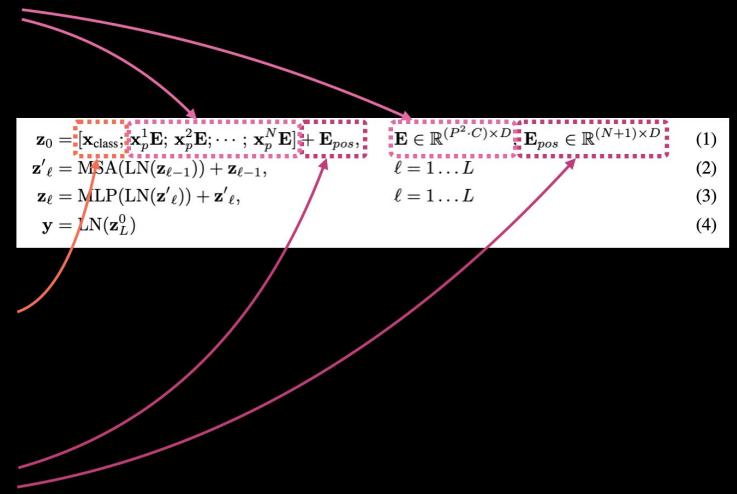
# 6. Create class token embedding
batch_size = patch_embedding.shape[0]
embedding_dimension = patch_embedding.shape[-1]
class_token = nn.Parameter(torch.ones(batch_size, 1, embedding_dimension),
                           requires_grad=True) # make sure it's learnable
print(f"Class token embedding shape: {class_token.shape}")

# 7. Prepend class token embedding to patch embedding
patch_embedding_class_token = torch.cat((class_token, patch_embedding), dim=1)
print(f"Patch embedding with class token shape: {patch_embedding_class_token.shape}")

# 8. Create position embedding
number_of_patches = int((height * width) / patch_size**2)
position_embedding = nn.Parameter(torch.ones(1, number_of_patches+1, embedding_dimension),
                                 requires_grad=True) # make sure it's learnable

# 9. Add position embedding to patch embedding with class token
patch_and_position_embedding = patch_embedding_class_token + position_embedding
print(f"Patch and position embedding shape: {patch_and_position_embedding.shape}")

```



Mapping equation 1 from the ViT paper to our PyTorch code. This is the essence of paper replicating, taking a research paper and turning it into usable code.

Now we've got a way to encode our images and pass them to the Transformer Encoder in Figure 1 of the ViT paper.



Animating the entire ViT workflow: from patch embeddings to transformer encoder to MLP head.

From a code perspective, creating the patch embedding is probably the largest section of replicating the ViT paper.

Many of the other parts of the ViT paper such as the Multi-Head Attention and Norm layers can be created using existing PyTorch layers.

Onwards!

## 5. Equation 2: Multi-Head Attention (MSA)

We've got our input data patchified and embedded, now let's move onto the next part of the ViT architecture.

To start, we'll break down the Transformer Encoder section into two parts (start small and increase when necessary).

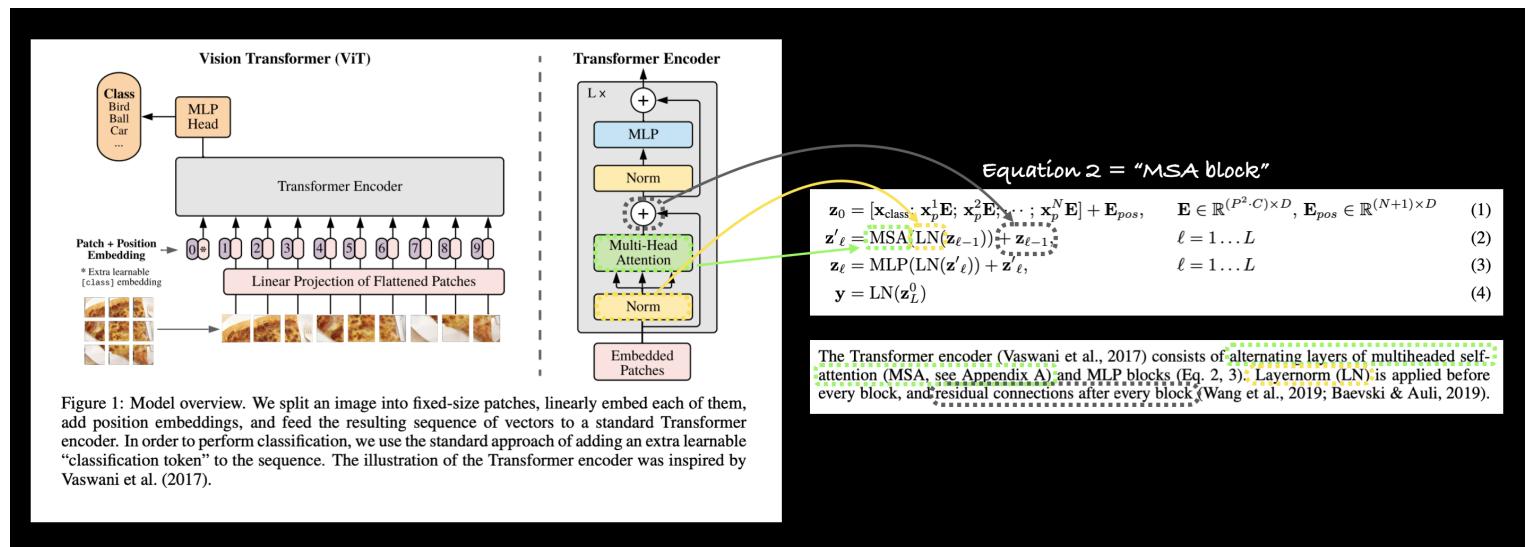
The first being equation 2 and the second being equation 3.

Recall equation 2 states:

$$\begin{aligned} & \text{\$\$ \begin{aligned} \mathbf{z}_{\ell}' &= \operatorname{MSA}(\operatorname{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \\ & \ell = 1 \dots L \end{aligned} \end{math}$$

This indicates a Multi-Head Attention (MSA) layer wrapped in a LayerNorm (LN) layer with a residual connection (the input to the layer gets added to the output of the layer).

We'll refer to equation 2 as the "MSA block".



**\*Left:** Figure 1 from the ViT paper with Multi-Head Attention and Norm layers as well as the residual connection (+) highlighted within the Transformer Encoder block. **Right:** Mapping the Multi-Head Self Attention (MSA) layer,

Norm layer and residual connection to their respective parts of equation 2 in the ViT paper.\*

Many layers you find in research papers are already implemented in modern deep learning frameworks such as PyTorch.

In saying this, to replicate these layers and residual connection with PyTorch code we can use:

- **Multi-Head Self Attention (MSA)** - `torch.nn.MultiheadAttention()` .
- **Norm (LN or LayerNorm)** - `torch.nn.LayerNorm()` .
- **Residual connection** - add the input to output (we'll see this later on when we create the full Transformer Encoder block in section 7.1).

## 5.1 The LayerNorm (LN) layer

[Layer Normalization](#) (`torch.nn.LayerNorm()` or Norm or LayerNorm or LN) normalizes an input over the last dimension.

You can find the formal definition of `torch.nn.LayerNorm()` in the [PyTorch documentation](#).

PyTorch's `torch.nn.LayerNorm()`'s main parameter is `normalized_shape` which we can set to be equal to the dimension size we'd like to noramlize over (in our case it'll be  $D$  or  $768$  for ViT-Base).

What does it do?

Layer Normalization helps improve training time and model generalization (ability to adapt to unseen data).

I like to think of any kind of normalization as "getting the data into a similar format" or "getting data samples into a similar distribution".

Imagine trying to walk up (or down) a set of stairs all with differing heights and lengths.

It'd take some adjustment on each step right?

And what you learn for each step wouldn't necessary help with the next one since they all differ, increasing the time it takes you to navigate the stairs.

Normalization (including Layer Normalization) is the equivalent of making all the stairs the same height and length except the stairs are your data samples.

So just like you can walk up (or down) stairs with similar heights and lengths much easier than those with unequal heights and widths, neural networks can optimize over data samples with similar distributions (similar mean and standard-deviations) easier than those with varying distributions.

## 5.2 The Multi-Head Self Attention (MSA) layer

The power of the self-attention and multi-head attention (self-attention applied multiple times) were revealed in the form of the original Transformer architecture introduced in the [\*Attention is all you need\*](#) research paper.

Originally designed for text inputs, the original self-attention mechanism takes a sequence of words and then calculates which word should pay more "attention" to another word.

In other words, in the sentence "the dog jumped over the fence", perhaps the word "dog" relates strongly to "jumped" and "fence".

This is simplified but the premise remains for images.

Since our input is a sequence of image patches rather than words, self-attention and in turn multi-head attention will calculate which patch of an image is most related to another patch, eventually forming a learned representation of an image.

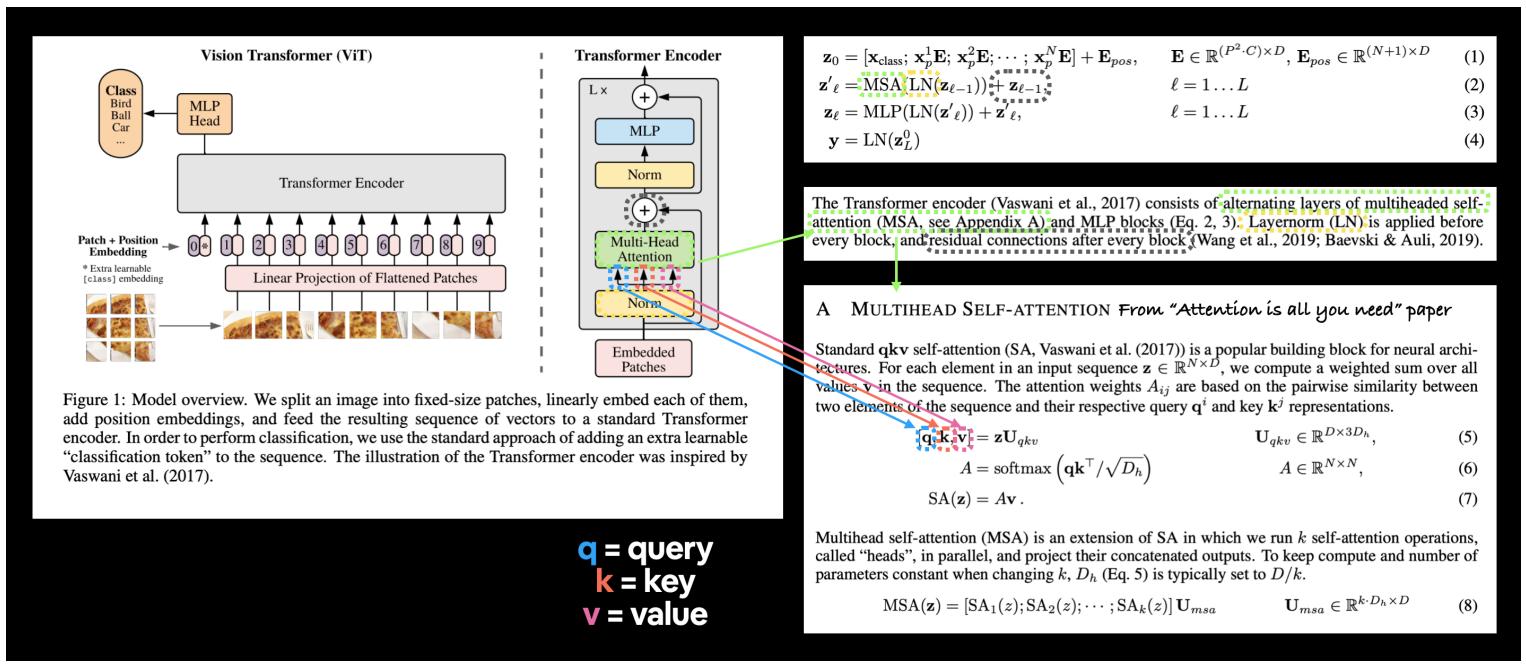
But what's most important is that the layer does this on its own given the data (we don't tell it what patterns to learn).

And if the learned representation the layers form using MSA are good, we'll see the results in our model's performance.

There are many resources online to learn more about the Transformer architecture and attention mechanism online such as Jay Alammar's wonderful [Illustrated Transformer post](#) and [Illustrated Attention post](#).

We're going to focus more on coding an existing PyTorch MSA implementation than creating our own.

However, you can find the formal definition of the ViT paper's MSA implementation is defined in Appendix A:



\*Left: Vision Transformer architecture overview from Figure 1 of the ViT paper. Right: Definitions of equation 2, section 3.1 and Appendix A of the ViT paper highlighted to reflect their respective parts in Figure 1.\*

The image above highlights the triple embedding input to the MSA layer.

This is known as **query**, **key**, **value** input or **qkv** for short which is fundamental to the self-attention mechanism.

In our case, the triple embedding input will be three versions of the output of the Norm layer, one for query, key and value.

Or three versions of our layer-normalized image patch and position embeddings created in section 4.8.

We can implement the MSA layer in PyTorch with `torch.nn.MultiheadAttention()` with the parameters:

- `embed_dim` - the embedding dimension from Table 1 (Hidden size \$D\$).
- `num_heads` - how many attention heads to use (this is where the term "multihead" comes from), this value is also in Table 1 (Heads).
- `dropout` - whether or not to apply dropout to the attention layer (according to Appendix B.1, dropout isn't used after the qkv-projections).
- `batch_first` - does our batch dimension come first? (yes it does)

### 5.3 Replicating Equation 2 with PyTorch layers

Let's put everything we've discussed about the LayerNorm (LN) and Multi-Head Attention (MSA) layers in equation 2 into practice.

To do so, we'll:

1. Create a class called `MultiheadSelfAttentionBlock` that inherits from `torch.nn.Module`.
2. Initialize the class with hyperparameters from Table 1 of the ViT paper for the ViT-Base model.
3. Create a layer normalization (LN) layer with `torch.nn.LayerNorm()` with the `normalized_shape` parameter the same as our embedding dimension ( $D$  from Table 1).
4. Create a multi-head attention (MSA) layer with the appropriate `embed_dim`, `num_heads`, `dropout` and `batch_first` parameters.
5. Create a `forward()` method for our class passing the inputs through the LN layer and MSA layer.

In [37]:



























**Note:** Unlike Figure 1, our `MultiheadSelfAttentionBlock` doesn't include a skip or residual connection (" $z_{\ell-1}$ " in equation 2), we'll include this when we create the entire Transformer Encoder later on in section 7.1.

MSABlock created!

Let's try it out by create an instance of our `MultiheadSelfAttentionBlock` and passing through the `patch_and_position_embedding` variable we created in section 4.8.

In [38]:



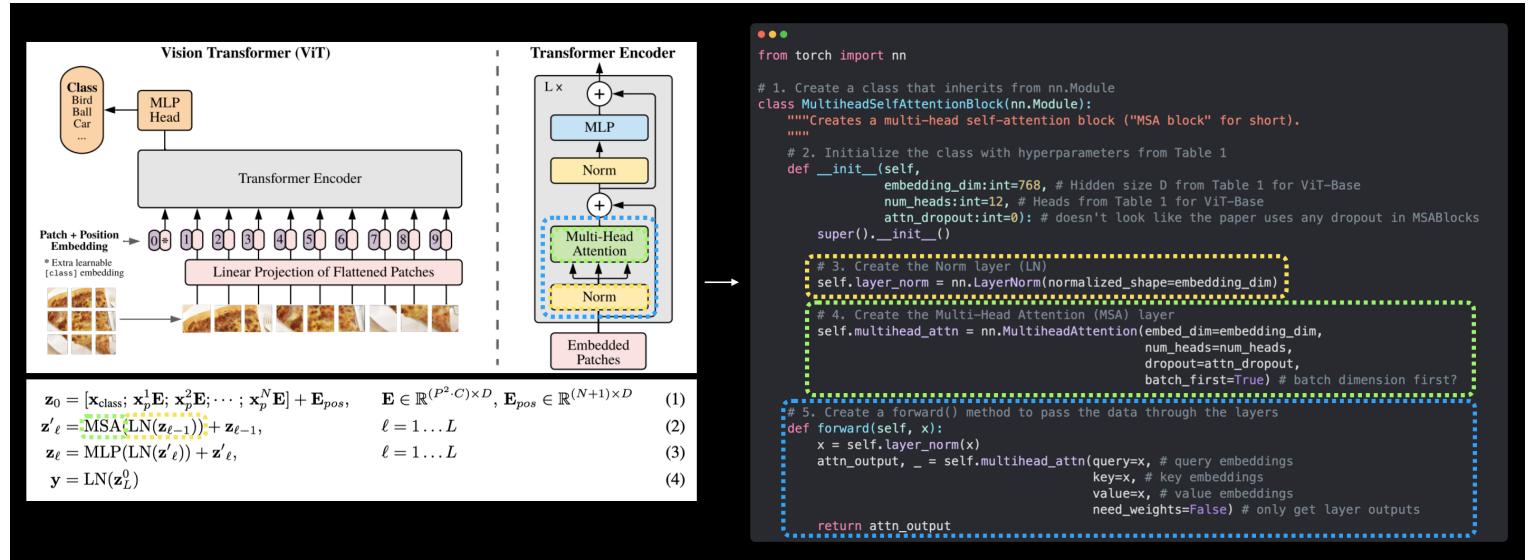


```
Input shape of MSA block: torch.Size([1, 197, 768])
Output shape MSA block: torch.Size([1, 197, 768])
```

Notice how the input and output shape of our data stays the same when it goes through the MSA block.

This doesn't mean the data doesn't change as it goes through.

You could try printing the input and output tensor to see how it changes (though this change will be across  $1 * 197 * 768$  values and could be hard to visualize).



**\*Left:** Vision Transformer architecture from Figure 1 with Multi-Head Attention and LayerNorm layers highlighted, these layers make up equation 2 from section 3.1 of the paper. **Right:** Replicating equation 2 (without the skip connection on the end) using PyTorch layers.\*

We've now officially replicated equation 2 (except for the residual connection on the end but we'll get to this in section 7.1)!

Onto the next!

## 6. Equation 3: Multilayer Perceptron (MLP)

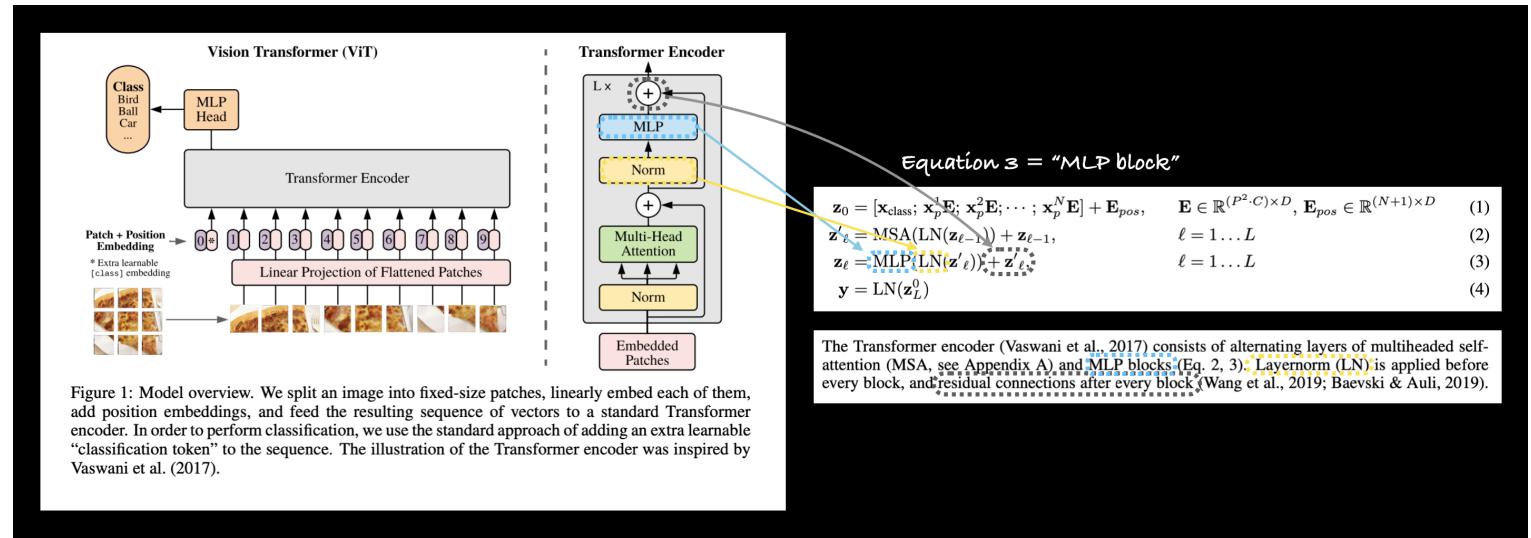
We're on a roll here!

Let's keep it going and replicate equation 3

Here MLP stands for "multilayer perceptron" and LN stands for "layer normalization" (as discussed above).

And the addition on the end is the skip/residual connection

We'll refer to equation 3 as the "MLP block" of the Transformer encoder (notice how we're continuing the trend of breaking down the architecture into smaller chunks).



**\*Left:** Figure 1 from the ViT paper with MLP and Norm layers as well as the residual connection (+) highlighted within the Transformer Encoder block. **Right:** Mapping the multilayer perceptron (MLP) layer, Norm layer (LN) and residual connection to their respective parts of equation 3 in the ViT paper.\*

### 6.1 The MLP layer(s)

The term **MLP** is quite broad as it can refer to almost any combination of *multiple* layers (hence the "multi" in multilayer perceptron).

But it generally follows the pattern of:

linear layer  $\rightarrow$  non-linear layer  $\rightarrow$  linear layer  $\rightarrow$  non-linear layer

In the the case of the ViT paper, the MLP structure is defined in section 3.1.

The MLP contains two layers with a GELU non-linearity

Where "two layers" refers to linear layers (`torch.nn.Linear()` in PyTorch) and "GELU non-linearity" is the GELU (Gaussian Error Linear Units) non-linear activation function (`torch.nn.GELU()` in PyTorch).

**Note:** A linear layer (`torch.nn.Linear()`) can sometimes also be referred to as a "dense layer" or "feedforward layer". Some papers even use all three terms to describe the same thing (as in the ViT paper).

Another sneaky detail about the MLP block doesn't appear until Appendix B.1 (Training):

Table 3 summarizes our training setups for our different models. ...Dropout, when used, is applied **after every dense layer except for the the qkv-projections and directly after adding positional- to patch embeddings.**

This means that every linear layer (or dense layer) in the MLP block has a dropout layer (`torch.nn.Dropout()` in PyTorch).

The value of which can be found in Table 3 of the ViT paper (for ViT-Base, `dropout=0.1`).

Knowing this, the structure of our MLP block will be:

```
layer norm -> linear layer -> non-linear layer -> dropout -> linear layer -> dropout
```

With hyperparameter values for the linear layers available from Table 1 (MLP size is the number of hidden units between the linear layers and hidden size \$D\$ is the output size of the MLP block).

## 6.2 Replicating Equation 3 with PyTorch layers

Let's put everything we've discussed about the LayerNorm (LN) and MLP (MSA) layers in equation 3 into practice.

To do so, we'll:

1. Create a class called `MLPBlock` that inherits from `torch.nn.Module`.
2. Initialize the class with hyperparameters from Table 1 and Table 3 of the ViT paper for the ViT-Base model.
3. Create a layer normalization (LN) layer with `torch.nn.LayerNorm()` with the `normalized_shape` parameter the same as our embedding dimension (\$D\$ from Table 1).
4. Create a sequential series of MLP layers(s) using `torch.nn.Linear()`, `torch.nn.Dropout()` and `torch.nn.GELU()` with appropriate hyperparameter values from Table 1 and Table 3.
5. Create a `forward()` method for our class passing the inputs through the LN layer and MLP layer(s).

In [39]:





















**Note:** Unlike Figure 1, our `MLPBlock()` doesn't include a skip or residual connection ( $z + \mathbf{M}z$  in equation 3), we'll include this when we create the entire Transformer encoder later on.

MLPBlock class created!

Let's try it out by create an instance of our `MLPBlock` and passing through the `patched_image_through_msa_block`

variable we created in section 5.3.

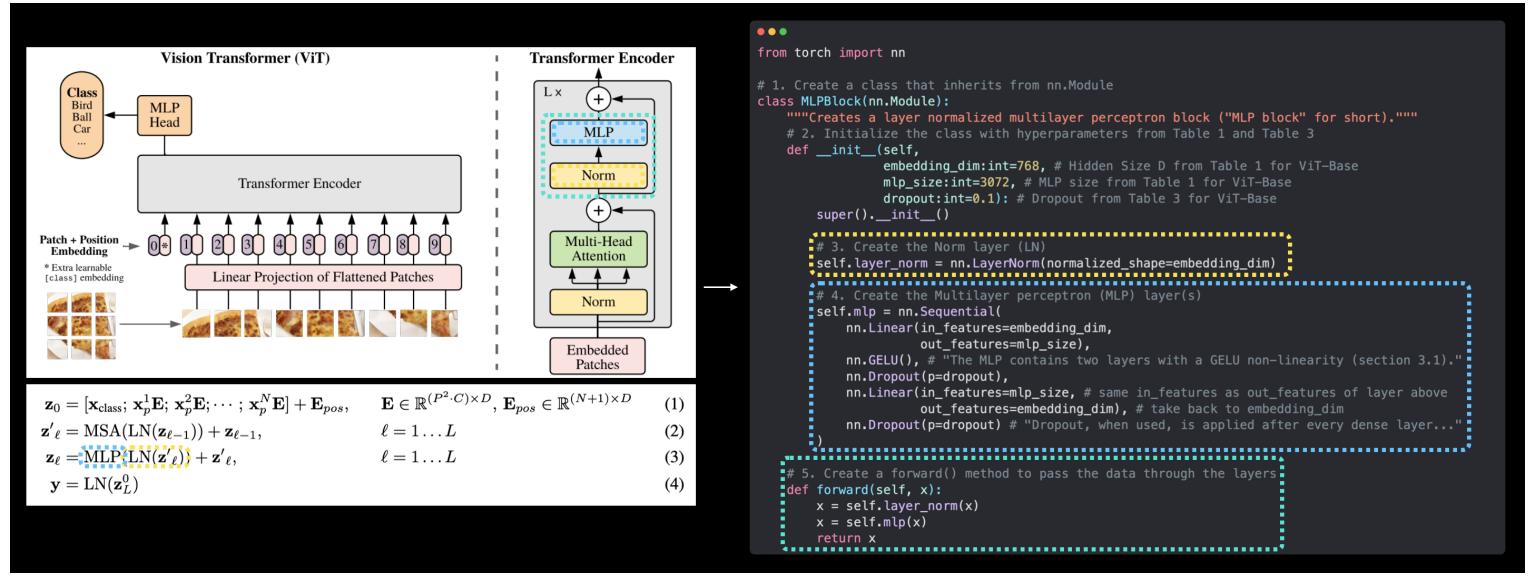
In [40]:



```
Input shape of MLP block: torch.Size([1, 197, 768])
Output shape MLP block: torch.Size([1, 197, 768])
```

Notice how the input and output shape of our data again stays the same when it goes in and out of the MLP block.

However, the shape does change when the data gets passed through the `nn.Linear()` layers within the MLP block (expanded to MLP size from Table 1 and then compressed back to Hidden size \$D\$ from Table 1).



*Left: Vision Transformer architecture from Figure 1 with MLP and Norm layers highlighted, these layers make up equation 3 from section 3.1 of the paper. Right: Replicating equation 3 (without the skip connection on the end) using PyTorch layers.*

Ho ho!

Equation 3 replicated (except for the residual connection on the end but we'll get to this in section 7.1)!

Now we've got equation's 2 and 3 in PyTorch code, let's now put them together to create the Transformer Encoder.

## 7. Create the Transformer Encoder

Time to stack together our `MultiheadSelfAttentionBlock` (equation 2) and `MLPBlock` (equation 3) and create the Transformer Encoder of the ViT architecture.

In deep learning, an "**encoder**" or "**auto encoder**" generally refers to a stack of layers that "encodes" an input (turns it into some form of numerical representation).

In our case, the Transformer Encoder will encode our patched image embedding into a learned representation using a series of alternating layers of MSA blocks and MLP blocks, as per section 3.1 of the ViT Paper:

The Transformer encoder (Vaswani et al., 2017) consists of alternating layers of multiheaded selfattention (MSA, see Appendix A) and MLP blocks (Eq. 2, 3). **Layernorm (LN) is applied before every block, and residual connections after every block** (Wang et al., 2019; Baevski & Auli, 2019).

We've created MSA and MLP blocks but what about the residual connections?

Residual connections (also called skip connections), were first introduced in the paper *Deep Residual Learning for Image Recognition* and are achieved by adding a layer(s) input to its subsequent output.

Where the subsequence output might be one or more layers later.

In the case of the ViT architecture, the residual connection means the input of the MSA block is added back to the output of the MSA block before it passes to the MLP block.

And the same thing happens with the MLP block before it goes onto the next Transformer Encoder block.

Or in pseudocode:

```
x_input -> MSA_block -> [MSA_block_output + x_input] -> MLP_block -> [MLP_block_output + MSA_block_output +
x_input] -> ...
```

What does this do?

One of the main ideas behind residual connections is that they prevent weight values and gradient updates from getting too small and thus allow deeper networks and in turn allow deeper representations to be learned.

**Note:** The iconic computer vision architecture "ResNet" is named so because of the introduction of *residual* connections. You can find many pretrained versions of ResNet architectures in [torchvision.models](#).

## 7.1 Creating a Transformer Encoder by combining our custom made layers

Enough talk, let's see this in action and make a ViT Transformer Encoder with PyTorch by combining our previously created layers.

To do so, we'll:

1. Create a class called `TransformerEncoderBlock` that inherits from `torch.nn.Module`.
2. Initialize the class with hyperparameters from Table 1 and Table 3 of the ViT paper for the ViT-Base model.
3. Instantiate a MSA block for equation 2 using our `MultiheadSelfAttentionBlock` from section 5.2 with the appropriate parameters.
4. Instantiate a MLP block for equation 3 using our `MLPBlock` from section 6.2 with the appropriate parameters.
5. Create a `forward()` method for our `TransformerEncoderBlock` class.
6. Create a residual connection for the MSA block (for equation 2).
7. Create a residual connection for the MLP block (for equation 3).

In [41]:





















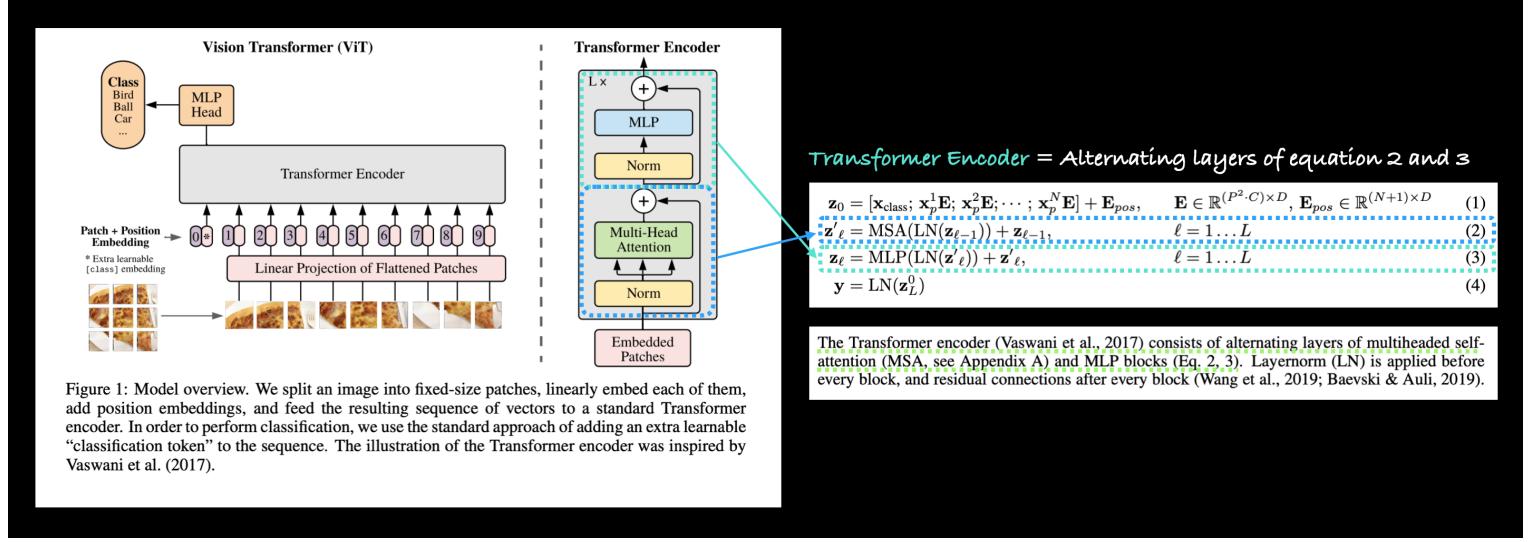






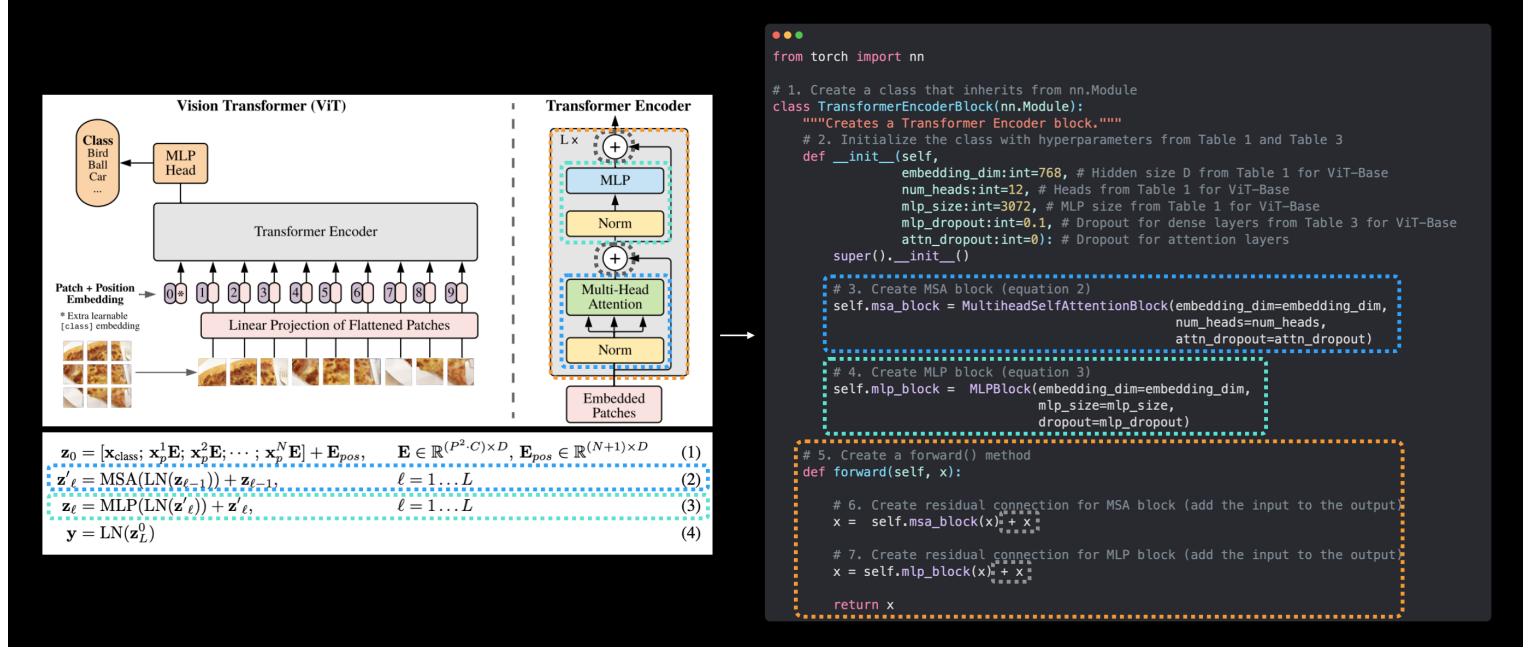
Beautiful!

Transformer Encoder block created!



\*Left: Figure 1 from the ViT paper with the Transformer Encoder of the ViT architecture highlighted. Right: Transformer Encoder mapped to equation 2 and 3 of the ViT paper, the Transformer Encoder is comprised of alternating blocks of equation 2 (Multi-Head Attention) and equation 3 (Multilayer perceptron).\*

See how we're starting to piece together the overall architecture like legos, coding one brick (or equation) at a time.



Mapping the ViT Transformer Encoder to code.

You might've noticed that Table 1 from the ViT paper has a Layers column. This refers to the number of Transformer Encoder blocks in the specific ViT architecture.

In our case, for ViT-Base, we'll be stacking together 12 of these Transformer Encoder blocks to form the backbone of our architecture (we'll get to this in section 8).

Let's get a `torchinfo.summary()` of passing an input of shape `(1, 197, 768) -> (batch_size, num_patches, embedding_dimension)` to our Transformer Encoder block.

In [42]:



Layer (type (var_name))	Input Shape	Output Shape	Param #	Trainable
<hr/>				
TransformerEncoderBlock (TransformerEncoderBlock)	[1, 197, 768]	[1, 197, 768]	--	True
└ MultiheadSelfAttentionBlock (msa_block)	[1, 197, 768]	[1, 197, 768]	--	True
└ LayerNorm (layer_norm)	[1, 197, 768]	[1, 197, 768]	1,536	True
└ MultiheadAttention (multihead_attn)	--	[1, 197, 768]	1,771,776	True
└ NonDynamicallyQuantizableLinear (out_proj)	--	--	590,592	True
└ MLPBlock (mlp_block)	[1, 197, 768]	[1, 197, 768]	--	True
└ LayerNorm (layer_norm)	[1, 197, 768]	[1, 197, 768]	1,536	True
└ Sequential (mlp)	[1, 197, 768]	[1, 197, 768]	--	True
└ Linear (0)	[1, 197, 768]	[1, 197, 3072]	2,362,368	True
└ GELU (1)	[1, 197, 3072]	[1, 197, 3072]	--	--
└ Dropout (2)	[1, 197, 3072]	[1, 197, 3072]	--	--
└ Linear (3)	[1, 197, 3072]	[1, 197, 768]	2,360,064	True
└ Dropout (4)	[1, 197, 768]	[1, 197, 768]	--	--
<hr/>				
Total params: 7,087,872				
Trainable params: 7,087,872				
Non-trainable params: 0				
Total mult-adds (M): 4.73				
<hr/>				
Input size (MB): 0.61				
Forward/backward pass size (MB): 8.47				
Params size (MB): 21.26				
Estimated Total Size (MB): 30.34				
<hr/>				

Woah! Check out all those parameters!

You can see our input changing shape as it moves through all of the various layers in the MSA block and MLP block of the Transformer Encoder block before finally returning to its original shape at the very end.

**Note:** Just because our input to the Transformer Encoder block has the same shape at the output of the block doesn't mean the values weren't manipulated, the whole goal of the Transformer Encoder block (and stacking them together) is to learn a deep representation of the input using the various layers in between.

## 7.2 Creating a Transformer Encoder with PyTorch's Transformer layers

So far we've built the components of and the Transformer Encoder layer itself ourselves.

But because of their rise in popularity and effectiveness, PyTorch now has in-built [Transformer layers as part of `torch.nn`](#).

For example, we can recreate the `TransformerEncoderBlock` we just created using `torch.nn.TransformerEncoderLayer()` and setting the same hyperparameters as above.

In [43]:

















Out[43]:

```
TransformerEncoderLayer(
    (self_attn): MultiheadAttention(
        (out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
    )
    (linear1): Linear(in_features=768, out_features=3072, bias=True)
    (dropout): Dropout(p=0.1, inplace=False)
    (linear2): Linear(in_features=3072, out_features=768, bias=True)
    (norm1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
    (norm2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
    (dropout1): Dropout(p=0.1, inplace=False)
    (dropout2): Dropout(p=0.1, inplace=False)
)
```

To inspect it further, let's get a summary with `torchinfo.summary()`.

In [44]:



```

=====
Layer (type (var_name))           Input Shape        Output Shape       Param #      Trainable
=====
TransformerEncoderLayer (TransformerEncoderLayer) [1, 197, 768]      [1, 197, 768]      3,072        True
|---LayerNorm (norm1)            [1, 197, 768]      [1, 197, 768]      (recursive)   True
|---MultiheadAttention (self_attn) [1, 197, 768]      [1, 197, 768]      2,362,368    True
|---Dropout (dropout1)          [1, 197, 768]      [1, 197, 768]      --           --
|---LayerNorm (norm2)          [1, 197, 768]      [1, 197, 768]      (recursive)   True
|---Linear (linear1)           [1, 197, 768]      [1, 197, 3072]     2,362,368    True
|---LayerNorm (norm1)          [1, 197, 768]      [1, 197, 768]      (recursive)   True
|---LayerNorm (norm2)          [1, 197, 768]      [1, 197, 768]      (recursive)   True
|---Dropout (dropout)          [1, 197, 3072]     [1, 197, 3072]     --           --
|---Dropout (dropout2)          [1, 197, 768]      [1, 197, 768]      --           --
|---Linear (linear2)           [1, 197, 3072]     [1, 197, 768]      2,360,064    True
|---Dropout (dropout2)          [1, 197, 768]      [1, 197, 768]      --           --
=====
Total params: 7,087,872
Trainable params: 7,087,872
Non-trainable params: 0
Total mult-adds (M): 4.73
=====
Input size (MB): 0.61
Forward/backward pass size (MB): 6.05
Params size (MB): 18.89
Estimated Total Size (MB): 25.55
=====
```

The output of the summary is slightly different to ours due to how `torch.nn.TransformerEncoderLayer()` constructs its layer.

But the layers it uses, number of parameters and input and output shapes are the same.

You might be thinking, "if we could create the Transformer Encoder so quickly with PyTorch layers, why did we bother reproducing equation 2 and 3?"

The answer is: practice.

Now we've replicated a series of equations and layers from a paper, if you need to change the layers and try something different you can.

But there are benefits of using the PyTorch pre-built layers, such as:

- **Less prone to errors** - Generally, if a layer makes it into the PyTorch standard library, its been tested and tried to work.
- **Potentially better performance** - As of July 2022 and PyTorch 1.12, the PyTorch implemented version of `torch.nn.TransformerEncoderLayer()` can see a speedup of more than 2x on many common workloads.

Finally, since the ViT architecture uses several Transformer Layers stacked on top of each other for the full architecture (Table 1 shows 12 Layers in the case of ViT-Base), you can do this with `torch.nn.TransformerEncoder(encoder_layer, num_layers)` where:

- `encoder_layer` - The target Transformer Encoder layer created with `torch.nn.TransformerEncoderLayer()`.
- `num_layers` - The number of Transformer Encoder layers to stack together.

## 8. Putting it all together to create ViT

Alright, alright, alright, we've come a long way!

But now it's time to do the exciting thing of putting together all of the pieces of the puzzle.

We're going to combine all of the blocks we've created to replicate the full ViT architecture.

From the patch and positional embedding to the Transformer Encoder(s) to the MLP Head.

But wait, we haven't created equation 4 yet...

$$\mathbf{y} = \text{LN}(\mathbf{z}_L^0)$$

Don't worry, we can put equation 4 into our overall ViT architecture class.

All we need is a `torch.nn.LayerNorm()` layer and a `torch.nn.Linear()` layer to convert the 0th index ( $\mathbf{z}_L^0$ ) of the Transformer Encoder logit outputs to the target number of classes we have.

To create the full architecture, we'll also need to stack a number of our `TransformerEncoderBlock`s on top of each other, we can do this by passing a list of them to `torch.nn.Sequential()` (this will make a sequential range of `TransformerEncoderBlock`s).

We'll focus on the ViT-Base hyperparameters from Table 1 but our code should be adaptable to other ViT variants.

Creating ViT will be our biggest code block yet but we can do it!

Finally, to bring our own implementation of ViT to life, let's:

1. Create a class called `ViT` that inherits from `torch.nn.Module`.

2. Initialize the class with hyperparameters from Table 1 and Table 3 of the ViT paper for the ViT-Base model.
3. Make sure the image size is divisible by the patch size (the image should be split into even patches).
4. Calculate the number of patches using the formula  $N=H \cdot W / P^2$ , where  $H$  is the image height,  $W$  is the image width and  $P$  is the patch size.
5. Create a learnable class embedding token (equation 1) as done above in section 4.6.
6. Create a learnable position embedding vector (equation 1) as done above in section 4.7.
7. Setup the embedding dropout layer as discussed in Appendix B.1 of the ViT paper.
8. Create the patch embedding layer using the `PatchEmbedding` class as above in section 4.5.
9. Create a series of Transformer Encoder blocks by passing a list of `TransformerEncoderBlock`s created in section 7.1 to `torch.nn.Sequential()` (equations 2 & 3).
10. Create the MLP head (also called classifier head or equation 4) by passing a `torch.nn.LayerNorm()` (LN) layer and a `torch.nn.Linear(out_features=num_classes)` layer (where `num_classes` is the target number of classes) linear layer to `torch.nn.Sequential()`.
11. Create a `forward()` method that accepts an input.
12. Get the batch size of the input (the first dimension of the shape).
13. Create the patching embedding using the layer created in step 8 (equation 1).
14. Create the class token embedding using the layer created in step 5 and expand it across the number of batches found in step 11 using `torch.Tensor.expand()` (equation 1).
15. Concatenate the class token embedding created in step 13 to the first dimension of the patch embedding created in step 12 using `torch.cat()` (equation 1).
16. Add the position embedding created in step 6 to the patch and class token embedding created in step 14 (equation 1).
17. Pass the patch and position embedding through the dropout layer created in step 7.
18. Pass the patch and position embedding from step 16 through the stack of Transformer Encoder layers created in step 9 (equations 2 & 3).
19. Pass index 0 of the output of the stack of Transformer Encoder layers from step 17 through the classifier head created in step 10 (equation 4).
20. Dance and shout woohoo!!! We just built a Vision Transformer!

You ready?

Let's go.

In [45]:







































































1. Woohoo!!! We just built a vision transformer!

What an effort!

Slowly but surely we created layers and blocks, inputs and outputs and put them all together to build our own ViT!

Let's create a quick demo to showcase what's happening with the class token embedding being expanded over the batch dimensions.

In [46]:





```
Shape of class token embedding single: torch.Size([1, 1, 768])
Shape of class token embedding expanded: torch.Size([32, 1, 768])
```

Notice how the first dimension gets expanded to the batch size and the other dimensions stay the same (because they're inferred by the "`-1`" dimensions in `.expand(batch_size, -1, -1)`).

Alright time to test out `ViT()` class.

Let's create a random tensor in the same shape as a single image, pass to an instance of `ViT` and see what happens.

In [47]:



Out[47]:

```
tensor([[-0.2377,  0.7360,  1.2137]], grad_fn=<AddmmBackward0>)
```

Outstanding!

It looks like our random image tensor made it all the way through our ViT architecture and it's outputting three logit

values (one for each class).

And because our `ViT` class has plenty of parameters we could customize the `img_size`, `patch_size` or `num_classes` if we wanted to.

## 8.1 Getting a visual summary of our ViT model

We handcrafted our own version of the ViT architecture and seen that a random image tensor can flow all the way through it.

How about we use `torchinfo.summary()` to get a visual overview of the input and output shapes of all the layers in our model?

**Note:** The ViT paper states the use of a batch size of 4096 for training, however, this requires a fair bit of CPU/GPU compute memory to handle (the larger the batch size the more memory required). So to make sure we don't get memory errors, we'll stick with a batch size of 32. You could always increase this later if you have access to hardware with more memory.

In [48]:





Layer (type (var_name))	Input Shape	Output Shape	Param #	Trainable
Vit (ViT)	[32, 3, 224, 224]	[32, 3]	152,064	True
Dropout (embedding_dropout)	[32, 197, 768]	[32, 197, 768]	--	--
PatchEmbedding (patch_embedding)	[32, 3, 224, 224]	[32, 196, 768]	--	True
└Conv2d (patcher)	[32, 3, 224, 224]	[32, 768, 14, 14]	590,592	True
└Flatten (flatten)	[32, 768, 14, 14]	[32, 768, 196]	--	--
Dropout (embedding_dropout)	[32, 197, 768]	[32, 197, 768]	--	--
Sequential (transformer_encoderdecoder)	[32, 197, 768]	[32, 197, 768]	--	True
└TransformerEncoderBlock (0)	[32, 197, 768]	[32, 197, 768]	--	True
└MultiheadSelfAttentionBlock (msa_block)	[32, 197, 768]	[32, 197, 768]	2,363,904	True
└MLPBlock (mlp_block)	[32, 197, 768]	[32, 197, 768]	4,723,968	True
└TransformerEncoderBlock (1)	[32, 197, 768]	[32, 197, 768]	--	True
└MultiheadSelfAttentionBlock (msa_block)	[32, 197, 768]	[32, 197, 768]	2,363,904	True
└MLPBlock (mlp_block)	[32, 197, 768]	[32, 197, 768]	4,723,968	True
└TransformerEncoderBlock (2)	[32, 197, 768]	[32, 197, 768]	--	True
└MultiheadSelfAttentionBlock (msa_block)	[32, 197, 768]	[32, 197, 768]	2,363,904	True
└MLPBlock (mlp_block)	[32, 197, 768]	[32, 197, 768]	4,723,968	True
└TransformerEncoderBlock (3)	[32, 197, 768]	[32, 197, 768]	--	True
└MultiheadSelfAttentionBlock (msa_block)	[32, 197, 768]	[32, 197, 768]	2,363,904	True
└MLPBlock (mlp_block)	[32, 197, 768]	[32, 197, 768]	4,723,968	True
└TransformerEncoderBlock (4)	[32, 197, 768]	[32, 197, 768]	--	True
└MultiheadSelfAttentionBlock (msa_block)	[32, 197, 768]	[32, 197, 768]	2,363,904	True
└MLPBlock (mlp_block)	[32, 197, 768]	[32, 197, 768]	4,723,968	True
└TransformerEncoderBlock (5)	[32, 197, 768]	[32, 197, 768]	--	True
└MultiheadSelfAttentionBlock (msa_block)	[32, 197, 768]	[32, 197, 768]	2,363,904	True
└MLPBlock (mlp_block)	[32, 197, 768]	[32, 197, 768]	4,723,968	True
└TransformerEncoderBlock (6)	[32, 197, 768]	[32, 197, 768]	--	True
└MultiheadSelfAttentionBlock (msa_block)	[32, 197, 768]	[32, 197, 768]	2,363,904	True
└MLPBlock (mlp_block)	[32, 197, 768]	[32, 197, 768]	4,723,968	True
└TransformerEncoderBlock (7)	[32, 197, 768]	[32, 197, 768]	--	True
└MultiheadSelfAttentionBlock (msa_block)	[32, 197, 768]	[32, 197, 768]	2,363,904	True
└MLPBlock (mlp_block)	[32, 197, 768]	[32, 197, 768]	4,723,968	True
└TransformerEncoderBlock (8)	[32, 197, 768]	[32, 197, 768]	--	True
└MultiheadSelfAttentionBlock (msa_block)	[32, 197, 768]	[32, 197, 768]	2,363,904	True
└MLPBlock (mlp_block)	[32, 197, 768]	[32, 197, 768]	4,723,968	True
└TransformerEncoderBlock (9)	[32, 197, 768]	[32, 197, 768]	--	True
└MultiheadSelfAttentionBlock (msa_block)	[32, 197, 768]	[32, 197, 768]	2,363,904	True
└MLPBlock (mlp_block)	[32, 197, 768]	[32, 197, 768]	4,723,968	True
└TransformerEncoderBlock (10)	[32, 197, 768]	[32, 197, 768]	--	True
└MultiheadSelfAttentionBlock (msa_block)	[32, 197, 768]	[32, 197, 768]	2,363,904	True
└MLPBlock (mlp_block)	[32, 197, 768]	[32, 197, 768]	4,723,968	True
└TransformerEncoderBlock (11)	[32, 197, 768]	[32, 197, 768]	--	True
└MultiheadSelfAttentionBlock (msa_block)	[32, 197, 768]	[32, 197, 768]	2,363,904	True
└MLPBlock (mlp_block)	[32, 197, 768]	[32, 197, 768]	4,723,968	True
Sequential (classifier)	[32, 768]	[32, 3]	--	True
└LayerNorm (0)	[32, 768]	[32, 768]	1,536	True
└Linear (1)	[32, 768]	[32, 3]	2,307	True
Total params: 85,800,963				
Trainable params: 85,800,963				
Non-trainable params: 0				
Total mult-adds (G): 5.52				
====				
Input size (MB): 19.27				
Forward/backward pass size (MB): 3292.20				
Params size (MB): 257.55				
Estimated Total Size (MB): 3569.02				
====				

Now those are some nice looking layers!

Checkout the total number of parameters too, 85,800,963, our biggest model yet!

The number is very close to PyTorch's pretrained ViT-Base with patch size 16 at `torch.vision.models.vit_b_16()` with 86,567,656 total parameters (though this number of parameters is for the 1000 classes in ImageNet).

**Exercise:** Try changing the `num_classes` parameter of our `vit()` model to 1000 and then creating another summary with `torchinfo.summary()` and see if the number of parameters lines up between our code and `torchvision.models.vit_b_16()`.

## 9. Setting up training code for our ViT model

Ok time for the easy part.

Training!

Why easy?

Because we've got most of what we need ready to go, from our model (`vit`) to our DataLoaders (`train_dataloader`, `test_dataloader`) to the training functions we created in [05. PyTorch Going Modular section 4](#).

To train our model we can import the `train()` function from `going_modular.going_modular.engine`.

All we need is a loss function and an optimizer.

### 9.1 Creating an optimizer

Searching the ViT paper for "optimizer", section 4.1 on Training & Fine-tuning states:

**Training & Fine-tuning.** We train all models, including ResNets, using Adam (Kingma & Ba, 2015 ) with  $\beta_1=0.9, \beta_2=0.999$ , a batch size of 4096 and apply a high weight decay of  $0.1$ , which we found to be useful for transfer of all models (Appendix D.1 shows that, in contrast to common practices, Adam works slightly better than SGD for ResNets in our setting).

So we can see they chose to use the "Adam" optimizer (`torch.optim.Adam()`) rather than SGD (stochastic gradient descent, `torch.optim.SGD()`).

The authors set Adam's  $\beta$  (beta) values to  $\beta_1=0.9, \beta_2=0.999$ , these are the default values for the `betas` parameter in `torch.optim.Adam(betas=(0.9, 0.999))`.

They also state the use of **weight decay** (slowly reducing the values of the weights during optimization to prevent overfitting), we can set this with the `weight_decay` parameter in `torch.optim.Adam(weight_decay=0.1)`.

We'll set the learning rate of the optimizer to 0.1 as per Table 3.

And as discussed previously, we're going to use a lower batch size than 4096 due to hardware limitations (if you have a large GPU, feel free to increase this).

## 9.2 Creating a loss function

Strangely, searching the ViT paper for "loss" or "loss function" or "criterion" returns no results.

However, since the target problem we're working with is multi-class classification (the same for the ViT paper), we'll use `torch.nn.CrossEntropyLoss()`.

## 9.3 Training our ViT model

Okay, now we know what optimizer and loss function we're going to use, let's setup the training code for training our ViT.

We'll start by importing the `engine.py` script from `going_modular.going_modular` then we'll setup the optimizer and loss function and finally we'll use the `train()` function from `engine.py` to train our ViT model for 10 epochs (we're using a smaller number of epochs than the ViT paper to make sure everything works).

In [49]:















```
0% |          | 0/10 [00:00<?, ?it/s]
Epoch: 1 | train_loss: 4.8759 | train_acc: 0.2891 | test_loss: 1.0465 | test_acc: 0.5417
Epoch: 2 | train_loss: 1.5900 | train_acc: 0.2617 | test_loss: 1.5876 | test_acc: 0.1979
Epoch: 3 | train_loss: 1.4644 | train_acc: 0.2617 | test_loss: 1.2738 | test_acc: 0.1979
Epoch: 4 | train_loss: 1.3159 | train_acc: 0.2773 | test_loss: 1.7498 | test_acc: 0.1979
Epoch: 5 | train_loss: 1.3114 | train_acc: 0.3008 | test_loss: 1.7444 | test_acc: 0.2604
Epoch: 6 | train_loss: 1.2445 | train_acc: 0.3008 | test_loss: 1.9704 | test_acc: 0.1979
Epoch: 7 | train_loss: 1.2050 | train_acc: 0.3984 | test_loss: 3.5480 | test_acc: 0.1979
Epoch: 8 | train_loss: 1.4368 | train_acc: 0.4258 | test_loss: 1.8324 | test_acc: 0.2604
Epoch: 9 | train_loss: 1.5757 | train_acc: 0.2344 | test_loss: 1.2848 | test_acc: 0.5417
Epoch: 10 | train_loss: 1.4658 | train_acc: 0.4023 | test_loss: 1.2389 | test_acc: 0.2604
```

Wonderful!

Our ViT model has come to life!

Though the results on our pizza, steak and sushi dataset don't look too good.

Perhaps it's because we're missing a few things?

## 9.4 What our training setup is missing

The original ViT architecture achieves good results on several image classification benchmarks (on par or better than many state-of-the-art results when it was released).

However, our results (so far) aren't as good.

There's a few reasons this could be but the main one is scale.

The original ViT paper uses a far larger amount of data than ours (in deep learning, more data is generally always a good thing) and a longer training schedule (see Table 3).

Hyperparameter value	ViT Paper	Our implementation
Number of training images	1.3M (ImageNet-1k), 14M (ImageNet-21k), 303M (JFT)	225
Epochs	7 (for largest dataset), 90, 300 (for ImageNet)	10
Batch size	4096	32
Learning rate warmup	10k steps (Table 3)	None
Learning rate decay	Linear/Cosine (Table 3)	None
Gradient clipping	Global norm 1 (Table 3)	None

Even though our ViT architecture is the same as the paper, the results from the ViT paper were achieved using far more data and a more elaborate training scheme than ours.

Because of the size of the ViT architecture and its high number of parameters (increased learning capabilities), and amount of data it uses (increased learning opportunities), many of the techniques used in the ViT paper training scheme such as learning rate warmup, learning rate decay and gradient clipping are specifically designed to [prevent overfitting](#) (regularization).

**Note:** For any technique you're unsure of, you can often quickly find an example by searching "pytorch TECHNIQUE NAME", for example, say you wanted to learn about learning rate warmup and what it does, you could search "pytorch learning rate warmup".

Good news is, there are many pretrained ViT models (using vast amounts of data) available online, we'll see one in action in section 10.

## 9.5 Plot the loss curves of our ViT model

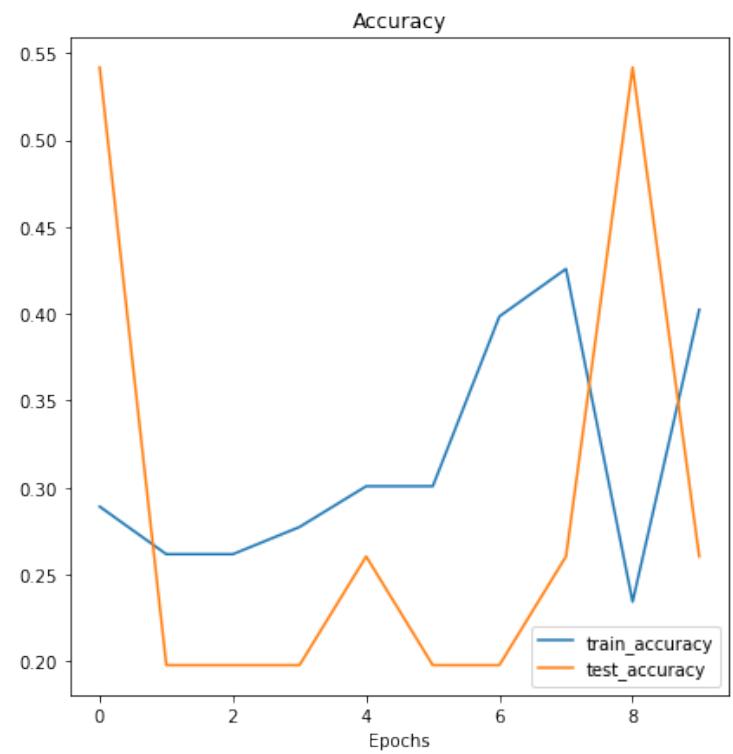
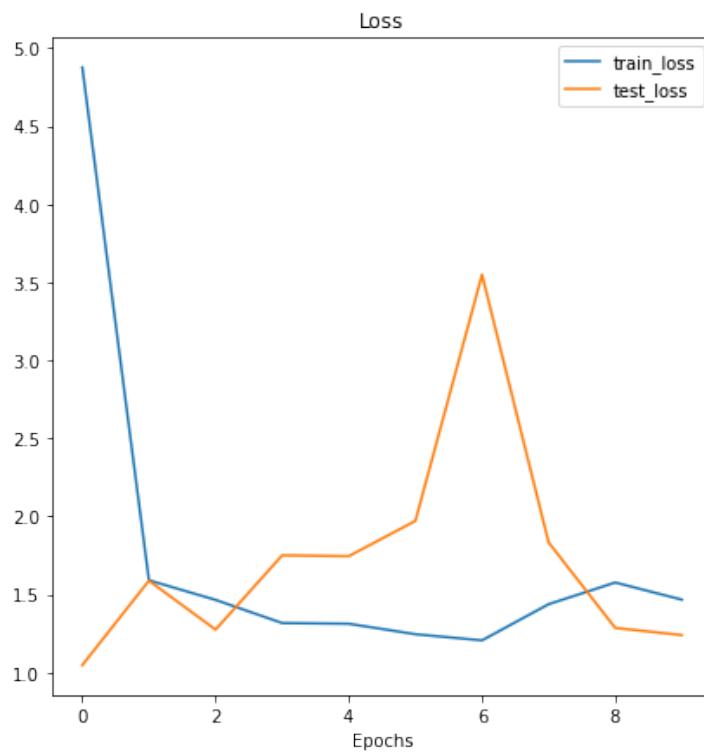
We've trained our ViT model and seen the results as numbers on a page.

But let's now follow the data explorer's motto of *visualize, visualize, visualize!*

And one of the best things to visualize for a model is its loss curves.

To check out our ViT model's loss curves, we can use the `plot_loss_curves` function from `helper_functions.py` we created in [04. PyTorch Custom Datasets section 7.8](#).

In [50]:



Hmm, it looks like our model's loss curves are all over the place.

At least the loss looks like it's heading the right direction but the accuracy curves don't really show much promise.

These results are likely because of the difference in data resources and training regime of our ViT model versus the ViT paper.

It seems our model is [severely underfitting](#) (not achieving the results we'd like it to).

How about we see if we can fix that by bringing in a pretrained ViT model?

## 10. Using a pretrained ViT from `torchvision.models` on the same dataset

We've discussed the benefits of using pretrained models in [06. PyTorch Transfer Learning](#).

But since we've now trained our own ViT from scratch and achieved less than optimal results, the benefits of transfer learning (using a pretrained model) really shine.

### 10.1 Why use a pretrained model?

An important note on many modern machine learning research papers is that much of the results are obtained with large datasets and vast compute resources.

And in modern day machine learning, the original fully trained ViT would likely not be considered a "super large" training setup (models are continually getting bigger and bigger).

Reading the ViT paper section 4.2:

Finally, the ViT-L/16 model pre-trained on the public ImageNet-21k dataset performs well on most datasets too, while taking fewer resources to pre-train: it could be trained using a standard cloud TPUv3 with 8 cores in approximately **30 days**.

As of July 2022, the [price for renting a TPUv3](#) (Tensor Processing Unit version 3) with 8 cores on Google Cloud is \$8 USD per hour.

To rent one for 30 straight days would cost **\$5,760 USD**.

This cost (monetary and time) may be viable for some larger research teams or enterprises but for many people it's not.

So having a pretrained model available through resources like [torchvision.models](#), the [timm](#) ([Torch Image Models](#)) [library](#), the [HuggingFace Hub](#) or even from the authors of the papers themselves (there's a growing trend for machine learning researchers to release the code and pretrained models from their research papers, I'm a big fan of this trend, many of these resources can be found on [Paperswithcode.com](#)).

If you're focused on leveraging the benefits of a specific model architecture rather than creating your custom architecture, I'd highly recommend using a pretrained model.

## 10.2 Getting a pretrained ViT model and creating a feature extractor

We can get a pretrained ViT model from [torchvision.models](#).

We'll go from the top by first making sure we've got the right versions of [torch](#) and [torchvision](#).

**Note:** The following code requires [torch v0.12+](#) and [torchvision v0.13+](#) to use the latest [torchvision](#) model weights API.

In [51]:

```
1.12.0+cu102  
0.13.0+cu102
```

Then we'll setup device-agnostic code.

In [52]:

```
'cuda'
```

Finally, we'll get the pretrained ViT-Base with patch size 16 from `torchvision.models` and prepare it for our FoodVision Mini use case by turning it into a feature extractor transfer learning model.

Out[52]:

Specifically, we'll:

1. Get the pretrained weights for ViT-Base trained on ImageNet-1k from `torchvision.models.ViT_B_16_Weights.DEFAULT` (`DEFAULT` stands for best available).
2. Setup a ViT model instance via `torchvision.models.vit_b_16`, pass it the pretrained weights step 1 and send it to the target device.
3. Freeze all of the parameters in the base ViT model created in step 2 by setting their `requires_grad` attribute to `False`.
4. Update the classifier head of the ViT model created in step 2 to suit our own problem by changing the number of `out_features` to our number of classes (pizza, steak, sushi).

We covered steps like this in 06. PyTorch Transfer Learning [section 3.2: Setting up a pretrained model](#) and [section 3.4: Freezing the base model and changing the output layer to suit our needs](#).

In [53]:





Pretrained ViT feature extractor model created!

Let's now check it out by printing a `torchinfo.summary()`.

In [54]:





Layer (type (var_name))	Input Shape	Output Shape	Param #	Trainable
VisionTransformer (VisionTransformer)	[32, 3, 224, 224]	[32, 3]	768	Partial
└Conv2d (conv_proj)	[32, 3, 224, 224]	[32, 768, 14, 14]	(590,592)	False
└Encoder (encoder)	[32, 197, 768]	[32, 197, 768]	151,296	False
└Dropout (dropout)	[32, 197, 768]	[32, 197, 768]	--	--
└Sequential (layers)	[32, 197, 768]	[32, 197, 768]	--	False
└EncoderBlock (encoder_layer_0)	[32, 197, 768]	[32, 197, 768]	(7,087,872)	False
└EncoderBlock (encoder_layer_1)	[32, 197, 768]	[32, 197, 768]	(7,087,872)	False
└EncoderBlock (encoder_layer_2)	[32, 197, 768]	[32, 197, 768]	(7,087,872)	False
└EncoderBlock (encoder_layer_3)	[32, 197, 768]	[32, 197, 768]	(7,087,872)	False
└EncoderBlock (encoder_layer_4)	[32, 197, 768]	[32, 197, 768]	(7,087,872)	False
└EncoderBlock (encoder_layer_5)	[32, 197, 768]	[32, 197, 768]	(7,087,872)	False
└EncoderBlock (encoder_layer_6)	[32, 197, 768]	[32, 197, 768]	(7,087,872)	False
└EncoderBlock (encoder_layer_7)	[32, 197, 768]	[32, 197, 768]	(7,087,872)	False
└EncoderBlock (encoder_layer_8)	[32, 197, 768]	[32, 197, 768]	(7,087,872)	False
└EncoderBlock (encoder_layer_9)	[32, 197, 768]	[32, 197, 768]	(7,087,872)	False
└EncoderBlock (encoder_layer_10)	[32, 197, 768]	[32, 197, 768]	(7,087,872)	False
└EncoderBlock (encoder_layer_11)	[32, 197, 768]	[32, 197, 768]	(7,087,872)	False
└LayerNorm (ln)	[32, 197, 768]	[32, 197, 768]	(1,536)	False
└Linear (heads)	[32, 768]	[32, 3]	2,307	True
Total params: 85,800,963				
Trainable params: 2,307				
Non-trainable params: 85,798,656				
Total mult-adds (G): 5.52				
Input size (MB): 19.27				
Forward/backward pass size (MB): 3330.74				
Params size (MB): 257.55				
Estimated Total Size (MB): 3607.55				

Woohoo!

Notice how only the output layer is trainable, where as, all of the rest of the layers are untrainable (frozen).

And the total number of parameters, 85,800,963, is the same as our custom made ViT model above.

But the number of trainable parameters for `pretrained_vit` is much, much lower than our custom `vit` at only 2,307 compared to 85,800,963 (in our custom `vit`, since we're training from scratch, all parameters are trainable).

This means the pretrained model should train a lot faster, we could potentially even use a larger batch size since less parameter updates are going to be taking up memory.

## 10.3 Preparing data for the pretrained ViT model

We downloaded and created DataLoaders for our own ViT model back in section 2.

So we don't necessarily need to do it again.

But in the name of practice, let's download some image data (pizza, steak and sushi images for Food Vision Mini), setup train and test directories and then transform the images into tensors and DataLoaders.

We can download pizza, steak and sushi images from the course GitHub and the `download_data()` function we creating in [07. PyTorch Experiment Tracking section 1](#).

In [55]:

```
[INFO] data/pizza_steak_sushi directory exists, skipping download.
```

Out[55]:

```
PosixPath('data/pizza_steak_sushi')
```

And now we'll setup the training and test directory paths.

In [56]:

Out[56]:

```
(PosixPath('data/pizza_steak_sushi/train'),  
 PosixPath('data/pizza_steak_sushi/test'))
```

Finally, we'll transform our images into tensors and turn the tensors into DataLoaders.

Since we're using a pretrained model from `torchvision.models` we can call the `transforms()` method on it to get its required transforms.

Remember, if you're going to use a pretrained model, it's generally important to **ensure your own custom data is transformed/formatted in the same way the data the original model was trained on**.

We covered this method of "automatic" transform creation in [06. PyTorch Transfer Learning section 2.2](#).

In [57]:

```
ImageClassification(  
    crop_size=[224]  
    resize_size=[256]  
    mean=[0.485, 0.456, 0.406]  
    std=[0.229, 0.224, 0.225]
```

```
    interpolation=InterpolationMode.BILINEAR  
)
```

And now we've got transforms ready, we can turn our images into DataLoaders using the `data_setup.create_dataloaders()` method we created in [05. PyTorch Going Modular section 2](#).

Since we're using a feature extractor model (less trainable parameters), we could increase the batch size to a higher value (if we set it to 1024, we'd be mimicing an improvement found in [\*Better plain ViT baselines for ImageNet-1k\*](#), a paper which improves upon the original ViT paper and suggested extra reading). But since we only have ~200 training samples total, we'll stick with 32.

In [58]:













## 10.4 Train feature extractor ViT model

Feature extractor model ready, DataLoaders ready, time to train!

As before we'll use the Adam optimizer (`torch.optim.Adam()`) with a learning rate of `1e-3` and `torch.nn.CrossEntropyLoss()` as the loss function.

Our `engine.train()` function we created in [05. PyTorch Going Modular section 4](#) will take care of the rest.

In [59]:











```
0% | 0/10 [00:00<?, ?it/s]
Epoch: 1 | train_loss: 0.7665 | train_acc: 0.7227 | test_loss: 0.5432 | test_acc: 0.8665
Epoch: 2 | train_loss: 0.3428 | train_acc: 0.9453 | test_loss: 0.3263 | test_acc: 0.8977
Epoch: 3 | train_loss: 0.2064 | train_acc: 0.9531 | test_loss: 0.2707 | test_acc: 0.9081
Epoch: 4 | train_loss: 0.1556 | train_acc: 0.9570 | test_loss: 0.2422 | test_acc: 0.9081
Epoch: 5 | train_loss: 0.1246 | train_acc: 0.9727 | test_loss: 0.2279 | test_acc: 0.8977
Epoch: 6 | train_loss: 0.1216 | train_acc: 0.9766 | test_loss: 0.2129 | test_acc: 0.9280
Epoch: 7 | train_loss: 0.0938 | train_acc: 0.9766 | test_loss: 0.2352 | test_acc: 0.8883
Epoch: 8 | train_loss: 0.0797 | train_acc: 0.9844 | test_loss: 0.2281 | test_acc: 0.8778
Epoch: 9 | train_loss: 0.1098 | train_acc: 0.9883 | test_loss: 0.2074 | test_acc: 0.9384
Epoch: 10 | train_loss: 0.0650 | train_acc: 0.9883 | test_loss: 0.1804 | test_acc: 0.9176
```

Holy cow!

Looks like our pretrained ViT feature extractor performed far better than our custom ViT model trained from scratch (in the same amount of time).

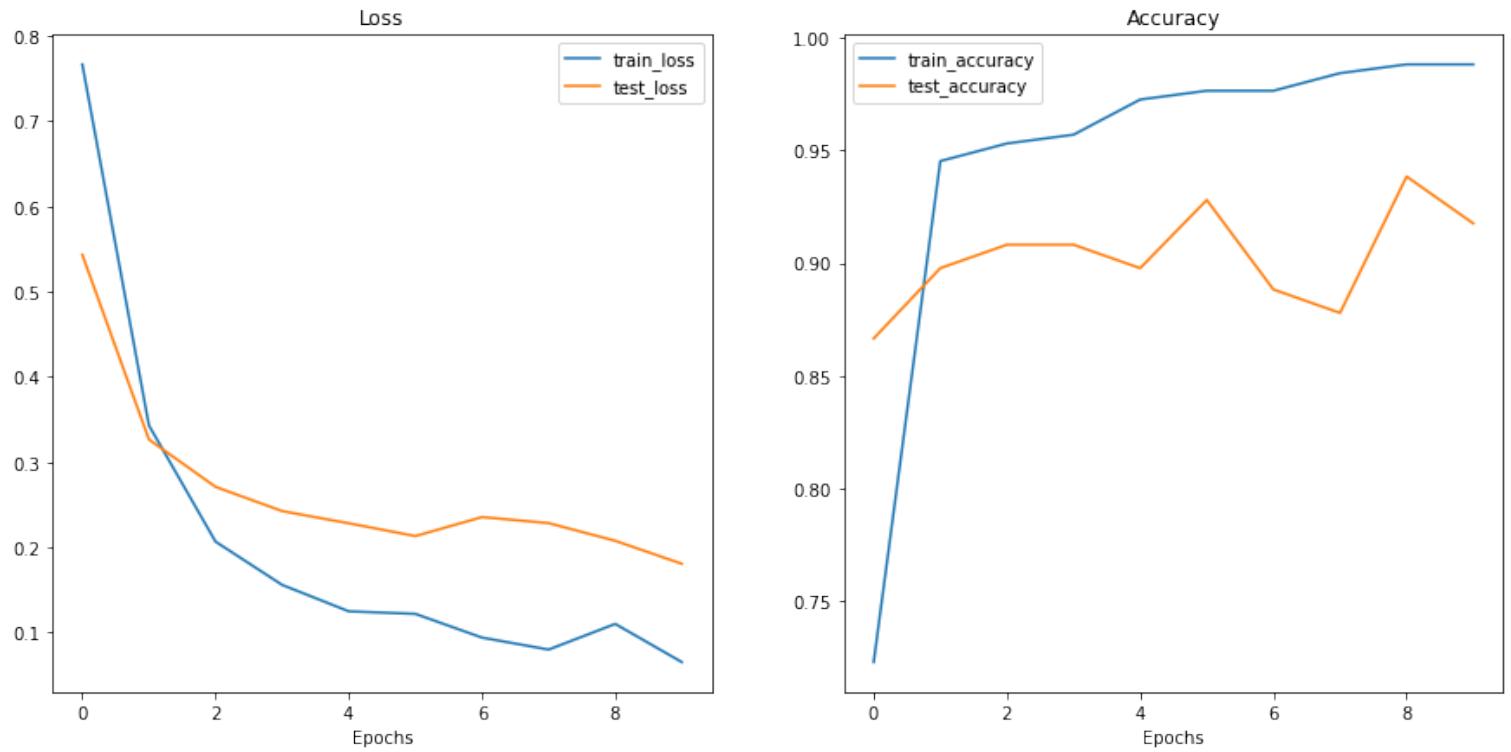
Let's get visual.

## 10.5 Plot feature extractor ViT model loss curves

Our pretrained ViT feature model numbers look good on the training and test sets.

How do the loss curves look?

In [60]:



Woah!

Those are some close to textbook looking (really good) loss curves (check out 04. PyTorch Custom Datasets section 8 for what an ideal loss curve should look like).

That's the power of transfer learning!

We managed to get outstanding results with the *same* model architecture, except our custom implementation was trained from scratch (worse performance) and this feature extractor model has the power of pretrained weights from ImageNet behind it.

What do you think?

Would our feature extractor model improve more if you kept training it?

## 10.6 Save feature extractor ViT model and check file size

It looks like our ViT feature extractor model is performing quite well for our Food Vision Mini problem.

Perhaps we might want to try deploying it and see how it goes in production (in this case, deploying means putting our trained model in an application someone could use, say taking photos on their smartphone of food and seeing if our model thinks its pizza, steak or sushi).

To do so we can first save our model with the `utils.save_model()` function we created in [05. PyTorch Going Modular section 5](#).

In [61]:

```
[INFO] Saving model to: models/08_pretrained_vit_feature_extractor_pizza_steak_sushi.pth
```

And since we're thinking about deploying this model, it'd be good to know the size of it (in megabytes or MB).

Since we want our Food Vision Mini application to run fast, generally a smaller model with good performance will be better than a larger model with great performance.

We can check the size of our model in bytes using the `st_size` attribute of Python's `pathlib.Path().stat()` method whilst passing it our model's filepath name.

We can then scale the size in bytes to megabytes.

In [62]:



```
Pretrained ViT feature extractor model size: 327 MB
```

Hmm, looks like our ViT feature extractor model for Food Vision Mini turned out to be about 327 MB in size.

How does this compare to the EffNetB2 feature extractor model in [07. PyTorch Experiment Tracking section 9?](#)

Model	Model size (MB)	Test loss	Test accuracy
EffNetB2 feature extractor <sup>^</sup>	29	~0.3906	~0.9384
ViT feature extractor	327	~0.1084	~0.9384

**Note:** <sup>^</sup> the EffNetB2 model in reference was trained with 20% of pizza, steak and sushi data (double the amount of images) rather than the ViT feature extractor which was trained with 10% of pizza, steak and sushi data. An exercise would be to train the ViT feature extractor model on the same amount of data and see how much the results improve.

The EffNetB2 model is ~11x smaller than the ViT model with similar results for test loss and accuracy.

However, the ViT model's results may improve more when trained with the same data (20% pizza, steak and sushi data).

But in terms of deployment, if we were comparing these two models, something we'd need to consider is whether the extra accuracy from the ViT model is worth the ~11x increase in model size?

Perhaps such a large model would take longer to load/run and wouldn't provide as good an experience as EffNetB2 which performs similarly but at a much reduced size.

## 11. Make predictions on a custom image

And finally, we'll finish with the ultimate test, predicting on our own custom data.

Let's download the pizza dad image (a photo of my dad eating pizza) and use our ViT feature extractor to predict on it.

To do we, let's can use the `pred_and_plot()` function we created in [06. PyTorch Transfer Learning section 6](#), for convenience, I saved this function to [going\\_modular.going\\_modular.predictions.py](#) on the course GitHub.

In [63]:







```
data/pizza_steak_sushi/04-pizza-dad.jpeg already exists, skipping download.
```

Pred: pizza | Prob: 0.988



Two thumbs up!

Congratulations!

We've gone all the way from research paper to usable model code on our own custom images!

## Main takeaways

- With the explosion of machine learning, new research papers detailing advancements come out every day. And it's impossible to keep up with it *all* but you can narrow things down to your own use case, such as what we did here, replicating a computer vision paper for FoodVision Mini.
- Machine learning research papers are often contain months of research by teams of smart people compressed into a few pages (so teasing out all the details and replicating the paper in full can be a bit of challenge).
- The goal of paper replicating is to turn machine learning research papers (text and math) into usable code.
  - With this being said, many machine learning research teams are starting to publish code with their papers and one of the best places to see this is at [Paperswithcode.com](https://paperswithcode.com)
- Breaking a machine learning research paper into inputs and outputs (what goes in and out of each layer/block/model?) and layers (how does each layer manipulate the input?) and blocks (a collection of layers) and replicating each part step by step (like we've done in this notebook) can be very helpful for understanding.
- Pretrained models are available for many state of the art model architectures and with the power of transfer learning, these often perform *very* well with little data.
- Larger models generally perform better but have a larger footprint too (they take up more storage space and can take longer to perform inference).
  - A big question is: deployment wise, is the extra performance of a larger model worth it/aligned with the use case?

# Exercises

**Note:** These exercises expect the use of `torchvision v0.13+` (released July 2022), previous versions may work but will likely have errors.

All of the exercises are focused on practicing the code above.

You should be able to complete them by referencing each section or by following the resource(s) linked.

All exercises should be completed using [device-agnostic code](#).

## Resources:

- [Exercise template notebook for 08](#).
- [Example solutions notebook for 08](#) (try the exercises *before* looking at this).
  - See a live [video walkthrough of the solutions on YouTube](#) (errors and all).

1. Replicate the ViT architecture we created with in-built PyTorch transformer layers.
  - You'll want to look into replacing our `TransformerEncoderBlock()` class with `torch.nn.TransformerEncoderLayer()` (these contain the same layers as our custom blocks).
  - You can stack `torch.nn.TransformerEncoderLayer()`'s on top of each other with `torch.nn.TransformerEncoder()`.
2. Turn the custom ViT architecture we created into a Python script, for example, `vit.py`.
  - You should be able to import an entire ViT model using something like `from vit import ViT`.
3. Train a pretrained ViT feature extractor model (like the one we made in [08. PyTorch Paper Replicating section 10](#)) on 20% of the pizza, steak and sushi data like the dataset we used in [07. PyTorch Experiment Tracking section 7.3](#).
  - See how it performs compared to the EffNetB2 model we compared it to in [08. PyTorch Paper Replicating section 10.6](#).
4. Try repeating the steps from exercise 3 but this time use the "`ViT_B_16_Weights.IMAGENET1K_SWAG_E2E_V1`" pretrained weights from `torchvision.models.vit_b_16()`.
  - **Note:** ViT pretrained with SWAG weights has a minimum input image size of `(384, 384)` (the pretrained ViT in exercise 3 has a minimum input size of `(224, 224)`), though this is accessible in the weights `.transforms()` method.
5. Our custom ViT model architecture closely mimics that of the ViT paper, however, our training recipe misses a

few things. Research some of the following topics from Table 3 in the ViT paper that we miss and write a sentence about each and how it might help with training:

- ImageNet-22k pretraining (more data).
- Learning rate warmup.
- Learning rate decay.
- Gradient clipping.

## Extra-curriculum

- There have been several iterations and tweaks to the Vision Transformer since its original release and the most concise and best performing (as of July 2022) can be viewed in [\*Better plain ViT baselines for ImageNet-1k\*](#). Despite of the upgrades, we stuck with replicating a "vanilla Vision Transformer" in this notebook because if you understand the structure of the original, you can bridge to different iterations.
- The [`vit-pytorch` repository on GitHub by lucidrains](#) is one of the most extensive resources of different ViT architectures implemented in PyTorch. It's a phenomenal reference and one I used often to create the materials we've been through in this chapter.
- PyTorch have their [`own implementation of the ViT architecture on GitHub`](#), it's used as the basis of the pretrained ViT models in `torchvision.models`.
- Jay Alammar has fantastic illustrations and explanations on his blog of the [`attention mechanism`](#) (the foundation of Transformer models) and [`Transformer models`](#).
- Adrish Dey has a fantastic [`write up of Layer Normalization`](#) (a main component of the ViT architecture) can help neural network training.
- The self-attention (and multi-head self-attention) mechanism is at the heart of the ViT architecture as well as many other Transformer architectures, it was originally introduced in the [\*Attention is all you need\*](#) paper.
- Yannic Kilcher's YouTube channel is a sensational resource for visual paper walkthroughs, you can see his videos for the following papers:
  - [\*Attention is all you need\*](#) (the paper that introduced the Transformer architecture).
  - [\*An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale\*](#) (the paper that introduced the ViT architecture).

[Previous](#)

## 07. PyTorch Experiment Tracking

Next

09. PyTorch Model Deployment

Made with Material for MkDocs Insiders



[View Source Code](#) | [View Slides](#)

## 09. PyTorch Model Deployment

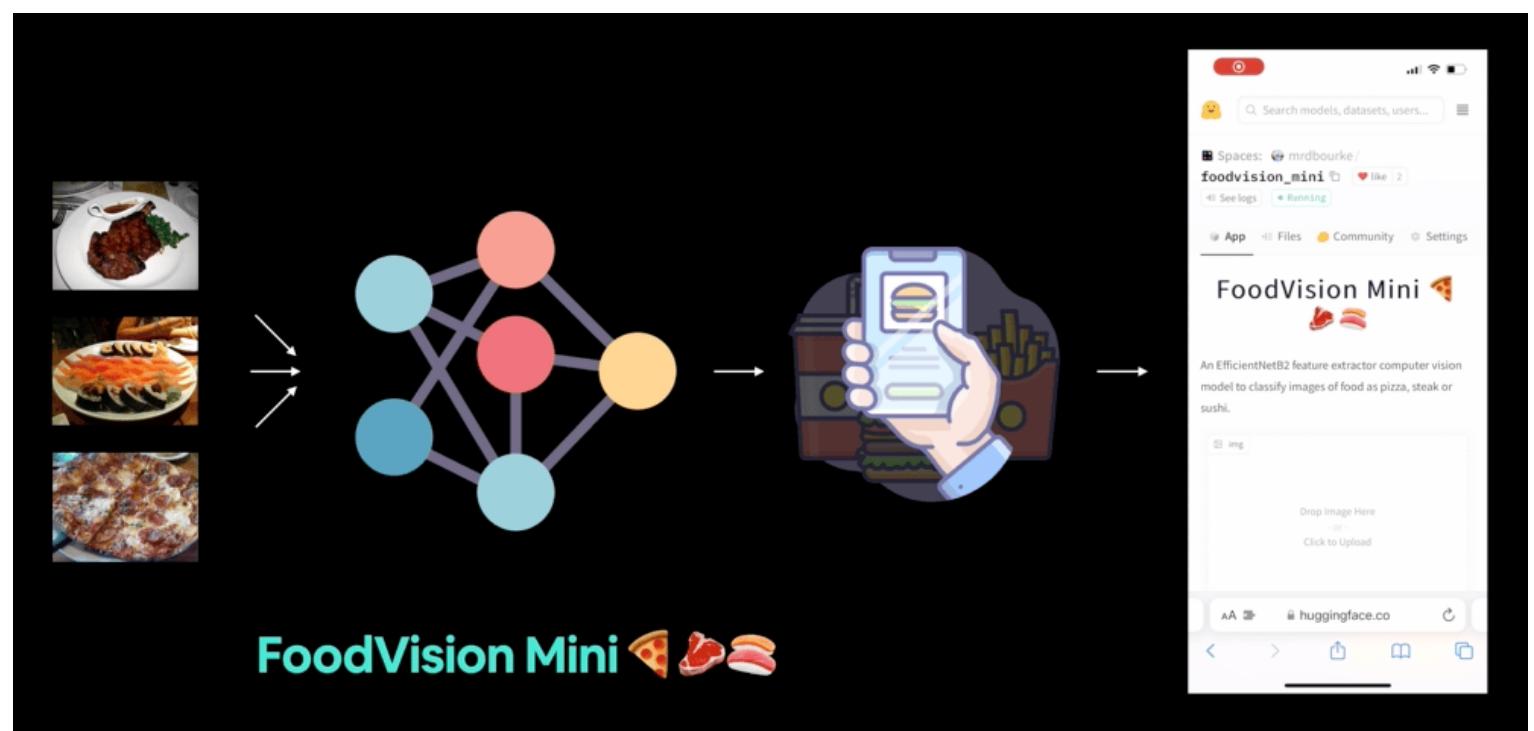
Welcome to Milestone Project 3: PyTorch Model Deployment!

We've come a long way with our FoodVision Mini project.

But so far our PyTorch models have only been accessible to us.

How about we bring FoodVision Mini to life and make it publically accessible?

In other words, **we're going to deploy our FoodVision Mini model to the internet as a usable app!**



*Trying out the [deployed version](#) of FoodVision Mini (what we're going to build) on my lunch. The model got it right too!*

Loading [MathJax]/extensions/Safe.js

# What is machine learning model deployment?

**Machine learning model deployment** is the process of making your machine learning model accessible to someone or something else.

Someone else being a person who can interact with your model in some way.

For example, someone taking a photo on their smartphone of food and then having our FoodVision Mini model classify it into pizza, steak or sushi.

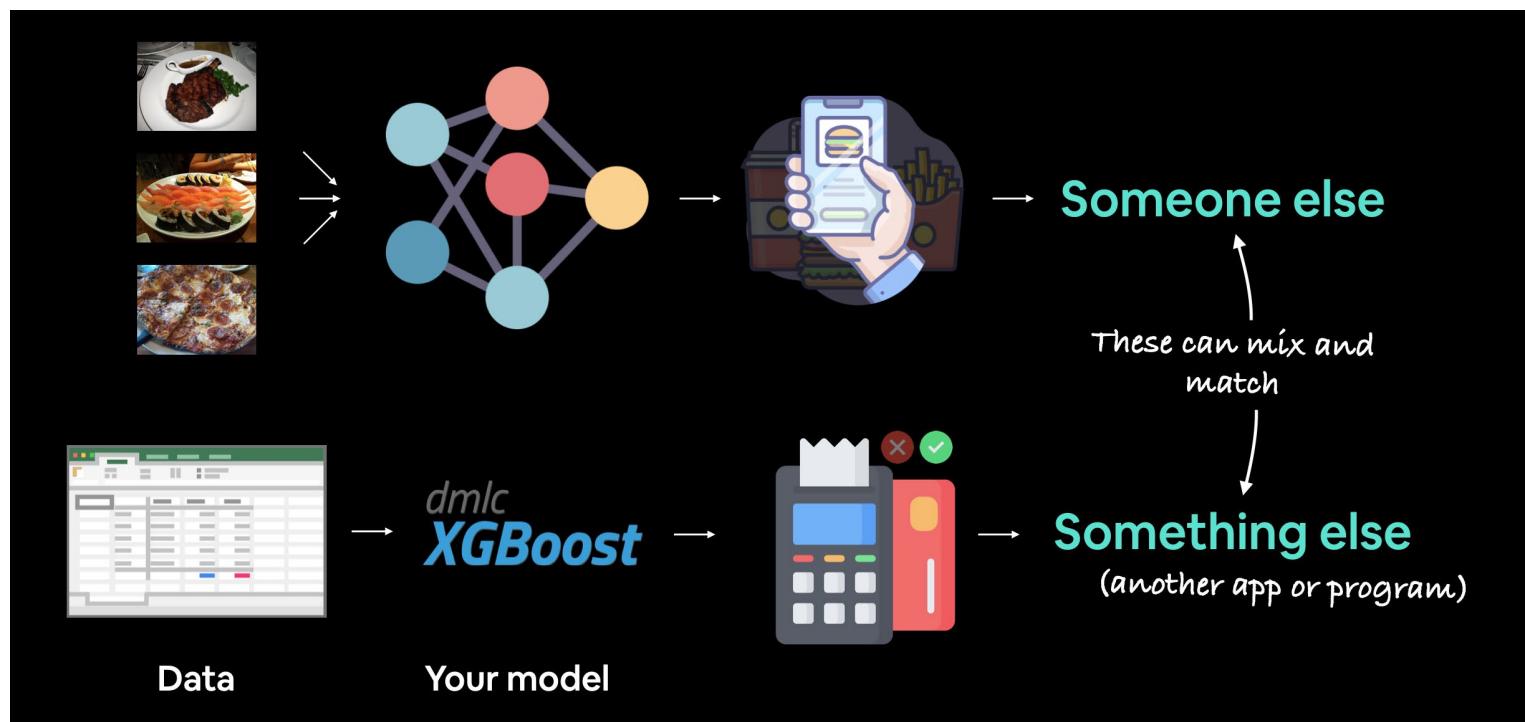
Something else might be another program, app or even another model that interacts with your machine learning model(s).

For example, a banking database might rely on a machine learning model making predictions as to whether a transaction is fraudulent or not before transferring funds.

Or an operating system may lower its resource consumption based on a machine learning model making predictions on how much power someone generally uses at specific times of day.

These use cases can be mixed and matched as well.

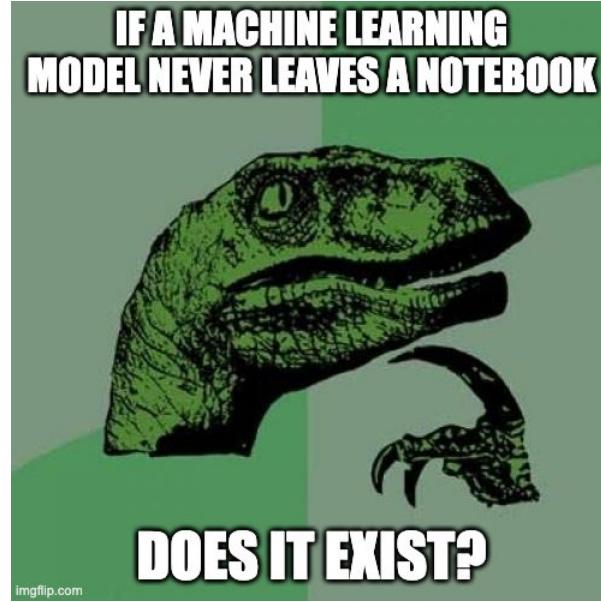
For example, a Tesla car's computer vision system will interact with the car's route planning program (something else) and then the route planning program will get inputs and feedback from the driver (someone else).



*Machine learning model deployment involves making your model available to someone or something else. For example, someone might use your model as part of a food recognition app (such as FoodVision Mini or Nutrify). And something else might be another model or program using your model such as a banking system using a machine learning model to detect if a transaction is fraud or not.*

## Why deploy a machine learning model?

One of the most important philosophical questions in machine learning is:



Deploying a model is as important as training one.

Because although you can get a pretty good idea of how your model's going to function by evaluating it on a well-crafted test set or visualizing its results, you never really know how it's going to perform until you release it to the wild.

Having people who've never used your model interact with it will often reveal edge cases you never thought of during training.

For example, what happens if someone was to upload a photo that *wasn't* of food to our FoodVision Mini model?

One solution would be to create another model that first classifies images as "food" or "not food" and passing the target image through that model first (this is what Nutrify does).

Then if the image is of "food" it goes to our FoodVision Mini model and gets classified into pizza, steak or sushi.

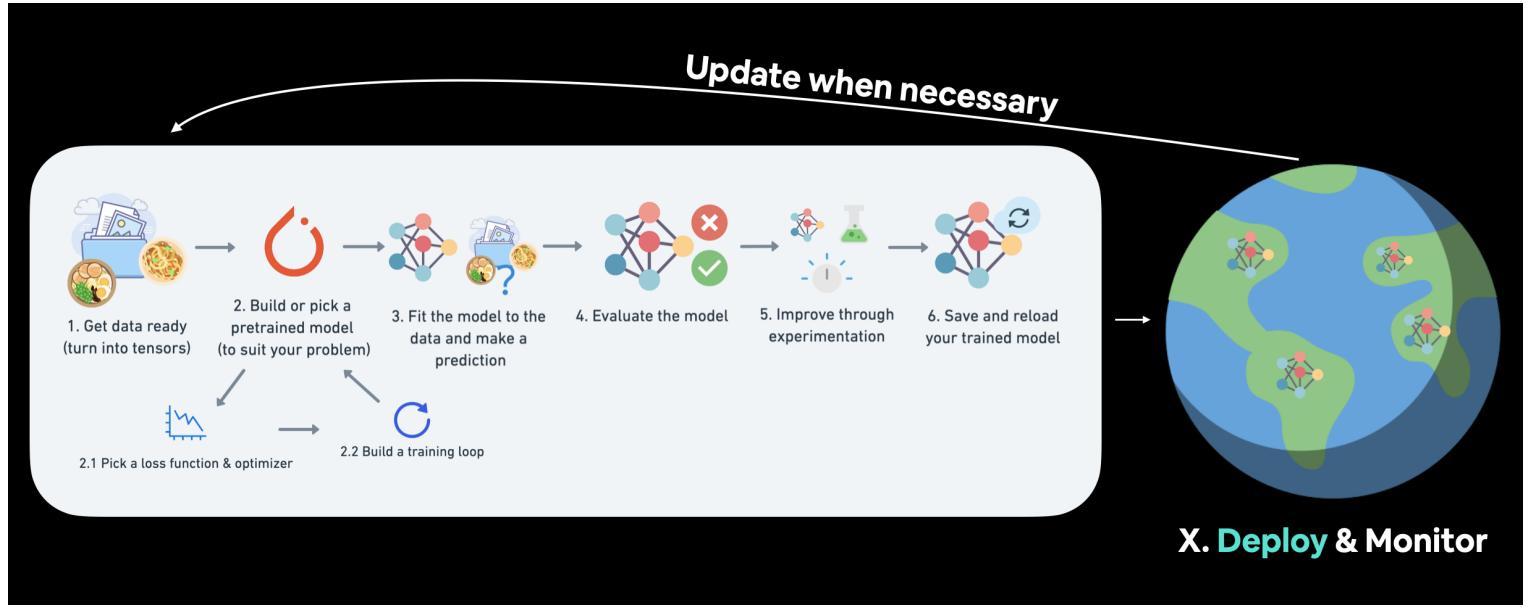
And if it's "not food", a message is displayed.

But what if these predictions were wrong?

What happens then?

You can see how these questions could keep going.

Thus this highlights the importance of model deployment: it helps you figure out errors in your model that aren't obvious during training/testing.



We covered a PyTorch workflow back in [01. PyTorch Workflow](#). But once you've got a good model, deployment is a good next step. Monitoring involves seeing how your model goes on the most important data split: data from the real world. For more resources on deployment and monitoring see [PyTorch Extra Resources](#).

## Different types of machine learning model deployment

Whole books could be written on the different types of machine learning model deployment (and many good ones are listed in [PyTorch Extra Resources](#)).

And the field is still developing in terms of best practices.

But I like to start with the question:

"What is the most ideal scenario for my machine learning model to be used?"

And then work backwards from there.

Of course, you may not know this ahead of time. But you're smart enough to imagine such things.

In the case of FoodVision Mini, our ideal scenario might be:

- Someone takes a photo on a mobile device (through an app or web browser).
- The prediction comes back fast.

Easy.

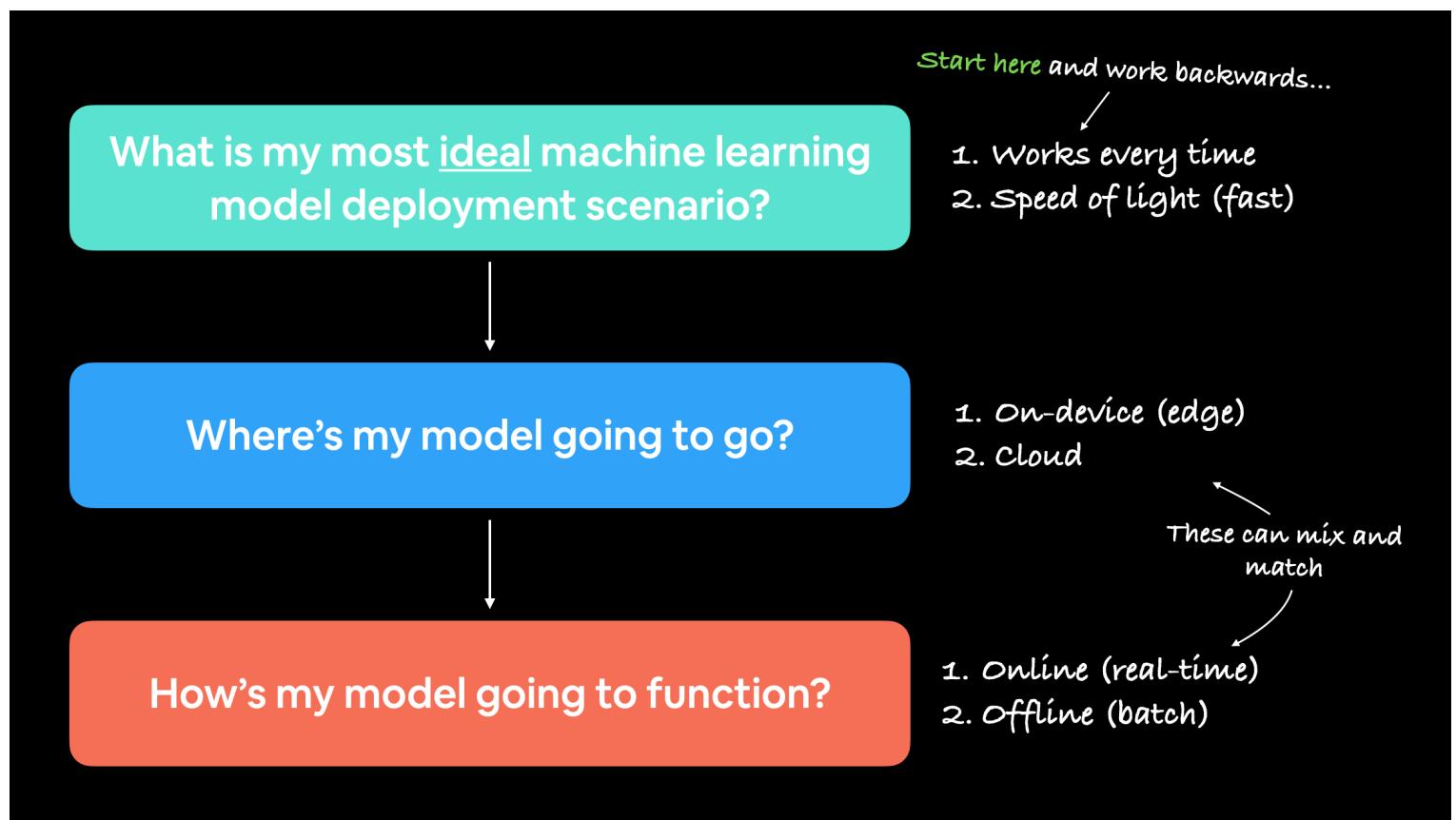
So we've got two main criteria:

1. The model should work on a mobile device (this means there will be some compute constraints).
2. The model should make predictions *fast* (because a slow app is a boring app).

And of course, depending on your use case, your requirements may vary.

You may notice the above two points break down into another two questions:

1. **Where's it going to go?** - As in, where is it going to be stored?
2. **How's it going to function?** - As in, does it return predictions immediately? Or do they come later?



When starting to deploy machine learning models, it's helpful to start by asking what's the most ideal use case and then work backwards from there, asking where the model's going to go and then how it's going to function.

## Where's it going to go?

When you deploy your machine learning model, where does it live?

The main debate here is usually on-device (also called edge/in the browser) or on the cloud (a computer/server that isn't the *actual* device someone/something calls the model from).

Both have their pros and cons.

<b>Deployment location</b>	<b>Pros</b>	<b>Cons</b>
<b>On-device (edge/in the browser)</b>	Can be very fast (since no data leaves the device)	Limited compute power (larger models take longer to run)
	Privacy preserving (again no data has to leave the device)	Limited storage space (smaller model size required)
	No internet connection required (sometimes)	Device-specific skills often required
<b>On cloud</b>	Near unlimited compute power (can scale up when needed)	Costs can get out of hand (if proper scaling limits aren't enforced)
	Can deploy one model and use everywhere (via API)	Predictions can be slower due to data having to leave device and predictions having to come back (network latency)
	Links into existing cloud ecosystem	Data has to leave device (this may cause privacy concerns)

There are more details to these but I've left resources in the [extra-curriculum](#) to learn more.

Let's give an example.

If we're deploying FoodVision Mini as an app, we want it to perform well and fast.

So which model would we prefer?

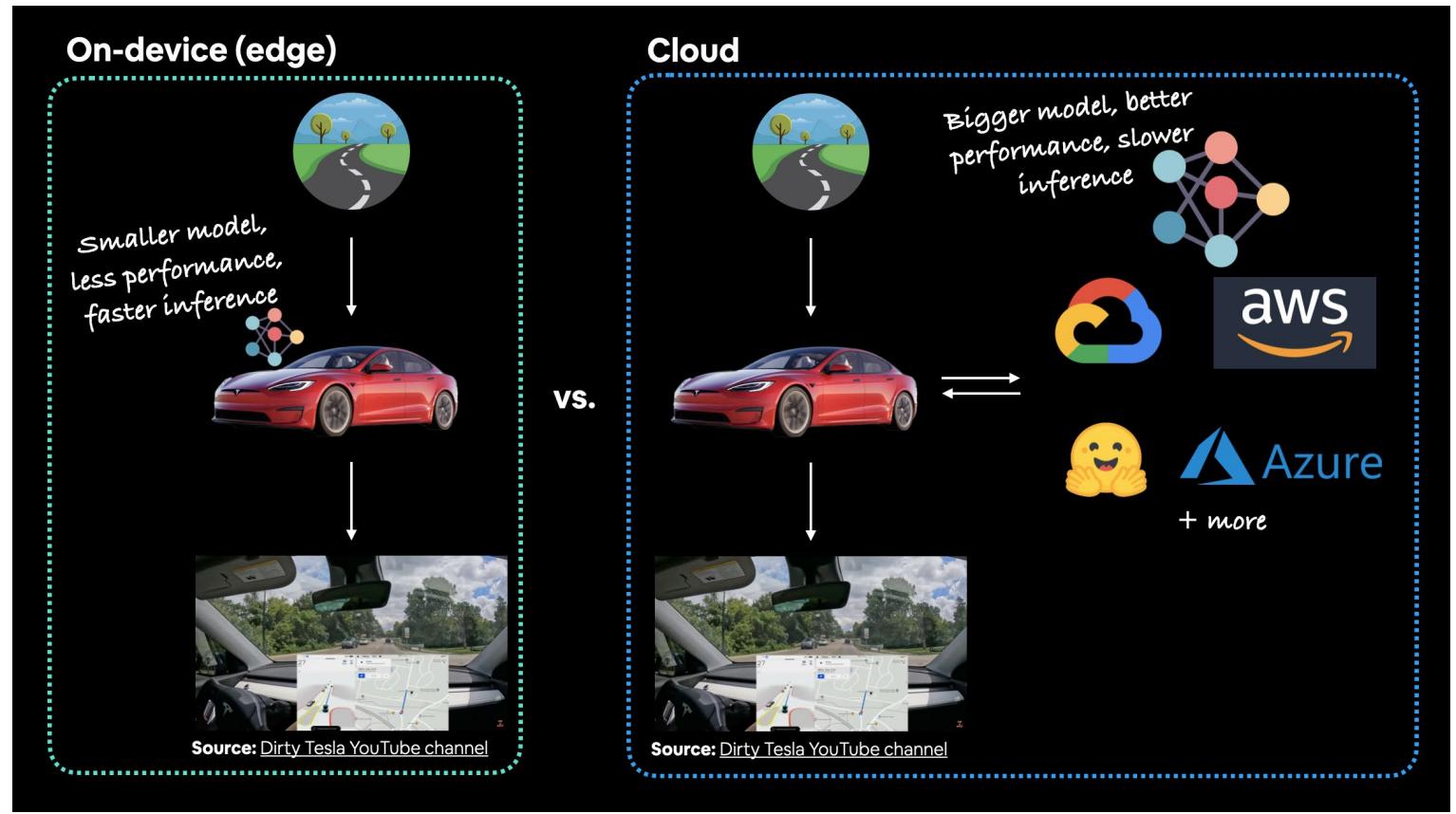
1. A model on-device that performs at 95% accuracy with an inference time (latency) of one second per prediction.
2. A model on the cloud that performs at 98% accuracy with an inference time of 10 seconds per prediction (bigger, better model but takes longer to compute).

I've made these numbers up but they showcase a potential difference between on-device and on the cloud.

Option 1 could potentially be a smaller less performant model that runs fast because its able to fit on a mobile device.

Option 2 could potentially a larger more performant model that requires more compute and storage but it takes a bit longer to run because we have to send data off the device and get it back (so even though the actual prediction might be fast, the network time and data transfer has to factored in).

For FoodVision Mini, we'd likely prefer option 1, because the small hit in performance is far outweighed by the faster inference speed.



In the case of a Tesla car's computer vision system, which would be better? A smaller model that performs well on device (model is on the car) or a larger model that performs better that's on the cloud? In this case, you'd much prefer the model being on the car. The extra network time it would take for data to go from the car to the cloud and then back to the car just wouldn't be worth it (or potentially even possible with poor signal areas).

**Note:** For a full example of seeing what it's like to deploy a PyTorch model to an edge device, see the [PyTorch tutorial on achieving real-time inference \(30fps+\)](#) with a computer vision model on a Raspberry Pi.

How's it going to function?

Back to the ideal use case, when you deploy your machine learning model, how should it work?

As in, would you like predictions returned immediately?

Or is it okay for them to happen later?

These two scenarios are generally referred to as:

- **Online (real-time)** - Predictions/inference happen **immediately**. For example, someone uploads an image, the image gets transformed and predictions are returned or someone makes a purchase and the transaction is verified to be non-fraudulent by a model so the purchase can go through.
- **Offline (batch)** - Predictions/inference happen **periodically**. For example, a photos application sorts your images into different categories (such as beach, mealtime, family, friends) whilst your mobile device is plugged into charge.

**Note:** "Batch" refers to inference being performed on multiple samples at a time. However, to add a little confusion, batch processing can happen immediately/online (multiple images being classified at once) and/or offline (multiple images being predicted/trained on at once).

The main difference between each being: predictions being made immediately or periodically.

Periodically can have a varying timescale too, from every few seconds to every few hours or days.

And you can mix and match the two.

In the case of FoodVision Mini, we'd want our inference pipeline to happen online (real-time), so when someone uploads an image of pizza, steak or sushi, the prediction results are returned immediately (any slower than real-time would make a boring experience).

But for our training pipeline, it's okay for it to happen in a batch (offline) fashion, which is what we've been doing throughout the previous chapters.

## Ways to deploy a machine learning model

We've discussed a couple of options for deploying machine learning models (on-device and cloud).

And each of these will have their specific requirements:

Tool/resource	Deployment type
---------------	-----------------

Google's ML Kit	On-device (Android and iOS)
-----------------	-----------------------------

Apple's Core ML and <code>coremltools</code> Python package	On-device (all Apple devices)
Amazon Web Service's (AWS) Sagemaker	Cloud
Google Cloud's Vertex AI	Cloud
Microsoft's Azure Machine Learning	Cloud
Hugging Face Spaces	Cloud
API with <code>FastAPI</code>	Cloud/self-hosted server
API with <code>TorchServe</code>	Cloud/self-hosted server
ONNX (Open Neural Network Exchange)	Many/general
Many more...	

**Note:** An [application programming interface \(API\)](#) is a way for two (or more) computer programs to interact with each other. For example, if your model was deployed as API, you would be able to write a program that could send data to it and then receive predictions back.

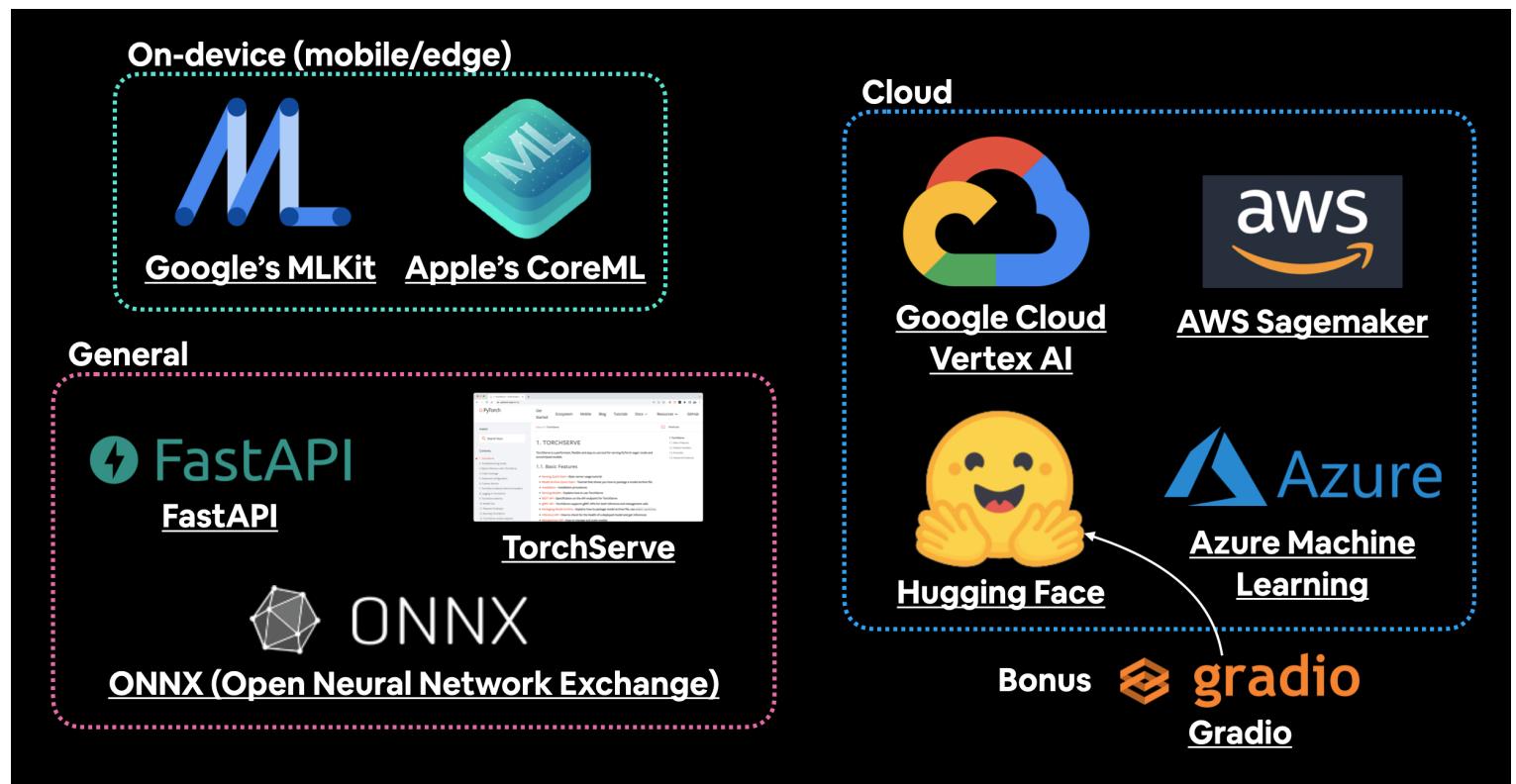
Which option you choose will be highly dependent on what you're building/who you're working with.

But with so many options, it can be very intimidating.

So best to start small and keep it simple.

And one of the best ways to do so is by turning your machine learning model into a demo app with [Gradio](#) and then deploying it on Hugging Face Spaces.

We'll be doing just that with FoodVision Mini later on.



A handful of places and tools to host and deploy machine learning models. There are plenty I've missed so if you'd like to add more, please leave a [discussion on GitHub](#).

## What we're going to cover

Enough talking about deploying a machine learning model.

Let's become machine learning engineers and actually deploy one.

Our goal is to deploy our FoodVision Model via a demo Gradio app with the following metrics:

1. **Performance:** 95%+ accuracy.
2. **Speed:** real-time inference of 30FPS+ (each prediction has a latency of lower than ~0.03s).

We'll start by running an experiment to compare our best two models so far: EffNetB2 and ViT feature extractors.

Then we'll deploy the one which performs closest to our goal metrics.

Finally, we'll finish with a (BIG) surprise bonus.

**0. Getting setup**

We've written a fair bit of useful code over the past few sections, let's download it and make sure we can use it again.

**1. Get data**

Let's download the [pizza\\_steak\\_sushi\\_20\\_percent.zip](#) dataset so we can train our previously best performing models on the same dataset.

**2. FoodVision Mini model deployment experiment outline**

Even on the third milestone project, we're still going to be running multiple experiments to see which model (EffNetB2 or ViT) achieves closest to our goal metrics.

**3. Creating an EffNetB2 feature extractor**

An EfficientNetB2 feature extractor performed the best on our pizza, steak, sushi dataset in [07. PyTorch Experiment Tracking](#), let's recreate it as a candidate for deployment.

**4. Creating a ViT feature extractor**

A ViT feature extractor has been the best performing model yet on our pizza, steak, sushi dataset in [08. PyTorch Paper Replicating](#), let's recreate it as a candidate for deployment alongside EffNetB2.

**5. Making predictions with our trained models and timing them**

We've built two of the best performing models yet, let's make predictions with them and track their results.

**6. Comparing model results, prediction times and size**

Let's compare our models to see which performs best with our goals.

**7. Bringing FoodVision Mini to life by creating a Gradio demo**

One of our models performs better than the other (in terms of our goals), so let's turn it into a working app demo!

**8. Turning our FoodVision Mini Gradio demo into a deployable app**

Our Gradio app demo works locally, let's prepare it for deployment!

**9. Deploying our Gradio demo to HuggingFace Spaces**

Let's take FoodVision Mini to the web and make it publically accessible for all!

**10. Creating a BIG surprise**

We've built FoodVision Mini, time to step things up a notch.

**11. Deploying our BIG surprise**

Deploying one app was fun, how about we make it two?

## Where can you get help?

All of the materials for this course [are available on GitHub](#).

If you run into trouble, you can ask a question on the course [GitHub Discussions page](#).

And of course, there's the [PyTorch documentation](#) and [PyTorch developer forums](#), a very helpful place for all things

PyTorch.

## 0. Getting setup

As we've done previously, let's make sure we've got all of the modules we'll need for this section.

We'll import the Python scripts (such as `data_setup.py` and `engine.py`) we created in [05. PyTorch Going Modular](#).

To do so, we'll download `going_modular` directory from the [pytorch-deep-learning repository](#) (if we don't already have it).

We'll also get the `torchinfo` package if it's not available.

`torchinfo` will help later on to give us a visual representation of our model.

And since later on we'll be using `torchvision v0.13` package (available as of July 2022), we'll make sure we've got the latest versions.

**Note:** If you're using Google Colab, and you don't have a GPU turned on yet, it's now time to turn one on via

Runtime -> Change runtime type -> Hardware accelerator -> GPU .

In [1]:







```
torch version: 1.13.0.dev20220824+cu113
torchvision version: 0.14.0.dev20220824+cu113
```

**Note:** If you're using Google Colab and the cell above starts to install various software packages, you may have to restart your runtime after running the above cell. After restarting, you can run the cell again and verify you've got the right versions of `torch` and `torchvision`.

Now we'll continue with the regular imports, setting up device agnostic code and this time we'll also get the `helper_functions.py` script from GitHub.

The `helper_functions.py` script contains several functions we created in previous sections:

- `set_seeds()` to set the random seeds (created in [07. PyTorch Experiment Tracking section 0](#)).
- `download_data()` to download a data source given a link (created in [07. PyTorch Experiment Tracking section 1](#)).
- `plot_loss_curves()` to inspect our model's training results (created in [04. PyTorch Custom Datasets section 7.8](#))

**Note:** It may be a better idea for many of the functions in the `helper_functions.py` script to be merged into `going_modular/going_modular/utils.py`, perhaps that's an extension you'd like to try.

In [2]:











Finally, we'll setup device-agnostic code to make sure our models run on the GPU.

In [3]:

```
'cuda'
```

Out[3]:

## 1. Getting data

We left off in [08. PyTorch Paper Replicating](#) comparing our own Vision Transformer (ViT) feature extractor model to the EfficientNetB2 (EffNetB2) feature extractor model we created in [07. PyTorch Experiment Tracking](#).

And we found that there was a slight difference in the comparison.

The EffNetB2 model was trained on 20% of the pizza, steak and sushi data from Food101 where as the ViT model was trained on 10%.

Since our goal is to deploy the best model for our FoodVision Mini problem, let's start by downloading the [20% pizza, steak and sushi dataset](#) and train an EffNetB2 feature extractor and ViT feature extractor on it and then compare the two models.

This way we'll be comparing apples to apples (one model trained on a dataset to another model trained on the same dataset).

**Note:** The dataset we're downloading is a sample of the entire [Food101 dataset](#) (101 food classes with 1,000 images each). More specifically, 20% refers to 20% of images from the pizza, steak and sushi classes selected at random.

You can see how this dataset was created in [extras/04\\_custom\\_data\\_creation.ipynb](#) and more details in [04. PyTorch Custom Datasets section 1](#).

We can download the data using the `download_data()` function we created in [07. PyTorch Experiment Tracking section 1](#) from [helper\\_functions.py](#).

In [4]:

```
[INFO] data/pizza_steak_sushi_20_percent directory exists, skipping download.
```

Out[4]:

```
PosixPath('data/pizza_steak_sushi_20_percent')
```

Wonderful!

Now we've got a dataset, let's create training and test paths.

In [5]:

## 2. FoodVision Mini model deployment experiment outline

The ideal deployed model FoodVision Mini performs well and fast.

We'd like our model to perform as close to real-time as possible.

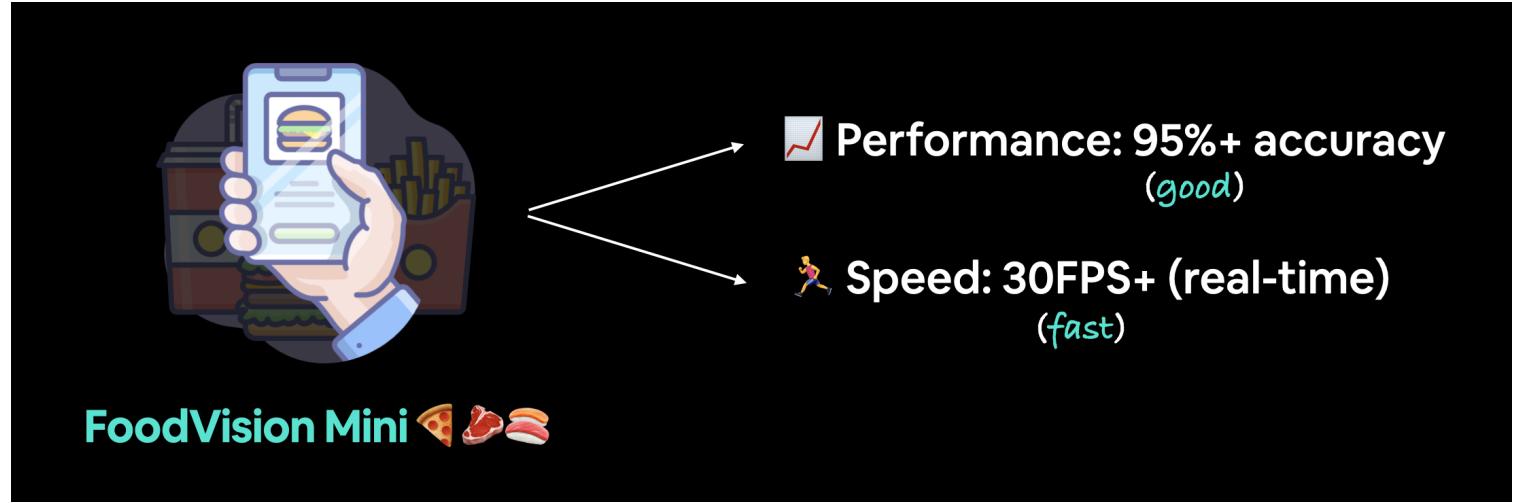
Real-time in this case being ~30FPS (frames per second) because that's [about how fast the human eye can see](#) (there is debate on this but let's just use ~30FPS as our benchmark).

And for classifying three different classes (pizza, steak and sushi), we'd like a model that performs at 95%+ accuracy.

Of course, higher accuracy would be nice but this might sacrifice speed.

So our goals are:

1. **Performance** - A model that performs at 95%+ accuracy.
2. **Speed** - A model that can classify an image at ~30FPS (0.03 seconds inference time per image, also known as latency).

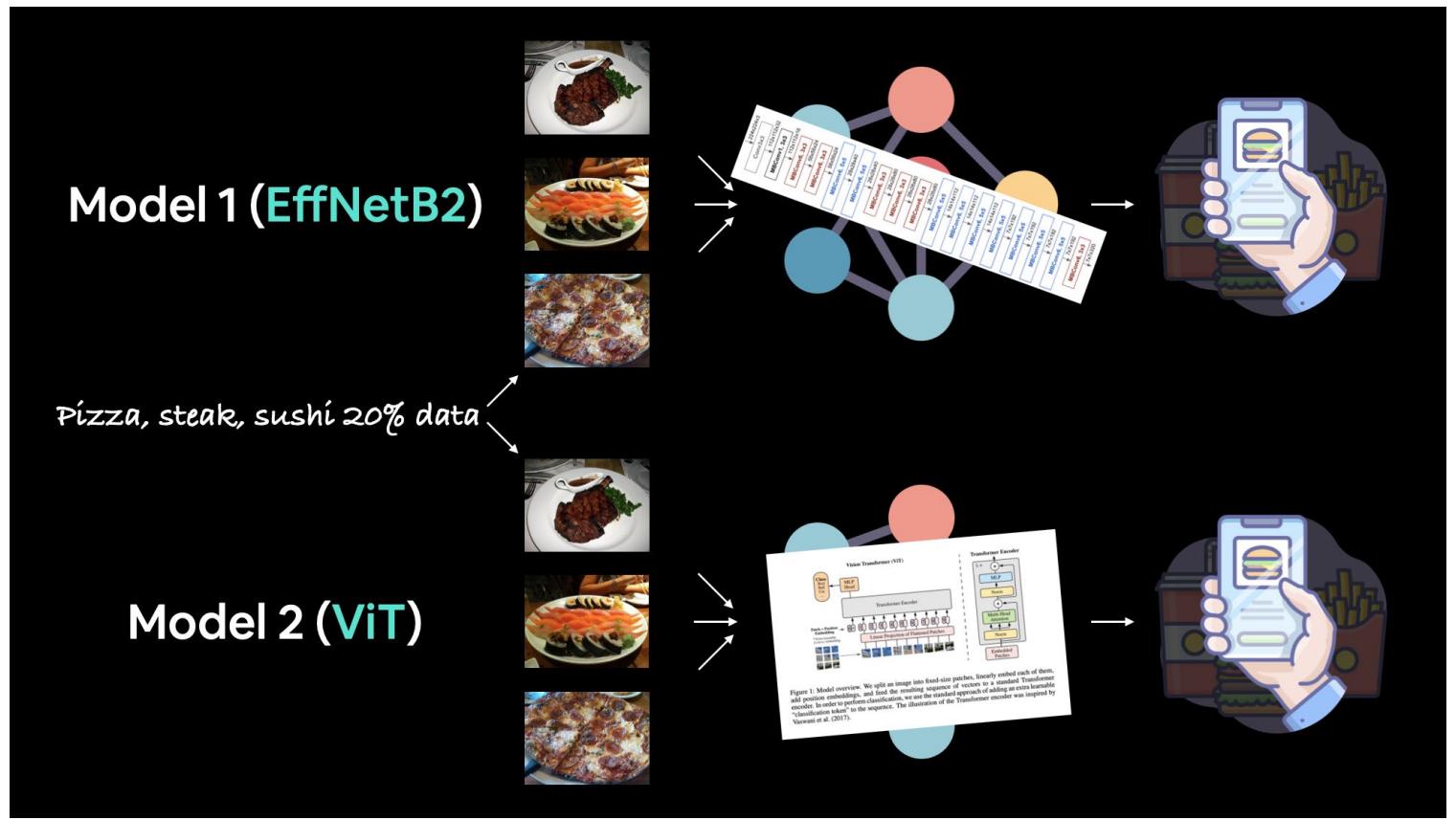


*FoodVision Mini deployment goals. We'd like a fast predicting well-performing model (because a slow app is boring).*

We'll put an emphasis on speed, meaning, we'd prefer a model performing at 90%+ accuracy at ~30FPS than a model performing 95%+ accuracy at 10FPS.

To try and achieve these results, let's bring in our best performing models from the previous sections:

1. **EffNetB2 feature extractor** (EffNetB2 for short) - originally created in [07. PyTorch Experiment Tracking](#) section [7.5](#) using `torchvision.models.efficientnet_b2()` with adjusted `classifier` layers.
2. **ViT-B/16 feature extractor** (ViT for short) - originally created in [08. PyTorch Paper Replicating](#) section [10](#) using `torchvision.models.vit_b_16()` with adjusted `head` layers.
  - Note ViT-B/16 stands for "Vision Transformer Base, patch size 16".



**Note:** A "feature extractor model" often starts with a model that has been pretrained on a dataset similar to your own problem. The pretrained model's base layers are often left frozen (the pretrained patterns/weights stay the same) whilst some of the top (or classifier/classification head) layers get customized to your own problem by training on your own data. We covered the concept of a feature extractor model in [06. PyTorch Transfer Learning section 3.4](#).

### 3. Creating an EffNetB2 feature extractor

We first created an EffNetB2 feature extractor model in [07. PyTorch Experiment Tracking section 7.5](#).

And by the end of that section we saw it performed very well.

So let's now recreate it here so we can compare its results to a ViT feature extractor trained on the same data.

To do so we can:

1. Setup the pretrained weights as `weights=torchvision.models.EfficientNet_B2_Weights.DEFAULT`, where "`DEFAULT`" means "best currently available" (or could use `weights="DEFAULT"`).
2. Get the pretrained model image transforms from the weights with the `transforms()` method (we need these so we can convert our images into the same format as the pretrained EffNetB2 was trained on).

3. Create a pretrained model instance by passing the weights to an instance of `torchvision.models.efficientnet_b2`.
4. Freeze the base layers in the model.
5. Update the classifier head to suit our own data.

In [6]:



Now to change the classifier head, let's first inspect it using the `classifier` attribute of our model.

In [7]:

```
Sequential(  
    (0): Dropout(p=0.3, inplace=True)  
    (1): Linear(in_features=1408, out_features=1000, bias=True)  
)
```

Excellent! To change the classifier head to suit our own problem, let's replace the `out_features` variable with the same number of classes we have (in our case, `out_features=3`, one for pizza, steak, sushi).

**Note:** This process of changing the output layers/classifier head will be dependent on the problem you're working on. For example, if you wanted a different *number* of outputs or a different *kind* of output, you would have to change the output layers accordingly.

In [8]:



Beautiful!

### 3.1 Creating a function to make an EffNetB2 feature extractor

Looks like our EffNetB2 feature extractor is ready to go, however, since there's quite a few steps involved here, how about we turn the code above into a function we can re-use later?

We'll call it `create_effnetb2_model()` and it'll take a customizable number of classes and a random seed parameter for reproducibility.

Ideally, it will return an EffNetB2 feature extractor along with its associated transforms.

In [9]:













Woohoo! That's a nice looking function, let's try it out.

In [10]:



No errors, nice, now to really try it out, let's get a summary with `torchinfo.summary()`.

In [11]:



Layer (type (var_name))	Input Shape	Output Shape	Param #	Trainable
EfficientNet (EfficientNet)	[1, 3, 224, 224]	[1, 3]	--	Partial
Sequential (features)	[1, 3, 224, 224]	[1, 1408, 7, 7]	--	False
Conv2dNormActivation (0)	[1, 3, 224, 224]	[1, 32, 112, 112]	--	False
Conv2d (0)	[1, 3, 224, 224]	[1, 32, 112, 112]	(864)	False
BatchNorm2d (1)	[1, 32, 112, 112]	[1, 32, 112, 112]	(64)	False
SiLU (2)	[1, 32, 112, 112]	[1, 32, 112, 112]	--	--
Sequential (1)	[1, 32, 112, 112]	[1, 16, 112, 112]	--	False
MBConv (0)	[1, 32, 112, 112]	[1, 16, 112, 112]	(1,448)	False
MBConv (1)	[1, 16, 112, 112]	[1, 16, 112, 112]	(612)	False
Sequential (2)	[1, 16, 112, 112]	[1, 24, 56, 56]	--	False
MBConv (0)	[1, 16, 112, 112]	[1, 24, 56, 56]	(6,004)	False
MBConv (1)	[1, 24, 56, 56]	[1, 24, 56, 56]	(10,710)	False
MBConv (2)	[1, 24, 56, 56]	[1, 24, 56, 56]	(10,710)	False
Sequential (3)	[1, 24, 56, 56]	[1, 48, 28, 28]	--	False
MBConv (0)	[1, 24, 56, 56]	[1, 48, 28, 28]	(16,518)	False
MBConv (1)	[1, 48, 28, 28]	[1, 48, 28, 28]	(43,308)	False
MBConv (2)	[1, 48, 28, 28]	[1, 48, 28, 28]	(43,308)	False
Sequential (4)	[1, 48, 28, 28]	[1, 88, 14, 14]	--	False
MBConv (0)	[1, 48, 28, 28]	[1, 88, 14, 14]	(50,300)	False
MBConv (1)	[1, 88, 14, 14]	[1, 88, 14, 14]	(123,750)	False
MBConv (2)	[1, 88, 14, 14]	[1, 88, 14, 14]	(123,750)	False
MBConv (3)	[1, 88, 14, 14]	[1, 88, 14, 14]	(123,750)	False
Sequential (5)	[1, 88, 14, 14]	[1, 120, 14, 14]	--	False
MBConv (0)	[1, 88, 14, 14]	[1, 120, 14, 14]	(149,158)	False
MBConv (1)	[1, 120, 14, 14]	[1, 120, 14, 14]	(237,870)	False
MBConv (2)	[1, 120, 14, 14]	[1, 120, 14, 14]	(237,870)	False
MBConv (3)	[1, 120, 14, 14]	[1, 120, 14, 14]	(237,870)	False
Sequential (6)	[1, 120, 14, 14]	[1, 208, 7, 7]	--	False
MBConv (0)	[1, 120, 14, 14]	[1, 208, 7, 7]	(301,406)	False
MBConv (1)	[1, 208, 7, 7]	[1, 208, 7, 7]	(686,868)	False
MBConv (2)	[1, 208, 7, 7]	[1, 208, 7, 7]	(686,868)	False
MBConv (3)	[1, 208, 7, 7]	[1, 208, 7, 7]	(686,868)	False
MBConv (4)	[1, 208, 7, 7]	[1, 208, 7, 7]	(686,868)	False
Sequential (7)	[1, 208, 7, 7]	[1, 352, 7, 7]	--	False
MBConv (0)	[1, 208, 7, 7]	[1, 352, 7, 7]	(846,900)	False
MBConv (1)	[1, 352, 7, 7]	[1, 352, 7, 7]	(1,888,920)	False
Conv2dNormActivation (8)	[1, 352, 7, 7]	[1, 1408, 7, 7]	--	False
Conv2d (0)	[1, 352, 7, 7]	[1, 1408, 7, 7]	(495,616)	False
BatchNorm2d (1)	[1, 1408, 7, 7]	[1, 1408, 7, 7]	(2,816)	False
SiLU (2)	[1, 1408, 7, 7]	[1, 1408, 7, 7]	--	--
AdaptiveAvgPool2d (avgpool)	[1, 1408, 7, 7]	[1, 1408, 1, 1]	--	--
Sequential (classifier)	[1, 1408]	[1, 3]	--	True
Dropout (0)	[1, 1408]	[1, 1408]	--	--
Linear (1)	[1, 1408]	[1, 3]	4,227	True

Total params:	7,705,221
Trainable params:	4,227
Non-trainable params:	7,700,994
Total mult-adds (M):	657.64
====	
Input size (MB):	0.60
Forward/backward pass size (MB):	156.80
Params size (MB):	30.82
Estimated Total Size (MB):	188.22
====	

Base layers frozen, top layers trainable and customized!

### 3.2 Creating DataLoaders for EffNetB2

Our EffNetB2 feature extractor is ready, time to create some `DataLoader S.`

We can do this by using the `data_setup.create_dataloaders()` function we created in [05. PyTorch Going Modular section 2](#).

We'll use a `batch_size` of 32 and transform our images using the `effnetb2_transforms` so they're in the same format that our `effnetb2` model was trained on.

In [12]:











### 3.3 Training EffNetB2 feature extractor

Model ready, `DataLoader`s ready, let's train!

Just like in [07. PyTorch Experiment Tracking section 7.6](#), ten epochs should be enough to get good results.

We can do so by creating an optimizer (we'll use `torch.optim.Adam()` with a learning rate of `1e-3`), a loss function (we'll use `torch.nn.CrossEntropyLoss()` for multi-class classification) and then passing these as well as our `DataLoader`s to the `engine.train()` function we created in [05. PyTorch Going Modular section 4](#).

In [13]:









```
0% | 0/10 [00:00<?, ?it/s]
Epoch: 1 | train_loss: 0.9856 | train_acc: 0.5604 | test_loss: 0.7408 | test_acc: 0.9347
Epoch: 2 | train_loss: 0.7175 | train_acc: 0.8438 | test_loss: 0.5869 | test_acc: 0.9409
Epoch: 3 | train_loss: 0.5876 | train_acc: 0.8917 | test_loss: 0.4909 | test_acc: 0.9500
Epoch: 4 | train_loss: 0.4474 | train_acc: 0.9062 | test_loss: 0.4355 | test_acc: 0.9409
Epoch: 5 | train_loss: 0.4290 | train_acc: 0.9104 | test_loss: 0.3915 | test_acc: 0.9443
```

```
Epoch: 6 | train_loss: 0.4381 | train_acc: 0.8896 | test_loss: 0.3512 | test_acc: 0.9688
Epoch: 7 | train_loss: 0.4245 | train_acc: 0.8771 | test_loss: 0.3268 | test_acc: 0.9563
Epoch: 8 | train_loss: 0.3897 | train_acc: 0.8958 | test_loss: 0.3457 | test_acc: 0.9381
Epoch: 9 | train_loss: 0.3749 | train_acc: 0.8812 | test_loss: 0.3129 | test_acc: 0.9131
Epoch: 10 | train_loss: 0.3757 | train_acc: 0.8604 | test_loss: 0.2813 | test_acc: 0.9688
```

### 3.4 Inspecting EffNetB2 loss curves

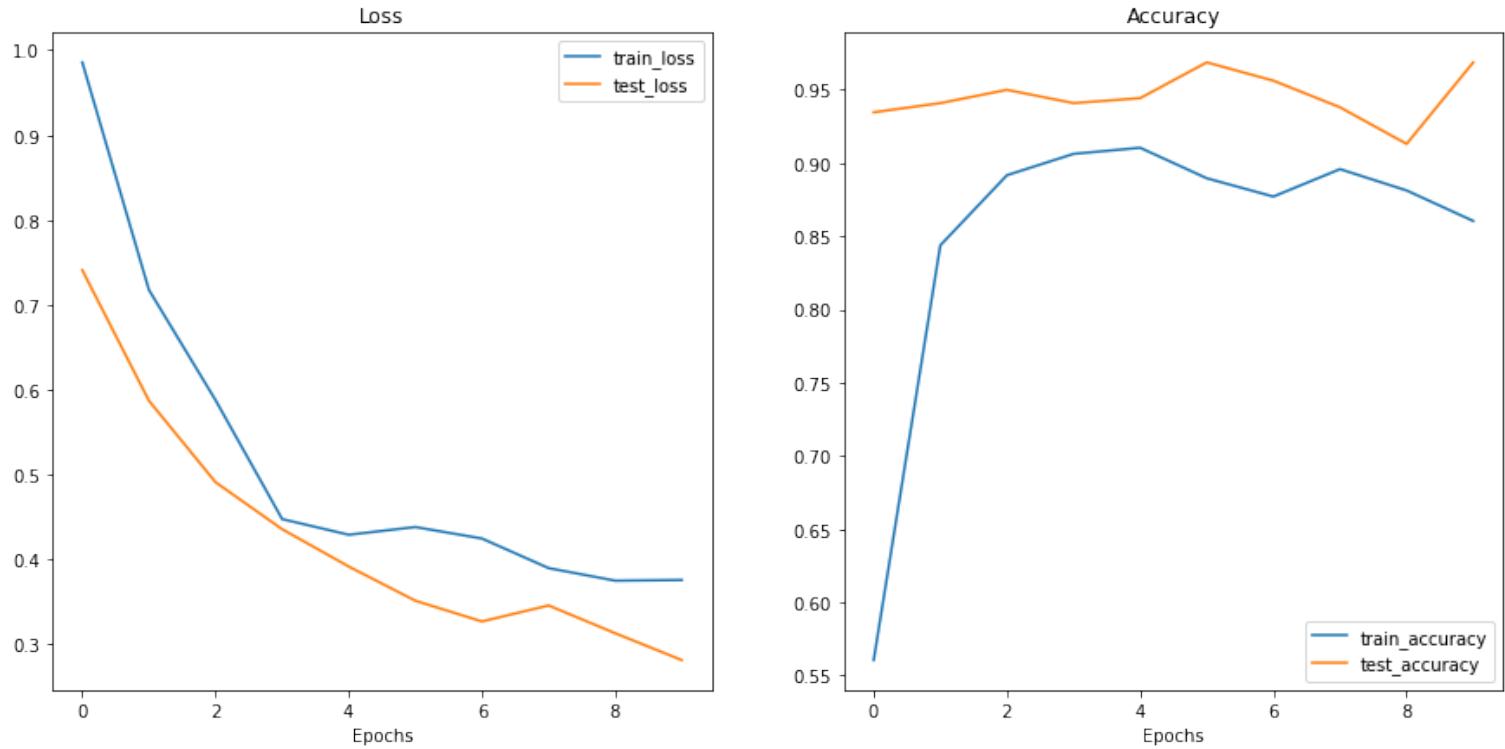
Nice!

As we saw in 07. PyTorch Experiment Tracking, the EffNetB2 feature extractor model works quite well on our data.

Let's turn its results into loss curves to inspect them further.

**Note:** Loss curves are one of the best ways to visualize how your model's performing. For more on loss curves, check out [04. PyTorch Custom Datasets section 8: What should an ideal loss curve look like?](#)

In [14]:



Woah!

Those are some nice looking loss curves.

It looks like our model is performing quite well and perhaps would benefit from a little longer training and potentially some [data augmentation](#) (to help prevent potential overfitting occurring from longer training).

### 3.5 Saving EffNetB2 feature extractor

Now we've got a well-performing trained model, let's save it to file so we can import and use it later.

To save our model we can use the `utils.save_model()` function we created in [05. PyTorch Going Modular section 5](#).

We'll set the `target_dir` to "models" and the `model_name` to

"09\_pretrained\_effnetb2\_feature\_extractor\_pizza\_steak\_sushi\_20\_percent.pth" (a little comprehensive but at least we know what's going on).

In [15]:

```
[INFO] Saving model to: models/09_pretrained_effnetb2_feature_extractor_pizza_steak_sushi_20_percent.pth
```

## 3.6 Checking the size of EffNetB2 feature extractor

Since one of our criteria for deploying a model to power FoodVision Mini is **speed** (~30FPS or better), let's check the size of our model.

Why check the size?

Well, while not always the case, the size of a model can influence its inference speed.

As in, if a model has more parameters, it generally performs more operations and each one of these operations requires some computing power.

And because we'd like our model to work on devices with limited computing power (e.g. on a mobile device or in a web browser), generally, the smaller the size the better (as long as it still performs well in terms of accuracy).

To check our model's size in bytes, we can use Python's `pathlib.Path.stat("path_to_model").st_size` and then we can convert it (roughly) to megabytes by dividing it by `(1024*1024)`.

In [16]:

```
Pretrained EffNetB2 feature extractor model size: 29 MB
```

### 3.7 Collecting EffNetB2 feature extractor stats

We've got a few statistics about our EffNetB2 feature extractor model such as test loss, test accuracy and model size, how about we collect them all in a dictionary so we can compare them to the upcoming ViT feature extractor.

And we'll calculate an extra one for fun, total number of parameters.

We can do so by counting the number of elements (or patterns/weights) in `effnetb2.parameters()`. We'll access the number of elements in each parameter using the `torch.numel()` (short for "number of elements") method.

In [17]:

```
7705221
```

Out[17]:

Excellent!

Now let's put everything in a dictionary so we can make comparisons later on.

In [18]:



Out[18]:

```
{'test_loss': 0.28128674924373626,  
 'test_acc': 0.96875,  
 'number_of_parameters': 7705221,  
 'model_size (MB)': 29}
```

Epic!

Looks like our EffNetB2 model is performing at over 95% accuracy!

Criteria number 1: perform at 95%+ accuracy, tick!

## 4. Creating a ViT feature extractor

Time to continue with our FoodVision Mini modelling experiments.

This time we're going to create a ViT feature extractor.

And we'll do it in much the same way as the EffNetB2 feature extractor except this time with

```
torchvision.models.vit_b_16() instead of torchvision.models.efficientnet_b2().
```

We'll start by creating a function called `create_vit_model()` which will be very similar to `create_effnetb2_model()` except of course returning a ViT feature extractor model and transforms rather than EffNetB2.

Another slight difference is that `torchvision.models.vit_b_16()`'s output layer is called `heads` rather than `classifier`.

In [19]:

Out[19]:

```
Sequential(  
    (head): Linear(in_features=768, out_features=1000, bias=True)  
)
```

Knowing this, we've got all the pieces of the puzzle we need.

In [20]:















ViT feature extraction model creation function ready!

Let's test it out.

In [21]:

No errors, lovely to see!

Now let's get a nice-looking summary of our ViT model using `torchinfo.summary()`.

In [22]:



Layer (type (var_name))	Input Shape	Output Shape	Param #	Trainable
VisionTransformer (VisionTransformer)	[1, 3, 224, 224]	[1, 3]	768	Partial
└Conv2d (conv_proj)	[1, 3, 224, 224]	[1, 768, 14, 14]	(590,592)	False
└Encoder (encoder)	[1, 197, 768]	[1, 197, 768]	151,296	False
└Dropout (dropout)	[1, 197, 768]	[1, 197, 768]	--	--
└Sequential (layers)	[1, 197, 768]	[1, 197, 768]	--	False
└EncoderBlock (encoder_layer_0)	[1, 197, 768]	[1, 197, 768]	(7,087,872)	False
└EncoderBlock (encoder_layer_1)	[1, 197, 768]	[1, 197, 768]	(7,087,872)	False
└EncoderBlock (encoder_layer_2)	[1, 197, 768]	[1, 197, 768]	(7,087,872)	False
└EncoderBlock (encoder_layer_3)	[1, 197, 768]	[1, 197, 768]	(7,087,872)	False
└EncoderBlock (encoder_layer_4)	[1, 197, 768]	[1, 197, 768]	(7,087,872)	False
└EncoderBlock (encoder_layer_5)	[1, 197, 768]	[1, 197, 768]	(7,087,872)	False
└EncoderBlock (encoder_layer_6)	[1, 197, 768]	[1, 197, 768]	(7,087,872)	False
└EncoderBlock (encoder_layer_7)	[1, 197, 768]	[1, 197, 768]	(7,087,872)	False
└EncoderBlock (encoder_layer_8)	[1, 197, 768]	[1, 197, 768]	(7,087,872)	False
└EncoderBlock (encoder_layer_9)	[1, 197, 768]	[1, 197, 768]	(7,087,872)	False
└EncoderBlock (encoder_layer_10)	[1, 197, 768]	[1, 197, 768]	(7,087,872)	False
└EncoderBlock (encoder_layer_11)	[1, 197, 768]	[1, 197, 768]	(7,087,872)	False
└LayerNorm (ln)	[1, 197, 768]	[1, 197, 768]	(1,536)	False
└Sequential (heads)	[1, 768]	[1, 3]	--	True
└Linear (0)	[1, 768]	[1, 3]	2,307	True
Total params: 85,800,963				
Trainable params: 2,307				
Non-trainable params: 85,798,656				
Total mult-adds (M): 172.47				
====				
Input size (MB): 0.60				
Forward/backward pass size (MB): 104.09				
Params size (MB): 257.55				
Estimated Total Size (MB): 362.24				
====				

Just like our EffNetB2 feature extractor model, our ViT model's base layers are frozen and the output layer is customized to our needs!

Do you notice the big difference though?

Our ViT model has *far* more parameters than our EffNetB2 model. Perhaps this will come into play when we compare our models across speed and performance later on.

## 4.1 Create DataLoaders for ViT

We've got our ViT model ready, now let's create some `DataLoader`s for it.

We'll do this in the same way we did for EffNetB2 except we'll use `vit_transforms` to transform our images into the same format the ViT model was trained on.

In [23]:









## 4.2 Training ViT feature extractor

You know what time it is...

...it's traininggggggg time (sung in the same tune as the song [Closing Time](#)).

Let's train our ViT feature extractor model for 10 epochs using our `engine.train()` function with `torch.optim.Adam()` and a learning rate of `1e-3` as our optimizer and `torch.nn.CrossEntropyLoss()` as our loss function.

We'll use our `set_seeds()` function before training to try and make our results as reproducible as possible.

In [24]:









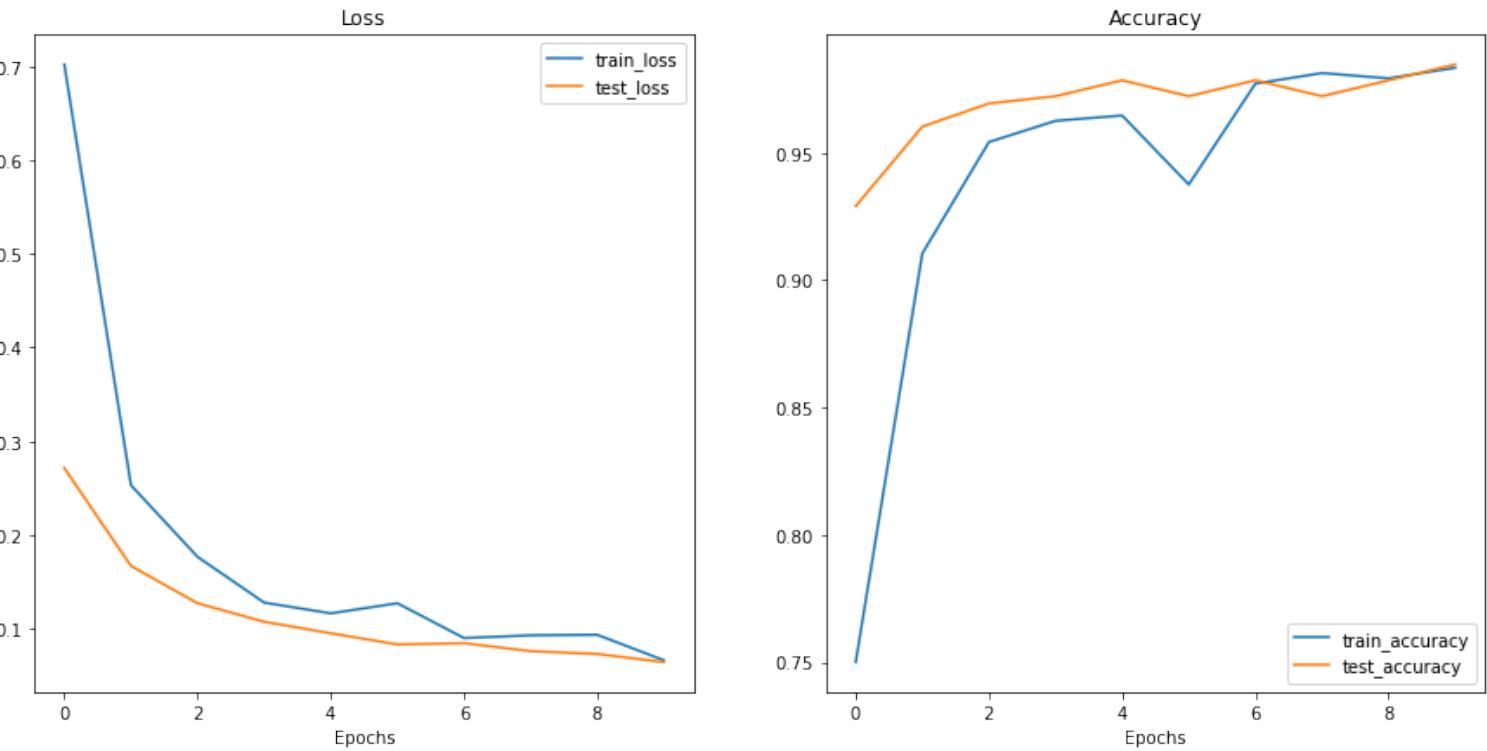
```
0% | 0/10 [00:00<?, ?it/s]
Epoch: 1 | train_loss: 0.7023 | train_acc: 0.7500 | test_loss: 0.2714 | test_acc: 0.9290
Epoch: 2 | train_loss: 0.2531 | train_acc: 0.9104 | test_loss: 0.1669 | test_acc: 0.9602
Epoch: 3 | train_loss: 0.1766 | train_acc: 0.9542 | test_loss: 0.1270 | test_acc: 0.9693
Epoch: 4 | train_loss: 0.1277 | train_acc: 0.9625 | test_loss: 0.1072 | test_acc: 0.9722
Epoch: 5 | train_loss: 0.1163 | train_acc: 0.9646 | test_loss: 0.0950 | test_acc: 0.9784
Epoch: 6 | train_loss: 0.1270 | train_acc: 0.9375 | test_loss: 0.0830 | test_acc: 0.9722
Epoch: 7 | train_loss: 0.0899 | train_acc: 0.9771 | test_loss: 0.0844 | test_acc: 0.9784
Epoch: 8 | train_loss: 0.0928 | train_acc: 0.9812 | test_loss: 0.0759 | test_acc: 0.9722
Epoch: 9 | train_loss: 0.0933 | train_acc: 0.9792 | test_loss: 0.0729 | test_acc: 0.9784
Epoch: 10 | train_loss: 0.0662 | train_acc: 0.9833 | test_loss: 0.0642 | test_acc: 0.9847
```

## 4.3 Inspecting ViT loss curves

Alright, alright, alright, ViT model trained, let's get visual and see some loss curves.

**Note:** Don't forget you can see what an ideal set of loss curves should look like in [04. PyTorch Custom Datasets section 8](#).

In [25]:



Ohh yeah!

Those are some nice looking loss curves. Just like our EffNetB2 feature extractor model, it looks our ViT model might benefit from a little longer training time and perhaps some [data augmentation](#) (to help prevent overfitting).

#### 4.4 Saving ViT feature extractor

Our ViT model is performing outstanding!

So let's save it to file so we can import it and use it later if we wish.

We can do so using the `utils.save_model()` function we created in [05. PyTorch Going Modular section 5](#).

In [26]:

```
[INFO] Saving model to: models/09_pretrained_vit_feature_extractor_pizza_steak_sushi_20_percent.pth
```

## 4.5 Checking the size of ViT feature extractor

And since we want to compare our EffNetB2 model to our ViT model across a number of characteristics, let's find out its size.

To check our model's size in bytes, we can use Python's `pathlib.Path.stat("path_to_model").st_size` and then we can convert it (roughly) to megabytes by dividing it by `(1024*1024)`.

In [27]:

```
Pretrained ViT feature extractor model size: 327 MB
```

Hmm, how does the ViT feature extractor model size compare to our EffNetB2 model size?

We'll find this out shortly when we compare all of our model's characteristics.

## 4.6 Collecting ViT feature extractor stats

Let's put together all of our ViT feature extractor model statistics.

We saw it in the summary output above but we'll calculate its total number of parameters.

In [28]:

Out[28]:

85800963

Woah, that looks like a fair bit more than our EffNetB2!

**Note:** A larger number of parameters (or weights/patterns) generally means a model has a higher *capacity* to learn, whether it actually uses this extra capacity is another story. In light of this, our EffNetB2 model has 7,705,221 parameters where as our ViT model has 85,800,963 (11.1x more) so we could assume that our ViT model has more of a capacity to learn, if given more data (more opportunities to learn). However, this larger capacity to learn often comes with an increased model filesize and a longer time to perform inference.

Now let's create a dictionary with some important characteristics of our ViT model.

In [29]:



Out[29]:

```
{'test_loss': 0.06418210905976593,  
 'test_acc': 0.984659090909091,  
 'number_of_parameters': 85800963,  
 'model_size (MB)': 327}
```

Nice! Looks like our ViT model achieves over 95% accuracy too.

## 5. Making predictions with our trained models and timing them

We've got a couple of trained models, both performing pretty well.

Now how about we test them out doing what we'd like them to do?

As in, let's see how they go making predictions (performing inference).

We know both of our models are performing at over 95% accuracy on the test dataset, but how fast are they?

Ideally, if we're deploying our FoodVision Mini model to a mobile device so people can take photos of their food and identify it, we'd like the predictions to happen at real-time (~30 frames per second).

That's why our second criteria is: a fast model.

To find out how long each of our models take to perform inference, let's create a function called `pred_and_store()` to iterate over each of the test dataset images one by one and perform a prediction.

We'll time each of the predictions as well as store the results in a common prediction format: a list of dictionaries (where each element in the list is a single prediction and each sinlge prediction is a dictionary).

**Note:** We time the predictions one by one rather than by batch because when our model is deployed, it will likely only be making a prediction on one image at a time. As in, someone takes a photo and our model predicts on that single image.

Since we'd like to make predictions across all the images in the test set, let's first get a list of all of the test image paths so we can iterate over them.

To do so, we'll use Python's `pathlib.Path("target_dir").glob("*/*.jpg")` to find all of the filepaths in a target directory with the extension `.jpg` (all of our test images).

In [30]:

```
[INFO] Finding all filepaths ending with '.jpg' in directory: data/pizza_steak_sushi_20_percent/t
```

est

Out[30]:

```
[PosixPath('data/pizza_steak_sushi_20_percent/test/steak/831681.jpg'),
 PosixPath('data/pizza_steak_sushi_20_percent/test/steak/3100563.jpg'),
 PosixPath('data/pizza_steak_sushi_20_percent/test/steak/2752603.jpg'),
 PosixPath('data/pizza_steak_sushi_20_percent/test/steak/39461.jpg'),
 PosixPath('data/pizza_steak_sushi_20_percent/test/steak/730464.jpg')]
```

## 5.1 Creating a function to make predictions across the test dataset

Now we've got a list of our test image paths, let's get to work on our `pred_and_store()` function:

1. Create a function that takes a list of paths, a trained PyTorch model, a series of transforms (to prepare images), a list of target class names and a target device.
2. Create an empty list to store prediction dictionaries (we want the function to return a list of dictionaries, one for each prediction).
3. Loop through the target input paths (steps 4-14 will happen inside the loop).
4. Create an empty dictionary for each iteration in the loop to store prediction values per sample.
5. Get the sample path and ground truth class name (we can do this by inferring the class from the path).
6. Start the prediction timer using Python's `timeit.default_timer()`.
7. Open the image using `PIL.Image.open(path)`.
8. Transform the image so it's capable of being used with the target model as well as add a batch dimension and send the image to the target device.
9. Prepare the model for inference by sending it to the target device and turning on `eval()` mode.
10. Turn on `torch.inference_mode()` and pass the target transformed image to the model and calculate the prediction probability using `torch.softmax()` and the target label using `torch.argmax()`.
11. Add the prediction probability and prediction class to the prediction dictionary created in step 4. Also make sure the prediction probability is on the CPU so it can be used with non-GPU libraries such as NumPy and pandas for later inspection.
12. End the prediction timer started in step 6 and add the time to the prediction dictionary created in step 4.
13. See if the predicted class matches the ground truth class from step 5 and add the result to the prediction dictionary created in step 4.
14. Append the updated prediction dictionary to the empty list of predictions created in step 2.
15. Return the list of prediction dictionaries.

A bunch of steps, but nothing we can't handle!

Let's do it.

In [31]:







































Ho, ho!

What a good looking function!

And you know what, since our `pred_and_store()` is a pretty good utility function for making and storing predictions, it could be stored to `going_modular.going_modular.predictions.py` for later use. That might be an extension you'd like to try, check out [05. PyTorch Going Modular](#) for ideas.

## 5.2 Making and timing predictions with EffNetB2

Time to test out our `pred_and_store()` function!

Let's start by using it to make predictions across the test dataset with our EffNetB2 model, paying attention to two details:

1. **Device** - We'll hard code the `device` parameter to use `"cpu"` because when we deploy our model, we won't always have access to a `"cuda"` (GPU) device.
  - Making the predictions on CPU will be a good indicator of speed of inference too because generally predictions on CPU devices are slower than GPU devices.
2. **Transforms** - We'll also be sure to set the `transform` parameter to `effnetb2_transforms` to make sure the images are opened and transformed in the same way our `effnetb2` model has been trained on.

In [32]:







```
0% | 0/150 [00:00<?, ?it/s]
```

Nice! Look at those predictions fly!

Let's inspect the first couple and see what they look like.

In [33]:

Out[33]:

```
[{'image_path': PosixPath('data/pizza_steak_sushi_20_percent/test/steak/831681.jpg'),  
 'class_name': 'steak',  
 'pred_prob': 0.9293,  
 'pred_class': 'steak',  
 'time_for_pred': 0.0494,  
 'correct': True},  
 {'image_path': PosixPath('data/pizza_steak_sushi_20_percent/test/steak/3100563.jpg'),  
 'class_name': 'steak',  
 'pred_prob': 0.9534,  
 'pred_class': 'steak',  
 'time_for_pred': 0.0264,  
 'correct': True}]
```

Woohoo!

It looks like our `pred_and_store()` function worked nicely.

Thanks to our list of dictionaries data structure, we've got plenty of useful information we can further inspect.

To do so, let's turn our list of dictionaries into a pandas DataFrame.

In [34]:

Out[34]:

	image_path	class_name	pred_prob	pred_class	time_for_pred	correct
0	data/pizza_steak_sushi_20_percent/test/steak/8...	steak	0.9293	steak	0.0494	True
1	data/pizza_steak_sushi_20_percent/test/steak/3...	steak	0.9534	steak	0.0264	True
2	data/pizza_steak_sushi_20_percent/test/steak/2...	steak	0.7532	steak	0.0256	True
3	data/pizza_steak_sushi_20_percent/test/steak/3...	steak	0.5935	steak	0.0263	True
4	data/pizza_steak_sushi_20_percent/test/steak/7...	steak	0.8959	steak	0.0269	True

Beautiful!

Look how easily those prediction dictionaries turn into a structured format we can perform analysis on.

Such as finding how many predictions our EffNetB2 model got wrong...

In [35]:

Out[35]:

```
True      145
False      5
Name: correct, dtype: int64
```

Five wrong predictions out of 150 total, not bad!

And how about the average prediction time?

In [36]:

```
EffNetB2 average time per prediction: 0.0269 seconds
```

Hmm, how does that average prediction time live up to our criteria of our model performing at real-time (~30FPS or 0.03 seconds per prediction)?

**Note:** Prediction times will be different across different hardware types (e.g. a local Intel i9 vs Google Colab CPU). The better and faster the hardware, generally, the faster the prediction. For example, on my local deep learning PC with an Intel i9 chip, my average prediction time with EffNetB2 is around 0.031 seconds (just under real-time). However, on Google Colab (I'm not sure what CPU hardware Colab uses but it looks like it might be an [Intel\(R\) Xeon\(R\)](#)), my average prediction time with EffNetB2 is about 0.1396 seconds (3-4x slower).

Let's add our EffNetB2 average time per prediction to our `effnetb2_stats` dictionary.

In [37]:

```
{'test_loss': 0.28128674924373626,  
 'test_acc': 0.96875,  
 'number_of_parameters': 7705221,  
 'model_size (MB)': 29,  
 'time_per_pred_cpu': 0.0269}
```

Out[37]:

## 5.3 Making and timing predictions with ViT

We've made predictions with our EffNetB2 model, now let's do the same for our ViT model.

To do so, we can use the `pred_and_store()` function we created above except this time we'll pass in our `vit` model as well as the `vit_transforms`.

And we'll keep the predictions on the CPU via `device="cpu"` (a natural extension here would be to test the prediction times on CPU and on GPU).

In [38]:







```
0% | 0/150 [00:00<?, ?it/s]
```

Predictions made!

Now let's check out the first couple.

In [39]:

Out[39]:

```
[{'image_path': PosixPath('data/pizza_steak_sushi_20_percent/test/steak/831681.jpg'),  
 'class_name': 'steak',  
 'pred_prob': 0.9933,  
 'pred_class': 'steak',  
 'time_for_pred': 0.1313,  
 'correct': True},  
 {'image_path': PosixPath('data/pizza_steak_sushi_20_percent/test/steak/3100563.jpg'),  
 'class_name': 'steak',  
 'pred_prob': 0.9893,  
 'pred_class': 'steak',  
 'time_for_pred': 0.0638,  
 'correct': True}]
```

Wonderful!

And just like before, since our ViT model's predictions are in the form of a list of dictionaries, we can easily turn them into a pandas DataFrame for further inspection.

In [40]:

Out[40]:

	image_path	class_name	pred_prob	pred_class	time_for_pred	correct
0	data/pizza_steak_sushi_20_percent/test/steak/8...	steak	0.9933	steak	0.1313	True
1	data/pizza_steak_sushi_20_percent/test/steak/3...	steak	0.9893	steak	0.0638	True
2	data/pizza_steak_sushi_20_percent/test/steak/2...	steak	0.9971	steak	0.0627	True
3	data/pizza_steak_sushi_20_percent/test/steak/3...	steak	0.7685	steak	0.0632	True
4	data/pizza_steak_sushi_20_percent/test/steak/7...	steak	0.9499	steak	0.0641	True

How many predictions did our ViT model get correct?

In [41]:

```
True      148
False      2
Name: correct, dtype: int64
```

Woah!

Our ViT model did a little better than our EffNetB2 model in terms of correct predictions, only two samples wrong across the whole test dataset.

As an extension you might want to visualize the ViT model's wrong predictions and see if there's any reason why it might've got them wrong.

How about we calculate how long the ViT model took per prediction?

In [42]:

```
ViT average time per prediction: 0.0641 seconds
```

Well, that looks a little slower than our EffNetB2 model's average time per prediction but how does it look in terms of our second criteria: speed?

For now, let's add the value to our `vit_stats` dictionary so we can compare it to our EffNetB2 model's stats.

**Note:** The average time per prediction values will be highly dependent on the hardware you make them on. For

example, for the ViT model, my average time per prediction (on the CPU) was 0.0693-0.0777 seconds on my local deep learning PC with an Intel i9 CPU. Whereas on Google Colab, my average time per prediction with the ViT model was 0.6766-0.7113 seconds.

In [43]:

```
{'test_loss': 0.06418210905976593,  
 'test_acc': 0.984659090909091,  
 'number_of_parameters': 85800963,  
 'model_size (MB)': 327,  
 'time_per_pred_cpu': 0.0641}
```

Out[43]:

## 6. Comparing model results, prediction times and size

Our two best model contenders have been trained and evaluated.

Now let's put them head to head and compare across their different statistics.

To do so, let's turn our `effnetb2_stats` and `vit_stats` dictionaries into a pandas DataFrame.

We'll add a column to view the model names as well as the convert the test accuracy to a whole percentage rather than decimal.

In [44]:

Out[44]:

	test_loss	test_acc	number_of_parameters	model_size (MB)	time_per_pred_cpu	model
0	0.281287	96.88	7705221	29	0.0269	EffNetB2
1	0.064182	98.47	85800963	327	0.0641	ViT

Wonderful!

It seems our models are quite close in terms of overall test accuracy but how do they look across the other fields?

One way to find out would be to divide the ViT model statistics by the EffNetB2 model statistics to find out the different ratios between the models.

Let's create another DataFrame to do so.

In [45]:

Out[45]:

	test_loss	test_acc	number_of_parameters	model_size (MB)	time_per_pred_cpu
<b>ViT to EffNetB2 ratios</b>	0.228173	1.016412	11.135432	11.275862	2.3829

It seems our ViT model outperforms the EffNetB2 model across the performance metrics (test loss, where lower is better and test accuracy, where higher is better) but at the expense of having:

- 11x+ the number of parameters.

- 11x+ the model size.
- 2.5x+ the prediction time per image.

Are these tradeoffs worth it?

Perhaps if we had unlimited compute power but for our use case of deploying the FoodVision Mini model to a smaller device (e.g. a mobile phone), we'd likely start out with the EffNetB2 model for faster predictions at a slightly reduced performance but dramatically smaller

## 6.1 Visualizing the speed vs. performance tradeoff

We've seen that our ViT model outperforms our EffNetB2 model in terms of performance metrics such as test loss and test accuracy.

However, our EffNetB2 model makes performs predictions faster and has a much small model size.

**Note:** Performance or inference time is also often referred to as "latency".

How about we make this fact visual?

We can do so by creating a plot with matplotlib:

1. Create a scatter plot from the comparison DataFrame to compare EffNetB2 and ViT `time_per_pred_cpu` and `test_acc` values.
2. Add titles and labels respective of the data and customize the fontsize for aesthetics.
3. Annotate the samples on the scatter plot from step 1 with their appropriate labels (the model names).
4. Create a legend based on the model sizes (`model_size (MB)`).

In [46]:











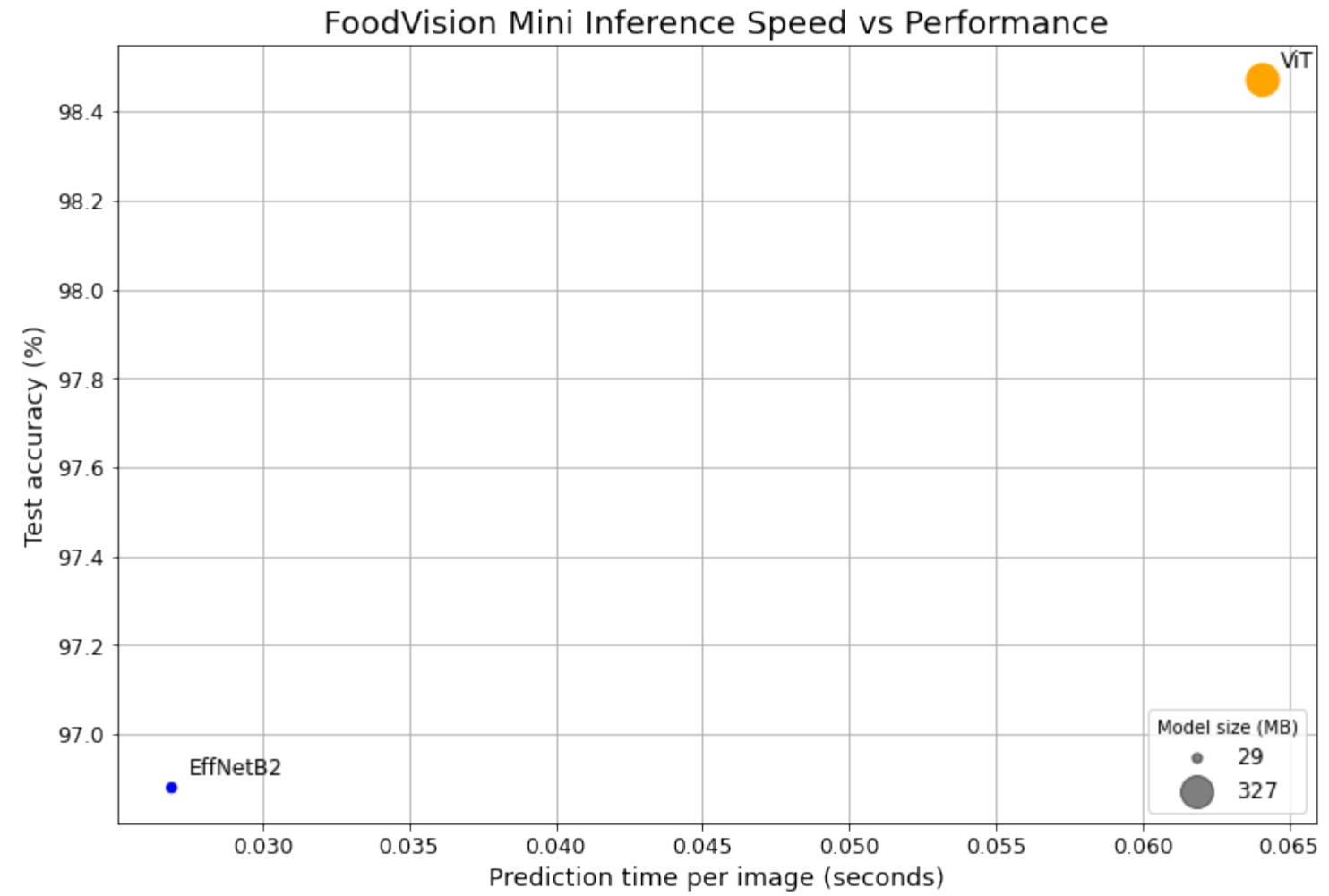












Woah!

The plot really visualizes the **speed vs. performance tradeoff**, in other words, when you have a larger, better performing deep model (like our ViT model), it *generally* takes longer to perform inference (higher latency).

There are exceptions to the rule and new research is being published all the time to help make larger models perform faster.

And it can be tempting to just deploy the *best* performing model but it's also good to take into consideration where the model is going to be performing.

In our case, the differences between our model's performance levels (on the test loss and test accuracy) aren't too extreme.

But since we'd like to put an emphasis on speed to begin with, we're going to stick with deploying EffNetB2 since it's faster and has a much smaller footprint.

**Note:** Prediction times will be different across different hardware types (e.g. Intel i9 vs Google Colab CPU vs GPU) so it's important to think about and test where your model is going to end up. Asking questions like "where is the model going to be run?" or "what is the ideal scenario for running the model?" and then running experiments to

try and provide answers on your way to deployment is very helpful.

## 7. Bringing FoodVision Mini to life by creating a Gradio demo

We've decided we'd like to deploy the EffNetB2 model (to begin with, this could always be changed later).

So how can we do that?

There are several ways to deploy a machine learning model each with specific use cases (as discussed above).

We're going to be focused on perhaps the quickest and certainly one of the most fun ways to get a model deployed to the internet.

And that's by using [Gradio](#).

What's Gradio?

The homepage describes it beautifully:

Gradio is the fastest way to demo your machine learning model with a friendly web interface so that anyone can use it, anywhere!

Why create a demo of your models?

Because metrics on the test set look nice but you never really know how your model performs until you use it in the wild.

So let's get deploying!

We'll start by importing Gradio with the common alias `gr` and if it's not present, we'll install it.

In [47]:

Gradio version: 3.1.4

Gradio ready!

Let's turn FoodVision Mini into a demo application.

## 7.1 Gradio overview

The overall premise of Gradio is very similar to what we've been repeating throughout the course.

What are our **inputs** and **outputs**?

And how should we get there?

Well that's what our machine learning model does.

```
inputs -> ML model -> outputs
```

In our case, for FoodVision Mini, our inputs are images of food, our ML model is EffNetB2 and our outputs are classes of food (pizza, steak or sushi).

```
images of food -> EffNetB2 -> outputs
```

Though the concepts of inputs and outputs can be bridged to almost any other kind of ML problem.

Your inputs and outputs might be any combination of the following:

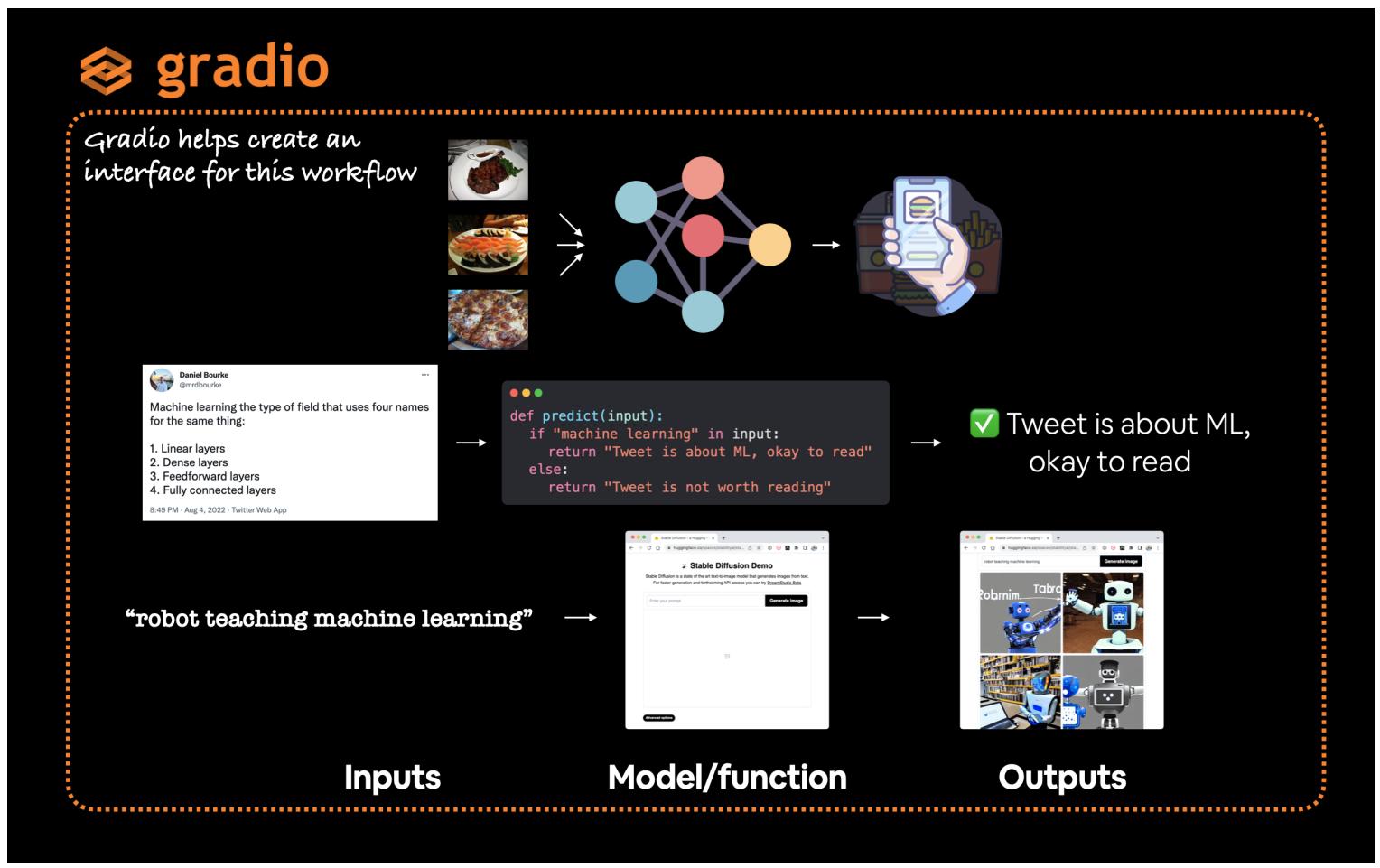
- Images
- Text
- Video
- Tabular data
- Audio
- Numbers
- & more

And the ML model you build will depend on your inputs and outputs.

Gradio emulates this paradigm by creating an interface (`gradio.Interface()`) to from inputs to outputs.

```
gradio.Interface(fn, inputs, outputs)
```

Where, `fn` is a Python function to map the `inputs` to the `outputs`.



Gradio provides a very helpful `Interface` class to easily create an inputs -> model/function -> outputs workflow where the inputs and outputs could be almost anything you want. For example, you might input Tweets (text) to see if they're about machine learning or not or [input a text prompt to generate images](#).

**Note:** Gradio has a vast number of possible `inputs` and `outputs` options known as "Components" from images to text to numbers to audio to videos and more. You can see all of these in the [Gradio Components documentation](#).

## 7.2 Creating a function to map our inputs and outputs

To create our FoodVision Mini demo with Gradio, we'll need a function to map our inputs to our outputs.

We created a function earlier called `pred_and_store()` to make predictions with a given model across a list of target files and store them in a list of dictionaries.

How about we create a similar function but this time focusing on a making a prediction on a single image with our EffNetB2 model?

More specifically, we want a function that takes an image as input, preprocesses (transforms) it, makes a prediction with EffNetB2 and then returns the prediction (pred or pred label for short) as well as the prediction probability (pred

prob).

And while we're here, let's return the time it took to do so too:

```
input: image -> transform -> predict with EffNetB2 -> output: pred, pred prob, time taken
```

This will be our `fn` parameter for our Gradio interface.

First, let's make sure our EffNetB2 model is on the CPU (since we're sticking with CPU-only predictions, however you could change this if you have access to a GPU).

In [48]:

```
device(type='cpu')
```

And now let's create a function called `predict()` to replicate the workflow above.

In [49]:













Beautiful!

Now let's see our function in action by performing a prediction on a random image from the test dataset.

We'll start by getting a list of all the image paths from the test directory and then randomly selecting one.

Then we'll open the randomly selected image with `PIL.Image.open()`.

Finally, we'll pass the image to our `predict()` function.

In [50]:





```
[INFO] Predicting on image at path: data/pizza_steak_sushi_20_percent/test/pizza/3770514.jpg
Prediction label and probability dictionary:
{'pizza': 0.9785208702087402, 'steak': 0.01169557310640812, 'sushi': 0.009783552028238773}
Prediction time: 0.027 seconds
```

Nice!

Running the cell above a few times we can see different prediction probabilities for each label from our EffNetB2 model as well as the time it took per prediction.

## 7.3 Creating a list of example images

Our `predict()` function enables us to go from inputs -> transform -> ML model -> outputs.

Which is exactly what we need for our Graido demo.

But before we create the demo, let's create one more thing: a list of examples.

Gradio's `Interface` class takes a list of `examples` of as an optional parameter (`gradio.Interface(examples=List[Any])`).

And the format for the `examples` parameter is a list of lists.

So let's create a list of lists containing random filepaths to our test images.

Three examples should be enough.

In [51]:

Out[51]:

```
[['data/pizza_steak_sushi_20_percent/test/sushi/804460.jpg'],
 ['data/pizza_steak_sushi_20_percent/test/steak/746921.jpg'],
 ['data/pizza_steak_sushi_20_percent/test/steak/2117351.jpg']]
```

Perfect!

Our Gradio demo will showcase these as example inputs to our demo so people can try it out and see what it does without uploading any of their own data.

## 7.4 Building a Gradio interface

Time to put everything together and bring our FoodVision Mini demo to life!

Let's create a Gradio interface to replicate the workflow:

```
input: image -> transform -> predict with EffNetB2 -> output: pred, pred prob, time taken
```

We can do with the `gradio.Interface()` class with the following parameters:

- `fn` - a Python function to map `inputs` to `outputs`, in our case, we'll use our `predict()` function.
- `inputs` - the input to our interface, such as an image using `gradio.Image()` or "image".
- `outputs` - the output of our interface once the `inputs` have gone through the `fn`, such as a label using `gradio.Label()` (for our model's predicted labels) or number using `gradio.Number()` (for our model's prediction time).
- **Note:** Gradio comes with many in-built `inputs` and `outputs` options known as "[Components](#)".
- `examples` - a list of examples to showcase for the demo.
- `title` - a string title of the demo.
- `description` - a string description of the demo.
- `article` - a reference note at the bottom of the demo.

Once we've created our demo instance of `gr.Interface()`, we can bring it to life using `gradio.Interface().launch()` or `demo.launch()` command.

Easy!

In [52]:













Running on local URL: <http://127.0.0.1:7860/>  
Running on public URL: <https://27541.gradio.app>

This share link expires in 72 hours. For free permanent hosting, check out Spaces: <https://huggingface.co/spaces>

Out[52]:

```
(<gradio.routes.App at 0x7f122dd0f0d0>,
 'http://127.0.0.1:7860/',
 'https://27541.gradio.app')
```

Time to put everything together and bring our FoodVision Mini demo to life!

Let's create a Gradio interface to replicate the workflow:

```
input: image -> transform -> predict with EffNetB2 -> output: pred, pred prob, time taken
```

We can do with the `gradio.Interface()` class with the following parameters:

- `fn` - a Python function to map `inputs` to `outputs`, in our case, we'll use our `predict()` function.
- `inputs` - the input to our interface, such as an image using `gradio.Image()` or "image".
- `outputs` - the output of our interface once the `inputs` have gone through the `fn`, such as a label using `gradio.Label()` (for our model's predicted labels) or number using `gradio.Number()` (for our model's prediction time).

◦ Note: Gradio comes with many in-built inputs and outputs options known as "[Components](#)".

- `examples` - a list of examples to showcase for the demo.
- `title` - a string title of the demo.
- `description` - a string description of the demo.
- `article` - a reference note at the bottom of the demo.

Once we've created our demo instance of `gr.Interface()`, we can bring it to life using `gradio.Interface().launch()` or `demo.launch()` command.

Easy!

```
[ ] 1 import gradio as gr
2
3 # Create title, description and article strings
4 title = "Foodvision Mini 🍕"
5 description = "An EfficientNetB2 feature extractor computer vision model to classify images of food as pizza, steak or sushi."
6 article = "Created at [09. PyTorch Model Deployment](https://www.learnpytorch.io/09_pytorch_model_deployment/)."
```

*FoodVision Mini Gradio demo running in Google Colab and in the browser (the link when running from Google Colab only lasts for 72 hours). You can see the [permanent live demo on Hugging Face Spaces](#).*

Woohoo!!! What an epic demo!!!

FoodVision Mini has officially come to life in an interface someone could use and try out.

If you set the parameter `share=True` in the `launch()` method, Gradio also provides you with a shareable link such as <https://123XYZ.gradio.app> (this link is an example only and likely expired) which is valid for 72-hours.

The link provides a proxy back to the Gradio interface you launched.

For more permanent hosting, you can upload your Gradio app to [Hugging Face Spaces](#) or anywhere that runs Python code.

## 8. Turning our FoodVision Mini Gradio Demo into a deployable app

We've seen our FoodVision Mini model come to life through a Gradio demo.

But what if we wanted to share it with our friends?

Well, we could use the provided Gradio link, however, the shared link only lasts for 72-hours.

To make our FoodVision Mini demo more permanent, we can package it into an app and upload it to [Hugging Face Spaces](#).

## 8.1 What is Hugging Face Spaces?

Hugging Face Spaces is a resource that allows you to host and share machine learning apps.

Building a demo is one of the best ways to showcase and test what you've done.

And Spaces allows you to do just that.

You can think of Hugging Face as the GitHub of machine learning.

If having a good GitHub portfolio showcases your coding abilities, having a good Hugging Face portfolio can showcase your machine learning abilities.

**Note:** There are many other places we could upload and host our Gradio app such as, Google Cloud, AWS (Amazon Web Services) or other cloud vendors, however, we're going to use Hugging Face Spaces due to the ease of use and wide adoption by the machine learning community.

## 8.2 Deployed Gradio app structure

To upload our demo Gradio app, we'll want to put everything relating to it into a single directory.

For example, our demo might live at the path `demos/foodvision_mini/` with the file structure:

```
demos/
└── foodvision_mini/
    ├── 09_pretrained_effnetb2_feature_extractor_pizza_steak_sushi_20_percent.pth
    ├── app.py
    └── examples/
        ├── example_1.jpg
        ├── example_2.jpg
        └── example_3.jpg
    ├── model.py
    └── requirements.txt
```

Where:

- `09_pretrained_effnetb2_feature_extractor_pizza_steak_sushi_20_percent.pth` is our trained PyTorch model file.
- `app.py` contains our Gradio app (similar to the code that launched the app).
  - **Note:** `app.py` is the default filename used for Hugging Face Spaces, if you deploy your app there, Spaces will by default look for a file called `app.py` to run. This is changable in settings.

- `examples/` contains example images to use with our Gradio app.
- `model.py` contains the model definition as well as any transforms associated with the model.
- `requirements.txt` contains the dependencies to run our app such as `torch`, `torchvision` and `gradio`.

Why this way?

Because it's one of the simplest layouts we could begin with.

Our focus is: *experiment, experiment, experiment!*

The quicker we can run smaller experiments, the better our bigger ones will be.

We're going to work towards recreating the structure above but you can see a live demo app running on Hugging Face Spaces as well as the file structure:

- [Live Gradio demo of FoodVision Mini](#) .
- [FoodVision Mini file structure on Hugging Face Spaces](#).

## 8.3 Creating a `demos` folder to store our FoodVision Mini app files

To begin, let's first create a `demos/` directory to store all of our FoodVision Mini app files.

We can do with Python's `pathlib.Path("path_to_dir")` to establish the directory path and `pathlib.Path("path_to_dir").mkdir()` to create it.

In [53]:







## 8.4 Creating a folder of example images to use with our FoodVision Mini demo

Now we've got a directory to store our FoodVision Mini demo files, let's add some examples to it.

Three example images from the test dataset should be enough.

To do so we'll:

1. Create an `examples/` directory within the `demos/foodvision_mini` directory.
2. Choose three random images from the test dataset and collect their filepaths in a list.
3. Copy the three random images from the test dataset to the `demos/foodvision_mini/examples/` directory.

In [54]:





```
[INFO] Copying data/pizza_steak_sushi_20_percent/test/sushi/592799.jpg to demos/foodvision_mini/examples/592799.jpg
[INFO] Copying data/pizza_steak_sushi_20_percent/test/steak/3622237.jpg to demos/foodvision_mini/examples/3622237.jpg
[INFO] Copying data/pizza_steak_sushi_20_percent/test/pizza/2582289.jpg to demos/foodvision_mini/examples/2582289.jpg
```

Now to verify our examples are present, let's list the contents of our `demos/foodvision_mini/examples/` directory with `os.listdir()` and then format the filepaths into a list of lists (so it's compatible with Gradio's `gradio.Interface()` `example` parameter).

In [55]:

Out[55]:

```
[['examples/3622237.jpg'], ['examples/592799.jpg'], ['examples/2582289.jpg']]
```

## 8.5 Moving our trained EffNetB2 model to our FoodVision Mini demo directory

We previously saved our FoodVision Mini EffNetB2 feature extractor model under

```
models/09_pretrained_effnetb2_feature_extractor_pizza_steak_sushi_20_percent.pth.
```

And rather double up on saved model files, let's move our model to our `demos/foodvision_mini` directory.

We can do so using Python's `shutil.move()` method and passing in `src` (the source path of the target file) and `dst` (the destination path of the target file to be moved to) parameters.

In [56]:







```
[INFO] Attempting to move models/09_pretrained_effnetb2_feature_extractor_pizza_steak_sushi_20_percent.pth to demos/foodvision_mini/09_pretrained_effnetb2_feature_extractor_pizza_steak_sushi_20_percent.pth
[INFO] Model move complete.
```

## 8.6 Turning our EffNetB2 model into a Python script (`model.py`)

Our current model's `state_dict` is saved to

```
demos/foodvision_mini/09_pretrained_effnetb2_feature_extractor_pizza_steak_sushi_20_percent.pth.
```

To load it in we can use `model.load_state_dict()` along with `torch.load()`.

**Note:** For a refresh on saving and loading a model (or a model's `state_dict` in PyTorch, see [01. PyTorch Workflow Fundamentals section 5: Saving and loading a PyTorch model](#) or see the PyTorch recipe for [What is a `state\_dict` in PyTorch?](#)

But before we can do this, we first need a way to instantiate a `model`.

To do this in a modular fashion we'll create a script called `model.py` which contains our `create_effnetb2_model()` function we created in [section 3.1: Creating a function to make an EffNetB2 feature extractor](#).

That way we can import the function in *another* script (see `app.py` below) and then use it to create our EffNetB2 `model` instance as well as get its appropriate transforms.

Just like in [05. PyTorch Going Modular](#), we'll use the `%%writefile path/to/file` magic command to turn a cell of code into a file.

In [57]:















```
Writing demos/foodvision_mini/model.py
```

## 8.7 Turning our FoodVision Mini Gradio app into a Python script (`app.py`)

We've now got a `model.py` script as well as a path to a saved model `state_dict` that we can load in.

Time to construct `app.py`.

We call it `app.py` because by default when you create a HuggingFace Space, it looks for a file called `app.py` to run and host (though you can change this in settings).

Our `app.py` script will put together all of the pieces of the puzzle to create our Gradio demo and will have four main parts:

- 1. Imports and class names setup** - Here we'll import the various dependencies for our demo including the `create_effnetb2_model()` function from `model.py` as well as setup the different class names for our FoodVision Mini app.
- 2. Model and transforms preparation** - Here we'll create an EffNetB2 model instance along with the transforms to go with it and then we'll load in the saved model weights/`state_dict`. When we load the model we'll also set `map_location=torch.device("cpu")` in `torch.load()` so our model gets loaded onto the CPU regardless of the device it trained on (we do this because we won't necessarily have a GPU when we deploy and we'll get an error if our model is trained on GPU but we try to deploy it to CPU without explicitly saying so).
- 3. Predict function** - Gradio's `gradio.Interface()` takes a `fn` parameter to map inputs to outputs, our `predict()` function will be the same as the one we defined above in [section 7.2: Creating a function to map our inputs and outputs](#), it will take in an image and then use the loaded transforms to preprocess it before using the loaded model to make a prediction on it.
  - Note:** We'll have to create the example list on the fly via the `examples` parameter. We can do so by creating a list of the files inside the `examples/` directory with: `[["examples/" + example] for example in os.listdir("examples")]`.
- 4. Gradio app** - This is where the main logic of our demo will live, we'll create a `gradio.Interface()` instance called `demo` to put together our inputs, `predict()` function and outputs. And we'll finish the script by calling

`demo.launch()` to launch our FoodVision Mini demo!

In [58]:



































```
Writing demos/foodvision_mini/app.py
```

## 8.8 Creating a requirements file for FoodVision Mini ( `requirements.txt` )

The last file we need to create for our FoodVision Mini app is a `requirements.txt` file.

This will be a text file containing all of the required dependencies for our demo.

When we deploy our demo app to Hugging Face Spaces, it will search through this file and install the dependencies we define so our app can run.

The good news is, there's only three!

1. `torch==1.12.0`
2. `torchvision==0.13.0`
3. `gradio==3.1.4`

The "`==1.12.0`" states the version number to install.

Defining the version number is not 100% required but we will for now so if any breaking updates occur in future releases, our app still runs (PS if you find any errors, feel free to post on the course [GitHub Issues](#)).

In [59]:

```
Writing demos/foodvision_mini/requirements.txt
```

Nice!

We've officially got all the files we need to deploy our FoodVision Mini demo!

## 9. Deploying our FoodVision Mini app to HuggingFace Spaces

We've got a file containing our FoodVision Mini demo, now how do we get it to run on Hugging Face Spaces?

There are two main options for uploading to a Hugging Face Space (also called a [Hugging Face Repository](#), similar to a git repository):

1. [Uploading via the Hugging Face Web interface \(easiest\).](#)

2. [Uploading via the command line or terminal.](#)

- **Bonus:** You can also use the [huggingface\\_hub library](#) to interact with Hugging Face, this would be a good extension to the above two options.

Feel free to read the documentation on both options but we're going to go with option two.

**Note:** To host anything on Hugging Face, you will to [sign up for a free Hugging Face account](#).

### 9.1 Downloading our FoodVision Mini app files

Let's check out the demo files we've got inside `demos/foodvision_mini`.

To do so we can use the `!ls` command followed by the target filepath.

`ls` stands for "list" and the `!` means we want to execute the command at the shell level.

In [60]:

```
09_pretrained_effnetb2_feature_extractor_pizza_steak_sushi_20_percent.pth
app.py
examples
model.py
requirements.txt
```

These are all files that we've created!

To begin uploading our files to Hugging Face, let's now download them from Google Colab (or wherever you're running this notebook).

To do so, we'll first compress the files into a single zip folder via the command:

```
zip -r ../foodvision_mini.zip * -x "*.pyc" "*.ipynb" "*__pycache__*" "*ipynb_checkpoints"
```

Where:

- `zip` stands for "zip" as in "please zip together the files in the following directory".
- `-r` stands for "recursive" as in, "go through all of the files in the target directory".
- `../foodvision_mini.zip` is the target directory we'd like our files to be zipped to.
- `*` stands for "all the files in the current directory".
- `-x` stands for "exclude these files".

We can download our zip file from Google Colab using `google.colab.files.download("demos/foodvision_mini.zip")` (we'll put this inside a `try` and `except` block just in case we're not running the code inside Google Colab, and if so we'll print a message saying to manually download the files).

Let's try it out!

In [61]:



```
updating: 09_pretrained_effnetb2_feature_extractor_pizza_steak_sushi_20_percent.pth (deflated 8%)
updating: app.py (deflated 57%)
updating: examples/ (stored 0%)
updating: examples/3622237.jpg (deflated 0%)
updating: examples/592799.jpg (deflated 1%)
updating: examples/2582289.jpg (deflated 17%)
updating: model.py (deflated 56%)
updating: requirements.txt (deflated 4%)
```

Not running in Google Colab, can't use `google.colab.files.download()`, please manually download.

Woohoo!

Looks like our `zip` command was successful.

If you're running this notebook in Google Colab, you should see a file start to download in your browser.

Otherwise, you can see the `foodvision_mini.zip` folder (and more) on the [course GitHub](#) under the `demos/ directory`.

## 9.2 Running our FoodVision Mini demo locally

If you download the `foodvision_mini.zip` file, you can test it locally by:

1. Unzipping the file.
2. Opening terminal or a command line prompt.
3. Changing into the `foodvision_mini` directory (`cd foodvision_mini`).
4. Creating an environment (`python3 -m venv env`).
5. Activating the environment (`source env/bin/activate`).
6. Installing the requirements (`pip install -r requirements.txt`, the "`-r`" is for recursive).
  - **Note:** This step may take 5-10 minutes depending on your internet connection. And if you're facing errors, you may need to upgrade pip first: `pip install --upgrade pip`.
7. Run the app (`python3 app.py`).

This should result in a Gradio demo just like the one we built above running locally on your machine at a URL such

as `http://127.0.0.1:7860/`.

**Note:** If you run the app locally and you notice a `flagged/` directory appear, it contains samples that have been "flagged".

For example, if someone tries the demo and the model produces an incorrect result, the sample can be "flagged" and reviewed for later.

For more on flagging in Gradio, see the [flagging documentation](#).

## 9.3 Uploading to Hugging Face

We've verified our FoodVision Mini app works locally, however, the fun of creating a machine learning demo is to show it to other people and allow them to use it.

To do so, we're going to upload our FoodVision Mini demo to Hugging Face.

**Note:** The following series of steps uses a Git (a file tracking system) workflow. For more on how Git works, I'd recommend going through the [Git and GitHub for Beginners tutorial](#) on freeCodeCamp.

1. [Sign up](#) for a Hugging Face account.
2. Start a new Hugging Face Space by going to your profile and then [clicking "New Space"](#).
  - **Note:** A Space in Hugging Face is also known as a "code repository" (a place to store your code/files) or "repo" for short.
3. Give the Space a name, for example, mine is called `mrdourke/foodvision_mini`, you can see it here: [https://huggingface.co/spaces/mrdourke/foodvision\\_mini](https://huggingface.co/spaces/mrdourke/foodvision_mini)
4. Select a license (I used [MIT](#)).
5. Select Gradio as the Space SDK (software development kit).
  - **Note:** You can use other options such as Streamlit but since our app is built with Gradio, we'll stick with that.
6. Choose whether your Space is it's public or private (I selected public since I'd like my Space to be available to others).
7. Click "Create Space".
8. Clone the repo locally by running something like: `git clone https://huggingface.co/spaces/[YOUR_USERNAME]/[YOUR_SPACE_NAME]` in terminal or command prompt.
  - **Note:** You can also add files via uploading them under the "Files and versions" tab.
9. Copy/move the contents of the downloaded `foodvision_mini` folder to the cloned repo folder.
10. To upload and track larger files (e.g. files over 10MB or in our case, our PyTorch model file) you'll need to [install Git LFS](#) (which stands for "git large file storage").
  11. After you've installed Git LFS, you can activate it by running `git lfs install`.
  12. In the `foodvision_mini` directory, track the files over 10MB with Git LFS with `git lfs track "*..file_extension"`.
    - Track EffNetB2 PyTorch model file with `git lfs track "09_pretrained_effnetb2_feature_extractor_pizza_steak_sushi_20_percent.pth"`.
  13. Track `.gitattributes` (automatically created when cloning from HuggingFace, this file will help ensure our larger files are tracked with Git LFS). You can see an example `.gitattributes` file on the [FoodVision Mini Hugging Face Space](#).
    - `git add .gitattributes`
  14. Add the rest of the `foodvision_mini` app files and commit them with:

- `git add *`
- `git commit -m "first commit"`

15. Push (upload) the files to Hugging Face:

- `git push`

16. Wait 3-5 minutes for the build to happen (future builds are faster) and your app to become live!

If everything worked, you should see a live running example of our FoodVision Mini Gradio demo like the one here:

[https://huggingface.co/spaces/mrdbourke/foodvision\\_mini](https://huggingface.co/spaces/mrdbourke/foodvision_mini)

And we can even embed our FoodVision Mini Gradio demo into our notebook as an `iframe` with

`IPython.display.IFrame` and a link to our space in the format

`https://hf.space/embed/[YOUR_USERNAME]/[YOUR_SPACE_NAME]/+.`

In [62]:

Out[62]:

## 10. Creating FoodVision Big

We've spent the past few sections and chapters working on bringing FoodVision Mini to life.

And now we've seen it working in a live demo, how about we step things up a notch?

How?

FoodVision Big!

Since FoodVision Mini is trained on pizza, steak and sushi images from the [Food101 dataset](#) (101 classes of food x 1000 images each), how about we make FoodVision Big by training a model on all 101 classes!

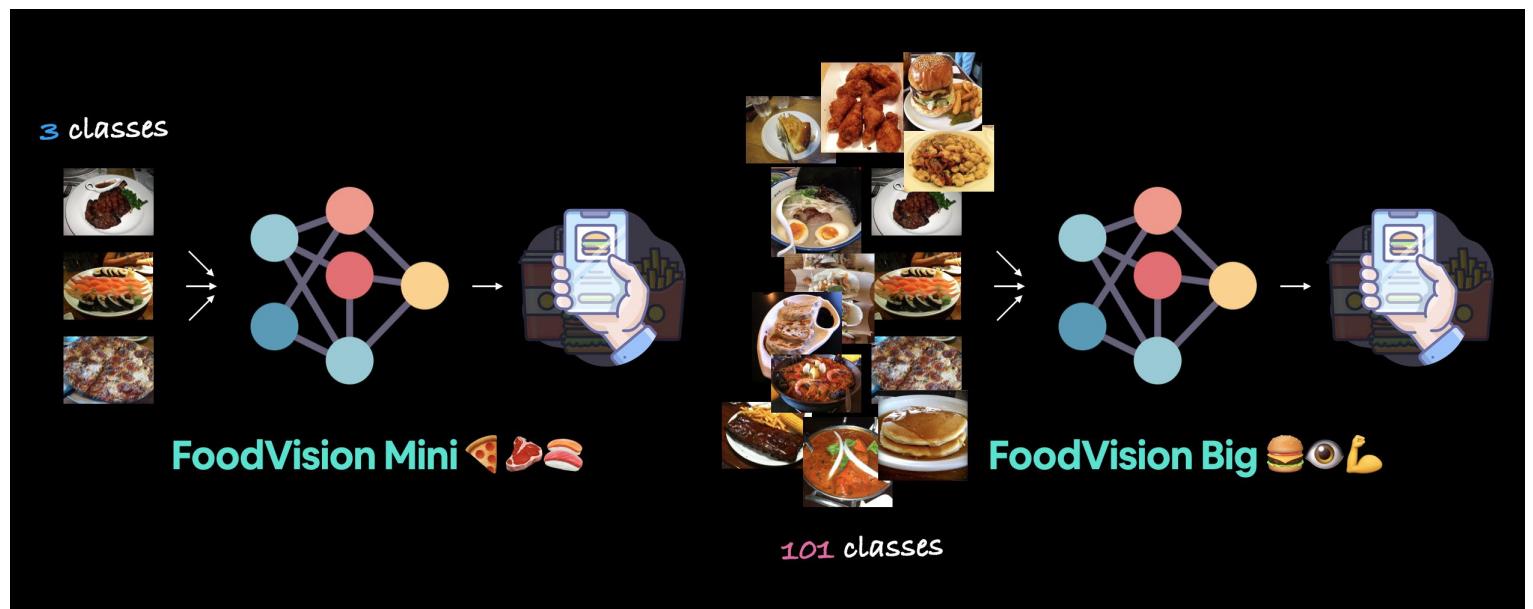
We'll go from three classes to 101!

From pizza, steak, sushi to pizza, steak, sushi, hot dog, apple pie, carrot cake, chocolate cake, french fries, garlic bread, ramen, nachos, tacos and more!

How?

Well, we've got all the steps in place, all we have to do is alter our EffNetB2 model slightly as well as prepare a different dataset.

To finish Milestone Project 3, let's recreate a Gradio demo similar to FoodVision Mini (three classes) but for FoodVision Big (101 classes).



*FoodVision Mini works with three food classes: pizza, steak and sushi. And FoodVision Big steps it up a notch to work across 101 food classes: all of the [classes in the Food101 dataset](#).*

## 10.1 Creating a model and transforms for FoodVision Big

When creating FoodVision Mini we saw that the EffNetB2 model was a good tradeoff between speed and performance (it performed well with a fast speed).

So we'll continue using the same model for FoodVision Big.

We can create an EffNetB2 feature extractor for Food101 by using our `create_effnetb2_model()` function we created above, in [section 3.1](#), and passing it the parameter `num_classes=101` (since Food101 has 101 classes).

In [63]:

Beautiful!

Let's now get a summary of our model.

In [64]:



Layer (type (var_name))	Input Shape	Output Shape	Param #	Trainable
EfficientNet (EfficientNet)	[1, 3, 224, 224]	[1, 101]	--	Partial
Sequential (features)	[1, 3, 224, 224]	[1, 1408, 7, 7]	--	False
└Conv2dNormActivation (0)	[1, 3, 224, 224]	[1, 32, 112, 112]	--	False
└Conv2d (0)	[1, 3, 224, 224]	[1, 32, 112, 112]	(864)	False
└BatchNorm2d (1)	[1, 32, 112, 112]	[1, 32, 112, 112]	(64)	False
└SiLU (2)	[1, 32, 112, 112]	[1, 32, 112, 112]	--	--
└Sequential (1)	[1, 32, 112, 112]	[1, 16, 112, 112]	--	False
└MBConv (0)	[1, 32, 112, 112]	[1, 16, 112, 112]	(1,448)	False
└MBConv (1)	[1, 16, 112, 112]	[1, 16, 112, 112]	(612)	False
└Sequential (2)	[1, 16, 112, 112]	[1, 24, 56, 56]	--	False
└MBConv (0)	[1, 16, 112, 112]	[1, 24, 56, 56]	(6,004)	False
└MBConv (1)	[1, 24, 56, 56]	[1, 24, 56, 56]	(10,710)	False
└MBConv (2)	[1, 24, 56, 56]	[1, 24, 56, 56]	(10,710)	False
└Sequential (3)	[1, 24, 56, 56]	[1, 48, 28, 28]	--	False
└MBConv (0)	[1, 24, 56, 56]	[1, 48, 28, 28]	(16,518)	False
└MBConv (1)	[1, 48, 28, 28]	[1, 48, 28, 28]	(43,308)	False
└MBConv (2)	[1, 48, 28, 28]	[1, 48, 28, 28]	(43,308)	False
└Sequential (4)	[1, 48, 28, 28]	[1, 88, 14, 14]	--	False
└MBConv (0)	[1, 48, 28, 28]	[1, 88, 14, 14]	(50,300)	False
└MBConv (1)	[1, 88, 14, 14]	[1, 88, 14, 14]	(123,750)	False
└MBConv (2)	[1, 88, 14, 14]	[1, 88, 14, 14]	(123,750)	False
└MBConv (3)	[1, 88, 14, 14]	[1, 88, 14, 14]	(123,750)	False
└Sequential (5)	[1, 88, 14, 14]	[1, 120, 14, 14]	--	False
└MBConv (0)	[1, 88, 14, 14]	[1, 120, 14, 14]	(149,158)	False
└MBConv (1)	[1, 120, 14, 14]	[1, 120, 14, 14]	(237,870)	False
└MBConv (2)	[1, 120, 14, 14]	[1, 120, 14, 14]	(237,870)	False
└MBConv (3)	[1, 120, 14, 14]	[1, 120, 14, 14]	(237,870)	False
└Sequential (6)	[1, 120, 14, 14]	[1, 208, 7, 7]	--	False
└MBConv (0)	[1, 120, 14, 14]	[1, 208, 7, 7]	(301,406)	False
└MBConv (1)	[1, 208, 7, 7]	[1, 208, 7, 7]	(686,868)	False
└MBConv (2)	[1, 208, 7, 7]	[1, 208, 7, 7]	(686,868)	False
└MBConv (3)	[1, 208, 7, 7]	[1, 208, 7, 7]	(686,868)	False
└MBConv (4)	[1, 208, 7, 7]	[1, 208, 7, 7]	(686,868)	False
└Sequential (7)	[1, 208, 7, 7]	[1, 352, 7, 7]	--	False
└MBConv (0)	[1, 208, 7, 7]	[1, 352, 7, 7]	(846,900)	False
└MBConv (1)	[1, 352, 7, 7]	[1, 352, 7, 7]	(1,888,920)	False
└Conv2dNormActivation (8)	[1, 352, 7, 7]	[1, 1408, 7, 7]	--	False
└Conv2d (0)	[1, 352, 7, 7]	[1, 1408, 7, 7]	(495,616)	False
└BatchNorm2d (1)	[1, 1408, 7, 7]	[1, 1408, 7, 7]	(2,816)	False
└SiLU (2)	[1, 1408, 7, 7]	[1, 1408, 7, 7]	--	--
└AdaptiveAvgPool2d (avgpool)	[1, 1408, 7, 7]	[1, 1408, 1, 1]	--	--
└Sequential (classifier)	[1, 1408]	[1, 101]	--	True
└Dropout (0)	[1, 1408]	[1, 1408]	--	--
└Linear (1)	[1, 1408]	[1, 101]	142,309	True

Total params: 7,843,303  
 Trainable params: 142,309  
 Non-trainable params: 7,700,994  
 Total mult-adds (M): 657.78

Input size (MB): 0.60  
 Forward/backward pass size (MB): 156.80  
 Params size (MB): 31.37  
 Estimated Total Size (MB): 188.77

Nice!

See how just like our EffNetB2 model for FoodVision Mini the base layers are frozen (these are pretrained on ImageNet) and the outer layers (the `classifier` layers) are trainable with an output shape of `[batch_size, 101]` (`101` for 101 classes in Food101).

Now since we're going to be dealing with a fair bit more data than usual, how about we add a little data augmentation to our transforms (`effnetb2_transforms`) to augment the training data.

**Note:** Data augmentation is a technique used to alter the appearance of an input training sample (e.g. rotating an image or slightly skewing it) to artificially increase the diversity of a training dataset to hopefully prevent overfitting. You can see more on data augmentation in [04. PyTorch Custom Datasets section 6](#).

Let's compose a `torchvision.transforms` pipeline to use `torchvision.transforms.TrivialAugmentWide()` (the same data augmentation used by the PyTorch team in their [computer vision recipes](#)) as well as the `effnetb2_transforms` to transform our training images.

In [65]:

Epic!

Now let's compare `food101_train_transforms` (for the training data) and `effnetb2_transforms` (for the testing/inference data).

In [66]:

Training transforms:

```
Compose(  
    TrivialAugmentWide(num_magnitude_bins=31, interpolation=InterpolationMode.NEAREST, fill=None)  
    ImageClassification(  
        crop_size=[288]  
        resize_size=[288]  
        mean=[0.485, 0.456, 0.406]  
        std=[0.229, 0.224, 0.225]  
        interpolation=InterpolationMode.BICUBIC  
)  
)
```

Testing transforms:

```
ImageClassification(  
    crop_size=[288]  
    resize_size=[288]  
    mean=[0.485, 0.456, 0.406]  
    std=[0.229, 0.224, 0.225])
```

```
    interpolation=InterpolationMode.BICUBIC  
)
```

## 10.2 Getting data for FoodVision Big

For FoodVision Mini, we made our own [custom data splits](#) of the entire Food101 dataset.

To get the whole Food101 dataset, we can use `torchvision.datasets.Food101()`.

We'll first setup a path to directory `data/` to store the images.

Then we'll download and transform the training and testing dataset splits using `food101_train_transforms` and `effnetb2_transforms` to transform each dataset respectively.

**Note:** If you're using Google Colab, the cell below will take ~3-5 minutes to fully run and download the Food101 images from PyTorch.

This is because there is over 100,000 images being downloaded (101 classes x 1000 images per class). If you restart your Google Colab runtime and come back to this cell, the images will have to redownload. Alternatively, if you're running this notebook locally, the images will be cached and stored in the directory specified by the `root` parameter of `torchvision.datasets.Food101()`.

In [67]:











Data downloaded!

Now we can get a list of all the class names using `train_data.classes`.

In [68]:

Out[68]:

```
['apple_pie',
 'baby_back_ribs',
 'baklava',
 'beef_carpaccio',
 'beef_tartare',
 'beet_salad',
 'beignets',
 'bibimbap',
 'bread_pudding',
 'breakfast_burrito']
```

Ho ho! Those are some delicious sounding foods (although I've never heard of "beignets"... update: after a quick Google search, beignets also look delicious).

You can see a full list of the Food101 class names on the course GitHub under [extras/food101\\_class\\_names.txt](https://github.com/learnpytorch/pytorch-tutorial/blob/main/extras/food101_class_names.txt).

## 10.3 Creating a subset of the Food101 dataset for faster experimenting

This is optional.

We don't *need* to create another subset of the Food101 dataset, we could train and evaluate a model across the whole 101,000 images.

But to keep training fast, let's create a 20% split of the training and test datasets.

Our goal will be to see if we can beat the original [Food101 paper's](#) best results with only 20% of the data.

To breakdown the datasets we've used/will use:

Notebook(s)	Project name	Dataset	Number of classes	Training images	Testing images
04, 05, 06, 07, 08	FoodVision Mini (10% data)	Food101 custom split	3 (pizza, steak, sushi)	225	75
07, 08, 09	FoodVision Mini (20% data)	Food101 custom split	3 (pizza, steak, sushi)	450	150
<b>09 (this one)</b>	FoodVision Big (20% data)	Food101 custom split	101 (all Food101 classes)	15150	5050
Extension	FoodVision Big	Food101 all data	101	75750	25250

Can you see the trend?

Just like our model size slowly increased overtime, so has the size of the dataset we've been using for experiments.

**Note:** To truly beat the original Food101 paper's results with 20% of the data, we'd have to train a model on 20% of the training data and then evaluate our model on the *whole* test set rather than the split we created. I'll leave this as an extension exercise for you to try. I'd also encourage you to try training a model on the entire Food101 training dataset.

To make our FoodVision Big (20% data) split, let's create a function called `split_dataset()` to split a given dataset into certain proportions.

We can use `torch.utils.data.random_split()` to create splits of given sizes using the `lengths` parameter.

The `lengths` parameter accepts a list of desired split lengths where the total of the list must equal the overall length of the dataset.

For example, with a dataset of size 100, you could pass in `lengths=[20, 80]` to receive a 20% and 80% split.

We'll want our function to return two splits, one with the target length (e.g. 20% of the training data) and the other with the remaining length (e.g. the remaining 80% of the training data).

Finally, we'll set `generator` parameter to a `torch.manual_seed()` value for reproducibility.

In [69]:





















Dataset split function created!

Now let's test it out by creating a 20% training and testing dataset split of Food101.

In [70]:





```
[INFO] Splitting dataset of length 75750 into splits of size: 15150 (20%), 60600 (80%)
[INFO] Splitting dataset of length 25250 into splits of size: 5050 (20%), 20200 (80%)
(15150, 5050)
```

Out[70]:

Excellent!

## 10.4 Turning our Food101 datasets into `DataLoader`s

Now let's turn our Food101 20% dataset splits into `DataLoader`'s using `torch.utils.data.DataLoader()`.

We'll set `shuffle=True` for the training data only and the batch size to `32` for both datasets.

And we'll set `num_workers` to `4` if the CPU count is available or `2` if it's not (though the value of `num_workers` is very experimental and will depend on the hardware you're using, there's an [active discussion thread about this on the PyTorch forums](#)).

In [71]:



















## 10.5 Training FoodVision Big model

FoodVision Big model and `DataLoader`s ready!

Time for training.

We'll create an optimizer using `torch.optim.Adam()` and a learning rate of `1e-3`.

And because we've got so many classes, we'll also setup a loss function using `torch.nn.CrossEntropyLoss()` with `label_smoothing=0.1`, inline with [torchvision's state-of-the-art training recipe](#).

What's [label smoothing](#)?

Label smoothing is a regularization technique (regularization is another word to describe the process of [preventing overfitting](#)) that reduces the value a model gives to anyone label and spreads it across the other labels.

In essence, rather than a model getting *too confident* on a single label, label smoothing gives a non-zero value to other labels to help aid in generalization.

For example, if a model *without* label smoothing had the following outputs for 5 classes:

```
[0, 0, 0.99, 0.01, 0]
```

A model *with* label smoothing may have the following outputs:

```
[0.01, 0.01, 0.96, 0.01, 0.01]
```

The model is still confident on its prediction of class 3 but giving small values to the other labels forces the model to

at least consider other options.

Finally, to keep things quick, we'll train our model for five epochs using the `engine.train()` function we created in [05. PyTorch Going Modular section 4](#) with the goal of beating the original Food101 paper's result of 56.4% accuracy on the test set.

Let's train our biggest model yet!

**Note:** Running the cell below will take ~15-20 minutes to run on Google Colab. This is because it's training the biggest model with the largest amount of data we've used so far (15,150 training images, 5050 testing images). And it's a reason we decided to split 20% of the full Food101 dataset off before (so training didn't take over an hour).

In [72]:













```
0% |          | 0/5 [00:00<?, ?it/s]
Epoch: 1 | train_loss: 3.6317 | train_acc: 0.2869 | test_loss: 2.7670 | test_acc: 0.4937
Epoch: 2 | train_loss: 2.8615 | train_acc: 0.4388 | test_loss: 2.4653 | test_acc: 0.5387
Epoch: 3 | train_loss: 2.6585 | train_acc: 0.4844 | test_loss: 2.3547 | test_acc: 0.5649
Epoch: 4 | train_loss: 2.5494 | train_acc: 0.5116 | test_loss: 2.3038 | test_acc: 0.5755
Epoch: 5 | train_loss: 2.5006 | train_acc: 0.5239 | test_loss: 2.2805 | test_acc: 0.5810
```

Woohoo!!!!

Looks like we beat the original Food101 paper's results of 56.4% accuracy with only 20% of the training data (though we only evaluated on 20% of the testing data too, to fully replicate the results, we could evaluate on 100% of the testing data).

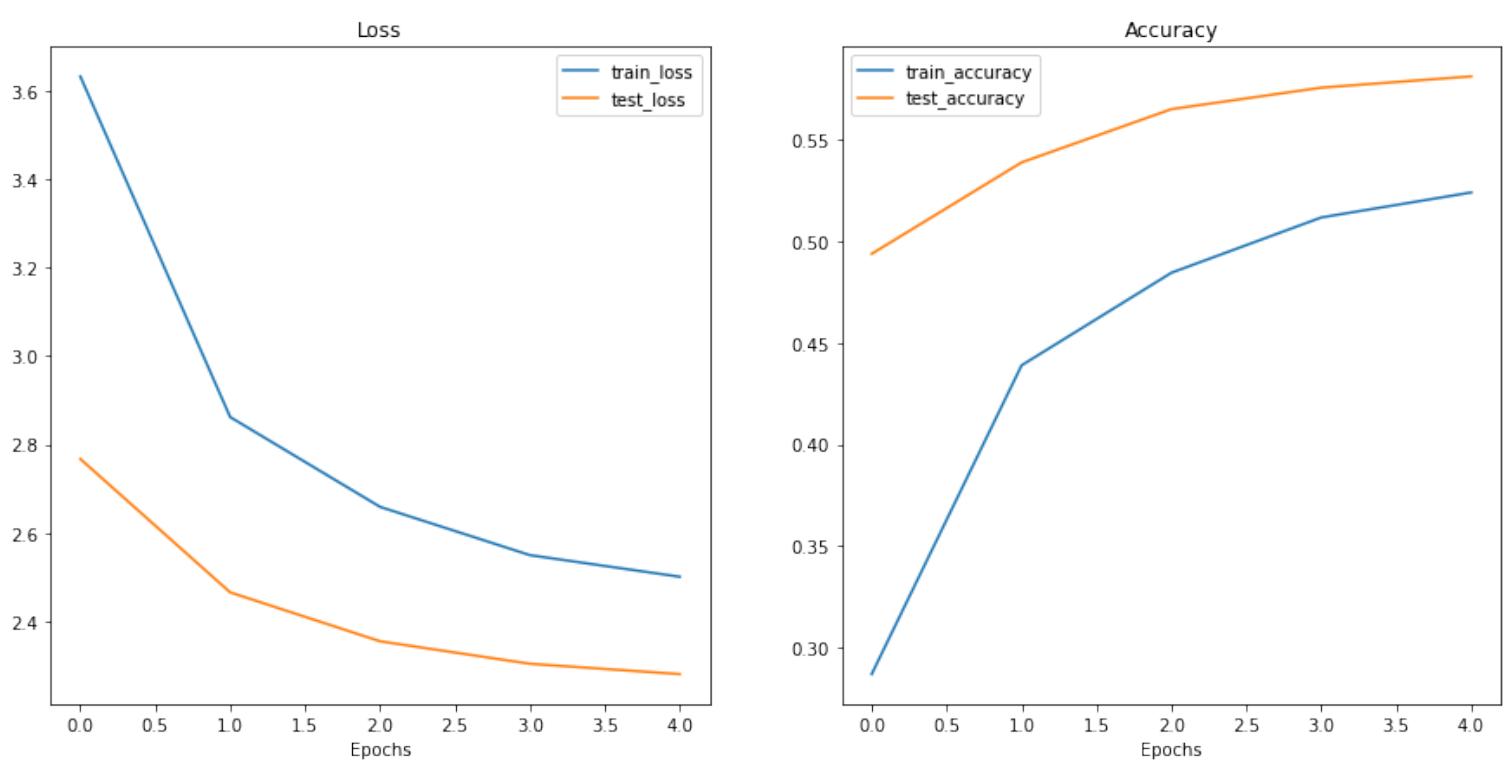
That's the power of transfer learning!

## 10.6 Inspecting loss curves of FoodVision Big model

Let's make our FoodVision Big loss curves visual.

We can do so with the `plot_loss_curves()` function from `helper_functions.py`.

In [73]:



Nice!!!

It looks like our regularization techniques (data augmentation and label smoothing) helped prevent our model from overfitting (the training loss is still higher than the test loss) this indicates our model has a bit more capacity to learn and could improve with further training.

## 10.7 Saving and loading FoodVision Big

Now we've trained our biggest model yet, let's save it so we can load it back in later.

In [74]:

```
[INFO] Saving model to: models/09_pretrained_effnetb2_feature_extractor_food101_20_percent.pth  
Model saved!
```

Before we move on, let's make sure we can load it back in.

We'll do so by creating a model instance first with `create_effnetb2_model(num_classes=101)` (101 classes for all Food101 classes).

And then loading the saved `state_dict()` with `torch.nn.Module.load_state_dict()` and `torch.load()`.

In [75]:

Out[75]:

```
<All keys matched successfully>
```

## 10.8 Checking FoodVision Big model size

Our FoodVision Big model is capable of classifying 101 classes versus FoodVision Mini's 3 classes, a 33.6x increase!

How does this effect the model size?

Let's find out.

In [76]:



```
Pretrained EffNetB2 feature extractor Food101 model size: 30 MB
```

Hmm, it looks like the model size stayed largely the same (30 MB for FoodVision Big and 29 MB for FoodVision Mini) despite the large increase in the number of classes.

This is because all the extra parameters for FoodVision Big are *only* in the last layer (the classifier head).

All of the base layers are the same between FoodVision Big and FoodVision Mini.

Going back up and comparing the model summaries will give more details.

<b>Model</b>	<b>Output shape (num classes)</b>	<b>Trainable parameters</b>	<b>Total parameters</b>	<b>Model size (MB)</b>
FoodVision Mini (EffNetB2 feature extractor)	3	4,227	7,705,221	29
FoodVision Big (EffNetB2 feature extractor)	101	142,309	7,843,303	30

## 11. Turning our FoodVision Big model into a deployable app

We've got a trained and saved EffNetB2 model on 20% of the Food101 dataset.

And instead of letting our model live in a folder all its life, let's deploy it!

We'll deploy our FoodVision Big model in the same way we deployed our FoodVision Mini model, as a Gradio demo

on Hugging Face Spaces.

To begin, let's create a `demos/foodvision_big/` directory to store our FoodVision Big demo files as well as a `demos/foodvision_big/examples` directory to hold an example image to test the demo with.

When we're finished we'll have the following file structure:

```
demos/
  foodvision_big/
    09_pretrained_effnetb2_feature_extractor_food101_20_percent.pth
    app.py
    class_names.txt
    examples/
      example_1.jpg
    model.py
    requirements.txt
```

Where:

- `09_pretrained_effnetb2_feature_extractor_food101_20_percent.pth` is our trained PyTorch model file.
- `app.py` contains our FoodVision Big Gradio app.
- `class_names.txt` contains all of the class names for FoodVision Big.
- `examples/` contains example images to use with our Gradio app.
- `model.py` contains the model definition as well as any transforms associated with the model.
- `requirements.txt` contains the dependencies to run our app such as `torch`, `torchvision` and `gradio`.

In [77]:

## 11.1 Downloading an example image and moving it to the `examples` directory

For our example image, we're going to use the faithful [pizza-dad image](#) (a photo of my dad eating pizza).

So let's download it from the course GitHub via the `!wget` command and then we can move it to `demos/foodvision_big/examples` with the `!mv` command (short for "move").

While we're here we'll move our trained Food101 EffNetB2 model from

```
models/09_pretrained_effnetb2_feature_extractor_food101_20_percent.pth to demos/foodvision_big as well.
```

In [78]:

```
--2022-08-25 14:24:41-- https://raw.githubusercontent.com/mrdbourke/pytorch-deep-learning/main/images/04-pizza-dad.jpeg
Resolving raw.githubusercontent.com (raw.githubusercontent.com) ... 185.199.111.133, 185.199.110.1
33, 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: 2874848 (2.7M) [image/jpeg]
Saving to: '04-pizza-dad.jpeg'

04-pizza-dad.jpeg    100%[=====]    2.74M  7.85MB/s    in 0.3s

2022-08-25 14:24:43 (7.85 MB/s) - '04-pizza-dad.jpeg' saved [2874848/2874848]
```

## 11.2 Saving Food101 class names to file (`class_names.txt`)

Because there are so many classes in the Food101 dataset, instead of storing them as a list in our `app.py` file, let's save them to a `.txt` file and read them in when necessary instead.

We'll just remind ourselves what they look like first by checking out `food101_class_names`.

In [79]:

Out[79]:

```
['apple_pie',
 'baby_back_ribs',
 'baklava',
 'beef_carpaccio',
 'beef_tartare',
 'beet_salad',
 'beignets',
 'bibimbap',
 'bread_pudding',
 'breakfast_burrito']
```

Wonderful, now we can write these to a text file by first creating a path to `demos/foodvision_big/class_names.txt` and then opening a file with Python's `open()` and then writing to it leaving a new line for each class.

Ideally, we want our class names to be saved like:

```
apple_pie
baby_back_ribs
baklava
beef_carpaccio
beef_tartare
...
```

In [80]:



```
[INFO] Saving Food101 class names to demos/foodvision_big/class_names.txt
```

Excellent, now let's make sure we can read them in.

To do so we'll use Python's `open()` in read mode (`"r"`) and then use the `readlines()` method to read each line of our `class_names.txt` file.

And we can save the class names to a list by stripping the newline value of each of them with a list comprehension and `strip()`.

In [81]:



Out[81]:

```
['apple_pie', 'baby_back_ribs', 'baklava', 'beef_carpaccio', 'beef_tartare']
```

### 11.3 Turning our FoodVision Big model into a Python script (`model.py`)

Just like the FoodVision Mini demo, let's create a script that's capable of instantiating an EffNetB2 feature extractor model along with its necessary transforms.

In [82]:













```
Overwriting demos/foodvision_big/model.py
```

## 11.4 Turning our FoodVision Big Gradio app into a Python script (`app.py`)

We've got a FoodVision Big `model.py` script, now let's create a FoodVision Big `app.py` script.

This will again mostly be the same as the FoodVision Mini `app.py` script except we'll change:

- 1. Imports and class names setup** - The `class_names` variable will be a list for all of the Food101 classes rather than pizza, steak, sushi. We can access these via `demos/foodvision_big/class_names.txt`.
- 2. Model and transforms preparation** - The `model` will have `num_classes=101` rather than `num_classes=3`. We'll also be sure to load the weights from `"09_pretrained_effnetb2_feature_extractor_food101_20_percent.pth"` (our FoodVision Big model path).
- 3. Predict function** - This will stay the same as FoodVision Mini's `app.py`.

**4. Gradio app** - The Gradio interface will have different `title`, `description` and `article` parameters to reflect the details of FoodVision Big.

We'll also make sure to save it to `demos/foodvision_big/app.py` using the `%%writefile` magic command.

In [83]:





























```
Overwriting demos/foodvision_big/app.py
```

## 11.5 Creating a requirements file for FoodVision Big (`requirements.txt`)

Now all we need is a `requirements.txt` file to tell our Hugging Face Space what dependencies our FoodVision Big app requires.

In [84]:

Overwriting demos/foodvision\_big/requirements.txt

## 11.6 Downloading our FoodVision Big app files

We've got all the files we need to deploy our FoodVision Big app on Hugging Face, let's now zip them together and download them.

We'll use the same process we used for the FoodVision Mini app above in [section 9.1: \*Downloading our Foodvision Mini app files\*](#).

In [85]:



```

updating: 09_pretrained_effnetb2_feature_extractor_food101_20_percent.pth (deflated 8%)
updating: app.py (deflated 54%)
updating: class_names.txt (deflated 48%)
updating: examples/ (stored 0%)
updating: flagged/ (stored 0%)
updating: model.py (deflated 56%)
updating: requirements.txt (deflated 4%)
updating: examples/04-pizza-dad.jpg (deflated 0%)
Not running in Google Colab, can't use google.colab.files.download()

```

## 11.7 Deploying our FoodVision Big app to HuggingFace Spaces

B, E, A, Utiful!

Time to bring our biggest model of the whole course to life!

Let's deploy our FoodVision Big Gradio demo to Hugging Face Spaces so we can test it interactively and let others experience the magic of our machine learning efforts!

**Note:** There are [several ways to upload files to Hugging Face Spaces](#). The following steps treat Hugging Face as a git repository to track files. However, you can also upload directly to Hugging Face Spaces via the [web interface](#) or by the [huggingface\\_hub library](#).

The good news is, we've already done the steps to do so with FoodVision Mini, so now all we have to do is customize them to suit FoodVision Big:

1. [Sign up](#) for a Hugging Face account.
2. Start a new Hugging Face Space by going to your profile and then [clicking "New Space"](#).
  - **Note:** A Space in Hugging Face is also known as a "code repository" (a place to store your code/files) or "repo" for short.
3. Give the Space a name, for example, mine is called `mrdbourke/foodvision_big`, you can see it here:  
[https://huggingface.co/spaces/mrdbourke/foodvision\\_big](https://huggingface.co/spaces/mrdbourke/foodvision_big)

4. Select a license (I used [MIT](#)).
5. Select Gradio as the Space SDK (software development kit).
  - **Note:** You can use other options such as Streamlit but since our app is built with Gradio, we'll stick with that.
6. Choose whether your Space is it's public or private (I selected public since I'd like my Space to be available to others).
7. Click "Create Space".
8. Clone the repo locally by running: `git clone https://huggingface.co/spaces/[YOUR_USERNAME]/[YOUR_SPACE_NAME]` in terminal or command prompt.
  - **Note:** You can also add files via uploading them under the "Files and versions" tab.
9. Copy/move the contents of the downloaded `foodvision_big` folder to the cloned repo folder.
10. To upload and track larger files (e.g. files over 10MB or in our case, our PyTorch model file) you'll need to [install Git LFS](#) (which stands for "git large file storage").
11. After you've installed Git LFS, you can activate it by running `git lfs install`.
12. In the `foodvision_big` directory, track the files over 10MB with Git LFS with `git lfs track "*.file_extension"`.
  - Track EffNetB2 PyTorch model file with `git lfs track "09_pretrained_effnetb2_feature_extractor_food101_20_percent.pth"`
  - **Note:** If you get any errors uploading images, you may have to track them with `git lfs` too, for example `git lfs track "examples/04-pizza-dad.jpg"`
13. Track `.gitattributes` (automatically created when cloning from HuggingFace, this file will help ensure our larger files are tracked with Git LFS). You can see an example `.gitattributes` file on the [FoodVision Big Hugging Face Space](#).
  - `git add .gitattributes`
14. Add the rest of the `foodvision_big` app files and commit them with:
  - `git add *`
  - `git commit -m "first commit"`
15. Push (upload) the files to Hugging Face:
  - `git push`
16. Wait 3-5 minutes for the build to happen (future builds are faster) and your app to become live!

If everything worked correctly, our FoodVision Big Gradio demo should be ready to classify!

You can see my version here: [https://huggingface.co/spaces/mrdbourke/foodvision\\_big/](https://huggingface.co/spaces/mrdbourke/foodvision_big/)

Or we can even embed our FoodVision Big Gradio demo right within our notebook as an **iframe** with `IPython.display.IFrame` and a link to our space in the format

`https://hf.space/embed/[YOUR_USERNAME]/[YOUR_SPACE_NAME]/+.`

In [86]:

Out[86]:

How cool is that!?

We've come a long way from building PyTorch models to predict a straight line... now we're building computer vision models accessible to people all around the world!

## Main takeaways

- **Deployment is as important as training.** Once you've got a good working model, your first question should be: how can I deploy this and make it accessible to others? Deployment allows you to test your model in the real world rather than on private training and test sets.
- **Three questions for machine learning model deployment:**
  - a. What's the most ideal use case for the model (how well and how fast does it perform)?
  - b. Where's the model going to go (is it on-device or on the cloud)?
  - c. How's the model going to function (are predictions online or offline)?
- **Deployment options are a plenty.** But best to start simple. One of the best current ways (I say current because these things are always changing) is to use Gradio to create a demo and host it on Hugging Face Spaces. Start simple and scale up when needed.
- **Never stop experimenting.** Your machine learning model needs will likely change overtime so deploying a single model is not the last step. You might find the dataset changes, so you'll have to update your model. Or new research gets released and there's a better architecture to use.
  - So deploying one model is an excellent step, but you'll likely want to update it over time.
- **Machine learning model deployment is part of the engineering practice of MLOps (machine learning operations).** MLOps is an extension of DevOps (development operations) and involves all the engineering parts around training a model: data collection and storage, data preprocessing, model deployment, model monitoring, versioning and more. It's a rapidly evolving field but there are some solid resources out there to learn more, many of which are in [PyTorch Extra Resources](#).

## Exercises

All of the exercises are focused on practicing the code above.

You should be able to complete them by referencing each section or by following the resource(s) linked.

**Resources:**

- Exercise template notebook for 09.
  - Example solutions notebook for 09 try the exercises *before* looking at this.
    - See a live [video walkthrough of the solutions on YouTube](#) (errors and all).
1. Make and time predictions with both feature extractor models on the test dataset using the GPU (`device="cuda"`). Compare the model's prediction times on GPU vs CPU - does this close the gap between them? As in, does making predictions on the GPU make the ViT feature extractor prediction times closer to the EffNetB2 feature extractor prediction times?
    - You'll find code to do these steps in [section 5. Making predictions with our trained models and timing them](#) and [section 6. Comparing model results, prediction times and size](#).
  2. The ViT feature extractor seems to have more learning capacity (due to more parameters) than EffNetB2, how does it go on the larger 20% split of the entire Food101 dataset?
    - Train a ViT feature extractor on the 20% Food101 dataset for 5 epochs, just like we did with EffNetB2 in [section 10. Creating FoodVision Big](#).
  3. Make predictions across the 20% Food101 test dataset with the ViT feature extractor from exercise 2 and find the "most wrong" predictions.
    - The predictions will be the ones with the highest prediction probability but with the wrong predicted label.
    - Write a sentence or two about why you think the model got these predictions wrong.
  4. Evaluate the ViT feature extractor across the whole Food101 test dataset rather than just the 20% version, how does it perform?
    - Does it beat the original Food101 paper's best result of 56.4% accuracy?
  5. Head to [Paperswithcode.com](#) and find the current best performing model on the Food101 dataset.
    - What model architecture does it use?
  6. Write down 1-3 potential failure points of our deployed FoodVision models and what some potential solutions might be.
    - For example, what happens if someone was to upload a photo that wasn't of food to our FoodVision Mini model?
  7. Pick any dataset from `torchvision.datasets` and train a feature extractor model on it using a model from `torchvision.models` (you could use one of the model's we've already created, e.g. EffNetB2 or ViT) for 5 epochs and then deploy your model as a Gradio app to Hugging Face Spaces.
    - You may want to pick smaller dataset/make a smaller split of it so training doesn't take too long.
    - I'd love to see your deployed models! So be sure to share them in Discord or on the [course GitHub](#)

[Discussions page.](#)

## Extra-curriculum

- Machine learning model deployment is generally an engineering challenge rather than a pure machine learning challenge, see the [PyTorch Extra Resources machine learning engineering section](#) for resources on learning more.
  - Inside you'll find recommendations for resources such as Chip Huyen's book [\*Designing Machine Learning Systems\*](#) (especially chapter 7 on model deployment) and Goku Mohandas's [\*Made with ML MLOps\*](#) course.
- As you start to build more and more of your own projects, you'll likely start using Git (and potentially GitHub) quite frequently. To learn more about both, I'd recommend the [\*Git and GitHub for Beginners - Crash Course\*](#) video on the freeCodeCamp YouTube channel.
- We've only scratched the surface with what's possible with Gradio. For more, I'd recommend checking out the [full documentation](#), especially:
  - All of the different kinds of [input and output components](#).
  - The [Gradio Blocks API](#) for more advanced workflows.
  - The Hugging Face Course chapter on [how to use Gradio with Hugging Face](#).
- Edge devices aren't limited to mobile phones, they include small computers like the Raspberry Pi and the PyTorch team have a [fantastic blog post tutorial](#) on deploying a PyTorch model to one.
- For a fanstastic guide on developing AI and ML-powered applications, see [Google's People + AI Guidebook](#). One of my favourites is the section on [setting the right expectations](#).
  - I covered more of these kinds of resources, including guides from Apple, Microsoft and more in the [\*April 2021 edition of Machine Learning Monthly\*](#) (a monthly newsletter I send out with the latest and greatest of the ML field).
- If you'd like to speed up your model's runtime on CPU, you should be aware of [TorchScript](#), [ONNX](#) (Open Neural Network Exchange) and [OpenVINO](#). Going from pure PyTorch to ONNX/OpenVINO models I've seen a ~2x+ increase in performance.
- For turning models into a deployable and scalable API, see the [TorchServe library](#).
- For a terrific example and rationale as to why deploying a machine learning model in the browser (a form of edge deployment) offers several benefits (no network transfer latency delay), see Jo Kristian Bergum's article on [\*Moving ML Inference from the Cloud to the Edge\*](#).

Previous

## 08. PyTorch Paper Replicating

Next

PyTorch Extra Resources

Made with Material for MkDocs Insiders

# PyTorch Extra Resources

Despite the full Zero to Mastery PyTorch course being over 40 hours, you'll likely finish being excited to learn more.

After all, the course is a PyTorch momentum builder.

The following resources are collected to extend the course.

A warning though: there's a lot here.

Best to choose 1 or 2 resources from each section (or less) to explore more. And put the rest in your bag for later.

Which one's the best?

Well, if they've made it on this list, you can consider them a quality resource.

Most are PyTorch-specific, fitting extensions to the course but a couple are non PyTorch-specific, however, they're still valuable in the world of machine learning.

## Pure PyTorch resources

- **PyTorch blog** — Stay up to date on the latest from PyTorch right from the source. I check the blog once a month or so for updates.
- **PyTorch documentation** — We'll have explored this plenty throughout the course but there's still a large amount we haven't touched. No trouble, explore often and when necessary.
- **PyTorch Performance Tuning Guide** — One of the first things you'll likely want to do after the course is to make your PyTorch models faster (training and inference), the PyTorch Performance Tuning Guide helps you do just that.
- **PyTorch Recipes** — PyTorch recipes is a collection of small tutorials to showcase common PyTorch features and workflows you may want to create, such as Loading Data in PyTorch and Saving and Loading models for Inference in PyTorch.
- **PyTorch Ecosystem** - A vast collection of tools that build on top of pure PyTorch to add specialized features for

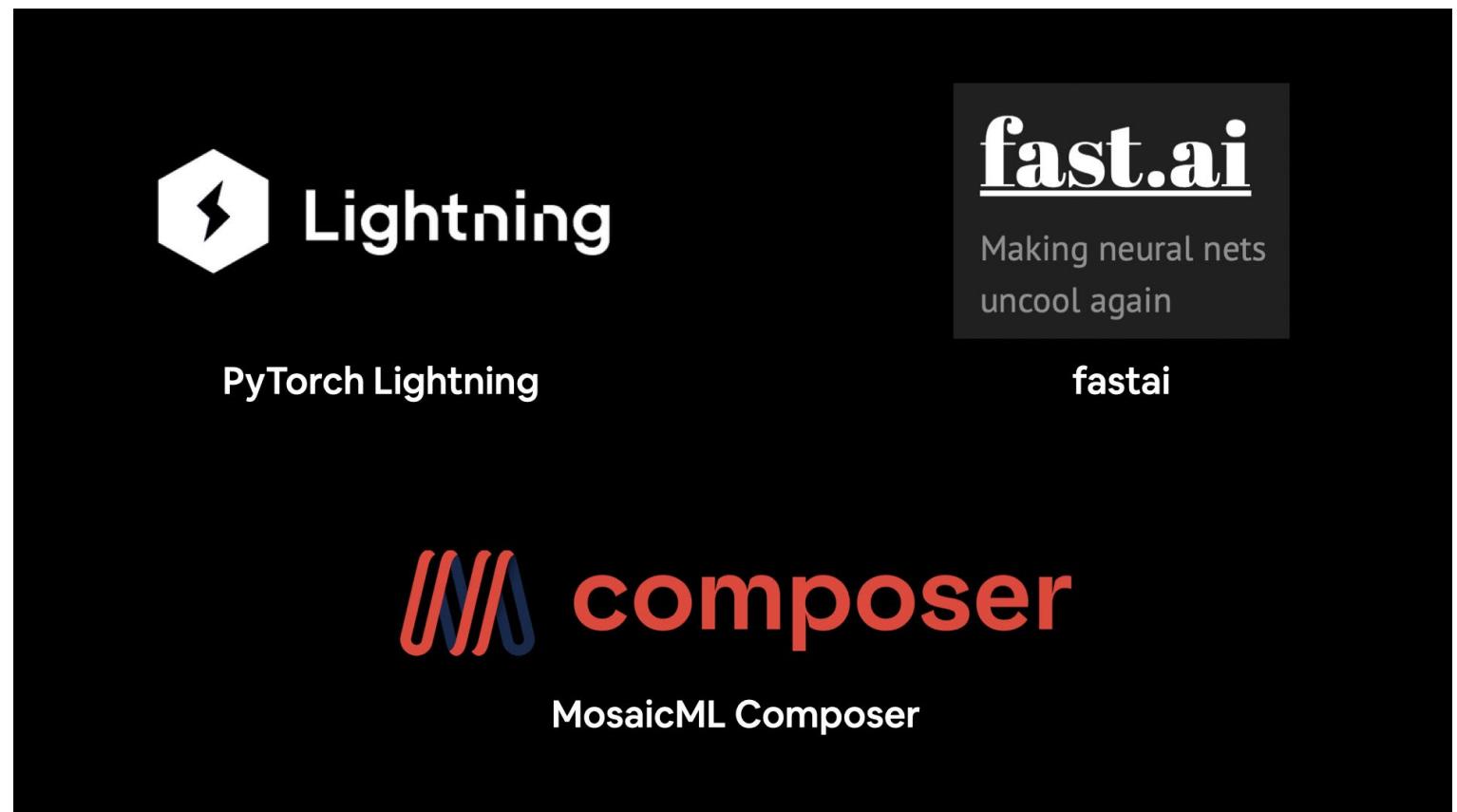
different fields, from PyTorch3D for 3D computer vision to Albumentations for fast data augmentation to TorchMetrics for model evaluation (thank you for the tip Alessandro).

- **Setting up PyTorch in VSCode** — VSCode is one of the most popular IDEs out there. And its PyTorch support is getting better and better. Throughout the Zero to Mastery PyTorch course, we use Google Colab because of its ease of use. But chances are you'll be developing in an IDE like VSCode soon.

## Libraries that make pure PyTorch better/add features

The course focuses on pure PyTorch (using minimal external libraries) because if you know how to write plain PyTorch, you can learn to use the various extension libraries.

- **fast.ai** — fastai is an open-source library that takes care of many of the boring parts of building neural networks and makes creating state-of-the-art models possible with a few lines of code. Their free library, course and documentation are all world-class.
- **MosaicML for more efficient model training** — The faster you can train models, the faster you can figure out what works and what doesn't. MosaicML's open-source `Composer` library helps you train neural networks with PyTorch faster by implementing speedup algorithms behind the scenes which means you can get better results out of your existing PyTorch models faster. All of their code is open-source and their docs are fantastic.
- **PyTorch Lightning for reducing boilerplate** — PyTorch Lightning takes care of many of the steps that you often have to do by hand in vanilla PyTorch, such as writing a training and test loop, model checkpointing, logging and more. PyTorch Lightning builds on top of PyTorch to allow you to make PyTorch models with less code.



*Libraries that extend/make pure PyTorch better.*

## Books for PyTorch

- **Machine Learning with PyTorch and Scikit-Learn: Develop machine learning and deep learning models with Python by Sebastian Raschka** — A fantastic introduction to machine learning and deep learning. Starting with traditional machine learning algorithms using Scikit-Learn for problems with structured data (tabular or rows and columns or Excel-style) and then switching to how to use PyTorch for deep learning on unstructured data (such as computer vision and natural language processing).
- **PyTorch Step-by-Step series by Daniel Voigt Godoy** — Where the Zero to Mastery PyTorch course works from a code-first perspective, the Step-by-Step series covers PyTorch and deep learning from a concept-first perspective with code examples to go along. With three editions, Fundamentals, Computer Vision and Sequences (NLP), the step-by-step series is one of my favourite resources for learning PyTorch from the ground up.
- **Dive into Deep Learning book** — Possibly one of the most comprehensive resources on the internet for deep learning concepts along with code examples in PyTorch, TensorFlow and Gluon. And all for free! For example, take a look at the author's explanation of the Vision Transformer we cover in 08. PyTorch Paper Replicating.
- **Bonus:** The fast.ai course (available free online) also comes as a freely available online book, Deep Learning for Coders with fastai & PyTorch.



*Textbooks to learn more about PyTorch as well as deep learning in general.*

## Resources for Machine Learning and Deep Learning Engineering

Machine Learning Engineering (also referred to as MLOps or ML operations) is the practice of getting the models you create into the hands of others. This may mean via a public app or working behind the scenes to make business decisions.

The following resources will help you learn more about the steps around deploying a machine learning model.

- **Designing Machine Learning Systems book by Chip Huyen** — If you want to build an ML system, it'd be good to know how others have done it. Chip's book focuses less on building a single machine learning model (though there's plenty of content on that in the book) but rather building a cohesive ML system. It covers everything from data engineering to model building to model deployment (online and offline) to model monitoring. Even better, it's a joy to read, you can tell the book is written by a writer (Chip has previously authored several books).
- **Made With ML by Goku Mohandas** — Whenever I want to learn or reference something to do with MLOps, I go to madewithml.com/mlops and see if there's a lesson on it. Made with ML not only teaches you the fundamentals of many different ML models but goes through how to build an end-to-end ML system with plenty of code and tooling examples.

- **The Machine Learning Engineering book by Andriy Burkov** — Even though this book is available to read online for free, I bought it as soon as it came out. I've used it as a reference and to learn more about ML engineering so much it's basically always on my desk/within arms reach. Burkov does an excellent job at getting to the point and referencing further materials when necessary.
- **Full Stack Deep Learning course** — I first did this course in 2021. And it's continued to evolve to cover the latest and greatest tools in the field. It'll teach you how to plan a project to solve an ML problem, how to source or create data, how to troubleshoot an ML project when it goes wrong and most of all, how to build ML-powered products.



*Resources to improve your machine learning engineering skills (all of the steps that go around building a machine learning model).*

## Where to find datasets

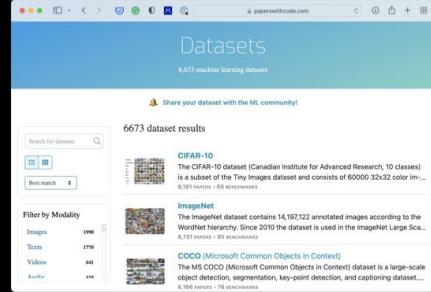
Machine learning projects begin with data.

No data, no ML.

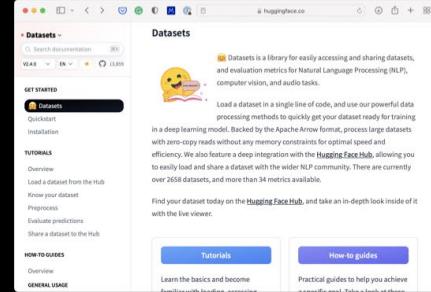
The following resources are some of the best for finding open-source and often ready-to-use datasets on a wide range of topics and problem domains.

- **Paperswithcode Datasets** — Search for the most used and common machine learning benchmark datasets, understand what they contain, where they came from and where they can be found. You can often also see the current best-performing model on each dataset.
- **HuggingFace Datasets** — Not just a resource to find datasets across a wide range of problem domains but also a library to download and start using them within a few lines of code.
- **Kaggle Datasets** — Find all kinds of datasets that usually accompany Kaggle Competitions, many of which come straight out of industry.
- **Google Dataset search** — Just like searching Google but specifically for datasets.

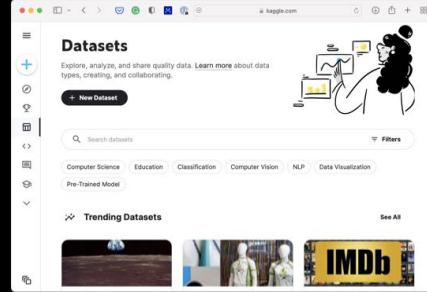
These should be plenty to get started, however, for your own specific problems you'll likely want to build your own dataset.



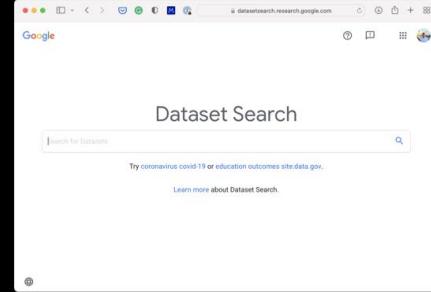
**Paperswithcode Datasets**



**Datasets**



**Kaggle Datasets**



**Google Dataset Search**

*Places to find existing and open-source datasets for a variety of problem spaces.*

## Tools for Deep Learning Domains

The following resources are focused on libraries and pretrained models for specific problem domains such as computer vision and recommendation engines/systems.

## Computer Vision

We cover computer vision in 03. PyTorch Computer Vision but as a quick recap, computer vision is the art of getting computers to see.

If your data is visual, images, x-rays, production line video or even hand-written documents, it may be a computer vision problem.

- **TorchVision** — PyTorch's resident computer vision library. Find plenty of methods for loading vision data as well as plenty of pretrained computer vision models to use for your own problems.
- **timm (Torch Image Models) library** — One of the most comprehensive computer vision libraries and resources for pretrained computer vision models. Almost all new research in that uses PyTorch for computer vision leverages the `timm` library in some way.
- **Yolov5 for object detection** — If you're looking to build an object detection model in PyTorch, the `yolov5` GitHub repository might be the quickest way to get started.
- **VISSL (Vision Self-Supervised Learning) library** — Self-supervised learning is the art of getting data to learn patterns in itself. Rather than providing labels for different classes and learning a representation like that, self-supervised learning tries to replicate similar results without labels. VISSL provides an easy to use way to get started using self-supervised learning computer vision models with PyTorch.

## Natural Language Processing (NLP)

Natural language processing involves finding patterns in text.

For example, you might want to extract important entities in support tickets or classify a document into different categories.

If your problem involves a large of amount of text, you'll want to look into the following resources.

- **TorchText** — PyTorch's in-built domain library for text. Like TorchVision, it contains plenty of pre-built methods for loading data and a healthy collection of pretrained models you can adapt to your own problems.
- **HuggingFace Transformers library** — The HuggingFace Transformers library has more stars on GitHub than the PyTorch library itself. And there's a reason. Not that HuggingFace Transformers is better than PyTorch but because it's the best at what it does: provide data loaders and pretrained state-of-the-art models for NLP (and a whole bunch more).
- **Bonus:** To learn more about how to HuggingFace Transformers library and all of the pieces around it, the HuggingFace team offer a free online course.

## Speech and Audio

If your problem deals with audio files or speech data, such as trying to classify a sound or transcribe speech into text, you'll want to look into the following resources.

- **TorchAudio** — PyTorch's domain library for everything audio. Find in-built methods for preparing data and pre-built model architectures for finding patterns in audio data.
- **SpeechBrain** — An open-source library built on top of PyTorch to handle speech problems such as recognition (turning speech into text), speech enhancement, speech processing, text-to-speech and more. You can try out many of their models on the HuggingFace Hub.

## Recommendation Engines

The internet is powered by recommendations. YouTube recommends videos, Netflix recommends movies and TV shows, Amazon recommends products, Medium recommends articles.

If you're building an online store or online marketplace, chances are you'll want to start recommending things to your customers.

For that, you'll want to look into building a recommendation engine.

- **TorchRec** — PyTorch's newest in-built domain library for powering recommendation engines with deep learning. TorchRec comes with recommendation datasets and models ready to try and use. Though if a custom recommendation engine isn't up to par with what you're after (or too much work), many cloud vendors offer recommendation engine services.

## Time Series

If your data has a time component and you'd like to leverage patterns from the past to predict the future, such as, predicting the price of Bitcoin next year (don't try this, stock forecasting is BS) or a more reasonable problem of predicting electricity demand for a city next week, you'll want to look into time series libraries.

Both of these libraries don't necessarily use PyTorch, however, since time series is such a common problem, I've included them here.

- **Salesforce Merlion** — Turn your time series data into intelligence by using Merlion's data loaders, pre-built models, AutoML (automated machine learning) hyperparameter tuning and more for time series forecasting and time series anomaly detection all inspired by practical use cases.
- **Facebook Kats** — Facebook's entire business depends on prediction: when's the best time to place an advertisement? So you can bet they're invested heavily in their time series prediction software. Kats (Kit to Analyze Time Series data) is their open-source library for time series forecasting, detection and data processing.

# How to get a job

Once you've finished an ML course, it's likely you'll want to use your ML skills.

And even better, get paid for them.

The following resources are good guides on what to do to get one.

- **"How can a beginner data scientist like me gain experience?" by Daniel Bourke** — I get the question of "how do I get experience?" often because many different job requirements state "experience needed". Well, it turns out one of the best ways to get experience (and a job) is to: *start the job before you have it*.
- **You Don't Really Need Another MOOC by Eugene Yan** — MOOC stands for massive online open course (or something similar). MOOCs are beautiful. They enable people all over the world at their own pace. However, it can be tempting to just continually do MOOCs over and over again thinking "if I just do one more, I'll be ready". The truth is, a few is enough, the returns of a MOOC quickly start to trail off. Instead, go off the trail, start to build, start to create, start to learn skills that can't be taught. Showcase those skills to get a job.
- **Bonus:** For the most thorough resource on the internet for machine learning interviews, check out Chip Huyen's free Introduction to Machine Learning Interviews book.

Previous

[09. PyTorch Model Deployment](#)

Made with Material for MkDocs Insiders