Open in Colab

View source code | Read notebook in online book format

Introduction to TensorFlow, Deep Learning and Transfer Learning (work in progress)

- **Project:** Dog Vision 🐶👁️ - Using computer vision to classify dog photos into different breeds.
- **Goals:** Learn TensorFlow, deep learning and transfer learning, beat the original research paper results (22% accuracy).
- **Domain:** Computer vision.
- **Data:** Images of dogs from Stanford Dogs Dataset (120 dog breeds, 20,000+ images).
- **Problem type:** Multi-class classification (120 different classes).
- **Runtime:** This project is designed to run end-to-end in Google Colab (for free GPU access and easy setup). If you'd like to run it locally, it will require environment setup.
- **Demo:** See a demo of the trained model running on Hugging Face Spaces.

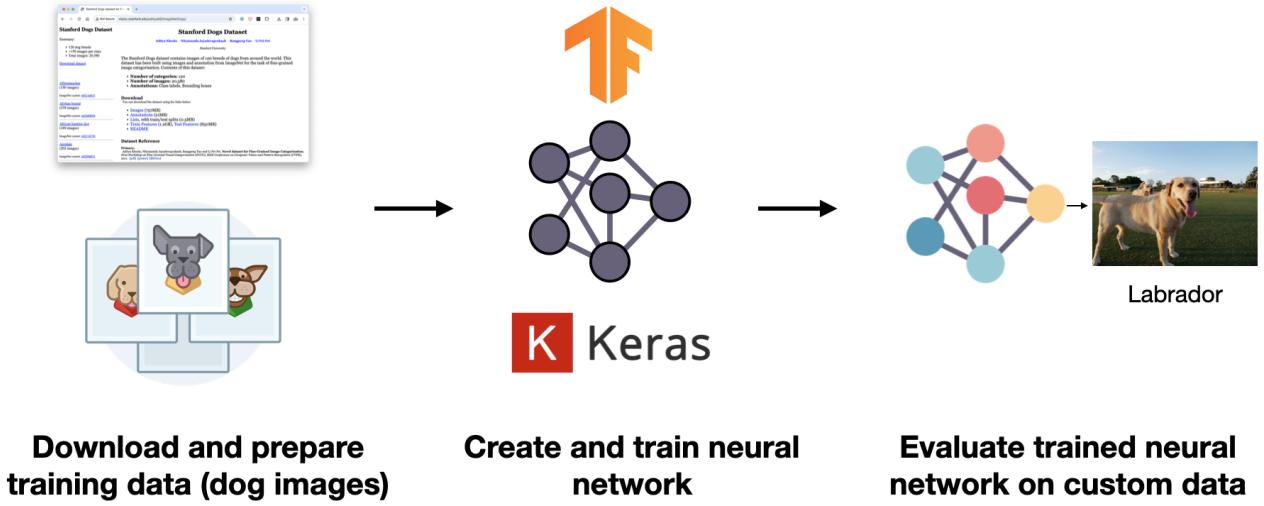
Welcome, welcome!

The focus of this notebook is to give a quick overview of deep learning with TensorFlow/Keras.

Table of contents

How?

We're going to go through the machine learning workflow steps and build a computer vision project to classify photos of dogs into their respective dog breed (a Predictive AI task, see below for more).



What we're going to build: Dog Vision 🐶👁️, a neural network capable of identifying different dog breeds in images. All the way from dataset preparation to model building, training and evaluation.

In [1]:
```
# Quick timestamp
import datetime
print(f"Last updated: {datetime.datetime.now()}")
```

```
Last updated: 2024-04-26 01:26:48.838163
```

Visualize a group of random images

Exploring the distribution of our

What we're going to cover

In this project, we're going to be introduced to the power of deep learning and more specifically, transfer learning using TensorFlow/Keras.

We'll go through each of these in the context of the 6 step machine learning framework:

1. **Problem defintion** - Use computer vision to classify photos of dogs into different dog breeds.
2. **Data** - 20,000+ images of dogs from 120 different dog breeds from the Stanford Dogs dataset.
3. **Evaluation** - We'd like to beat the original paper's results (22% mean accuracy across all classes, **tip:** A good way to practice your skills is to find some results online and try to beat them).
4. **Features** - Because we're using deep learning, our model will learn the features on its own.
5. **Modelling** - We're going to use a pretrained convolutional neural network (CNN) and transfer learning.
6. **Experiments** - We'll try different amounts of data with the same model to see the effects on our results.

> **Note:** It's okay not to know these exact steps ahead of time. When starting a new project, it's often the case you'll figure it out as you go. These steps are only filled out because I've had practice working on several machine learning projects. You'll pick up these ideas overtime.

Table of contents

1. Getting Setup
2. Getting Data (dog images and their breeds)
3. Exploring the data (exploratory data analysis)
4. Creating training and test splits
5. Turning our datasets into TensorFlow Dataset(s)
6. Creating a neural network with TensorFlow
7. Model 0 - Train a model on 10% of the training data
8. Putting it all together: create, compile, fit
9. Model 1 - Train a model on 100% of the training data
10. Make and evaluate predictions of the best model
11. Save and load the best model
12. Make predictions on custom images with the best model (bringing Dog Vision 🐶👁️ to life!)
13. Key takeaways
14. Extensions & exercises

Where can 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 there too.

You can also:

- Search for questions online and end up at places such as Stack Overflow (a great resource of developer-focused Q&A).
- Ask AI Assistants such as ChatGPT, Gemini and Claude for help with various coding problems and errors.

Quick definitions

Let's start by breaking down some of the most important topics we're going to go through.

What is TensorFlow/Keras?

TensorFlow is an open source machine learning and deep learning framework originally developed by Google. Inside TensorFlow, you can also use Keras which is another very helpful machine learning framework known for its ease of use.

Why use TensorFlow?

TensorFlow allows you to manipulate data and write deep learning algorithms using Python code.

It also has several built-in capabilities to leverage accelerated computing hardware (e.g. GPUs, Graphics Processing Units and TPUs, Tensor Processing Units).

Many of world's largest companies power their machine learning workloads with TensorFlow.

What is deep learning?

Deep learning is a form of machine learning where data passes through a series of progressive layers which all contribute to learning an overall representation of that data.

Each layer performs a pre-defined operation.

The series of progressive layers combine to form what's referred to as a **neural network**.

For example, a photo may be turned into numbers (e.g. red, green and blue pixel values) and those numbers are then manipulated mathematically through each progressive layer to learn patterns in the photo.

The "deep" in deep learning comes from the number of layers used in the neural network.

So when someone says deep learning or (artificial neural networks), they're typically referring to same thing.

> **Note:** Artificial intelligence (AI), machine learning (ML) and deep learning are all broad terms. You can think of AI as the overall technology, machine learning as a type of AI, and deep learning as a type of machine learning. So if someone refers to AI, you can often assume they are often talking about machine learning or deep learning.

What can deep learning be used for?

Deep learning is such a powerful technique that new use cases are being discovered everyday.

Most of the modern forms of artifical intelligence (AI) applications you see, are powered by deep learning.

Two of the most useful types of AI are predictive and generative.

Predictive AI learns the relationship between data and labels such as photos of dog and their breeds (supervised learning). So that when it sees a new photo of a dog, it can predict its breed based on what its learned.

Generative AI generates something new given an input such as creating new text given input text.

Some examples of Predictive AI problems include:

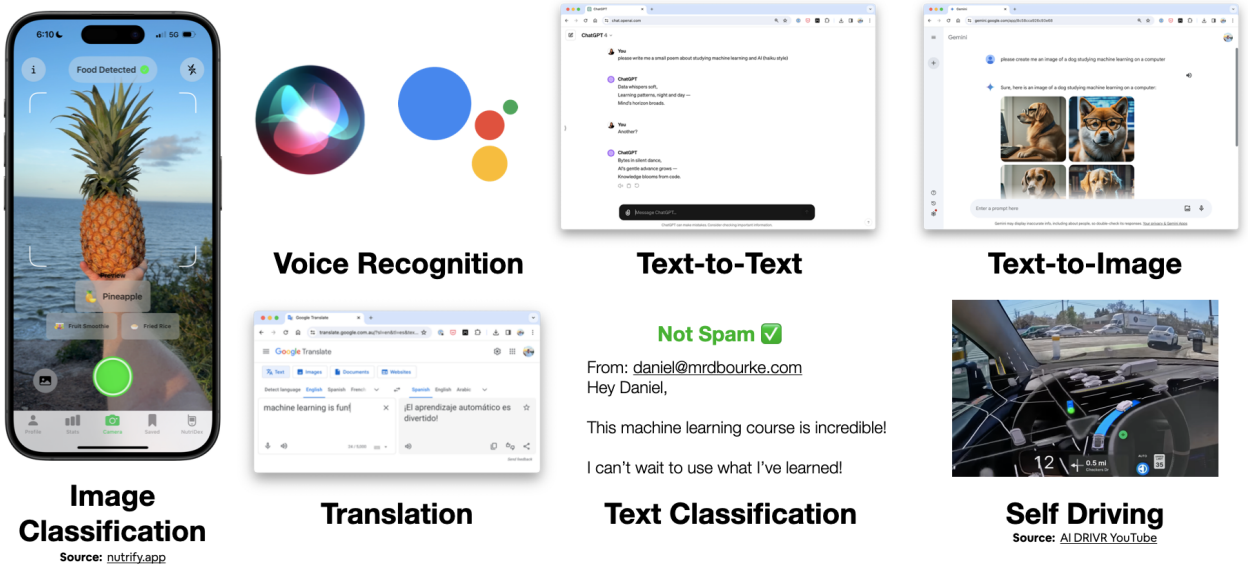
- Tesla's self-driving cars use deep learning use object detection models to power their computer vision systems.
- Apple's Photos app uses deep learning to recognize faces in images and create Photo Memories.
- Siri and Google Assistant use deep learning to transcribe speech and understand voice commands.
- Nutrify (an app my brother and I build) uses predictive AI to recognize food in images.
- Magika uses deep learning to classify a file into what type it is (e.g. `.jpeg`, `.py`, `.txt`).
- Text classification models such as DeBERTa use deep learning to classify text into different categories such as "positive" and "negative" or "spam" or "not spam".

Some examples of Generative AI problems include:

- Stable Diffusion uses generative AI to generate images given a text prompt.
- ChatGPT and other large language models (LLMs) such as Llama, Claude, Gemini and Mistral use deep learning to process text and return a response.
- GitHub Copilot uses generative AI to generate code snippets given surrounding context.

All of these AI use cases are powered by deep learning.

And more often than not, whenever you get started on a deep learning problem, you'll start with transfer learning.



Example of different every day problems where AI/machine learning gets used.

What is transfer learning?

Transfer learning is one of the most powerful and useful techniques in modern AI and machine learning.

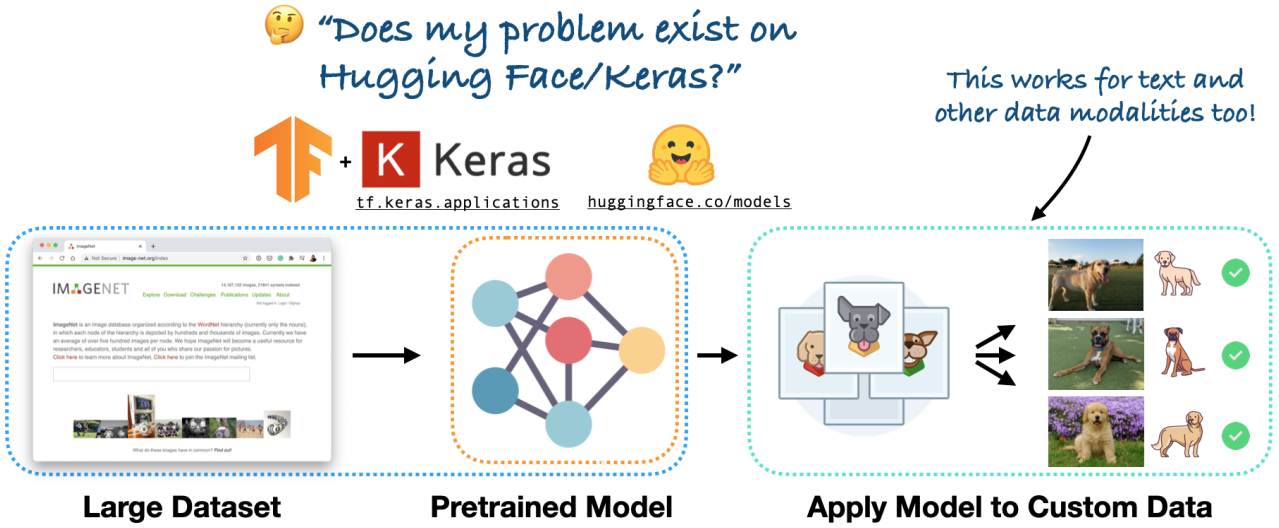
It involves taking what one model (or neural network) has learned in a similar domain and applying to your own.

In our case, we're going to use transfer learning to take the patterns a neural network has learned from the 1 million+ images and over 1000 classes in ImageNet (a gold standard computer vision benchmark) and apply them to our own problem of recognizing dog breeds.

However, this concept can be applied to many different domains.

You could take a large language model (LLM) that has been pre-trained on *most* of the text on the internet and learned very well the patterns in naturual language and customize it for your own specific chat use-case.

The biggest benefit of transfer learning is that it often allows you to get outstanding results with less data and time.



A transfer learning workflow. Many publicly available models have been pretrained on large datasets such as ImageNet (1 million+ images). These models can then be applied to similar tasks downstream. For example, we can take a model pretrained on ImageNet and apply it to our Dog Vision 🐶👁️ problem. This same process can be repeated for many different styles of data and problem.

1. Getting setup

This notebook is designed to run in Google Colab, an online Jupyter Notebook that provides free access to GPUs (Graphics Processing Units, we'll hear more on these later).

For a quick rundown on how to use Google Colab, see their introductory guide (it's quite similar to a Jupyter Notebook with a few different options).

Google Colab also comes with many data science and machine learning libraries pre-installed, including TensorFlow/Keras.

Getting a GPU on Google Colab

Before running any code, we'll make sure our Google Colab instance is connected to a GPU.

You can do this via going to Runtime -> Change runtime type -> GPU (this may restart your existing runtime).

Why use a GPU?

Since neural networks perform a large amount of calculations behind the scenes (the main one being matrix multiplication), you need a computer chip that perform these calculations quickly, otherwise you'll be waiting all day for a model to train.

And in short, GPUs are much faster at performing matrix multiplications than CPUs.

Why this is the case is behind the scope of this project (you can search "why are GPUs faster than CPUs for machine learning?" for more).

The main thing to remember is: generally, in deep learning, GPUs = faster than CPUs.

> **Note:** A good experiment would be to run the neural networks we're going to build later on with and without a GPU and see the difference in their training times.

Ok, enough talking, let's start by importing TensorFlow!

We'll do so using the common abbreviation `tf`.

In [2]:
```
import tensorflow as tf
tf.__version__
```

Out[2]:
```
'2.15.0'
```

Nice!

> **Note:** If you want to run TensorFlow locally, you can follow the TensorFlow installation guide.

Now let's check to see if TensorFlow has access to a GPU (this isn't 100% required to complete this project but will speed things up dramatically).

We can do so with the method `tf.config.list_physical_devices()`.

In [3]:
```
# Do we have access to a GPU?
device_list = tf.config.list_physical_devices()
if "GPU" in [device.device_type for device in device_list]:
  print(f"[INFO] TensorFlow has GPU available to use. Woohoo!! Computing will be
  print(f"[INFO] Accessible devices:\n{device_list}")
else:
  print(f"[INFO] TensorFlow does not have GPU available to use. Models may take a
  print(f"[INFO] Accessible devices:\n{device_list}")
```

```
[INFO] TensorFlow has GPU available to use. Woohoo!! Computing will be sped up!
[INFO] Accessible devices:
[PhysicalDevice(name='/physical_device:CPU:0', device_type='CPU'), PhysicalDevice(n
ame='/physical_device:GPU:0', device_type='GPU')]
```

2. Getting Data

All machine learning (and deep learning) projects start with data.

If you have no data, you have no project.

If you have no project, you have no cool models to show your friends or improve your business.

Not to worry!

There are several options and locations to get data for a deep learning project.

Resource	Description
Kaggle Datasets	A collection of datasets across a wide range of topics.
TensorFlow Datasets	A collection of ready-to-use machine learning datasets ready for use under the `tf.data.Datasets` API. You can see a list of all available datasets in the TensorFlow documentation.
Hugging Face Datasets	A continually growing resource of datasets broken into several different kinds of topics.
Google Dataset Search	A search engine by Google specifically focused on searching online datasets.
Original sources	Datasets which are made available by researchers or companies with the release of a product or research paper (sources for these will vary, they could be a link on a website or a link to an application form).
Custom datasets	These are datasets comprised of your own custom source of data. You may build these from scratch on your own or have access to them from an existing product or service. For example, your entire photos library could be your own custom dataset or your entire notes and documents folder or your company's customer order history.

In our case, the dataset we're going to use is called the Stanford Dogs dataset (or ImageNet dogs, as the images are dogs separated from ImageNet).

Because the Stanford Dogs dataset has been around for a while (since 2011, which as of writing this in 2024 is like a lifetime in deep learning), it's available from several resources:

- The original project website via link download.

- Inside TensorFlow datasets under `stanford_dogs`.
- On Kaggle as a downloadable dataset.

The point here is that when you're starting out with practicing deep learning projects, there's no shortage of datasets available.

However, when you start wanting to work on your own projects or within a company environment, you'll likely start to work on custom datasets (datasets you build yourself or aren't available publicly online).

The main difference between existing datasets and custom datasets is that existing datasets often come preformatted and ready to use.

Where as custom datasets often require some preprocessing before they're ready to use within a machine learning project.

To practice formatting a dataset for a machine learning problem, we're going to download the Stanford Dogs dataset from the original website.

Before we do so, the following code is an example of how we'd get the Stanford Dogs dataset from TensorFlow Datasets.

In [4]:
```
# Download the dataset into train and test split using TensorFlow Datasets
# import tensorflow_datasets as tfds
# ds_train, ds_test = tfds.load('stanford_dogs', split=['train', 'test'])
```

Download data directly from Stanford Dogs website

Our overall project goal is to build a computer vision model which performs better than the original Stanford Dogs paper (average of 22% accuracy per class across 120 classes).

To do so, we need some data.

Let's download the original Stanford Dogs dataset from the project website.

The data comes in three main files:

1. Images (757MB) - `images.tar`
2. Annotations (21MB) - `annotation.tar`
3. Lists with train/test splits (0.5MB) - `lists.tar`

Our goal is to get a file structure like this:

```
dog_vision_data/
    images.tar
    annotation.tar
    lists.tar
```

> **Note:** If you're using Google Colab for this project, remember that any data uploaded to the Google Colab session gets deleted if the session disconnects. So to save us redownloading the data every time, we're going to download it once and save it to Google Drive.
>
> **Resource:** For a good guide on getting data in and out of Google Colab, see the Google Colab `io.ipynb` tutorial.

To make sure we don't have to keep redownloading the data every time we leave and come back to Google Colab, we're going to:

1. Download the data if it doesn't already exist on Google Drive.
2. Copy it to Google Drive (because Google Colab connects nicely with Google Drive) if it isn't already there.
3. If the data already exists on Google Drive (we've been through steps 1 & 2), we'll import it instead.

There are two main options to connect Google Colab instances to Google Drive:

1. Click "Mount Drive" in "Files" menu on the left.
2. Mount programmatically with `from google.colab import drive` -> `drive.mount('/content/drive')`.

More specifically, we're going to follow the following steps:

1. Mount Google Drive.
2. Setup constants such as our base directory to save files to, the target files we'd like to download and target URL we'd like to download from.
3. Setup our target local path to save to.
4. Check if the target files all exist in Google Drive and if they do, copy them locally.
5. If the target files don't exist in Google Drive, download them from the target URL with the `!wget` command.
6. Create a file on Google Drive to store the download files.
7. Copy the downloaded files to Google Drive for use later if needed.

A fair few steps, but nothing we can't handle!

Plus, this is all good practice for dealing with and manipulating data, a very important skill in the machine learning engineers toolbox.

> **Note:** The following data download section is designed to run in Google Colab. If you are running locally, feel free to modify the code to save to a local directory instead of Google Drive.

In [5]:

```
from pathlib import Path
from google.colab import drive

# 1. Mount Google Drive (this will bring up a pop-up to sign-in/authenticate)
# Note: This step is specifically for Google Colab, if you're working locally, you
drive.mount("/content/drive")

# 2. Setup constants
# Note: For constants like this, you'll often see them created as variables with al
TARGET_DRIVE_PATH = Path("drive/MyDrive/tensorflow/dog_vision_data")
TARGET_FILES = ["images.tar", "annotation.tar", "lists.tar"]
TARGET_URL = "http://vision.stanford.edu/aditya86/ImageNetDogs"

# 3. Setup local path
local_dir = Path("dog_vision_data")

# 4. Check if the target files exist in Google Drive, if so, copy them to Google Co
if all((TARGET_DRIVE_PATH / file).is_file() for file in TARGET_FILES):
  print(f"[INFO] Copying Dog Vision files from Google Drive to local directory...
  print(f"[INFO] Source dir: {TARGET_DRIVE_PATH} -> Target dir: {local_dir}")
  !cp -r {TARGET_DRIVE_PATH} .
  print("[INFO] Good to go!")

else:
  # 5. If the files don't exist in Google Drive, download them
  print(f"[INFO] Target files not found in Google Drive.")
  print(f"[INFO] Downloading the target files... this shouldn't take too long..."
  for file in TARGET_FILES:
```

```
    # wget is short for "world wide web get", as in "get a file from the web"
    # -nc or --no-clobber = don't download files that already exist locally
    # -P = save the target file to a specified prefix, in our case, local_dir
    !wget -nc {TARGET_URL}/{file} -P {local_dir} # the "!" means to execute the co

print(f"[INFO] Saving the target files to Google Drive, so they can be loaded l

# 6. Ensure target directory in Google Drive exists
TARGET_DRIVE_PATH.mkdir(parents=True, exist_ok=True)

# 7. Copy downloaded files to Google Drive (so we can use them later and not have
!cp -r {local_dir}/* {TARGET_DRIVE_PATH}/
```

```
Mounted at /content/drive
[INFO] Copying Dog Vision files from Google Drive to local directory...
[INFO] Source dir: drive/MyDrive/tensorflow/dog_vision_data -> Target dir: dog_visi
on_data
[INFO] Good to go!
```

Data downloaded!

Nice work! This may seem like a bit of work but it's an important step with any deep learning project. Getting data to work with.

Now if we get the contents of `local_dir` (`dog_vision_data`), what do we get?

We can first make sure it exists with `Path.exists()` and then we can iterate through its contents with `Path.iterdir()` and print out the `.name` attribute of each file.

In [6]:
```
if local_dir.exists():
  print(str(local_dir) + "/")
  for item in local_dir.iterdir():
    print("   ", item.name)
```

```
dog_vision_data/
   lists.tar
```

```
images.tar
annotation.tar
```

Excellent! That's exactly the format we wanted.

Now you might've noticed that each file ends in `.tar`.

What's this?

Searching "what is .tar?", I found:

> *In computing, tar is a computer software utility for collecting many files into one archive file, often referred to as a tarball, for distribution or backup purposes.*
>
> Source: Wikipedia tar page).

Exploring a bit more, I found that the `.tar` format is similar to `.zip`, however, `.zip` offers compression, where as `.tar` mostly combines many files into one.

So how do we "untar" the files in `images.tar`, `annotation.tar` and `lists.tar`?

We can use the `!tar` command (or just `tar` from outside of a Jupyter Cell)!

Doing this will expand all of the files within each of the `.tar` archives.

We'll also use a couple of flags to help us out:

- The `-x` flag tells `tar` to extract files from an archive.
- The `-f` flag specifies that the following argument is the name of the archive file.
- You can combine flags by putting them together `-xf`.

Let's try it out!

In [7]:
```
# Untar images, notes/tags:
# -x = extract files from the zipped file
# -v = verbose
# -z = decompress files
# -f = tell tar which file to deal with
!tar -xf dog_vision_data/images.tar
!tar -xf dog_vision_data/annotation.tar
!tar -xf dog_vision_data/lists.tar
```

What new files did we get?

We can check in Google Colab by inspecting the "Files" tab on the left.

Or with Python by using `os.listdir(".")` where `"."` means "the current directory".

In [8]:
```
import os

os.listdir(".") # "." stands for "here" or "current directory"
```

Out[8]:
```
['.config',
 'dog_vision_data',
 'file_list.mat',
 'drive',
 'train_list.mat',
 'Images',
 'Annotation',
 'test_list.mat',
 'sample_data']
```

Ooooh!

Looks like we've got some new files!

Specifically:

- `train_list.mat` - a list of all the training set images.
- `test_list.mat` - a list of all the testing set images.
- `Images/` - a folder containing all of the images of dogs.
- `Annotation/` - a folder containing all of the annotations for each image.
- `file_list.mat` - a list of all the files (training and test list combined).

Our next step is to go through them and see what we've got.

3. Exploring the data

Once you've got a dataset, before building a model, it's wise to explore it for a bit to see what kind of data you're working with.

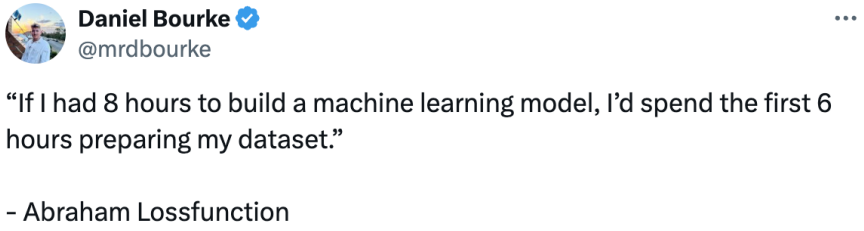
Exploring a dataset can mean many things.

But a few rules of thumb when exploring new data:

- **View at least 100+ random samples for a "vibe check".** For example, if you have a large dataset of images, randomly sample 10 images at a time and view them. Or if you have a large dataset of texts, what do some of them say? The same with audio. It will often be impossible to view all samples in your dataset, but you can start to get a good idea of what's inside by randomly inspecting samples.
- ***Visualize, viuslaize, visualize!*** This is the data explorer's motto. Use it often. As in, it's good to get statistics about your dataset but it's often even better to view 100s of samples with your own eyes (see the point above).

- **Check the distributions and other various statistics.** How many samples are there? If you're dealing with classification, how many classes and labels per class are there? Which classes don't you understand? If you don't have labels, investigate clustering methods to put similar samples close together.

As Abraham Lossfunction says...



A play on words of Abraham Lincoln's famous quote on sharpening an axe before cutting down a tree in theme of machine learning. Source: Daniel Bourke X/Twitter.

Our target data format

Since our goal is to build a computer vision model to classify dog breeds, we need a way to tell our model what breed of dog is in what image.

A common data format for a classification problem is to have samples stored in folders named after their class name.

For example:

```
images_split/
├── train/
│   ├── class_1/
│   │   ├── train_image1.jpg
│   │   ├── train_image2.jpg
│   │   └── ...
│   ├── class_2/
│   │   ├── train_image1.jpg
│   │   ├── train_image2.jpg
│   │   └── ...
└── test/
    ├── class_1/
    │   ├── test_image1.jpg
    │   ├── test_image2.jpg
    │   └── ...
    ├── class_2/
    │   ├── test_image1.jpg
    │   ├── test_image2.jpg
...
```

In the case of dog images, we'd put all of the images labelled "chihuahua" in a folder called `chihuahua/` (and so on for all the other classes and images).

We could split these folders so that training images go in `train/chihuahua/` and testing images go in `test/chihuahua/`.

This is what we'll be working towards creating.

> **Note:** This structure of folder format doesn't just work for only images, it can work for text, audio and other kind of classification data too.

Exploring the file lists

How about we check out the `train_list.mat`, `test_list.mat` and `full_list.mat` files?

Searching online, for "what is a .mat file?", I found that it's a MATLAB file. Before Python became the default language for machine learning and deep learning, many models and datasets were built in MATLAB.

Then I searched, "how to open a .mat file with Python?" and found an answer on Stack Overflow saying I could use the `scipy` library (a scientific computing library).

The good news is, Google Colab comes with `scipy` preinstalled.

We can use the `scipy.io.loadmat()` method to open a `.mat` file.

In [9]:
```
import scipy

# Open lists of train and test .mat
train_list = scipy.io.loadmat("train_list.mat")
test_list = scipy.io.loadmat("test_list.mat")
file_list = scipy.io.loadmat("file_list.mat")

# Let's inspect the output and type of the train_list
train_list, type(train_list)
```

Out[9]:
```
({'__header__': b'MATLAB 5.0 MAT-file, Platform: GLNXA64, Created on: Sun Oct  9
08:36:13 2011',
  '__version__': '1.0',
  '__globals__': [],
  'file_list': array([[array(['n02085620-Chihuahua/n02085620_5927.jpg'], dtype='<
U38')],
         [array(['n02085620-Chihuahua/n02085620_4441.jpg'], dtype='<U38')],
         [array(['n02085620-Chihuahua/n02085620_1502.jpg'], dtype='<U38')],
         ...,
         [array(['n02116738-African_hunting_dog/n02116738_6754.jpg'], dtype='<U4
8')],
         [array(['n02116738-African_hunting_dog/n02116738_9333.jpg'], dtype='<U4
8')],
         [array(['n02116738-African_hunting_dog/n02116738_2503.jpg'], dtype='<U4
8')]],
        dtype=object),
  'annotation_list': array([[array(['n02085620-Chihuahua/n02085620_5927'], dtype
='<U34')],
         [array(['n02085620-Chihuahua/n02085620_4441'], dtype='<U34')],
         [array(['n02085620-Chihuahua/n02085620_1502'], dtype='<U34')],
         ...,
         [array(['n02116738-African_hunting_dog/n02116738_6754'], dtype='<U44')],
         [array(['n02116738-African_hunting_dog/n02116738_9333'], dtype='<U44')],
         [array(['n02116738-African_hunting_dog/n02116738_2503'], dtype='<U4
4')]],
        dtype=object),
  'labels': array([[  1],
         [  1],
         [  1],
         ...,
         [120],
         [120],
         [120]], dtype=uint8)},
 dict)
```

Okay, looks like we get a dictionary with several fields we may be interested in.

Let's check out the keys of the dictionary.

In [10]:
```
train_list.keys()
```

Out[10]:
```
dict_keys(['__header__', '__version__', '__globals__', 'file_list', 'annotation_list', 'labels'])
```

My guess is that the `file_list` key is what we're after, as this looks like a large array of image names (the files all end in `.jpg`).

How about we see how many files are in each `file_list` key?

In [11]:
```
# Check the length of the file_list key
print(f"Number of files in training list: {len(train_list['file_list'])}")
print(f"Number of files in testing list: {len(test_list['file_list'])}")
print(f"Number of files in full list: {len(file_list['file_list'])}")
```

```
Number of files in training list: 12000
Number of files in testing list: 8580
Number of files in full list: 20580
```

Beautiful! Looks like these lists contain our training and test splits and the full list has a list of all the files in the dataset.

Let's inspect the `train_list['file_list']` further.

In [12]:
```
train_list['file_list']
```

Out[12]:
```
array([[array(['n02085620-Chihuahua/n02085620_5927.jpg'], dtype='<U38')],
       [array(['n02085620-Chihuahua/n02085620_4441.jpg'], dtype='<U38')],
       [array(['n02085620-Chihuahua/n02085620_1502.jpg'], dtype='<U38')],
       ...,
       [array(['n02116738-African_hunting_dog/n02116738_6754.jpg'], dtype='<U4
```

```
8')],
       [array(['n02116738-African_hunting_dog/n02116738_9333.jpg'], dtype='<U4
8')],
       [array(['n02116738-African_hunting_dog/n02116738_2503.jpg'], dtype='<U4
8')]],
      dtype=object)
```

Looks like we've got an array of arrays.

How about we turn them into a Python list for easier handling?

We can do so by extracting each individual item via indexing and list comprehension.

Let's see what it's like to get a single file name.

In [13]:
```
# Get a single filename
train_list['file_list'][0][0][0]
```

Out[13]:
```
'n02085620-Chihuahua/n02085620_5927.jpg'
```

Now let's get a Python list of all the individual file names (e.g. `n02097130-giant_schnauzer/n02097130_2866.jpg`) so we can use them later.

In [14]:
```
# Get a Python list of all file names for each list
train_file_list = list([item[0][0] for item in train_list["file_list"]])
test_file_list = list([item[0][0] for item in test_list["file_list"]])
full_file_list = list([item[0][0] for item in file_list["file_list"]])

len(train_file_list), len(test_file_list), len(full_file_list)
```

Out[14]:
```
(12000, 8580, 20580)
```

Wonderful!

How about we view a random sample of the filenames we extracted?

> **Note:** One of my favourite things to do whilst exploring data is to continually view random samples of it. Whether it be file names or images or text snippets. Why? You can always view the first X number of samples, however, I find that continually viewing random samples of the data gives you a better of overview of the different kinds of data you're working with. It also gives you the small chance of stumbling upon a potential error.

We can view random samples of the data using Python's `random.sample()` method.

In [15]:
```
import random

random.sample(train_file_list, k=10)
```

Out[15]:
```
['n02094258-Norwich_terrier/n02094258_439.jpg',
 'n02113624-toy_poodle/n02113624_3624.jpg',
 'n02102973-Irish_water_spaniel/n02102973_3635.jpg',
 'n02102318-cocker_spaniel/n02102318_2048.jpg',
 'n02098286-West_Highland_white_terrier/n02098286_1261.jpg',
 'n02088238-basset/n02088238_10095.jpg',
 'n02108915-French_bulldog/n02108915_9457.jpg',
 'n02098286-West_Highland_white_terrier/n02098286_5979.jpg',
 'n02109047-Great_Dane/n02109047_31274.jpg',
 'n02095889-Sealyham_terrier/n02095889_760.jpg']
```

Now let's do a quick check to make sure none of the training image file names appear in the testing image file names list.

This is important because the number 1 rule in machine learning is: **always keep the test set separate from the training set**.

We can check that there are no overlaps by turning `train_file_list` into a Python `set()` and using the `intersection()` method.

```
In [16]: # How many files in the training set intersect with the testing set?
         len(set(train_file_list).intersection(test_file_list))
```

```
Out[16]: 0
```

Excellent! Looks like there are no overlaps.

We could even put an `assert` check to raise an error if there are any overlaps (e.g. the length of the intersection is greater than 0).

`assert` works in the fashion: `assert expression, message_if_expression_fails`.

If the `assert` check doesn't output anything, we're good to go!

```
In [17]: # Make an assertion statement to check there are no overlaps (try changing test_fil
         assert len(set(train_file_list).intersection(test_file_list)) == 0, "There are ov
```

Woohoo!

Looks like there's no overlaps, let's keep exploring the data.

Exploring the Annotation folder

How about we look at the `Annotation` folder next?

We can click the folder on the file explorer on the left to see what's inside.

But we can also explore the contents of the folder with Python.

Let's use `os.listdir()` to see what's inside.

```
In [18]: os.listdir("Annotation")[:10]
```

```
Out[18]: ['n02111129-Leonberg',
          'n02102973-Irish_water_spaniel',
          'n02110806-basenji',
          'n02105251-briard',
          'n02093991-Irish_terrier',
          'n02099267-flat-coated_retriever',
          'n02110627-affenpinscher',
          'n02112137-chow',
          'n02094114-Norfolk_terrier',
          'n02095570-Lakeland_terrier']
```

Looks like there are files each with a dog breed name with several numbered files inside.

Each of the files contains a HTML version of an annotation relating to an image.

For example, `Annotation/n02085620-Chihuahua/n02085620_10074`:

```
<annotation>
        <folder>02085620</folder>
        <filename>n02085620_10074</filename>
        <source>
                <database>ImageNet database</database>
        </source>
        <size>
                <width>333</width>
                <height>500</height>
                <depth>3</depth>
```

```
        </size>
        <segment>0</segment>
        <object>
                <name>Chihuahua</name>
                <pose>Unspecified</pose>
                <truncated>0</truncated>
                <difficult>0</difficult>
                <bndbox>
                        <xmin>25</xmin>
                        <ymin>10</ymin>
                        <xmax>276</xmax>
                        <ymax>498</ymax>
                </bndbox>
        </object>
</annotation>
```

The fields include the name of the image, the size of the image, the label of the object and where it is (bounding box coordinates).

If we were performing object detection (finding the location of a thing in an image), we'd pay attention to the `<bndbox>` coordinates.

However, since we're focused on classification, our main consideration is the mapping of image name to class name.

Since we're dealing with 120 classes of dog breed, let's write a function to check the number of subfolders in the `Annotation` directory (there should be 120 subfolders, one for each breed of dog).

To do so, we can use Python's `pathlib.Path` class, along with `Path.iterdir()` to loop over the contents of `Annotation` and `Path.is_dir()` to check if the target item is a directory.

In [19]:
```python
from pathlib import Path

def count_subfolders(directory_path: str) -> int:
    """
    Count the number of subfolders in a given directory.

    Args:
    directory_path (str): The path to the directory in which to count subfolders.

    Returns:
    int: The number of subfolders in the specified directory.

    Examples:
    >>> count_subfolders('/path/to/directory')
    3  # if there are 3 subfolders in the specified directory
    """
    return len([name for name in Path(directory_path).iterdir() if name.is_dir()])


directory_path = "Annotation"
folder_count = count_subfolders(directory_path)
print(f"Number of subfolders in {directory_path} directory: {folder_count}")
```

```
Number of subfolders in Annotation directory: 120
```

Perfect!

There are 120 subfolders of annotations, one for each class of dog we'd like to identify.

But on further inspection of our file lists, it looks like the class name is already in the filepath.

In [20]:
```python
# View a single training file pathname
train_file_list[0]
```

```
Out[20]: 'n02085620-Chihuahua/n02085620_5927.jpg'
```

With this information we know, that image `n02085620_5927.jpg` should contain a `Chihuahua`.

Let's check.

I searched "how to display an image in Google Colab" and found another answer on Stack Overflow.

Turns out you can use `IPython.display.Image()`, as Google Colab comes with IPython (Interactive Python) built-in.

```
In [21]: from IPython.display import Image
         Image(Path("Images", train_file_list[0]))
```

Out[21]:

Woah!

We get an image of a dog!

Exploring the Images folder

We've explored the `Annotations` folder, now let's check out our `Images` folder.

We know that the image file names come in the format `class_name/image_name`, for example, `n02085620-Chihuahua/n02085620_5927.jpg`.

To make things a little simpler, let's create the following:

1. A mapping from folder name -> class name in dictionary form, for example, `{'n02113712-miniature_poodle': 'miniature_poodle', 'n02092339-Weimaraner': 'weimaraner', 'n02093991-Irish_terrier': 'irish_terrier'...}`. This will help us when visualizing our data from its original folder.
2. A list of all unique dog class names with simple formatting, for example, `['affenpinscher', 'afghan_hound', 'african_hunting_dog', 'airedale', 'american_staffordshire_terrier'...]`.

Let's start by getting a list of all the folders in the `Images` directory with `os.listdir()`.

In [22]:
```
# Get a list of all image folders
image_folders = os.listdir("Images")
image_folders[:10]
```

Out[22]:
```
['n02111129-Leonberg',
 'n02102973-Irish_water_spaniel',
```

```
'n02110806-basenji',
'n02105251-briard',
'n02093991-Irish_terrier',
'n02099267-flat-coated_retriever',
'n02110627-affenpinscher',
'n02112137-chow',
'n02094114-Norfolk_terrier',
'n02095570-Lakeland_terrier']
```

Excellent!

Now let's make a dictionary which maps from the folder name to a simplified version of the class name, for example:

```
{'n02085782-Japanese_spaniel': 'japanese_spaniel',
'n02106662-German_shepherd': 'german_shepherd',
'n02093256-Staffordshire_bullterrier': 'staffordshire_bullterrier',
...}
```

In [23]:

```python
# Create folder name -> class name dict
folder_to_class_name_dict = {}
for folder_name in image_folders:
  # Turn folder name into class_name
  # E.g. "n02089078-black-and-tan_coonhound" -> "black_and_tan_coonhound"
  # We'll split on the first "-" and join the rest of the string with "_" and then
  class_name = "_".join(folder_name.split("-")[1:]).lower()
  folder_to_class_name_dict[folder_name] = class_name

# Make sure there are 120 entries in the dictionary
assert len(folder_to_class_name_dict) == 120
```

Folder name to class name mapping created, let's view the first 10.

```
In [24]: list(folder_to_class_name_dict.items())[:10]
```

```
Out[24]: [('n02111129-Leonberg', 'leonberg'),
          ('n02102973-Irish_water_spaniel', 'irish_water_spaniel'),
          ('n02110806-basenji', 'basenji'),
          ('n02105251-briard', 'briard'),
          ('n02093991-Irish_terrier', 'irish_terrier'),
          ('n02099267-flat-coated_retriever', 'flat_coated_retriever'),
          ('n02110627-affenpinscher', 'affenpinscher'),
          ('n02112137-chow', 'chow'),
          ('n02094114-Norfolk_terrier', 'norfolk_terrier'),
          ('n02095570-Lakeland_terrier', 'lakeland_terrier')]
```

And we can get a list of unique dog names by getting the `values()` of the `folder_to_class_name_dict` and turning it into a list.

```
In [25]: dog_names = sorted(list(folder_to_class_name_dict.values()))
         dog_names[:10]
```

```
Out[25]: ['affenpinscher',
          'afghan_hound',
          'african_hunting_dog',
          'airedale',
          'american_staffordshire_terrier',
          'appenzeller',
          'australian_terrier',
          'basenji',
          'basset',
          'beagle']
```

Perfect!

Now we've got:

1. `folder_to_class_name_dict` - a mapping from the folder name to the class name.
2. `dog_names` - a list of all the unique dog breeds we're working with.

Visualize a group of random images

How about we follow the data explorers motto of *visualize, visualize, visualize* and view some random images?

To help us visualize, let's create a function that takes in a list of image paths and then randomly selects 10 of those paths to display.

The function will:

1. Take in a select list of image paths.
2. Create a grid of matplotlib plots (e.g. 2x5 = 10 plots to plot on).
3. Randomly sample 10 image paths from the input image path list (using `random.sample()`).
4. Iterate through the flattened axes via `axes.flat` which is a reference to the attribute `numpy.ndarray.flat`.
5. Extract the sample path from the list of samples.
6. Get the sample title from the parent folder of the path using `Path.parent.stem` and then extract the formatted dog breed name by indexing `folder_to_class_name_dict`.
7. Read the image with `plt.imread()` and show it on the target `ax` with `ax.imshow()`.
8. Set the title of the plot to the parent folder name with `ax.set_title()` and turn the axis marks of with `ax.axis("off")` (this makes for pretty plots).

9. Show the plot with `plt.show()`.

Woah!

A lot of steps! But nothing we can't handle, let's do it.

In [26]:

```python
import random

from pathlib import Path
from typing import List

import matplotlib.pyplot as plt

# 1. Take in a select list of image paths
def plot_10_random_images_from_path_list(path_list: List[Path],
                                         extract_title: bool=True) -> None:
  # 2. Set up a grid of plots
  fig, axes = plt.subplots(nrows=2, ncols=5, figsize=(20, 10))

  # 3. Randomly sample 10 paths from the list
  samples = random.sample(path_list, 10)

  # 4. Iterate through the flattened axes and corresponding sample paths
  for i, ax in enumerate(axes.flat):

    # 5. Get the target sample path (e.g. "Images/n02087394-Rhodesian_ridgeback/n02
    sample_path = samples[i]

    # 6. Extract the parent directory name to use as the title (if necessary)
    # (e.g. n02087394-Rhodesian_ridgeback/n02087394_1161.jpg -> n02087394-Rhodesian
    if extract_title:
      sample_title = folder_to_class_name_dict[sample_path.parent.stem]
    else:
      sample_title = sample_path.parent.stem
```

```
    # 7. Read the image file and plot it on the corresponding axis
    ax.imshow(plt.imread(sample_path))

    # 8. Set the title of the axis and turn of the axis (for pretty plots)
    ax.set_title(sample_title)
    ax.axis("off")

  # 9. Display the plot
  plt.show()

plot_10_random_images_from_path_list(path_list=[Path("Images") / Path(file) for f
```

Those are some nice looking dogs!

What I like to do here is rerun the random visualizations until I've seen 100+ samples so I've got an idea of the data we're working with.

> **Question:** Here's something to think about, how would you code a system of rules to differentiate between all the different breeds of dogs? Perhaps you write an algorithm to look at the shapes or the colours? For example, if the dog had black fur, it's unlikely to be a golden retriever. You might be thinking "that would take quite a long time..." And you'd be right. Then how would we do it? With machine learning of course!

Exploring the distribution of our data

After visualization, another valuable way to explore the data is by checking the data distribution.

Distribution refers to the "spread" of data.

In our case, how many images of dogs do we have per breed?

A balanced distribution would mean having roughly the same number of images for each breed (e.g. 100 images per dog breed).

> **Note:** There's a deeper level of distribution than just images per dog breed. Ideally, the images for each different breed are well distributed as well. For example, we wouldn't want to have 100 of the same image per dog breed. Not only would we like a similar number of images per breed, we'd like the images of each particular breed to be in different scenarios, different lighting, different angles. We want this because we want to our model to be able to recognize the correct dog breed no matter what angle the photo is taken from.

To figure out how many images we have per class, let's write a function count the number of images per subfolder in a given directory.

Specifically, we'll want the function to:

1. Take in a target directory/folder.
2. Create a list of all the subdirectories/subfolders in the target folder.
3. Create an empty list, `image_class_counts` to append subfolders and their counts to.
4. Iterate through all of the subdirectories.
5. Get the class name of the target folder as the name of the folder.
6. Count the number of images in the target folder using the length of the list of image paths (we can get these with `Path().rglob(*.jpg)` where `*.jpg` means "all files with the extension `.jpg`.
7. Append a dictionary of `{"class_name": class_name, "image_count": image_count}` to the `image_class_counts` list (we create a list of dictionaries so we can turn this into a pandas DataFrame).
8. Return the `image_class_counts` list.

In [27]:

```python
# Create a dictionary of image counts
from pathlib import Path
from typing import List, Dict

# 1. Take in a target directory
def count_images_in_subdirs(target_directory: str) -> List[Dict[str, int]]:
    """
    Counts the number of JPEG images in each subdirectory of the given directory.

    Each subdirectory is assumed to represent a class, and the function counts
    the number of '.jpg' files within each one. The result is a list of
    dictionaries with the class name and corresponding image count.

    Args:
        target_directory (str): The path to the directory containing subdirectori
```

```
    Returns:
        List[Dict[str, int]]: A list of dictionaries with 'class_name' and 'image

    Examples:
        >>> count_images_in_subdirs('/path/to/directory')
        [{'class_name': 'beagle', 'image_count': 50}, {'class_name': 'poodle', 'i
    """
    # 2. Create a list of all the subdirectoires in the target directory (these cor
    images_dir = Path(target_directory)
    image_class_dirs = [directory for directory in images_dir.iterdir() if direct

    # 3. Create an empty list to append image counts to
    image_class_counts = []

    # 4. Iterate through all of the subdirectories
    for image_class_dir in image_class_dirs:

        # 5. Get the class name from image directory (e.g. "Images/n02116738-Africa
        class_name = image_class_dir.stem

        # 6. Count the number of images in the target subdirectory
        image_count = len(list(image_class_dir.rglob("*.jpg")))  # get length all

        # 7. Append a dictionary of class name and image count to count list
        image_class_counts.append({"class_name": class_name,
                                   "image_count": image_count})

    # 8. Return the list
    return image_class_counts
```

Ho ho, what a function!

Let's run it on our target directory `Images` and view the first few indexes.

In [28]:
```
image_class_counts = count_images_in_subdirs("Images")
image_class_counts[:3]
```

Out[28]:
```
[{'class_name': 'n02111129-Leonberg', 'image_count': 210},
 {'class_name': 'n02102973-Irish_water_spaniel', 'image_count': 150},
 {'class_name': 'n02110806-basenji', 'image_count': 209}]
```

Nice!

Since our `image_class_counts` variable is the form of a list of dictionaries, we can turn it into a pandas `DataFrame`.

Let's sort the `DataFrame` by `"image_count"` so the classes with the most images appear at the top, we can do so with `DataFrame.sort_values()`.

In [29]:
```
# Create a DataFrame
import pandas as pd
image_counts_df = pd.DataFrame(image_class_counts).sort_values(by="image_count",
image_counts_df.head()
```

Out[29]:

	class_name	image_count
116	n02085936-Maltese_dog	252
53	n02088094-Afghan_hound	239
111	n02092002-Scottish_deerhound	232
103	n02112018-Pomeranian	219
54	n02107683-Bernese_mountain_dog	218

And let's cleanup the `"class_name"` column to be more readable by mapping the the values to our `folder_to_class_name_dict`.

In [30]:
```
# Make class name column easier to read
image_counts_df["class_name"] = image_counts_df["class_name"].map(folder_to_class
image_counts_df.head()
```

Out[30]:

	class_name	image_count
116	maltese_dog	252
53	afghan_hound	239
111	scottish_deerhound	232
103	pomeranian	219
54	bernese_mountain_dog	218

Now we've got a `DataFrame` of image counts per class, we can make them more visual by turning them into a plot.

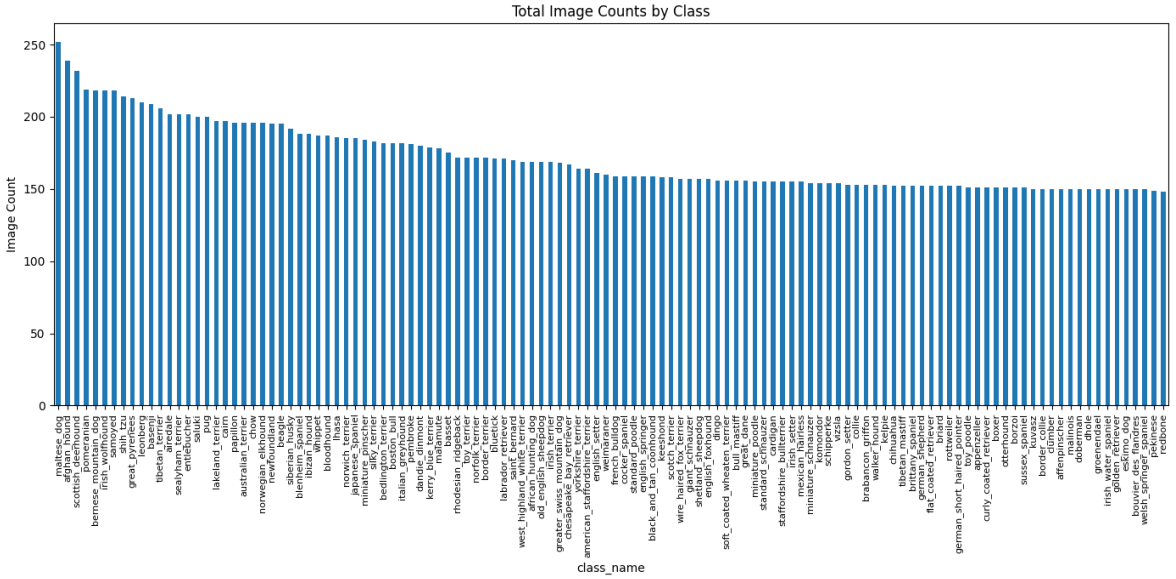
We covered plotting data directly from pandas `DataFrame`'s in *Section 3 of the Introduction to Matplotlib notebook: Plotting data directly with pandas*.

To do so, we can use `image_counts_df.plot(kind="bar", ...)` along with some other customization.

In [31]:
```
# Turn the image counts DataFrame into a graph
import matplotlib.pyplot as plt
plt.figure(figsize=(14, 7))
image_counts_df.plot(kind="bar",
                     x="class_name",
                     y="image_count",
                     legend=False,
                     ax=plt.gca()) # plt.gca() = "get current axis", get the plt w

# Add customization
plt.ylabel("Image Count")
```

```
plt.title("Total Image Counts by Class")
plt.xticks(rotation=90, # Rotate the x labels for better visibility
           fontsize=8) # Make the font size smaller for easier reading
plt.tight_layout() # Ensure things fit nicely
plt.show()
```



Beautiful! It looks like our classes are quite balanced. Each breed of dog has ~150 or more images.

We can find out some other quick stats about our data with `DataFrame.describe()`.

In [32]:
```
# Get various statistics about our data distribution
image_counts_df.describe()
```

Out[32]:

	image_count
count	120.000000

mean	171.500000
std	23.220898
min	148.000000
25%	152.750000
50%	159.500000
75%	186.250000
max	252.000000

And the table shows a similar story to the plot. We can see the minimum number of images per class is 148, where as the maximum number of images is 252.

If one class had 10x less images than another class, we may look into collecting more data to improve the balance.

The main takeaway(s):

- When working on a classification problem, ideally, all classes have a similar number of samples (however, in some problems this may be unattainable, such as fraud detection, where you may have 1000x more "not fraud" samples to "fraud" samples.
- If you wanted to add a new class of dog breed to the existing 120, ideally, you'd have at least ~150 images for it (though as we'll see with transfer learning, the number of required images could be less as long as they're high quality).

4. Creating training and test data split directories

After exploring the data, one of the next best things you can do is create experimental data splits.

This includes:

Set Name	Description	Typical Percentage of Data
Training Set	A dataset for the model to learn on	70-80%
Testing Set	A dataset for the model to be evaluated on	20-30%
(Optional) Validation Set	A dataset to tune the model on	50% of the test data
(Optional) Smaller Training Set	A smaller size dataset to run quick experiments on	5-20% of the training set

Our dog dataset already comes with specified training and test set splits.

So we'll stick with those.

But we'll also create a smaller training set (a random 10% of the training data) so we can stick to the machine learning engineers motto of *experiment, experiment, experiment!* and run quicker experiments.

> **Note:** One of the most important things in machine learning is being able to experiment quickly. As in, try a new model, try a new set of hyperparameters or try a new training setup. When you start out, you want the time between your experiments to be as small as possible so you can quickly figure out what *doesn't work* so you can spend more time on and run larger experiments with what *does* work.

As previously discussed, we're working towards a directory structure of:

```
images_split/
├── train/
```

```
|   ├── class_1/
|   |   ├── train_image1.jpg
|   |   ├── train_image2.jpg
|   |   └── ...
|   ├── class_2/
|   |   ├── train_image1.jpg
|   |   ├── train_image2.jpg
|   |   └── ...
└── test/
    ├── class_1/
    |   ├── test_image1.jpg
    |   ├── test_image2.jpg
    |   └── ...
    ├── class_2/
    |   ├── test_image1.jpg
    |   ├── test_image2.jpg
...
```

So let's write some code to create:

- `images/train/` directory to hold all of the training images.
- `images/test/` directory to hold all of the testing images.
- Make a directory inside each of `images/train/` and `images/test/` for each of the dog breed classes.

We can make each of the directories we need using `Path.mkdir()`.

For the dog breed directories, we'll loop through the list of `dog_names` and create a folder for each inside the `images/train/` and `images/test/` directories.

In [33]:

```python
from pathlib import Path

# Define the target directory for image splits to go
images_split_dir = Path("images_split")

# Define the training and test directories
train_dir = images_split_dir / "train"
test_dir = images_split_dir / "test"

# Using Path.mkdir with exist_ok=True ensures the directory is created only if it d
train_dir.mkdir(parents=True, exist_ok=True)
test_dir.mkdir(parents=True, exist_ok=True)
print(f"Directory {train_dir} is exists.")
print(f"Directory {test_dir} is exists.")

# Make a folder for each dog name
for dog_name in dog_names:
  # Make training dir folder
  train_class_dir = train_dir / dog_name
  train_class_dir.mkdir(parents=True, exist_ok=True)
  # print(f"Making directory: {train_class_dir}")

  # Make testing dir folder
  test_class_dir = test_dir / dog_name
  test_class_dir.mkdir(parents=True, exist_ok=True)
  # print(f"Making directory: {test_class_dir}")

# Make sure there is 120 subfolders in each
assert count_subfolders(train_dir) == len(dog_names)
assert count_subfolders(test_dir) == len(dog_names)
```

```
Directory images_split/train is exists.
Directory images_split/test is exists.
```

Excellent!