



# Projects

## Cats vs dogs

Take a machine vision model that is bad at recognising cats and dogs, and train it to be really good at it.



### Step 1 Introduction

#### What you will make

You will take an existing model trained to recognise lots of different kinds of images and retrain it to determine whether an image shows dogs or cats. The existing model already knows how to identify interesting features of an image. You will retrain the final layer of the model, which decides what those interesting features make up.

You will also measure how well the model can do this new job, and show how much you've improved on the original model in this task.



#### What you should already know

This project assumes you already know some Python. Specifically, it assumes you know how to use:

- Variables
- Lists
- Functions, including creating your own function that accepts parameters

The project also assumes that you know the basics of how to interact with an image classifying model and get a prediction from it. If you don't, you can learn this in the **'Testing your computer's vision' project** (<https://projects.raspberrypi.org/en/projects/testing-your-computers-vision>).



#### What you will need

- A computer
- An internet connection
- A Google account



### What you will learn

- The structure of image recognition models
- How those models are trained
- How to measure how well a model works



### Additional information for educators

If you need to print this project, please use the **printer-friendly version** (<https://projects.raspberrypi.org/en/projects/cats-vs-dogs/print>).

Here is a link to the resources for this project (<https://rpf.io/p/en/cats-vs-dogs-go>).

## Step 2 Prepare your model for retraining

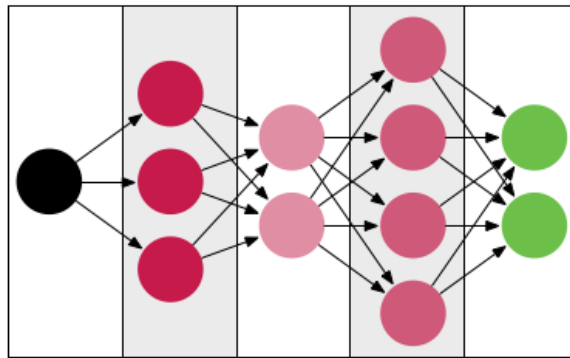
Instead of training a whole new model, for this project you'll load an existing one, called MobileNetV2, and change what it classifies. The MobileNetV2 model is trained to classify lots of different images, which means it is already trained to identify interesting features of an image. So retraining it to use these features to recognise cats or dogs will be much quicker than if you created an 'Is this a cat or a dog?' model from scratch.



### What is a machine learning model?

A model is a set of rules a computer has learned to complete a task, such as determining what is in a picture it's shown.

These rules are divided up into **layers**. Each layer looks at the results of the one before it and tests them against some rules to decide what results to pass to the next layer.



Within each layer there are nodes, which have each learned a rule while the model was being trained, and will test it and produce a result when the model is predicting.

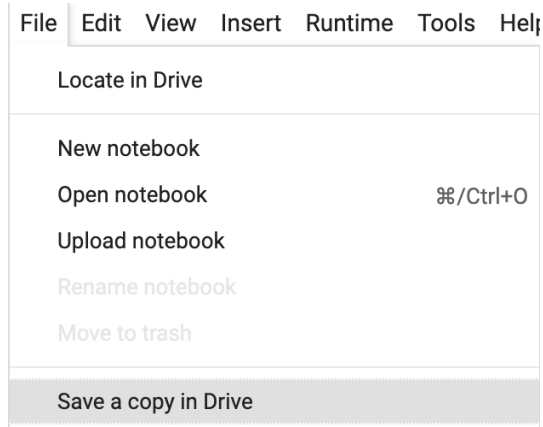
The very first layer is the input to the model — an image in this case. The first layer is often called the input layer for this reason. The last (output) layer of a classifier model like this will always have a number of nodes equal to the number of classifications the model is trained to identify. For example, in the model for this project, there will be two nodes in the final layer, as it will classify its inputs into either pictures of dogs or pictures of cats.

A Google Colab project has been prepared for you with some starter code. The first thing to do is to open that up and save your own copy to work on.

Open the **Google Colab starter notebook** (<https://colab.research.google.com/drive/1uKqhEOSu9pIKVwgw4G0Hqeq-jzPaYYMH#scrollTo=gebsfn75wKRg>) for this project in a new tab in your browser.



Before you start changing anything, make sure you save the notebook to your drive so you can keep your work! Choose **File > Save a copy in Drive** and sign in to your Google account if prompted.



First, define the size of the images that you will use. The dataset that you'll use to train the model is made up of 160x160 pixel images and that value is already stored in an **IMAGE\_SIZE** variable. However, because of how colour works on computers, the images are actually three sets of 160x160 pixels — one each of the red, blue, and green values that combine to form the colour displayed at any given pixel. You can see more details on this below.



In the first empty cell, create an **IMAGE\_SHAPE** variable:

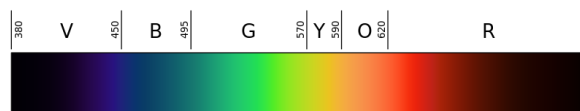
```
IMAGE_SHAPE = (IMAGE_SIZE, IMAGE_SIZE, 3)
```

This tells the program that images will be 160x160 pixels, with three layers of colour.



### Representing colours with numbers

The colour of an object depends on the colour of the light that it reflects or emits. Light can have different wavelengths, and the colour of light depends on the wavelength it has. The colour of light according to its wavelength can be seen in the diagram below. You might recognise this as the colours of the rainbow.



Humans see colour because of special cells in our eyes. These cells are called *cones*. We have three types of cone cells, and each type detects either red, blue, or green light. Therefore all the colours that we see are just mixtures of the colours red, blue, and green.



In additive colour mixing, three colours (red, green, and blue) are used to make other colours. In the image above, there are three spotlights of equal brightness, one for each colour. In the absence of any colour the result is black. If all three colours are mixed, the result is white. When red and green combine, the result is yellow. When red and blue combine, the result is magenta. When blue and green combine, the result is cyan. It's possible to make even more colours than this by varying the brightness of the three original colours used.

Computers store everything as 1s and 0s. These 1s and 0s are often organised into sets of 8, called **bytes**.

A single `byte()` can represent any number from 0 up to 255.

When we want to represent a colour in a computer program, we can do this by defining the amounts of red, blue, and green that make up that colour. These amounts are usually stored as a single `byte()` and therefore as a number between 0 and 255.

Here's a table showing some colour values:

#### Red Green Blue Colour

255	0	0	Red
0	255	0	Green
0	0	255	Blue
255	255	0	Yellow
255	0	255	Magenta
0	255	255	Cyan

You can find a nice **colour picker to play with at w3schools** ([https://www.w3schools.com/colors/colors\\_rgb.asp](https://www.w3schools.com/colors/colors_rgb.asp)).

Now, below your `IMAGE_SHAPE` variable, import the MobileNetV2 model – which is trained to identify loads of different objects – pass it your `IMAGE_SHAPE` as its `input_shape` and store it in an `original_model` variable.



```
original_model = tf.keras.applications.MobileNetV2(input_shape=IMAGE_SHAPE)
```

Since MobileNetV2 is designed to run on a mobile device with limited battery, like a phone, it's not as large or as powerful as some other models. This means it doesn't always make the best guesses. Before you start changing it, test how good it is by asking it to identify a photo of a dog. Functions to let you do this easily have already been included in the notebook, but to understand how they work, check out the **'Testing your computer's vision' project**.

Below the model import, add this line:



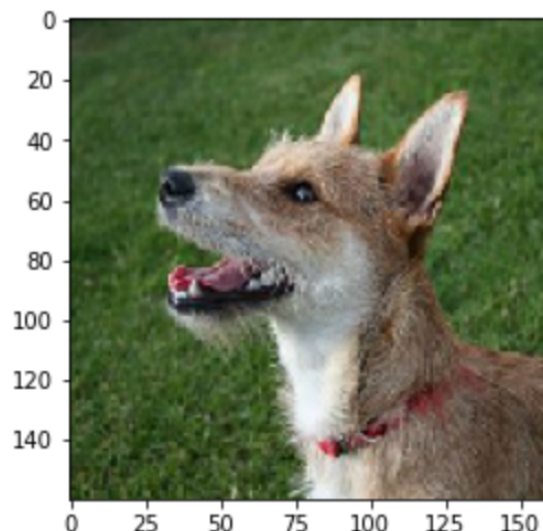
```
predict_with_old_model('https://dojo.soy/predict-dog')
```

Now run all the code and see how good the model's predictions are!



You can run all the code by opening the **Runtime** menu and choosing **Run all**. The first time you do this, it might take a while, because your program will have to download a lot of data both for the training dataset and the model itself.

1. matchstick 49.70%
2. pinwheel 8.62%
3. plunger 4.91%
4. syringe 3.78%
5. window\_screen 2.28%
6. umbrella 1.87%
7. Chihuahua 1.87%
8. envelope 1.82%
9. missile 1.38%
10. lipstick 1.28%
11. projectile 1.15%
12. jigsaw\_puzzle 1.14%
13. screwdriver 0.92%
14. space\_shuttle 0.76%
15. German\_shepherd 0.60%

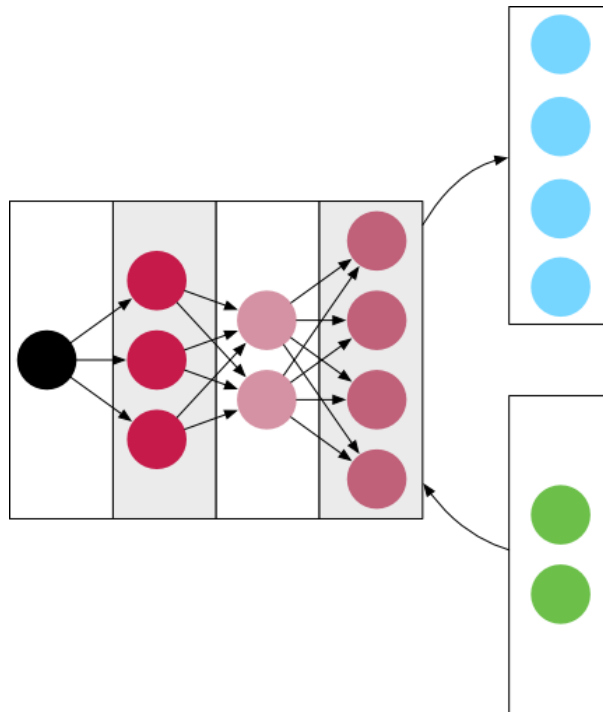


There are a couple of dog breeds in there, but most of the model's preferred classifications aren't very good! That's why you're going to improve it!

Remove the call to `predict_with_old_model`. You only needed it for testing.



You need to remove the top layer from the existing MobileNetV2 model. This is where the model decides which of the objects it's been trained to identify are in the image, so you need to remove it to add your own layers related to cats and dogs. You can do this when you load the model.



Update the line where you load the `original_model` to add the `include_top` parameter and set it to `False`.



```
original_model = tf.keras.applications.MobileNetV2(input_shape=IMAGE_SHAPE, include_top=False)
```

Finally, because you don't want to change anything else in the original model, you should set it to be untrainable.

Below the line where you create `original_model`, set its `trainable` property to `False`.



```
original_model.trainable = False
```



**Save your project**

## Step 3 Get and split training data

---

Next, you'll need to train a model with some data. Models learn from examples, usually thousands of them, and use those to create their own rules. Some of those rules may be to recognise parts and then combine that recognition into a whole. This can lead to problems if you're not using varied data to train your model. For example, if the training data for a cats vs dogs classifier has no pictures of black dogs, but many of black cats, the model may learn the rule 'black = cat'. This would cause problems if you later tried to classify pictures of a black dog. So you have to make sure that the full variety of things the model may eventually be asked to classify are represented. However, for this project, you will use an existing dataset of cats and dogs that is provided by the TensorFlow library, so someone has already done that work for you.

It's worth noting that one of the cool things about machine learning models is that they remember everything they learned from the training data, without needing to store the training data itself. This means you can use millions of images to train a model without making the model any bigger than if you'd used a few hundred images! However, a model trained on millions of images will be much better at guessing things correctly than one trained on a few hundred.

There's already some code in the second cell that loads the `cats_vs_dogs` training data. This data is a collection of image files and **labels** for these files, it tells the computer which image is of a cat, and which of a dog.

```
import tensorflow_datasets as tfds
(raw_training, raw_validation, raw_testing), metadata = tfds.load(
    'cats_vs_dogs',
    split=['train[:80%]', 'train[80%:90%]', 'train[90%:]'],
    with_info=True,
    as_supervised=True,
)
```

The data gets broken into three groups, with the percentages defined in the `split` parameter of the load call:

Training data: Used to train the model — to learn rules and decide how important they are.

Validation data: Used to evaluate how well the model is performing while it is being trained. It is checked regularly during the training process. It has to be separate to the training data, or the model might learn only the exact images in the training data, with no general rules for identifying a dog or cat.

Testing data: Not used during the training process, but is used to evaluate how well it performs on unseen data. This check is used to avoid the risk of **overfitting**, where the model learns rules that are specific to the training and validation datasets, but do not apply to all cats and dogs.

The data in those three groups needs to be broken into **batches** — groups of images. Each batch gets used to train the model before the model's **weights**, which define how important each rule is, get updated. The bigger the batch, the longer the gap between updates.

There's no real rule for how big your batches should be, and it may be worth experimenting with different batch sizes on different data, to see if you get better results. Smaller batches can also help you train a model on a computer with less memory (for example, on a Raspberry Pi). Popular sizes are 32, 64, 128, and 256. It is possible to use batches as small as a single image, or as large as your whole dataset. For now, you'll use 32.



In the second blank cell in the notebook, create a `BATCH_SIZE` variable with the value of 32.



```
BATCH_SIZE = 32
```

The use of upper case letters here is a convention for variables that are manually set by the programmer, but not changed in the course of the program. You could just enter 32 directly where the variable gets used, but this makes it easier to update later.

Each batch is chosen randomly from a **shuffle buffer** of images selected from the full test, validation, or training dataset. You have to define the size of this buffer. The bigger it is, the more randomly shuffled your images will be, which is usually a good thing, but the slower the program will run. Again, start by creating a variable for the buffer size.

Below your previous `BATCH_SIZE` variable, add this code:



```
SHUFFLE_BUFFER_SIZE = 1000
```

Now that you've set your batch and buffer sizes, you need to break your training, validation, and testing data up into batches. This code creates the shuffle buffers and chooses the batches from them. Add it below the two variables you just created.



```
training_batches = training_data.shuffle(SHUFFLE_BUFFER_SIZE).batch(BATCH_SIZE)
validation_batches = validation_data.shuffle(SHUFFLE_BUFFER_SIZE).batch(BATCH_SIZE)
testing_batches = testing_data.shuffle(SHUFFLE_BUFFER_SIZE).batch(BATCH_SIZE)
```



**Save your project**

## Step 4 Add your new layers

In order to train the MobileNetV2 model you've loaded to identify cats and dogs, you need to add two layers to it. One to convert the existing outputs of the model into a format that makes sense for your images, and the other to give the final classification of the image as either a cat or a dog.

The model doesn't actually know these are cats and dogs, of course, it's just using the two nodes to represent two different categories of things. Which category is cats, and which is dogs, is something that the pre-written `predict_image` function — which you'll use to test images later — will translate for you by using the **labels** that came with the original training data.

In the next blank cell in the notebook, add this line of code to create the layer that reshapes the outputs of the existing model and stores it in a variable.



```
global_average_layer = tf.keras.layers.GlobalAveragePooling2D()
```

You just used that layer type as TensorFlow supplies it, but for the next one you need to take what you know about your data and apply it: because this classifier has to decide between two classes, its output should be two numbers. The higher number will indicate the predicted class.

Below the `global_average_layer`, add a variable for `prediction_layer`:



```
prediction_layer = tf.keras.layers.Dense(2)
```

Now you need to put your layers together with the original model you imported. This is done by using the `Sequential` function, because you assemble them in the sequence you pass them to the function, from bottom to top.

Below your `prediction_layer` variable, add the following code to combine your layers with the MobileNetV2 model.




```
model = tf.keras.Sequential([
    original_model,
    global_average_layer,
    prediction_layer
])
```

This stacks the `global_average_layer` and the `prediction_layer` on top of the `original_model`.

Next, you have to **compile** your model — convert it into a form that you can train and use to make predictions. To do this, you need to set two values: the **learning rate** of your model, and a **loss function**.

The learning rate tells your model how quickly to learn. You don't want it too small, as it might take far too long to learn anything. However, you don't want it too big or your model may rush ahead with the first thing that seems right, and miss an important rule or insight. You'll use a learning rate of **0.0001**.

The loss function is how your model checks its performance during training. You'll use one called cross-entropy loss, which is a good choice for image classification.


Below your existing code, create a variable for the base learning rate, then compile the model with that parameter and cross entropy loss. Also, tell the model to print out its accuracy as it is training. 

```
BASE_LEARNING_RATE = 0.0001
model.compile(optimizer=tf.keras.optimizers.RMSprop(lr=BASE_LEARNING_RATE),
              loss=tf.keras.losses.SparseCategoricalCrossentropy(),
              metrics=['accuracy'])
```

Finally, take a look at the model you've compiled.

Below the code you already have, add this line: 

```
model.summary()
```

Run all the code in the notebook by opening the **Runtime** menu and choosing **Run all**. 

Layer (type)	Output Shape	Param #
mobilenetv2_1.00_160 (Model)	(None, 5, 5, 1280)	2257984
global_average_pooling2d_2 (	(None, 1280)	0
dense_1 (Dense)	(None, 1)	1281
Total params: 2,259,265		
Trainable params: 1,281		
Non-trainable params: 2,257,984		

Look at the summary of the model that's printed out. In particular, look at the last three lines. This model has over two million parameters, but you're only training a little over a thousand of them. This is the huge advantage of retraining a model, over building an entirely new one; there's a lot less to train, so it will train much faster.



**Save your project**

## Step 5 Retrain the model

You've prepared your data, you've built your model, now it's time to get training!

The first thing you need to do is set the number of **epochs** you'll train for. An epoch is a complete pass through the training data. Usually, you want to go through the training data multiple times, in order to get the best result. However, each epoch adds to the time it will take to train the model, so you don't want to use too many. For now, you'll use ten epochs.

In the next blank cell, create a **TRAINING\_EPOCHS** variable and set it to **10**.



```
TRAINING_EPOCHS = 10
```

However, even with only ten epochs, this is still a lot of data and will take a long time to train. In fact, if you were to train the model right now, it would take over an hour! Luckily, Colab allows you to set your notebook to use GPUs – graphics processing units, the same hardware used for video games – instead of CPUs – central processing units, the general-purpose processor that does most of the work on a computer. Because of the kind of mathematics the computer carries out for machine learning, it turns out the GPUs are much faster than CPUs.

So tell TensorFlow to use the GPU device when it fits your model. You'll need to provide the training and validation batches you created earlier to the **model.fit** function.

Below your **TRAINING\_EPOCHS** line, add the following:




```
with tf.device('/device:GPU:0'):
    history = model.fit(training_batches,
                        epochs=TRAINING_EPOCHS,
                        validation_data=validation_batches)
```

Run all the code in the notebook by opening the **Runtime** menu and choosing **Run all**.



This is going to take some time, probably more than ten minutes. Leave the tab open and check back about once a minute to see your model printing out the results for each epoch. Watch as the loss goes down and the accuracy goes up!

```
Epoch 1/10
582/582 [=====] - 81s 139ms/step - loss: 1.4232 - accuracy: 0.8192 - val_loss: 0.3525 - val_accuracy: 0.9420
Epoch 2/10
582/582 [=====] - 76s 131ms/step - loss: 0.3300 - accuracy: 0.9546 - val_loss: 0.2301 - val_accuracy: 0.9669
Epoch 3/10
582/582 [=====] - 79s 135ms/step - loss: 0.2393 - accuracy: 0.9677 - val_loss: 0.2127 - val_accuracy: 0.9751
Epoch 4/10
582/582 [=====] - 77s 132ms/step - loss: 0.2130 - accuracy: 0.9737 - val_loss: 0.1918 - val_accuracy: 0.9781
Epoch 5/10
582/582 [=====] - 79s 136ms/step - loss: 0.1966 - accuracy: 0.9769 - val_loss: 0.1899 - val_accuracy: 0.9785
Epoch 6/10
582/582 [=====] - 79s 129ms/step - loss: 0.1861 - accuracy: 0.9790 - val_loss: 0.1799 - val_accuracy: 0.9794
Epoch 7/10
582/582 [=====] - 78s 134ms/step - loss: 0.1714 - accuracy: 0.9800 - val_loss: 0.1780 - val_accuracy: 0.9789
Epoch 8/10
582/582 [=====] - 76s 131ms/step - loss: 0.1653 - accuracy: 0.9819 - val_loss: 0.1810 - val_accuracy: 0.9794
Epoch 9/10
582/582 [=====] - 79s 135ms/step - loss: 0.1601 - accuracy: 0.9823 - val_loss: 0.1869 - val_accuracy: 0.9789
Epoch 10/10
582/582 [=====] - 75s 128ms/step - loss: 0.1549 - accuracy: 0.9833 - val_loss: 0.2024 - val_accuracy: 0.9811
```

 Save your project

## Step 6 Test your model

Now you've got a working model, you can use pictures of cats and dogs to test it!

In the last empty cell, add a call to `predict_image` and pass it the URL to a test image.



```
predict_image('https://dojo.soy/predict-dog')
```

Remember how long all that training took? You don't want to wait through that every time you want to test an image, so you need to stop using the **Run all** option. Instead, click the ► button that appears to the left of the cell to run only the contents of that cell.



```
predict_image('https://dojo.soy/predict-dog')
```

Run the image prediction code and check out the results!



### Save your project

Try to load a few different images into it to see what predictions it makes. You'll have to use images hosted on the internet. You can use the instructions below to store the images in Google Drive and create URLs for them that you can pass to `predict_image`.



You'll also have to make sure they're `.jpg` files, or modify the `get_image_from_url` function that was supplied with the notebook.



### Hosting an image in Google Drive

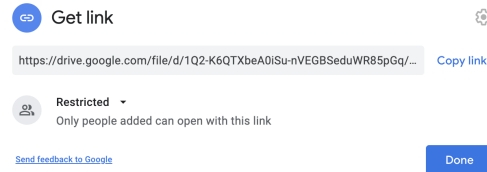
Follow these instructions to host an image in Google Drive and get a link that you can use to include that image for download in programs, web pages, etc.

```
<p>Go to <a href="https://drive.google.com/">drive.google.com</a> and drag an image from your computer to the drive. Wait for it to finish uploading.</p>
```

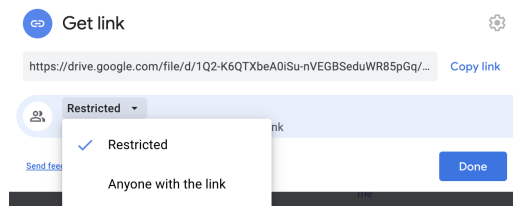




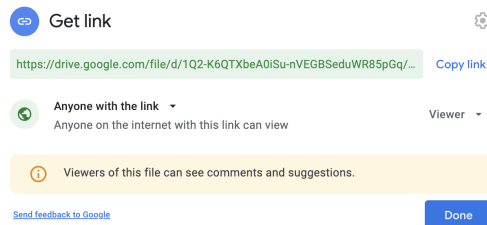
Once the image is in drive, right-click on it and choose 'Get shareable link'.



In the dialogue that opens, change who has permission to view the image from 'Restricted' to 'Anyone with the link'



There should now be a link highlighted in the dialogue box.



In this link you can find the ID code for the file between `/d/` and `/view?usp=sharing`. It should look something like this:

1xunlhWWxA6e59gSL\_gTo\_CiZYBNqbMDy

Copy this ID.



<p>Use the ID you have just copied to complete this URL, inserting it in place of <code>[IMAGE ID]</code>:  
</p>

`https://drive.google.com/uc?export=download&id=[IMAGE ID]`

You should get something like this:

`https://drive.google.com/uc?export=download&id=1xunlhWWxA6e59gSL_gTo_CiZYBNqbMDy`

This is the link you need to include in your code to allow the image to be downloaded.



## What next?

Now that you've created a model based on an existing one, you can make your own model in the **'Teach a computer to read' project**. (<https://projects.raspberrypi.org/en/projects/teach-a-computer-to-read>)

---

Published by Raspberry Pi Foundation (<https://www.raspberrypi.org>) under a Creative Commons license (<https://creativecommons.org/licenses/by-sa/4.0/>).

View project & license on GitHub (<https://github.com/RaspberryPiLearning/cats-vs-dogs>)