Transfer learning and fine-tuning

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np
import os
import tensorflow as tf
```

Data preprocessing

Data download

In this tutorial, you will use a dataset containing several thousand images of cats and dogs. Download and extract a zip file containing the images, then create a tf.data.Dataset for training and validation using the tf.keras.utils.image_dataset_from_directory utility. You can learn more about loading images in this tutorial.

In [2]:
```
_URL = 'https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip'
path_dir = tf.keras.utils.get_file('cats_and_dogs.zip', origin=_URL, extract=True)
PATH = os.path.join(path_dir, 'cats_and_dogs_filtered')

train_dir = os.path.join(PATH, 'train')
validation_dir = os.path.join(PATH, 'validation')

BATCH_SIZE = 32
IMG_SIZE = (160, 160)   # We'll use 160x160 images as expected by MobileNetV2
```

```
Downloading data from https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip
68606236/68606236 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
```

In [3]:
```
train_dataset = tf.keras.utils.image_dataset_from_directory(train_dir,
                                                            shuffle=True,
                                                            batch_size=BATCH_SIZE,
                                                            image_size=IMG_SIZE)
```

```
Found 2000 files belonging to 2 classes.
```

In [4]:
```
validation_dataset = tf.keras.utils.image_dataset_from_directory(validation_dir,
                                                                 shuffle=True,
                                                                 batch_size=BATCH_SIZE,
                                                                 image_size=IMG_SIZE)
```

```
Found 1000 files belonging to 2 classes.
```

Show the first nine images and labels from the training set:

In [6]:
```
class_names = train_dataset.class_names

plt.figure(figsize=(10, 10))
for images, labels in train_dataset.take(1):         # take one batch from the training dataset
  for i in range(9):                                 # display the first 9 images of the batch
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(images[i].numpy().astype("uint8"))    # convert image tensor to uint8 and show
    plt.title(class_names[labels[i]])                # set title as the class name
    plt.axis("off")
```

As the original dataset doesn't contain a test set, you will create one. To do so, determine how many batches of data are available in the validation set using tf.data.experimental.cardinality, then move 20% of them to a test set.

In [7]:
```
val_batches = tf.data.experimental.cardinality(validation_dataset)      # Determine the number of batches in the
test_dataset = validation_dataset.take(val_batches // 5)                # Reserve 20% of those batches for a te
validation_dataset = validation_dataset.skip(val_batches // 5)          # Skip the first 1/5th, keeping the res
```

In [8]:
```
# Print out how many batches are now in validation and test sets

print('Number of validation batches: %d' % tf.data.experimental.cardinality(validation_dataset))
print('Number of test batches: %d' % tf.data.experimental.cardinality(test_dataset))
```

```
Number of validation batches: 26
Number of test batches: 6
```

Configure the dataset for performance

Use buffered prefetching to load images from disk without having I/O become blocking. To learn more about this method see the data performance guide.

In [9]:
```
AUTOTUNE = tf.data.AUTOTUNE   # Constant that lets TensorFlow automatically choose an optimal prefetch buffer s


# Use prefetching on the datasets to improve performance.
# This overlaps data preprocessing and model execution while training.
train_dataset = train_dataset.prefetch(buffer_size=AUTOTUNE)

validation_dataset = validation_dataset.prefetch(buffer_size=AUTOTUNE)

test_dataset = test_dataset.prefetch(buffer_size=AUTOTUNE)
```

Use data augmentation

When you don't have a large image dataset, it's a good practice to artificially introduce sample diversity by applying random, yet realistic, transformations to the training images, such as rotation and horizontal flipping. This helps expose the model to different aspects of the training data and reduce overfitting. You can learn more about data augmentation in this tutorial.

In [10]:

```
# Define a data augmentation pipeline using Keras Sequential model

data_augmentation = tf.keras.Sequential([
  tf.keras.layers.RandomFlip('horizontal'),
  tf.keras.layers.RandomRotation(0.2),
])
```

Note: These layers are active only during training, when you call Model.fit. They are inactive when the model is used in inference mode in Model.evaluate, Model.predict, or Model.call.

Let's repeatedly apply these layers to the same image and see the result.

In [11]:

```
# Take one batch of images from the training set to demonstrate augmentation

for image, _ in train_dataset.take(1):
  plt.figure(figsize=(10, 10))      # New figure for the augmented images
  first_image = image[0]            # Extract the first image from this batch

  # Generate and plot 9 augmented versions of this image
  for i in range(9):
    ax = plt.subplot(3, 3, i + 1)   # Create a 3x3 grid of subplots
    augmented_image = data_augmentation(tf.expand_dims(first_image, 0))
    plt.imshow(augmented_image[0] / 255)
    plt.axis('off')
```

Rescale pixel values

In a moment, you will download `tf.keras.applications.MobileNetV2` for use as your base model. This model expects pixel values in `[-1, 1]`, but at this point, the pixel values in your images are in `[0, 255]`. To rescale them, use the preprocessing method included with the model.

In [12]:
```
preprocess_input = tf.keras.applications.mobilenet_v2.preprocess_input
```

Note: Alternatively, you could rescale pixel values from `[0, 255]` to `[-1, 1]` using `tf.keras.layers.Rescaling`.

In [13]:
```
rescale = tf.keras.layers.Rescaling(1./127.5, offset=-1)
```

Note: If using other `tf.keras.applications`, be sure to check the API doc to determine if they expect pixels in `[-1, 1]` or `[0, 1]`, or use the included `preprocess_input` function.

Create the base model from the pre-trained convnets

You will create the base model from the **MobileNet V2** model developed at Google. This is pre-trained on the ImageNet dataset, a large dataset consisting of 1.4M images and 1000 classes. ImageNet is a research training dataset with a wide variety of categories like `jackfruit` and `syringe`. This base of knowledge will help us classify cats and dogs from our specific dataset.

First, you need to pick which layer of MobileNet V2 you will use for feature extraction. The very last classification layer (on "top", as most diagrams of machine learning models go from bottom to top) is not very useful. Instead, you will follow the common practice to depend on the very last layer before the flatten operation. This layer is called the "bottleneck layer". The bottleneck layer features retain more generality as compared to the final/top layer.

First, instantiate a MobileNet V2 model pre-loaded with weights trained on ImageNet. By specifying the **include_top=False** argument, you load a network that doesn't include the classification layers at the top, which is ideal for feature extraction.

In [14]:
```
# Create the base model from the pre-trained model MobileNet V2
# Define the input shape expected by the base model: 160x160 pixels with 3 color channels (RGB)
IMG_SHAPE = IMG_SIZE + (3,)       # This gives (160, 160, 3)

# Load the MobileNetV2 model pre-trained on ImageNet.
base_model = tf.keras.applications.MobileNetV2(input_shape=IMG_SHAPE,
                                               include_top=False,    # Exclude the top classification layer of 
                                               weights='imagenet')   # Initialize with weights pre-trained on t
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v2/mobilenet_v2_wei
ghts_tf_dim_ordering_tf_kernels_1.0_160_no_top.h5
9406464/9406464 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

This feature extractor converts each `160x160x3` image into a `5x5x1280` block of features. Let's see what it does to an example batch of images:

In [15]:
```
# Get one batch of images from the training dataset
image_batch, label_batch = next(iter(train_dataset))
# Pass this batch through the base model to extract features
feature_batch = base_model(image_batch)
# Print the shape of the feature batch
print(feature_batch.shape)
```

```
(32, 5, 5, 1280)
```

Feature extraction

In this step, you will freeze the convolutional base created from the previous step and to use as a feature extractor. Additionally, you add a classifier on top of it and train the top-level classifier.

Freeze the convolutional base

It is important to freeze the convolutional base before you compile and train the model. Freezing (by setting layer.trainable = False) prevents the weights in a given layer from being updated during training. MobileNet V2 has many layers, so setting the entire model's `trainable` flag to False will freeze all of them.

In [16]:
```
base_model.trainable = False      # Freeze all the layers in the base MobileNetV2 model
```

Important note about BatchNormalization layers

Many models contain `tf.keras.layers.BatchNormalization` layers. This layer is a special case and precautions should be taken in the context of fine-tuning, as shown later in this tutorial.

When you set `layer.trainable = False`, the `BatchNormalization` layer will run in inference mode, and will not update its mean and variance statistics.

When you unfreeze a model that contains BatchNormalization layers in order to do fine-tuning, you should keep the BatchNormalization layers in inference mode by passing `training = False` when calling the base model. Otherwise, the updates applied to the non-trainable weights will destroy what the model has learned.

For more details, see the Transfer learning guide.

In [17]:
```
# Let's take a look at the base model architecture
base_model.summary()
```

Model: "mobilenetv2_1.00_160"

Layer (type)	Output Shape	Param #	Connected to
input_layer_1 (InputLayer)	(None, 160, 160, 3)	0	-
Conv1 (Conv2D)	(None, 80, 80, 32)	864	input_layer_1[0]…
bn_Conv1 (BatchNormalizatio…	(None, 80, 80, 32)	128	Conv1[0][0]
Conv1_relu (ReLU)	(None, 80, 80, 32)	0	bn_Conv1[0][0]
expanded_conv_dept… (DepthwiseConv2D)	(None, 80, 80, 32)	288	Conv1_relu[0][0]
expanded_conv_dept… (BatchNormalizatio…	(None, 80, 80, 32)	128	expanded_conv_de…
expanded_conv_dept… (ReLU)	(None, 80, 80, 32)	0	expanded_conv_de…
expanded_conv_proj… (Conv2D)	(None, 80, 80, 16)	512	expanded_conv_de…
expanded_conv_proj… (BatchNormalizatio…	(None, 80, 80, 16)	64	expanded_conv_pr…
block_1_expand (Conv2D)	(None, 80, 80, 96)	1,536	expanded_conv_pr…
block_1_expand_BN (BatchNormalizatio…	(None, 80, 80, 96)	384	block_1_expand[0…
block_1_expand_relu (ReLU)	(None, 80, 80, 96)	0	block_1_expand_B…
block_1_pad (ZeroPadding2D)	(None, 81, 81, 96)	0	block_1_expand_r…
block_1_depthwise (DepthwiseConv2D)	(None, 40, 40, 96)	864	block_1_pad[0][0]
block_1_depthwise_… (BatchNormalizatio…	(None, 40, 40, 96)	384	block_1_depthwis…
block_1_depthwise_… (ReLU)	(None, 40, 40, 96)	0	block_1_depthwis…
block_1_project (Conv2D)	(None, 40, 40, 24)	2,304	block_1_depthwis…
block_1_project_BN (BatchNormalizatio…	(None, 40, 40, 24)	96	block_1_project[…
block_2_expand (Conv2D)	(None, 40, 40, 144)	3,456	block_1_project_…
block_2_expand_BN (BatchNormalizatio…	(None, 40, 40, 144)	576	block_2_expand[0…
block_2_expand_relu (ReLU)	(None, 40, 40, 144)	0	block_2_expand_B…
block_2_depthwise (DepthwiseConv2D)	(None, 40, 40, 144)	1,296	block_2_expand_r…
block_2_depthwise_… (BatchNormalizatio…	(None, 40, 40, 144)	576	block_2_depthwis…

block_2_depthwise_… (ReLU)	(None, 40, 40, 144)	0	block_2_depthwis…
block_2_project (Conv2D)	(None, 40, 40, 24)	3,456	block_2_depthwis…
block_2_project_BN (BatchNormalizatio…	(None, 40, 40, 24)	96	block_2_project[…
block_2_add (Add)	(None, 40, 40, 24)	0	block_1_project_… block_2_project_…
block_3_expand (Conv2D)	(None, 40, 40, 144)	3,456	block_2_add[0][0]
block_3_expand_BN (BatchNormalizatio…	(None, 40, 40, 144)	576	block_3_expand[0…
block_3_expand_relu (ReLU)	(None, 40, 40, 144)	0	block_3_expand_B…
block_3_pad (ZeroPadding2D)	(None, 41, 41, 144)	0	block_3_expand_r…
block_3_depthwise (DepthwiseConv2D)	(None, 20, 20, 144)	1,296	block_3_pad[0][0]
block_3_depthwise_… (BatchNormalizatio…	(None, 20, 20, 144)	576	block_3_depthwis…
block_3_depthwise_… (ReLU)	(None, 20, 20, 144)	0	block_3_depthwis…
block_3_project (Conv2D)	(None, 20, 20, 32)	4,608	block_3_depthwis…
block_3_project_BN (BatchNormalizatio…	(None, 20, 20, 32)	128	block_3_project[…
block_4_expand (Conv2D)	(None, 20, 20, 192)	6,144	block_3_project_…
block_4_expand_BN (BatchNormalizatio…	(None, 20, 20, 192)	768	block_4_expand[0…
block_4_expand_relu (ReLU)	(None, 20, 20, 192)	0	block_4_expand_B…
block_4_depthwise (DepthwiseConv2D)	(None, 20, 20, 192)	1,728	block_4_expand_r…
block_4_depthwise_… (BatchNormalizatio…	(None, 20, 20, 192)	768	block_4_depthwis…
block_4_depthwise_… (ReLU)	(None, 20, 20, 192)	0	block_4_depthwis…
block_4_project (Conv2D)	(None, 20, 20, 32)	6,144	block_4_depthwis…
block_4_project_BN (BatchNormalizatio…	(None, 20, 20, 32)	128	block_4_project[…
block_4_add (Add)	(None, 20, 20, 32)	0	block_3_project_… block_4_project_…
block_5_expand (Conv2D)	(None, 20, 20, 192)	6,144	block_4_add[0][0]
block_5_expand_BN (BatchNormalizatio…	(None, 20, 20, 192)	768	block_5_expand[0…
block_5_expand_relu (ReLU)	(None, 20, 20, 192)	0	block_5_expand_B…
block_5_depthwise (DepthwiseConv2D)	(None, 20, 20, 192)	1,728	block_5_expand_r…
block_5_depthwise_… (BatchNormalizatio…	(None, 20, 20, 192)	768	block_5_depthwis…
block_5_depthwise_… (ReLU)	(None, 20, 20, 192)	0	block_5_depthwis…

block_5_project (Conv2D)	(None, 20, 20, 32)	6,144	block_5_depthwis…
block_5_project_BN (BatchNormalizatio…	(None, 20, 20, 32)	128	block_5_project[…
block_5_add (Add)	(None, 20, 20, 32)	0	block_4_add[0][0… block_5_project_…
block_6_expand (Conv2D)	(None, 20, 20, 192)	6,144	block_5_add[0][0]
block_6_expand_BN (BatchNormalizatio…	(None, 20, 20, 192)	768	block_6_expand[0…
block_6_expand_relu (ReLU)	(None, 20, 20, 192)	0	block_6_expand_B…
block_6_pad (ZeroPadding2D)	(None, 21, 21, 192)	0	block_6_expand_r…
block_6_depthwise (DepthwiseConv2D)	(None, 10, 10, 192)	1,728	block_6_pad[0][0]
block_6_depthwise_… (BatchNormalizatio…	(None, 10, 10, 192)	768	block_6_depthwis…
block_6_depthwise_… (ReLU)	(None, 10, 10, 192)	0	block_6_depthwis…
block_6_project (Conv2D)	(None, 10, 10, 64)	12,288	block_6_depthwis…
block_6_project_BN (BatchNormalizatio…	(None, 10, 10, 64)	256	block_6_project[…
block_7_expand (Conv2D)	(None, 10, 10, 384)	24,576	block_6_project_…
block_7_expand_BN (BatchNormalizatio…	(None, 10, 10, 384)	1,536	block_7_expand[0…
block_7_expand_relu (ReLU)	(None, 10, 10, 384)	0	block_7_expand_B…
block_7_depthwise (DepthwiseConv2D)	(None, 10, 10, 384)	3,456	block_7_expand_r…
block_7_depthwise_… (BatchNormalizatio…	(None, 10, 10, 384)	1,536	block_7_depthwis…
block_7_depthwise_… (ReLU)	(None, 10, 10, 384)	0	block_7_depthwis…
block_7_project (Conv2D)	(None, 10, 10, 64)	24,576	block_7_depthwis…
block_7_project_BN (BatchNormalizatio…	(None, 10, 10, 64)	256	block_7_project[…
block_7_add (Add)	(None, 10, 10, 64)	0	block_6_project_… block_7_project_…
block_8_expand (Conv2D)	(None, 10, 10, 384)	24,576	block_7_add[0][0]
block_8_expand_BN (BatchNormalizatio…	(None, 10, 10, 384)	1,536	block_8_expand[0…
block_8_expand_relu (ReLU)	(None, 10, 10, 384)	0	block_8_expand_B…
block_8_depthwise (DepthwiseConv2D)	(None, 10, 10, 384)	3,456	block_8_expand_r…
block_8_depthwise_… (BatchNormalizatio…	(None, 10, 10, 384)	1,536	block_8_depthwis…
block_8_depthwise_… (ReLU)	(None, 10, 10, 384)	0	block_8_depthwis…
block_8_project	(None, 10, 10,	24,576	block_8_depthwis…

(Conv2D)	64)		
block_8_project_BN (BatchNormalizatio…	(None, 10, 10, 64)	256	block_8_project[…
block_8_add (Add)	(None, 10, 10, 64)	0	block_7_add[0][0… block_8_project_…
block_9_expand (Conv2D)	(None, 10, 10, 384)	24,576	block_8_add[0][0]
block_9_expand_BN (BatchNormalizatio…	(None, 10, 10, 384)	1,536	block_9_expand[0…
block_9_expand_relu (ReLU)	(None, 10, 10, 384)	0	block_9_expand_B…
block_9_depthwise (DepthwiseConv2D)	(None, 10, 10, 384)	3,456	block_9_expand_r…
block_9_depthwise_… (BatchNormalizatio…	(None, 10, 10, 384)	1,536	block_9_depthwis…
block_9_depthwise_… (ReLU)	(None, 10, 10, 384)	0	block_9_depthwis…
block_9_project (Conv2D)	(None, 10, 10, 64)	24,576	block_9_depthwis…
block_9_project_BN (BatchNormalizatio…	(None, 10, 10, 64)	256	block_9_project[…
block_9_add (Add)	(None, 10, 10, 64)	0	block_8_add[0][0… block_9_project_…
block_10_expand (Conv2D)	(None, 10, 10, 384)	24,576	block_9_add[0][0]
block_10_expand_BN (BatchNormalizatio…	(None, 10, 10, 384)	1,536	block_10_expand[…
block_10_expand_re… (ReLU)	(None, 10, 10, 384)	0	block_10_expand_…
block_10_depthwise (DepthwiseConv2D)	(None, 10, 10, 384)	3,456	block_10_expand_…
block_10_depthwise… (BatchNormalizatio…	(None, 10, 10, 384)	1,536	block_10_depthwi…
block_10_depthwise… (ReLU)	(None, 10, 10, 384)	0	block_10_depthwi…
block_10_project (Conv2D)	(None, 10, 10, 96)	36,864	block_10_depthwi…
block_10_project_BN (BatchNormalizatio…	(None, 10, 10, 96)	384	block_10_project…
block_11_expand (Conv2D)	(None, 10, 10, 576)	55,296	block_10_project…
block_11_expand_BN (BatchNormalizatio…	(None, 10, 10, 576)	2,304	block_11_expand[…
block_11_expand_re… (ReLU)	(None, 10, 10, 576)	0	block_11_expand_…
block_11_depthwise (DepthwiseConv2D)	(None, 10, 10, 576)	5,184	block_11_expand_…
block_11_depthwise… (BatchNormalizatio…	(None, 10, 10, 576)	2,304	block_11_depthwi…
block_11_depthwise… (ReLU)	(None, 10, 10, 576)	0	block_11_depthwi…
block_11_project (Conv2D)	(None, 10, 10, 96)	55,296	block_11_depthwi…
block_11_project_BN (BatchNormalizatio…	(None, 10, 10, 96)	384	block_11_project…

block_11_add (Add)	(None, 10, 10, 96)	0	block_10_project… block_11_project…
block_12_expand (Conv2D)	(None, 10, 10, 576)	55,296	block_11_add[0][…
block_12_expand_BN (BatchNormalizatio…	(None, 10, 10, 576)	2,304	block_12_expand[…
block_12_expand_re… (ReLU)	(None, 10, 10, 576)	0	block_12_expand_…
block_12_depthwise (DepthwiseConv2D)	(None, 10, 10, 576)	5,184	block_12_expand_…
block_12_depthwise… (BatchNormalizatio…	(None, 10, 10, 576)	2,304	block_12_depthwi…
block_12_depthwise… (ReLU)	(None, 10, 10, 576)	0	block_12_depthwi…
block_12_project (Conv2D)	(None, 10, 10, 96)	55,296	block_12_depthwi…
block_12_project_BN (BatchNormalizatio…	(None, 10, 10, 96)	384	block_12_project…
block_12_add (Add)	(None, 10, 10, 96)	0	block_11_add[0][… block_12_project…
block_13_expand (Conv2D)	(None, 10, 10, 576)	55,296	block_12_add[0][…
block_13_expand_BN (BatchNormalizatio…	(None, 10, 10, 576)	2,304	block_13_expand[…
block_13_expand_re… (ReLU)	(None, 10, 10, 576)	0	block_13_expand_…
block_13_pad (ZeroPadding2D)	(None, 11, 11, 576)	0	block_13_expand_…
block_13_depthwise (DepthwiseConv2D)	(None, 5, 5, 576)	5,184	block_13_pad[0][…
block_13_depthwise… (BatchNormalizatio…	(None, 5, 5, 576)	2,304	block_13_depthwi…
block_13_depthwise… (ReLU)	(None, 5, 5, 576)	0	block_13_depthwi…
block_13_project (Conv2D)	(None, 5, 5, 160)	92,160	block_13_depthwi…
block_13_project_BN (BatchNormalizatio…	(None, 5, 5, 160)	640	block_13_project…
block_14_expand (Conv2D)	(None, 5, 5, 960)	153,600	block_13_project…
block_14_expand_BN (BatchNormalizatio…	(None, 5, 5, 960)	3,840	block_14_expand[…
block_14_expand_re… (ReLU)	(None, 5, 5, 960)	0	block_14_expand_…
block_14_depthwise (DepthwiseConv2D)	(None, 5, 5, 960)	8,640	block_14_expand_…
block_14_depthwise… (BatchNormalizatio…	(None, 5, 5, 960)	3,840	block_14_depthwi…
block_14_depthwise… (ReLU)	(None, 5, 5, 960)	0	block_14_depthwi…
block_14_project (Conv2D)	(None, 5, 5, 160)	153,600	block_14_depthwi…
block_14_project_BN (BatchNormalizatio…	(None, 5, 5, 160)	640	block_14_project…
block_14_add (Add)	(None, 5, 5, 160)	0	block_13_project… block_14_project…

block_15_expand (Conv2D)	(None, 5, 5, 960)	153,600	block_14_add[0][…
block_15_expand_BN (BatchNormalizatio…	(None, 5, 5, 960)	3,840	block_15_expand[…
block_15_expand_re… (ReLU)	(None, 5, 5, 960)	0	block_15_expand_…
block_15_depthwise (DepthwiseConv2D)	(None, 5, 5, 960)	8,640	block_15_expand_…
block_15_depthwise… (BatchNormalizatio…	(None, 5, 5, 960)	3,840	block_15_depthwi…
block_15_depthwise… (ReLU)	(None, 5, 5, 960)	0	block_15_depthwi…
block_15_project (Conv2D)	(None, 5, 5, 160)	153,600	block_15_depthwi…
block_15_project_BN (BatchNormalizatio…	(None, 5, 5, 160)	640	block_15_project…
block_15_add (Add)	(None, 5, 5, 160)	0	block_14_add[0][… block_15_project…
block_16_expand (Conv2D)	(None, 5, 5, 960)	153,600	block_15_add[0][…
block_16_expand_BN (BatchNormalizatio…	(None, 5, 5, 960)	3,840	block_16_expand[…
block_16_expand_re… (ReLU)	(None, 5, 5, 960)	0	block_16_expand_…
block_16_depthwise (DepthwiseConv2D)	(None, 5, 5, 960)	8,640	block_16_expand_…
block_16_depthwise… (BatchNormalizatio…	(None, 5, 5, 960)	3,840	block_16_depthwi…
block_16_depthwise… (ReLU)	(None, 5, 5, 960)	0	block_16_depthwi…
block_16_project (Conv2D)	(None, 5, 5, 320)	307,200	block_16_depthwi…
block_16_project_BN (BatchNormalizatio…	(None, 5, 5, 320)	1,280	block_16_project…
Conv_1 (Conv2D)	(None, 5, 5, 1280)	409,600	block_16_project…
Conv_1_bn (BatchNormalizatio…	(None, 5, 5, 1280)	5,120	Conv_1[0][0]
out_relu (ReLU)	(None, 5, 5, 1280)	0	Conv_1_bn[0][0]

Total params: 2,257,984 (8.61 MB)
Trainable params: 0 (0.00 B)
Non-trainable params: 2,257,984 (8.61 MB)

Add a classification head

To generate predictions from the block of features, average over the spatial `5x5` spatial locations, using a `tf.keras.layers.GlobalAveragePooling2D` layer to convert the features to a single 1280-element vector per image.

In [18]:
```
global_average_layer = tf.keras.layers.GlobalAveragePooling2D()  # Global Average Pooling layer will average th
feature_batch_average = global_average_layer(feature_batch)      # Apply the global average pooling to the feat
print(feature_batch_average.shape)
```

```
(32, 1280)
```

Apply a `tf.keras.layers.Dense` layer to convert these features into a single prediction per image. You don't need an activation function here because this prediction will be treated as a `logit`, or a raw prediction value. Positive numbers predict class 1, negative numbers predict class 0.

In [19]:
```python
# Define a Dense layer that will produce a single output (logit) for each image.
prediction_layer = tf.keras.layers.Dense(1)
# Apply the Dense layer to the averaged features to get predictions for the batch.
prediction_batch = prediction_layer(feature_batch_average)
print(prediction_batch.shape)
```

```
(32, 1)
```

Build a model by chaining together the data augmentation, rescaling, `base_model` and feature extractor layers using the Keras Functional API. As previously mentioned, use `training=False` as our model contains a `BatchNormalization` layer.

In [20]:
```python
inputs = tf.keras.Input(shape=(160, 160, 3))   # Define the input layer of the model, expecting images of shape
x = data_augmentation(inputs)
x = preprocess_input(x)
x = base_model(x, training=False)   # We set training=False to ensure the base model runs in inference mode (im
x = global_average_layer(x)
x = tf.keras.layers.Dropout(0.2)(x) # Add a dropout layer for regularization. 0.2 means 20% of elements will be
outputs = prediction_layer(x)
model = tf.keras.Model(inputs, outputs)
```

In [21]:
```python
model.summary()
```

Model: "functional_1"

Layer (type)	Output Shape	Param #
input_layer_2 (InputLayer)	(None, 160, 160, 3)	0
sequential (Sequential)	(None, 160, 160, 3)	0
true_divide (TrueDivide)	(None, 160, 160, 3)	0
subtract (Subtract)	(None, 160, 160, 3)	0
mobilenetv2_1.00_160 (Functional)	(None, 5, 5, 1280)	2,257,984
global_average_pooling2d (GlobalAveragePooling2D)	(None, 1280)	0
dropout (Dropout)	(None, 1280)	0
dense (Dense)	(None, 1)	1,281

Total params: 2,259,265 (8.62 MB)
Trainable params: 1,281 (5.00 KB)
Non-trainable params: 2,257,984 (8.61 MB)

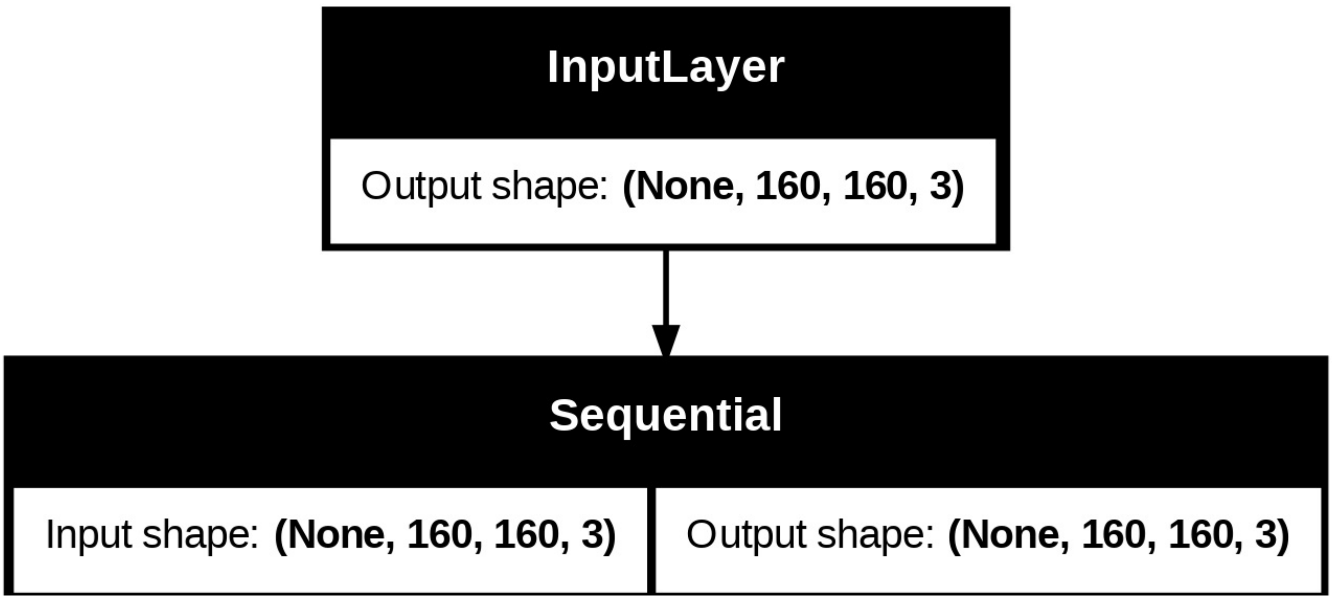
The 8+ million parameters in MobileNet are frozen, but there are 1.2 thousand *trainable* parameters in the Dense layer. These are divided between two `tf.Variable` objects, the weights and biases.

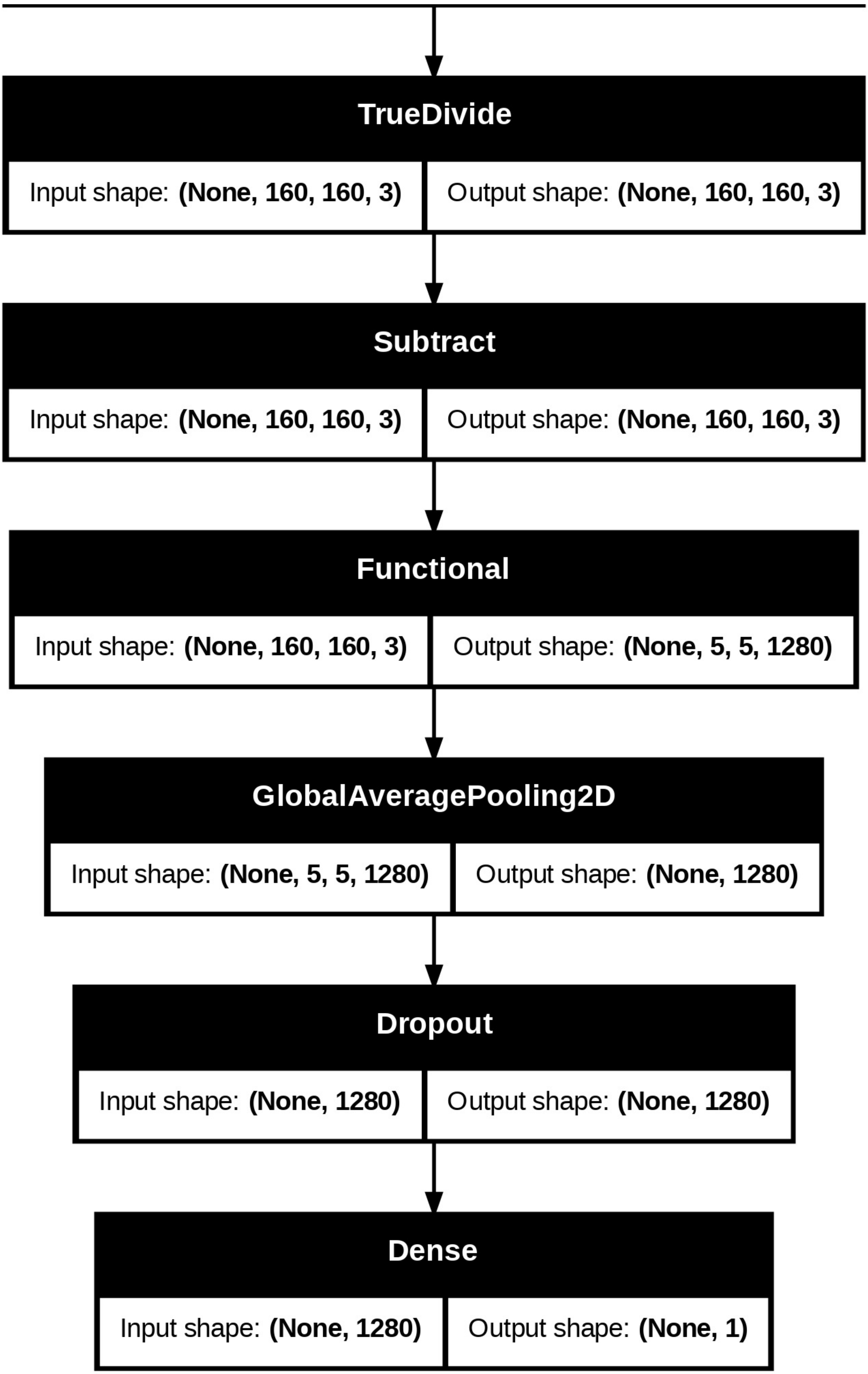
In [22]:
```python
# Print the number of trainable variables in the model.
len(model.trainable_variables)
```

Out[22]: 2

In [23]:
```python
tf.keras.utils.plot_model(model, show_shapes=True)
```

Out[23]:





Compile the model

Compile the model before training it. Since there are two classes, use the `tf.keras.losses.BinaryCrossentropy` loss with `from_logits=True` since the model provides a linear output.

In [24]:
```
base_learning_rate = 0.0001

# Compile the model with an optimizer, loss function, and evaluation metric.
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=base_learning_rate),
              loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=[tf.keras.metrics.BinaryAccuracy(threshold=0, name='accuracy')])   # Measure accuracy; lo
```

Train the model

After training for 10 epochs, you should see ~96% accuracy on the validation set.

In [28]:
```
initial_epochs = 30    # We plan to train for 30 epochs initially (feature extraction phase)

# Evaluate the model on the validation set before training (this gives a baseline performance).
loss0, accuracy0 = model.evaluate(validation_dataset)
```

```
26/26 ━━━━━━━━━━━━━━━━━━━━ 1s 32ms/step - accuracy: 0.3903 - loss: 0.9172
```

In [29]:
```
print("initial loss: {:.2f}".format(loss0))
print("initial accuracy: {:.2f}".format(accuracy0))
```

```
initial loss: 0.92
initial accuracy: 0.39
```

In [30]:
```
# Train the model on the training set for `initial_epochs` (30) epochs,
# validating on the validation dataset at the end of each epoch.

history = model.fit(train_dataset,
                    epochs=initial_epochs,
                    validation_data=validation_dataset)
```

```
Epoch 1/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 10s 58ms/step - accuracy: 0.4443 - loss: 0.9122 - val_accuracy: 0.6386 - val_loss: 0.
6340
Epoch 2/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 3s 47ms/step - accuracy: 0.6264 - loss: 0.6668 - val_accuracy: 0.7995 - val_loss: 0.4
682
Epoch 3/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 3s 46ms/step - accuracy: 0.7453 - loss: 0.5142 - val_accuracy: 0.8911 - val_loss: 0.3
457
Epoch 4/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 7s 70ms/step - accuracy: 0.8200 - loss: 0.4202 - val_accuracy: 0.9245 - val_loss: 0.2
783
Epoch 5/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 3s 48ms/step - accuracy: 0.8435 - loss: 0.3562 - val_accuracy: 0.9406 - val_loss: 0.2
451
Epoch 6/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 7s 80ms/step - accuracy: 0.9121 - loss: 0.2813 - val_accuracy: 0.9455 - val_loss: 0.2
156
Epoch 7/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 3s 47ms/step - accuracy: 0.8914 - loss: 0.3010 - val_accuracy: 0.9567 - val_loss: 0.1
871
Epoch 8/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 3s 47ms/step - accuracy: 0.9043 - loss: 0.2586 - val_accuracy: 0.9604 - val_loss: 0.1
713
Epoch 9/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 3s 47ms/step - accuracy: 0.9034 - loss: 0.2518 - val_accuracy: 0.9616 - val_loss: 0.1
549
Epoch 10/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 5s 53ms/step - accuracy: 0.9061 - loss: 0.2356 - val_accuracy: 0.9604 - val_loss: 0.1
480
Epoch 11/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 3s 47ms/step - accuracy: 0.9225 - loss: 0.2118 - val_accuracy: 0.9641 - val_loss: 0.1
379
Epoch 12/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 6s 65ms/step - accuracy: 0.9157 - loss: 0.2262 - val_accuracy: 0.9678 - val_loss: 0.1
270
Epoch 13/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 3s 47ms/step - accuracy: 0.9221 - loss: 0.1995 - val_accuracy: 0.9666 - val_loss: 0.1
188
Epoch 14/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 5s 48ms/step - accuracy: 0.9206 - loss: 0.1953 - val_accuracy: 0.9678 - val_loss: 0.1
141
Epoch 15/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 4s 64ms/step - accuracy: 0.9304 - loss: 0.1844 - val_accuracy: 0.9740 - val_loss: 0.1
049
Epoch 16/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 4s 47ms/step - accuracy: 0.9355 - loss: 0.1721 - val_accuracy: 0.9740 - val_loss: 0.1
030
```

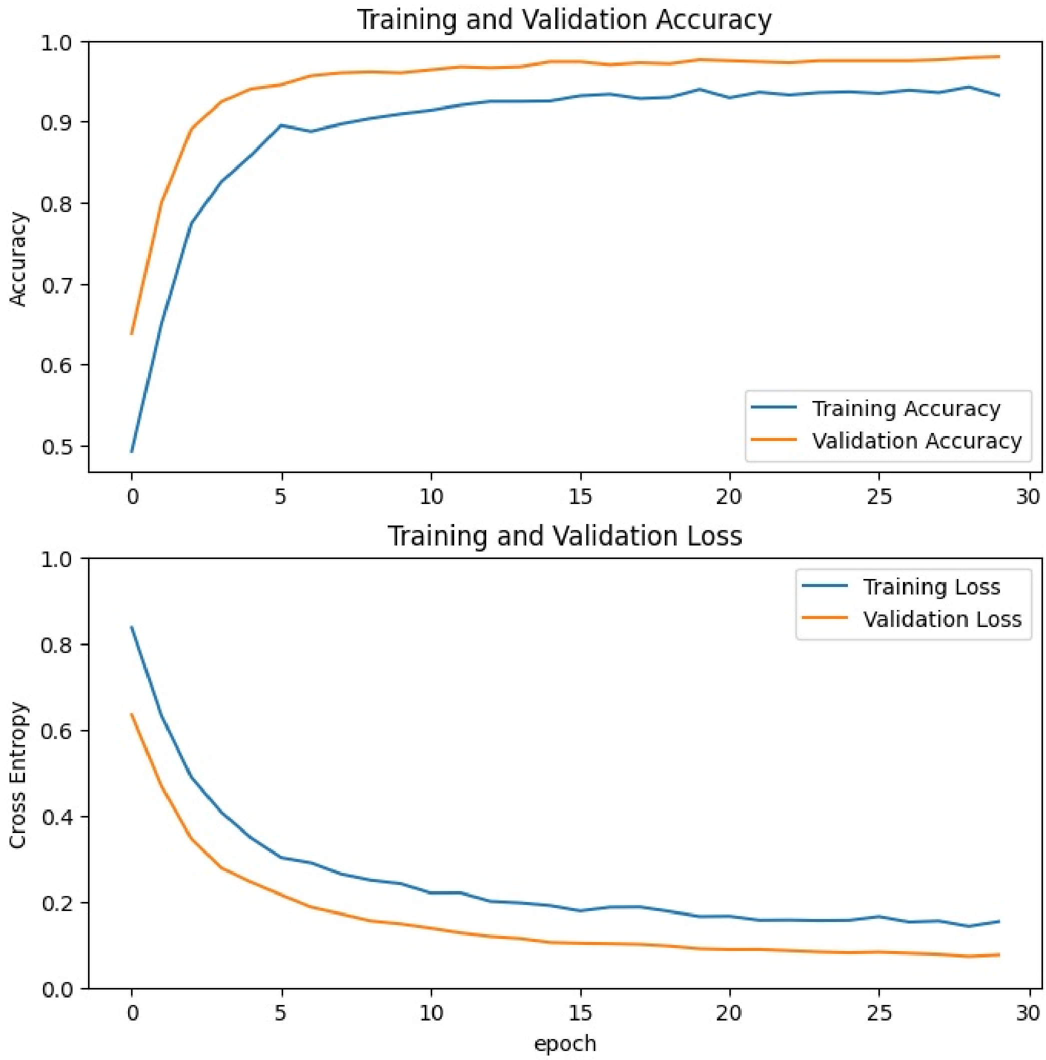
```
Epoch 17/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 5s 48ms/step - accuracy: 0.9352 - loss: 0.1797 - val_accuracy: 0.9703 - val_loss: 0.1
020
Epoch 18/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 5s 47ms/step - accuracy: 0.9353 - loss: 0.1791 - val_accuracy: 0.9728 - val_loss: 0.1
005
Epoch 19/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 5s 48ms/step - accuracy: 0.9346 - loss: 0.1708 - val_accuracy: 0.9715 - val_loss: 0.0
964
Epoch 20/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 6s 54ms/step - accuracy: 0.9459 - loss: 0.1618 - val_accuracy: 0.9765 - val_loss: 0.0
901
Epoch 21/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 3s 47ms/step - accuracy: 0.9216 - loss: 0.1700 - val_accuracy: 0.9752 - val_loss: 0.0
884
Epoch 22/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 7s 74ms/step - accuracy: 0.9406 - loss: 0.1467 - val_accuracy: 0.9740 - val_loss: 0.0
886
Epoch 23/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 3s 47ms/step - accuracy: 0.9399 - loss: 0.1480 - val_accuracy: 0.9728 - val_loss: 0.0
857
Epoch 24/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 5s 48ms/step - accuracy: 0.9343 - loss: 0.1586 - val_accuracy: 0.9752 - val_loss: 0.0
828
Epoch 25/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 5s 50ms/step - accuracy: 0.9397 - loss: 0.1492 - val_accuracy: 0.9752 - val_loss: 0.0
810
Epoch 26/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 5s 49ms/step - accuracy: 0.9446 - loss: 0.1536 - val_accuracy: 0.9752 - val_loss: 0.0
824
Epoch 27/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 6s 64ms/step - accuracy: 0.9392 - loss: 0.1527 - val_accuracy: 0.9752 - val_loss: 0.0
799
Epoch 28/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 3s 47ms/step - accuracy: 0.9354 - loss: 0.1531 - val_accuracy: 0.9765 - val_loss: 0.0
772
Epoch 29/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 5s 47ms/step - accuracy: 0.9401 - loss: 0.1391 - val_accuracy: 0.9790 - val_loss: 0.0
725
Epoch 30/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 5s 48ms/step - accuracy: 0.9348 - loss: 0.1470 - val_accuracy: 0.9802 - val_loss: 0.0
759
```

Learning curves

Let's take a look at the learning curves of the training and validation accuracy/loss when using the MobileNetV2 base model as a fixed feature extractor.

In [31]:

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

plt.figure(figsize=(8, 8))
plt.subplot(2, 1, 1)
plt.plot(acc, label='Training Accuracy')
plt.plot(val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.ylabel('Accuracy')
plt.ylim([min(plt.ylim()),1])
plt.title('Training and Validation Accuracy')

plt.subplot(2, 1, 2)
plt.plot(loss, label='Training Loss')
plt.plot(val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.ylabel('Cross Entropy')
plt.ylim([0,1.0])
plt.title('Training and Validation Loss')
plt.xlabel('epoch')
plt.show()
```



Note: If you are wondering why the validation metrics are clearly better than the training metrics, the main factor is because layers like `tf.keras.layers.BatchNormalization` and `tf.keras.layers.Dropout` affect accuracy during training. They are turned off when calculating validation loss.

To a lesser extent, it is also because training metrics report the average for an epoch, while validation metrics are evaluated after the epoch, so validation metrics see a model that has trained slightly longer.

Fine tuning

In the feature extraction experiment, you were only training a few layers on top of an MobileNetV2 base model. The weights of the pre-trained network were **not** updated during training.

One way to increase performance even further is to train (or "fine-tune") the weights of the top layers of the pre-trained model alongside the training of the classifier you added. The training process will force the weights to be tuned from generic feature maps to features associated specifically with the dataset.

Note: This should only be attempted after you have trained the top-level classifier with the pre-trained model set to non-trainable. If you add a randomly initialized classifier on top of a pre-trained model and attempt to train all layers jointly, the magnitude of the gradient updates will be too large (due to the random weights from the classifier) and your pre-trained model will forget what it has learned.

Also, you should try to fine-tune a small number of top layers rather than the whole MobileNet model. In most convolutional networks, the higher up a layer is, the more specialized it is. The first few layers learn very simple and generic features that generalize to almost all types of images. As you go higher up, the features are increasingly more specific to the dataset on which the model was trained. The goal of fine-tuning is to adapt these specialized features to work with the new dataset, rather than overwrite the generic learning.

Un-freeze the top layers of the model

All you need to do is unfreeze the `base_model` and set the bottom layers to be un-trainable. Then, you should recompile the model (necessary for these changes to take effect), and resume training.

In [32]:
```
# Unfreeze the base model to allow training on some layers of MobileNetV2
base_model.trainable = True
```

In [33]:
```
# Let's take a look to see how many layers are in the base model
```

```python
print("Number of layers in the base model: ", len(base_model.layers))
# MobileNetV2 has 154 layers in this configuration. We will choose how many to unfreeze.


fine_tune_at = 100    # We will unfreeze from layer 100 to the end (fine-tuning the top 54 layers or so)

# Freeze all layers below the fine_tune_at layer index, keep others trainable.
for layer in base_model.layers[:fine_tune_at]:
  layer.trainable = False
```

```
Number of layers in the base model:  154
```

Compile the model

As you are training a much larger model and want to readapt the pretrained weights, it is important to use a lower learning rate at this stage. Otherwise, your model could overfit very quickly.

In [34]:
```python
# It's important to use a very low learning rate for fine-tuning, to avoid large weight updates.
# We use RMSprop optimizer here with a learning rate 10 times smaller than the base learning rate.

model.compile(loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              optimizer = tf.keras.optimizers.RMSprop(learning_rate=base_learning_rate/10),
              metrics=[tf.keras.metrics.BinaryAccuracy(threshold=0, name='accuracy')])
```

In [35]:
```python
model.summary()
```

Model: "functional_1"

Layer (type)	Output Shape	Param #
input_layer_2 (InputLayer)	(None, 160, 160, 3)	0
sequential (Sequential)	(None, 160, 160, 3)	0
true_divide (TrueDivide)	(None, 160, 160, 3)	0
subtract (Subtract)	(None, 160, 160, 3)	0
mobilenetv2_1.00_160 (Functional)	(None, 5, 5, 1280)	2,257,984
global_average_pooling2d (GlobalAveragePooling2D)	(None, 1280)	0
dropout (Dropout)	(None, 1280)	0
dense (Dense)	(None, 1)	1,281

Total params: 2,259,265 (8.62 MB)
Trainable params: 1,862,721 (7.11 MB)
Non-trainable params: 396,544 (1.51 MB)

In [36]:
```python
len(model.trainable_variables)
```

Out[36]: 56

Continue training the model

If you trained to convergence earlier, this step will improve your accuracy by a few percentage points.

In [37]:
```python
fine_tune_epochs = 20
total_epochs =  initial_epochs + fine_tune_epochs

# Continue training the model. We start from the previous final epoch.
history_fine = model.fit(train_dataset,
                         epochs=total_epochs,             # Train up to the total number of epochs (inclu
                         initial_epoch=history.epoch[-1],       # Begin at the end of previous training (epoch .
                         validation_data=validation_dataset)
```

```
Epoch 30/50
63/63 ━━━━━━━━━━━━━━━━━━━━ 17s 100ms/step - accuracy: 0.7776 - loss: 0.4470 - val_accuracy: 0.9740 - val_loss: 0
.0719
Epoch 31/50
63/63 ━━━━━━━━━━━━━━━━━━━━ 7s 67ms/step - accuracy: 0.8715 - loss: 0.3121 - val_accuracy: 0.9691 - val_loss: 0.0
775
Epoch 32/50
63/63 ━━━━━━━━━━━━━━━━━━━━ 5s 77ms/step - accuracy: 0.9236 - loss: 0.1976 - val_accuracy: 0.9715 - val_loss: 0.0
780
Epoch 33/50
63/63 ━━━━━━━━━━━━━━━━━━━━ 4s 67ms/step - accuracy: 0.9184 - loss: 0.1839 - val_accuracy: 0.9802 - val_loss: 0.0
656
Epoch 34/50
63/63 ━━━━━━━━━━━━━━━━━━━━ 5s 72ms/step - accuracy: 0.9542 - loss: 0.1422 - val_accuracy: 0.9827 - val_loss: 0.0
557
Epoch 35/50
63/63 ━━━━━━━━━━━━━━━━━━━━ 5s 68ms/step - accuracy: 0.9320 - loss: 0.1623 - val_accuracy: 0.9839 - val_loss: 0.0
592
Epoch 36/50
63/63 ━━━━━━━━━━━━━━━━━━━━ 4s 65ms/step - accuracy: 0.9504 - loss: 0.1257 - val_accuracy: 0.9814 - val_loss: 0.0
588
Epoch 37/50
63/63 ━━━━━━━━━━━━━━━━━━━━ 5s 78ms/step - accuracy: 0.9465 - loss: 0.1314 - val_accuracy: 0.9851 - val_loss: 0.0
537
Epoch 38/50
63/63 ━━━━━━━━━━━━━━━━━━━━ 5s 72ms/step - accuracy: 0.9528 - loss: 0.1125 - val_accuracy: 0.9851 - val_loss: 0.0
497
Epoch 39/50
63/63 ━━━━━━━━━━━━━━━━━━━━ 4s 65ms/step - accuracy: 0.9457 - loss: 0.1339 - val_accuracy: 0.9827 - val_loss: 0.0
448
Epoch 40/50
63/63 ━━━━━━━━━━━━━━━━━━━━ 5s 77ms/step - accuracy: 0.9595 - loss: 0.1120 - val_accuracy: 0.9864 - val_loss: 0.0
435
Epoch 41/50
63/63 ━━━━━━━━━━━━━━━━━━━━ 4s 65ms/step - accuracy: 0.9644 - loss: 0.0964 - val_accuracy: 0.9864 - val_loss: 0.0
417
Epoch 42/50
63/63 ━━━━━━━━━━━━━━━━━━━━ 4s 71ms/step - accuracy: 0.9651 - loss: 0.0817 - val_accuracy: 0.9790 - val_loss: 0.0
486
Epoch 43/50
63/63 ━━━━━━━━━━━━━━━━━━━━ 5s 75ms/step - accuracy: 0.9585 - loss: 0.1011 - val_accuracy: 0.9839 - val_loss: 0.0
447
Epoch 44/50
63/63 ━━━━━━━━━━━━━━━━━━━━ 4s 65ms/step - accuracy: 0.9625 - loss: 0.0966 - val_accuracy: 0.9839 - val_loss: 0.0
447
Epoch 45/50
63/63 ━━━━━━━━━━━━━━━━━━━━ 6s 77ms/step - accuracy: 0.9635 - loss: 0.1011 - val_accuracy: 0.9839 - val_loss: 0.0
509
Epoch 46/50
63/63 ━━━━━━━━━━━━━━━━━━━━ 4s 65ms/step - accuracy: 0.9691 - loss: 0.0787 - val_accuracy: 0.9851 - val_loss: 0.0
462
Epoch 47/50
63/63 ━━━━━━━━━━━━━━━━━━━━ 5s 71ms/step - accuracy: 0.9682 - loss: 0.0766 - val_accuracy: 0.9876 - val_loss: 0.0
392
Epoch 48/50
63/63 ━━━━━━━━━━━━━━━━━━━━ 5s 69ms/step - accuracy: 0.9753 - loss: 0.0740 - val_accuracy: 0.9864 - val_loss: 0.0
408
Epoch 49/50
63/63 ━━━━━━━━━━━━━━━━━━━━ 4s 64ms/step - accuracy: 0.9747 - loss: 0.0716 - val_accuracy: 0.9864 - val_loss: 0.0
454
Epoch 50/50
63/63 ━━━━━━━━━━━━━━━━━━━━ 6s 78ms/step - accuracy: 0.9792 - loss: 0.0712 - val_accuracy: 0.9864 - val_loss: 0.0
381
```

Let's take a look at the learning curves of the training and validation accuracy/loss when fine-tuning the last few layers of the MobileNetV2 base model and training the classifier on top of it. The validation loss is much higher than the training loss, so you may get some overfitting.

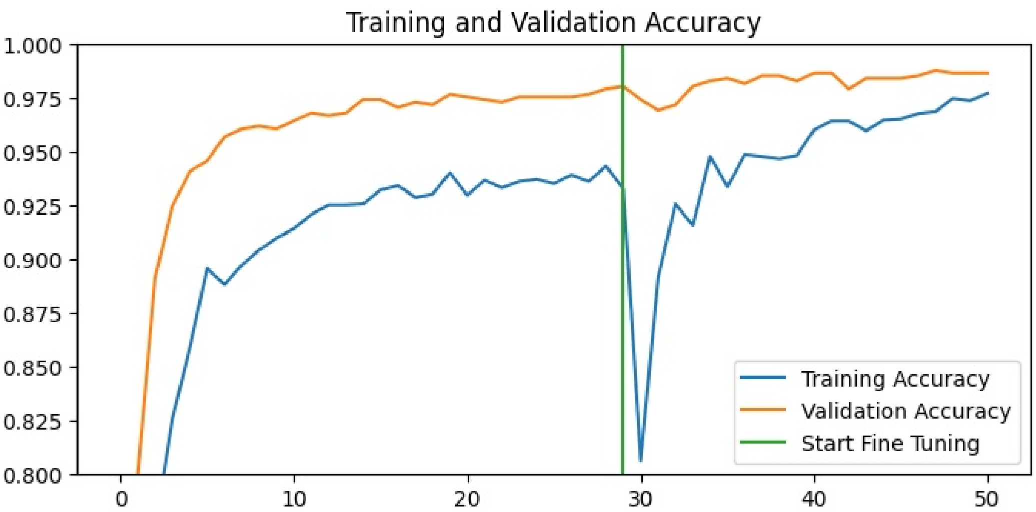
You may also get some overfitting as the new training set is relatively small and similar to the original MobileNetV2 datasets.

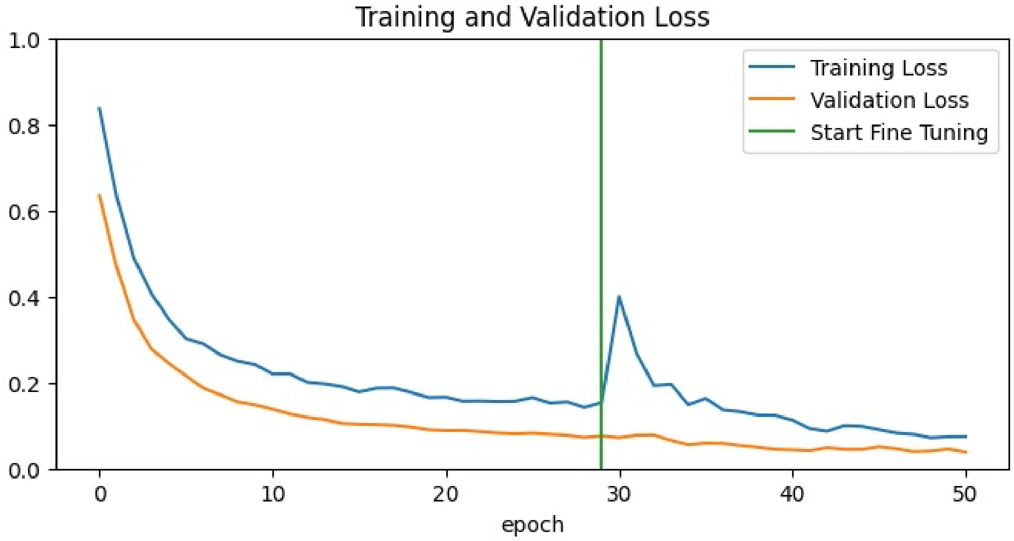
After fine tuning the model nearly reaches 98% accuracy on the validation set.

In [38]:
```
# Extend the original accuracy and loss lists with the fine-tuning results
acc += history_fine.history['accuracy']
val_acc += history_fine.history['val_accuracy']

loss += history_fine.history['loss']
val_loss += history_fine.history['val_loss']
```

In [39]:
```
plt.figure(figsize=(8, 8))
plt.subplot(2, 1, 1)
plt.plot(acc, label='Training Accuracy')
```

```
plt.plot(val_acc, label='Validation Accuracy')
plt.ylim([0.8, 1])
plt.plot([initial_epochs-1,initial_epochs-1],
          plt.ylim(), label='Start Fine Tuning')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(2, 1, 2)
plt.plot(loss, label='Training Loss')
plt.plot(val_loss, label='Validation Loss')
plt.ylim([0, 1.0])
plt.plot([initial_epochs-1,initial_epochs-1],
         plt.ylim(), label='Start Fine Tuning')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.xlabel('epoch')
plt.show()
```





Evaluation and prediction

Finally you can verify the performance of the model on new data using test set.

In [40]:
```
loss, accuracy = model.evaluate(test_dataset)
print('Test accuracy :', accuracy)
```

```
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 49ms/step - accuracy: 0.9950 - loss: 0.0199
Test accuracy : 0.9947916865348816
```

And now you are all set to use this model to predict if your pet is a cat or dog.

In [41]:
```
# Get one batch of test images and labels
image_batch, label_batch = test_dataset.as_numpy_iterator().next()
predictions = model.predict_on_batch(image_batch).flatten()

# Apply a sigmoid since our model returns logits
predictions = tf.nn.sigmoid(predictions)
predictions = tf.where(predictions < 0.5, 0, 1)  # Convert probabilities to class labels: if prob < 0.5 => clas

print('Predictions:\n', predictions.numpy())
print('Labels:\n', label_batch)
```

```
plt.figure(figsize=(10, 10))
for i in range(9):
  ax = plt.subplot(3, 3, i + 1)
  plt.imshow(image_batch[i].astype("uint8"))
  plt.title(class_names[predictions[i]])
  plt.axis("off")
```

```
Predictions:
 [1 0 0 0 0 0 0 1 1 1 0 0 1 0 1 0 1 1 1 0 0 1 0 1 0 0 0 0 1 1 0 1]
Labels:
 [1 0 0 0 0 0 0 1 1 1 0 0 1 0 1 0 1 1 1 0 0 1 0 1 0 0 0 0 1 1 0 1]
```

Summary

- **Using a pre-trained model for feature extraction**: When working with a small dataset, it is a common practice to take advantage of features learned by a model trained on a larger dataset in the same domain. This is done by instantiating the pre-trained model and adding a fully-connected classifier on top. The pre-trained model is "frozen" and only the weights of the classifier get updated during training. In this case, the convolutional base extracted all the features associated with each image and you just trained a classifier that determines the image class given that set of extracted features.

- **Fine-tuning a pre-trained model**: To further improve performance, one might want to repurpose the top-level layers of the pre-trained models to the new dataset via fine-tuning. In this case, you tuned your weights such that your model learned high-level features specific to the dataset. This technique is usually recommended when the training dataset is large and very similar to the original dataset that the pre-trained model was trained on.

To learn more, visit the Transfer learning guide.