< Deep Learning - PART2 TF2 CNNs >

Ch 5. CNNs Workshop 6 - Transfer learning with Pretrained CNN

2021/10/01

[Reference] :

1. TensorFlow Core - Tutorials: **Transfer learning with a pretrained ConvNet** https://www.tensorflow.org/tutorials/images/transfer_learning?hl=zh_tw (https://www.tensorflow.org/tutorials/images/transfer_learning?hl=zh_tw)
2. Keras Documentation - **Keras Applications** https://keras.io/applications/ (https://keras.io/applications/)
3. Andrew Ng - Coursera : **Transfer learning** https://zh-tw.coursera.org/lecture/machine-learning-projects/transfer-learning-WNPap (https://zh-tw.coursera.org/lecture/machine-learning-projects/transfer-learning-WNPap)

< NOTE > : *Run the following code on `Google Colab` with GPU!*

In this tutorial, you will learn how to classify images of cats and dogs by using **transfer learning** from a pre-trained network.

- A **pre-trained model** is a saved network that was previously trained on a large dataset, typically on a large-scale image-classification task. You either use the pretrained model as is or use transfer learning to customize this model to a given task.

- The intuition behind transfer learning for image classification is that if a model is trained on a large and general enough dataset, this model will effectively serve as a generic model of the visual world. You can then take advantage of these learned feature maps without having to start from scratch by training a large model on a large dataset.

In this notebook, you will try two ways to customize a pretrained model:

1. **Feature Extraction**: Use the representations learned by a previous network to extract meaningful features from new samples. You simply add a new classifier, which will be trained from scratch, on top of the pretrained model so that you can repurpose the feature maps learned previously for the dataset.

   You do not need to (re)train the entire model. The base convolutional network already contains features that are generically useful for classifying pictures. However, the final, classification part of the pretrained model is specific to the original classification task, and subsequently specific to the set of classes on which the model was trained.
2. **Fine-Tuning**: Unfreeze a few of the top layers of a frozen model base and jointly train both the newly-added classifier layers and the last layers of the base model. This allows us to "fine-tune" the higher-order feature representations in the base model in order to make them more relevant for the specific task.

You will follow the general machine learning workflow.

1. Examine and understand the data
2. Build an input pipeline, in this case using Keras ImageDataGenerator
3. Compose the model
   - Load in the pretrained base model (and pretrained weights)
   - Stack the classification layers on top
4. Train the model
5. Evaluate model

In [1]:

```
import matplotlib.pyplot as plt
import numpy as np
import os
import tensorflow as tf

from tensorflow.keras.preprocessing import image_dataset_from_directory
```

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.preprocessing.image_dataset_from_directory` utility. You can learn more about loading images in this tutorial (https://www.tensorflow.org/tutorials/load_data/images).

In [2]:

```
_URL = 'https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtere
path_to_zip = tf.keras.utils.get_file('cats_and_dogs.zip', origin=_URL, ext
PATH = os.path.join(os.path.dirname(path_to_zip), '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)

train_dataset = image_dataset_from_directory(train_dir,
                                             shuffle=True,
                                             batch_size=BATCH_SIZE,
                                             image_size=IMG_SIZE)
```

```
Downloading data from https://storage.googleapis.com/mledu-datase
ts/cats_and_dogs_filtered.zip (https://storage.googleapis.com/mle
du-datasets/cats_and_dogs_filtered.zip)
68608000/68606236 [==============================] - 1s 0us/step
68616192/68606236 [==============================] - 1s 0us/step
Found 2000 files belonging to 2 classes.
```

In [3]:

```
validation_dataset = image_dataset_from_directory(validation_dir,
                                                  shuffle=True,
                                                  batch_size=BATCH_SIZE,
                                                  image_size=IMG_SIZE)
```

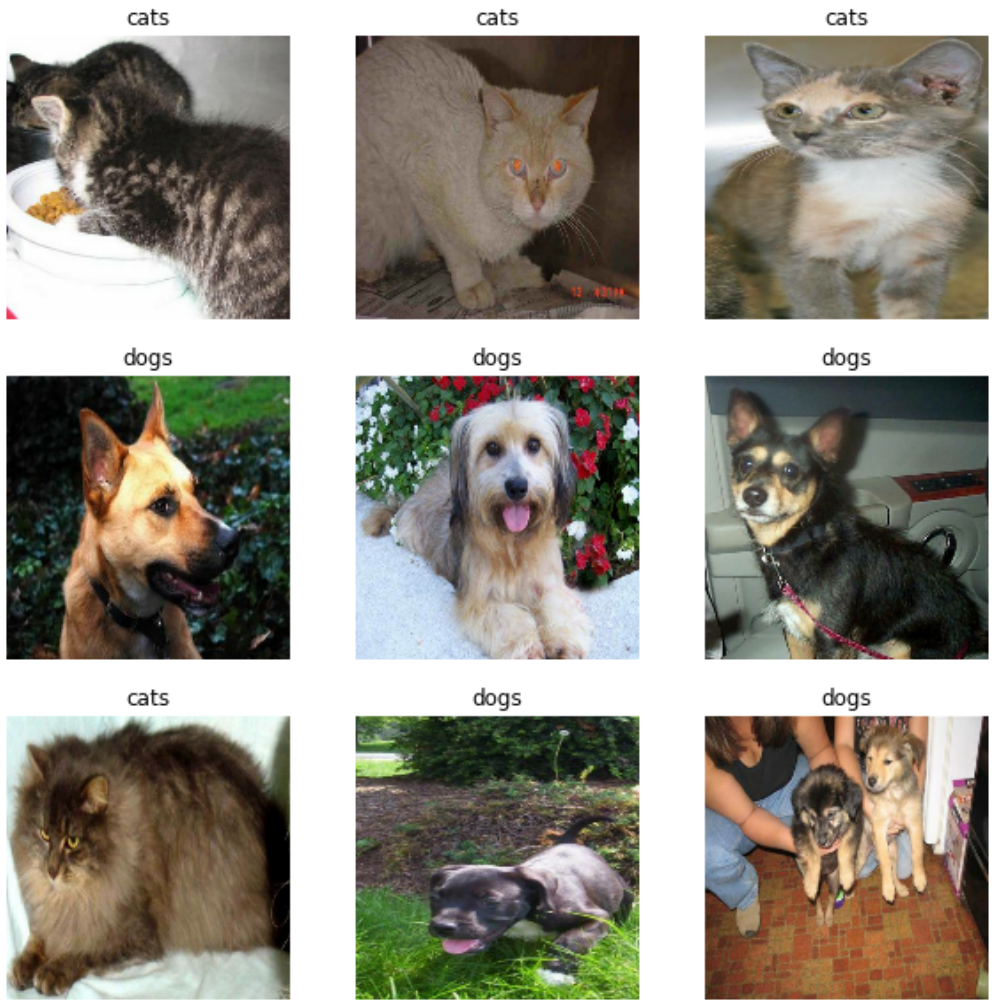
```
Found 1000 files belonging to 2 classes.
```

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

In [4]:

```
class_names = train_dataset.class_names

plt.figure(figsize=(10, 10))
for images, labels in train_dataset.take(1):
  for i in range(9):
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(images[i].numpy().astype("uint8"))
    plt.title(class_names[labels[i]])
    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 [5]:

```
val_batches = tf.data.experimental.cardinality(validation_dataset)
test_dataset = validation_dataset.take(val_batches // 5)
validation_dataset = validation_dataset.skip(val_batches // 5)
```

In [6]:

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

```
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 (https://www.tensorflow.org/guide/data_performance)](https://www.tensorflow.org/guide/data_performance) guide.

In [7]:

```
AUTOTUNE = tf.data.AUTOTUNE

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 (https://www.tensorflow.org/tutorials/keras/overfit_and_underfit)](https://www.tensorflow.org/tutorials/keras/overfit_and_underfit). You can learn more about data augmentation in this [tutorial (https://www.tensorflow.org/tutorials/images/data_augmentation)](https://www.tensorflow.org/tutorials/images/data_augmentation).

In [8]:

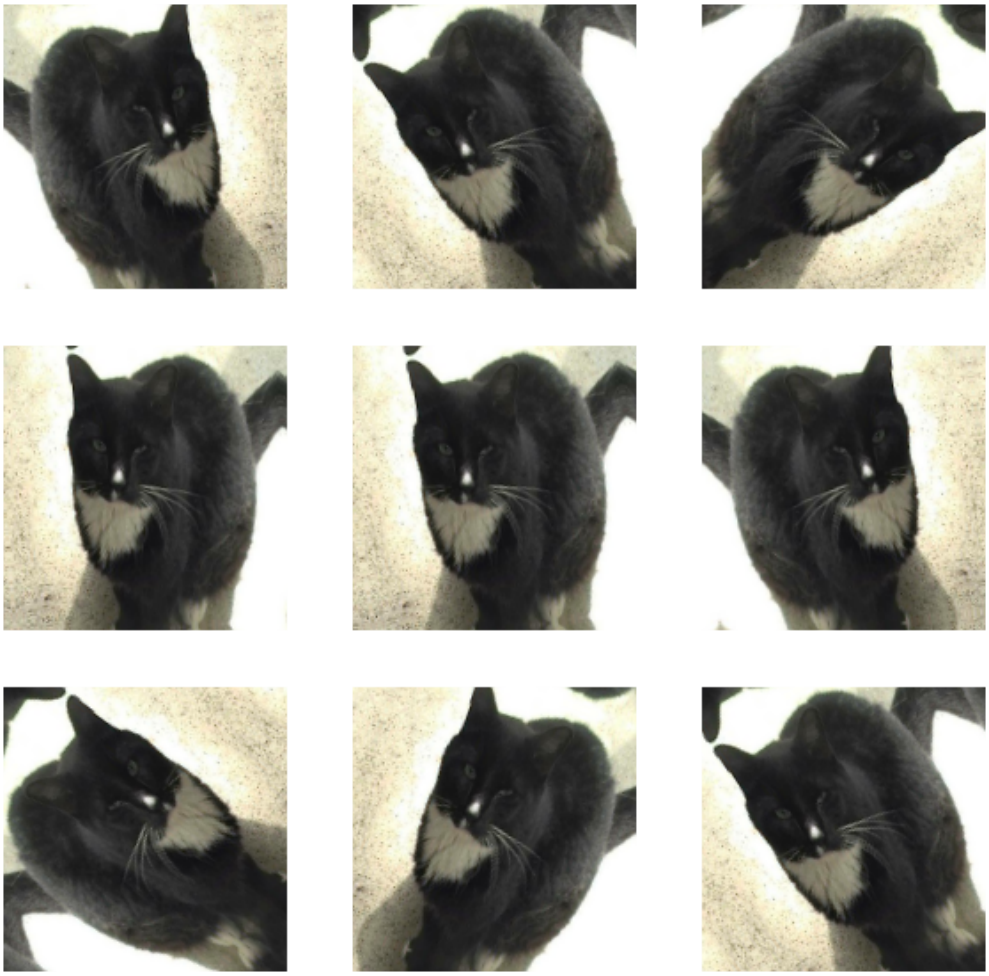
```
data_augmentation = tf.keras.Sequential([
  tf.keras.layers.experimental.preprocessing.RandomFlip('horizontal'),
  tf.keras.layers.experimental.preprocessing.RandomRotation(0.2),
])
```

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.evaulate` or `model.fit` .

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

In [9]:

```
for image, _ in train_dataset.take(1):
  plt.figure(figsize=(10, 10))
  first_image = image[0]
  for i in range(9):
    ax = plt.subplot(3, 3, i + 1)
    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 [10]:

```
1 preprocess_input = tf.keras.applications.mobilenet_v2.preprocess_input
```

Note: Alternatively, you could rescale pixel values from `[0, 255]` to `[-1, 1]` using a [Rescaling](https://www.tensorflow.org/api_docs/python/tf/keras/layers/experimental/preprocessing/Rescaling) layer.

In [11]:

```
1 rescale = tf.keras.layers.experimental.preprocessing.Rescaling(1./127.5, of
```

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 [12]:

```
# Create the base model from the pre-trained model MobileNet V2
IMG_SHAPE = IMG_SIZE + (3,)
base_model = tf.keras.applications.MobileNetV2(input_shape=IMG_SHAPE,
                                               include_top=False,
                                               weights='imagenet')
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v2/mobilenet_v2_weights_tf_dim_ordering_tf_kernels_1.0_160_no_top.h5 (https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v2/mobilenet_v2_weights_tf_dim_ordering_tf_kernels_1.0_160_no_top.h5)
9412608/9406464 [==============================] - 0s 0us/step
9420800/9406464 [==============================] - 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 [13]:

```
image_batch, label_batch = next(iter(train_dataset))
feature_batch = base_model(image_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 [14]:

```
base_model.trainable = False
```

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 (https://www.tensorflow.org/guide/keras/transfer_learning).

In [15]:

```
# 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_1 (InputLayer)         [(None, 160, 160, 3) 0
_________________________________________________________________
Conv1 (Conv2D)               (None, 80, 80, 32)   864
input_1[0][0]
_________________________________________________________________
bn_Conv1 (BatchNormalization)   (None, 80, 80, 32)   128
Conv1[0][0]
_________________________________________________________________
Conv1_relu (ReLU)            (None, 80, 80, 32)   0
```

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 [16]:

```
global_average_layer = tf.keras.layers.GlobalAveragePooling2D()
feature_batch_average = global_average_layer(feature_batch)
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 [17]:

```
prediction_layer = tf.keras.layers.Dense(1)
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 (https://www.tensorflow.org/guide/keras/functional)](https://www.tensorflow.org/guide/keras/functional). As previously mentioned, use `training=False` as our model contains a `BatchNormalization` layer.

In [18]:

```
inputs = tf.keras.Input(shape=(160, 160, 3))
x = data_augmentation(inputs)
x = preprocess_input(x)
x = base_model(x, training=False)
x = global_average_layer(x)
x = tf.keras.layers.Dropout(0.2)(x)
outputs = prediction_layer(x)
model = tf.keras.Model(inputs, outputs)
```

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 [19]:

```
base_learning_rate = 0.0001
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=base_learnin
              loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

In [20]:

```
model.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 160, 160, 3)]     0
_________________________________________________________________
sequential (Sequential)      (None, 160, 160, 3)       0
_________________________________________________________________
tf.math.truediv (TFOpLambda) (None, 160, 160, 3)       0
_________________________________________________________________
tf.math.subtract (TFOpLambda (None, 160, 160, 3)       0
_________________________________________________________________
mobilenetv2_1.00_160 (Functi (None, 5, 5, 1280)        2257984
_________________________________________________________________
global_average_pooling2d (Gl (None, 1280)              0
_________________________________________________________________
dropout (Dropout)            (None, 1280)              0
_________________________________________________________________
dense (Dense)                (None, 1)                 1281
=================================================================
Total params: 2,259,265
Trainable params: 1,281
Non-trainable params: 2,257,984
_________________________________________________________________
```

The 2.5M parameters in MobileNet are frozen, but there are 1.2K *trainable* parameters in the Dense layer. These are divided between two `tf.Variable` objects, the weights and biases.

In [21]:

```
len(model.trainable_variables)
```

Out[21]:

2

Train the model

After training for 10 epochs, you should see ~94% accuracy on the validation set.

In [22]:

```
initial_epochs = 10

loss0, accuracy0 = model.evaluate(validation_dataset)
```

```
26/26 [==============================] - 4s 63ms/step - loss: 1.0
916 - accuracy: 0.3787
```

In [23]:

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

```
initial loss: 1.09
initial accuracy: 0.38
```

In [24]:

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

```
Epoch 1/10
63/63 [==============================] - 10s 94ms/step - loss: 0.
8983 - accuracy: 0.4465 - val_loss: 0.6495 - val_accuracy: 0.6176
Epoch 2/10
63/63 [==============================] - 6s 88ms/step - loss: 0.6
385 - accuracy: 0.6320 - val_loss: 0.4551 - val_accuracy: 0.8057
Epoch 3/10
63/63 [==============================] - 6s 89ms/step - loss: 0.4
912 - accuracy: 0.7465 - val_loss: 0.3450 - val_accuracy: 0.8849
Epoch 4/10
63/63 [==============================] - 6s 89ms/step - loss: 0.4
007 - accuracy: 0.8175 - val_loss: 0.2739 - val_accuracy: 0.9171
Epoch 5/10
63/63 [==============================] - 6s 88ms/step - loss: 0.3
471 - accuracy: 0.8390 - val_loss: 0.2198 - val_accuracy: 0.9431
Epoch 6/10
63/63 [==============================] - 6s 86ms/step - loss: 0.2
997 - accuracy: 0.8770 - val_loss: 0.1897 - val_accuracy: 0.9505
Epoch 7/10
63/63 [==============================] - 6s 88ms/step - loss: 0.2
727 - accuracy: 0.8865 - val_loss: 0.1694 - val_accuracy: 0.9493
Epoch 8/10
63/63 [==============================] - 6s 88ms/step - loss: 0.2
514 - accuracy: 0.8975 - val_loss: 0.1563 - val_accuracy: 0.9567
Epoch 9/10
63/63 [==============================] - 6s 88ms/step - loss: 0.2
404 - accuracy: 0.8995 - val_loss: 0.1392 - val_accuracy: 0.9616
Epoch 10/10
63/63 [==============================] - 6s 89ms/step - loss: 0.2
296 - accuracy: 0.9085 - val_loss: 0.1290 - val_accuracy: 0.9616
```

Learning curves

Let's take a look at the learning curves of the training and validation accuracy/loss when using the MobileNet V2 base model as a fixed feature extractor.

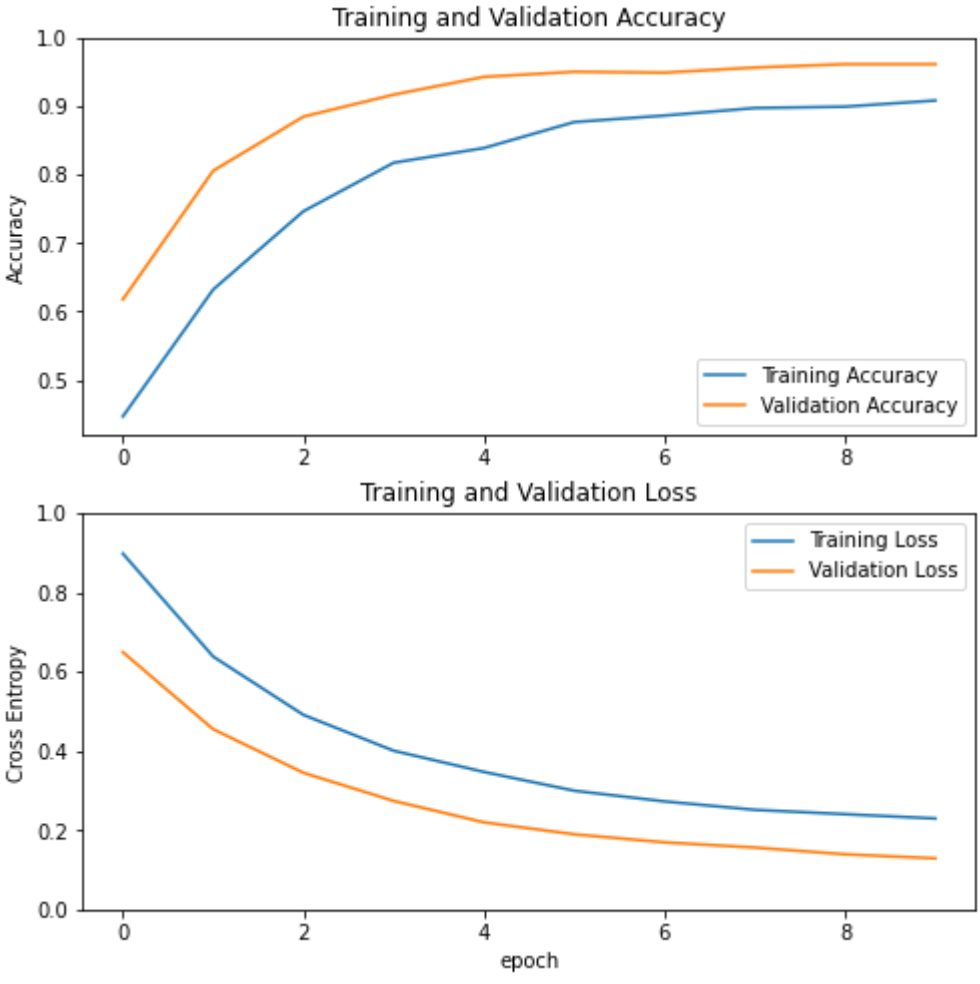
In [25]:

```
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 MobileNet V2 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 [26]:

```
base_model.trainable = True
```

In [27]:

```
# Let's take a look to see how many layers are in the base model
print("Number of layers in the base model: ", len(base_model.layers))

# Fine-tune from this layer onwards
fine_tune_at = 100

# Freeze all the layers before the `fine_tune_at` layer
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 [28]:

```
model.compile(loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              optimizer = tf.keras.optimizers.RMSprop(lr=base_learning_rate
              metrics=['accuracy'])
```

```
/usr/local/lib/python3.7/dist-packages/keras/optimizer_v2/optimiz
er_v2.py:356: UserWarning: The `lr` argument is deprecated, use `
learning_rate` instead.
  "The `lr` argument is deprecated, use `learning_rate` instea
d.")
```

In [29]:

```
1 model.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 160, 160, 3)]     0
_________________________________________________________________
sequential (Sequential)      (None, 160, 160, 3)       0
_________________________________________________________________
tf.math.truediv (TFOpLambda) (None, 160, 160, 3)       0
_________________________________________________________________
tf.math.subtract (TFOpLambda (None, 160, 160, 3)       0
_________________________________________________________________
mobilenetv2_1.00_160 (Functi (None, 5, 5, 1280)        2257984
_________________________________________________________________
global_average_pooling2d (Gl (None, 1280)              0
_________________________________________________________________
dropout (Dropout)            (None, 1280)              0
_________________________________________________________________
dense (Dense)                (None, 1)                 1281
=================================================================
Total params: 2,259,265
Trainable params: 1,862,721
Non-trainable params: 396,544
_________________________________________________________________
```

In [30]:

```
1 len(model.trainable_variables)
```

Out[30]:

```
56
```

Continue training the model

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

In [31]:

```
fine_tune_epochs = 10
total_epochs =  initial_epochs + fine_tune_epochs

history_fine = model.fit(train_dataset,
                         epochs=total_epochs,
                         initial_epoch=history.epoch[-1],
                         validation_data=validation_dataset)
```

```
Epoch 10/20
63/63 [==============================] - 16s 132ms/step - loss:
0.1609 - accuracy: 0.9230 - val_loss: 0.0503 - val_accuracy: 0.98
02
Epoch 11/20
63/63 [==============================] - 8s 115ms/step - loss: 0.
1349 - accuracy: 0.9470 - val_loss: 0.0497 - val_accuracy: 0.9814
Epoch 12/20
63/63 [==============================] - 8s 116ms/step - loss: 0.
0974 - accuracy: 0.9615 - val_loss: 0.0390 - val_accuracy: 0.9889
Epoch 13/20
63/63 [==============================] - 7s 115ms/step - loss: 0.
1028 - accuracy: 0.9560 - val_loss: 0.0400 - val_accuracy: 0.9851
Epoch 14/20
63/63 [==============================] - 7s 114ms/step - loss: 0.
0858 - accuracy: 0.9665 - val_loss: 0.0384 - val_accuracy: 0.9839
Epoch 15/20
63/63 [==============================] - 8s 115ms/step - loss: 0.
0981 - accuracy: 0.9620 - val_loss: 0.0449 - val_accuracy: 0.9802
Epoch 16/20
63/63 [==============================] - 8s 115ms/step - loss: 0.
0771 - accuracy: 0.9670 - val_loss: 0.0512 - val_accuracy: 0.9703
Epoch 17/20
63/63 [==============================] - 7s 114ms/step - loss: 0.
0807 - accuracy: 0.9675 - val_loss: 0.0406 - val_accuracy: 0.9889
Epoch 18/20
63/63 [==============================] - 8s 117ms/step - loss: 0.
0664 - accuracy: 0.9735 - val_loss: 0.0334 - val_accuracy: 0.9901
Epoch 19/20
63/63 [==============================] - 8s 117ms/step - loss: 0.
0642 - accuracy: 0.9760 - val_loss: 0.0320 - val_accuracy: 0.9889
Epoch 20/20
63/63 [==============================] - 8s 115ms/step - loss: 0.
0522 - accuracy: 0.9800 - val_loss: 0.0412 - val_accuracy: 0.9790
```

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 MobileNet V2 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 MobileNet V2 datasets.

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

In [32]:

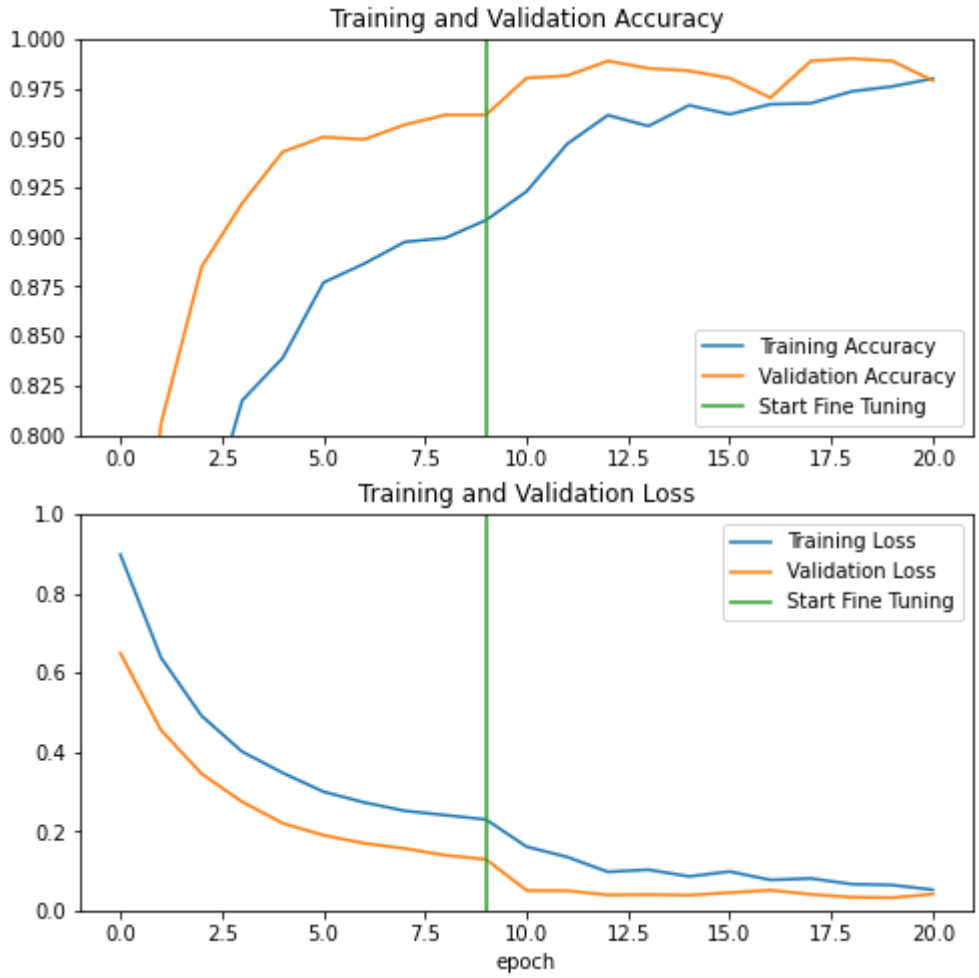
```
acc += history_fine.history['accuracy']
val_acc += history_fine.history['val_accuracy']

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

In [33]:

```
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

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

In [34]:

```
loss, accuracy = model.evaluate(test_dataset)
print('Test accuracy :', accuracy)
```

```
6/6 [==============================] - 1s 65ms/step - loss: 0.030
1 - accuracy: 0.9896
Test accuracy : 0.9895833134651184
```

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

In [35]:

```
# Retrieve a batch of images from the test set
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)

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

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 (https://www.tensorflow.org/guide/keras/transfer_learning).