Initialization

In [1]:
    %load_ext autoreload
    %autoreload 2
    %matplotlib inline

In [2]:
    import pandas as pd
    import numpy as np
    import matplotlib.pyplot as plt

In [11]:
    import os
    from tensorflow.keras.preprocessing.image import load_img, img_to_array
    from tensorflow.keras.applications import VGG16
    from tensorflow.keras.applications.vgg16 import preprocess_input, decode_predictions
    from tensorflow.keras.preprocessing.image import ImageDataGenerator
    from tensorflow.keras.models import Sequential
    from tensorflow.keras.layers import Flatten, Dense
    from tensorflow.keras.optimizers import RMSprop

Acknowledgments

- The diagram is based closely on Figure 11-4 in: A. Géron: *Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow (2nd edn)*, O'Reilly, 2019
- The code is based closely on some of the code in section 5.3 of: François Chollet: *Deep Learning with Python*, Manning Publications, 2018
Warning

- It will probably be impossible for you to run the code in this notebook: it takes unreasonable amounts of time to run.
- You can reduce run times a little by inserting code for early stopping.

Introduction

- A pretrained network is a saved network that was trained, usually on a large dataset.
- Remarkably, these are increasingly being made available for image classification, speech recognition and other tasks:
  - E.g. one 'model zoo' for TensorFlow: https://github.com/tensorflow/models
  - E.g. one model zoo for Caffe: https://github.com/BVLC/caffe/wiki/Model-Zoo
- Consider ImageNet (http://www.image-net.org/):
  - 1.4 million images, each manually labeled with one class per image
  - Thousands of classes, mostly animals and everyday objects
  - Annual competitions (ImageNet Large Scale Visual Recognition Challenges, ILSVRC), now hosted on Kaggle.
- There are several pretrained neural networks for ImageNet, half a dozen of which are made available directly in Keras, including:
  - VGG16
  - VGG19
  - Inception V3
  - Xception
- These networks incorporate a number of innovations, which we don't have time to study in depth, including:
  - Local response normalization: where strongly activated neurons in one feature map can inhibit neurons in the same position in neighbouring feature maps, which encourages feature maps to specialize.
  - Skip connections: where the input into one layer is also added to the output of a layer a bit higher up the stack. This means that the later layer must in effect learn the difference (residual) between what it would have learned and the amount that has been added. This boosts the signal across the network, which can encourage learning.
  - Inception modules: where (roughly) there are multiple convolutions with different window sizes operating at the same level of the network, making the network wider rather than deeper. They help reduce dimensionality and result in networks with fewer parameters.
  - Depthwise separable convolutional layers: where (roughly) there are convolutional layers whose neurons connect to only one feature map in the layer below (to detect spatial patterns) and then regular convolutional layers (but with $1 \times 1$ windows) that look for patterns between the feature maps of the spatial layer.

Every year sees further innovations. Many are trying to find better ways of learning how objects are composed of smaller parts; for one example, see capsules.

- In this lecture, we'll see how to use a pretrained network (VGG16) for transfer learning: using it as the lower layers of our own network.

VGG16

```python
In [4]: vgg16_base = VGG16(weights="imagenet", include_top=False, input_shape=(224, 224, 3))
```
In [5]:

```python
vgg16_base.summary()
```

Model: "vgg16"

<table>
<thead>
<tr>
<th>Layer (type)</th>
<th>Output Shape</th>
<th>Param #</th>
</tr>
</thead>
<tbody>
<tr>
<td>input_1 (InputLayer)</td>
<td>[(None, 224, 224, 3)]</td>
<td>0</td>
</tr>
<tr>
<td>block1_conv1 (Conv2D)</td>
<td>(None, 224, 224, 64)</td>
<td>1792</td>
</tr>
<tr>
<td>block1_conv2 (Conv2D)</td>
<td>(None, 224, 224, 64)</td>
<td>36928</td>
</tr>
<tr>
<td>block1_pool (MaxPooling2D)</td>
<td>(None, 112, 112, 64)</td>
<td>0</td>
</tr>
<tr>
<td>block2_conv1 (Conv2D)</td>
<td>(None, 112, 112, 128)</td>
<td>73856</td>
</tr>
<tr>
<td>block2_conv2 (Conv2D)</td>
<td>(None, 112, 112, 128)</td>
<td>147584</td>
</tr>
<tr>
<td>block2_pool (MaxPooling2D)</td>
<td>(None, 56, 56, 128)</td>
<td>0</td>
</tr>
<tr>
<td>block3_conv1 (Conv2D)</td>
<td>(None, 56, 56, 256)</td>
<td>295168</td>
</tr>
<tr>
<td>block3_conv2 (Conv2D)</td>
<td>(None, 56, 56, 256)</td>
<td>590080</td>
</tr>
<tr>
<td>block3_conv3 (Conv2D)</td>
<td>(None, 56, 56, 256)</td>
<td>590080</td>
</tr>
<tr>
<td>block3_pool (MaxPooling2D)</td>
<td>(None, 28, 28, 256)</td>
<td>0</td>
</tr>
<tr>
<td>block4_conv1 (Conv2D)</td>
<td>(None, 28, 28, 512)</td>
<td>1180160</td>
</tr>
<tr>
<td>block4_conv2 (Conv2D)</td>
<td>(None, 28, 28, 512)</td>
<td>2359808</td>
</tr>
<tr>
<td>block4_conv3 (Conv2D)</td>
<td>(None, 28, 28, 512)</td>
<td>2359808</td>
</tr>
<tr>
<td>block4_pool (MaxPooling2D)</td>
<td>(None, 14, 14, 512)</td>
<td>0</td>
</tr>
<tr>
<td>block5_conv1 (Conv2D)</td>
<td>(None, 14, 14, 512)</td>
<td>2359808</td>
</tr>
<tr>
<td>block5_conv2 (Conv2D)</td>
<td>(None, 14, 14, 512)</td>
<td>2359808</td>
</tr>
<tr>
<td>block5_conv3 (Conv2D)</td>
<td>(None, 14, 14, 512)</td>
<td>2359808</td>
</tr>
<tr>
<td>block5_pool (MaxPooling2D)</td>
<td>(None, 7, 7, 512)</td>
<td>0</td>
</tr>
</tbody>
</table>

Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0

In [6]:

```python
vgg16 = VGG16(weights="imagenet", include_top=True)
```
Let's read in the same four images we used as the end of the previous lecture, turn them into a tensor and see what the VGG16 model predicts for the images in this tensor.
In [8]:
    path = "datasets/dataset_wikipedia_images"
    pathnames = [os.path.join(path, filename) for filename in sorted(os.listdir(path))]
    imgs = [load_img(img_path, target_size=(224, 224)) for img_path in pathnames]

In [9]:
    img_tensor = np.array([img_to_array(img) for img in imgs])
    img_tensor = preprocess_input(img_tensor)

In [10]:
    preds = vgg16.predict(img_tensor)
    decoded_preds = decode_predictions(preds, top=3)
    for dp in decoded_preds:
        print("Predicted: " + str(dp))

    Predicted: [('n02124075', 'Egyptian_cat', 0.5274658), ('n02123045', 'tabby', 0.3074059), ('n02123159', 'tiger_cat', 0.121831566)]
    Predicted: [('n02099601', 'golden_retriever', 0.6207083), ('n02100735', 'English_setter', 0.08977295), ('n02100877', 'Irish_setter', 0.05257782)]
    Predicted: [('n02412080', 'ram', 0.99507445), ('n02395406', 'hog', 0.004475237), ('n02105412', 'kelpie', 0.00021943483)]
    Predicted: [('n04398044', 'teapot', 0.98947453), ('n03063689', 'coffeepot', 0.007865551), ('n04560804', 'water_jug', 0.00071867165)]
Transfer Learning

- **Transfer learning:**
  - taking a model that was learned when solving one problem and re-using it for solving a different but related problem
- **Advantages:**
  - It speeds-up training for the new problem
  - It means that less training data may be needed for the new problem
- Deep neural networks are more amenable to transfer learning than many other machine learning techniques:
  - Take a pre-trained network
  - Re-use its lower layers, even **freezing** their weights

Re-using the Convolutional Base of a Pretrained ConvNet

- Convolutional Neural Networks typically comprise two parts:
  - the **convolutional base**: the convolutional and pooling layers
  - the densely-connected top layers for, e.g. classification
- We want to reuse the convolutional base:
  - the features learned by these layers are likely to be more generic
  - the features learned by the top layers will be more specific to the original task

- E.g.
  - We have several networks that are pre-trained on ImageNet, including VGG16:
    - Trained to classify images into 1000 classes (various animals, vehicles, etc.)
    - We can re-use the lower layers in a new network that is trained to classify images of just cats and dogs, or different types of vehicles, or for face recognition, or perhaps even facial expression recognition
  - Of course, this will only work well if the original and new tasks share similar low-level features
    - The more similar the new problem is to the original problem, the more layers we may want to re-use
It's Raining Cats and Dogs, again

- We'll re-use the convolutional base of the VGG16 model within a new network for classifying cats and dogs
- In Keras we use `include_top=True` to instantiate all layers of one of its pre-trained networks or `include_top=False` to instantiate just the convolutional base

```
In [12]: vgg16_base = VGG16(weights="imagenet", include_top=False, input_shape=(150,150,3))

In [13]: vgg16_base.summary()
```

<table>
<thead>
<tr>
<th>Layer (type)</th>
<th>Output Shape</th>
<th>Param #</th>
</tr>
</thead>
<tbody>
<tr>
<td>input_3 (InputLayer)</td>
<td>[(None, 150, 150, 3)]</td>
<td>0</td>
</tr>
<tr>
<td>block1_conv1 (Conv2D)</td>
<td>(None, 150, 150, 64)</td>
<td>1792</td>
</tr>
<tr>
<td>block1_conv2 (Conv2D)</td>
<td>(None, 150, 150, 64)</td>
<td>36928</td>
</tr>
<tr>
<td>block1_pool (MaxPooling2D)</td>
<td>(None, 75, 75, 64)</td>
<td>0</td>
</tr>
<tr>
<td>block2_conv1 (Conv2D)</td>
<td>(None, 75, 75, 128)</td>
<td>73856</td>
</tr>
<tr>
<td>block2_conv2 (Conv2D)</td>
<td>(None, 75, 75, 128)</td>
<td>147584</td>
</tr>
<tr>
<td>block2_pool (MaxPooling2D)</td>
<td>(None, 37, 37, 128)</td>
<td>0</td>
</tr>
<tr>
<td>block3_conv1 (Conv2D)</td>
<td>(None, 37, 37, 256)</td>
<td>295168</td>
</tr>
<tr>
<td>block3_conv2 (Conv2D)</td>
<td>(None, 37, 37, 256)</td>
<td>590080</td>
</tr>
<tr>
<td>block3_conv3 (Conv2D)</td>
<td>(None, 37, 37, 256)</td>
<td>590080</td>
</tr>
<tr>
<td>block3_pool (MaxPooling2D)</td>
<td>(None, 18, 18, 256)</td>
<td>0</td>
</tr>
<tr>
<td>block4_conv1 (Conv2D)</td>
<td>(None, 18, 18, 512)</td>
<td>1180160</td>
</tr>
<tr>
<td>block4_conv2 (Conv2D)</td>
<td>(None, 18, 18, 512)</td>
<td>2359808</td>
</tr>
<tr>
<td>block4_conv3 (Conv2D)</td>
<td>(None, 18, 18, 512)</td>
<td>2359808</td>
</tr>
<tr>
<td>block4_pool (MaxPooling2D)</td>
<td>(None, 9, 9, 512)</td>
<td>0</td>
</tr>
<tr>
<td>block5_conv1 (Conv2D)</td>
<td>(None, 9, 9, 512)</td>
<td>2359808</td>
</tr>
<tr>
<td>block5_conv2 (Conv2D)</td>
<td>(None, 9, 9, 512)</td>
<td>2359808</td>
</tr>
<tr>
<td>block5_conv3 (Conv2D)</td>
<td>(None, 9, 9, 512)</td>
<td>2359808</td>
</tr>
<tr>
<td>block5_pool (MaxPooling2D)</td>
<td>(None, 4, 4, 512)</td>
<td>0</td>
</tr>
</tbody>
</table>

Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0

- Now we create our new network but we add the pre-trained model, just as we would add a layer
In [14]:
    def build_network():
        network = Sequential()
        network.add(vgg16_base);
        vgg16_base.trainable = False
        network.add(Flatten())
        network.add(Dense(256, activation="relu")
        network.add(Dense(1, activation="sigmoid")
        network.compile(optimizer=RMSprop(lr=0.00003), loss="binary_crossentropy",
        metrics=["accuracy"])
        return network
        
    Note how we froze the weights in the layers of the convolutional base
    If we did not, then the features that were learned previously would be lost

In [15]:
    network = build_network()

In [16]:
    network.summary()

    Model: "sequential"
    ________________________________________________________________
    Layer (type)                 Output Shape              Param #
    =------------------------------------------------------------------
    vgg16 (Model)                (None, 4, 4, 512)         14714688
    flatten (Flatten)            (None, 8192)              0
    dense (Dense)                (None, 256)               2097408
    dense_1 (Dense)              (None, 1)                 257
    =------------------------------------------------------------------
    Total params: 16,812,353
    Trainable params: 2,097,665
    Non-trainable params: 14,714,688
    
    From now on, the code becomes familiar

In [17]:
    base_dir = "datasets/dataset_cats_and_dogs"
    train_dir = os.path.join(base_dir, "train")
    val_dir = os.path.join(base_dir, "validation")
    test_dir = os.path.join(base_dir, "test")
In [18]:
    augmented_train_data_generator = ImageDataGenerator(
        rescale=1./255,
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode="nearest")

    augmented_train_generator = augmented_train_data_generator.flow_from_directory(
        train_dir, target_size=(150, 150), batch_size=20, class_mode="binary")

    val_data_generator = ImageDataGenerator(rescale=1./255)
    val_generator = val_data_generator.flow_from_directory(
        val_dir, target_size=(150, 150), batch_size=20, class_mode="binary")

    test_data_generator = ImageDataGenerator(rescale=1./255)
    test_generator = test_data_generator.flow_from_directory(
        test_dir, target_size=(150, 150), batch_size=20, class_mode="binary")

    Found 2000 images belonging to 2 classes.
    Found 1000 images belonging to 2 classes.
    Found 1000 images belonging to 2 classes.

In [19]:
    history = network.fit_generator(
        augmented_train_generator, steps_per_epoch=100, epochs=30,
        validation_data=val_generator, validation_steps=50,
        verbose=0)

In [20]:
    x_vals = range(1, 31)

In [21]:
    train_loss = history.history["loss"]
    val_loss = history.history["val_loss"]

    fig = plt.figure()
    plt.title("Loss")
    plt.plot(x_vals, train_loss, label="Training loss", color="purple")
    plt.plot(x_vals, val_loss, label="Validation loss", color="orange")
    plt.legend()
    plt.show()
In [22]:

```python
train_acc = history.history["accuracy"]
val_acc = history.history["val_accuracy"]

fig = plt.figure()
plt.title("Accuracy")
plt.plot(x_vals, train_acc, label="Training accuracy", color = "purple")
plt.plot(x_vals, val_acc, label="Validation accuracy", color = "orange")
plt.legend()
plt.show()
```

In [23]:

```python
test_loss, test_acc = network.evaluate_generator(test_generator, steps=50)
test_acc
```

Out[23]: 0.881

- In lecture 13, we achieved an accuracy in the mid-seventies%
- Here we have accuracy in the high eighties%

**Unfreezing Parts of the Convolutonal Base**

- It might be helpful to unfreeze the top layers of the pre-trained network
- This allows them to adjust their weights to learn features more suitable to the problem at hand
- In fact, it can be better to take the following approach:
  - First, do everything that we did above:
    - Create a new network on top of a convolutional base
    - Freeze the convolutional base
    - Train on our examples
  - Only now do we unfreeze the top layers of the convolutional base
  - Train again
- Why this more complicated approach?

**Cats and Dogs, Resumed**

- We've already done part of the job: trained with a frozen convolutional base
- So now, continuing on from what we've already done, we unfreeze some layers and re-train
In [24]:

```python
def build_network():
    network = Sequential()
    network.add(vgg16_base);
    # Unfreeze all layers
    vgg16_base.trainable = True
    # Re-freeze lower layers, up to and including block5_conv1
    trainable = False
    for layer in vgg16_base.layers:
        if layer.name == "block5_conv1":
            trainable = True
        layer.trainable = trainable
    network.add(Flatten())
    network.add(Dense(256, activation="relu"))
    network.add(Dense(1, activation="sigmoid"))
    network.compile(optimizer=RMSprop(lr=0.00003), loss="binary_crossentropy",
                     metrics=["accuracy"])
    return network
```

In [25]:

```python
network = build_network()
```

In [26]:

```python
history = network.fit_generator(
    augmented_train_generator, steps_per_epoch=100, epochs=30,
    validation_data=val_generator, validation_steps=50,
    verbose=0)
```

In [27]:

```python
train_loss = history.history["loss"]
val_loss = history.history["val_loss"]

fig = plt.figure()
plt.title("Loss")
plt.plot(x_vals, train_loss, label="Training loss", color = "purple")
plt.plot(x_vals, val_loss, label="Validation loss", color = "orange")
plt.legend()
plt.show()
```
In [28]:

```python
train_acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

fig = plt.figure()
plt.title('Accuracy')
plt.plot(x_vals, train_acc, label='Training accuracy', color='purple')
plt.plot(x_vals, val_acc, label='Validation accuracy', color='orange')
plt.legend()
plt.show()
```

![Accuracy graph](image)  

In [29]:

```python
test_loss, test_acc = network.evaluate_generator(test_generator, steps=50)
test_acc
```

Out[29]: 0.927

**Concluding Remarks**

- It's often said that we need lots of data for deep learning
- But transfer learning helps us in cases where we have more limited data
- E.g. we want to do face recognition but we have only a few pictures of each person, not enough to train a good classifier
  - Collect loads of pictures of faces of random people from the web
  - Train a network to detect whether or not two different pictures portray the same person
  - This network is presumably a good feature detector for faces
  - Use this network as the lower layers of your face recognition classifier
- E.g. we want to do some natural language processing but we don’t have a large dataset
  - Collect loads of sentences from the web, label them ‘good’
  - Corrupt these sentences in various ways, and label them ‘bad’
  - Train a network to classify sentences and non-sentences (‘good’ vs. ‘bad’)
  - Use this network as the lower layers of your natural language processing system