In [3]:

```
import tensorflow as tf
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)
```

In [4]:

```
import keras
keras.__version__
```

Out[4]:

```
'2.3.0'
```

Using a pre-trained convnet

A common and highly effective approach to deep learning on small image datasets is to leverage a pre-trained network. A pre-trained network is simply a saved network previously trained on a large dataset, typically on a large-scale image classification task. If this original dataset is large enough and general enough, then the spatial feature hierarchy learned by the pre-trained network can effectively act as a generic model of our visual world, and hence its features can prove useful for many different computer vision problems, even though these new problems might involve completely different classes from those of the original task. For instance, one might train a network on ImageNet (where classes are mostly animals and everyday objects) and then re-purpose this trained network for something as remote as identifying furniture items in images. Such portability of learned features across different problems is a key advantage of deep learning compared to many older shallow learning approaches, and it makes deep learning very effective for small-data problems.

In our case, we will consider a large convnet trained on the ImageNet dataset (1.4 million labeled images and 1000 different classes). ImageNet contains many animal classes, including different species of cats and dogs, and we can thus expect to perform very well on our cat vs. dog classification problem.

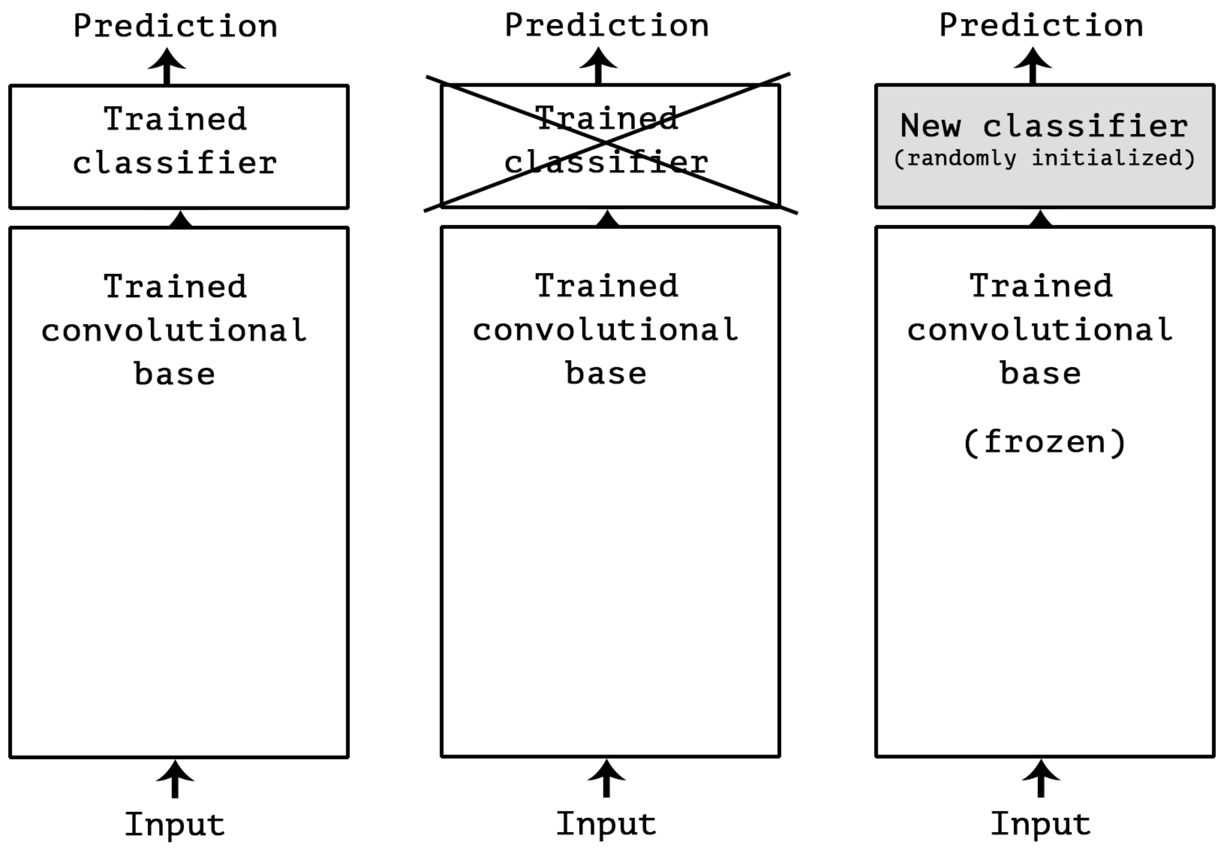
We will use the VGG16 architecture, developed by Karen Simonyan and Andrew Zisserman in 2014, a simple and widely used convnet architecture for ImageNet. Although it is a bit of an older model, far from the current state of the art and somewhat heavier than many other recent models, we chose it because its architecture is similar to what you are already familiar with, and easy to understand without introducing any new concepts. This may be your first encounter with one of these cutesie model names -- VGG, ResNet, Inception, Inception-ResNet, Xception... you will get used to them, as they will come up frequently if you keep doing deep learning for computer vision.

There are two ways to leverage a pre-trained network: *feature extraction* and *fine-tuning*. We will cover both of them. Let's start with feature extraction.

Feature extraction

Feature extraction consists of using the representations learned by a previous network to extract interesting features from new samples. These features are then run through a new classifier, which is trained from scratch.

As we saw previously, convnets used for image classification comprise two parts: they start with a series of pooling and convolution layers, and they end with a densely-connected classifier. The first part is called the "convolutional base" of the model. In the case of convnets, "feature extraction" will simply consist of taking the convolutional base of a previously-trained network, running the new data through it, and training a new classifier on top of the output.



Why only reuse the convolutional base? Could we reuse the densely-connected classifier as well? In general, it should be avoided. The reason is simply that the representations learned by the convolutional base are likely to be more generic and therefore more reusable: the feature maps of a convnet are presence maps of generic concepts over a picture, which is likely to be useful regardless of the computer vision problem at hand. On the other end, the representations learned by the classifier will necessarily be very specific to the set of classes that the model was trained on -- they will only contain information about the presence probability of this or that class in the entire picture. Additionally, representations found in densely-connected layers no longer contain any information about *where* objects are located in the input image: these layers get rid of the notion of space, whereas the object location is still described by convolutional feature maps. For problems where object location matters, densely-connected features would be largely useless.

Note that the level of generality (and therefore reusability) of the representations extracted by specific convolution layers depends on the depth of the layer in the model. Layers that come earlier in the model extract local, highly generic feature maps (such as visual edges, colors, and textures), while layers higher-up extract more abstract concepts (such as "cat ear" or "dog eye"). So if your new dataset differs a lot from the dataset that the original model was trained on, you may be better off using only the first few layers of the model to do feature extraction, rather than using the entire convolutional base.

In our case, since the ImageNet class set did contain multiple dog and cat classes, it is likely that it would be beneficial to reuse the information contained in the densely-connected layers of the original model. However, we will chose not to, in order to cover the more general case where the class set of the new problem does not overlap with the class set of the original model.

Let's put this in practice by using the convolutional base of the VGG16 network, trained on ImageNet, to extract interesting features from our cat and dog images, and then training a cat vs. dog classifier on top of these features.

The VGG16 model, among others, comes pre-packaged with Keras. You can import it from the `keras.applications` module. Here's the list of image classification models (all pre-trained on the ImageNet dataset) that are available as part of `keras.applications` :

- Xception
- InceptionV3
- ResNet50
- VGG16
- VGG19
- MobileNet

Let's instantiate the VGG16 model:

In [5]:

```
from keras.applications import VGG16

conv_base = VGG16(weights='imagenet',
                  include_top=False,
                  input_shape=(150, 150, 3))
```

We passed three arguments to the constructor:

- `weights` , to specify which weight checkpoint to initialize the model from
- `include_top` , which refers to including or not the densely-connected classifier on top of the network. By default, this densely-connected classifier would correspond to the 1000 classes from ImageNet. Since we intend to use our own densely-connected classifier (with only two classes, cat and dog), we don't need to include it.
- `input_shape` , the shape of the image tensors that we will feed to the network. This argument is purely optional: if we don't pass it, then the network will be able to process inputs of any size.

Here's the detail of the architecture of the VGG16 convolutional base: it's very similar to the simple convnets that you are already familiar with.

In [6]:

```
conv_base.summary()
```

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         (None, 150, 150, 3)       0         
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 150, 150, 64)      1792      
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 150, 150, 64)      36928     
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 75, 75, 64)        0         
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 75, 75, 128)       73856     
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 75, 75, 128)       147584    
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 37, 37, 128)       0         
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 37, 37, 256)       295168    
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 37, 37, 256)       590080    
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 37, 37, 256)       590080    
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 18, 18, 256)       0         
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 18, 18, 512)       1180160   
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 18, 18, 512)       2359808   
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 18, 18, 512)       2359808   
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 9, 9, 512)         0         
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 9, 9, 512)         2359808   
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 9, 9, 512)         2359808   
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 9, 9, 512)         2359808   
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 4, 4, 512)         0         
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
```

The final feature map has shape `(4, 4, 512)`. That's the feature on top of which we will stick a densely-connected classifier.

At this point, there are two ways we could proceed:

- Running the convolutional base over our dataset, recording its output to a Numpy array on disk, then using this data as input to a standalone densely-connected classifier similar to those you have seen in the first chapters of this book. This solution is very fast and cheap to run, because it only requires running the convolutional base once for every input image, and the convolutional base is by far the most expensive part of the pipeline. However, for the exact same reason, this technique would not allow us to leverage data augmentation at all.
- Extending the model we have (`conv_base`) by adding `Dense` layers on top, and running the whole thing end-to-end on the input data. This allows us to use data augmentation, because every input image is going through the convolutional base every time it is seen by the model. However, for this same reason, this technique is far more expensive than the first one.

We will cover both techniques. Let's walk through the code required to set-up the first one: recording the output of `conv_base` on our data and using these outputs as inputs to a new model.

We will start by simply running instances of the previously-introduced `ImageDataGenerator` to extract images as Numpy arrays as well as their labels. We will extract features from these images simply by calling the `predict` method of the `conv_base` model.

In [7]:

```
import os
import numpy as np
from keras.preprocessing.image import ImageDataGenerator

base_dir = 'dogs-vs-cats-sample/'

train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')

datagen = ImageDataGenerator(rescale=1./255)
batch_size = 20

def extract_features(directory, sample_count):
    features = np.zeros(shape=(sample_count, 4, 4, 512))
    labels = np.zeros(shape=(sample_count))
    generator = datagen.flow_from_directory(
        directory,
        target_size=(150, 150),
        batch_size=batch_size,
        class_mode='binary')
    i = 0
    for inputs_batch, labels_batch in generator:
        features_batch = conv_base.predict(inputs_batch)
        features[i * batch_size : (i + 1) * batch_size] = features_batch
        labels[i * batch_size : (i + 1) * batch_size] = labels_batch
        i += 1
        if i * batch_size >= sample_count:
            # Note that since generators yield data indefinitely in a loop,
            # we must `break` after every image has been seen once.
            break
    return features, labels

train_features, train_labels = extract_features(train_dir, 2000)
validation_features, validation_labels = extract_features(validation_dir, 1000)
test_features, test_labels = extract_features(test_dir, 1000)
```

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

The extracted features are currently of shape `(samples, 4, 4, 512)`. We will feed them to a densely-connected classifier, so first we must flatten them to `(samples, 8192)`:

In [8]:

```
train_features = np.reshape(train_features, (2000, 4 * 4 * 512))
validation_features = np.reshape(validation_features, (1000, 4 * 4 * 512))
test_features = np.reshape(test_features, (1000, 4 * 4 * 512))
```

At this point, we can define our densely-connected classifier (note the use of dropout for regularization), and train it on the data and labels that we just recorded:

In [9]:

```
from keras import models
from keras import layers
from keras import optimizers

model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_dim=4 * 4 * 512))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer=optimizers.RMSprop(lr=2e-5),
              loss='binary_crossentropy',
              metrics=['acc'])

history = model.fit(train_features, train_labels,
                    epochs=30,
                    batch_size=20,
                    validation_data=(validation_features, validation_labels))
```

```
Train on 2000 samples, validate on 1000 samples
Epoch 1/30
2000/2000 [==============================] - 1s 377us/step - loss:
0.6001 - acc: 0.6685 - val_loss: 0.4373 - val_acc: 0.8220
Epoch 2/30
2000/2000 [==============================] - 0s 203us/step - loss:
0.4213 - acc: 0.8130 - val_loss: 0.3586 - val_acc: 0.8620
Epoch 3/30
2000/2000 [==============================] - 0s 201us/step - loss:
0.3634 - acc: 0.8465 - val_loss: 0.3228 - val_acc: 0.8740
Epoch 4/30
2000/2000 [==============================] - 0s 195us/step - loss:
0.3275 - acc: 0.8635 - val_loss: 0.2999 - val_acc: 0.8850
Epoch 5/30
2000/2000 [==============================] - 0s 197us/step - loss:
0.2896 - acc: 0.8865 - val_loss: 0.2831 - val_acc: 0.8970
Epoch 6/30
2000/2000 [==============================] - 0s 197us/step - loss:
0.2641 - acc: 0.8910 - val_loss: 0.2742 - val_acc: 0.9050
Epoch 7/30
2000/2000 [==============================] - 0s 199us/step - loss:
0.2525 - acc: 0.9055 - val_loss: 0.2643 - val_acc: 0.8950
Epoch 8/30
2000/2000 [==============================] - 0s 204us/step - loss:
0.2232 - acc: 0.9180 - val_loss: 0.2677 - val_acc: 0.8870
Epoch 9/30
2000/2000 [==============================] - 0s 205us/step - loss:
0.2211 - acc: 0.9165 - val_loss: 0.2573 - val_acc: 0.8920
Epoch 10/30
2000/2000 [==============================] - 0s 198us/step - loss:
0.2060 - acc: 0.9185 - val_loss: 0.2534 - val_acc: 0.8970
Epoch 11/30
2000/2000 [==============================] - 0s 197us/step - loss:
0.1965 - acc: 0.9270 - val_loss: 0.2440 - val_acc: 0.9060
Epoch 12/30
2000/2000 [==============================] - 0s 196us/step - loss:
0.1832 - acc: 0.9350 - val_loss: 0.2411 - val_acc: 0.9100
Epoch 13/30
2000/2000 [==============================] - 0s 200us/step - loss:
0.1828 - acc: 0.9305 - val_loss: 0.2471 - val_acc: 0.8990
Epoch 14/30
2000/2000 [==============================] - 0s 204us/step - loss:
0.1712 - acc: 0.9400 - val_loss: 0.2395 - val_acc: 0.9040
Epoch 15/30
2000/2000 [==============================] - 0s 205us/step - loss:
0.1649 - acc: 0.9395 - val_loss: 0.2379 - val_acc: 0.9060
Epoch 16/30
2000/2000 [==============================] - 0s 203us/step - loss:
0.1513 - acc: 0.9420 - val_loss: 0.2423 - val_acc: 0.9010
Epoch 17/30
2000/2000 [==============================] - 0s 197us/step - loss:
0.1553 - acc: 0.9465 - val_loss: 0.2346 - val_acc: 0.9050
Epoch 18/30
2000/2000 [==============================] - 0s 196us/step - loss:
0.1438 - acc: 0.9475 - val_loss: 0.2436 - val_acc: 0.9010
Epoch 19/30
2000/2000 [==============================] - 0s 205us/step - loss:
0.1342 - acc: 0.9520 - val_loss: 0.2332 - val_acc: 0.9060
Epoch 20/30
2000/2000 [==============================] - 0s 206us/step - loss:
0.1312 - acc: 0.9570 - val_loss: 0.2330 - val_acc: 0.9030
```

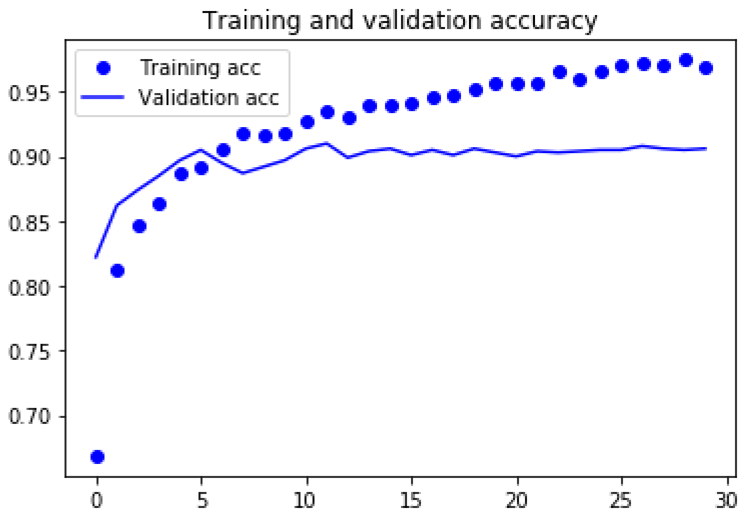
```
Epoch 21/30
2000/2000 [==============================] - 0s 206us/step - loss:
0.1248 - acc: 0.9570 - val_loss: 0.2503 - val_acc: 0.9000
Epoch 22/30
2000/2000 [==============================] - 0s 200us/step - loss:
0.1263 - acc: 0.9570 - val_loss: 0.2358 - val_acc: 0.9040
Epoch 23/30
2000/2000 [==============================] - 0s 197us/step - loss:
0.1148 - acc: 0.9665 - val_loss: 0.2335 - val_acc: 0.9030
Epoch 24/30
2000/2000 [==============================] - 0s 198us/step - loss:
0.1118 - acc: 0.9595 - val_loss: 0.2340 - val_acc: 0.9040
Epoch 25/30
2000/2000 [==============================] - 0s 200us/step - loss:
0.1072 - acc: 0.9660 - val_loss: 0.2350 - val_acc: 0.9050
Epoch 26/30
2000/2000 [==============================] - 0s 197us/step - loss:
0.1026 - acc: 0.9710 - val_loss: 0.2345 - val_acc: 0.9050
Epoch 27/30
2000/2000 [==============================] - 0s 200us/step - loss:
0.0999 - acc: 0.9725 - val_loss: 0.2398 - val_acc: 0.9080
Epoch 28/30
2000/2000 [==============================] - 0s 207us/step - loss:
0.0989 - acc: 0.9700 - val_loss: 0.2437 - val_acc: 0.9060
Epoch 29/30
2000/2000 [==============================] - 0s 207us/step - loss:
0.0890 - acc: 0.9750 - val_loss: 0.2440 - val_acc: 0.9050
Epoch 30/30
2000/2000 [==============================] - 0s 200us/step - loss:
0.0925 - acc: 0.9695 - val_loss: 0.2403 - val_acc: 0.9060
```

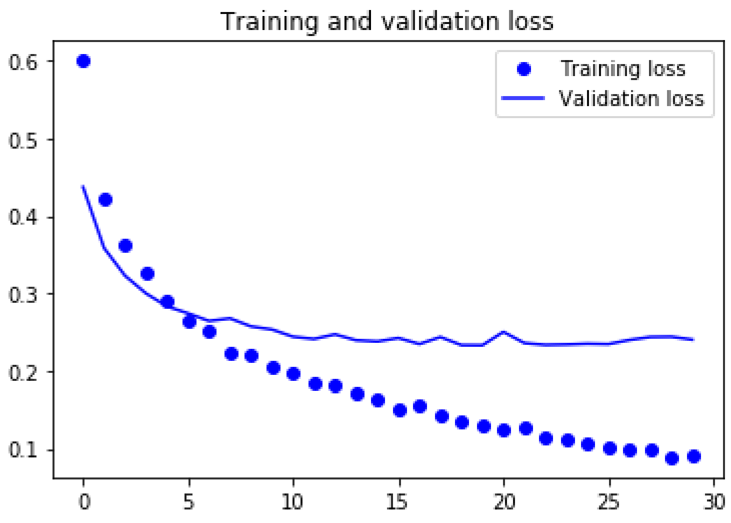
Training is very fast, since we only have to deal with two `Dense` layers -- an epoch takes less than one second even on CPU.

Let's take a look at the loss and accuracy curves during training:

In [11]:

```
import matplotlib.pyplot as plt

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

epochs = range(len(acc))

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```





We reach a validation accuracy of about 90%, much better than what we could achieve in the previous section with our small model trained from scratch. However, our plots also indicate that we are overfitting almost from the start -- despite using dropout with a fairly large rate. This is because this technique does not leverage data augmentation, which is essential to preventing overfitting with small image datasets.

Now, let's review the second technique we mentioned for doing feature extraction, which is much slower and more expensive, but which allows us to leverage data augmentation during training: extending the `conv_base` model and running it end-to-end on the inputs. Note that this technique is in fact so expensive that you should only attempt it if you have access to a GPU: it is absolutely intractable on CPU. If you cannot run your code on GPU, then the previous technique is the way to go.

Because models behave just like layers, you can add a model (like our `conv_base`) to a `Sequential` model just like you would add a layer. So you can do the following:

In [12]:

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

This is what our model looks like now:

In [13]:

```
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
vgg16 (Model)                (None, 4, 4, 512)         14714688  
_________________________________________________________________
flatten_1 (Flatten)          (None, 8192)              0         
_________________________________________________________________
dense_3 (Dense)              (None, 256)               2097408   
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 257       
=================================================================
Total params: 16,812,353
Trainable params: 16,812,353
Non-trainable params: 0
_________________________________________________________________
```

As you can see, the convolutional base of VGG16 has 14,714,688 parameters, which is very large. The classifier we are adding on top has 2 million parameters.

Before we compile and train our model, a very important thing to do is to freeze the convolutional base. "Freezing" a layer or set of layers means preventing their weights from getting updated during training. If we don't do this, then the representations that were previously learned by the convolutional base would get modified during training. Since the `Dense` layers on top are randomly initialized, very large weight updates would be propagated through the network, effectively destroying the representations previously learned.

In Keras, freezing a network is done by setting its `trainable` attribute to `False` :

In [14]:

```
print('This is the number of trainable weights '
      'before freezing the conv base:', len(model.trainable_weights))
```

```
This is the number of trainable weights before freezing the conv bas
e: 30
```

In [15]:

```
conv_base.trainable = False
```

In [16]:

```
print('This is the number of trainable weights '
      'after freezing the conv base:', len(model.trainable_weights))
```

```
This is the number of trainable weights after freezing the conv bas
e: 4
```

With this setup, only the weights from the two `Dense` layers that we added will be trained. That's a total of four weight tensors: two per layer (the main weight matrix and the bias vector). Note that in order for these changes to take effect, we must first compile the model. If you ever modify weight trainability after compilation, you should then re-compile the model, or these changes would be ignored.

Now we can start training our model, with the same data augmentation configuration that we used in our previous example:

In [18]:

```
from keras.preprocessing.image import ImageDataGenerator

train_datagen = 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')

# Note that the validation data should not be augmented!
test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        # This is the target directory
        train_dir,
        # All images will be resized to 150x150
        target_size=(150, 150),
        batch_size=20,
        # Since we use binary_crossentropy loss, we need binary labels
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=2e-5),
              metrics=['acc'])

history = model.fit_generator(
      train_generator,
      steps_per_epoch=100,
      epochs=30,
      validation_data=validation_generator,
      validation_steps=50,
      verbose=2)
```

```
Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
Epoch 1/30
 - 15s - loss: 0.5280 - acc: 0.7585 - val_loss: 0.3613 - val_acc: 0.
8330
Epoch 2/30
 - 13s - loss: 0.4669 - acc: 0.7940 - val_loss: 0.4898 - val_acc: 0.
8550
Epoch 3/30
 - 13s - loss: 0.4325 - acc: 0.8035 - val_loss: 0.4186 - val_acc: 0.
8750
Epoch 4/30
 - 13s - loss: 0.4049 - acc: 0.8270 - val_loss: 0.3709 - val_acc: 0.
8740
Epoch 5/30
 - 13s - loss: 0.3776 - acc: 0.8345 - val_loss: 0.2617 - val_acc: 0.
8830
Epoch 6/30
 - 13s - loss: 0.3677 - acc: 0.8375 - val_loss: 0.3126 - val_acc: 0.
8860
Epoch 7/30
 - 13s - loss: 0.3695 - acc: 0.8475 - val_loss: 0.2016 - val_acc: 0.
8900
Epoch 8/30
 - 13s - loss: 0.3644 - acc: 0.8365 - val_loss: 0.1850 - val_acc: 0.
8880
Epoch 9/30
 - 13s - loss: 0.3389 - acc: 0.8540 - val_loss: 0.3669 - val_acc: 0.
8910
Epoch 10/30
 - 13s - loss: 0.3284 - acc: 0.8585 - val_loss: 0.2310 - val_acc: 0.
8900
Epoch 11/30
 - 13s - loss: 0.3248 - acc: 0.8690 - val_loss: 0.3101 - val_acc: 0.
8920
Epoch 12/30
 - 13s - loss: 0.3339 - acc: 0.8485 - val_loss: 0.2047 - val_acc: 0.
8870
Epoch 13/30
 - 13s - loss: 0.3259 - acc: 0.8555 - val_loss: 0.2909 - val_acc: 0.
8910
Epoch 14/30
 - 13s - loss: 0.3193 - acc: 0.8595 - val_loss: 0.4800 - val_acc: 0.
8930
Epoch 15/30
 - 13s - loss: 0.3165 - acc: 0.8645 - val_loss: 0.1500 - val_acc: 0.
8930
Epoch 16/30
 - 13s - loss: 0.3146 - acc: 0.8630 - val_loss: 0.5308 - val_acc: 0.
8950
Epoch 17/30
 - 13s - loss: 0.3198 - acc: 0.8540 - val_loss: 0.2278 - val_acc: 0.
9030
Epoch 18/30
 - 13s - loss: 0.2957 - acc: 0.8735 - val_loss: 0.2524 - val_acc: 0.
8990
Epoch 19/30
 - 13s - loss: 0.3068 - acc: 0.8705 - val_loss: 0.1882 - val_acc: 0.
9000
Epoch 20/30
 - 13s - loss: 0.2989 - acc: 0.8745 - val_loss: 0.5603 - val_acc: 0.
```

```
9000
Epoch 21/30
 - 13s - loss: 0.2923 - acc: 0.8735 - val_loss: 0.1127 - val_acc: 0.
9000
Epoch 22/30
 - 13s - loss: 0.3026 - acc: 0.8720 - val_loss: 0.3073 - val_acc: 0.
8980
Epoch 23/30
 - 13s - loss: 0.2982 - acc: 0.8690 - val_loss: 0.3693 - val_acc: 0.
8890
Epoch 24/30
 - 13s - loss: 0.3063 - acc: 0.8620 - val_loss: 0.3826 - val_acc: 0.
9020
Epoch 25/30
 - 13s - loss: 0.3006 - acc: 0.8705 - val_loss: 0.5228 - val_acc: 0.
8990
Epoch 26/30
 - 13s - loss: 0.3095 - acc: 0.8670 - val_loss: 0.1852 - val_acc: 0.
9000
Epoch 27/30
 - 13s - loss: 0.2905 - acc: 0.8700 - val_loss: 0.1448 - val_acc: 0.
8990
Epoch 28/30
 - 13s - loss: 0.2882 - acc: 0.8845 - val_loss: 0.2543 - val_acc: 0.
9100
Epoch 29/30
 - 13s - loss: 0.2814 - acc: 0.8785 - val_loss: 0.0919 - val_acc: 0.
9030
Epoch 30/30
 - 13s - loss: 0.2941 - acc: 0.8735 - val_loss: 0.4608 - val_acc: 0.
9010
```

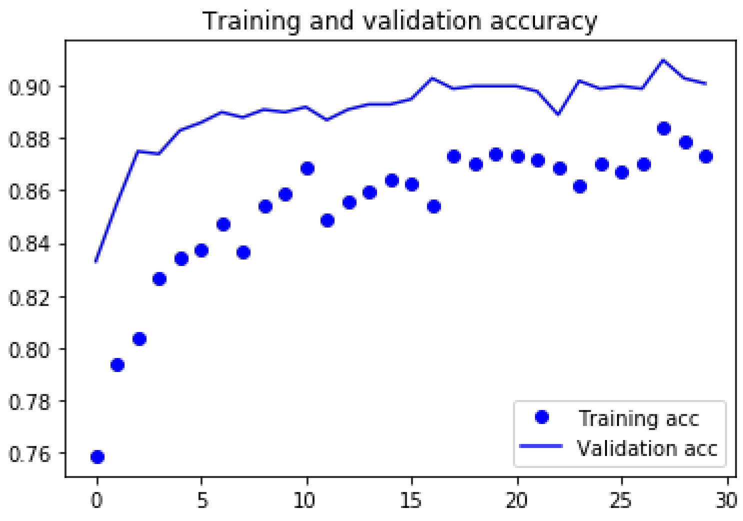
In [20]:

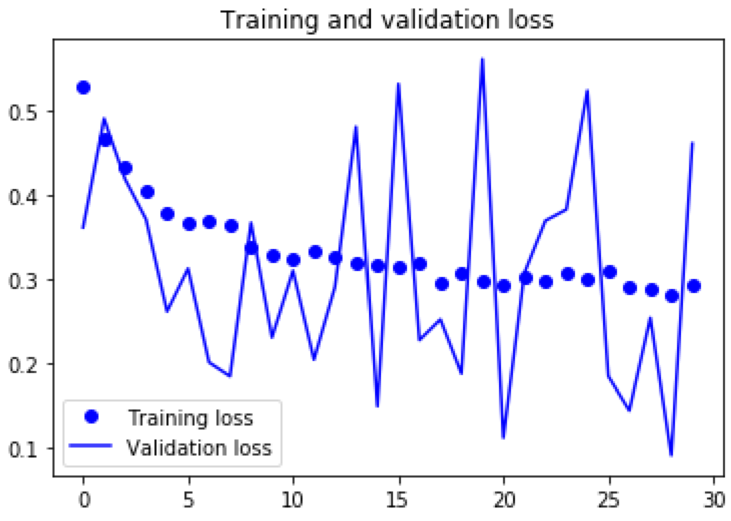
```
model.save('cats_and_dogs_small_4.h5')
```

Let's plot our results again:

In [21]:

```
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(len(acc))

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```

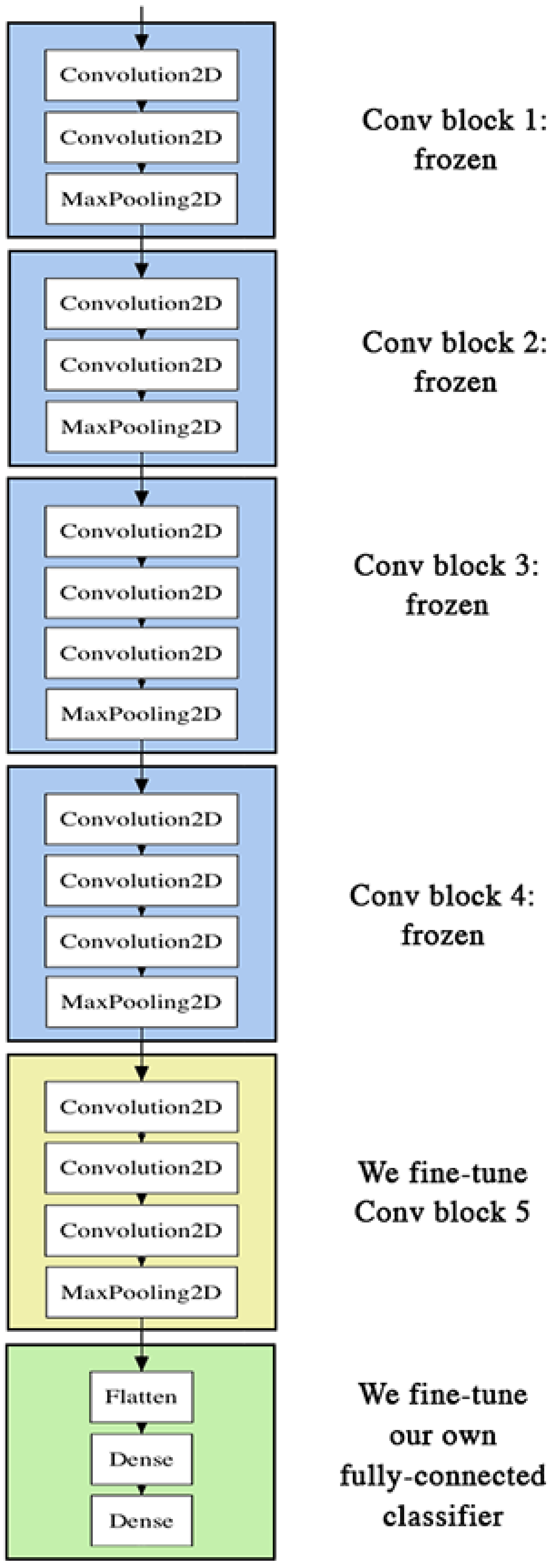




As you can see, we reach a validation accuracy of about 96%. This is much better than our small convnet trained from scratch.

Fine-tuning

Another widely used technique for model reuse, complementary to feature extraction, is *fine-tuning*. Fine-tuning consists in unfreezing a few of the top layers of a frozen model base used for feature extraction, and jointly training both the newly added part of the model (in our case, the fully-connected classifier) and these top layers. This is called "fine-tuning" because it slightly adjusts the more abstract representations of the model being reused, in order to make them more relevant for the problem at hand.


Convolution2D
Convolution2D
MaxPooling2D
Conv block 1: frozen
Convolution2D
Convolution2D
MaxPooling2D
Conv block 2: frozen
Convolution2D
Convolution2D
Convolution2D
MaxPooling2D
Conv block 3: frozen
Convolution2D
Convolution2D
Convolution2D
MaxPooling2D
Conv block 4: frozen
Convolution2D
Convolution2D
Convolution2D
MaxPooling2D
We fine-tune Conv block 5
Flatten
Dense
Dense
We fine-tune our own fully-connected classifier

We have stated before that it was necessary to freeze the convolution base of VGG16 in order to be able to train a randomly initialized classifier on top. For the same reason, it is only possible to fine-tune the top layers of the convolutional base once the classifier on top has already been trained. If the classified wasn't already trained, then the error signal propagating through the network during training would be too large, and the representations previously learned by the layers being fine-tuned would be destroyed. Thus the steps for fine-tuning a network are as follow:

- 1) Add your custom network on top of an already trained base network.
- 2) Freeze the base network.
- 3) Train the part you added.
- 4) Unfreeze some layers in the base network.
- 5) Jointly train both these layers and the part you added.

We have already completed the first 3 steps when doing feature extraction. Let's proceed with the 4th step: we will unfreeze our `conv_base` , and then freeze individual layers inside of it.

As a reminder, this is what our convolutional base looks like:

In [22]:

```
conv_base.summary()
```

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         (None, 150, 150, 3)       0         
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 150, 150, 64)      1792      
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 150, 150, 64)      36928     
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 75, 75, 64)        0         
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 75, 75, 128)       73856     
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 75, 75, 128)       147584    
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 37, 37, 128)       0         
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 37, 37, 256)       295168    
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 37, 37, 256)       590080    
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 37, 37, 256)       590080    
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 18, 18, 256)       0         
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 18, 18, 512)       1180160   
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 18, 18, 512)       2359808   
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 18, 18, 512)       2359808   
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 9, 9, 512)         0         
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 9, 9, 512)         2359808   
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 9, 9, 512)         2359808   
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 9, 9, 512)         2359808   
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 4, 4, 512)         0         
=================================================================
Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688
_________________________________________________________________
```

We will fine-tune the last 3 convolutional layers, which means that all layers up until `block4_pool` should be frozen, and the layers `block5_conv1` , `block5_conv2` and `block5_conv3` should be trainable.

Why not fine-tune more layers? Why not fine-tune the entire convolutional base? We could. However, we need to consider that:

- Earlier layers in the convolutional base encode more generic, reusable features, while layers higher up encode more specialized features. It is more useful to fine-tune the more specialized features, as these are the ones that need to be repurposed on our new problem. There would be fast-decreasing returns in fine-tuning lower layers.
- The more parameters we are training, the more we are at risk of overfitting. The convolutional base has 15M parameters, so it would be risky to attempt to train it on our small dataset.

Thus, in our situation, it is a good strategy to only fine-tune the top 2 to 3 layers in the convolutional base.

Let's set this up, starting from where we left off in the previous example:

In [23]:

```
conv_base.trainable = True

set_trainable = False
for layer in conv_base.layers:
    if layer.name == 'block5_conv1':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
```

Now we can start fine-tuning our network. We will do this with the RMSprop optimizer, using a very low learning rate. The reason for using a low learning rate is that we want to limit the magnitude of the modifications we make to the representations of the 3 layers that we are fine-tuning. Updates that are too large may harm these representations.

Now let's proceed with fine-tuning:

In [25]:

```
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-5),
              metrics=['acc'])

history = model.fit_generator(
      train_generator,
      steps_per_epoch=100,
      epochs=100,
      validation_data=validation_generator,
      validation_steps=50)
```

```
Epoch 1/100
100/100 [==============================] - 15s 147ms/step - loss: 0.
2788 - acc: 0.8855 - val_loss: 0.5837 - val_acc: 0.8970
Epoch 2/100
100/100 [==============================] - 13s 135ms/step - loss: 0.
2567 - acc: 0.8925 - val_loss: 0.2119 - val_acc: 0.9140
Epoch 3/100
100/100 [==============================] - 14s 135ms/step - loss: 0.
2283 - acc: 0.9045 - val_loss: 0.3724 - val_acc: 0.9250
Epoch 4/100
100/100 [==============================] - 13s 132ms/step - loss: 0.
2166 - acc: 0.9110 - val_loss: 0.3520 - val_acc: 0.9120
Epoch 5/100
100/100 [==============================] - 13s 133ms/step - loss: 0.
2002 - acc: 0.9150 - val_loss: 0.1777 - val_acc: 0.9280
Epoch 6/100
100/100 [==============================] - 13s 133ms/step - loss: 0.
2021 - acc: 0.9210 - val_loss: 0.1189 - val_acc: 0.9320
Epoch 7/100
100/100 [==============================] - 13s 134ms/step - loss: 0.
1834 - acc: 0.9250 - val_loss: 0.1751 - val_acc: 0.9260
Epoch 8/100
100/100 [==============================] - 13s 135ms/step - loss: 0.
1648 - acc: 0.9300 - val_loss: 0.1195 - val_acc: 0.9260
Epoch 9/100
100/100 [==============================] - 14s 135ms/step - loss: 0.
1611 - acc: 0.9385 - val_loss: 0.0389 - val_acc: 0.9380
Epoch 10/100
100/100 [==============================] - 13s 132ms/step - loss: 0.
1492 - acc: 0.9440 - val_loss: 0.1999 - val_acc: 0.9180
Epoch 11/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
1336 - acc: 0.9445 - val_loss: 0.0474 - val_acc: 0.9340
Epoch 12/100
100/100 [==============================] - 13s 133ms/step - loss: 0.
1520 - acc: 0.9400 - val_loss: 0.2605 - val_acc: 0.9250
Epoch 13/100
100/100 [==============================] - 13s 132ms/step - loss: 0.
1241 - acc: 0.9515 - val_loss: 0.0433 - val_acc: 0.9430
Epoch 14/100
100/100 [==============================] - 13s 134ms/step - loss: 0.
1294 - acc: 0.9460 - val_loss: 0.0217 - val_acc: 0.9210
Epoch 15/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
1225 - acc: 0.9505 - val_loss: 0.3547 - val_acc: 0.9330
Epoch 16/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
1138 - acc: 0.9580 - val_loss: 0.0057 - val_acc: 0.9310
Epoch 17/100
100/100 [==============================] - 13s 133ms/step - loss: 0.
1087 - acc: 0.9555 - val_loss: 0.5546 - val_acc: 0.9370
Epoch 18/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
1180 - acc: 0.9570 - val_loss: 0.1064 - val_acc: 0.9260
Epoch 19/100
100/100 [==============================] - 13s 132ms/step - loss: 0.
1050 - acc: 0.9575 - val_loss: 0.0312 - val_acc: 0.9430
Epoch 20/100
100/100 [==============================] - 13s 134ms/step - loss: 0.
1039 - acc: 0.9555 - val_loss: 0.0123 - val_acc: 0.9350
Epoch 21/100
```

```
100/100 [==============================] - 13s 133ms/step - loss: 0.
0880 - acc: 0.9650 - val_loss: 0.1480 - val_acc: 0.9210
Epoch 22/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0928 - acc: 0.9675 - val_loss: 0.2415 - val_acc: 0.9330
Epoch 23/100
100/100 [==============================] - 13s 130ms/step - loss: 0.
0836 - acc: 0.9670 - val_loss: 0.1546 - val_acc: 0.9380
Epoch 24/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0921 - acc: 0.9620 - val_loss: 0.0183 - val_acc: 0.9380
Epoch 25/100
100/100 [==============================] - 13s 132ms/step - loss: 0.
0832 - acc: 0.9690 - val_loss: 0.0765 - val_acc: 0.9390
Epoch 26/100
100/100 [==============================] - 13s 135ms/step - loss: 0.
0756 - acc: 0.9700 - val_loss: 0.3352 - val_acc: 0.9380
Epoch 27/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0782 - acc: 0.9695 - val_loss: 0.1515 - val_acc: 0.9300
Epoch 28/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0823 - acc: 0.9630 - val_loss: 0.3366 - val_acc: 0.9350
Epoch 29/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0664 - acc: 0.9710 - val_loss: 0.0969 - val_acc: 0.9340
Epoch 30/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0598 - acc: 0.9790 - val_loss: 0.0491 - val_acc: 0.9330
Epoch 31/100
100/100 [==============================] - 13s 132ms/step - loss: 0.
0680 - acc: 0.9755 - val_loss: 0.0718 - val_acc: 0.9290
Epoch 32/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0639 - acc: 0.9780 - val_loss: 0.5193 - val_acc: 0.9340
Epoch 33/100
100/100 [==============================] - 13s 133ms/step - loss: 0.
0651 - acc: 0.9755 - val_loss: 0.2060 - val_acc: 0.9340
Epoch 34/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0598 - acc: 0.9775 - val_loss: 0.2974 - val_acc: 0.9350
Epoch 35/100
100/100 [==============================] - 13s 130ms/step - loss: 0.
0614 - acc: 0.9770 - val_loss: 0.5580 - val_acc: 0.9370
Epoch 36/100
100/100 [==============================] - 13s 130ms/step - loss: 0.
0549 - acc: 0.9830 - val_loss: 0.1997 - val_acc: 0.9230
Epoch 37/100
100/100 [==============================] - 13s 133ms/step - loss: 0.
0626 - acc: 0.9765 - val_loss: 0.2634 - val_acc: 0.9330
Epoch 38/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0601 - acc: 0.9770 - val_loss: 0.3757 - val_acc: 0.9310
Epoch 39/100
100/100 [==============================] - 13s 132ms/step - loss: 0.
0534 - acc: 0.9815 - val_loss: 0.6302 - val_acc: 0.9330
Epoch 40/100
100/100 [==============================] - 13s 130ms/step - loss: 0.
0516 - acc: 0.9830 - val_loss: 0.0150 - val_acc: 0.9310
Epoch 41/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
```

```
0448 - acc: 0.9840 - val_loss: 0.2177 - val_acc: 0.9290
Epoch 42/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0498 - acc: 0.9785 - val_loss: 0.0384 - val_acc: 0.9400
Epoch 43/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0519 - acc: 0.9835 - val_loss: 0.1531 - val_acc: 0.9360
Epoch 44/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0469 - acc: 0.9815 - val_loss: 0.0874 - val_acc: 0.9350
Epoch 45/100
100/100 [==============================] - 13s 132ms/step - loss: 0.
0513 - acc: 0.9820 - val_loss: 0.1744 - val_acc: 0.9350
Epoch 46/100
100/100 [==============================] - 13s 130ms/step - loss: 0.
0444 - acc: 0.9845 - val_loss: 0.0450 - val_acc: 0.9350
Epoch 47/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0427 - acc: 0.9855 - val_loss: 0.0653 - val_acc: 0.9340
Epoch 48/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0338 - acc: 0.9870 - val_loss: 0.0047 - val_acc: 0.9320
Epoch 49/100
100/100 [==============================] - 13s 133ms/step - loss: 0.
0404 - acc: 0.9855 - val_loss: 0.5494 - val_acc: 0.9340
Epoch 50/100
100/100 [==============================] - 13s 134ms/step - loss: 0.
0511 - acc: 0.9830 - val_loss: 0.5440 - val_acc: 0.9330
Epoch 51/100
100/100 [==============================] - 13s 134ms/step - loss: 0.
0386 - acc: 0.9850 - val_loss: 0.0096 - val_acc: 0.9350
Epoch 52/100
100/100 [==============================] - 13s 130ms/step - loss: 0.
0432 - acc: 0.9850 - val_loss: 0.5707 - val_acc: 0.9320
Epoch 53/100
100/100 [==============================] - 13s 133ms/step - loss: 0.
0409 - acc: 0.9845 - val_loss: 0.6232 - val_acc: 0.9430
Epoch 54/100
100/100 [==============================] - 13s 132ms/step - loss: 0.
0292 - acc: 0.9910 - val_loss: 0.0637 - val_acc: 0.9410
Epoch 55/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0380 - acc: 0.9850 - val_loss: 0.4841 - val_acc: 0.9260
Epoch 56/100
100/100 [==============================] - 13s 132ms/step - loss: 0.
0345 - acc: 0.9855 - val_loss: 0.1029 - val_acc: 0.9360
Epoch 57/100
100/100 [==============================] - 13s 133ms/step - loss: 0.
0333 - acc: 0.9890 - val_loss: 0.4289 - val_acc: 0.9350
Epoch 58/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0300 - acc: 0.9905 - val_loss: 0.0418 - val_acc: 0.9300
Epoch 59/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0318 - acc: 0.9880 - val_loss: 1.1686 - val_acc: 0.9320
Epoch 60/100
100/100 [==============================] - 13s 130ms/step - loss: 0.
0298 - acc: 0.9885 - val_loss: 0.2978 - val_acc: 0.9390
Epoch 61/100
100/100 [==============================] - 13s 130ms/step - loss: 0.
0374 - acc: 0.9880 - val_loss: 0.1631 - val_acc: 0.9320
```

```
Epoch 62/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0299 - acc: 0.9900 - val_loss: 1.0380 - val_acc: 0.9280
Epoch 63/100
100/100 [==============================] - 13s 132ms/step - loss: 0.
0281 - acc: 0.9895 - val_loss: 0.0723 - val_acc: 0.9320
Epoch 64/100
100/100 [==============================] - 13s 130ms/step - loss: 0.
0312 - acc: 0.9885 - val_loss: 0.2605 - val_acc: 0.9380
Epoch 65/100
100/100 [==============================] - 13s 129ms/step - loss: 0.
0280 - acc: 0.9900 - val_loss: 0.7029 - val_acc: 0.9360
Epoch 66/100
100/100 [==============================] - 13s 130ms/step - loss: 0.
0283 - acc: 0.9890 - val_loss: 0.2882 - val_acc: 0.9410
Epoch 67/100
100/100 [==============================] - 13s 132ms/step - loss: 0.
0307 - acc: 0.9915 - val_loss: 0.0019 - val_acc: 0.9360
Epoch 68/100
100/100 [==============================] - 13s 130ms/step - loss: 0.
0375 - acc: 0.9865 - val_loss: 4.3191e-05 - val_acc: 0.9350
Epoch 69/100
100/100 [==============================] - 13s 130ms/step - loss: 0.
0281 - acc: 0.9905 - val_loss: 0.0910 - val_acc: 0.9380
Epoch 70/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0266 - acc: 0.9905 - val_loss: 0.2402 - val_acc: 0.9310
Epoch 71/100
100/100 [==============================] - 13s 130ms/step - loss: 0.
0260 - acc: 0.9915 - val_loss: 0.1193 - val_acc: 0.9340
Epoch 72/100
100/100 [==============================] - 13s 133ms/step - loss: 0.
0243 - acc: 0.9895 - val_loss: 0.0417 - val_acc: 0.9320
Epoch 73/100
100/100 [==============================] - 13s 132ms/step - loss: 0.
0267 - acc: 0.9925 - val_loss: 0.2512 - val_acc: 0.9300
Epoch 74/100
100/100 [==============================] - 13s 132ms/step - loss: 0.
0204 - acc: 0.9930 - val_loss: 0.0600 - val_acc: 0.9400
Epoch 75/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0204 - acc: 0.9905 - val_loss: 7.3643e-05 - val_acc: 0.9310
Epoch 76/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0213 - acc: 0.9905 - val_loss: 0.0038 - val_acc: 0.9370
Epoch 77/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0232 - acc: 0.9915 - val_loss: 0.2765 - val_acc: 0.9230
Epoch 78/100
100/100 [==============================] - 13s 129ms/step - loss: 0.
0232 - acc: 0.9920 - val_loss: 0.2414 - val_acc: 0.9310
Epoch 79/100
100/100 [==============================] - 13s 130ms/step - loss: 0.
0259 - acc: 0.9905 - val_loss: 0.0157 - val_acc: 0.9360
Epoch 80/100
100/100 [==============================] - 13s 130ms/step - loss: 0.
0219 - acc: 0.9930 - val_loss: 0.0504 - val_acc: 0.9370
Epoch 81/100
100/100 [==============================] - 13s 130ms/step - loss: 0.
0253 - acc: 0.9915 - val_loss: 0.1569 - val_acc: 0.9330
Epoch 82/100
```

```
100/100 [==============================] - 13s 132ms/step - loss: 0.
0294 - acc: 0.9895 - val_loss: 0.0316 - val_acc: 0.9380
Epoch 83/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0171 - acc: 0.9930 - val_loss: 0.1017 - val_acc: 0.9400
Epoch 84/100
100/100 [==============================] - 13s 130ms/step - loss: 0.
0203 - acc: 0.9915 - val_loss: 0.2301 - val_acc: 0.9320
Epoch 85/100
100/100 [==============================] - 13s 130ms/step - loss: 0.
0309 - acc: 0.9915 - val_loss: 0.3792 - val_acc: 0.9300
Epoch 86/100
100/100 [==============================] - 13s 129ms/step - loss: 0.
0180 - acc: 0.9945 - val_loss: 0.1152 - val_acc: 0.9290
Epoch 87/100
100/100 [==============================] - 13s 130ms/step - loss: 0.
0240 - acc: 0.9920 - val_loss: 1.0074 - val_acc: 0.9430
Epoch 88/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0238 - acc: 0.9910 - val_loss: 0.1270 - val_acc: 0.9370
Epoch 89/100
100/100 [==============================] - 13s 130ms/step - loss: 0.
0102 - acc: 0.9975 - val_loss: 0.8237 - val_acc: 0.9320
Epoch 90/100
100/100 [==============================] - 13s 129ms/step - loss: 0.
0229 - acc: 0.9905 - val_loss: 0.1208 - val_acc: 0.9340
Epoch 91/100
100/100 [==============================] - 13s 130ms/step - loss: 0.
0180 - acc: 0.9950 - val_loss: 0.0989 - val_acc: 0.9420
Epoch 92/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0178 - acc: 0.9950 - val_loss: 0.0316 - val_acc: 0.9340
Epoch 93/100
100/100 [==============================] - 13s 130ms/step - loss: 0.
0203 - acc: 0.9930 - val_loss: 0.7374 - val_acc: 0.9380
Epoch 94/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0284 - acc: 0.9905 - val_loss: 0.1067 - val_acc: 0.9400
Epoch 95/100
100/100 [==============================] - 13s 129ms/step - loss: 0.
0243 - acc: 0.9925 - val_loss: 0.2350 - val_acc: 0.9350
Epoch 96/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0253 - acc: 0.9900 - val_loss: 0.0064 - val_acc: 0.9330
Epoch 97/100
100/100 [==============================] - 13s 132ms/step - loss: 0.
0203 - acc: 0.9925 - val_loss: 0.6750 - val_acc: 0.9390
Epoch 98/100
100/100 [==============================] - 13s 132ms/step - loss: 0.
0179 - acc: 0.9925 - val_loss: 0.3429 - val_acc: 0.9330
Epoch 99/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0182 - acc: 0.9930 - val_loss: 0.7694 - val_acc: 0.9190
Epoch 100/100
100/100 [==============================] - 13s 131ms/step - loss: 0.
0140 - acc: 0.9950 - val_loss: 0.1551 - val_acc: 0.9400
```

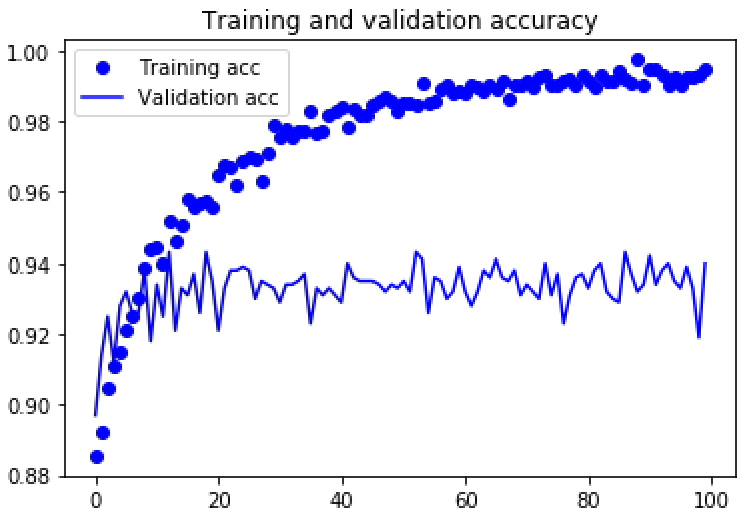
In [26]:

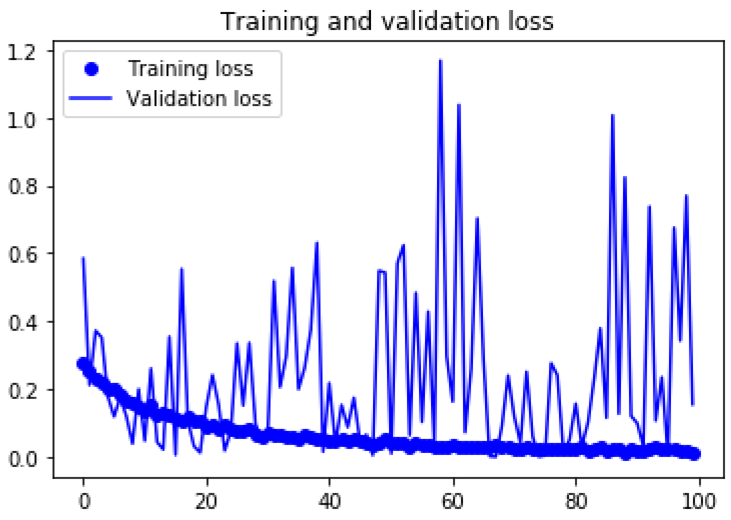
```
model.save('cats_and_dogs_small_5.h5')
```

Let's plot our results using the same plotting code as before:

In [27]:

```
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(len(acc))

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```

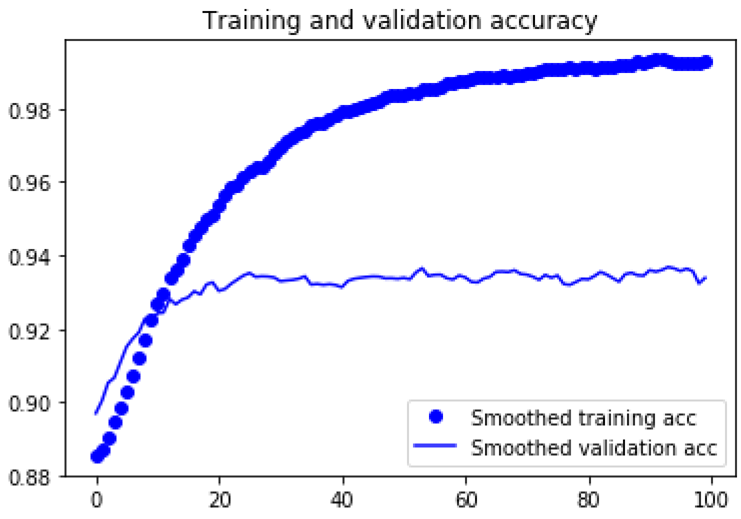


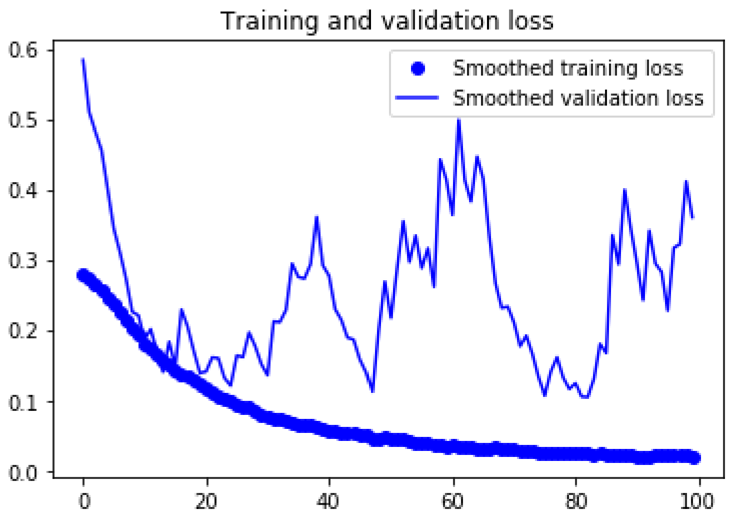


These curves look very noisy. To make them more readable, we can smooth them by replacing every loss and accuracy with exponential moving averages of these quantities. Here's a trivial utility function to do this:

In [28]:

```
def smooth_curve(points, factor=0.8):
    smoothed_points = []
    for point in points:
        if smoothed_points:
            previous = smoothed_points[-1]
            smoothed_points.append(previous * factor + point * (1 - factor))
        else:
            smoothed_points.append(point)
    return smoothed_points

plt.plot(epochs, smooth_curve(acc), 'bo', label='Smoothed training acc')
plt.plot(epochs, smooth_curve(val_acc), 'b', label='Smoothed validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, smooth_curve(loss), 'bo', label='Smoothed training loss')
plt.plot(epochs, smooth_curve(val_loss), 'b', label='Smoothed validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show();
```





These curves look much cleaner and more stable. We are seeing a nice 1% absolute improvement.

Note that the loss curve does not show any real improvement (in fact, it is deteriorating). You may wonder, how could accuracy improve if the loss isn't decreasing? The answer is simple: what we display is an average of pointwise loss values, but what actually matters for accuracy is the distribution of the loss values, not their average, since accuracy is the result of a binary thresholding of the class probability predicted by the model. The model may still be improving even if this isn't reflected in the average loss.

We can now finally evaluate this model on the test data:

In [29]:

```
test_generator = test_datagen.flow_from_directory(
        test_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')

test_loss, test_acc = model.evaluate_generator(test_generator, steps=50)
print('test acc:', test_acc)
```

```
Found 1000 images belonging to 2 classes.
test acc: 0.9399999976158142
```

Here we get a test accuracy of 97%. In the original Kaggle competition around this dataset, this would have been one of the top results. However, using modern deep learning techniques, we managed to reach this result using only a very small fraction of the training data available (about 10%). There is a huge difference between being able to train on 20,000 samples compared to 2,000 samples!

Take-aways: using convnets with small datasets

Here's what you should take away from the exercises of these past two sections:

- Convnets are the best type of machine learning models for computer vision tasks. It is possible to train one from scratch even on a very small dataset, with decent results.
- On a small dataset, overfitting will be the main issue. Data augmentation is a powerful way to fight overfitting when working with image data.
- It is easy to reuse an existing convnet on a new dataset, via feature extraction. This is a very valuable technique for working with small image datasets.
- As a complement to feature extraction, one may use fine-tuning, which adapts to a new problem some of the representations previously learned by an existing model. This pushes performance a bit further.

Now you have a solid set of tools for dealing with image classification problems, in particular with small datasets.