In [0]:

```
import keras
keras.__version__
```

```
Using TensorFlow backend.
```

Out[1]:

```
'2.2.4'
```

In [0]:

```
!git clone https://github.com/jlee1991/Python-Practice.git
```

```
Cloning into 'Python-Practice'...
remote: Enumerating objects: 37479, done.
remote: Counting objects: 100% (37479/37479), done.
remote: Compressing objects: 100% (37462/37462), done.
remote: Total 37517 (delta 21), reused 37473 (delta 15), pack-reused 3
8
Receiving objects: 100% (37517/37517), 868.28 MiB | 48.48 MiB/s, done.
Resolving deltas: 100% (26/26), done.
Checking out files: 100% (41561/41561), done.
```

Using a pre-trained convnet

This notebook contains the code sample found in Chapter 5, Section 3 of [Deep Learning with Python (https://www.manning.com/books/deep-learning-with-python?a_aid=keras&a_bid=76564dff)](https://www.manning.com/books/deep-learning-with-python?a_aid=keras&a_bid=76564dff). Note that the original text features far more content, in particular further explanations and figures: in this notebook, you will only find source code and related comments.

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.

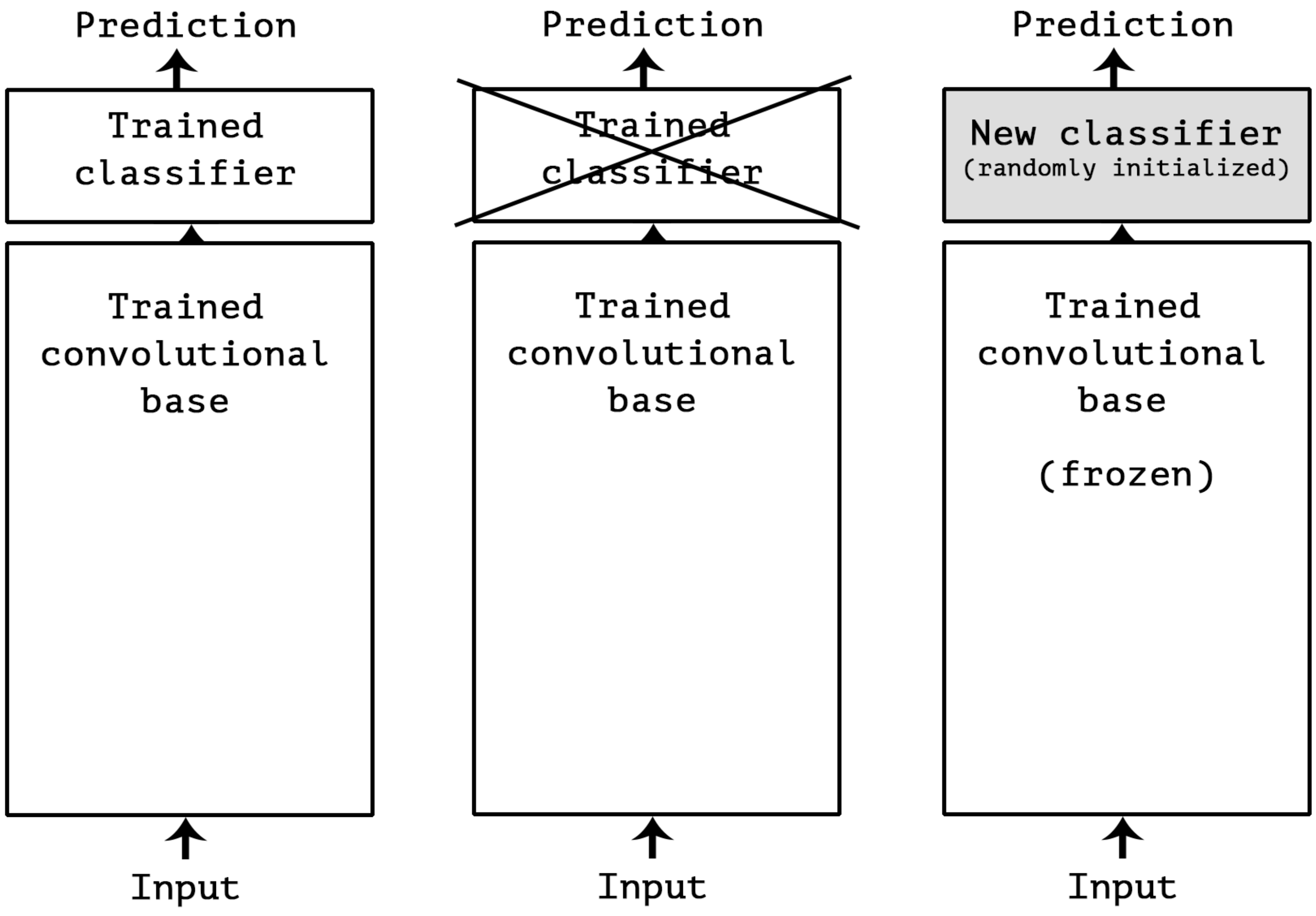
We will use the VGG16 architecture, developed by Karen Simonyan and Andrew Zisserman in 2014, a simple and widely used convnet architecture for ImageNet. Although it is a bit of an older model, far from the current state of the art and somewhat heavier than many other recent models, we chose it because 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.


Prediction
Trained classifier
Trained convolutional base
Input
Prediction
~~Trained classifier~~
Trained convolutional base
Input
Prediction
New classifier (randomly initialized)
Trained convolutional base (frozen)
Input

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 [0]:

```
from keras.applications import VGG16

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

```
WARNING: Logging before flag parsing goes to stderr.
W0714 18:26:22.276896 140336381278080 deprecation_wrapper.py:119] From
/usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backen
d.py:74: The name tf.get_default_graph is deprecated. Please use tf.co
mpat.v1.get_default_graph instead.

W0714 18:26:22.323606 140336381278080 deprecation_wrapper.py:119] From
/usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backen
d.py:517: The name tf.placeholder is deprecated. Please use tf.compat.
v1.placeholder instead.

W0714 18:26:22.332378 140336381278080 deprecation_wrapper.py:119] From
/usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backen
d.py:4138: The name tf.random_uniform is deprecated. Please use tf.ran
dom.uniform instead.

W0714 18:26:22.374543 140336381278080 deprecation_wrapper.py:119] From
/usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backen
d.py:3976: The name tf.nn.max_pool is deprecated. Please use tf.nn.max
_pool2d instead.


Downloading data from https://github.com/fchollet/deep-learning-models
/releases/download/v0.1/vgg16_weights_tf_dim_ordering_tf_kernels_notop
.h5 (https://github.com/fchollet/deep-learning-models/releases/downloa
d/v0.1/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5)
58892288/58889256 [==============================] - 1s 0us/step

W0714 18:26:23.388418 140336381278080 deprecation_wrapper.py:119] From
/usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backen
d.py:174: The name tf.get_default_session is deprecated. Please use tf
.compat.v1.get_default_session instead.

W0714 18:26:23.389605 140336381278080 deprecation_wrapper.py:119] From
/usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backen
d.py:181: The name tf.ConfigProto is deprecated. Please use tf.compat.
v1.ConfigProto instead.
```

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 [0]:

```
conv_base.summary()
```

```
_________________________________________________________________
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 [0]:

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

base_dir = '/content/Python-Practice/Harvard CSCI E-63/HW3/small'

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 [0]:

```python
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 [0]:

```
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=15,
                    batch_size=20,
                    validation_data=(validation_features, validation_labels))
```

```
W0714 20:32:01.684337 140336381278080 deprecation.py:506] From /usr/lo
cal/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py:34
45: calling dropout (from tensorflow.python.ops.nn_ops) with keep_prob
is deprecated and will be removed in a future version.
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `rate
= 1 - keep_prob`.
W0714 20:32:01.723428 140336381278080 deprecation_wrapper.py:119] From
/usr/local/lib/python3.6/dist-packages/keras/optimizers.py:790: The na
me tf.train.Optimizer is deprecated. Please use tf.compat.v1.train.Opt
imizer instead.

W0714 20:32:01.734540 140336381278080 deprecation.py:323] From /usr/lo
cal/lib/python3.6/dist-packages/tensorflow/python/ops/nn_impl.py:180:
add_dispatch_support.<locals>.wrapper (from tensorflow.python.ops.arra
y_ops) is deprecated and will be removed in a future version.
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where

Train on 2000 samples, validate on 1000 samples
Epoch 1/15
2000/2000 [==============================] - 2s 782us/step - loss: 0.6
276 - acc: 0.6465 - val_loss: 0.4576 - val_acc: 0.8180
Epoch 2/15
2000/2000 [==============================] - 1s 281us/step - loss: 0.4
393 - acc: 0.8000 - val_loss: 0.3713 - val_acc: 0.8460
Epoch 3/15
2000/2000 [==============================] - 1s 282us/step - loss: 0.3
596 - acc: 0.8445 - val_loss: 0.3257 - val_acc: 0.8720
Epoch 4/15
```

```
2000/2000 [==============================] - 1s 269us/step - loss: 0.3
147 - acc: 0.8685 - val_loss: 0.3055 - val_acc: 0.8780
Epoch 5/15
2000/2000 [==============================] - 1s 273us/step - loss: 0.2
911 - acc: 0.8770 - val_loss: 0.2853 - val_acc: 0.8880
Epoch 6/15
2000/2000 [==============================] - 1s 267us/step - loss: 0.2
671 - acc: 0.8965 - val_loss: 0.2757 - val_acc: 0.8880
Epoch 7/15
2000/2000 [==============================] - 1s 274us/step - loss: 0.2
502 - acc: 0.9020 - val_loss: 0.2655 - val_acc: 0.8960
Epoch 8/15
2000/2000 [==============================] - 1s 267us/step - loss: 0.2
308 - acc: 0.9105 - val_loss: 0.2590 - val_acc: 0.8960
Epoch 9/15
2000/2000 [==============================] - 1s 275us/step - loss: 0.2
194 - acc: 0.9140 - val_loss: 0.2579 - val_acc: 0.8930
Epoch 10/15
2000/2000 [==============================] - 1s 265us/step - loss: 0.2
074 - acc: 0.9255 - val_loss: 0.2518 - val_acc: 0.8990
Epoch 11/15
2000/2000 [==============================] - 1s 268us/step - loss: 0.1
979 - acc: 0.9240 - val_loss: 0.2492 - val_acc: 0.9020
Epoch 12/15
2000/2000 [==============================] - 1s 268us/step - loss: 0.1
842 - acc: 0.9305 - val_loss: 0.2504 - val_acc: 0.9010
Epoch 13/15
2000/2000 [==============================] - 1s 275us/step - loss: 0.1
841 - acc: 0.9315 - val_loss: 0.2487 - val_acc: 0.9040
Epoch 14/15
2000/2000 [==============================] - 1s 278us/step - loss: 0.1
744 - acc: 0.9340 - val_loss: 0.2408 - val_acc: 0.9020
Epoch 15/15
2000/2000 [==============================] - 1s 290us/step - loss: 0.1
641 - acc: 0.9440 - val_loss: 0.2432 - val_acc: 0.9020
```

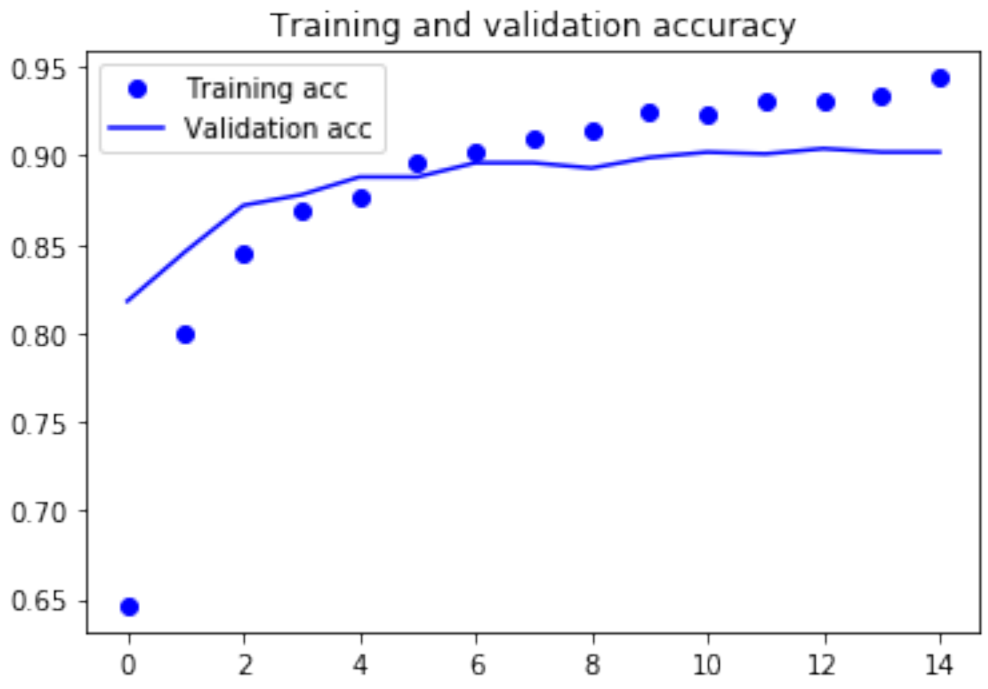
Training is very fast, since we only have to deal with two `Dense` layers -- an epoch takes less than one second even on CPU.

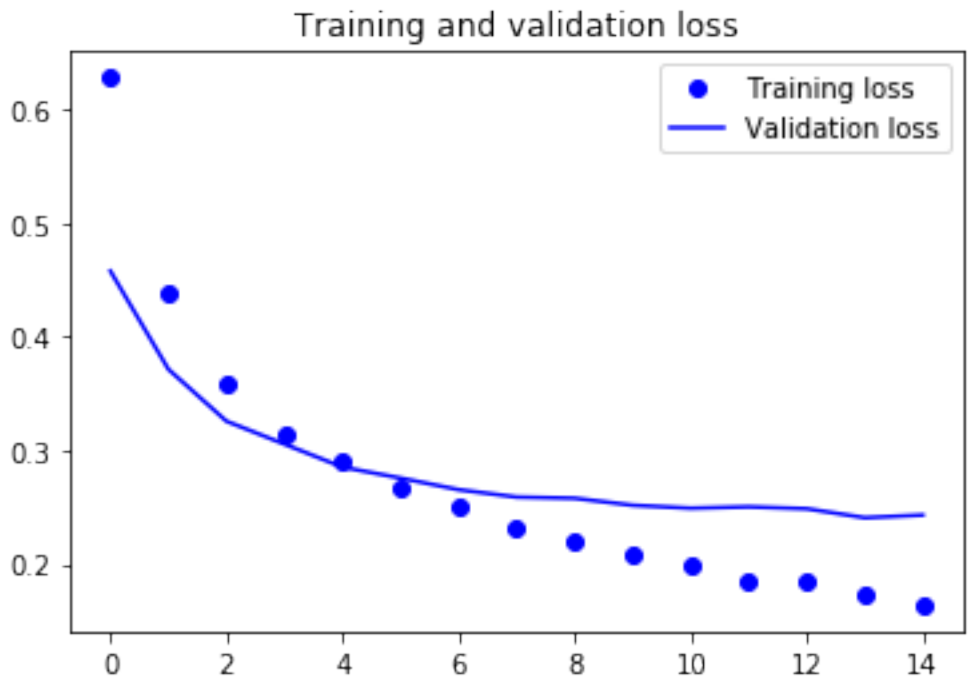
Let's take a look at the loss and accuracy curves during training:

In [0]:

```python
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 [0]:

```
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 [0]:

```
model.summary()
```

```
_________________________________________________________________
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 [0]:

```
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 base:
30
```

In [0]:

```
conv_base.trainable = False
```

In [0]:

```
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 base:
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 [0]:

```
from keras.preprocessing.image import ImageDataGenerator
from keras import optimizers

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=5,
      validation_data=validation_generator,
      validation_steps=50,
      verbose=2)
```

```
Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
Epoch 1/5
 - 18s - loss: 0.6046 - acc: 0.6850 - val_loss: 0.4646 - val_acc: 0.81
00
Epoch 2/5
 - 16s - loss: 0.4821 - acc: 0.7965 - val_loss: 0.3736 - val_acc: 0.85
80
Epoch 3/5
 - 16s - loss: 0.4434 - acc: 0.7965 - val_loss: 0.3696 - val_acc: 0.84
40
Epoch 4/5
 - 16s - loss: 0.4092 - acc: 0.8180 - val_loss: 0.3081 - val_acc: 0.88
50
Epoch 5/5
 - 16s - loss: 0.3812 - acc: 0.8280 - val_loss: 0.3221 - val_acc: 0.84
70
```

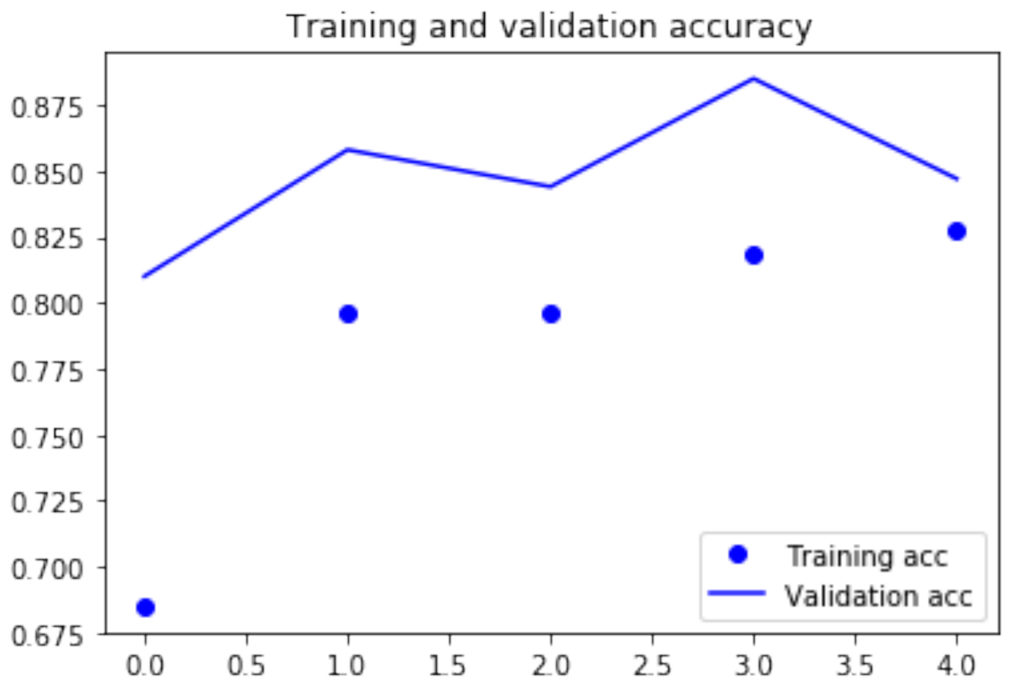
In [0]:

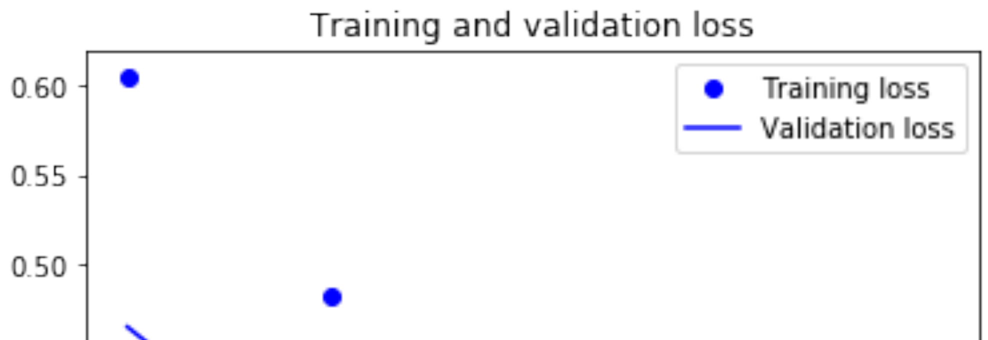
```
model.save('cats_and_dogs_small_3.h5')
```

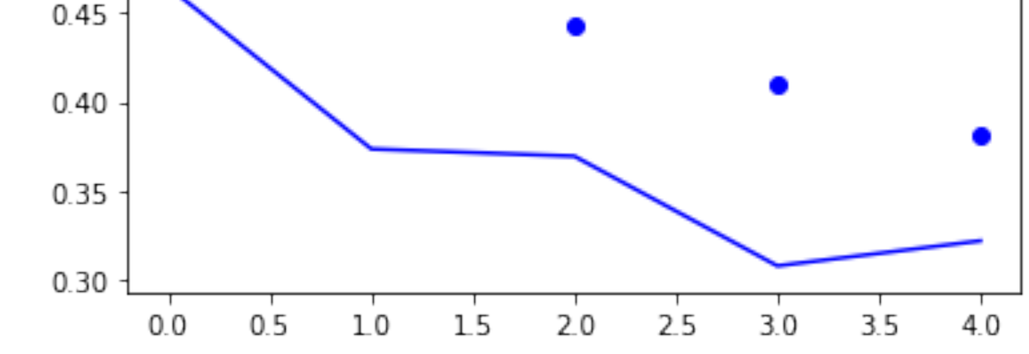
Let's plot our results again:

In [0]:

```
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()
```



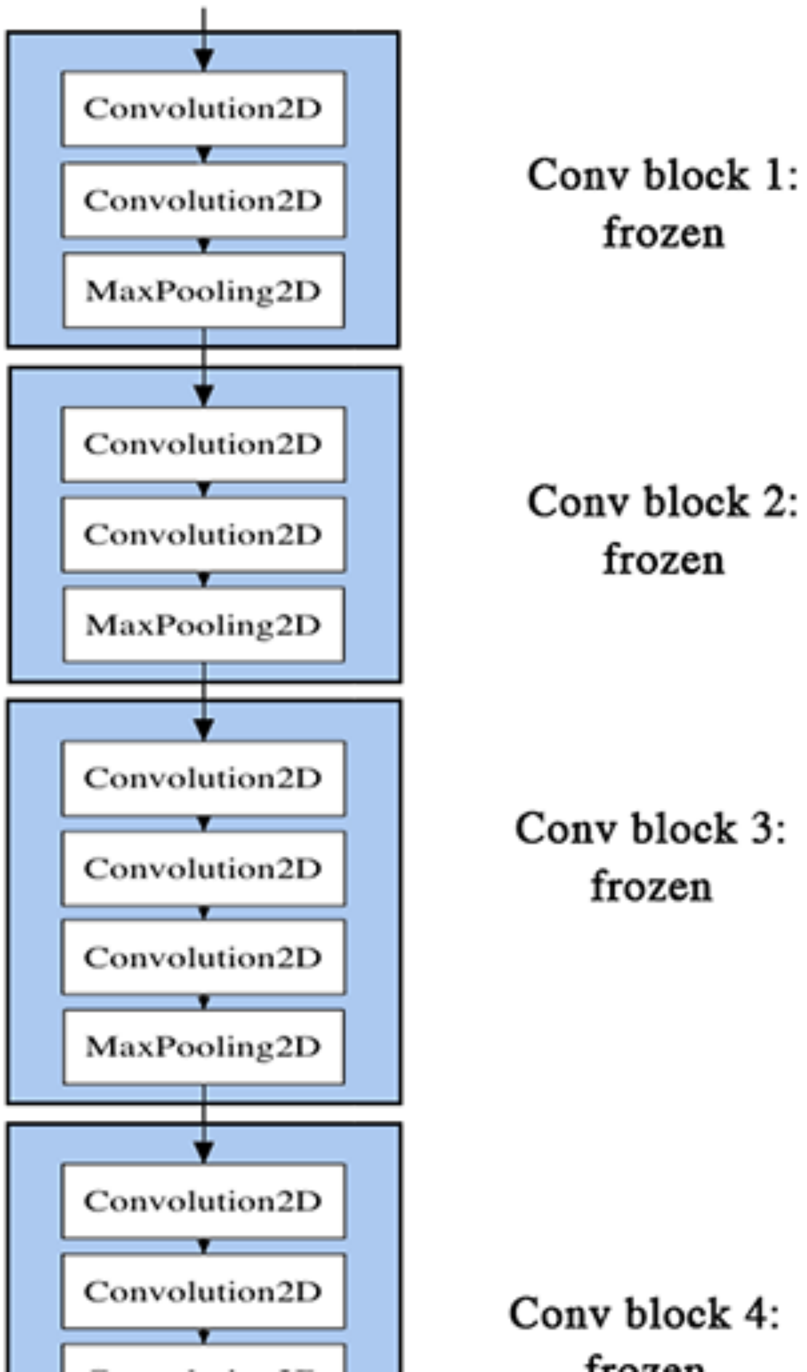


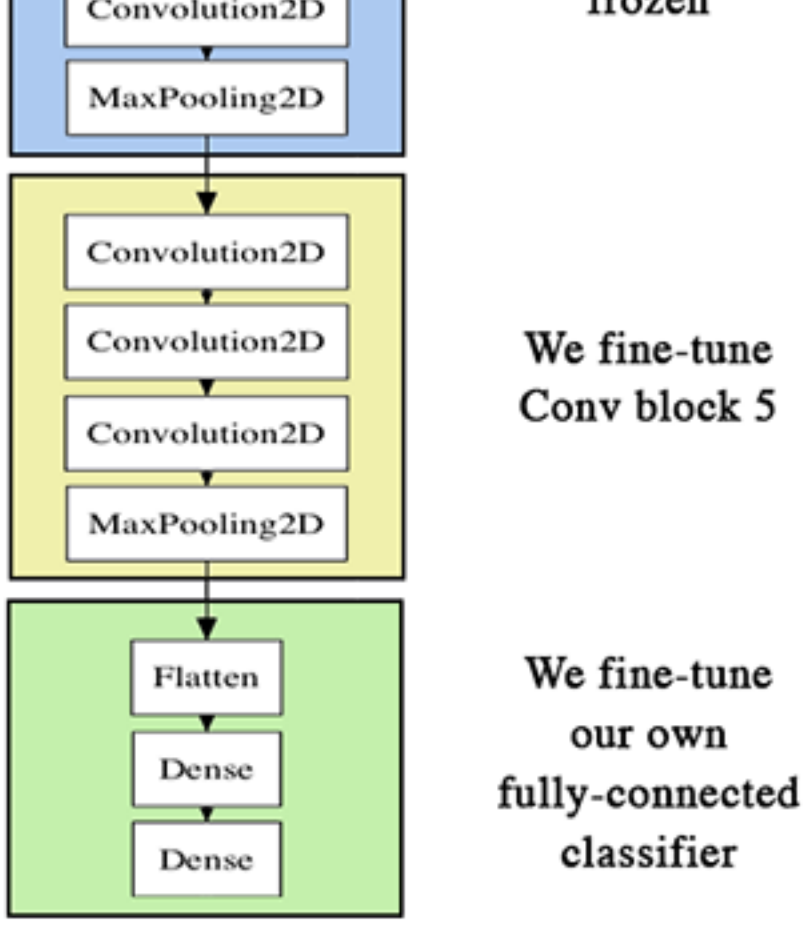


As you can see, we reach a validation accuracy of about 96%. This is much better than our small convnet trained from scratch.

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.





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 [0]:

```
conv_base.summary()
```

```
_________________________________________________________________
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 [0]:

```
conv_base.trainable = True

set_trainable = False
for layer in conv_base.layers:
    if layer.name == 'block5_conv3':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
```

In [0]:

```
model.summary()
```

```
_________________________________________________________________
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: 14,452,545
Trainable params: 2,097,665
Non-trainable params: 12,354,880
_________________________________________________________________

/usr/local/lib/python3.6/dist-packages/keras/engine/training.py:490: U
serWarning: Discrepancy between trainable weights and collected traina
ble weights, did you set `model.trainable` without calling `model.comp
ile` after ?
  'Discrepancy between trainable weights and collected trainable'
```

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 [0]:

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

history = model.fit_generator(
      train_generator,
      steps_per_epoch=100,
      epochs=20, #100 Epochs is still a lot even in Google Colab
      validation_data=validation_generator,
      validation_steps=50)
```

```
Epoch 1/20
100/100 [==============================] - 19s 186ms/step - loss: 0.29
18 - acc: 0.8710 - val_loss: 0.2112 - val_acc: 0.9100
Epoch 2/20
100/100 [==============================] - 17s 166ms/step - loss: 0.27
```

```
65 - acc: 0.8805 - val_loss: 0.2074 - val_acc: 0.9140
Epoch 3/20
100/100 [==============================] - 17s 167ms/step - loss: 0.25
23 - acc: 0.8900 - val_loss: 0.2367 - val_acc: 0.9070
Epoch 4/20
100/100 [==============================] - 17s 166ms/step - loss: 0.25
62 - acc: 0.8860 - val_loss: 0.2005 - val_acc: 0.9110
Epoch 5/20
100/100 [==============================] - 17s 168ms/step - loss: 0.24
74 - acc: 0.8930 - val_loss: 0.2298 - val_acc: 0.9100
Epoch 6/20
100/100 [==============================] - 17s 168ms/step - loss: 0.24
35 - acc: 0.8930 - val_loss: 0.2120 - val_acc: 0.9090
Epoch 7/20
100/100 [==============================] - 17s 169ms/step - loss: 0.26
34 - acc: 0.8900 - val_loss: 0.2071 - val_acc: 0.9190
Epoch 8/20
100/100 [==============================] - 17s 167ms/step - loss: 0.25
22 - acc: 0.8865 - val_loss: 0.2070 - val_acc: 0.9120
Epoch 9/20
100/100 [==============================] - 17s 167ms/step - loss: 0.22
96 - acc: 0.9020 - val_loss: 0.1871 - val_acc: 0.9240
Epoch 10/20
100/100 [==============================] - 17s 169ms/step - loss: 0.23
38 - acc: 0.9065 - val_loss: 0.2220 - val_acc: 0.9130
Epoch 11/20
100/100 [==============================] - 17s 168ms/step - loss: 0.23
54 - acc: 0.9030 - val_loss: 0.2145 - val_acc: 0.9120
Epoch 12/20
100/100 [==============================] - 17s 167ms/step - loss: 0.22
64 - acc: 0.9070 - val_loss: 0.1856 - val_acc: 0.9260
Epoch 13/20
100/100 [==============================] - 17s 167ms/step - loss: 0.21
75 - acc: 0.9050 - val_loss: 0.1937 - val_acc: 0.9200
Epoch 14/20
100/100 [==============================] - 17s 167ms/step - loss: 0.22
39 - acc: 0.9005 - val_loss: 0.2002 - val_acc: 0.9190
Epoch 15/20
100/100 [==============================] - 17s 168ms/step - loss: 0.23
72 - acc: 0.9000 - val_loss: 0.2189 - val_acc: 0.9090
Epoch 16/20
100/100 [==============================] - 17s 167ms/step - loss: 0.20
56 - acc: 0.9245 - val_loss: 0.2007 - val_acc: 0.9240
Epoch 17/20
100/100 [==============================] - 17s 167ms/step - loss: 0.20
49 - acc: 0.9195 - val_loss: 0.1994 - val_acc: 0.9200
Epoch 18/20
100/100 [==============================] - 17s 167ms/step - loss: 0.20
60 - acc: 0.9190 - val_loss: 0.2432 - val_acc: 0.8890
Epoch 19/20
100/100 [==============================] - 17s 167ms/step - loss: 0.19
38 - acc: 0.9225 - val_loss: 0.1845 - val_acc: 0.9250
Epoch 20/20
```

```
100/100 [==============================] - 17s 169ms/step - loss: 0.19
84 - acc: 0.9235 - val_loss: 0.1978 - val_acc: 0.9220
```

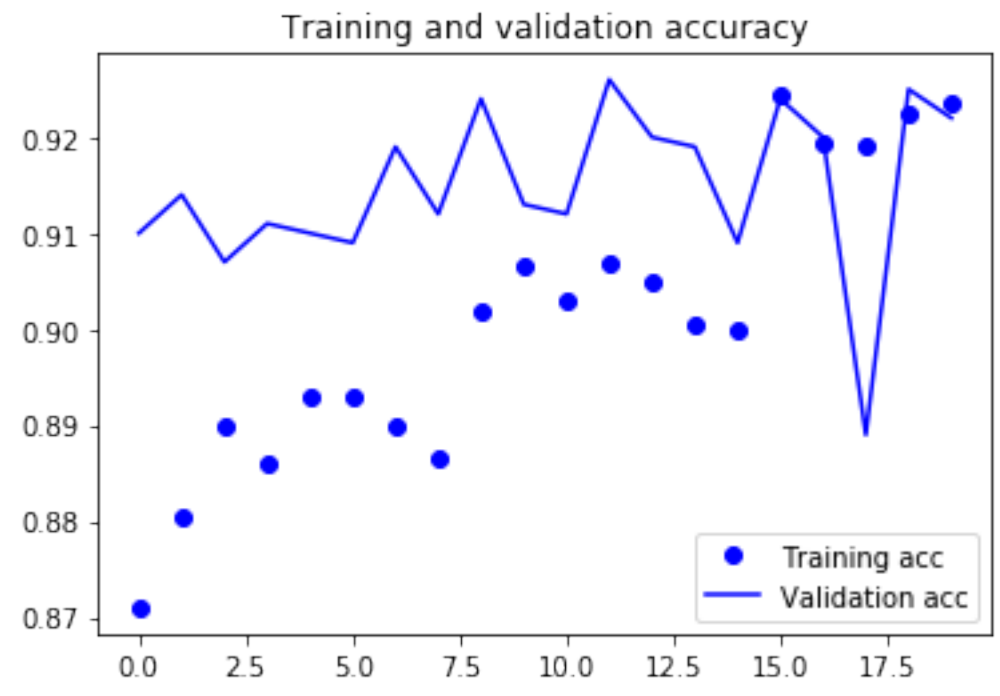
In [0]:

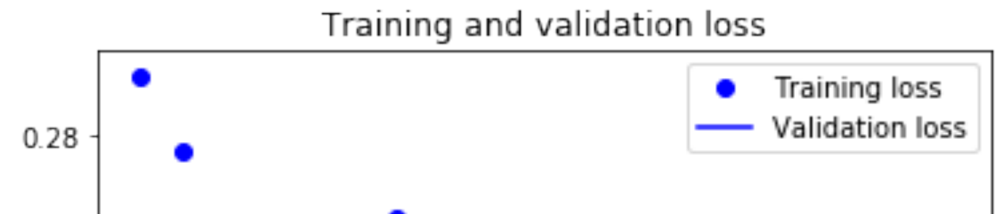
```
model.save('cats_and_dogs_small_4.h5')
```

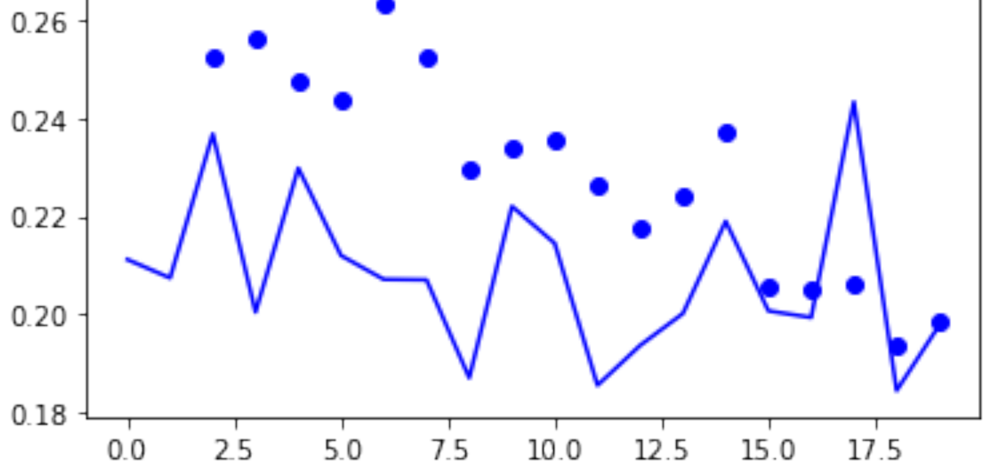
Let's plot our results using the same plotting code as before:

In [0]:

```
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()
```



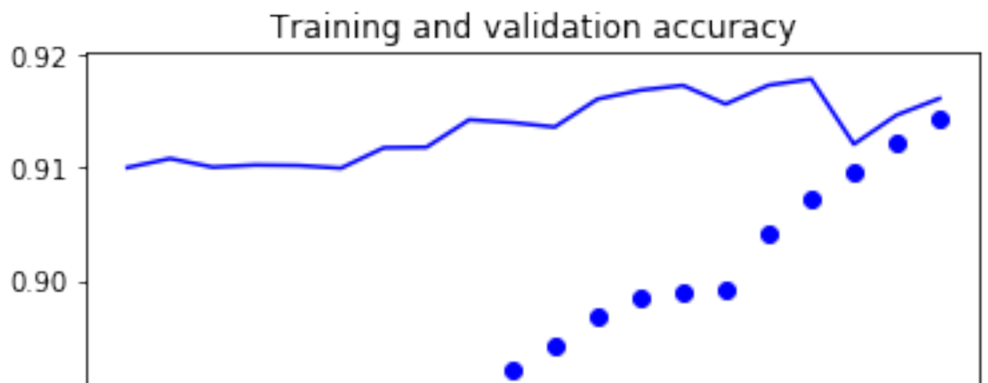


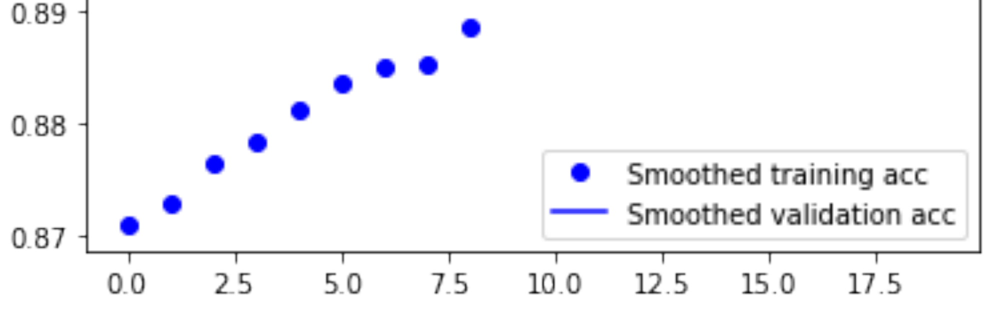


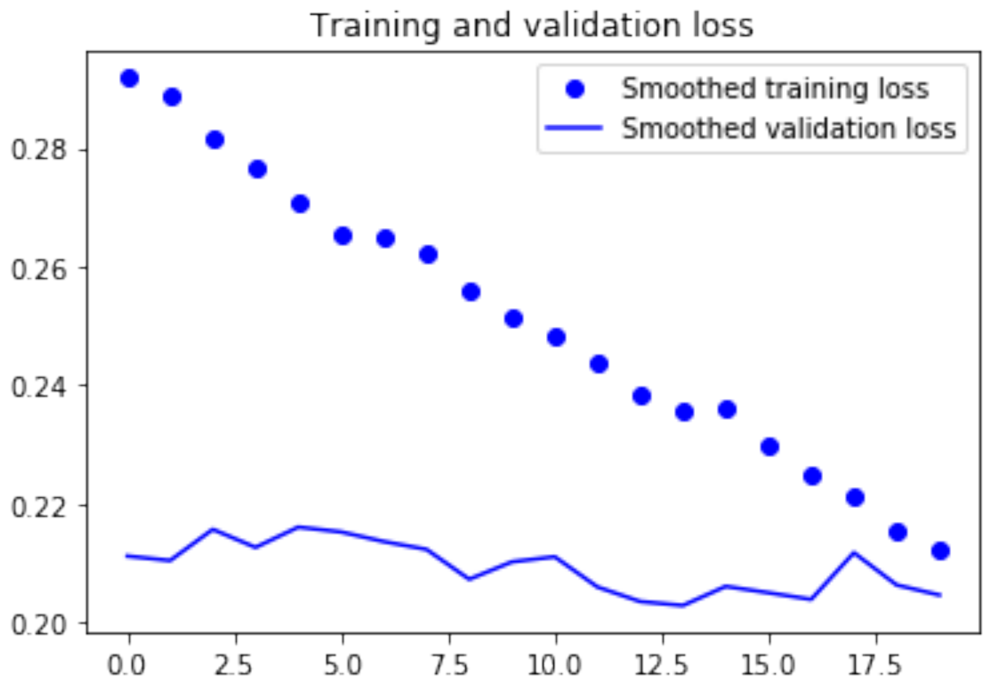
These curves look very noisy. To make them more readable, we can smooth them by replacing every loss and accuracy with exponential moving averages of these quantities. Here's a trivial utility function to do this:

In [0]:

```
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 [0]:

```
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: 0.9179999947547912
```

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.

Problem 4

Consider Jupyter notebook 5.3-using-a-pretrainedconvnet00. ipynb which demonstrates another two techniques for training networks on small dataset. We will use pre-trained VGG16 network. We are asking you to examine in detail so called Feature Extraction technique. You are welcome to test Fine Tuning which is described towards the end of the notebook. You will need a moderately powerful GPU machine (like Google Colab) to do that. Run all your test on small 2 numbers of epochs to estimate duration of the training, before you commit to a long run.

First run the experiment in cell #9, as provided. Then change that cell and rather than adding just one Flatten and two Dense layers, try adding the trainable portion with one Conv2D, one Flatten and two Dense layers. Capture the history of both training runs and plot loss and accuracy. Submit working notebook (with or without fine tuning portion) and it PDF. (25%)

In [0]:

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(conv_base)
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)))
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

In [0]:

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
vgg16 (Model)                (None, 4, 4, 512)         14714688  
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 2, 2, 32)          147488    
_________________________________________________________________
flatten_2 (Flatten)          (None, 128)               0         
_________________________________________________________________
dense_5 (Dense)              (None, 256)               33024     
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 257       
=================================================================
Total params: 14,895,457
Trainable params: 2,540,577
Non-trainable params: 12,354,880
_________________________________________________________________
```

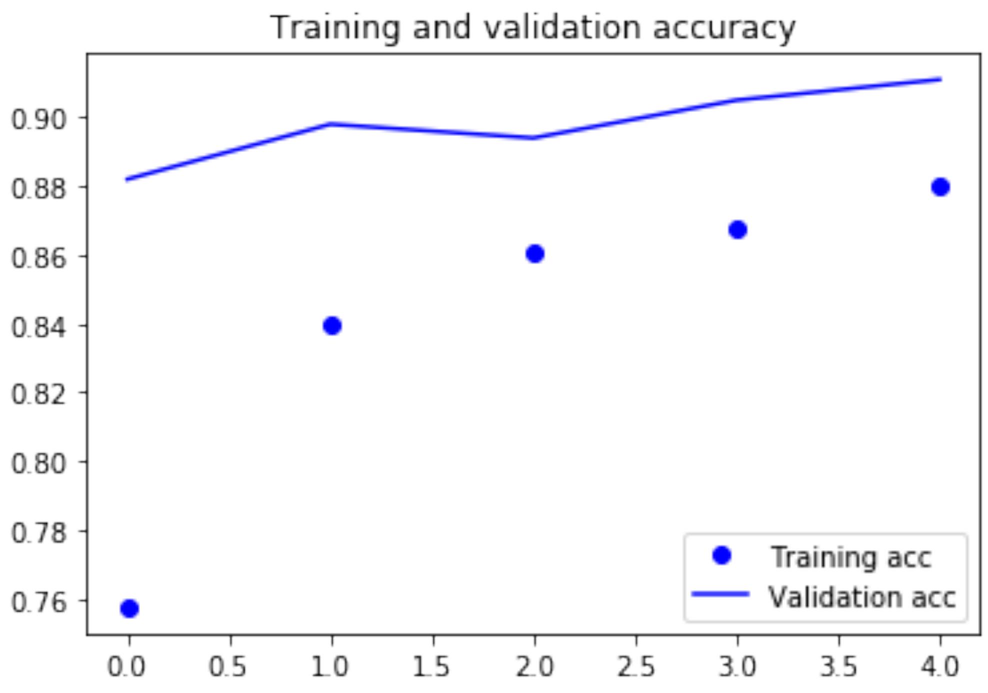
In [0]:

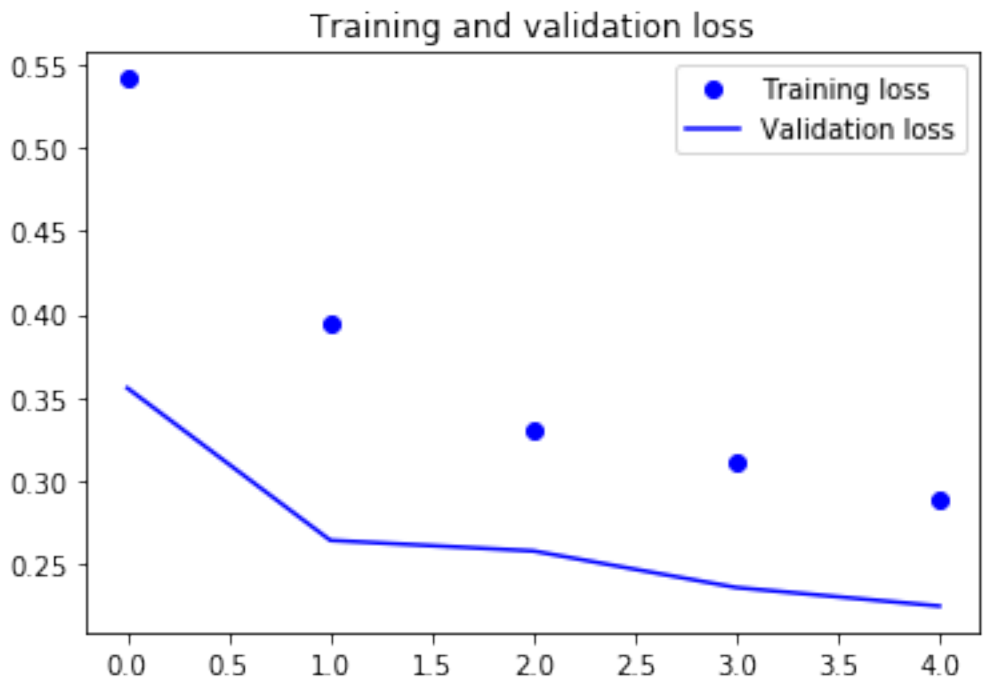
```
from keras.preprocessing.image import ImageDataGenerator
from keras import optimizers

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=5,
      validation_data=validation_generator,
      validation_steps=50,
      verbose=2)
```

```
Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
Epoch 1/5
 - 18s - loss: 0.5416 - acc: 0.7575 - val_loss: 0.3556 - val_acc: 0.88
20
Epoch 2/5
 - 16s - loss: 0.3945 - acc: 0.8400 - val_loss: 0.2640 - val_acc: 0.89
80
Epoch 3/5
 - 16s - loss: 0.3306 - acc: 0.8605 - val_loss: 0.2576 - val_acc: 0.89
40
Epoch 4/5
 - 16s - loss: 0.3110 - acc: 0.8675 - val_loss: 0.2356 - val_acc: 0.90
50
Epoch 5/5
 - 16s - loss: 0.2878 - acc: 0.8805 - val_loss: 0.2245 - val_acc: 0.91
10
```

In [0]:

```
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()
```





This result tests the performance of the feature extraction technique:

NN: At the end of the simple NN applied to the original convolutional base model, we see that the last Epoch gives us an accuracy of 84.7% (based on the validation data result):

```
loss: 0.3812 - acc: 0.8280 - val_loss: 0.3221 - val_acc: 0.8470
```

CNN: Using the conv2D filter to make the NN into a CNN, the results change (and improve) to 91.1%:

```
loss: 0.2878 - acc: 0.8805 - val_loss: 0.2245 - val_acc: 0.9110
```

Comparing the two results, we see the validation accuracy increase by a fair amount (84.7% to 91.1%). And to the point of the question, we also see a decrease in training loss (from 38.1% to 28.8%) which is a significant decrease. There is no overfitting in either scenario compared to the original convolutional base model, which overfits from the start.

When we look at the fine tuning portion (which I was only able to run on Google Colaboratory), the accuracy for the NN model increases from 85.7% to 92.4%, an even more accurate result than applying an additional dense layer. As we can also see from the graphs and the description, the training accuracy increases but validation loss does not decrease as the distribution of losses (and not average) is what accuracy takes into account.