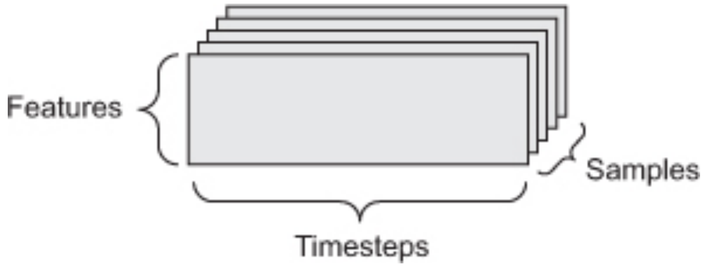
DL-based CV: Practical

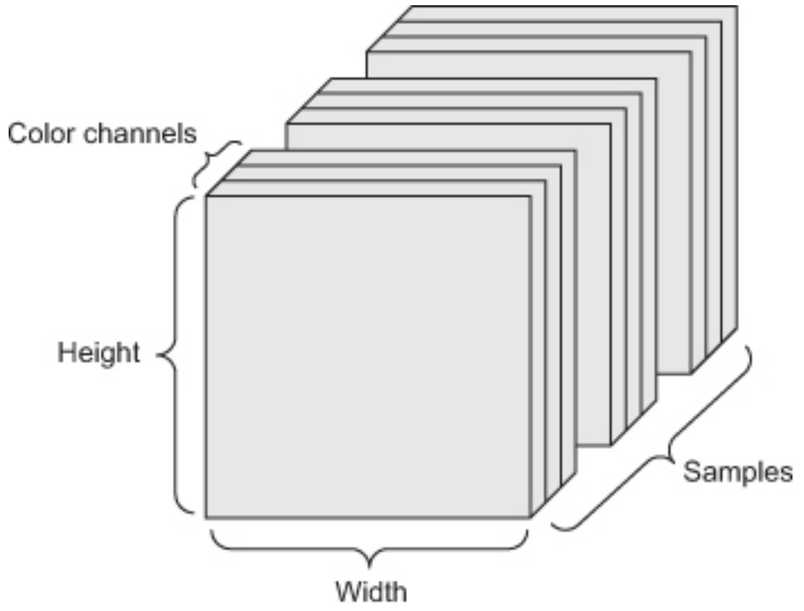
FCNN, CNNs, Keras, etc.

https://www.manning.com/books/deep-learning-with-python

Tensors & Tensor operations

- Tensors:
  - Scalars (0D tensors)
  - Vectors (1D tensors)
  - Matrices (2D tensors): (samples, features)
  - 3D tensors: Timeseries: (samples, timesteps, features)
  - Higher-dimensional tensors:
    - Images: (samples, height, width, channels)
    - Video: (samples, frames, height, width, channels)
- Key attributes
  - Number of axes (rank): >>> print(train_images.**ndim**) -> 3
  - Shape: >>> print(train_images.**shape**) -> (60000, 28, 28)
  - Data type: print(train_images.**dtype**) -> uint8





Keras

https://keras.io

- Keras workflow:
  - Define your training **data**: input tensors and target tensors.
  - Define a network of layers (or **model**) that maps your inputs to your targets.
  - **Configure** the learning process by choosing a loss function, an optimizer, and some metrics to monitor.
  - Iterate on your training data by calling the **fit**() method of your model.
  - evaluate / predict (testing / new unseen data)

Figure 3.2. Google web search interest for different deep-learning frameworks over time

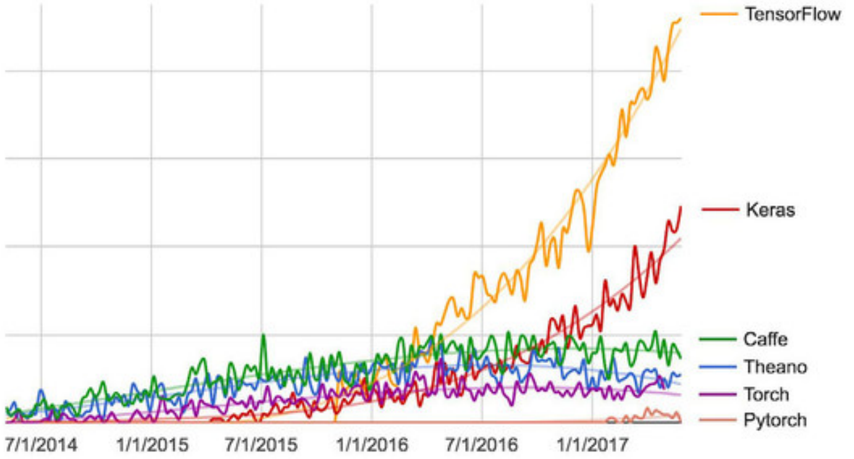
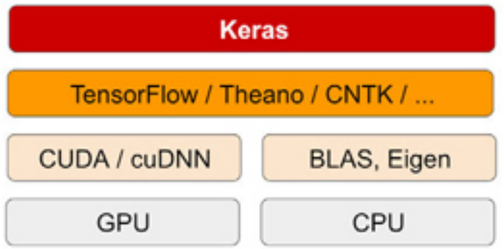


Figure 3.3. The deep-learning software and hardware stack



Intro to dense NNs & Keras

- “Hello World” of DL: “solving” MNIST
- MNIST dataset comes preloaded in Keras (Numpy arrays)
  - training set: train_images and train_labels
  - test set: test_images and test_labels

Figure 2.1. MNIST sample digits

Listing 2.1. Loading the MNIST dataset in Keras

```
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

```
>>> train_images.shape
(60000, 28, 28)
>>> len(train_labels)
60000
>>> train_labels
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

```
>>> test_images.shape
(10000, 28, 28)
>>> len(test_labels)
10000
>>> test_labels
array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

Intro to dense NNs & Keras (2)

- Build the network: layers (lego):
  - densely connected = fully connected layers
  - «**compile**» = make the network ready for training
- Preprocess the data:
  - labels: one-hot encoding / categorical encoding
- Workflow:
  - Feed the training data to the NN which will learn to **associate** images and labels
  - Produce **predictions** for test_images, and verify whether these predictions **match** the labels from test_labels

Listing 2.2. The network architecture

```
from keras import models
from keras import layers

network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

Listing 2.3. The compilation step

```
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

Listing 2.4. Preparing the image data

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

Listing 2.5. Preparing the labels

```
from keras.utils import to_categorical

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

Intro to dense NNs & Keras (3)

- «**fit**»: train the network
  - loss and accuracy
  - Training accuracy: 98.9%
- «**evaluate**»: check that the model performs well on the unsent test set
  - Test accuracy: 97.8%
  - gap is an indication of overfitting
- «**predict**»:
  - >>> model.predict(x_test)

```
>>> network.fit(train_images, train_labels, epochs=5, batch_size=128)
Epoch 1/5
60000/60000 [==============================] - 9s - loss: 0.2524 - acc: 0.9273
Epoch 2/5
51328/60000 [========================>.....] - ETA: 1s - loss: 0.1035 - acc: 0.9
```

```
>>> test_loss, test_acc = network.evaluate(test_images, test_labels)

>>> print('test_acc:', test_acc)
test_acc: 0.9785
```

Intro to dense NNs & Keras (4)

Another dataset, not MNIST

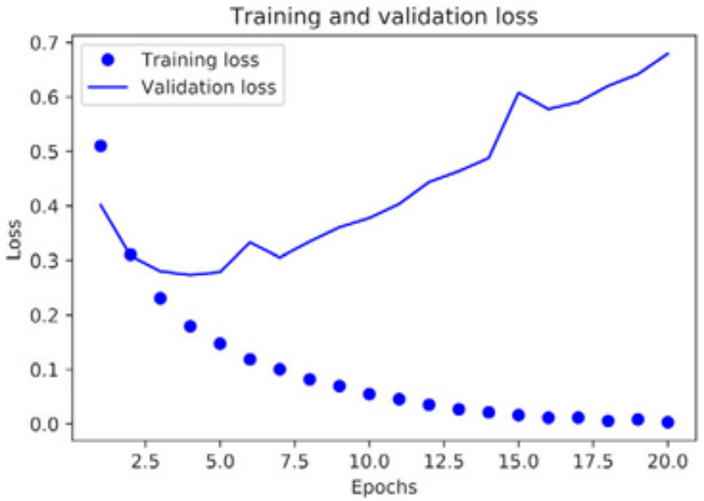
- Validation:
  - create a validation set by using some of the samples from the original training data
  - monitor during training the accuracy of the model on data it has never seen before

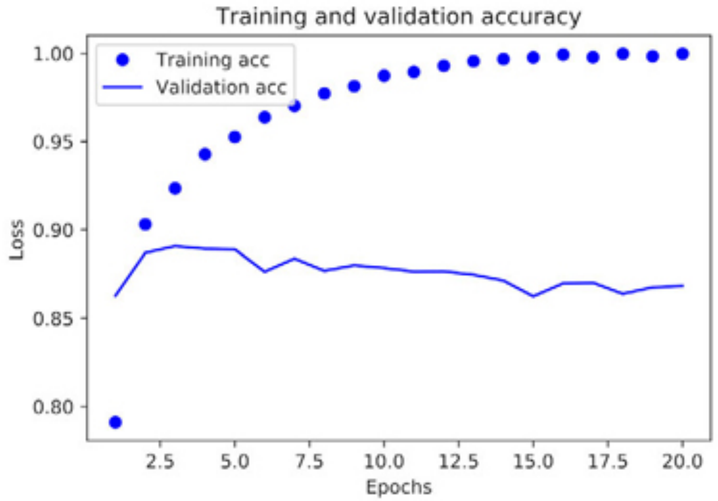
Listing 3.7. Setting aside a validation set

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

Listing 3.8. Training your model

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])

history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```





Summary: Chap 2

- **Learning** means finding a *combination* of model parameters that *minimizes* a loss function for a given set of training data samples and their corresponding targets.

- Learning happens by drawing random **batches** of data samples and their targets, and computing the **gradient** of the network parameters with respect to the **loss** on the batch. The network parameters are then moved a bit (the magnitude of the move is defined by the **learning rate**) in the opposite direction from the gradient.

- The entire learning process is made possible by the fact that neural networks are **chains of differentiable tensor operations**, and thus it's possible to apply the **chain rule** of derivation to find the gradient function mapping the current parameters and current batch of data to a gradient value.

- Two key concepts you'll see frequently in future chapters are **loss and optimizers**. These are the two things you need to define before you begin feeding data into a network.

- The **loss** is the quantity you'll attempt to **minimize** during training, so it should represent a measure of **success** for the task you're trying to solve.

- The **optimizer** specifies the exact way in which the **gradient** of the loss will be used to **update** parameters: for instance, it could be the RMSProp optimizer, **SGD** with momentum, and so on.

Summary: Chap 3

- You're now able to handle the most **common** kinds of machine-learning tasks on **vector data**: binary classification, multi-class classification, and scalar regression. The **"Wrapping up"** sections earlier in the chapter summarize the important points you've learned regarding these types of tasks.
- You'll usually need to **preprocess** raw data before feeding it into a neural network.
- When your data has features with **different ranges**, **scale** each feature independently as part of preprocessing.
- As training progresses, neural networks eventually begin to **overfit** and obtain worse results on never-before-seen data.
- If you don't have much training data, use a **small network** with only one or two hidden layers, to avoid severe overfitting.
- If your data is divided into many categories, you may cause information bottlenecks if you make the intermediate layers too small.
- **Regression** uses different loss functions and different evaluation metrics than classification.
- When you're working with little data, K-fold validation can help reliably evaluate your model.

5.1. Introduction to convnets

Listing 5.1. Instantiating a small convnet

```
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

- input_shape=(28, 28, 1)
- (image_height, image_width, image_channels)

5.1. Introduction to convnets (2)

(3 x 3 x 1) x 32 + 32 = 9 x 32 + 32 = 288 + 32 = 320

(3 x 3 x 32) x 64 + 64 = 288 x 64 + 64 = 18432 + 64 = 18496

```
>>> model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
maxpooling2d_1 (MaxPooling2D)(None, 13, 13, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
maxpooling2d_2 (MaxPooling2D)(None, 5, 5, 64)          0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 3, 3, 64)          36928
=================================================================
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
```

- Conv2D and MaxPooling2D layers
- Output: 3D tensor of shape (height, width, channels)

Listing 5.1. Instantiating a small convnet

```
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

5.1. Introduction to convnets (3)

Listing 5.2. Adding a classifier on top of the convnet

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

- Flatten the 3D outputs tensor of shape (3, 3, 64) (feature extraction) to 1D input tensor of shape? (classifier)
- two densely connected layers
- 10-way classification and softmax activation

5.1. Introduction to convnets (4)

```
>>> model.summary()
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
maxpooling2d_1 (MaxPooling2D)(None, 13, 13, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
maxpooling2d_2 (MaxPooling2D)(None, 5, 5, 64)          0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 3, 3, 64)          36928
_________________________________________________________________
flatten_1 (Flatten)          (None, 576)               0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                36928
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650
=================================================================
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
```

- (3, 3, 64) outputs flattened into (576,)

(3 x 3 x 64) x 64 + 64 = 576 x 64 + 64 = 36864 + 64 = 36928

Listing 5.2. Adding a classifier on top of the convnet

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

5.1. Introduction to convnets (5)

Listing 5.3. Training the convnet on MNIST images

```
from keras.datasets import mnist
from keras.utils import to_categorical

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=5, batch_size=64)
```

- Train the convnet on the MNIST digits
- astype
- to_categorical

5.1. Introduction to convnets (6)

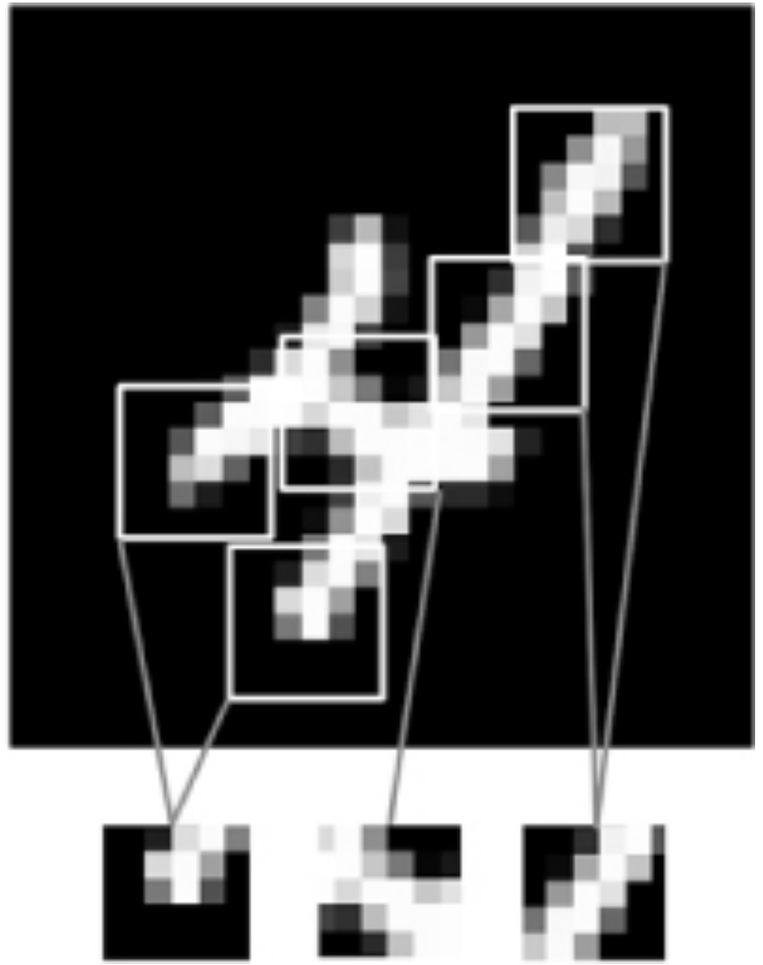
```
>>> test_loss, test_acc = model.evaluate(test_images, test_labels)
>>> test_acc
0.9908000000000001
```

- Evaluate the model
- From 97.8% (FC/Dense) to 99.1% (simple CNN), error rate down by 68%
- Why?

5.1.1. The convolution operation

- Fundamental difference
  - Dense layer: **global** patterns (all pixels)
  - Convolution layer: **local** patterns (small 2D windows)

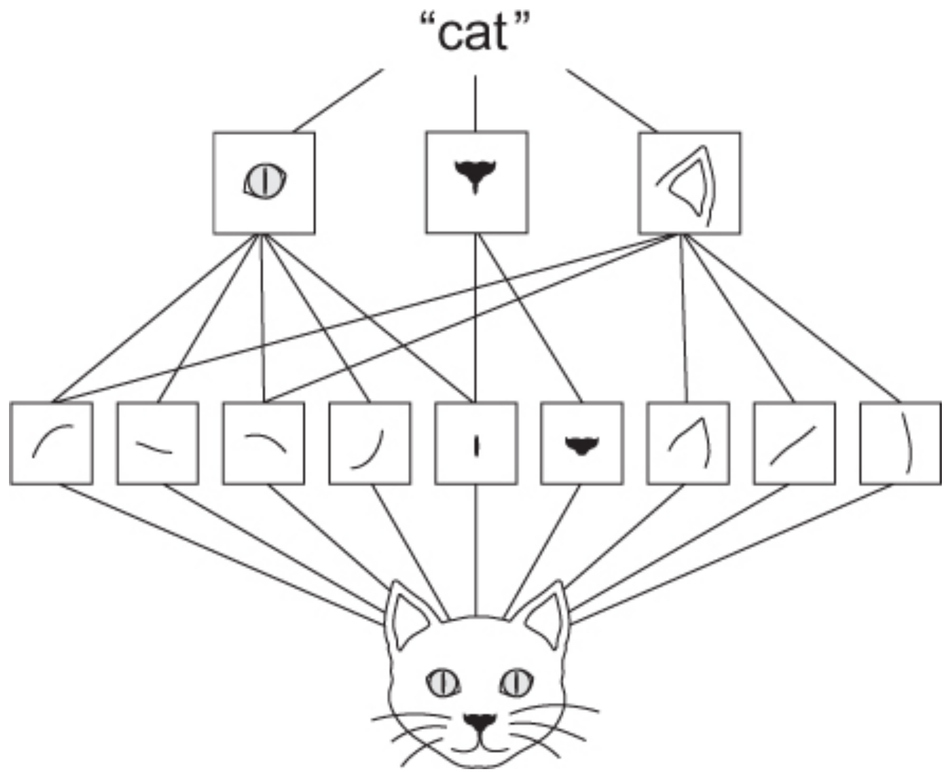
Figure 5.1. Images can be broken into **local patterns** such as edges, textures, and so on.



5.1.1. The convolution operation (2)

- Key characteristics / properties of convnets:
  - The patterns they learn are **translation invariant** (patterns learned are recognized anywhere, visual world is trans. inv.)
  - They can learn **spatial hierarchies** of patterns (1: learn small local patterns such as edges, 2: learn larger patterns made of the features of the first layers, 3: and so on, i.e. learn increasingly complex and abstract visual concepts)
- The **visual world** is fundamentally **translation invariant & spatially hierarchical**

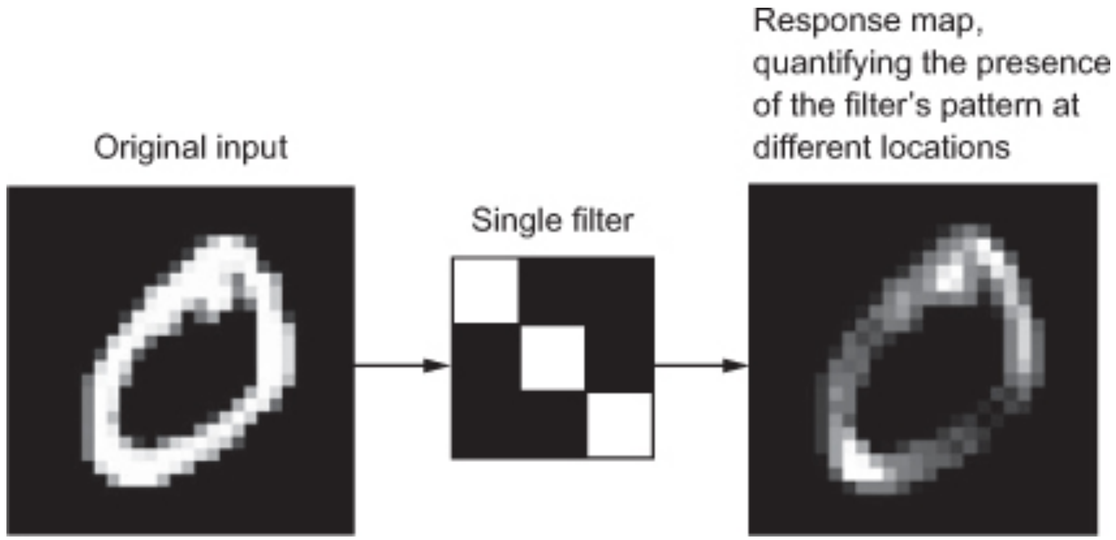
Figure 5.2. The visual world forms a **spatial hierarchy** of visual modules: **hyperlocal edges** combine into **local objects** such as eyes or ears, which combine into **high-level concepts** such as “cat.”



5.1.1. The convolution operation (3)

- Convolution operations (images): 3D input tensors / **feature maps** -> 3D output tensors / feature maps
  - height, width and depth / **channels** axis (e.g. RGB (3) or gray (1)).
  - extracts **patches** from input feature map, apply the same **transformation** to all of these patches, producing an output feature map.
  - the different channels in that depth axis stand for **filters** that encode specific aspects of the input data (high level: presence of a face)
  - MNIST: (28, 28, 1) -> (26, 26, 32), i.e. 32 filters / features, each a 2D spatial **response map**, i.e. the response of this filter over the input (patch by patch).

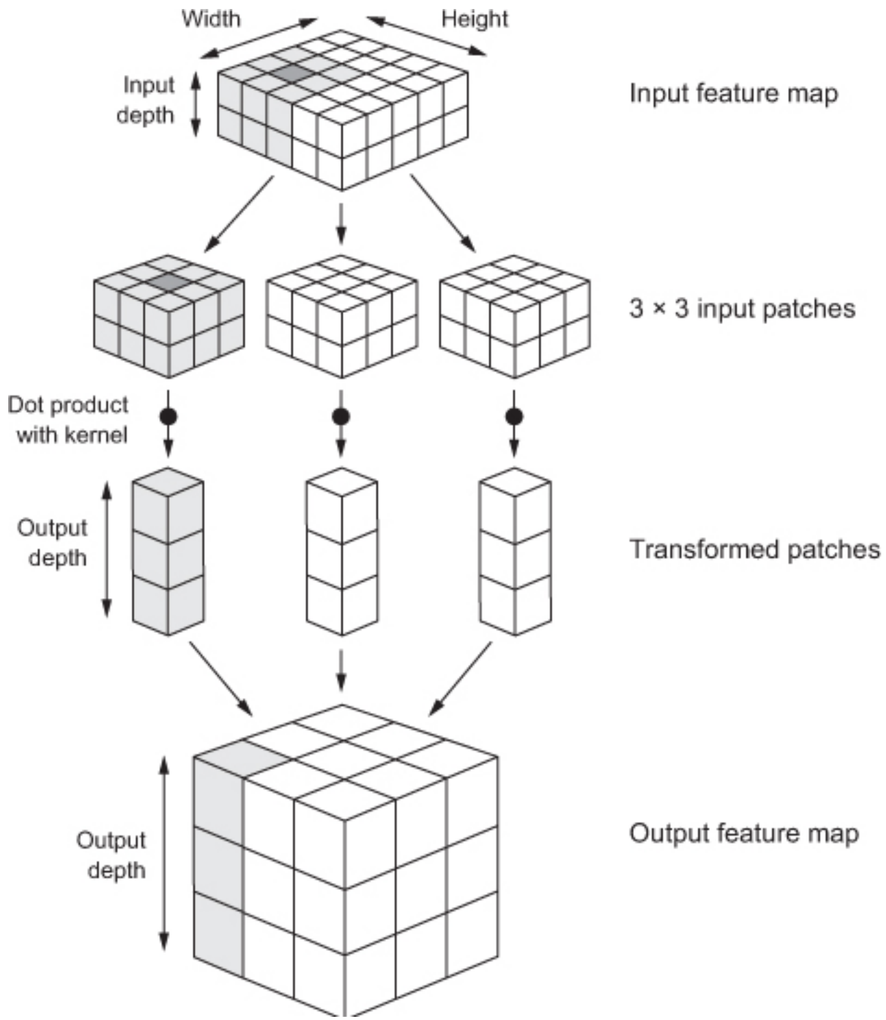
Figure 5.3. The concept of a **response map**: a 2D map of the **presence** of a **pattern** at different **locations** in an input



5.1.1. The convolution operation (4)

- Convolutions are defined by two key parameters:
  - **Size of the patches** extracted from the inputs (3 × 3 or 5 × 5)
  - **Depth** of the output feature map: **# filters** computed
- Works by sliding these windows (e.g. 3 × 3) over the 3D input feature map
  - extract a 3D patch of shape (window_height, window_width, input_depth).
  - transform via a tensor product with the same learned weight matrix, called the **convolution kernel**.
  - a new **product** for each filter / kernel, making a 1D vector of shape (output_depth,)
  - spatially reassemble into a 3D output map of shape (height, width, output_depth). Every spatial **location** in the output feature map corresponds to the same location in the input feature map
- output width and height may **differ** from the input width and height for two reasons: border effects (padding) & strides

Figure 5.4. How convolution **works**



5.1.1: Border effects and padding

- Consider a 5 × 5 feature map (25 tiles total):
  - can only center 9 3x3 windows forming a 3x3 grid so the output feature map will be 3×3.
  - it shrinks by two tiles alongside each dimension (MNIST: 28×28 become 26×26)
- Use **padding** to get an output feature map with the same spatial dimensions as the input
  - add appropriate number of rows and columns to fit center convolution windows around every input tile
  - valid or same

Figure 5.5. **Valid** locations of 3 × 3 **patches** in a 5 × 5 input **feature map**

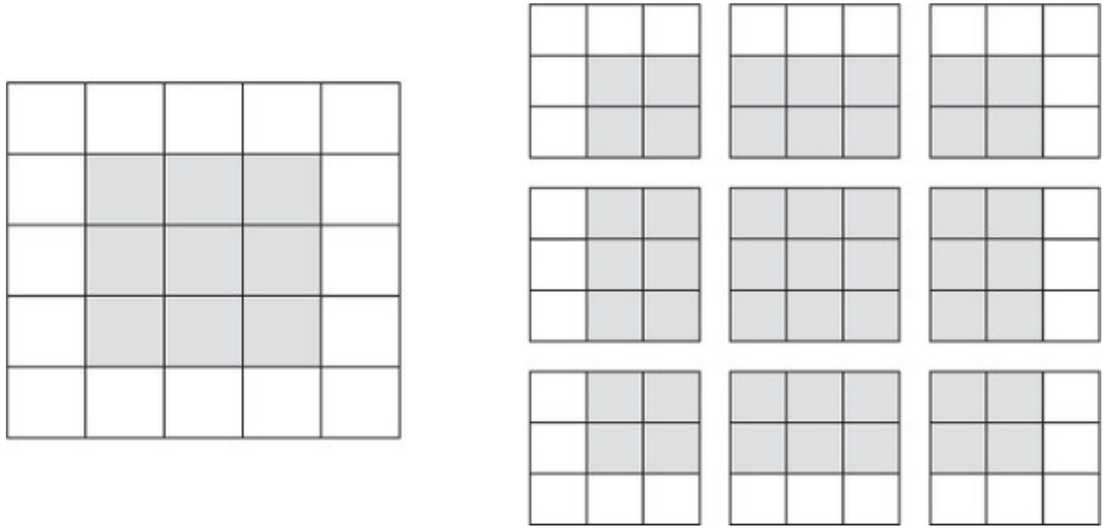
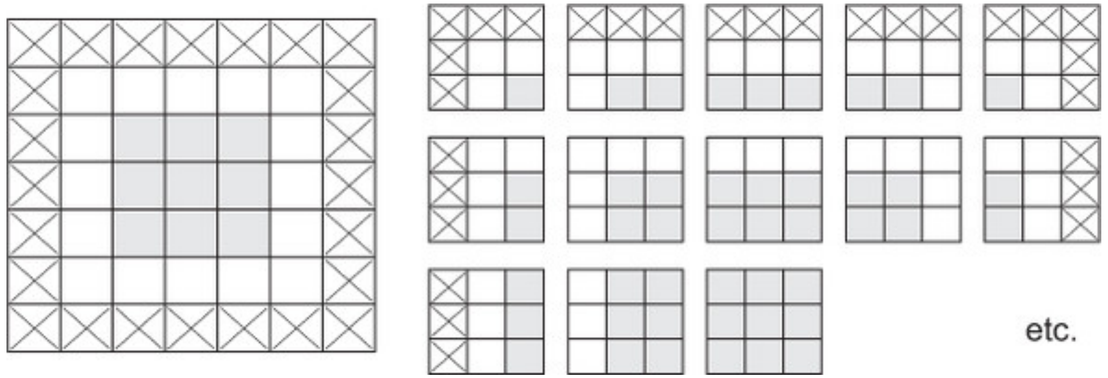


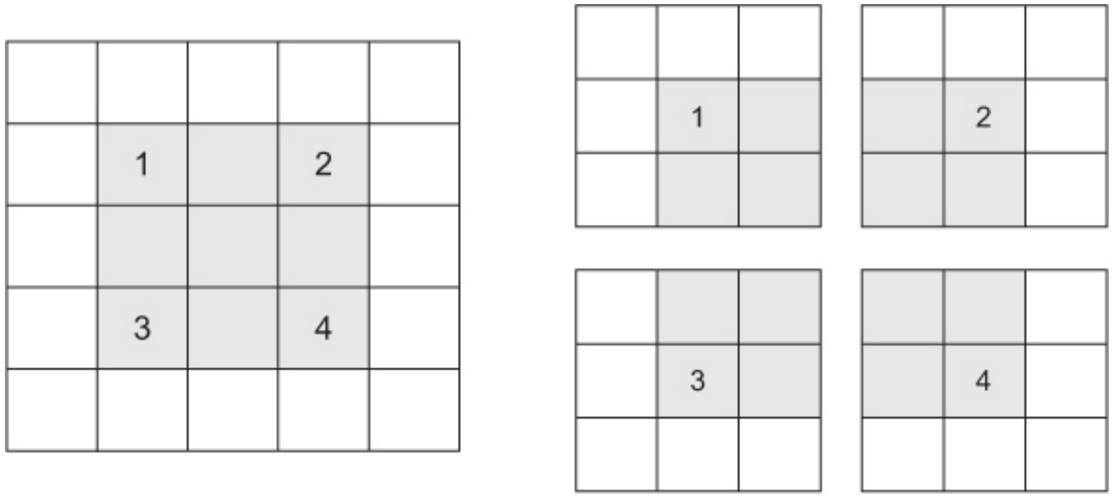
Figure 5.6. **Padding** a 5 × 5 input in order to be able to extract 25 3 × 3 patches



5.1.1: Strides

- **stride**: distance between two successive windows (hyper-parameter, default 1)
- **strided convolutions**: stride > 1 (stride 2 = feature map are **downsampled** by a factor of 2 (in addition to any changes induced by border effects).

Figure 5.7. 3 × 3 convolution patches with 2 × 2 **strides** (without padding).



5.1.2. The max-pooling operation

- **Pooling**: aggressively **downsample** feature maps (much like strided convolutions).
- MNIST: input feature maps: 26x26, output: 13x13
- **Max pooling**: extract windows from the input feature maps and output the max value of each channel
  - convolution op.: transform local patches via a learned linear convolution kernel
    - usually done with 3 × 3 windows and no stride (i.e. stride 1)
  - max-pooling op.: transform via a hardcoded max tensor operation.
    - usually done with 2 × 2 windows and stride 2 (to downsample the feature maps by a factor of 2)

5.1.2. The max-pooling operation

- Why downsample feature maps?
  - **reduce** the number of **feature-map coefficients** to process
    - 30,976 coefficients per sample, dense layer of size 512 on top, that layer would have 15.8 million parameters, small model, overfitting.
  - **induce spatial-filter hierarchies** by making successive convolution layers look at increasingly large windows
    - the 3×3 windows in the third layer will only contain information coming from 7×7 windows in the initial input, we need the features from the last convolution layer to contain information about the entire image

```
1  model_no_max_pool = models.Sequential()
2  model_no_max_pool.add(layers.Conv2D(32, (3, 3), activation='relu',
3                                      input_shape=(28, 28, 1)))
4  model_no_max_pool.add(layers.Conv2D(64, (3, 3), activation='relu'))
5  model_no_max_pool.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

```
1  >>> model_no_max_pool.summary()
2
3  Layer (type)                 Output Shape              Param #
4  =================================================================
5  conv2d_4 (Conv2D)            (None, 26, 26, 32)        320
6
7  _________________________________________________________________
8  conv2d_5 (Conv2D)            (None, 24, 24, 64)        18496
9
10 _________________________________________________________________
11 conv2d_6 (Conv2D)            (None, 22, 22, 64)        36928
12 =================================================================
13 Total params: 55,744
14 Trainable params: 55,744
15 Non-trainable params: 0
```

22 x 22 x 64 = 30976

Summary

- Basics of convnets: feature maps, convolution, and max pooling
- Why not strides instead of max pooling & average pooling instead of max pooling
  - features tend to encode the spatial **presence** of some **pattern** or concept over the different tiles of the feature map (hence, the term feature map)
    - it's more informative to look at the **maximal presence** of different features than at their average presence
    - the most reasonable subsampling strategy is to first produce **dense maps** of features (via unstrided convolutions) and then look at the **maximal activation** of the features over small patches, rather than looking at **sparser windows** of the inputs (via strided convolutions) or averaging input patches, which could cause you to miss or dilute feature-presence information.

5.2. Training a convnet from scratch on a **small** dataset

- cats or dogs (4000, 2000 each, 2000 for training, 1000 for validation and 1000 for testing)
- Strategies (small datasets):
  - training a new model from **scratch** (71%, no regularization, overfitting)
    - data augmentation (82%)
  - **feature extraction** using a pretrained model (90% to 96%)
  - **fine-tuning** a pretrained model (97%)

5.2.1. The relevance of deep learning for small-data problems

- DL Needs lots of data?
  - Partly valid: can find **interesting features** in the training data **on its own**, without any need for **manual feature engineering**, and this can only be achieved when lots of training examples are available (high-dimensional input, e.g. images)
  - Relative: e.g. to the size and depth of the network.
    - not possible to train a convnet to solve a complex problem with just a few tens of samples.
    - a few hundred can potentially suffice if the model is small and well regularized and the task is simple
    - convnets are highly data efficient (learn local, translation-invariant features)
  - Training a convnet from scratch on a very small image dataset will still yield reasonable results without the need for feature engineering
  - DL models are highly repurposable: take a image class model trained on a large-scale dataset and reuse it on a significantly different problem with only minor changes.
    - many **pretrained** models (usually trained on the Image-Net dataset) are now publicly **available**, can be **reused** and work pretty well on **limited data**.

5.2.2. Downloading the data

- Cats vs. Dogs dataset:
  - Kaggle competition, late 2013, won by a convnet, 95% accuracy
  - Here: use less than 10% of the data from the competition.
  - Kaggle dataset: 25 000 images, 12 500 from each class, 543 MB (compressed).
  - Our dataset: training: **2000**, 1000 cats & 1000 dogs, validation (**1000**, 500 each) and test (**1000**, 500 each)

Figure 5.8. Samples from the Dogs vs. Cats dataset. Sizes weren't modified: the samples are heterogeneous in size, appearance, and so on.

Listing 5.4. Copying images to training, validation, and test directories

```
import os, shutil

original_dataset_dir = '/Users/fchollet/Downloads/kaggle_original_data'

base_dir = '/Users/fchollet/Downloads/cats_and_dogs_small'
os.mkdir(base_dir)

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

train_cats_dir = os.path.join(train_dir, 'cats')
os.mkdir(train_cats_dir)

train_dogs_dir = os.path.join(train_dir, 'dogs')
os.mkdir(train_dogs_dir)

validation_cats_dir = os.path.join(validation_dir, 'cats')
os.mkdir(validation_cats_dir)

validation_dogs_dir = os.path.join(validation_dir, 'dogs')
os.mkdir(validation_dogs_dir)

test_cats_dir = os.path.join(test_dir, 'cats')
os.mkdir(test_cats_dir)

test_dogs_dir = os.path.join(test_dir, 'dogs')
os.mkdir(test_dogs_dir)

fnames = ['cat.{}.jpg'.format(i) for i in range(1000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(train_cats_dir, fname)
    shutil.copyfile(src, dst)

fnames = ['cat.{}.jpg'.format(i) for i in range(1000, 1500)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(validation_cats_dir, fname)
    shutil.copyfile(src, dst)

fnames = ['cat.{}.jpg'.format(i) for i in range(1500, 2000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(test_cats_dir, fname)
    shutil.copyfile(src, dst)

fnames = ['dog.{}.jpg'.format(i) for i in range(1000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(train_dogs_dir, fname)
    shutil.copyfile(src, dst)
fnames = ['dog.{}.jpg'.format(i) for i in range(1000, 1500)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(validation_dogs_dir, fname)
    shutil.copyfile(src, dst)

fnames = ['dog.{}.jpg'.format(i) for i in range(1500, 2000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(test_dogs_dir, fname)
    shutil.copyfile(src, dst)
```

5.2.2 (2)

```
>>> print('total training cat images:', len(os.listdir(train_cats_dir)))
total training cat images: 1000

>>> print('total training dog images:', len(os.listdir(train_dogs_dir)))
total training dog images: 1000

>>> print('total validation cat images:', len(os.listdir(validation_cats_dir)))
total validation cat images: 500

>>> print('total validation dog images:', len(os.listdir(validation_dogs_dir)))
total validation dog images: 500

>>> print('total test cat images:', len(os.listdir(test_cats_dir)))
total test cat images: 500

>>> print('total test dog images:', len(os.listdir(test_dogs_dir)))
total test dog images: 500
```

Balanced binary-classification problem!

5.2.3. Building your network

- convnet: stack of alternated Conv2D (w/ relu activation) and MaxPooling2D layers (4 of each)
- bigger images (150x150) and a more complex problem: larger model (one more Conv2D / MaxPooling2D): increase capacity & reduce last feature map
- Binary-classification problem: single unit with a sigmoid activation (probability for one class or the other)

Listing 5.5. Instantiating a small convnet for dogs vs. cats classification

```
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

5.2.3. Building your network (2)

- The depth of the feature maps progressively increases in the network (from 32 to 128), whereas the size of the feature maps decreases (from 148×148 to 7×7). This is a pattern you'll see in almost all convnets.

```
>>> model.summary()
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 148, 148, 32)      896
_________________________________________________________________
maxpooling2d_1 (MaxPooling2D)(None, 74, 74, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 72, 72, 64)        18496
_________________________________________________________________
maxpooling2d_2 (MaxPooling2D)(None, 36, 36, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 34, 34, 128)       73856
_________________________________________________________________
maxpooling2d_3 (MaxPooling2D)(None, 17, 17, 128)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 15, 15, 128)       147584
_________________________________________________________________
maxpooling2d_4 (MaxPooling2D)(None, 7, 7, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 6272)              0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               3211776
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 513
=================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
```

5.2.3. Building your network (3)

- sigmoid unit and binary cross-entropy as the loss
- RMSprop optimizer

Listing 5.6. Configuring the model for training

```
from keras import optimizers

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])
```

Table 4.1. Choosing the right last-layer activation and loss function for your model

Problem type	Last-layer activation	Loss function
Binary classification	sigmoid	binary_crossentropy
Multiclass, single-label classification	softmax	categorical_crossentropy
Multiclass, multilabel classification	sigmoid	binary_crossentropy
Regression to arbitrary values	None	mse
Regression to values between 0 and 1	sigmoid	mse or binary_crossentropy

5.2.4. Data preprocessing

- **Steps** for getting the data into the network:
  - Read the picture files.
  - Decode the JPEG content to RGB grids of pixels.
  - Convert these into floating-point tensors.
  - Rescale the pixel values (between 0 and 255) to the [0, 1] interval (as you know, neural networks prefer to deal with small input values).
- Keras utilities:
  - from keras.preprocessing.image import **ImageDataGenerator**

Listing 5.7. Using `ImageDataGenerator` to read images from directories

```
1  from keras.preprocessing.image import ImageDataGenerator
2
3  train_datagen = ImageDataGenerator(rescale=1./255)
4  test_datagen = ImageDataGenerator(rescale=1./255)
5
6  train_generator = train_datagen.flow_from_directory(
7          train_dir,
8          target_size=(150, 150)
9          batch_size=20,
10         class_mode='binary')
11
12 validation_generator = test_datagen.flow_from_directory(
13         validation_dir,
14         target_size=(150, 150),
15         batch_size=20,
16         class_mode='binary')
```

5.2.4. Data preprocessing (2)

- generator similar to iterator
  - for ... in, built using the **yield** operator
- train_generator yields **batches** of 150×150 RGB images (shape (20, 150, 150, 3)) and binary labels (shape (20,))

UNDERSTANDING PYTHON GENERATORS

A *Python generator* is an object that acts as an iterator: it's an object you can use with the `for` `...` `in` operator. Generators are built using the `yield` operator.

Here is an example of a generator that yields integers:

```
def generator():
    i = 0
    while True:
        i += 1
        yield i

for item in generator():
    print(item)
    if item > 4:
        break
```

copy

It prints this:

```
>>> for data_batch, labels_batch in train_generator:
>>>     print('data batch shape:', data_batch.shape)
>>>     print('labels batch shape:', labels_batch.shape)
>>>     break
data batch shape: (20, 150, 150, 3)
labels batch shape: (20,)
```

5.2.4. Data preprocessing (3)

- Fit the model to the data using the generator
- **fit_generator** method equivalent to **fit**
  - 1st argument is a **generator** that will yield **batches** of inputs and targets indefinitely
  - 2nd argument: how many samples (i.e. gradient descent steps, 20 images each) to draw from the generator before declaring an epoch over
- Save your models after training

Listing 5.8. Fitting the model using a batch generator

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

100 x 20 = 2000

50 x 20 = 1000

Listing 5.9. Saving the model

```
model.save('cats_and_dogs_small_1.h5')
```

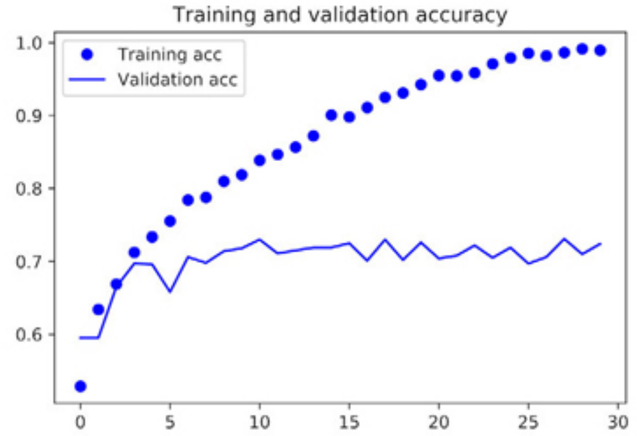
5.2.4. Data preprocessing (4)

- Plots are characteristic of **overfitting**: training accuracy increases to nearly 100%, validation accuracy stalls at 70–72%, the validation loss reaches its minimum after only five epochs and then stalls

- **Mitigate** overfitting: dropout, weight decay (L2 regularization) and **data augmentation**

Listing 5.10. Displaying curves of loss and accuracy during training

```
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(1, len(acc) + 1)

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()
```



5.2.5. Using data augmentation

- **Overfitting**: too few samples, unable to generalize to new data

- Given **infinite** data, your model would be **exposed** to every possible **aspect** of the data distribution at hand: you would never overfit.

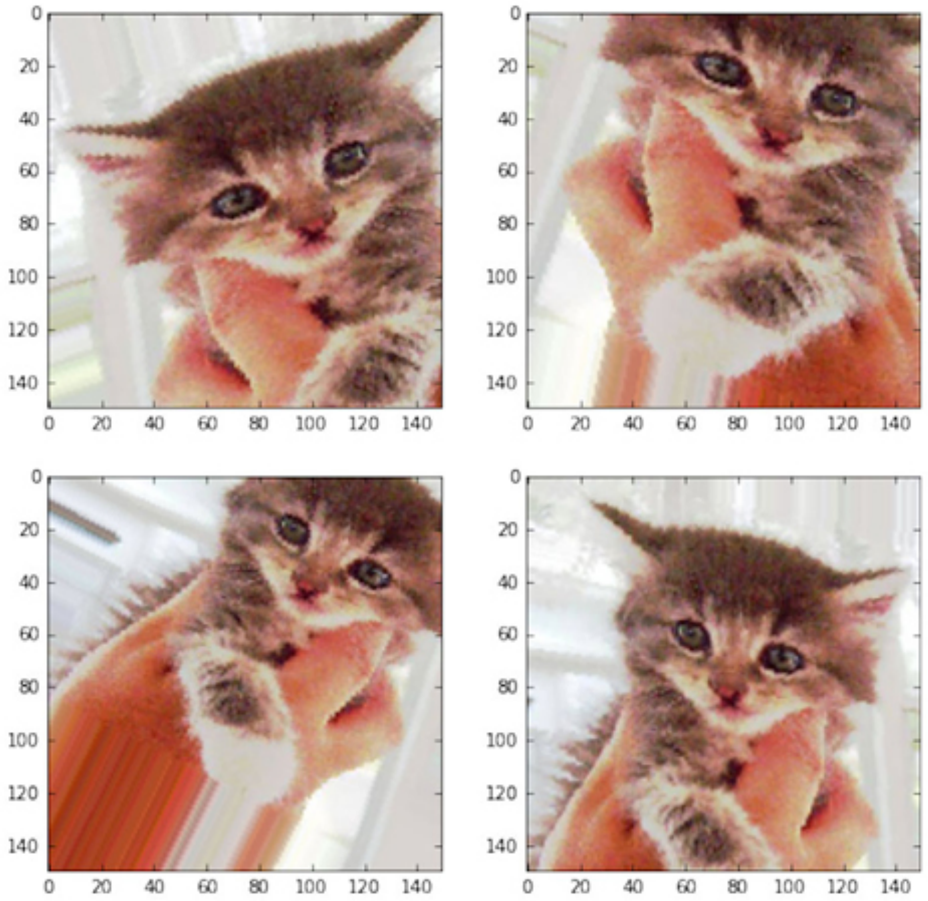
- Data **augmentation**: generating more (believable-looking) training data from existing training samples via transformations without changing the label. Goal is to never see the exact same picture twice during training exposing the model to more aspects of the data and generalize better.

- Keras: use an **ImageDataGenerator** yielding a number of random transformations.

Listing 5.11. Setting up a data augmentation configuration via

```
datagen = ImageDataGenerator(
      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')
```

5.2.5. Using data augmentation (2)



Listing 5.12. Displaying some randomly augmented training images

```
from keras.preprocessing import image

fnames = [os.path.join(train_cats_dir, fname) for
     fname in os.listdir(train_cats_dir)]

img_path = fnames[3]

img = image.load_img(img_path, target_size=(150, 150))

x = image.img_to_array(img)
x = x.reshape((1,) + x.shape)

i = 0
for batch in datagen.flow(x, batch_size=1):
    plt.figure(i)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
    i += 1
    if i % 4 == 0:
        break

plt.show()
```

5.2.5. Using data augmentation (3)

Listing 5.13. Defining a new convnet that includes dropout

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dropout(0.5))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])
```

Listing 5.15. Saving the model

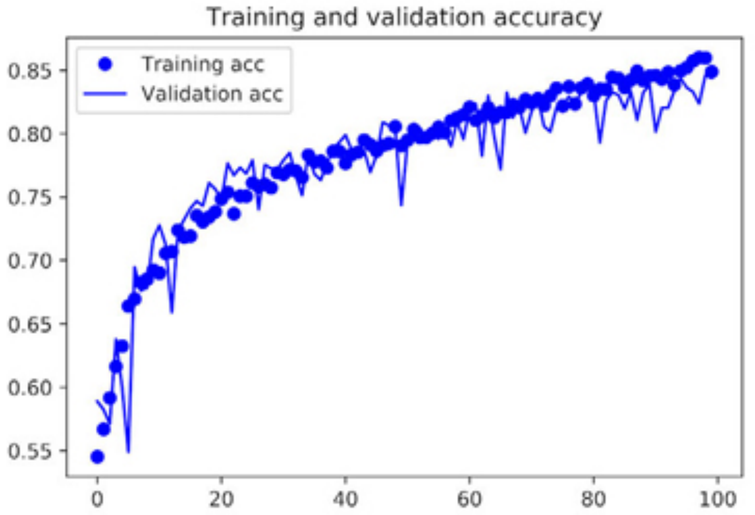
```
model.save('cats_and_dogs_small_2.h5')
```

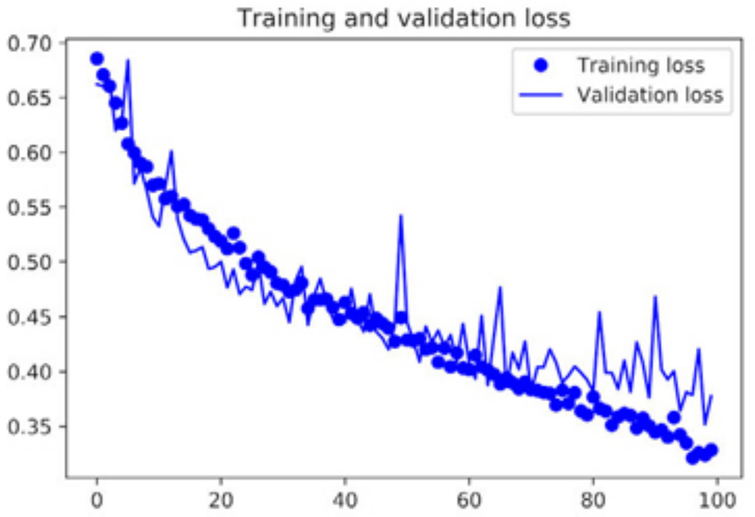
Listing 5.14. Training the convnet using data-augmentation generators

```
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,)

test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        train_dir,
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')

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

5.2.5. Using data augmentation (4)

- Results with data augmentation and dropout:
  - No evidence of overfitting: training curves are closely tracking the validation curves, accuracy of 82%
  - By exploring further regularization techniques and tuning the network's parameters (e.g. number of filters per convolution layer, number of layers in the network etc.): up to 86% or 87% accuracy.
  - Difficult to go any higher just by training your own convnet from **scratch** (too little data), to improve your accuracy you'll have to use a **pretrained model**





5.3. Using a pretrained convnet

- Pretrained network: **saved** network that was previously trained on a **large** dataset
- If dataset is **large** enough and **general** enough, then the **spatial hierarchy** of **features learned** by the pretrained network can effectively act as a **generic model** of the **visual world**.
- Train a network on Image-Net (mostly animals and everyday objects) and then **repurpose** this trained network for something completely different, e.g. identifying furniture items in pictures or finding structures in ultrasound images.
- This **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** also for small-data problems.
- ImageNet: 1.4 million labeled images and 1,000 different classes (contains many animal classes, including different species of cats and dogs)
- Will use VGG16 architecture (others: ResNet, Inception, Inception-ResNet, Xception, and so on)
- **Pretrained** networks can be used in two different ways: **feature extraction** and **fine-tuning**

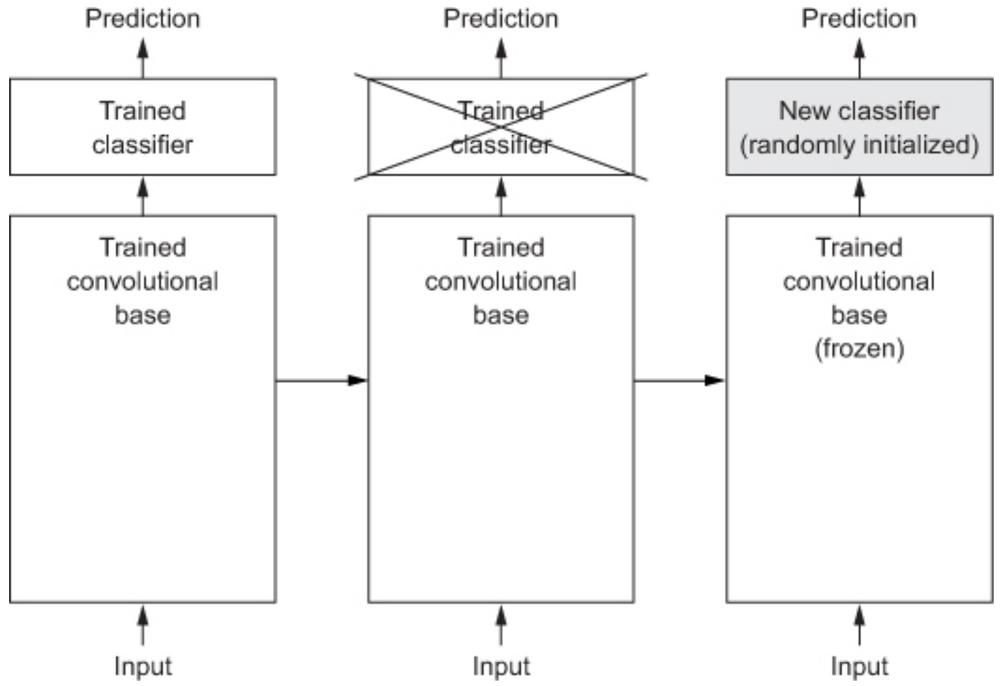
5.3.1. Feature extraction

- Feature extraction: keep the **convolutional base** of a previously trained network (running the new data through it) and train a new **classifier** (on top of the output).

- The representations learned by the convolutional **base**: feature maps or **presence maps** of **generic** concepts over a picture. Representations learned by the **classifier** will necessarily be **specific** to the set of classes on which the model was trained (also, contain no information about **where** objects are located in the input image)

- Level of generality and reusability depends on the depth of the layer: early layer extract local, highly **generic** feature maps (such as visual edges, colors, and textures), higher layers extract more-**abstract** concepts (such as “cat ear” or “dog eye”). Depending on the dataset - not use the entire convolutional base.

Figure 5.14. Swapping **classifiers** while keeping the same **convolutional base**



5.3.1. Feature extraction (2)

- Keras: prepackaged image-classification models (**VGG16**, VGG19, ResNet50, Inception V3, Xception, MobileNet)

- The final feature map has shape (4, 4, 512), two alternatives:

  - Running the convolutional base over your dataset, recording its output to a Numpy array on disk, and then using this data as input to a standalone, densely connected classifier (back to start)

  - Extending the model you have (conv_base) by adding Dense layers on top (allows data augmentation).

Listing 5.16. Instantiating the VGG16 convolutional base

```
from keras.applications import VGG16

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

```
>>> conv_base.summary()
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 150, 150, 3)       0

_________________________________________________________________
block1_conv1 (Convolution2D) (None, 150, 150, 64)      1792

_________________________________________________________________
block1_conv2 (Convolution2D) (None, 150, 150, 64)      36928

_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 75, 75, 64)        0

_________________________________________________________________
block2_conv1 (Convolution2D) (None, 75, 75, 128)       73856

_________________________________________________________________
block2_conv2 (Convolution2D) (None, 75, 75, 128)       147584

_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 37, 37, 128)       0

_________________________________________________________________
block3_conv1 (Convolution2D) (None, 37, 37, 256)       295168

_________________________________________________________________
block3_conv2 (Convolution2D) (None, 37, 37, 256)       590080

_________________________________________________________________
block3_conv3 (Convolution2D) (None, 37, 37, 256)       590080

_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 18, 18, 256)       0

_________________________________________________________________
block4_conv1 (Convolution2D) (None, 18, 18, 512)       1180160

_________________________________________________________________
block4_conv2 (Convolution2D) (None, 18, 18, 512)       2359808

_________________________________________________________________
block4_conv3 (Convolution2D) (None, 18, 18, 512)       2359808

_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 9, 9, 512)         0

_________________________________________________________________
block5_conv1 (Convolution2D) (None, 9, 9, 512)         2359808

_________________________________________________________________
block5_conv2 (Convolution2D) (None, 9, 9, 512)         2359808

_________________________________________________________________
block5_conv3 (Convolution2D) (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
```

5.3.1:
Fast feature extraction **without** data augmentation

Listing 5.17. Extracting features using the pretrained convolutional base

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

base_dir = '/Users/fchollet/Downloads/cats_and_dogs_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:
            break                                                    1
    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)
```

(samples, 4, 4, 512) -> (samples, 8192)

```
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))
```

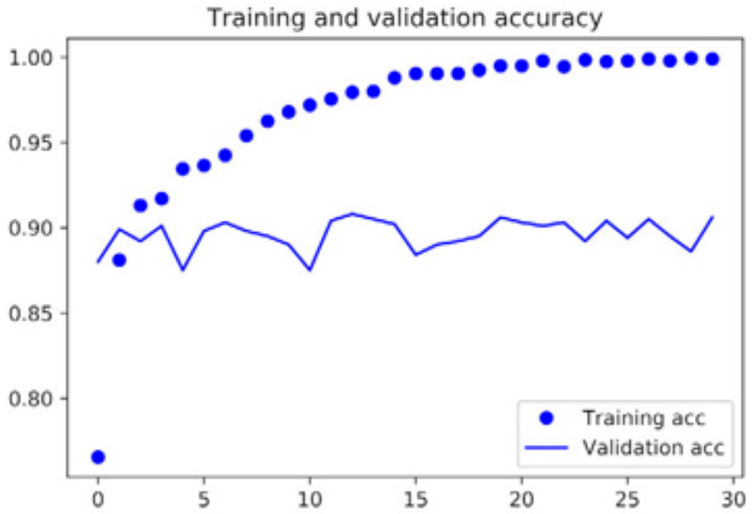
Define your densely connected classifier and train on recorded data

```
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))
```

5.3.1:
Fast feature extraction without data augmentation (2)

Listing 5.19. Plotting the results

```
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(1, len(acc) + 1)

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()
```



90% accuracy (better than from scratch approach), plots also indicate overfitting

5.3.1:
Feature extraction with data augmentation

- Extending the conv_base model and running it end to end on the inputs.
- Much slower and more expensive (need a GPU)
- Part of a model can be added just like a layer
- The convolutional base of VGG16 has 14,714,688 parameters, the added classifier has 2 mill.

Listing 5.20. Adding a densely connected classifier on top of the convolutional base

```
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'))
```

```
>>> model.summary()
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Model)                (None, 4, 4, 512)         14714688
_________________________________________________________________
flatten_1 (Flatten)          (None, 8192)              0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               2097408
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 257
=================================================================
Total params: 16,812,353
Trainable params: 16,812,353
Non-trainable params: 0
```

5.3.1:
Feature extraction with data augmentation (2)

```
>>> 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
>>> conv_base.trainable = False
>>> 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
```

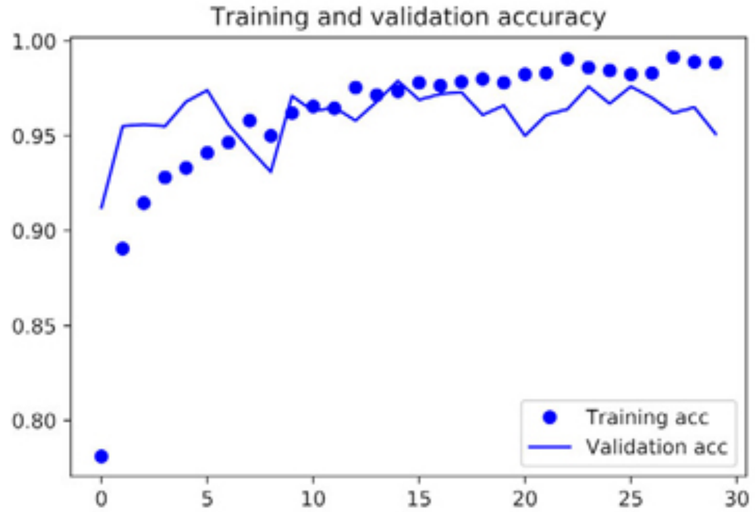
- In Keras, you freeze a model by setting its **trainable** attribute to False
- Only the weights from the two Dense layers will be trained, four weight tensors: two per layer (the main weight matrix and the bias vector).
- Now you can start to train your model (with data-augmentation).

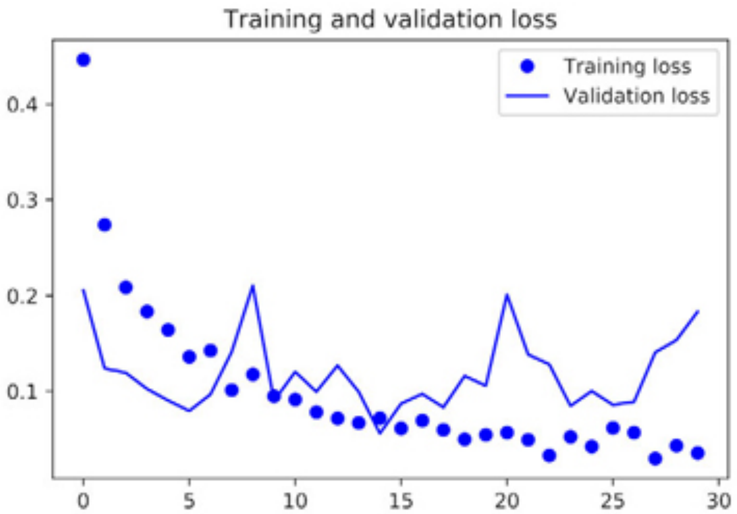
Listing 5.21. Training the model end to end with a frozen convolutional base

```
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')

test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        train_dir,
        target_size=(150, 150),
        batch_size=20,
        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)
```

5.3.1:
Feature extraction with data augmentation (3)

- We get an accuracy (validation) of about **96%** (much better then the train from scratch approach).

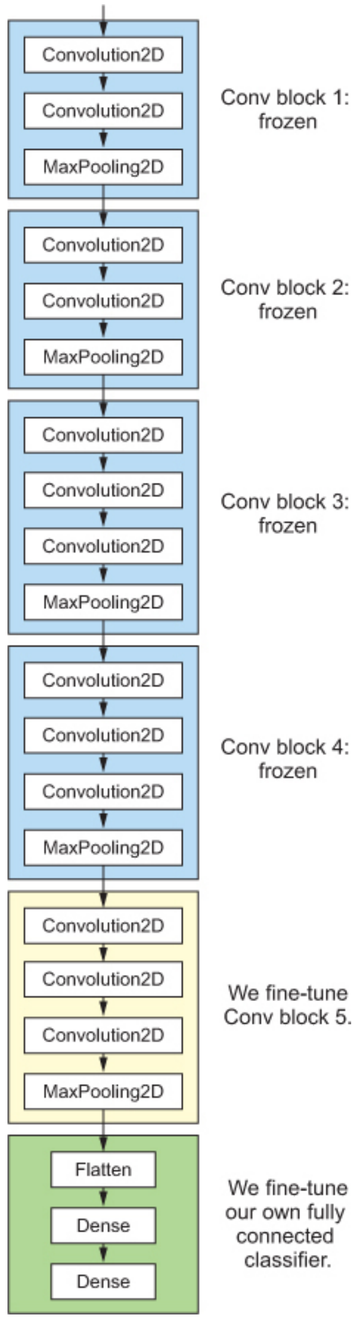




5.3.2. Fine-tuning

- **Fine-tuning** (alt. model reuse method):
  - unfreezing a few of the top layers of a frozen model base
  - jointly training both the newly added part of the model (in this case, the fully connected classifier) and these top layers
  - slightly adjusts the more abstract representations of the model being reused, in order to make them more relevant for the problem at hand

Figure 5.19. Fine-tuning the last convolutional block of the VGG16 network



5.3.2. Fine-tuning (2)

- A reminder of the convolutional base
- Steps for fine-tuning a network (NB: it's only possible to fine-tune the top layers of the convolutional base once the classifier on top has already been trained)
  - Add your custom network on top of an already-trained base network.
  - Freeze the base network.
  - Train the part you added.
  - **Unfreeze some layers in the base network**.
  - Jointly train both these layers and the part you added.
- Layers up to block4_pool should be frozen, and the layers block5_conv1, block5_conv2, and block5_conv3 should be trainable.

```
>>> conv_base.summary()
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 150, 150, 3)       0
_________________________________________________________________
block1_conv1 (Convolution2D) (None, 150, 150, 64)      1792
_________________________________________________________________
block1_conv2 (Convolution2D) (None, 150, 150, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 75, 75, 64)        0
_________________________________________________________________
block2_conv1 (Convolution2D) (None, 75, 75, 128)       73856
_________________________________________________________________
block2_conv2 (Convolution2D) (None, 75, 75, 128)       147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 37, 37, 128)       0
_________________________________________________________________
block3_conv1 (Convolution2D) (None, 37, 37, 256)       295168
_________________________________________________________________
block3_conv2 (Convolution2D) (None, 37, 37, 256)       590080
_________________________________________________________________
block3_conv3 (Convolution2D) (None, 37, 37, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 18, 18, 256)       0
_________________________________________________________________
block4_conv1 (Convolution2D) (None, 18, 18, 512)       1180160
_________________________________________________________________
block4_conv2 (Convolution2D) (None, 18, 18, 512)       2359808
_________________________________________________________________
block4_conv3 (Convolution2D) (None, 18, 18, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 9, 9, 512)         0


_________________________________________________________________
block5_conv1 (Convolution2D) (None, 9, 9, 512)         2359808
_________________________________________________________________
block5_conv2 (Convolution2D) (None, 9, 9, 512)         2359808
_________________________________________________________________
block5_conv3 (Convolution2D) (None, 9, 9, 512)         2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 4, 4, 512)         0
=================================================================
Total params: 14714688
```

5.3.2. Fine-tuning (3)

- Why not fine-tune more layers? Why not fine-tune the entire convolutional base?
  - Earlier layers encode more-generic, reusable features, whereas layers higher up encode more-specialized features (which needs to be repurposed)
  - The more parameters you're training, the more you're at risk of overfitting, especially when you attempt to train it on a small dataset.
- Good strategy to fine-tune only the top two or three layers
- Starting from where you left off in the previous example:
  - Freeze the last block in the base
  - Fine-tuning the network using a very low learning rate

Listing 5.22. Freezing all layers up to a specific one

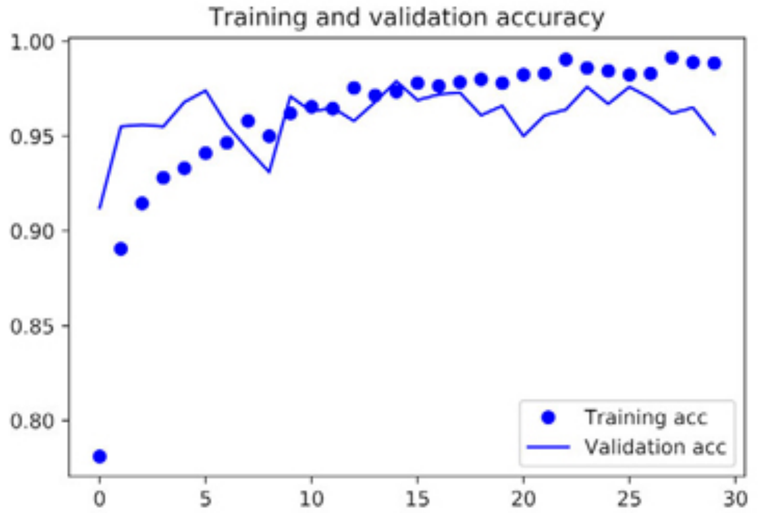
```
conv_base.trainable = True

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
```

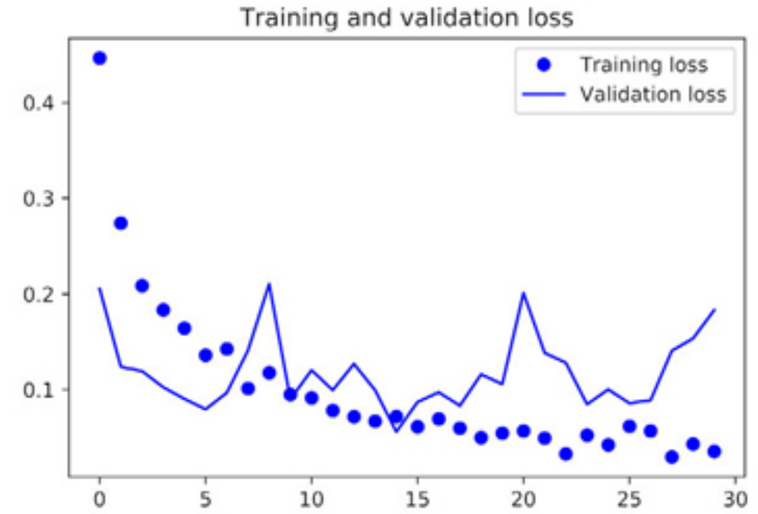
Listing 5.23. Fine-tuning the model

```
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)
```

5.3.2. Fine-tuning (4)

- Plot the results
  - Curves look noisy: smooth them by replacing every loss and accuracy with exponential moving averages of these quantities - see next slide.

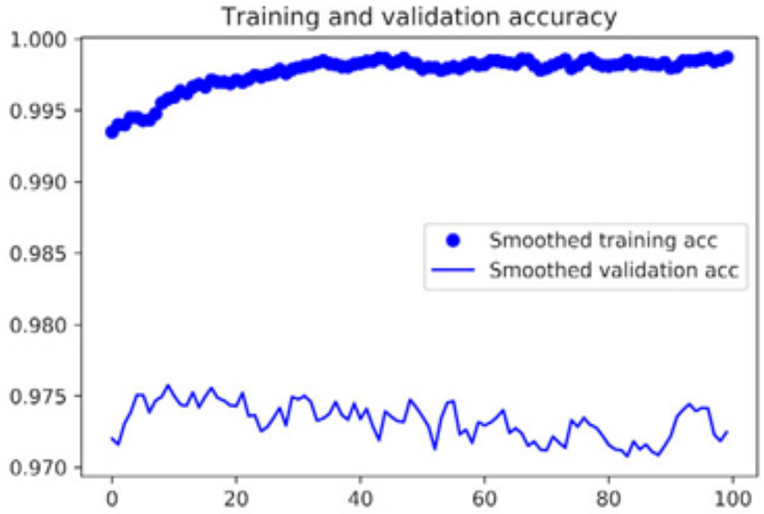


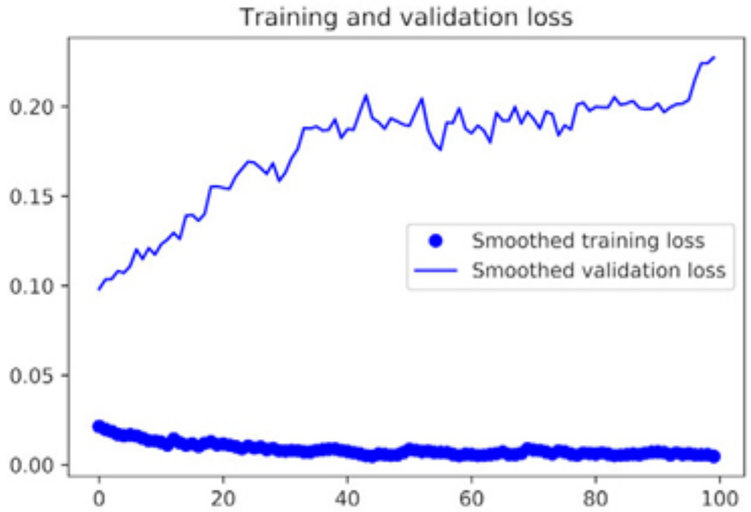


5.3.2. Fine-tuning (5)

Listing 5.24. Smoothing the plots

```
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()
```





5.3.2. Fine-tuning (6)

```
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)
```

- The validation accuracy curve look much cleaner. You're seeing a nice 1% absolute improvement in accuracy, from about 96% to above 97%
- Evaluation of the model on the test data show an accuracy of 97%
- This result is better than the winner of the Kaggle competition, **using only 10% av the training data (2' vs. 20').** This is make possible by modern deep-learning techniques.

5.3.3. Wrapping up

- Convnets are the best type of machine-learning models for computer-vision tasks
- A convnets can be train from scratch even on a very small dataset, with decent results.
  - Small dataset, overfitting, data augmentation
- It’s easy to reuse an existing convnet on a new dataset via feature extraction
- Fine-tuning adapts to a new problem some of the representations previously learned by an existing model.

5.4. Visualizing what convnets learn

- DL = “black boxes”, i.e. hard to understand and interpret.
- Convnets: representations of visual concepts, can be visualized.
- Techniques for visualizing:
  - intermediate **activations**: understanding how layers **transform** their input and getting the **meaning** of individual convnet filters
  - convnets **filters**: understanding precisely what visual **pattern** or **concept** each **filter** in a convnet is **receptive to**
  - **heatmaps** of class activation in an image: understanding which **parts** of an image were identified as belonging to a given class, thus allowing you to **localize** objects in images.

5.4.1. Visualizing intermediate activations

- Display the output **feature maps** from convolution and pooling layers, i.e. the **activations** (output from activation functions)
- View into how an input is **decomposed** into the different **filters** learned by the network.
- Each **channel** encodes different **features**, show the contents of every channel as a 2D **image**

```
>>> from keras.models import load_model
>>> model = load_model('cats_and_dogs_small_2.h5')
>>> model.summary()  <1> As a reminder.
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 148, 148, 32)      896
_________________________________________________________________
maxpooling2d_5 (MaxPooling2D)(None, 74, 74, 32)        0

_________________________________________________________________
conv2d_6 (Conv2D)            (None, 72, 72, 64)        18496
_________________________________________________________________
maxpooling2d_6 (MaxPooling2D)(None, 36, 36, 64)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 34, 34, 128)       73856
_________________________________________________________________
maxpooling2d_7 (MaxPooling2D)(None, 17, 17, 128)       0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 15, 15, 128)       147584
_________________________________________________________________
maxpooling2d_8 (MaxPooling2D)(None, 7, 7, 128)         0
_________________________________________________________________
flatten_2 (Flatten)          (None, 6272)              0
_________________________________________________________________
dropout_1 (Dropout)          (None, 6272)              0
_________________________________________________________________
dense_3 (Dense)              (None, 512)               3211776
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 513
=================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
```

5.4.1. Visualizing intermediate activations

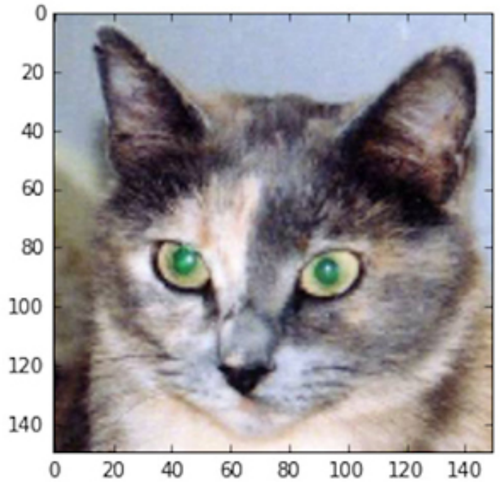
- Get a cat image (not part of the training set).
- Display the image

Get a cat image (not used for training)

Listing 5.25. Preprocessing a single image

```
img_path = '/Users/fchollet/Downloads/cats_and_dogs_small/test/cats/cat.1700.
                                                                              1
from keras.preprocessing import image
import numpy as np

img = image.load_img(img_path, target_size=(150, 150))
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
img_tensor /= 255.                                                            2

<1> Its shape is (1, 150, 150, 3)
print(img_tensor.shape)
```

Figure 5.24. The test cat picture



Listing 5.26. Displaying the test picture

```
import matplotlib.pyplot as plt

plt.imshow(img_tensor[0])
plt.show()
```

5.4.1. Visualizing intermediate **activations** (2)

- Create a Keras model that takes images as input and outputs the **activations** of all convolution and pooling layers (class Model, multiple outputs)

- Extracts the outputs of the top eight layers

- Creates a model that will return these outputs, given the model input

- One input (image) and eight outputs: one output per layer activation

- Activation of the first convolution layer for the cat image input: a 148×148 feature map with 32 channels (which can be plotted)

Listing 5.27. Instantiating a model from an input tensor and a list of output tensors

```
from keras import models

layer_outputs = [layer.output for layer in model.layers[:8]]
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

Listing 5.28. Running the model in predict mode

```
activations = activation_model.predict(img_tensor)
```

```
>>> first_layer_activation = activations[0]

>>> print(first_layer_activation.shape)
(1, 148, 148, 32)
```

5.4.1. Visualizing intermediate **activations** (3)

- First layer activations: 148×148 feature map with 32 channels
  - Plot fourth channel: edge detector.
  - Plot seventh channel: dot detector

Listing 5.29. Visualizing the fourth channel

```
1  import matplotlib.pyplot as plt
2
3  plt.matshow(first_layer_activation[0, :, :, 4], cmap='viridis')
```

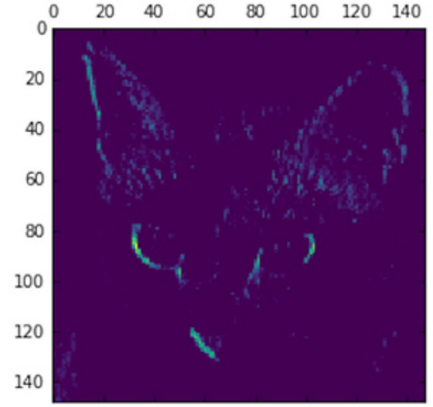


Figure 5.25.
Fourth channel of the activation of the **first layer** on the test cat picture

This **channel** appears to encode a **diagonal edge** detector

Listing 5.30. Visualizing the seventh channel

```
1  plt.matshow(first_layer_activation[0, :, :, 7], cmap='viridis')
```

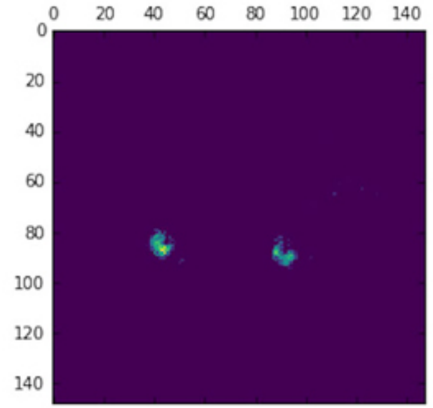


Figure 5.26.
Seventh channel of the activation of the **first layer** on the test cat picture

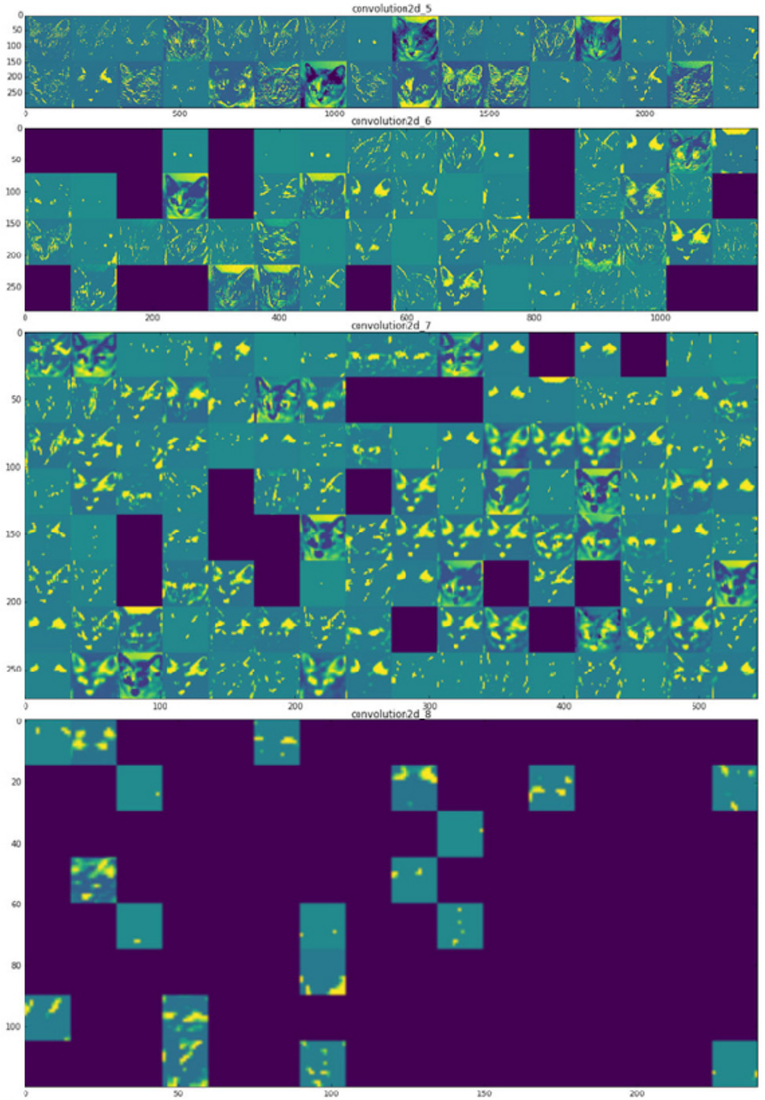
“bright green dot” detector, useful to encode cat eyes

5.4.1. Visualizing intermediate **activations** (4)

Listing 5.31. Visualizing every channel in every intermediate activation

```
layer_names = []
for layer in model.layers[:8]:
    layer_names.append(layer.name)

images_per_row = 16

for layer_name, layer_activation in zip(layer_names, activations):
    n_features = layer_activation.shape[-1]

    size = layer_activation.shape[1]

    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                             :, :,
                                             col * images_per_row + row]
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                        scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')
```

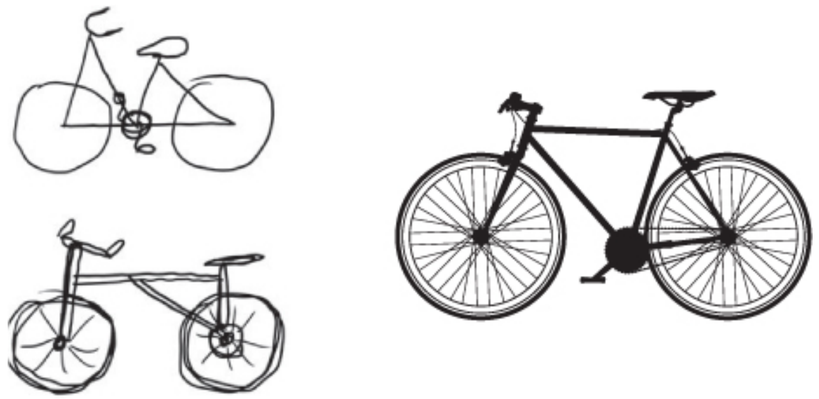
Figure 5.27. Every **channel** of every **layer** activation on the test **cat** picture



5.4.1. Visualizing intermediate **activations** (5)

- Things to note:
  - First layer acts as a collection of various edge detectors. Activations retain most info in image.
  - Higher layers: activations become increasingly abstract (encode higher-level concepts) and less visually interpretable. Less info about **visual** contents and more info about **class**.
  - Sparsity of the activations increases with the depth of the layer. More and more filters are blank, this means the **pattern encoded by the filter** isn't found in the input image.
- Deep NNs act as information distillation pipelines, analogous to the way humans perceive the world

Figure 5.28. Left: attempts to draw a bicycle from memory. Right: what a schematic bicycle should look like.



5.4.2. Visualizing convnet filters

- Display the visual pattern that each filter is meant to respond to.
- Find the input image that the chosen filter is maximally responsive to (starting from a blank input image).
- Gradient ascent in input space: applying gradient descent to the value of the input image of a convnet so as to maximize the response of a specific filter
- Process: build a **loss** function that **maximizes** the value of a given **filter** in a given convolution **layer**, and then you'll use stochastic **gradient descent** to adjust the values of the **input image** so as to maximize this activation value.

5.4.2. Visualizing convnet filters (2)

1. Builds a **loss** function that maximizes the activation of the nth filter of the layer under consideration

2. Computes the **gradient** of the input picture with regard to this loss

3. Normalization trick: **normalizes** the gradient

4. Returns the loss and grads given the input picture

5. Starts from a gray image with some noise

6. Runs gradient **ascent** for 40 steps

Listing 5.38. Function to generate filter visualizations

```
def generate_pattern(layer_name, filter_index, size=150):
    layer_output = model.get_layer(layer_name).output                    1
    loss = K.mean(layer_output[:, :, :, filter_index])                   1

    grads = K.gradients(loss, model.input)[0]                            2

    grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)                    3

    iterate = K.function([model.input], [loss, grads])                   4

    input_img_data = np.random.random((1, size, size, 3)) * 20 + 128.    5

    step = 1.                                                            6
    for i in range(40):                                                  6
        loss_value, grads_value = iterate([input_img_data])              6
        input_img_data += grads_value * step                             6

    img = input_img_data[0]
    return deprocess_image(img)
```

5.4.2. Visualizing convnet filters (3)

Listing 5.38. Function to generate filter visualizations

```
def generate_pattern(layer_name, filter_index, size=150):
    layer_output = model.get_layer(layer_name).output                    1
    loss = K.mean(layer_output[:, :, :, filter_index])                   1

    grads = K.gradients(loss, model.input)[0]                            2

    grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)                    3

    iterate = K.function([model.input], [loss, grads])                   4

    input_img_data = np.random.random((1, size, size, 3)) * 20 + 128.    5

    step = 1.                                                            6
    for i in range(40):                                                  6
        loss_value, grads_value = iterate([input_img_data])              6
        input_img_data += grads_value * step                             6

    img = input_img_data[0]
    return deprocess_image(img)
```

Listing 5.37. Utility function to convert a tensor into a valid image

```
def deprocess_image(x):
    x -= x.mean()
    x /= (x.std() + 1e-5)
    x *= 0.1

    x += 0.5
    x = np.clip(x, 0, 1)

    x *= 255
    x = np.clip(x, 0, 255).astype('uint8')
    return x
```

Listing 5.36. Loss maximization via stochastic gradient descent

```
input_img_data = np.random.random((1, 150, 150, 3)) * 20 + 128.

step = 1.
for i in range(40):
    loss_value, grads_value = iterate([input_img_data])

    input_img_data += grads_value * step
```

Listing 5.32. Defining the loss tensor for filter visualization

```
from keras.applications import VGG16
from keras import backend as K

model = VGG16(weights='imagenet',
              include_top=False)

layer_name = 'block3_conv1'
filter_index = 0

layer_output = model.get_layer(layer_name).output
loss = K.mean(layer_output[:, :, :, filter_index])
```

Listing 5.33. Obtaining the gradient of the loss with regard to the input

```
grads = K.gradients(loss, model.input)[0]          1
```

Listing 5.34. Gradient-normalization trick

```
ads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)
```

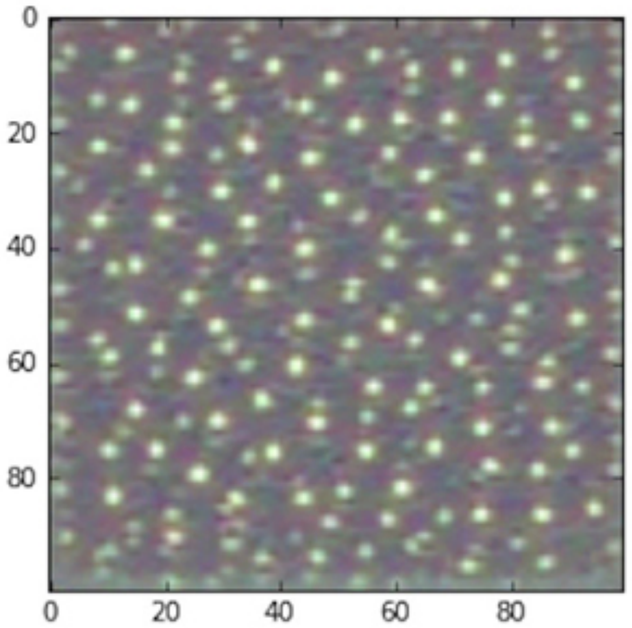
Listing 5.35. Fetching Numpy output values given Numpy input values

```
iterate = K.function([model.input], [loss, grads])

import numpy as np
loss_value, grads_value = iterate([np.zeros((1, 150, 150, 3))])
```

5.4.2. Visualizing convnet filters (4)

```
>>> plt.imshow(generate_pattern('block3_conv1', 0))
```

Figure 5.29. **Pattern** that the zeroth channel in layer block3_conv1 **responds** to maximally



Listing 5.39. Generating a grid of all filter response patterns in a layer

```
layer_name = 'block1_conv1'
size = 64
margin = 5

results = np.zeros((8 * size + 7 * margin, 8 * size + 7 * margin, 3))

for i in range(8):
    for j in range(8):
        filter_img = generate_pattern(layer_name, i + (j * 8), size=size)

        horizontal_start = i * size + i * margin
        horizontal_end = horizontal_start + size
        vertical_start = j * size + j * margin
        vertical_end = vertical_start + size
        results[horizontal_start: horizontal_end,
                vertical_start: vertical_end, :] = filter_img

plt.figure(figsize=(20, 20))
plt.imshow(results)
```

5.4.2. Visualizing convnet **filters** (5)

Figure 5.30. Filter **patterns** for layer **block1_conv1**

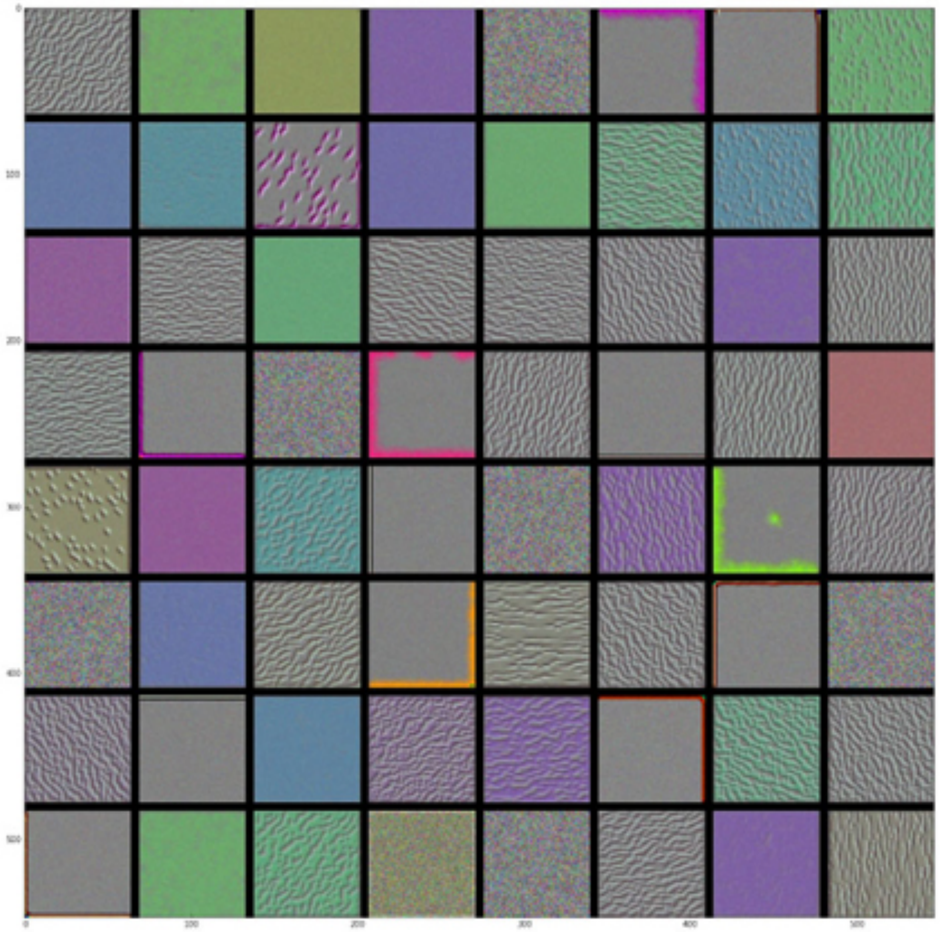
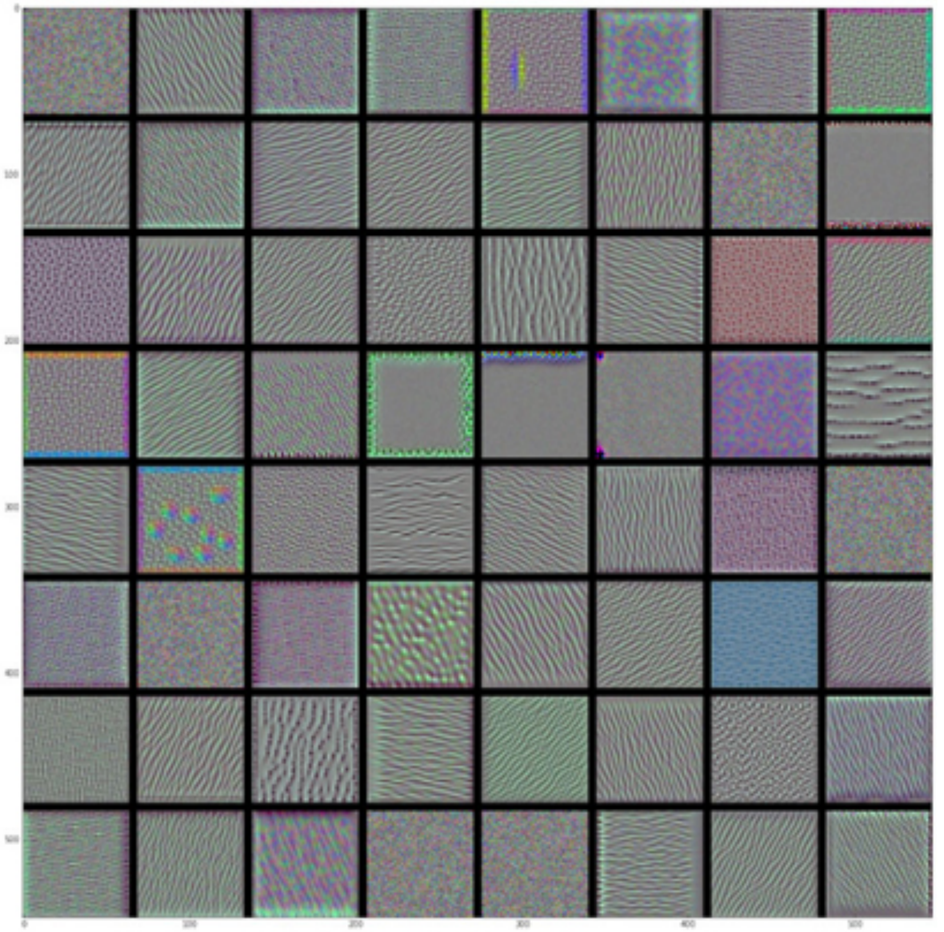


Figure 5.31. Filter **patterns** for layer **block2_conv1**



5.4.2. Visualizing convnet **filters** (6)

Figure 5.32. Filter **patterns** for layer **block3_conv1**

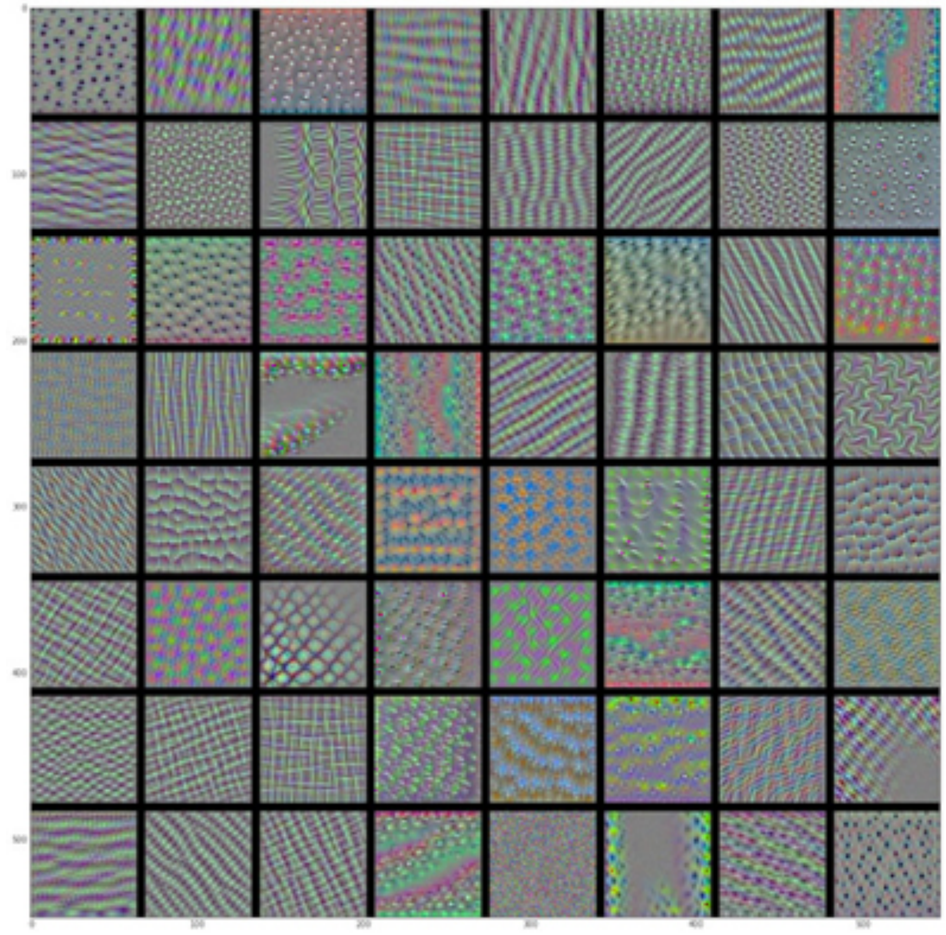
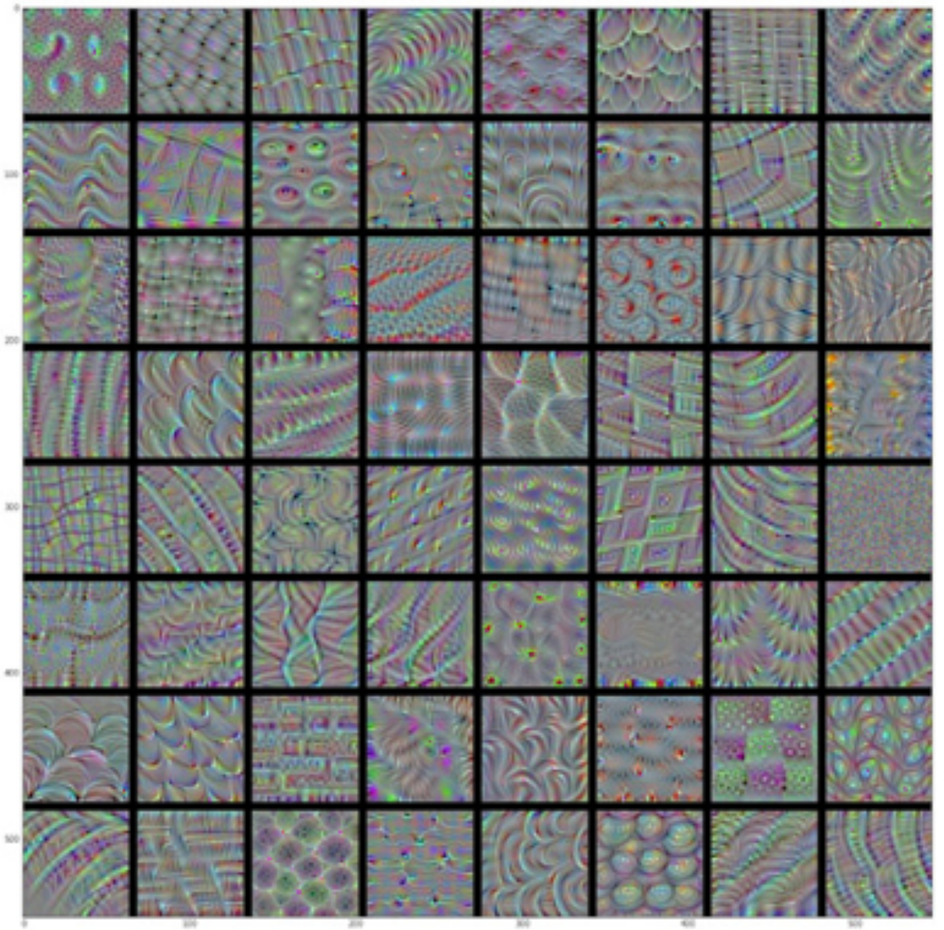


Figure 5.33. Filter **patterns** for layer **block4_conv1**



5.4.2. Visualizing convnet filters (7)

- Filter visualizations tell you a lot about how convnet layers see the world
  - each layer in a convnet learns a collection of filters
  - such that their inputs can be expressed as a combination of the filters (similar to how the Fourier transform decomposes signals onto a bank of cosine functions)
  - The filters in these convnet filter banks get increasingly complex and refined as you go higher in the model:
    - filters from the first layer in the model encode **simple directional edges and colors** (or colored edges, in some cases).
    - filters from the next layer encode **simple textures** made from **combinations** of edges and colors.
    - filters in higher layers begin to resemble textures found in natural images (e.g eyes, leaves etc.)

5.4.3. Visualizing **heatmaps** of class activation

- Understanding which **parts** of a given image led a convnet to its final classification decision.

- Helpful for **debugging** the decision process of a convnet (classification mistake) and locate specific objects in an image.

- Class activation map (CAM) visualization: produce heatmaps of class activation over input images (a 2D grid of scores associated with a specific output class, computed for every location in any input image, indicating how important each location is with respect to the class under consideration, indicating how cat/dog-like different parts of the image are)

5.4.3. Visualizing **heatmaps** of class activation (2)

- Implementation: “**Grad-CAM**: Visual Explanations from Deep Networks via Gradient-based Localization

  - **Idea**: take the output feature map of a convolution layer, given an input image, and weighing every channel in that feature map by the gradient of the class with respect to the channel

  - **Intuitively:** you’re weighting a spatial map of “how intensely the input image activates different channels” by “how important each channel is with regard to the class,” resulting in a spatial map of “how intensely the input image activates the class.”

5.4.3

- Example using a pretrained VGG16 network including the densely connected classifier on top.
- Consider the image of two African elephants (calf and mother)
- Convert image:
  - size 224 × 244, Numpy float32 tensor, apply keras.applications.vgg16.preprocess_input

Listing 5.40. Loading the VGG16 network with pretrained weights

```
from keras.applications.vgg16 import VGG16

model = VGG16(weights='imagenet')
```

Figure 5.34. Test picture of African elephants

Listing 5.41. Preprocessing an input image for VGG16

```
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input, decode_predictions
import numpy as np

img_path = '/Users/fchollet/Downloads/creative_commons_elephant.jpg'

img = image.load_img(img_path, target_size=(224, 224))

x = image.img_to_array(img)

x = np.expand_dims(x, axis=0)

x = preprocess_input(x)
```

5.4.3. Visualizing **heatmaps** of class activation (4)

- Top three classes predicted:
  - African elephant (with 92.5% probability)
  - Tusker (with 7% probability)
  - Indian elephant (with 0.4% probability)
-

```
>>> preds = model.predict(x)
>>> print('Predicted:', decode_predictions(preds, top=3)[0])
Predicted:', [(u'n02504458', u'African_elephant', 0.92546833),
(u'n01871265', u'tusker', 0.070257246),
(u'n02504013', u'Indian_elephant', 0.0042589349)]
```

```
>>> np.argmax(preds[0])
386
```

5.4.3. Visualizing heatmaps of class activation (5)

- Grad-CAM process:
  - 1) “African elephant” entry in the prediction vector
  - 2) Output feature map of the block5_conv3 layer, the last convolutional layer in VGG16
  - 3) Gradient of the “African elephant” class with regard to the output feature map of block5_conv3
  - 4) Vector of shape (512,), where each entry is the mean intensity of the gradient over a specific feature-map channel
  - 5) Lets you access the values of the quantities you just defined: pooled_grads and the output feature map of block5_conv3, given a sample image
  - 6) Values of these two quantities, as Numpy arrays, given the sample image of two elephants
  - 7) Multiplies each channel in the feature-map array by “how important this channel is” with regard to the “elephant” class
  - 8) The channel-wise mean of the resulting feature map is the heatmap of the class activation

Listing 5.42. Setting up the Grad-CAM algorithm

```
african_e66lephant_output = model.output[:, 386]                          1
                                                                           2
last_conv_layer = model.get_layer('block5_conv3')                          3
                                                                           4
grads = K.gradients(african_elephant_output, last_conv_layer.output)[0]    5
                                                                           6
pooled_grads = K.mean(grads, axis=(0, 1, 2))                               7
                                                                           8
iterate = K.function([model.input],                                        9
                     [pooled_grads, last_conv_layer.output[0]])            10
                                                                           11
pooled_grads_value, conv_layer_output_value = iterate([x])                12
                                                                           13
for i in range(512):                                                       14
    conv_layer_output_value[:, :, i] *= pooled_grads_value[i]              15
                                                                           16
heatmap = np.mean(conv_layer_output_value, axis=-1)                        17
```

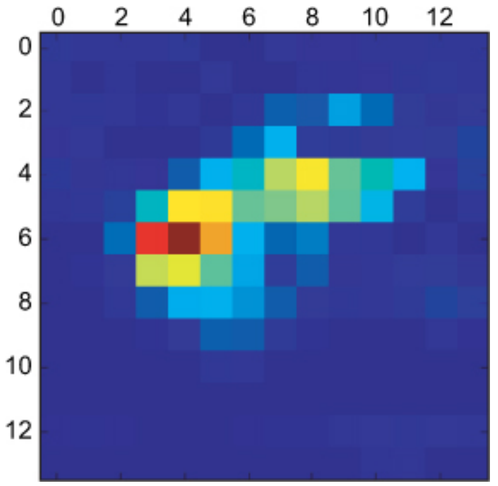
5.4.3. Visualizing heatmaps of class activation (6)

- Visualization technique answers two important questions:
  - Why did the network think this image contained an African elephant?
  - Where is the African elephant located in the picture?
- ears of the elephant calf are strongly activated: probably difference between African and Indian elephants.

Listing 5.43. Heatmap post-processing

```
heatmap = np.maximum(heatmap, 0)
heatmap /= np.max(heatmap)
plt.matshow(heatmap)
```

Figure 5.35. African elephant class activation heatmap over the test picture



Listing 5.44. Superimposing the heatmap with the original picture

```
import cv2

img = cv2.imread(img_path)                                              1

heatmap = cv2.resize(heatmap, (img.shape[1], img.shape[0]))             2

heatmap = np.uint8(255 * heatmap)                                       3

heatmap = cv2.applyColorMap(heatmap, cv2.COLORMAP_JET)                  4

superimposed_img = heatmap * 0.4 + img                                  5

cv2.imwrite('/Users/fchollet/Downloads/elephant_cam.jpg', superimposed_img)6
```

Figure 5.36. Superimposing the class activation heatmap on the original picture

Summary: Chap 5

- Convnets are the best tool for attacking **visual**-classification problems.
- Convnets work by learning a **hierarchy** of modular **patterns and concepts** to represent the visual world.
- The representations they learn are easy to **inspect**—convnets are the opposite of black boxes!
- You're now capable of training your own convnet from scratch to solve an image-classification problem.
- You understand how to use visual data **augmentation** to fight overfitting.
- You know how to use a **pretrained** convnet to do **feature extraction and fine-tuning**.
- You can generate **visualizations** of the filters learned by your convnets, as well as heatmaps of class activity.