Convolutional Networks (ConvNets): Part II

May 4, 2021

ddebarr@uw.edu

http://cross-entropy.net/ml530/Deep_Learning_3.pdf

https://en.wikipedia.org/wiki/Star_Wars_Day

Agenda for Tonight

- Homework Review

- [DLP] Chapter 5: Deep Learning for Computer Vision

The Three General Types of Network Layers

- Densely Connected Layers: each output value is based on all values from the previous layer (“global” features)

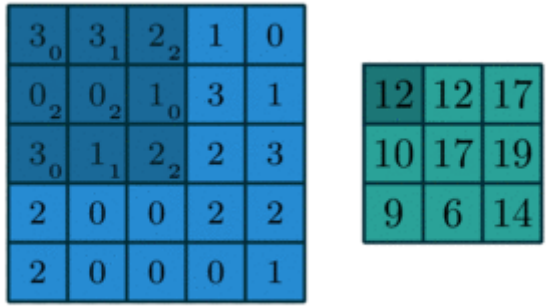
* We are here *

- Convolutional Layers: each output value is based on a spatially adjacent subset of values from the previous layer (“local” features)

- Embeddings and Recurrent Layers: soon [also Transformers]

Convolutional Operation Example

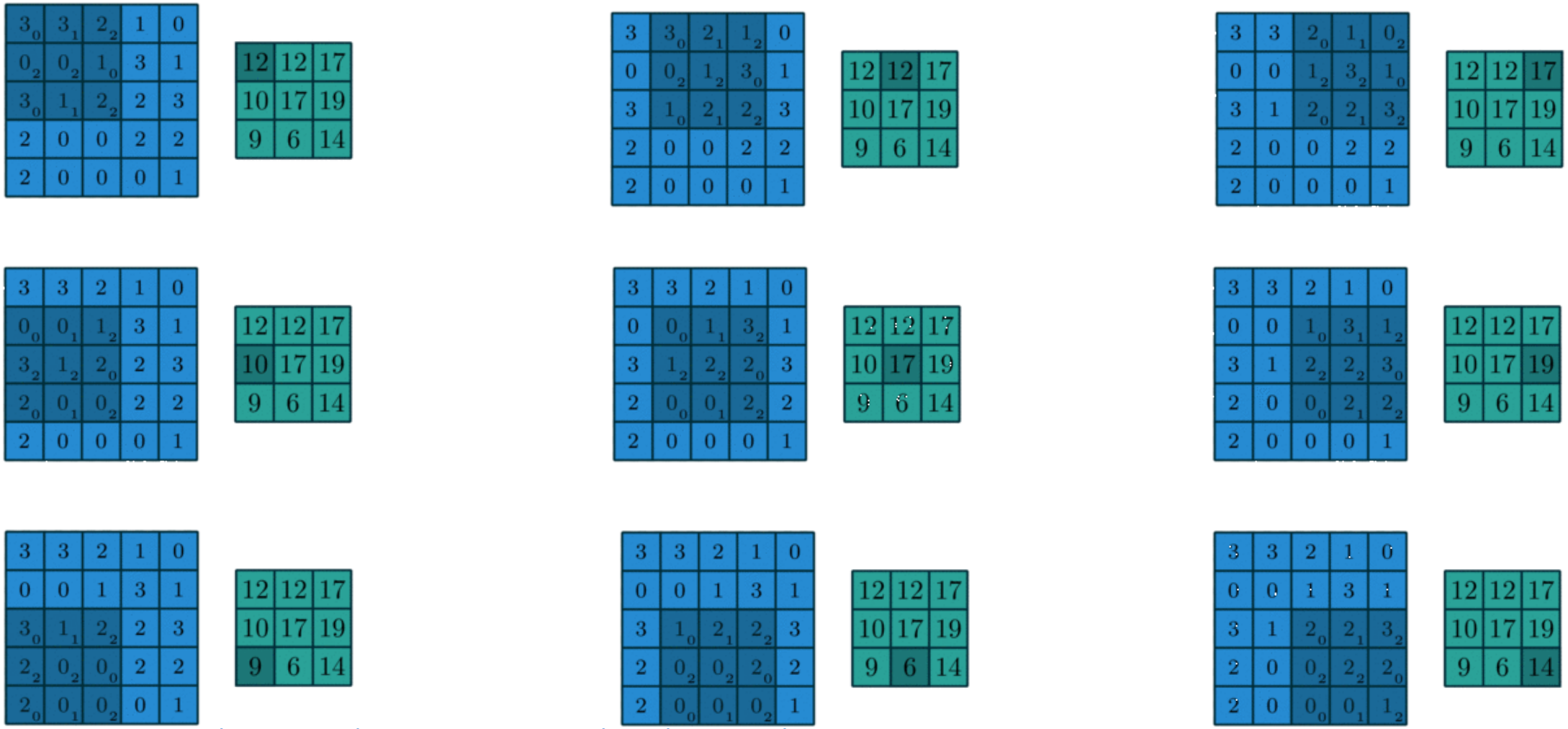
- Input Image: the 5x5, light-blue matrix
- Convolution Filter: the 3x3, dark-blue matrix
- Output Feature Map: the 3x3, green matrix
- The convolution filter is moved from left to right, and top to bottom, where each value in the output feature map is the sum of products between a 3x3 subimage of the input and the 3x3 convolution filter



http://deeplearning.net/software/theano_versions/dev/tutorial/conv_arithmetic.html

Computing the 9 Output Values for the Filter

One output channel (matrix) per filter. The stride determines the step size when moving to the right or down

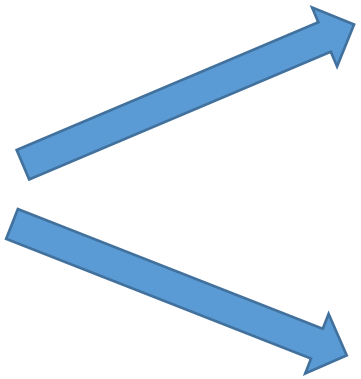


http://deeplearning.net/software/theano_versions/dev/tutorial/conv_arithmetic.html

Pooling

pool_size = (2, 2) with strides = (2, 2) reduces (4, 4) “image” to (2, 2)

0.03	0.16	0.13	0.04
0.10	0.09	0.14	0.12
0.05	0.11	0.06	0.02
0.01	0.07	0.15	0.08



0.16	0.14
0.11	0.15

Max pooling

0.0950	0.1075
0.0600	0.0775

Average pooling

[DLP] Deep Learning for Computer Vision

1. Introduction to ConvNets
2. Training a ConvNet from Scratch on a Small DataSet
3. Using a PreTrained ConvNet
4. Visualizing What ConvNets Learn

Instantiation 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'))
```

```
>>> 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
```

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'))
```

```
>>> 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
```

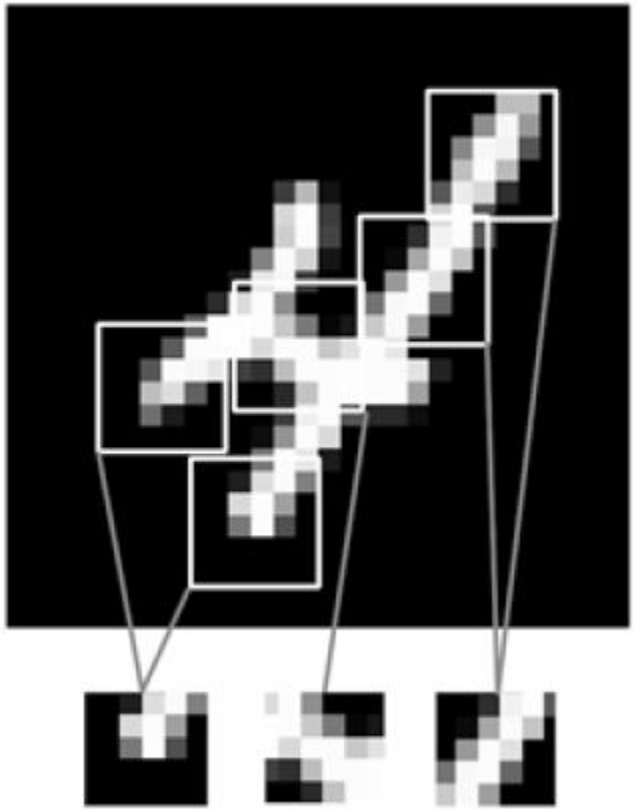
Training the ConvNet on MNIST Images

```
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)
```

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

Images Can Be Broken (Decomposed) Into Local Patterns

- Edges, Textures, etc.
- 5x5 patches

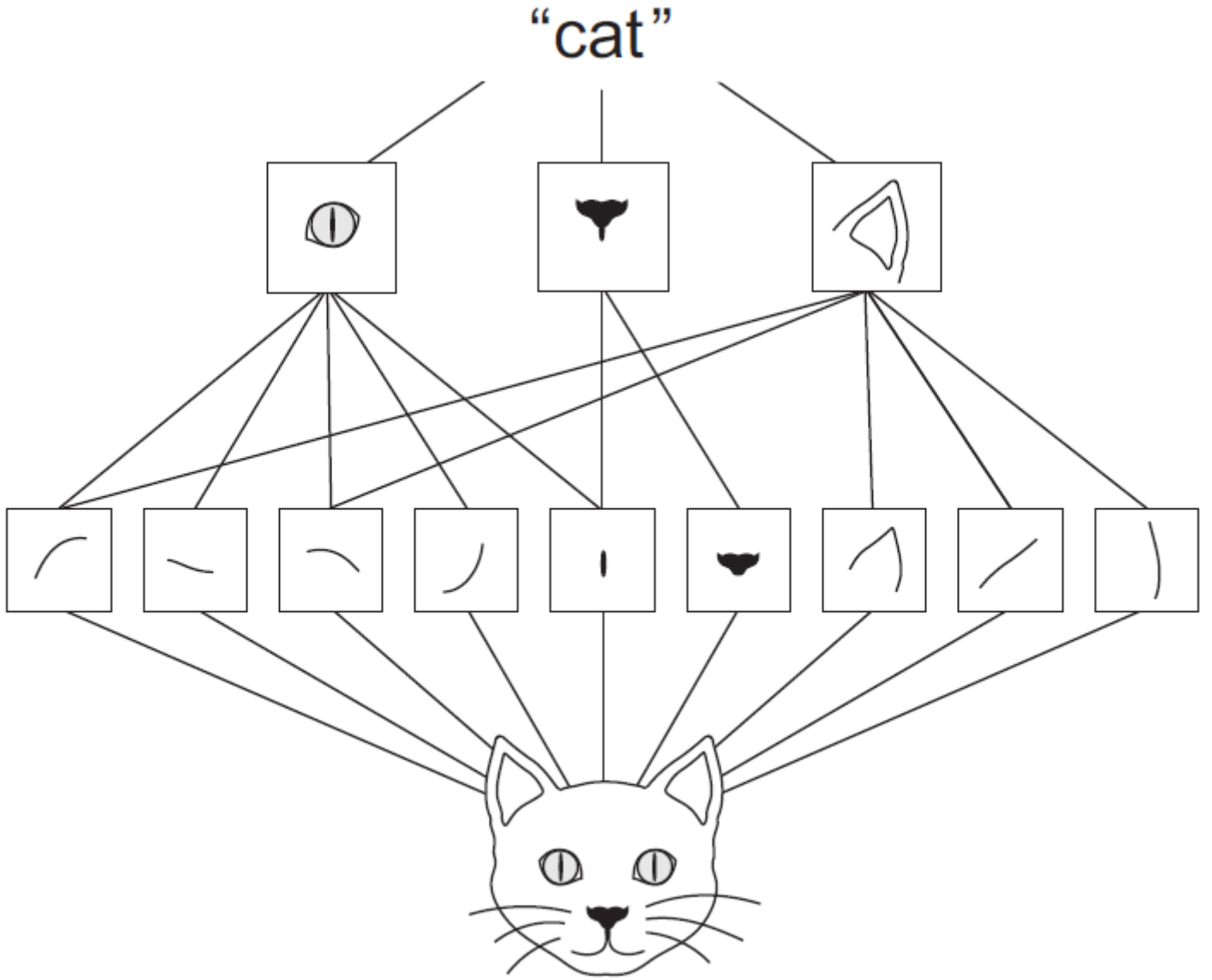


Local Patterns

- *The patterns they learn are translation invariant.* After learning a certain pattern in the lower-right corner of a picture, a convnet can recognize it anywhere: for example, in the upper-left corner. A densely connected network would have to learn the pattern anew if it appeared at a new location. This makes convnets data efficient when processing images (because *the visual world is fundamentally translation invariant*): they need fewer training samples to learn representations that have generalization power.
- *They can learn spatial hierarchies of patterns (see figure 5.2).* A first convolution layer will learn small local patterns such as edges, a second convolution layer will learn larger patterns made of the features of the first layers, and so on. This allows convnets to efficiently learn increasingly complex and abstract visual concepts (because *the visual world is fundamentally spatially hierarchical*).

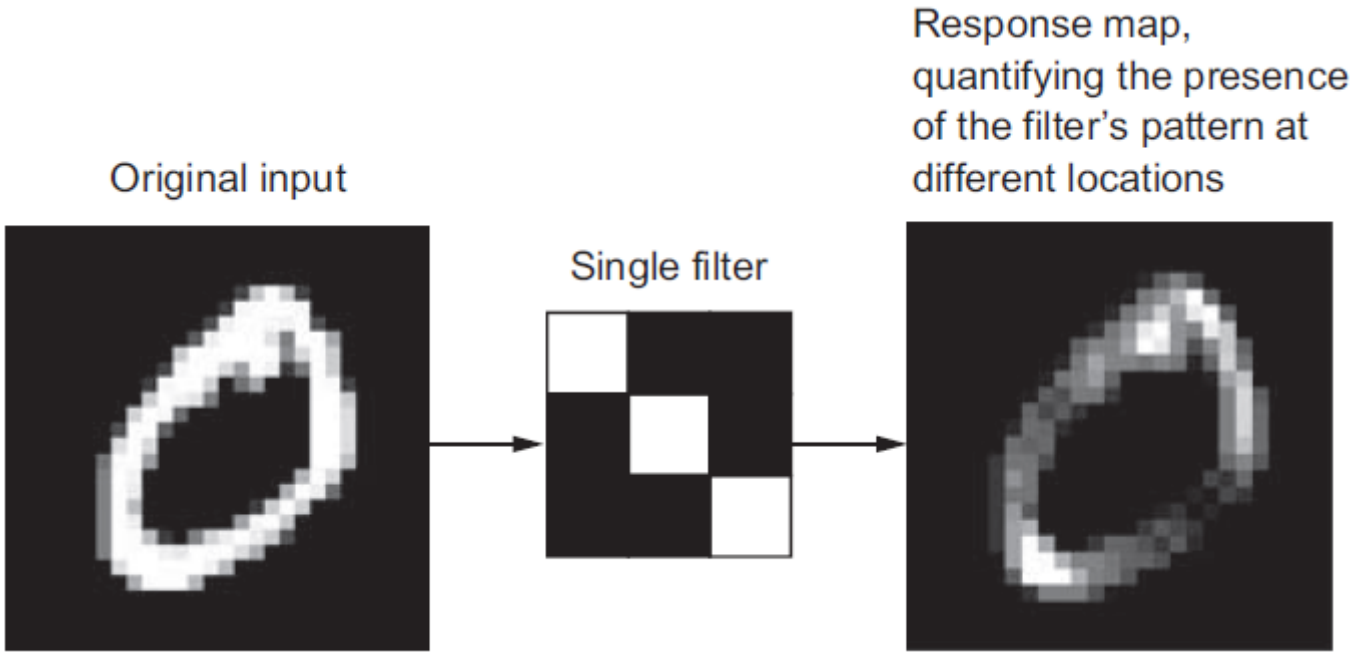
Local Edges Combine Into Local Objects

- Edges combine into objects, such as eyes, nose, and ears
- Eyes, nose, and ears combine into higher-level concepts, such as “cat”
- Read the picture from the bottom up [input to output]



Response Map

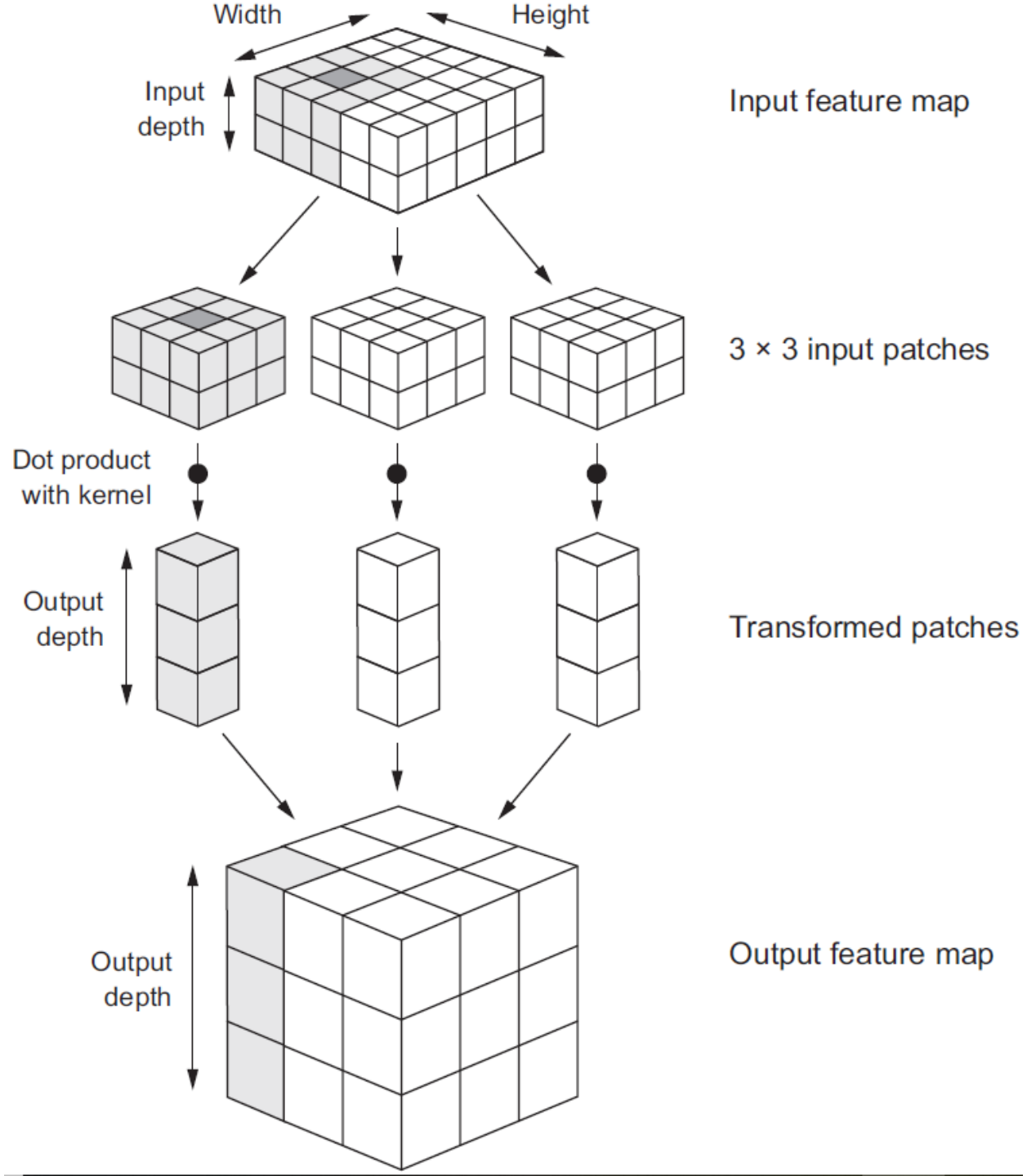
The Response Map is a two-dimensional map of the presence of a pattern at different feature locations in an input



Key Parameters for Convolutions

- *Size of the patches extracted from the inputs*—These are typically 3×3 or 5×5. In the example, they were 3×3, which is a common choice.
- *Depth of the output feature map*—The number of filters computed by the convolution. The example started with a depth of 32 and ended with a depth of 64.

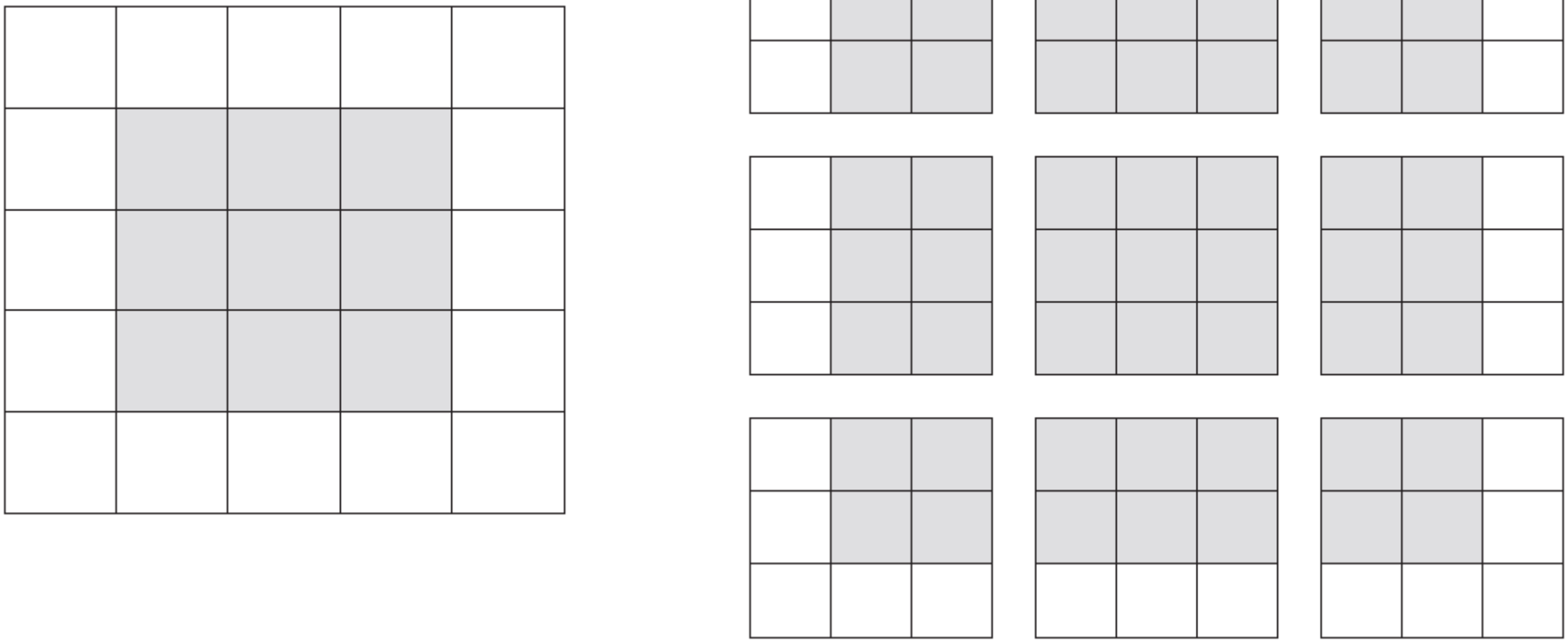
How Convolution Works



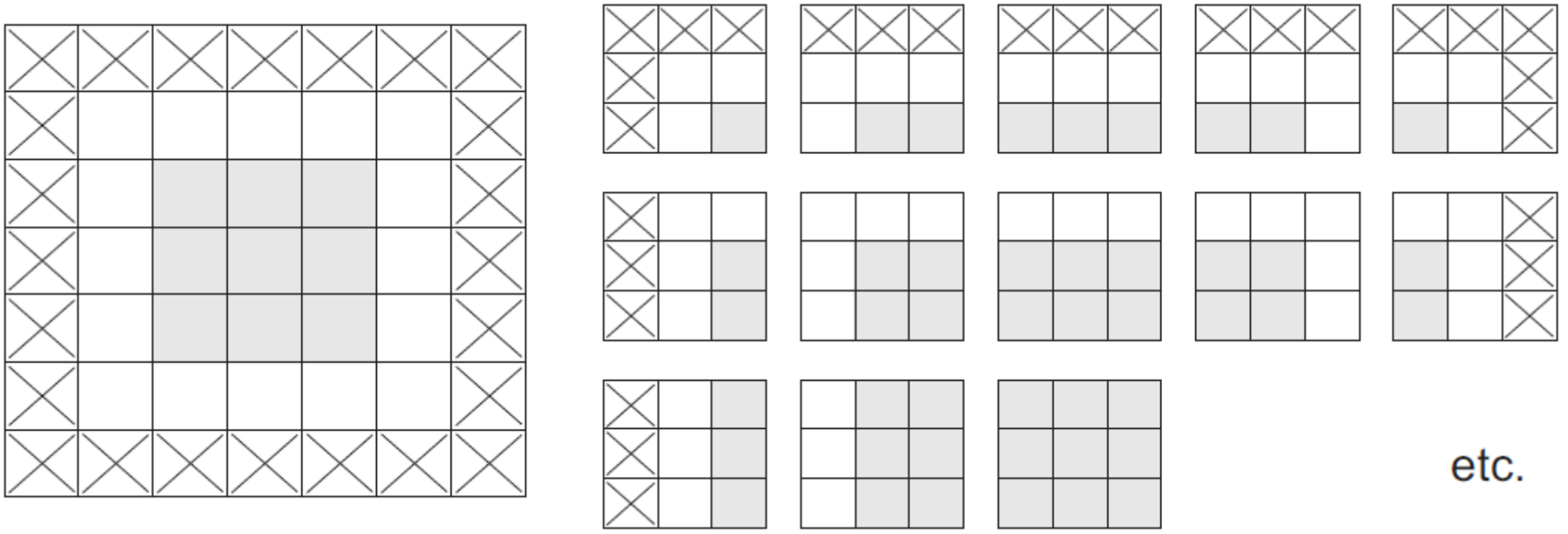
Output Width and Height May Differ from Input Width and Height

- Border effects, which can be countered by padding the input feature map
- The use of *strides*, which I'll define in a second

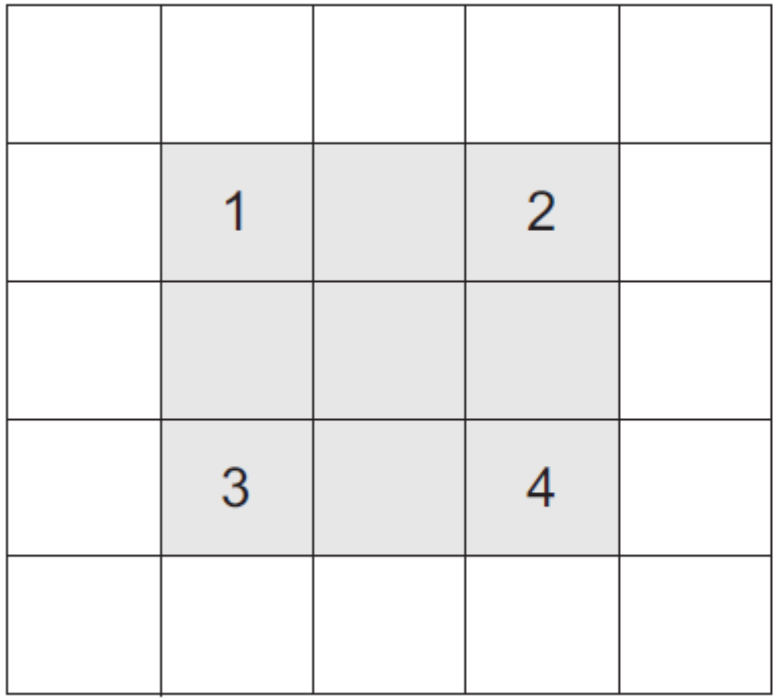
“Valid” Locations of 3x3 Convolution Patches for a 5x5 Image

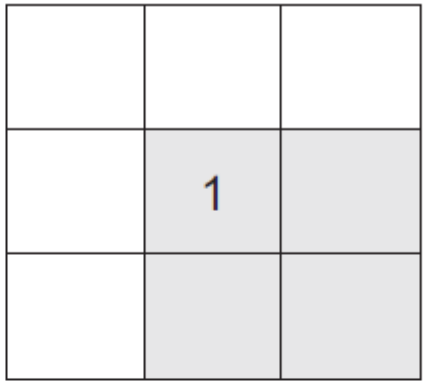


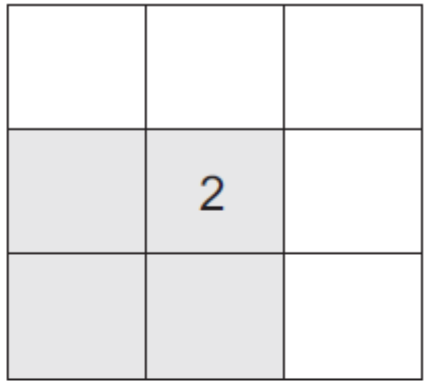
Padding a 5x5 Image for “Same” Size Output from a 3x3 Convolution Patch

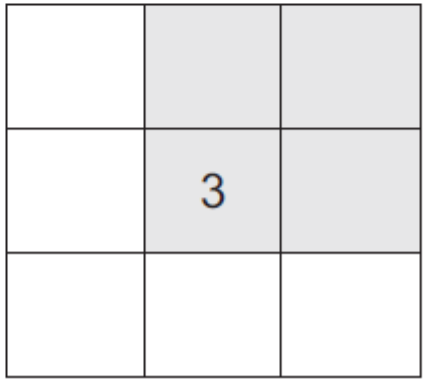


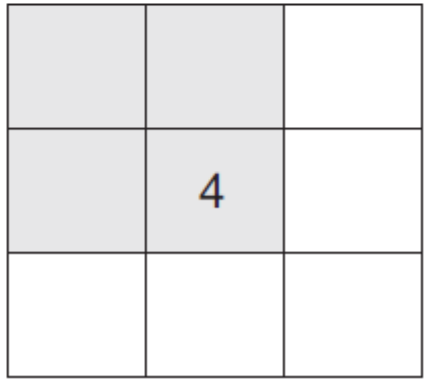
3x3 Convolution Patches with 2x2 Strides











Model Without Max Pooling

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

```
>>> model_no_max_pool.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 24, 24, 64)        18496
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 22, 22, 64)        36928
=================================================================
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
```

What's Wrong with Model Without Max Pooling?

- It isn't conducive to learning a spatial hierarchy of features. The 3×3 windows in the third layer will only contain information coming from 7×7 windows in the initial input. The high-level patterns learned by the convnet will still be very small with regard to the initial input, which may not be enough to learn to classify digits (try recognizing a digit by only looking at it through windows that are 7×7 pixels!). We need the features from the last convolution layer to contain information about the totality of the input.
- The final feature map has $22 \times 22 \times 64 = 30{,}976$ total coefficients per sample. This is huge. If you were to flatten it to stick a `Dense` layer of size 512 on top, that layer would have 15.8 million parameters. This is far too large for such a small model and would result in intense overfitting.

First layer: 3x3 patches; Second layer: 5x5 patches (3x3 of 3x3 patches); Third layer: 7x7 patches (3x3 of 5x5 patches)

Dogs versus Cats

Images are heterogeneous in size and appearance

http://www.kaggle.com/c/dogs-vs-cats/data

Preprocessing: Creating Partition Directories

- 25,000 images per class: 543 megabytes compressed
- For this exercise: 1,000 per class trn; 500 per class val; and 500 per class tst

Path to the directory where the original dataset was uncompressed

Directory where you'll store your smaller dataset

```
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)
```

Directories for the training, validation, and test splits

Preprocessing: Creating Class Directories

```
train_cats_dir = os.path.join(train_dir, 'cats')
os.mkdir(train_cats_dir)
```
Directory with training cat pictures

```
train_dogs_dir = os.path.join(train_dir, 'dogs')
os.mkdir(train_dogs_dir)
```
Directory with training dog pictures

```
validation_cats_dir = os.path.join(validation_dir, 'cats')
os.mkdir(validation_cats_dir)
```
Directory with validation cat pictures

```
validation_dogs_dir = os.path.join(validation_dir, 'dogs')
os.mkdir(validation_dogs_dir)
```
Directory with validation dog pictures

```
test_cats_dir = os.path.join(test_dir, 'cats')
os.mkdir(test_cats_dir)
```
Directory with test cat pictures

```
test_dogs_dir = os.path.join(test_dir, 'dogs')
os.mkdir(test_dogs_dir)
```
Directory with test dog pictures

Preprocessing: Copying Cat Files

```
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)
```

Copies the first 1,000 cat images to train_cats_dir

```
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)
```

Copies the next 500 cat images to validation_cats_dir

```
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)
```

Copies the next 500 cat images to test_cats_dir

Preprocessing: Copying Dog Files

```
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)
```

Copies the first 1,000 dog images to train_dogs_dir

```
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)
```

Copies the next 500 dog images to validation_dogs_dir

```
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)
```

Copies the next 500 dog images to test_dogs_dir

Sanity Check: File Counts

```
>>> 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
```

ConvNet for Dogs versus Cats: Definition

```
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'))
```

ConvNet for Dogs versus Cats: 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

Configuring the Model

```
from keras import optimizers

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

Steps for Creating Tensors

1. Read the picture files.
2. Decode the JPEG content to RGB grids of pixels.
3. Convert these into floating-point tensors.
4. 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).

Code for Creating Tensors

```
from keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(rescale=1./255)
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')
```

Rescales all images by 1/255

Target directory

Resizes all images to 150 × 150

Because you use binary_crossentropy loss, you need binary labels.

Understanding Python Generators

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

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

```
1
2
3
4
5
```

Example Train Batch

```
>>> 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,)
```

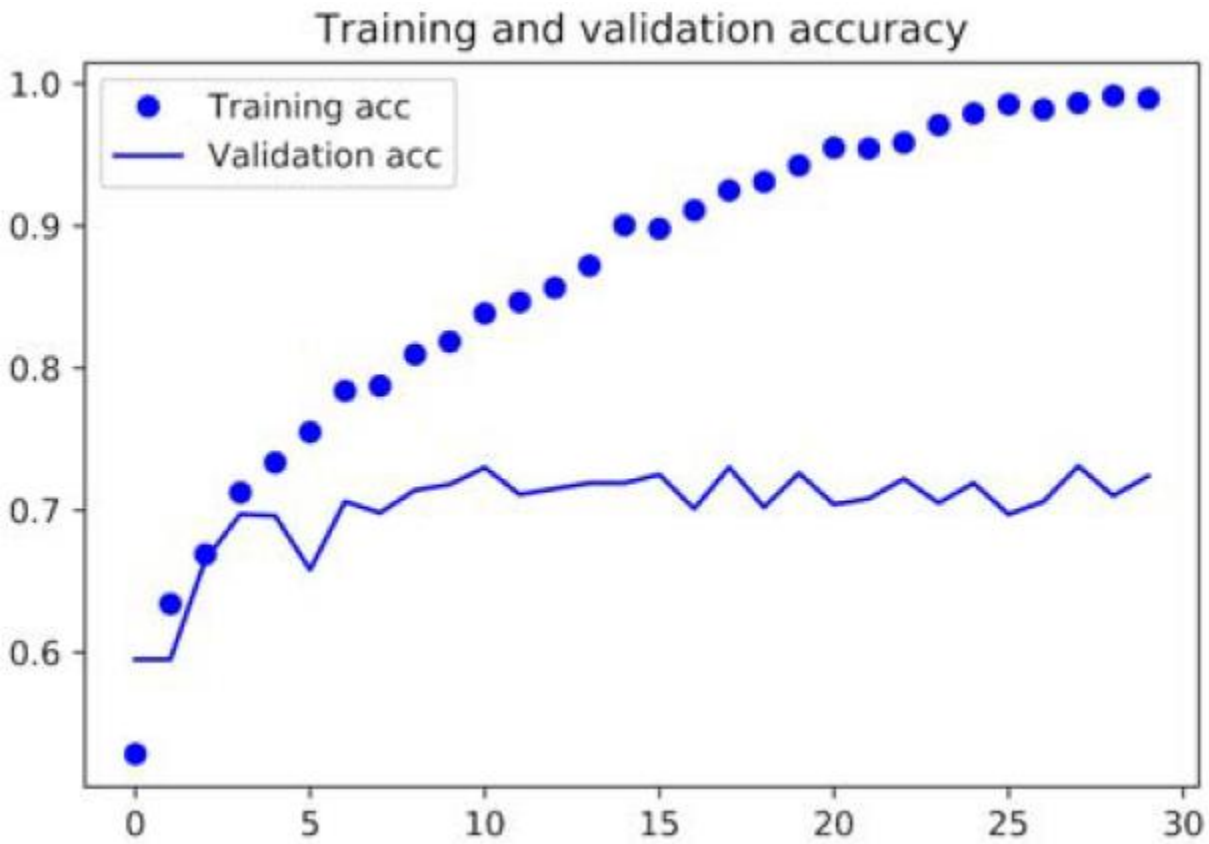
Fitting and Saving the Model

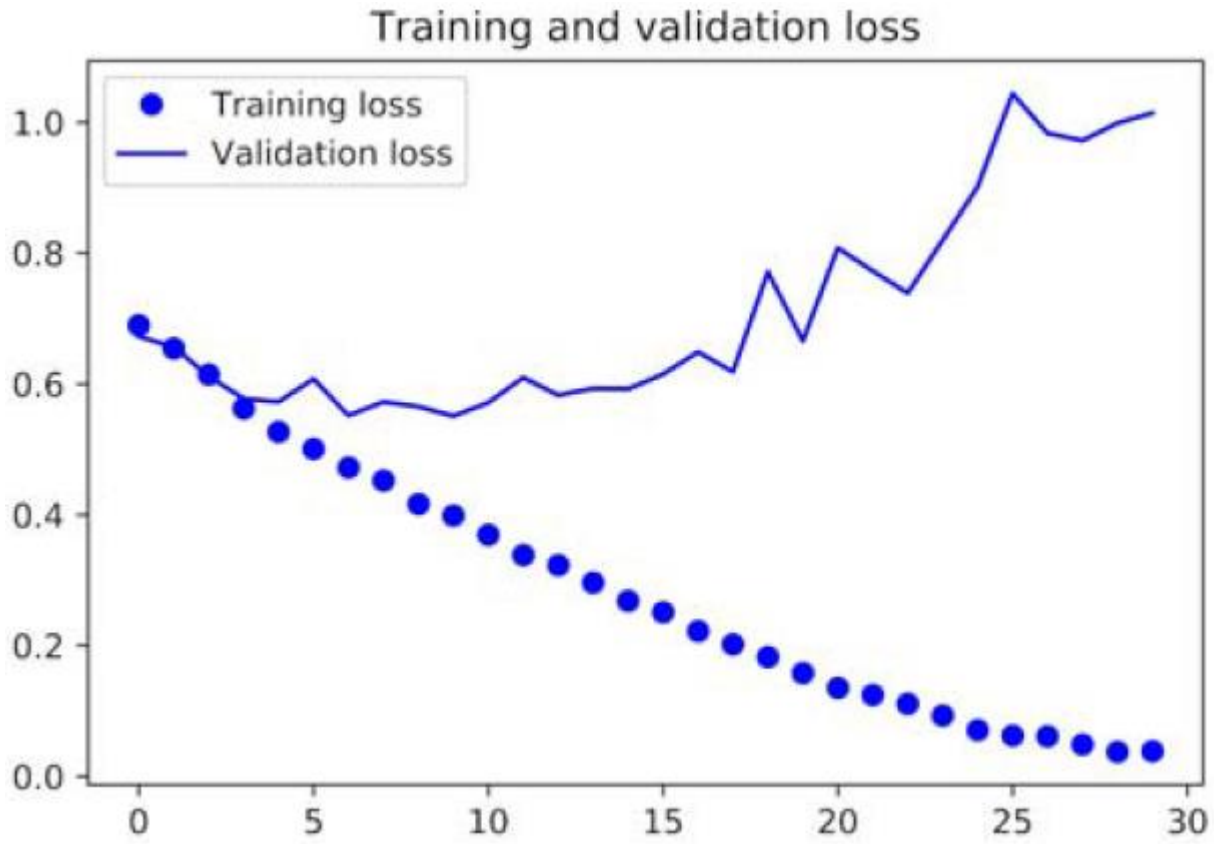
steps_per_epoch = trnX.shape[0] / batch_size

validation_steps = valX.shape[0] / batch_size

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


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

Accuracy and Cross Entropy





Training new model with small data: 70-72% accuracy

Data Augmentation

- Fill modes
  - “constant”: kkkkkkkk|abcd|kkkkkkkk (cval=k)
  - “nearest”: aaaaaaaa|abcd|dddddddd [extend “last” pixel to border]
  - “reflect“: abcddcba|abcd|dcbaabcd
  - “wrap“: abcdabcd|abcd|abcdabcd

```
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')
```

https://en.wikipedia.org/wiki/Shear_mapping

Data Augmentation Options

- `rotation_range` is a value in degrees (0–180), a range within which to randomly rotate pictures.
- `width_shift` and `height_shift` are ranges (as a fraction of total width or height) within which to randomly translate pictures vertically or horizontally.
- `shear_range` is for randomly applying shearing transformations.
- `zoom_range` is for randomly zooming inside pictures.
- `horizontal_flip` is for randomly flipping half the images horizontally—relevant when there are no assumptions of horizontal asymmetry (for example, real-world pictures).
- `fill_mode` is the strategy used for filling in newly created pixels, which can appear after a rotation or a width/height shift.

Displaying 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()
```

Module with image-preprocessing utilities

Chooses one image to augment

Reads the image and resizes it

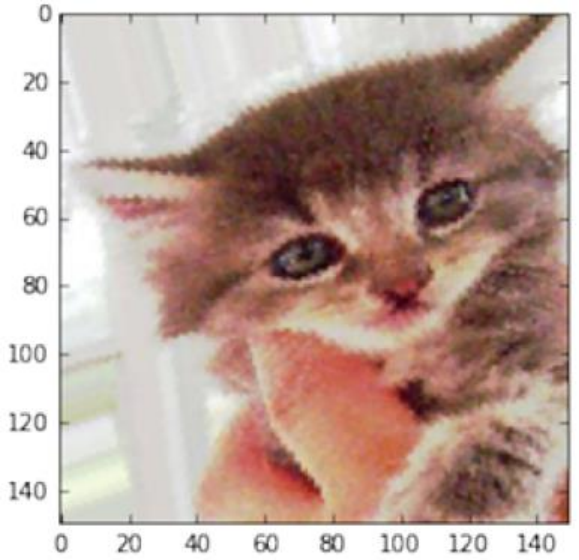
Converts it to a Numpy array with shape (150, 150, 3)

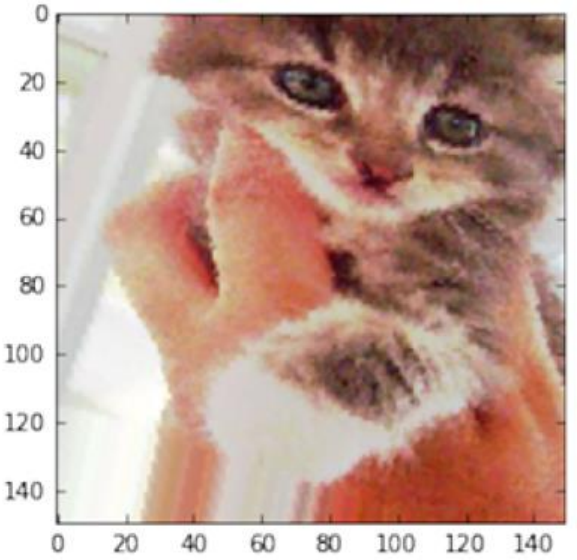
Reshapes it to (1, 150, 150, 3)

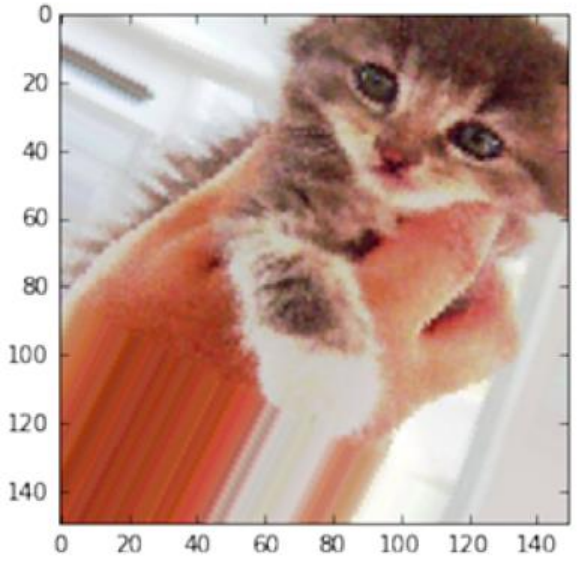
Generates batches of randomly transformed images. Loops indefinitely, so you need to break the loop at some point!

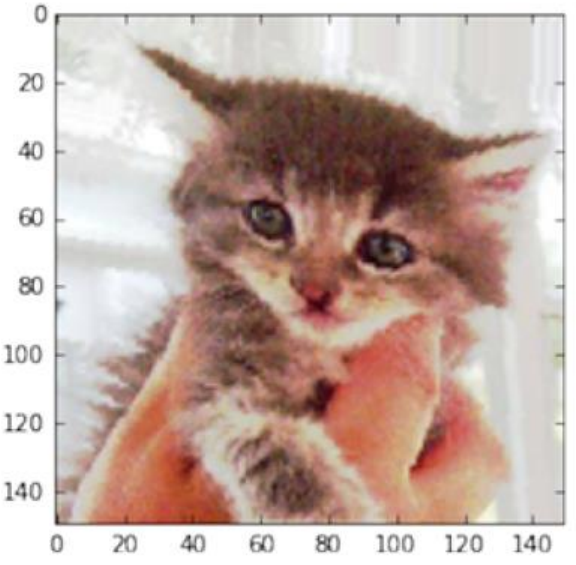
Augmented Images

- cat.100.jpg: 499x403 => 150x150
- Transformations include Horizontal Flip, Rotation, Shear, Translation, and Zoom









Model with Dropout for Augmented Data

```
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'])
```

Training the ConvNet with Augmented Data

```
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)
```

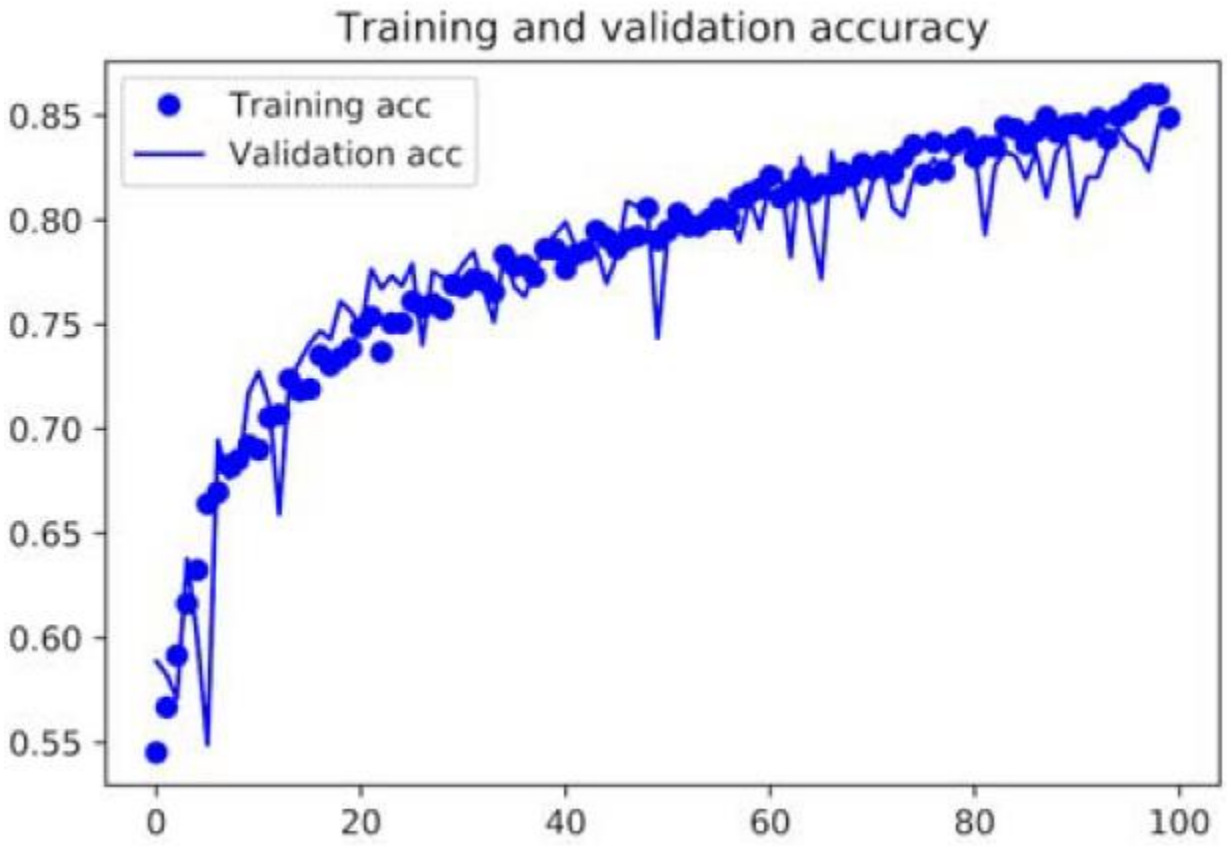
Note that the validation data shouldn't be augmented!

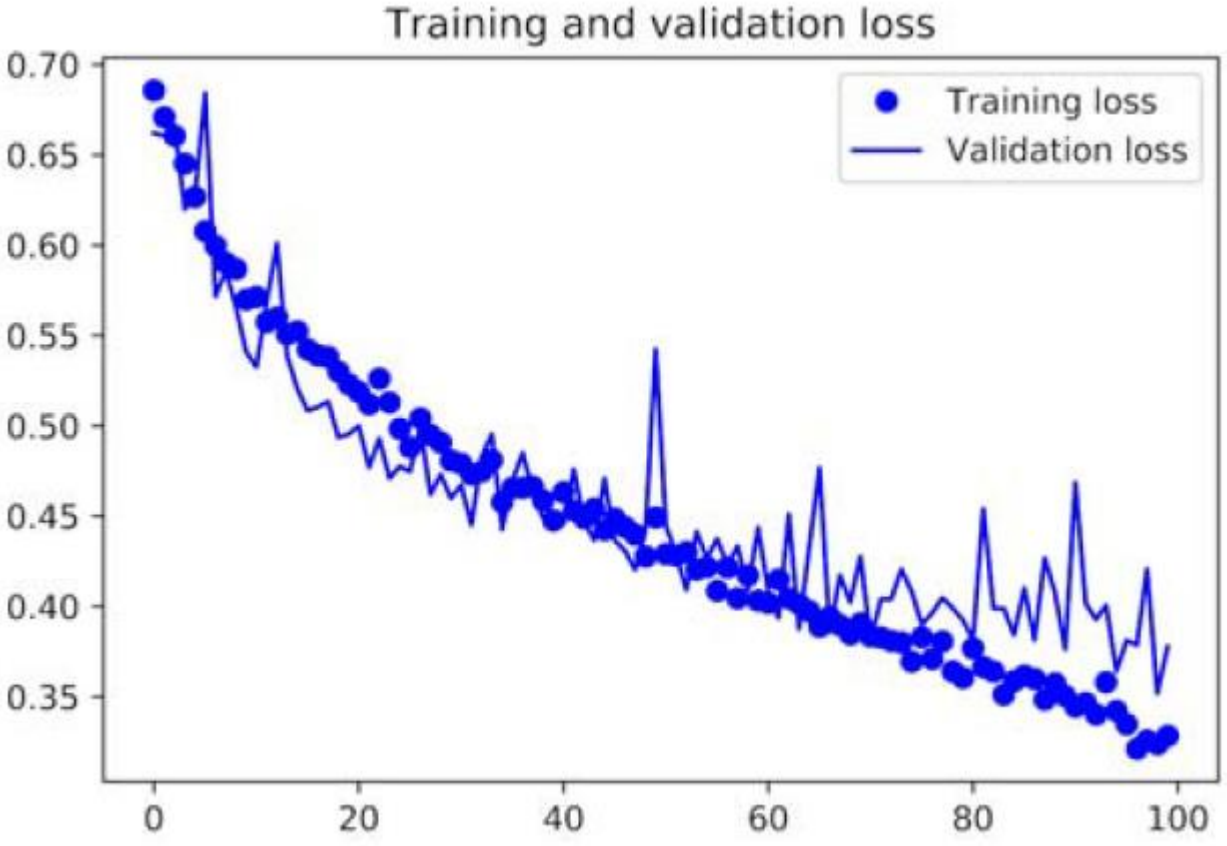
Target directory

Resizes all images to 150 × 150

Because you use binary_crossentropy loss, you need binary labels.

Accuracy and Cross Entropy

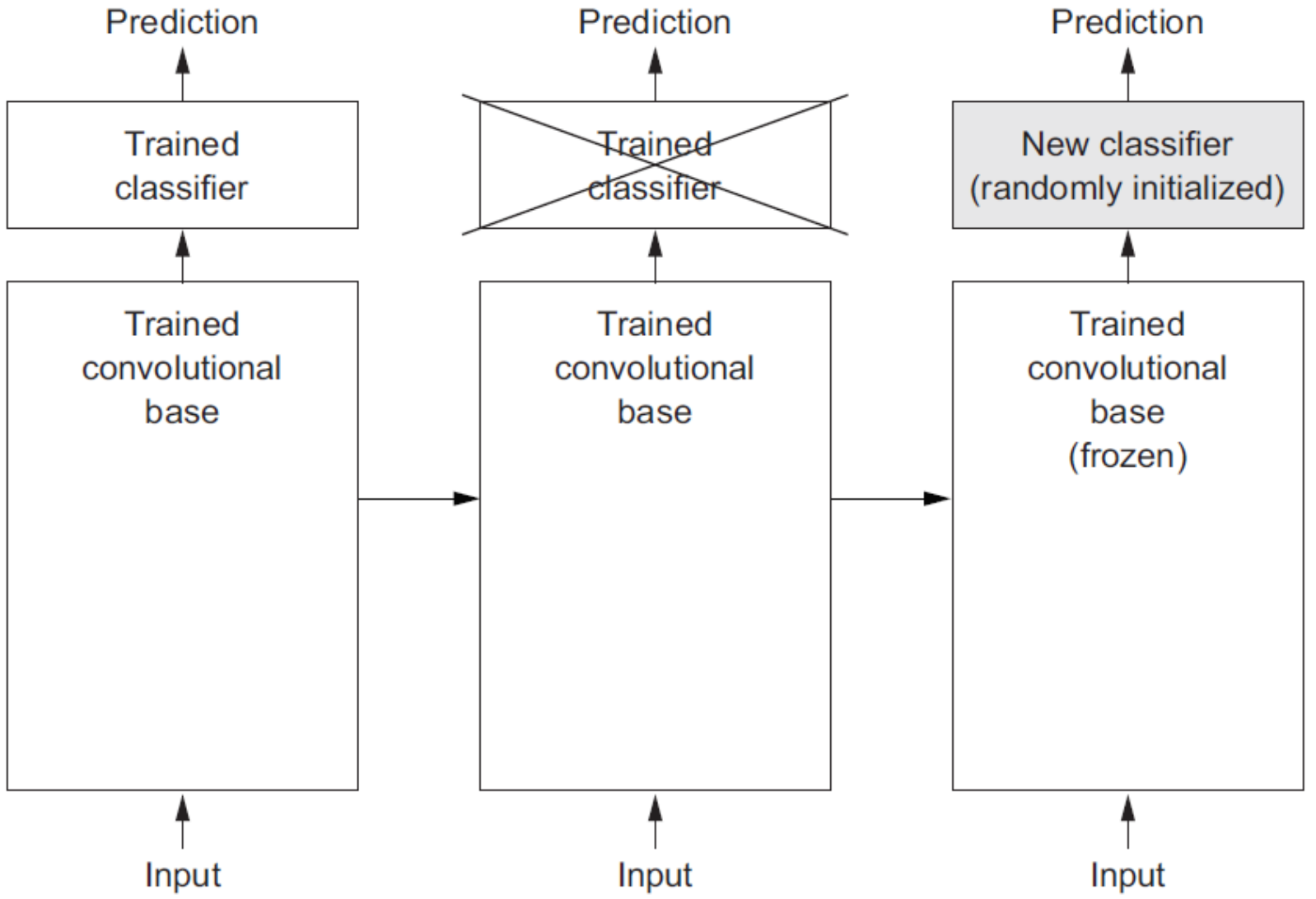




Training new model with small data (and augmentation and dropout): 82% accuracy [71% to 82%: +15% relative improvement]

Swapping Classifiers for the Same Convolutional Base

- Transferring learning from one application to another
- Visual features from one application can help in another application, especially when there is less data for the new application



VGG16

- ConvNet built by the University of Oxford's Visual Geometry Group
- 16 Layers have Parameters (weights and biases)

```
from keras.applications import VGG16

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

- `weights` specifies the weight checkpoint from which to initialize the model.
- `include_top` refers to including (or not) the densely connected classifier on top of the network. By default, this densely connected classifier corresponds to the 1,000 classes from ImageNet. Because you intend to use your own densely connected classifier (with only two classes: `cat` and `dog`), you don't need to include it.
- `input_shape` is the shape of the image tensors that you'll feed to the network. This argument is purely optional: if you don't pass it, the network will be able to process inputs of any size.

VGG16 Summary: Blocks 1 and 2

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

VGG16 Summary: Blocks 3 and 4

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

VGG16 Summary: Block 5

```
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
```

VGG16 Summary: Top [removed for this application]

3 fully connected layers …

flatten (Flatten)	(None, 25088)	0
fc1 (Dense)	(None, 4096)	102,764,544
fc2 (Dense)	(None, 4096)	16,781,312
predictions (Dense)	(None, 1000)	4,097,000

Alternatives for Using the Convolutional Base

- 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 similar to those you saw in part 1 of this book. This solution is 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. But for the same reason, this technique won't allow you to use data augmentation.
- Extending the model you have (`conv_base`) by adding `Dense` layers on top, and running the whole thing end to end on the input data. This will allow you to use data augmentation, because every input image goes through the convolutional base every time it's seen by the model. But for the same reason, this technique is far more expensive than the first.

Extracting Features: Define the Image Generator

```
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
```

Extracting Features: for Trn, Val, and Tst

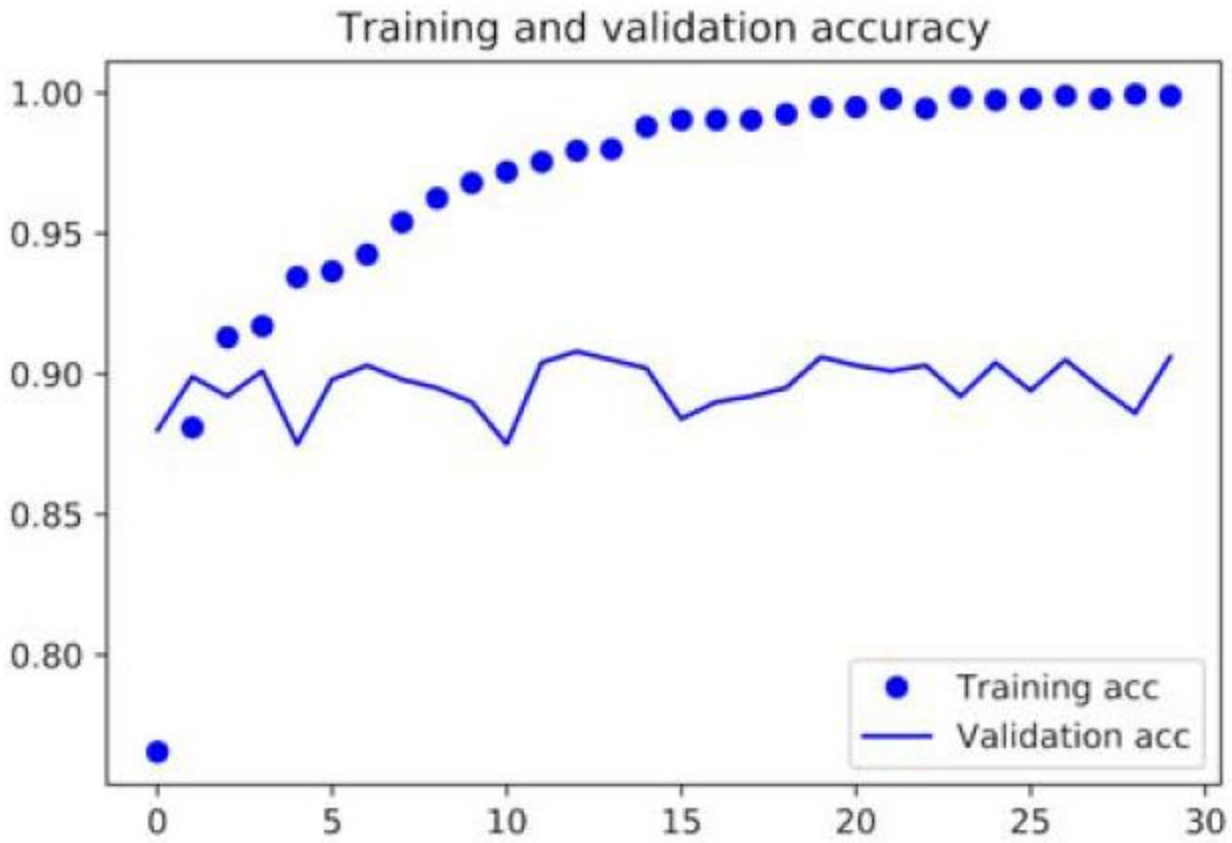
```
def extract_features(directory, sample_count):
    features = np.zeros(shape=(sample_count, 4, 4, 512))
    labels = np.zeros(shape=(sample_count))
    generator = datagen.flow_from_directory(
        directory,
        target_size=(150, 150),
        batch_size=batch_size,
        class_mode='binary')
    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
    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)
```

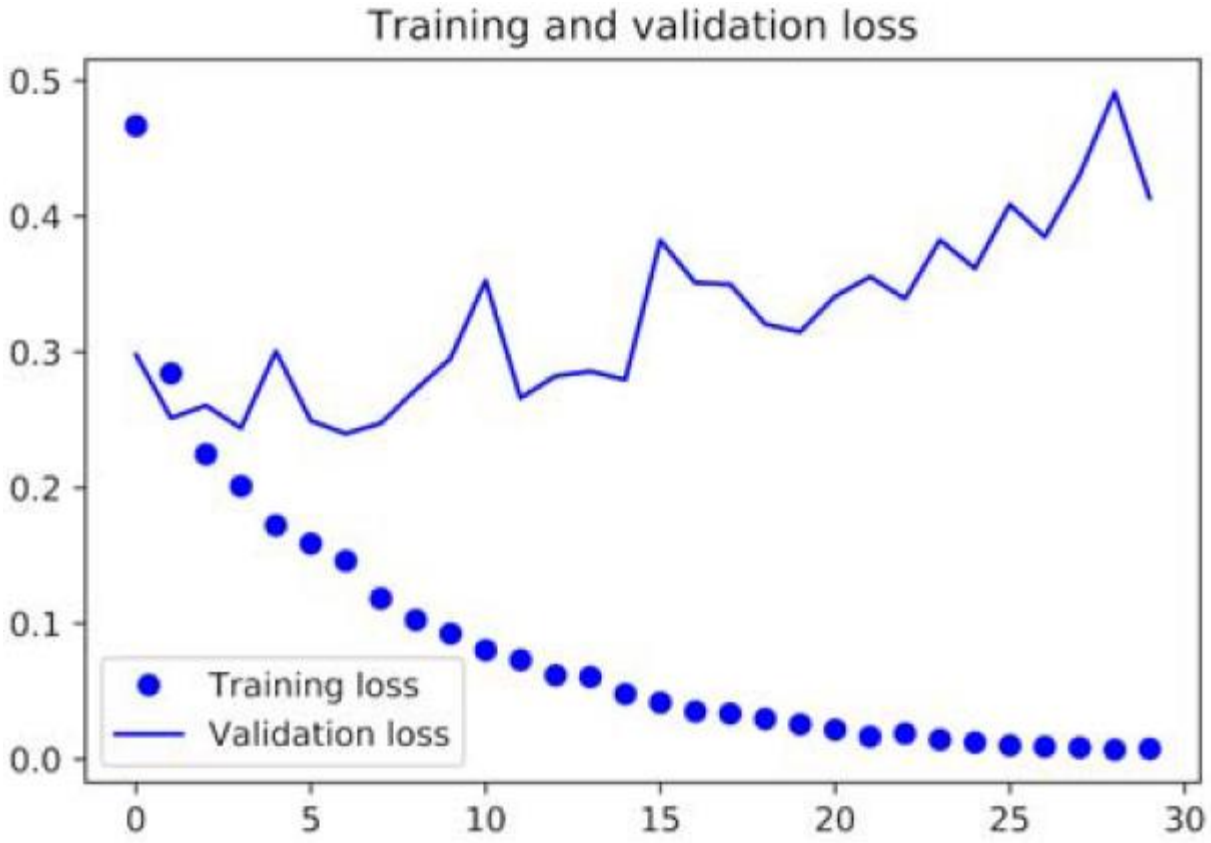
Note that because generators yield data indefinitely in a loop, you must break after every image has been seen once.

Training a Model with the Convolutional Features

```
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))

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))
```

Accuracy and Cross Entropy





Training new model with VGG16 features: 90% accuracy

Feature Extraction with Data Augmentation: the Warning

NOTE This technique is so expensive that you should only attempt it if you have access to a GPU—it's absolutely intractable on CPU. If you can't run your code on GPU, then the previous technique is the way to go.

Using the Convolutional Base as a Layer

```
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
```

Freezing the Convolutional Base

```
(13 + 2) * 2 = 30
( 0 + 2) * 2 =  4
2: Weight Matrix plus Bias Vector
```

```
>>> 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
```

Configuring the Data Generators

```
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')
```

Note that the validation data shouldn't be augmented!

Target directory

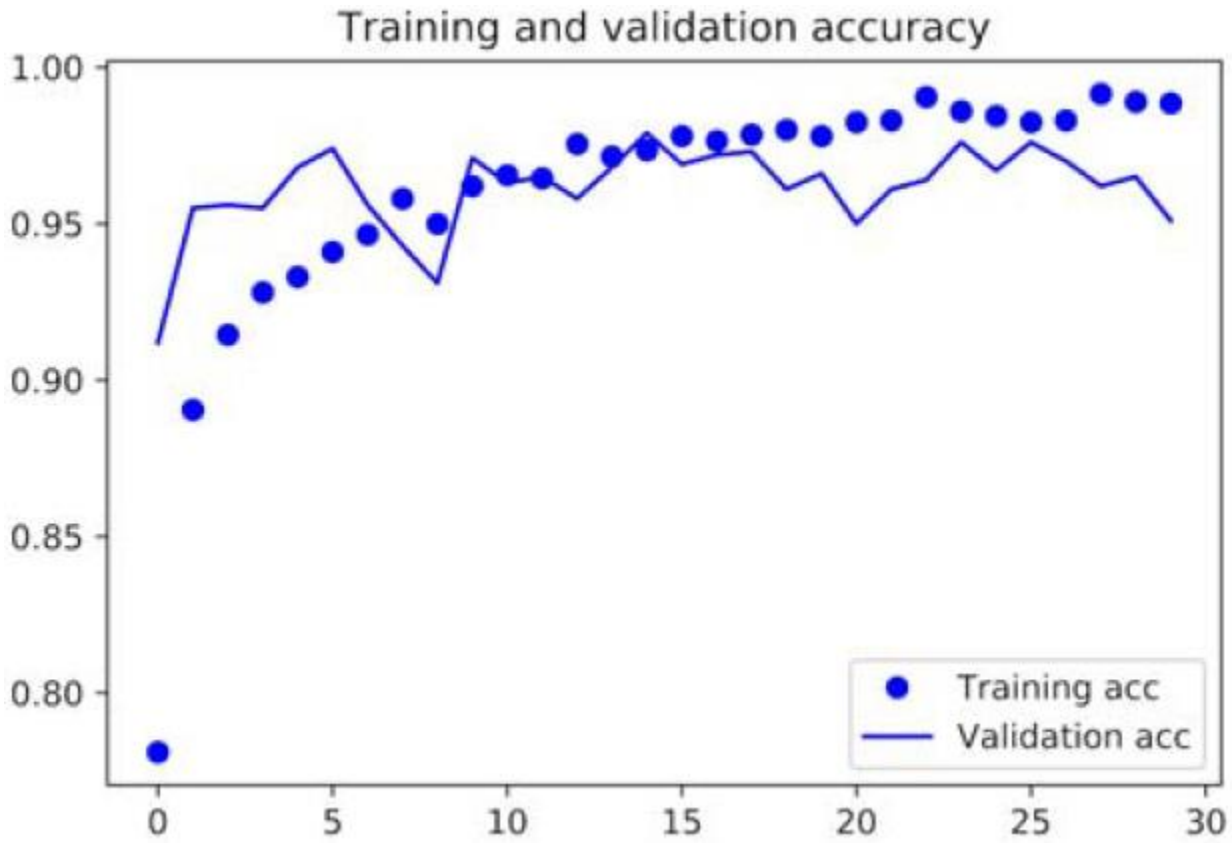
Resizes all images to 150 × 150

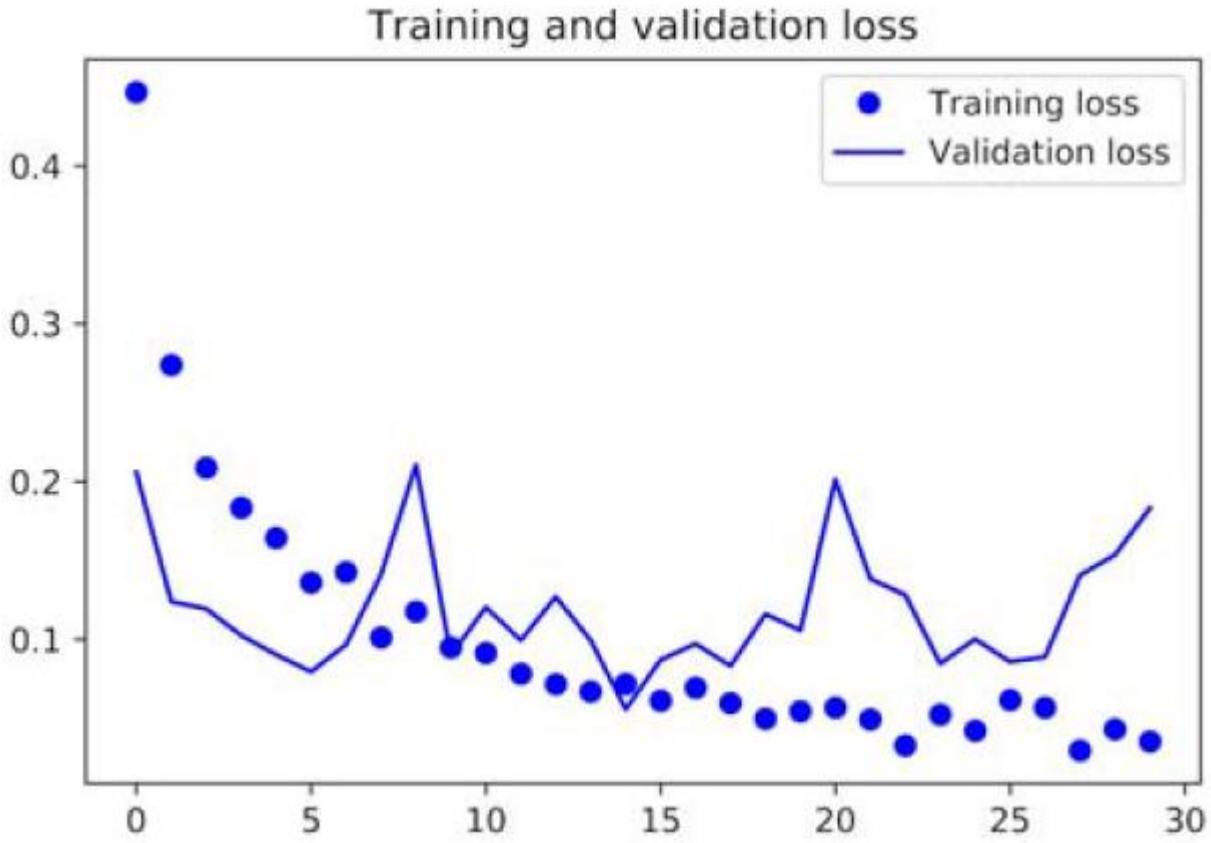
Because you use binary_crossentropy loss, you need binary labels.

Running the Generators with the Pretrained Base

```
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)
```

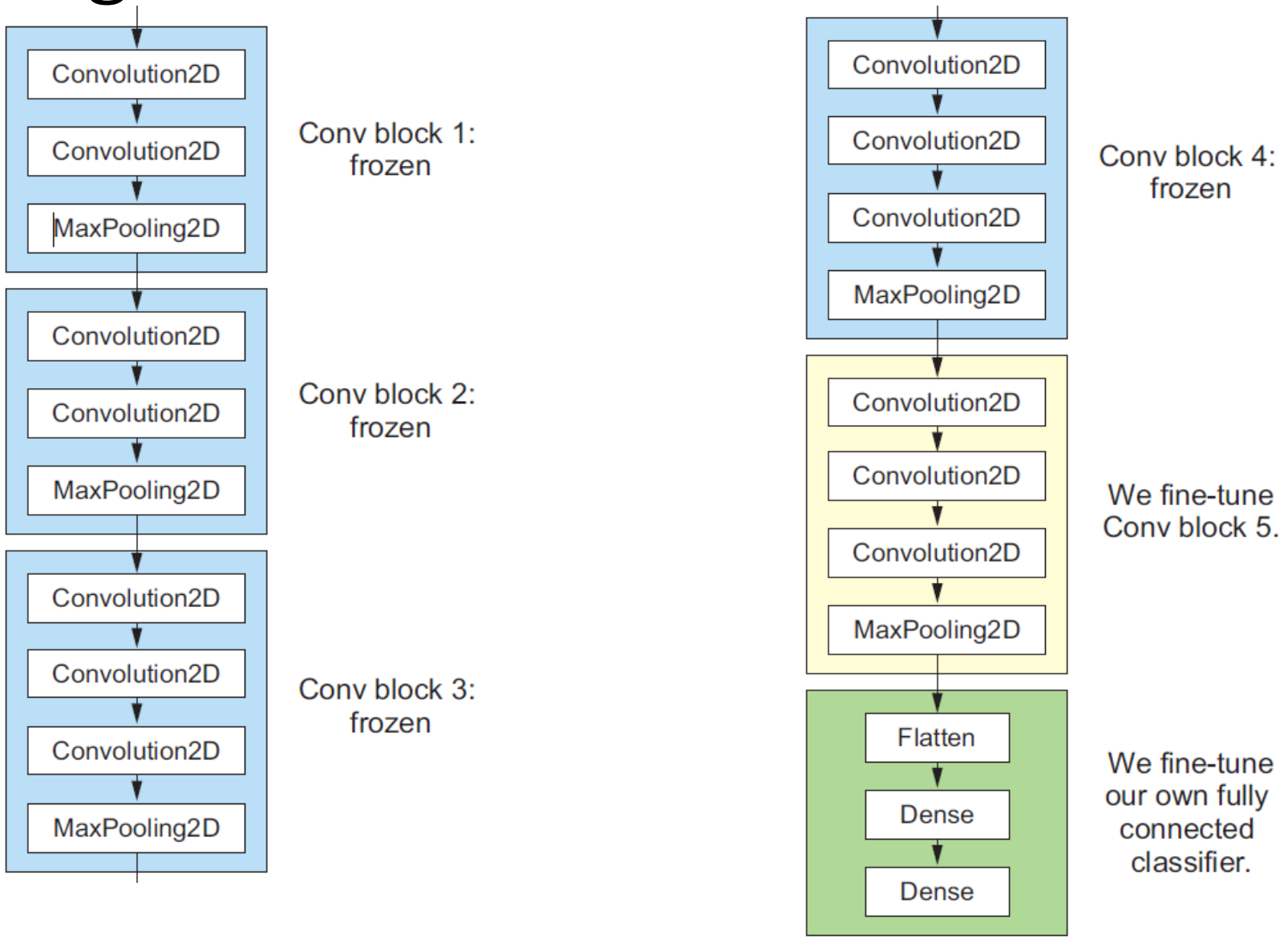
Accuracy and Cross Entropy





Training new model with VGG16 features, with data augmentation: 96% accuracy

Fine-Tuning Last Block of the VGG16 Base



Fine-Tuning a Pretrained Network

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.

```
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
```

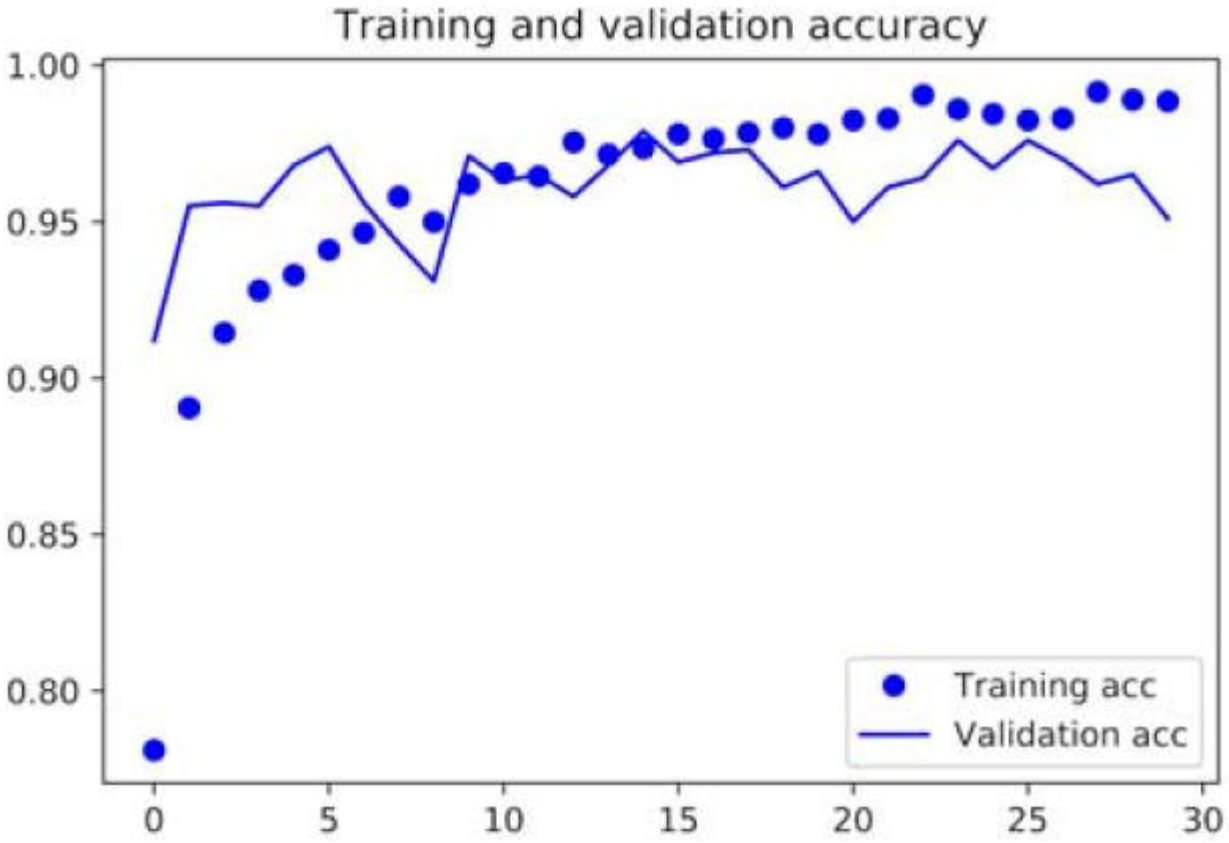
Fine-Tuning the Last Block of the VGG16 Base

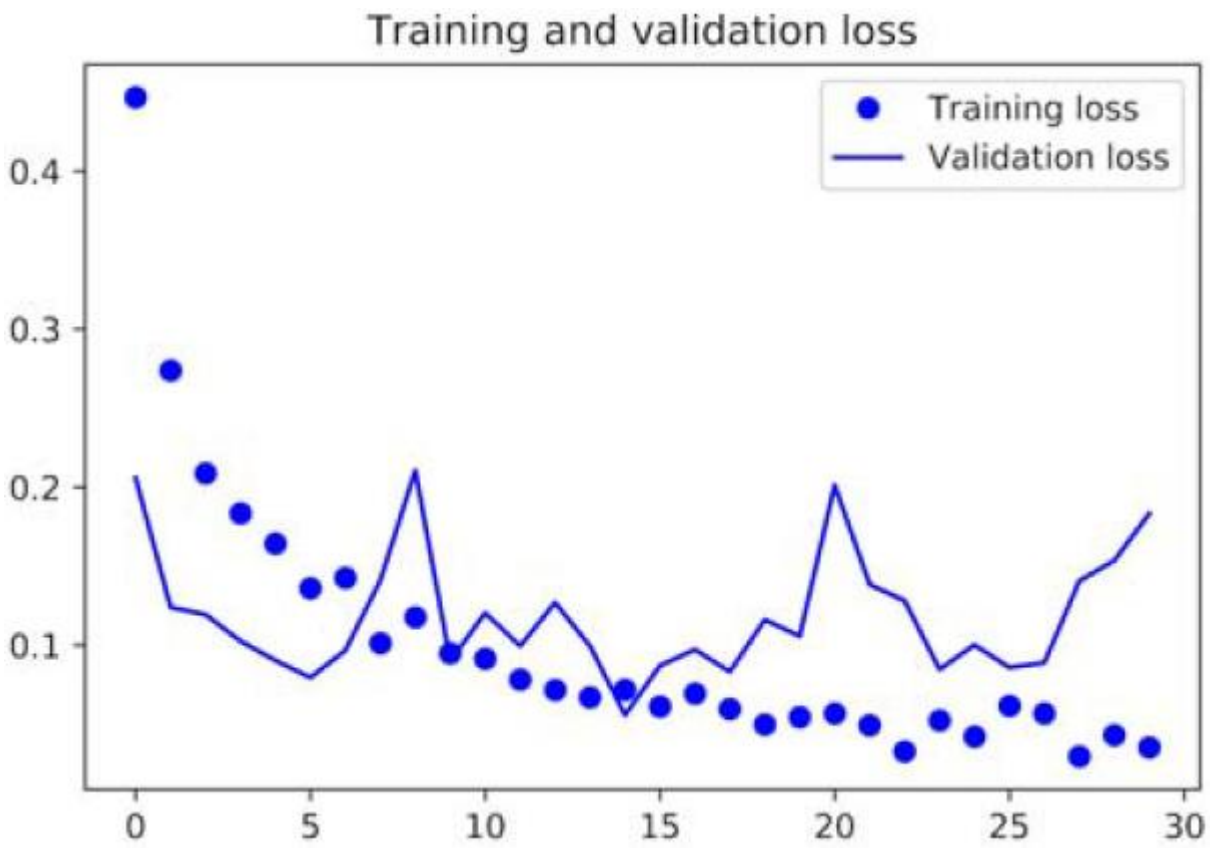
Half the learning rate

More epochs

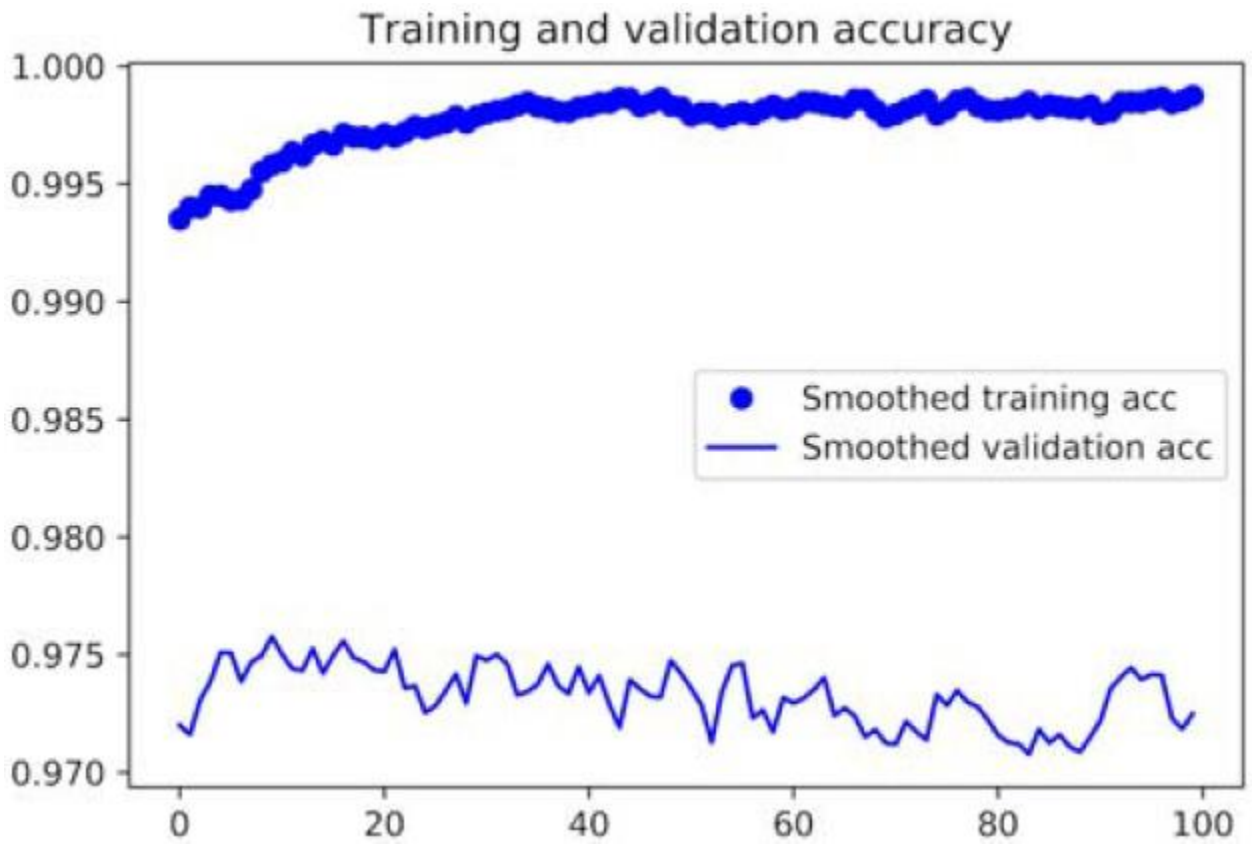
```
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-5),
              metrics=['acc'])

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

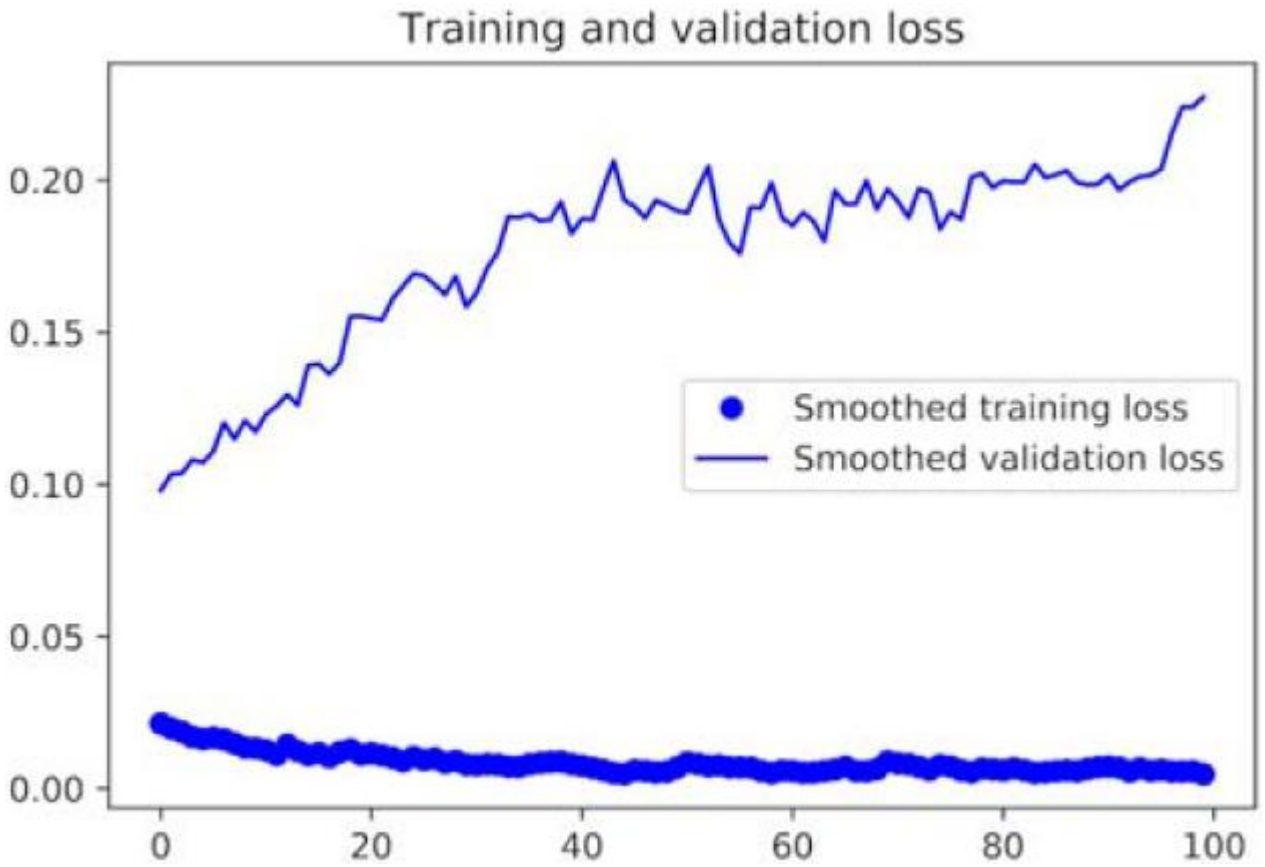
Accuracy and Cross Entropy





Smoothed Accuracy and Cross Entropy





Fine-tuning last block of VGG16 base, with data augmentation: 97% accuracy … finally ready for testing!

Wrapping Up

- Convnets are the best type of machine-learning models for computer-vision tasks. It's 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 you're working with image data.
- It's easy to reuse an existing convnet on a new dataset via feature extraction. This is a valuable technique for working with small image datasets.
- As a complement to feature extraction, you can 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.

Visualizing What ConvNets Learn

- *Visualizing intermediate convnet outputs (intermediate activations)*—Useful for understanding how successive convnet layers transform their input, and for getting a first idea of the meaning of individual convnet filters.
- *Visualizing convnets filters*—Useful for understanding precisely what visual pattern or concept each filter in a convnet is receptive to.
- *Visualizing heatmaps of class activation in an image*—Useful for understanding which parts of an image were identified as belonging to a given class, thus allowing you to localize objects in images.

Reloading Earlier Model

```
>>> 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

Processing a Single Image

```
img_path = '/Users/fchollet/Downloads/cats_and_dogs_small/test/cats/cat.1700.jpg'

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.

print(img_tensor.shape)
```

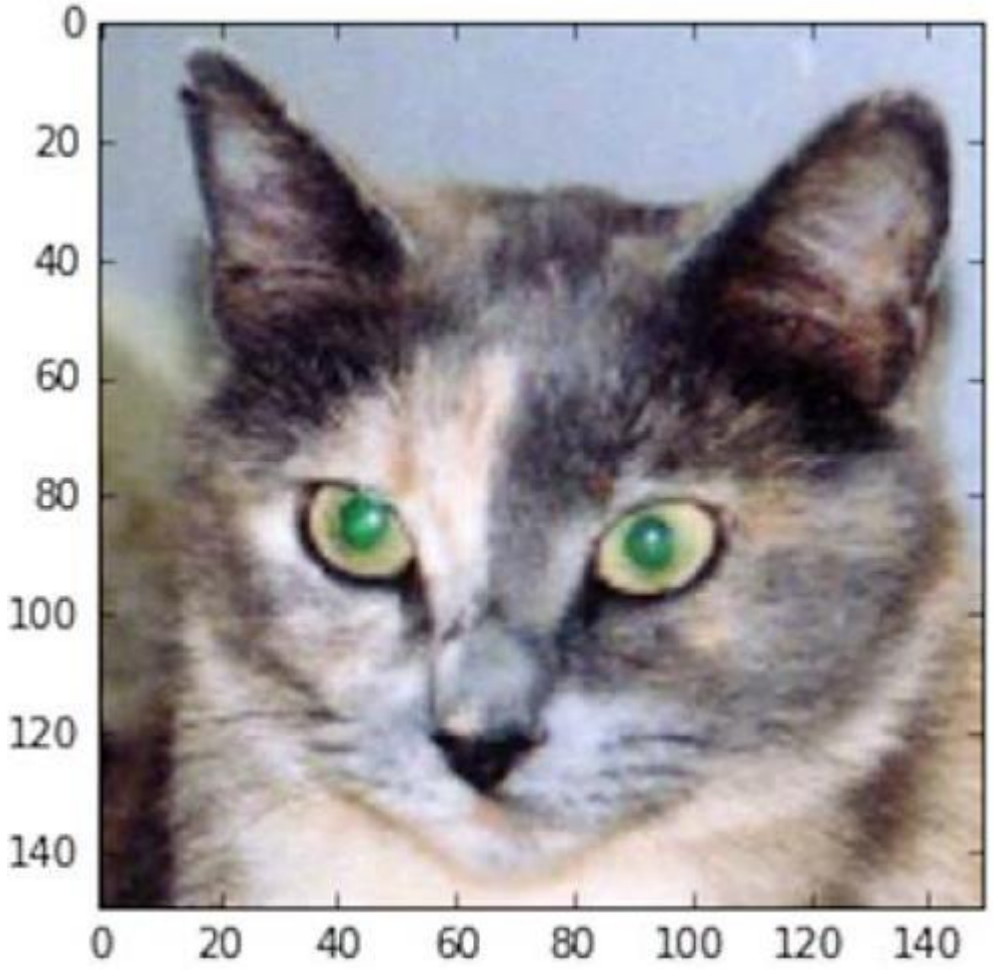
Preprocesses the image into a 4D tensor

Remember that the model was trained on inputs that were preprocessed this way.

Its shape is (1, 150, 150, 3)

Displaying the Image

```
import matplotlib.pyplot as plt

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



Creating an Activations Model for the Convolutional Base

```
from keras import models

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

Extracts the outputs of the top eight layers

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

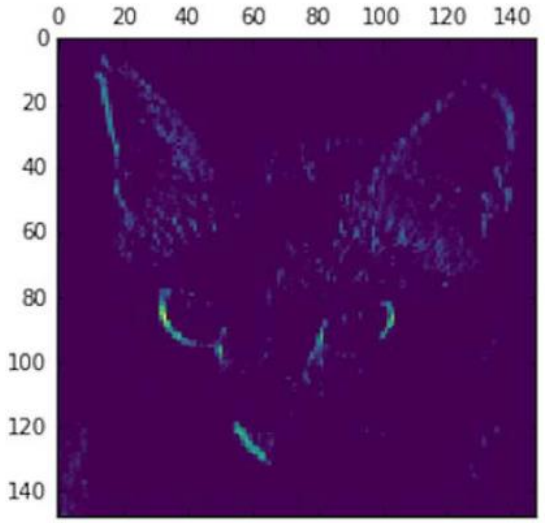
Visualizing the Activations for the "4th" Convolution Filter of the First Layer

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

Returns a list of five Numpy arrays: one array per layer activation

```
>>> first_layer_activation = activations[0]
>>> print(first_layer_activation.shape)
(1, 148, 148, 32)
```

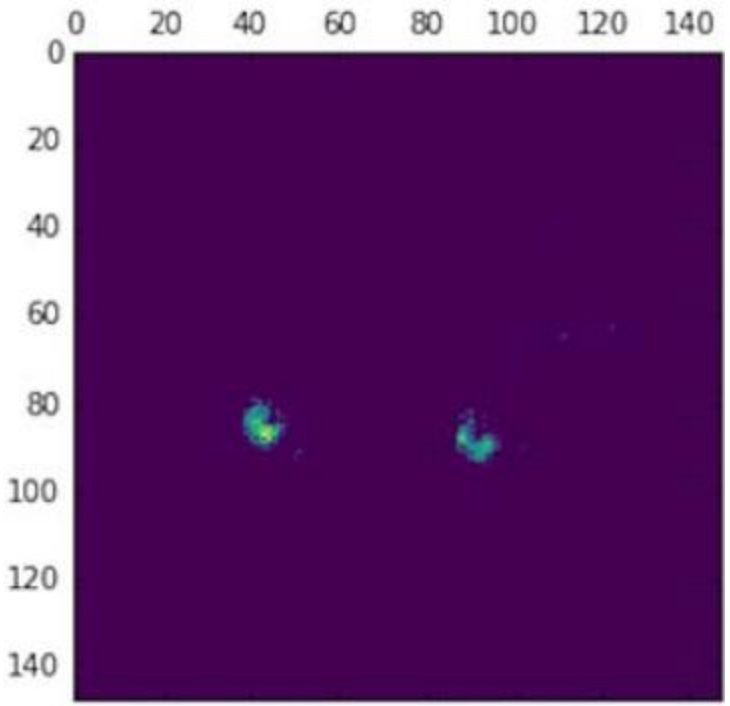
```
import matplotlib.pyplot as plt

plt.matshow(first_layer_activation[0, :, :, 4], cmap='viridis')
```



Looks like a diagonal edge detector
Check out the nose and ear

Visualizing the Activations for the “7th“ Convolution Filter of the First Layer

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



Looks like a color detector
Check out the eyes

Visualizing All of the Convolution Activations

```
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')
```

Names of the layers, so you can have them as part of your plot

Displays the feature maps

Number of features in the feature map

The feature map has shape (l, size, size, n_features).

Tiles the activation channels in this matrix

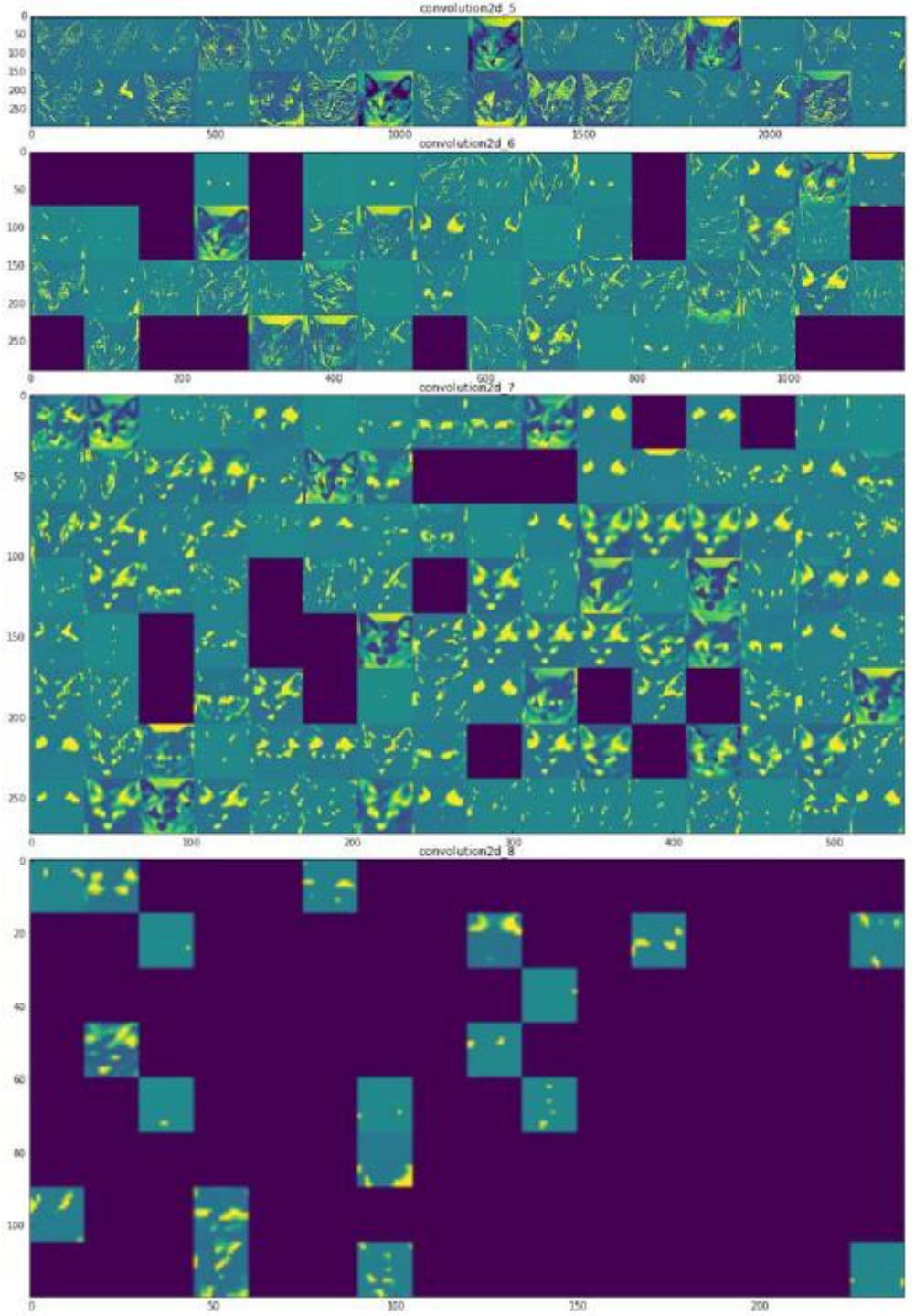
Tiles each filter into a big horizontal grid

Post-processes the feature to make it visually palatable

Displays the grid

Activations

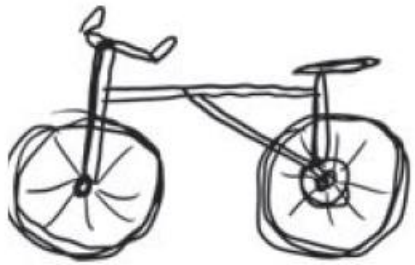
- Counts: [32, 64, 128, 128]
- 1st layer acts as a collection of various edge detectors
- as we go from the 1st convolution layer to the 4th convolution layer
  - Features become more abstract (higher-level concepts) and less visually interpretable
  - Activations become more sparse



Abstract Representation

- "... a human can remember which abstract objects were present in [a scene] (bicycle, tree) but can't remember the specific appearance of these objects. In fact, if you tried to draw a generic bicycle from memory, chances are you couldn't get it even remotely right, even though you've seen thousands of bicycles in your lifetime."

Attempts to draw from memory



Actual picture

Visualizing ConvNet Filters

- We want to visualize an image that maximizes a convolution filter's mean output
- We'll be using gradient ascent on the convolution output
- [better to call it activation rather than loss]

```
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])
```

We're using VGG16

Fetching Gradients

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

The call to gradients returns a list of tensors (of size 1 in this case). Hence, you keep only the first element—which is a tensor.

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

Add 1e–5 before dividing to avoid accidentally dividing by 0.

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

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

Optimizing the Input

Starts from a gray image with some noise

```
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
```

Magnitude of each gradient update

Runs gradient ascent for 40 steps

Computes the loss value and gradient value

Adjusts the input image in the direction that maximizes the loss

Gradient ascent to maximize activations:
simply adding the gradients to the image

Converting the Input Tensor to an 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
```

Normalizes the tensor: centers on 0, ensures that std is 0.1

Clips to [0, 1]

Converts to an RGB array

Putting the Pieces Together

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

Computes the gradient of the input picture with regard to this loss

Normalization trick: normalizes the gradient

Returns the loss and grads given the input picture

Runs gradient ascent for 40 steps

Starts from a gray image with some noise

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

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

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

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

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

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

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

Visualizing "Optimal" Input for All the Filters

```
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
```

Empty (black) image to store results

Iterates over the rows of the results grid

Iterates over the columns of the results grid

Generates the pattern for filter i + (j * 8) in layer_name

Puts the result in the square (i, j) of the results grid

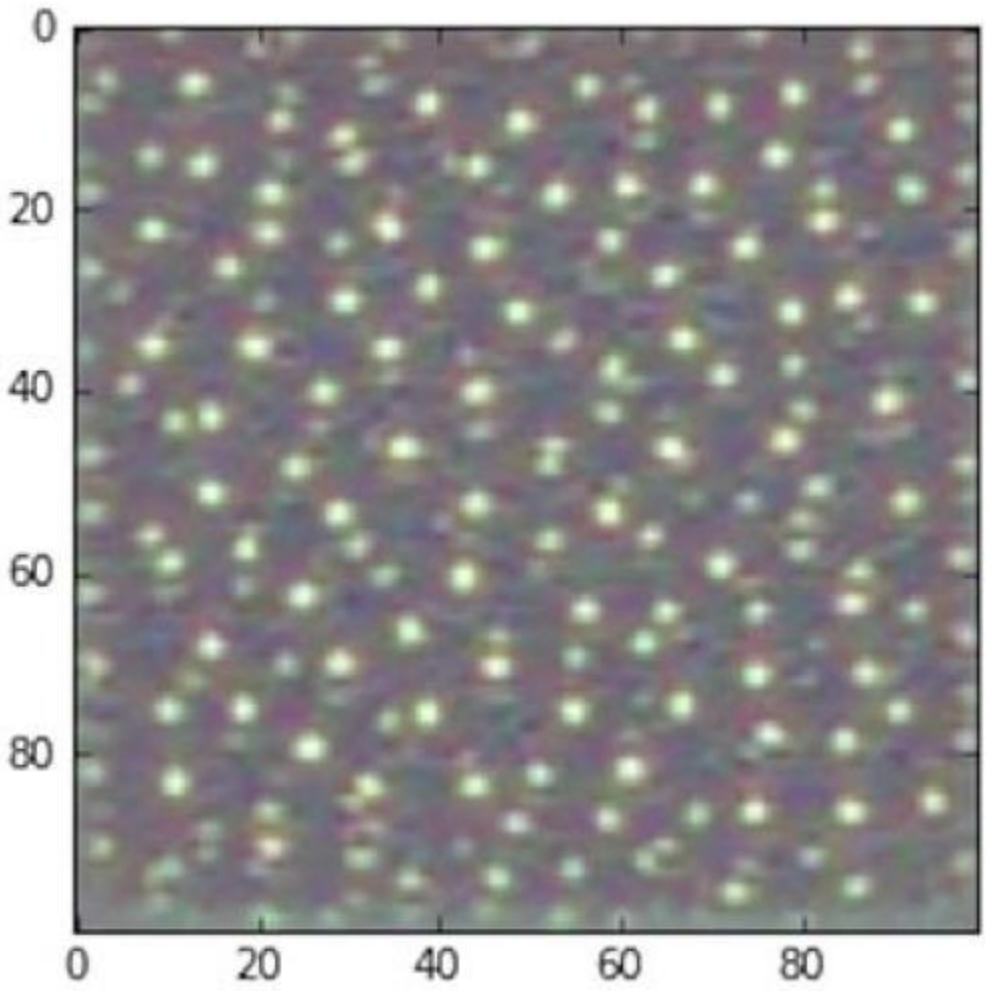
```
plt.figure(figsize=(20, 20))
plt.imshow(results)
```

Displays the results grid

Presenting the Polka Dot Filter ☺

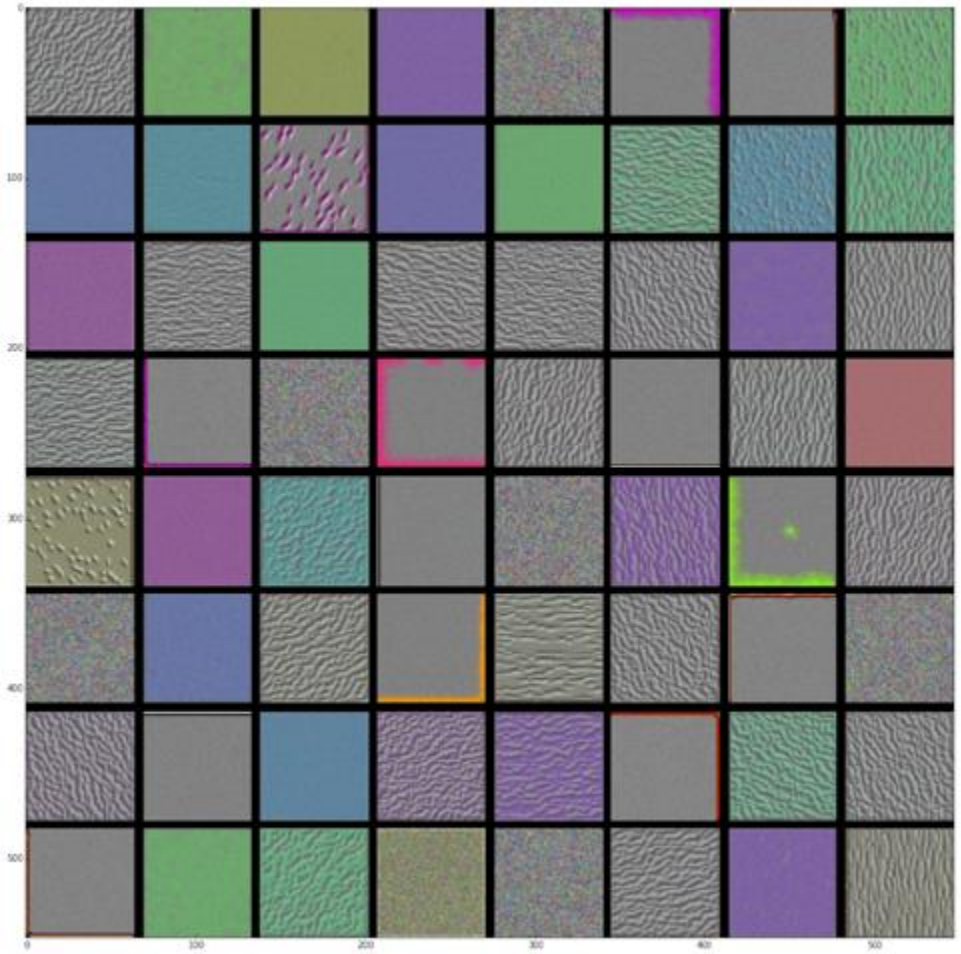
from block3_conv1

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



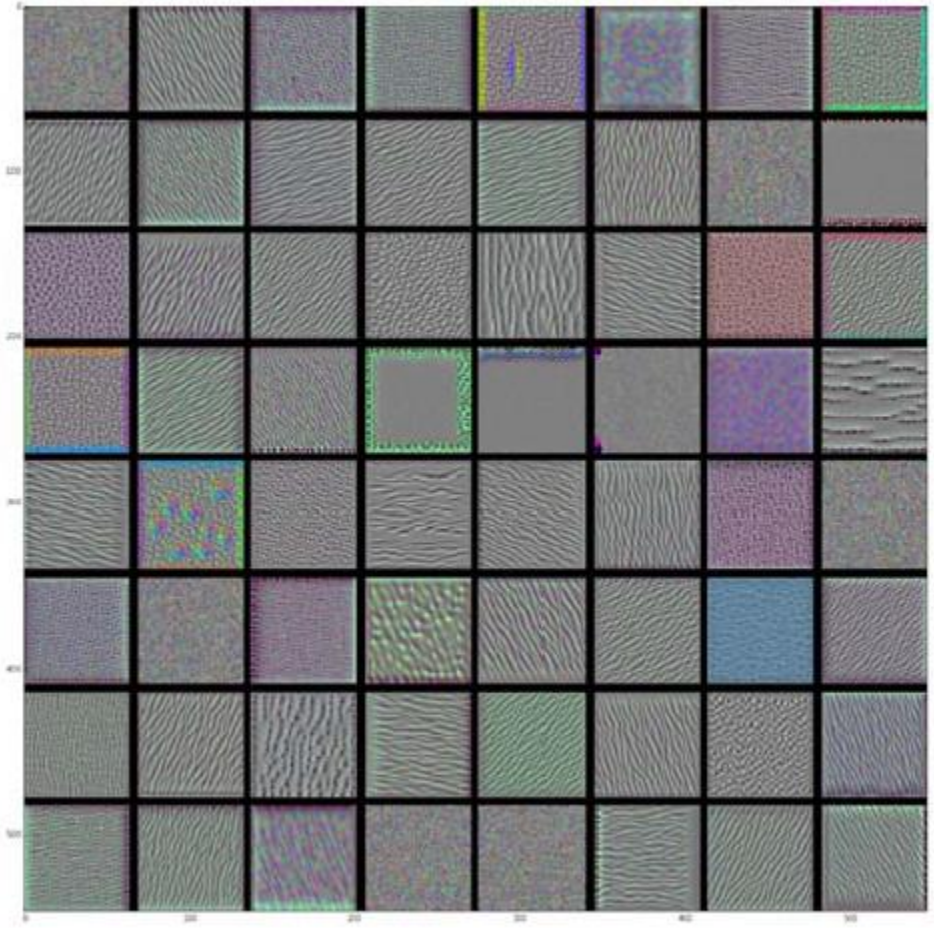
“Optimal” Images for block1_conv1

Simple directional edges and colors



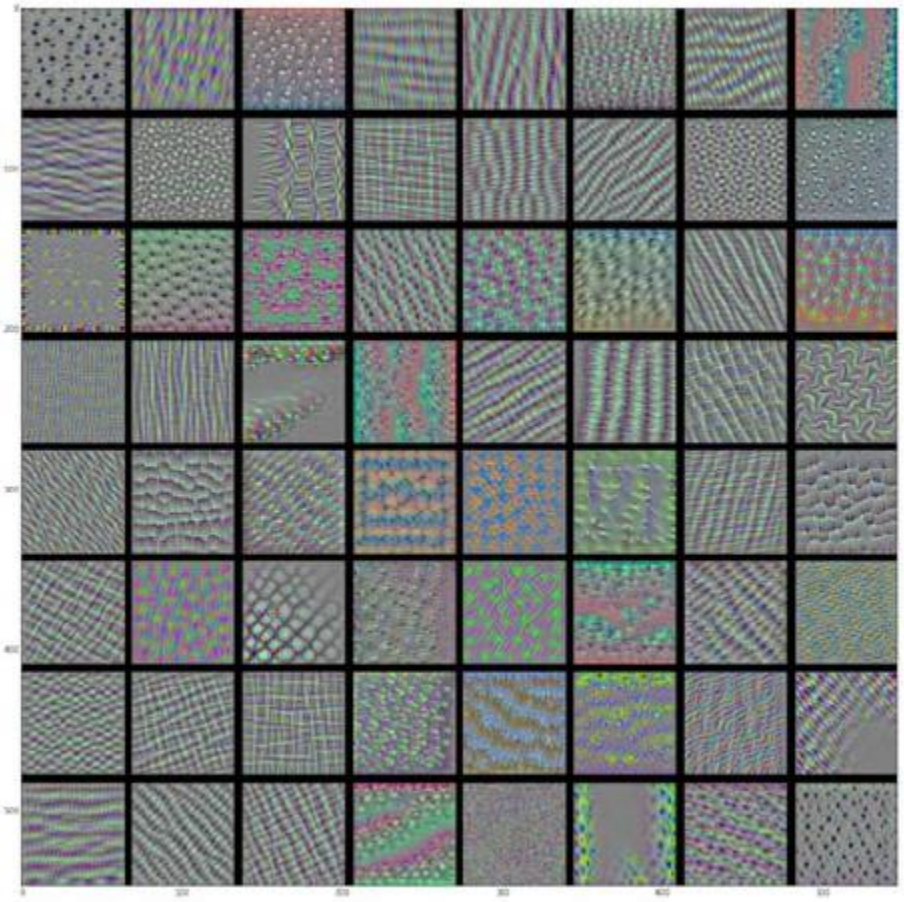
“Optimal” Images for block2_conv1

Simple textures from combinations of edges and colors



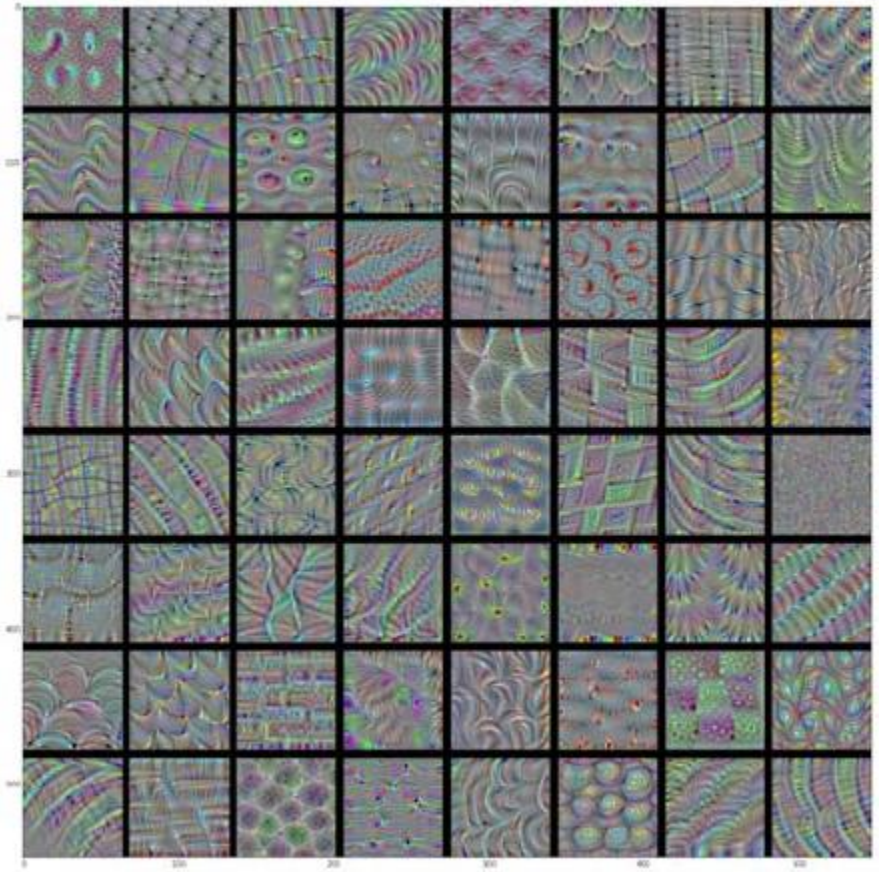
“Optimal” Images for block3_conv1

Includes the polka dot filter



“Optimal” Images for block4_conv1

Higher layer filters are activated by more complex textures, resembling textures found in natural images



Loading the VGG16 Convolutional Base

```
from keras.applications.vgg16 import VGG16
model = VGG16(weights='imagenet')
```

Note that you include the densely connected classifier on top; in all previous cases, you discarded it.

We've loaded the top as well

https://github.com/rstudio/keras/blob/master/vignettes/examples/creative_commons_elephant.jpg

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)
```

Local path to the target image

Python Imaging Library (PIL) image of size 224 × 224

float32 Numpy array of shape (224, 224, 3)

Adds a dimension to transform the array into a batch of size (1, 224, 224, 3)

Preprocesses the batch (this does channel-wise color normalization)

Using VGG16 to Make Predictions

```
>>> 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)]
```

The top three classes predicted for this image are as follows:

- African elephant (with 92.5% probability)
- Tusker (with 7% probability)
- Indian elephant (with 0.4% probability)

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

Gradients of the Elephant Softmax Activation with respect to the block5_conv3 Activation

"African elephant" entry in the prediction vector

```
african_e66lephant_output = model.output[:, 386]

last_conv_layer = model.get_layer('block5_conv3')
```

Output feature map of the block5_conv3 layer, the last convolutional layer in VGG16

Gradient of the "African elephant" class with regard to the output feature map of block5_conv3

```
grads = K.gradients(african_elephant_output, last_conv_layer.output)[0]

pooled_grads = K.mean(grads, axis=(0, 1, 2))
```

Vector of shape (512,), where each entry is the mean intensity of the gradient over a specific feature-map channel

Generating the Heatmap for block5_conv3 Activations

```
iterate = K.function([model.input],
                     [pooled_grads, last_conv_layer.output[0]])

pooled_grads_value, conv_layer_output_value = iterate([x])

for i in range(512):
    conv_layer_output_value[:, :, i] *= pooled_grads_value[i]

heatmap = np.mean(conv_layer_output_value, axis=-1)
```

Values of these two quantities, as Numpy arrays, given the sample image of two elephants

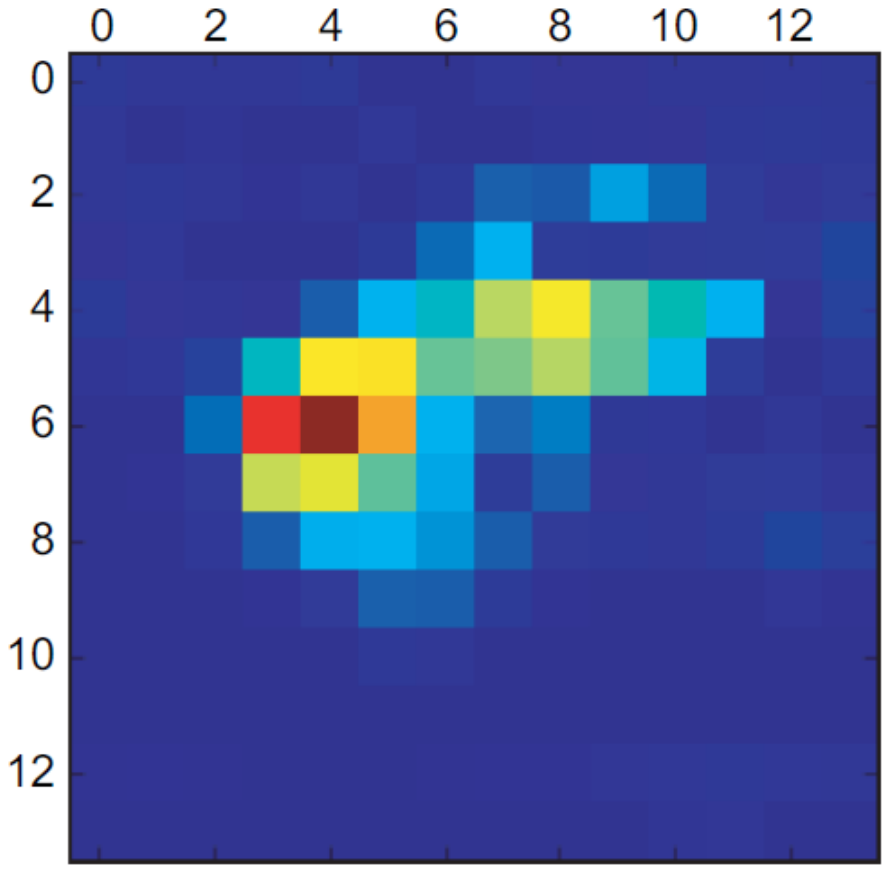
Lets you access the values of the quantities you just defined: pooled_grads and the output feature map of block5_conv3, given a sample image

The channel-wise mean of the resulting feature map is the heatmap of the class activation.

Multiplies each channel in the feature-map array by "how important this channel is" with regard to the "elephant" class

Normalizing and Displaying the Heatmap

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



Superimposing the Heatmap onto the Original Picture

```
import cv2
img = cv2.imread(img_path)
heatmap = cv2.resize(heatmap, (img.shape[1], img.shape[0]))
heatmap = np.uint8(255 * heatmap)
heatmap = cv2.applyColorMap(heatmap, cv2.COLORMAP_JET)
superimposed_img = heatmap * 0.4 + img
cv2.imwrite('/Users/fchollet/Downloads/elephant_cam.jpg', superimposed_img)
```

Uses cv2 to load the original image

Resizes the heatmap to be the same size as the original image

Converts the heatmap to RGB

0.4 here is a heatmap intensity factor.

Applies the heatmap to the original image

Saves the image to disk

Class Activation Heatmap Superimposed

This 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?

Chapter Summary

- 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.

Runtime Complexity

- When describing the runtime complexity of a computation, we're allowed to ignore constants; so if a computation involves 2*k+1 operations, we would say the runtime complexity is O(k)
  - Big 'O' ["oh"]: an upper-bound on the number of operations
  - Big 'Ω' ["omega"]: a lower-bound on the number of operations
  - Big 'Θ' ["theta"]: both a lower-bound and an upper-bound on the number of operations

Runtime Complexity for a Dense Layer

- Reminder: a Dense layer neuron has a weight for every input feature from the previous layer, so Dense layer neurons are said to generate "global" features
- Runtime (inference) complexity for a dense layer is the same as runtime complexity for matrix multilplication:

  num_input_observations * num_output_features * num_input_features
- So for MNIST training images with a dense layer of 512 neurons, the runtime complexity is 60,000 * 512 * 784 = 24,084,480,000

  we're essentially ignoring the addition operations in our count, as the additional time for addition is somewhat negligible

Runtime Complexity for a Convolutional Layer

- Reminder: a Conv2D layer neuron has weights for only a small neighborhood of input pixels, so Conv2D layer neurons are said to generate "local" features
- Runtime complexity for a Conv2D layer involves terms for the number of neighborhoods evaluated as well as the size of the neighborhood:

Same filter can detect pattern in a different image location

$$\text{ImageCount} * \text{FilterCount} * \left(\left(\frac{InputImageHeight - FilterHeight + 2 * Padding}{StrideHeight} + 1\right) * \left(\frac{InputImageWidth - FilterWidth + 2 * Padding}{StrideWidth} + 1\right)\right) * (FilterHeight * FilterWidth * InputChannels)$$

- So for MNIST training images with a Conv2D layer of 64 3x3 filters, 1x1 stride, and "same" padding [padding = (filter_size – 1)/2 = 1], the runtime complexity is:

  60,000 * 64 * 28 * 28 * (3 * 3 * 1) = 27,095,040,000

- What happens to the runtime complexity if there are 64 input channels?

Notebooks and Other Examples

- Deep Learning with Python
  https://github.com/fchollet/deep-learning-with-python-notebooks

- Deep Learning Illustrated
  https://github.com/the-deep-learners/deep-learning-illustrated

- https://keras.io/examples/

PyTorch

- PyTorch, the other major neural network implementation framework, refers to a Dense layer as a Linear layer:

  https://discuss.pytorch.org/t/pytorch-equivalent-of-keras/29412

- While the implementation details vary somewhat, the underlying concepts for Tensorflow and PyTorch are the same

  “shields up” ☺