CHAPTER 5: DEEP LEARNING WITH COMPUTER VISION

1. Convolutional neural networks (convnets): universal used in computer vision application.

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'))

Convnet takes as input tensors of shape (image\_height, image\_width,

image\_channels).

output of every Conv2D and MaxPooling2D layer is a 3D tensor of

shape (height, width, channels) .

The width and height dimensions tend to shrink as you go deeper in the network. The number of channels is controlled by the first

argument passed to the Conv2D layers (32 or 64).

model.add(layers.Flatten())

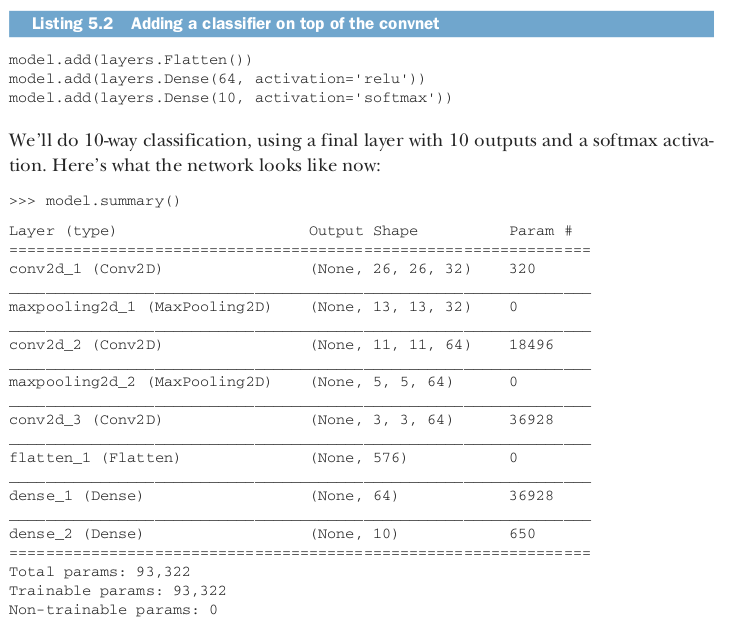
model.add(layers.Dense(64, activation='relu'))

model.add(layers.Dense(10, activation='softmax'))

Need to flatten the tensor into vector before feeding it indo Dense layer

2. The convolution operations: convolution layers learn local patterns

While Dense layer learn global.

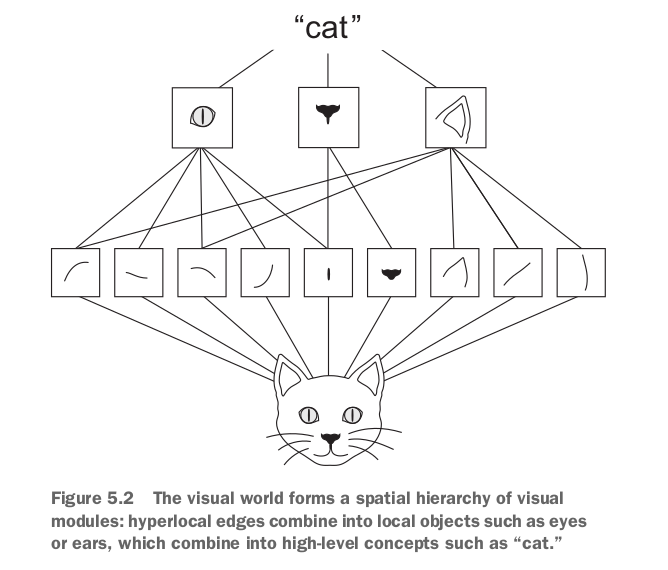




2 KEYS characteristics:

- The patterns they learn are translation invariant (can recognize from anywhere)

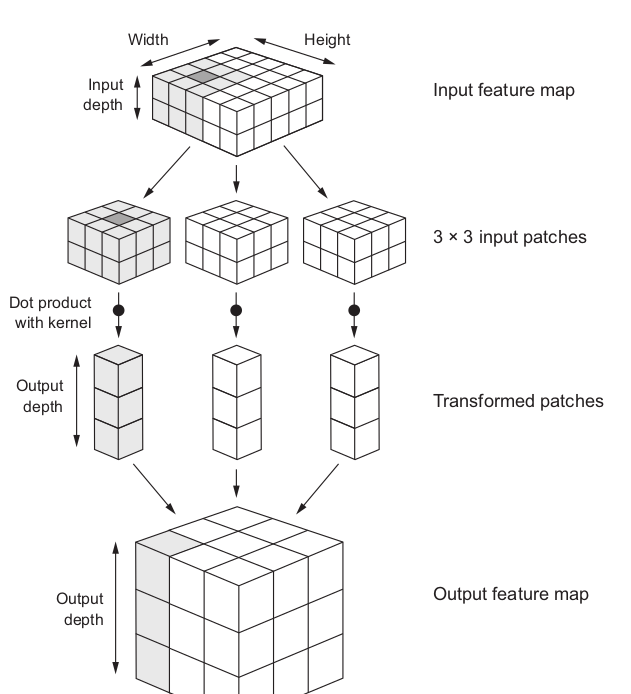
- They can learn spatial hierarchies of patterns (can learn abstract and complex visual concept) because later layer will learn patterns made by previous layers



3. Feature maps: 3D tensors with 2 spatial axes (height and width) as well a depth axis (channel). For RGB, dimension of depth is 3.

4. Output feature map: take input from a feature maps and applies the same transformation to all the patches. Output is still a feature map 3D.

- The new depth can be arbitrary, no longer stand for color like RGB, rather, now it stands for filters. Filter encode specific “aspects” (face...) of the input data.



5. Response map: output channels from filtered input.

6. Convolutions are defined by 2 key parameters

- Size of the patches extracted from the input (3x3 or 5x5)

- Depth of the output feature map: number of filters (32,64…)

Conv2D(output\_depth, (window\_height, window\_width))

7. Convolution kernel: tensor product with the same learned weight matrix

8. Output width and height may differ from the input width and height, because:

- Border effects, can be countered by padding the input feature map

- The use of strides

9. Padding (adding an appropriate number of rows and columns on each side of the input feature map so as to make it possible to fit center convolution windows around every input tile) to get output feature map the same with input.

- for 3x3, add 1 … for 5x5, add 2

padding argument = “valid”/”same”

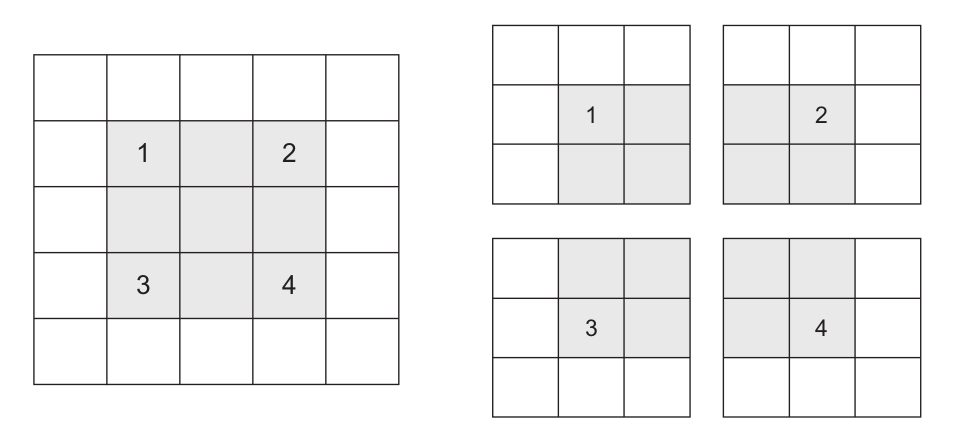
“valid” means no padding

10. Conv2D has padding argument: 2 option: “valid” or “same”

Default is “valid”, meaning no padding

11. Strides: is the distance between two successive windows, default is 1

- Using stride 2 means the width and height of the feature map are downsampled by a factor of 2, rarely used



12. Max-pooling (operation): used to downsample feature maps (instead of stride).

Same with stride operation, aggressively downsample

model.add(layers.MaxPooling2D((2, 2)))

- consists of extracting windows from the input feature maps and outputting the max value of each channel. (same concept with convolution – but not with linear transformation, instead by **max** tensor operation.

- Usually done with 2x2 windows and stride 2,

13. **In short, the reason to use downsampling is to reduce the number of feature-map**

**coefficients to process, as well as to induce spatial-filter hierarchies by making succes-sive convolution layers look at increasingly large windows (in terms of the fraction of the original input they cover).**

Also, can use stride or average pooling instead.

**In a nut-shell, the reason is that 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), and it’s more informative to look at the maximal presence of different features than at their average presence. So 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.**

14. **Together, these three strategies—training a small model fromscratch, doing feature extraction using a pretrained model, and fine-tuning a pre-trained model—will constitute your future toolbox for tackling the problem of per-forming image classification with small datasets.**

**15. convnets learn local, translation-invariant features, they’re highly data efficient on perceptual problems.** Training a convnet from scratch on a very small image dataset will still yield reasonable results despite a relative lack of data, without the need for any custom feature engineering. You’ll see this in action in this section

**CATs and DOGs**

16. So you do indeed have 2,000 training images, 1,000 validation images, and 1,000 test images. Each split contains the same number of samples from each class: this is a **balanced binary-classification problem**, which **means classification accuracy will be an appropriate measure of success.**

**17. Building our network.**

Same general structure with MNIST dataset

**- Conv2D** + **relu** + **MaxPooling2D**

However, since we are dealing with bigger images and a more complex problem, we will make our network accordingly larger: it will have one more Conv2D + MaxPooling2D stage. This serves both to augment the capacity of the network, and to further reduce the size of the feature maps, so that they aren't overly large when we reach the Flatten layer. Here, since we start from inputs of size 150x150 (a somewhat arbitrary choice), we end up with feature maps of size 7x7 right before the Flatten layer.

Note that the depth of the feature maps is progressively increasing in the network (from 32 to 128), while the size of the feature maps is decreasing (from 148x148 to 7x7). This is a pattern that you will see in almost all convnets.

18. **Data preprocessing**

data should be formatted into appropriately pre-processed floating point tensors before being fed into our network. Currently, our data sits on a drive as JPEG files, so the steps for getting it into our network are roughly:

* Read the picture files.
* Decode the JPEG content to RBG 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.preprocessing.image. In particular, it contains the class ImageDataGenerator

from keras.preprocessing.image import I**mageDataGenerator**

# All images will be rescaled by 1./255

train\_datagen = ImageDataGenerator(rescale=1./255)

test\_datagen = ImageDataGenerator(rescale=1./255)

train\_generator = train\_datagen.flow\_from\_directory(

# This is the target directory

train\_dir,

# All images will be resized to 150x150

target\_size=(150, 150),

batch\_size=20,

# Since we use binary\_crossentropy loss, we need binary labels

class\_mode='binary')

validation\_generator = test\_datagen.flow\_from\_directory(

validation\_dir,

target\_size=(150, 150),

batch\_size=20,

class\_mode='binary')

for data\_batch, labels\_batch in train\_generator:

print('data batch shape:', data\_batch.shape)

print('labels batch shape:', labels\_batch.shape)

break

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')

**#Plot the result**

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()

19. **Less data => more severe overfitting.**

**USE A PRETRAINED CONVNET**

20. VGG16 architecture - convnet architecture for ImageNet.

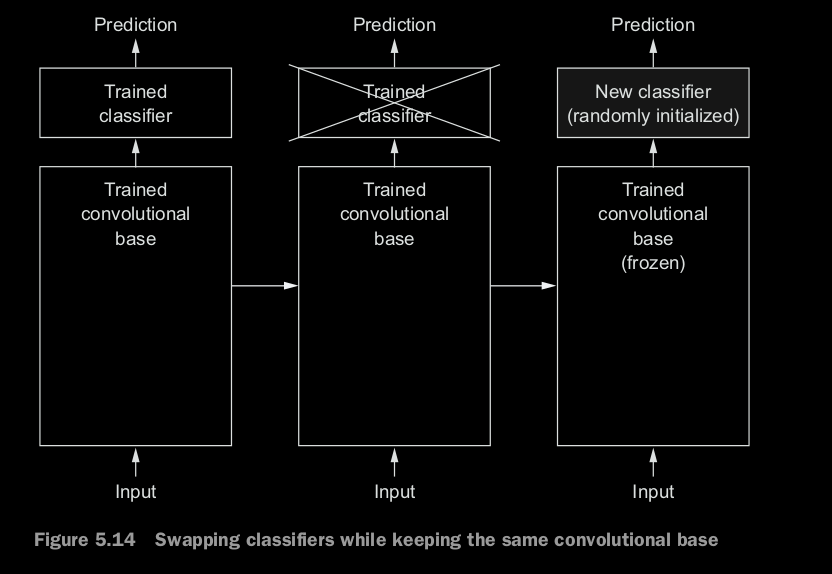
VGG, ResNet, Inception, Inception-ResNet, Xception, and so on.

21. Two ways: **feature extraction** and **fine-tuning**

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

23. **Convolutional base** of the model (base layers like pooling and convolution and densely connected. In **feature extraction** and **convnet,** use

previous trained network, run new data on it, and train new classifier on top of the output. DO NOT USE densely classifier layer in base.



24. **Generality <=> reusability** depends on the depth of the layer in the model. The earlier of the layers, the deeper.

The deeper layers tend to capture more detailed features, later layers capture more abstract feature (like ear, eye).

So, if new data differ a lot from original data, better use only first few layers of the base, rather than the whole base.

25.

from keras.applications import VGG16

conv\_base = VGG16(weights='imagenet', #check point

include\_top=False, #include original classifier or not

input\_shape=(150, 150, 3)) #shape of tensor input

26. 2 ways to work with convolutional base:

- Run the base and save as numpy array => just add an classifier on top (Cannot do data augmentation)

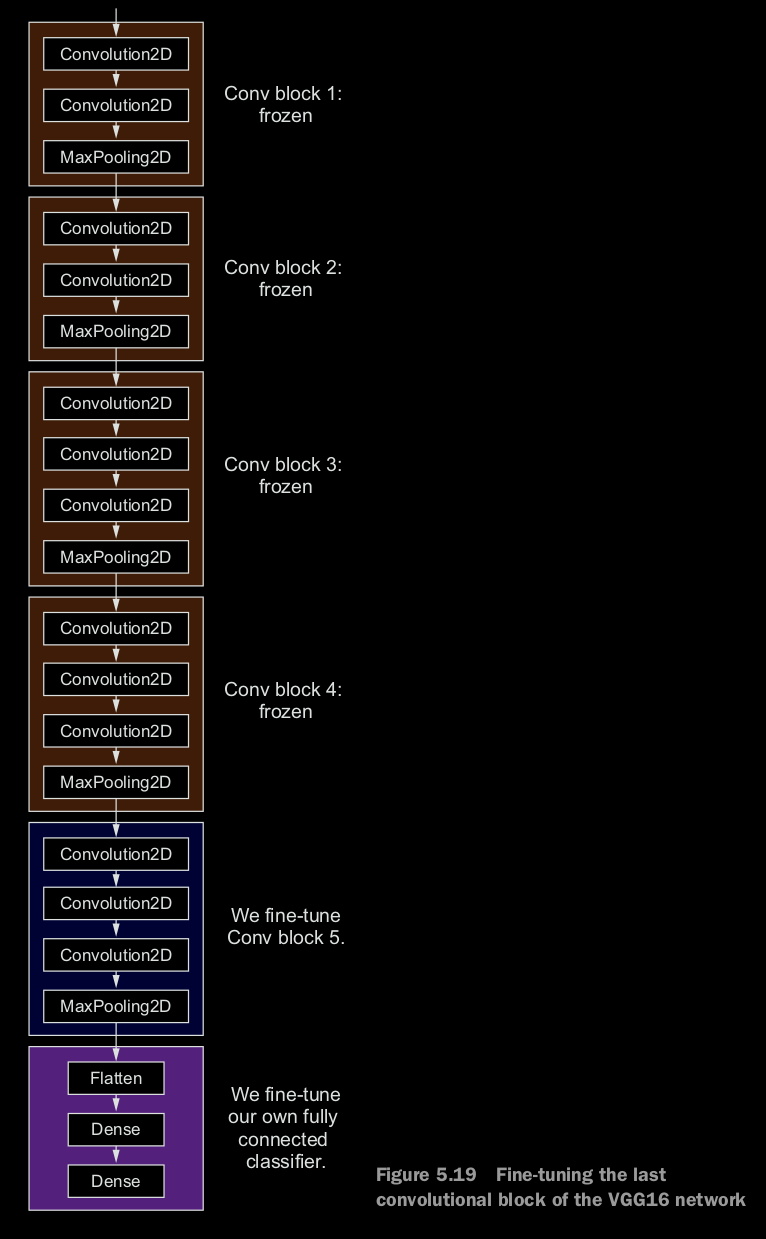
- Run every input with the base (computational expensive)

27. Remember to **freeze** the base layers to avoid the weight from being updated.

conv\_base.trainable = False

len(model.trainable\_weights) #check how many trainable weight

28. **Fine-tuning:** unfreeze 1 or 2 conv last block(s), so it slightly adjusts the more abstract representations of the model being reused => more relevant for the problem at hand.



29. **Unfreeze last networks**

conv\_base.trainable = True

set\_trainable = False

for layer in conv\_base.layers:

if layer.name == 'block5\_conv1':

set\_trainable = True

if set\_trainable:

layer.trainable = True

else:

layer.trainable = False

Now you can begin fine-tuning the network. You’ll do this with the RMSP rop optimizer, using a very low learning rate. The reason for using a low learning rate is that you want to limit the magnitude of the modifications you make to the representations of the three layers you’re fine-tuning. Updates that are too large may harm these representations.

30. **Smooth the noisy graph** by replacing every loss and accuracy with exponential moving averages of these quantities.

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

31. **Why loss curve worse when accuracy curve stays the same?**

The answer is simple: what you display is an average of pointwise loss values; but what matters for accuracy is the distribution of the loss values, not their average, because accuracy is the result of a binary thresholding of the class probability predicted by the model. The model may still be improving even if this isn’t reflected in the average loss.

**Wrapping up**

**Here’s what you should take away from the exercises in the past two sections:**

** 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.**

**32. Visualizing intermediate convnet outputs (intermediate activations)**

displaying the feature maps that are output by various convolution and pooling layers in a network. Output of activation func is called ‘activation’.

Give insight how an input is decomposed into many different filters.

Visualize feature maps with three dimensions: **width, height, depth (channels)**

img\_path = '/Users/fchollet/Downloads/cats\_and\_dogs\_small/test/cats/cat.1700.jpg'

# We preprocess the image into a 4D tensor

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)

# Remember that the model was trained on inputs

# that were preprocessed in the following way:

img\_tensor /= 255.

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

print(img\_tensor.shape)

import matplotlib.pyplot as plt

plt.imshow(img\_tensor[0])

plt.show()

**33. Model (**keras class): In order to extract the feature maps we want to look at, we will create a Keras model that takes batches of images as input, and outputs the activations of all convolution and pooling layers.

from keras import models

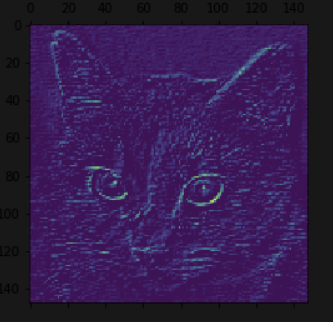
# Extracts the outputs of the top 8 layers:

layer\_outputs = [layer.output for layer in model.layers[:8]]

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

activation\_model = models.Model(inputs=model.input, outputs=layer\_outputs)

**In the general case, a model could have any number of inputs and outputs. This one has one input and 8 outputs, one output per layer activation.**



# This will return a list of 5 Numpy arrays:

# one array per layer activation

activations = activation\_model.predict(img\_tensor)

first\_layer\_activation = activations[0]

print(first\_layer\_activation.shape)

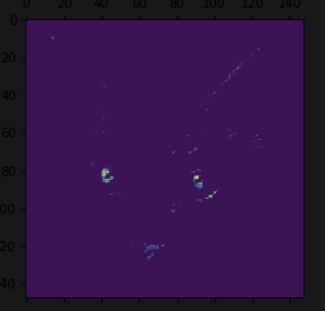
(1, 148, 148, 32) #32 channels

import matplotlib.pyplot as plt

plt.matshow(first\_layer\_activation[0, :, :, 3], cmap='viridis') #3rd channel

#diagonal edge detector

plt.show()

plt.matshow(first\_layer\_activation[0, :, :, 30], cmap='viridis') #30th channel

plt.show() #”bright green dot” detector

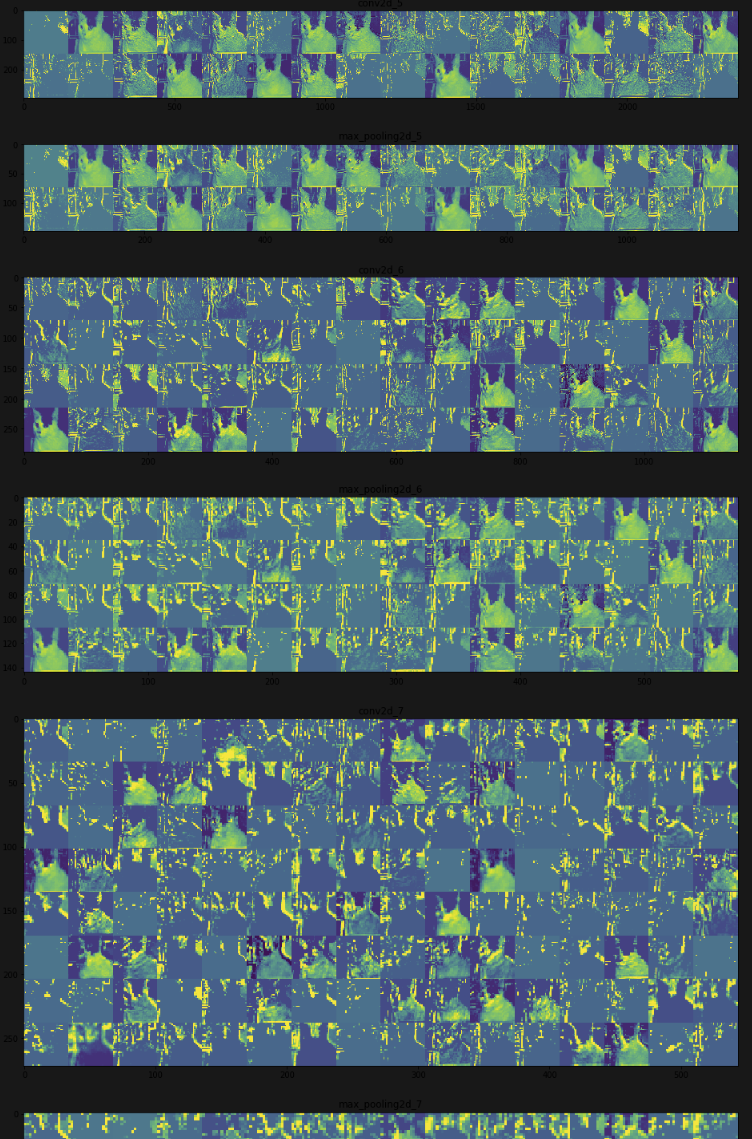
#useful for cat eyes

**Code for display all the channels**

import keras

# These are the names of the layers, so can have them as part of our plot

layer\_names = []

for layer in model.layers[:8]:

layer\_names.append(layer.name)

images\_per\_row = 16

# Now let's display our feature maps

for layer\_name, layer\_activation in zip(layer\_names, activations):

# This is the number of features in the feature map

n\_features = layer\_activation.shape[-1]

# The feature map has shape (1, size, size, n\_features)

size = layer\_activation.shape[1]

# We will tile the activation channels in this matrix

n\_cols = n\_features // images\_per\_row

display\_grid = np.zeros((size \* n\_cols, images\_per\_row \* size))

# We'll tile each filter into this big horizontal grid

for col in range(n\_cols):

for row in range(images\_per\_row):

channel\_image = layer\_activation[0,

:, :,

col \* images\_per\_row + row]

# Post-process the feature to make it visually palatable

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

# Display the grid

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')

plt.show()

**A few remarkable things to note here:**

* **The first layer acts as a collection of various edge detectors. At that stage, the activations are still retaining almost all of the information present in the initial picture.**
* **As we go higher-up, the activations become increasingly abstract and less visually interpretable. They start encoding higher-level concepts such as "cat ear" or "cat eye". Higher-up presentations carry increasingly less information about the visual contents of the image, and increasingly more information related to the class of the image.**
* **The sparsity of the activations is increasing with the depth of the layer: in the first layer, all filters are activated by the input image, but in the**
* **following layers more and more filters are blank. This means that the pattern encoded by the filter isn't found in the input image.**

34**. information distillation pipeline:** with raw data going in (in our case, RBG pictures), and getting repeatedly transformed so that irrelevant information gets filtered out (e.g. the specific visual appearance of the image) while useful information get magnified and refined (e.g. the class of the image).

## **35. Visualizing convnet filters:** inspect the filters learned by convnets is to display the visual pattern that each filter is meant to respond to. Using **gradient ascent in input space,** apply **gradient descent** to the value of the input image of a convnet so as to maximize the response of a specific filter (from blank input image). Result is maximized filter.

## **Process:** build a loss function that maximizes the value of a given filter in a given convolution layers => use **stochastic gradient descent** to adjust the values of input => maximize this activation value

## 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

## def generate\_pattern(layer\_name, filter\_index, size=150):

## *# Build a loss function that maximizes the activation*

## *# of the nth filter of the layer considered.*

## layer\_output = model.get\_layer(layer\_name).output

## loss = K.mean(layer\_output[:, :, :, filter\_index])

## *# Compute the gradient of the input picture wrt this loss*

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

## *# Normalization trick: we normalize the gradient*

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

## *# This function returns the loss and grads given the input picture*

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

## *# We start from a gray image with some noise*

## input\_img\_data = np.random.random((1, size, size, 3)) \* 20 + 128.

## *# Run gradient ascent for 40 steps*

## 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)

## plt.imshow(generate\_pattern('block3\_conv1', 0))

**Read more about this please =>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> code in notebook**

These filter visualizations tell us a lot about how convnet layers see the world: each layer in a convnet simply learns a collection of filters such that their inputs can be expressed as a combination of the filters. This is 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 we go higher-up in the model:

* The filters from the first layer in the model (block1\_conv1) encode simple directional edges and colors (or colored edges in some cases).
* The filters from block2\_conv1 encode simple textures made from combinations of edges and colors.
* The filters in higher-up layers start resembling textures found in natural images: feathers, eyes, leaves, etc.

**36. Visualizing heatmaps of class activation:** useful to understand which parts of a given image led a convnet to its final classification decision. For “**debug**” the decision made by the convnet.

**- Class Activation Map (CAM)** visualization: consists in producing heatmaps of "class activation" over input images. A "class activation" heatmap is a 2D grid of scores associated with an specific output class, computed for every location in any input image, indicating how important each location is with respect to the class considered.

it consists in taking 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, one way to understand this trick is that we are 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".

37. **Code used to load image and resize, turn to tensor and add more**

from keras.preprocessing import image

from keras.applications.vgg16 import preprocess\_input, decode\_predictions

import numpy as np

# The local path to our target image

img\_path = '/Users/fchollet/Downloads/creative\_commons\_elephant.jpg'

# `img` is a PIL image of size 224x224

img = image.load\_img(img\_path, target\_size=(224, 224))

# `x` is a float32 Numpy array of shape (224, 224, 3)

x = image.img\_to\_array(img)

# We add a dimension to transform our array into a "batch"

# of size (1, 224, 224, 3)

x = np.expand\_dims(x, axis=0)

# Finally we preprocess the batch

# (this does channel-wise color normalization)

x = preprocess\_input(x)dimension

**#make prediction and evaluation percent**

preds = model.predict(x)

print('Predicted:', decode\_predictions(preds, top=3)[0])

np.argmax(preds[0]) #index of the class.

38. **To visualize which parts of the image were the most “African elephent” like.**

# This is the "african elephant" entry in the prediction vector

african\_elephant\_output = model.output[:, 386]

# The is the output feature map of the `block5\_conv3` layer,

# the last convolutional layer in VGG16

last\_conv\_layer = model.get\_layer('block5\_conv3')

# This is the 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]

# This is a vector of shape (512,), where each entry

# is the mean intensity of the gradient over a specific feature map channel

pooled\_grads = K.mean(grads, axis=(0, 1, 2))

# This function allows us to access the values of the quantities we just defined:

# `pooled\_grads` and the output feature map of `block5\_conv3`,

# given a sample image

iterate = K.function([model.input], [pooled\_grads, last\_conv\_layer.output[0]])

# These are the values of these two quantities, as Numpy arrays,

# given our sample image of two elephants

pooled\_grads\_value, conv\_layer\_output\_value = iterate([x])

# We multiply each channel in the feature map array

# by "how important this channel is" with regard to the elephant class

for i in range(512):

conv\_layer\_output\_value[:, :, i] \*= pooled\_grads\_value[i]

# The channel-wise mean of the resulting feature map

# is our heatmap of class activation

heatmap = np.mean(conv\_layer\_output\_value, axis=-1)

heatmap = np.maximum(heatmap, 0)

heatmap /= np.max(heatmap)

plt.matshow(heatmap)

plt.show()

**USE OPENCV TO GENERATE IMAGE THAT SUPERIMPOSES THE ORIGINAL WITH THE HEATMAP.**

import cv2

# We use cv2 to load the original image

img = cv2.imread(img\_path)

# We resize the heatmap to have the same size as the original image

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

# We convert the heatmap to RGB

heatmap = np.uint8(255 \* heatmap)

# We apply the heatmap to the original image

heatmap = cv2.applyColorMap(heatmap, cv2.COLORMAP\_JET)



# 0.4 here is a heatmap intensity factor

superimposed\_img = heatmap \* 0.4 + img

# Save the image to disk

cv2.imwrite('/Users/fchollet/Downloads/elephant\_cam.jpg', superimposed\_img)