
بسم الله الرحمن الرحیم

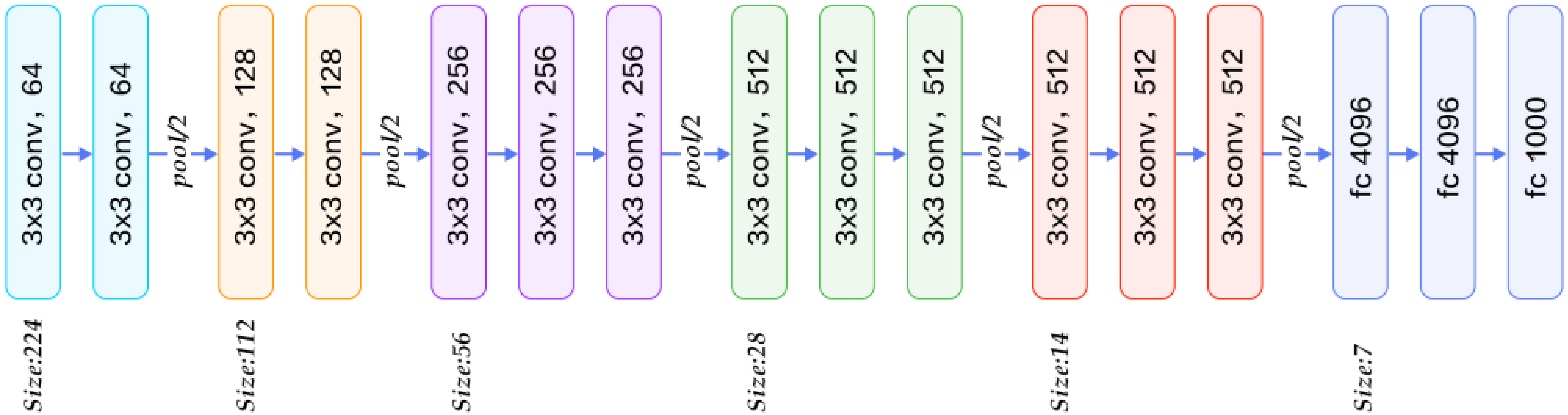
Deep Learning

Mohammad Reza Mohammadi

2021

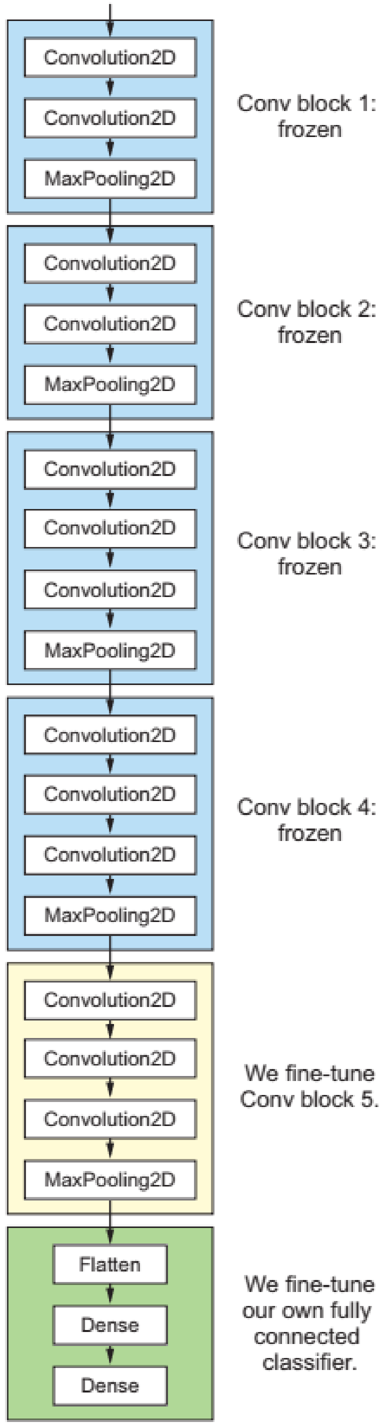
Pretrained convnet

- There are two typical ways to use a pre-trained network:
  - Feature extraction
  - Fine tuning



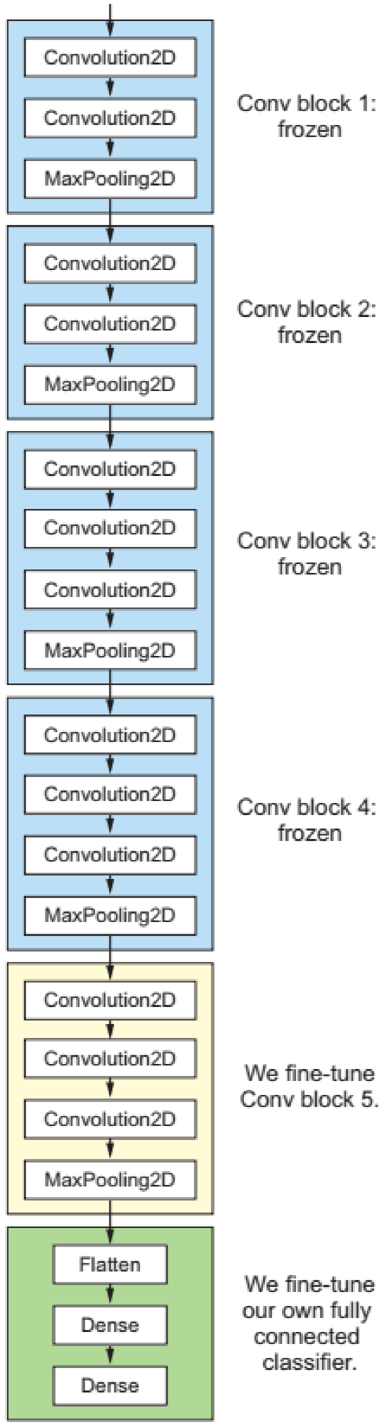
Fine-tuning

- Another widely used technique for model reuse
- Unfreeze a few of the top layers of a frozen model base used for feature extraction, and jointly train both the newly added part of the model (in this case, the fully connected classifier) and these top layers



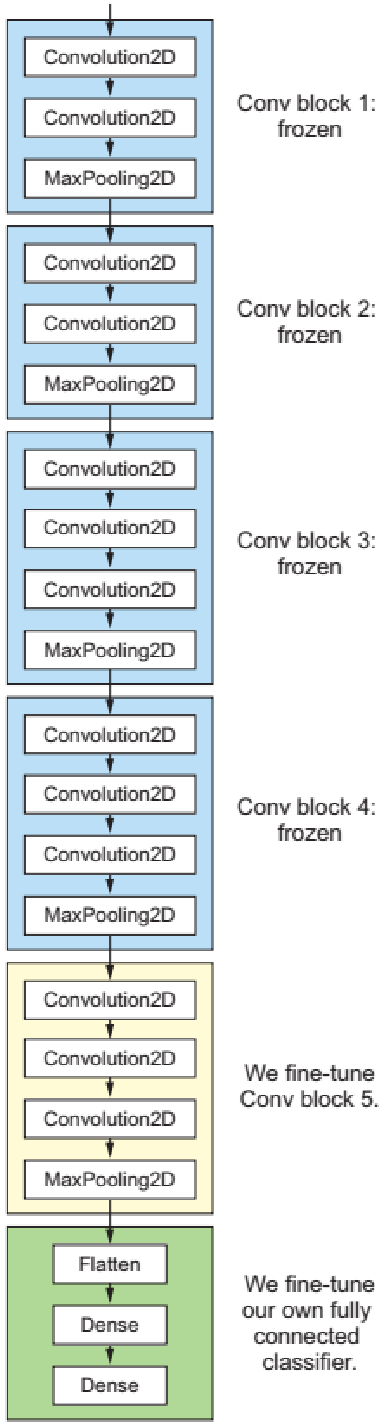
Fine Tuning

- It is suggested to fine-tune the top layers of the convolutional base once the classifier on top has already been trained
- If the classifier isn’t already trained, then the error signal propagating through the network during training will be too large, and the representations previously learned by the layers being fine-tuned will be destroyed



Fine Tuning

- Thus the steps for fine-tuning a network are as follow:
  - 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

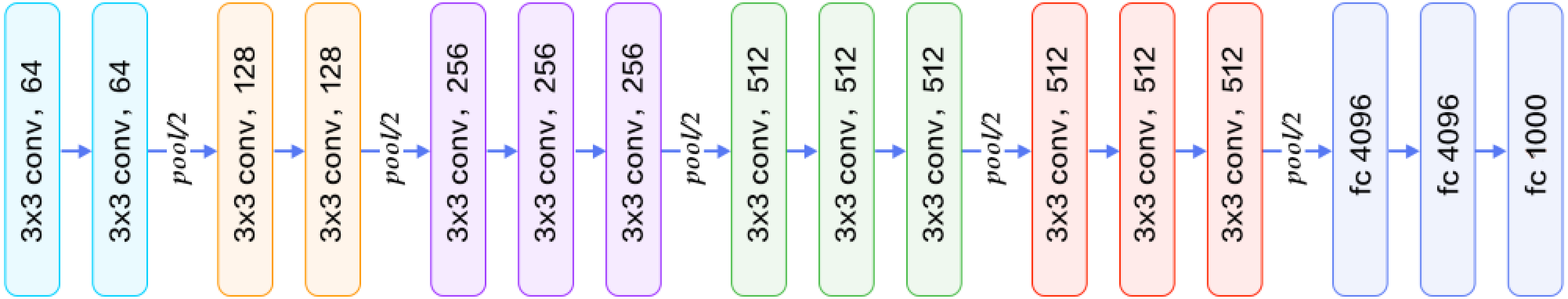


https://colab.research.google.com/drive/1t6DF8L1cZfhtgE6qBHHgELmNf5TjbZPI?usp=sharing

Visualization

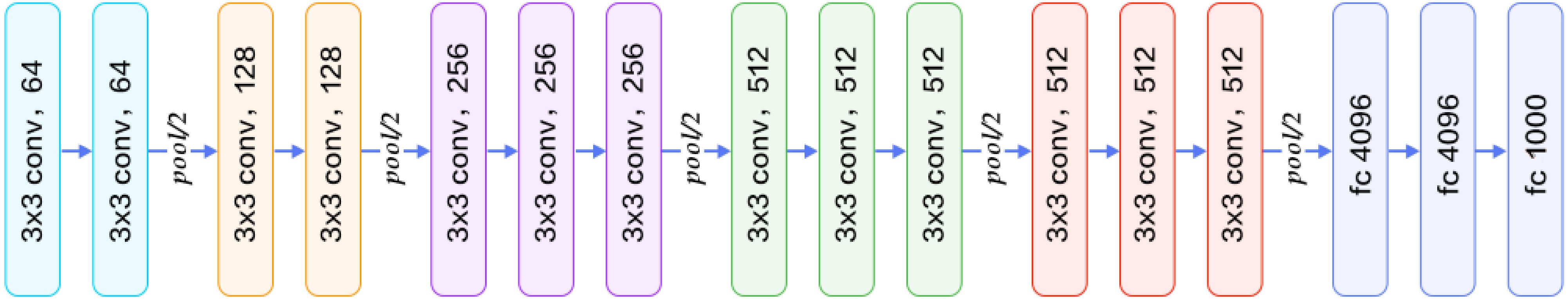
Visualizing what convnets learn

- It's often said that deep-learning models are "black boxes"
  - definitely not true for convnets
- The representations learned by convnets are highly amenable to visualization, in large part because they're representations of visual concepts



Visualizing what convnets learn

- Different ways to visualize or interpret NN representations, three common ones:
  - 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

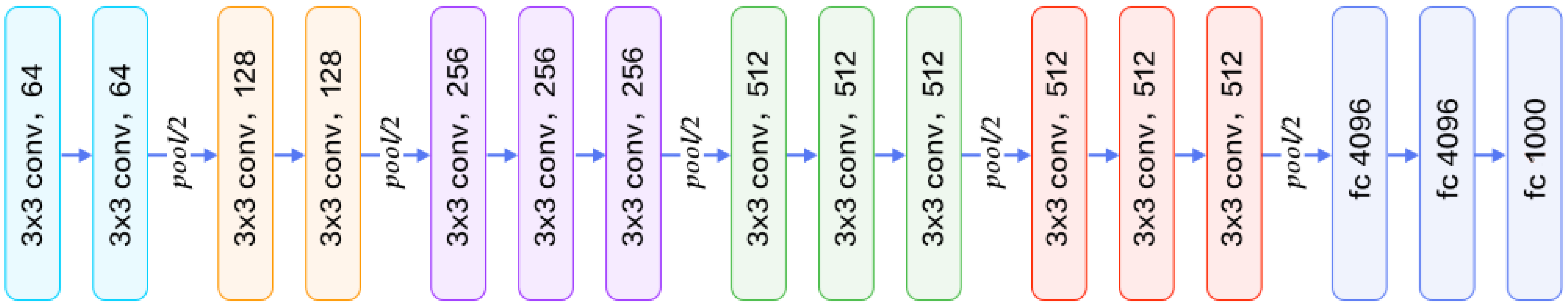


Visualizing what convnets learn

- Different ways to visualize or interpret NN representations, three common ones:
  - 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

Visualizing intermediate activations

- Displaying the feature maps that are output by various convolution and pooling layers in a network, given a certain input
- This gives a view into how an input is decomposed into the different filters learned by the network
- Feature maps have three dimensions: width, height, and depth (channels)



```
>>> from keras.models import load_model
>>> model = load_model('cats_and_dogs_small_2.h5')
>>> model.summary()
```

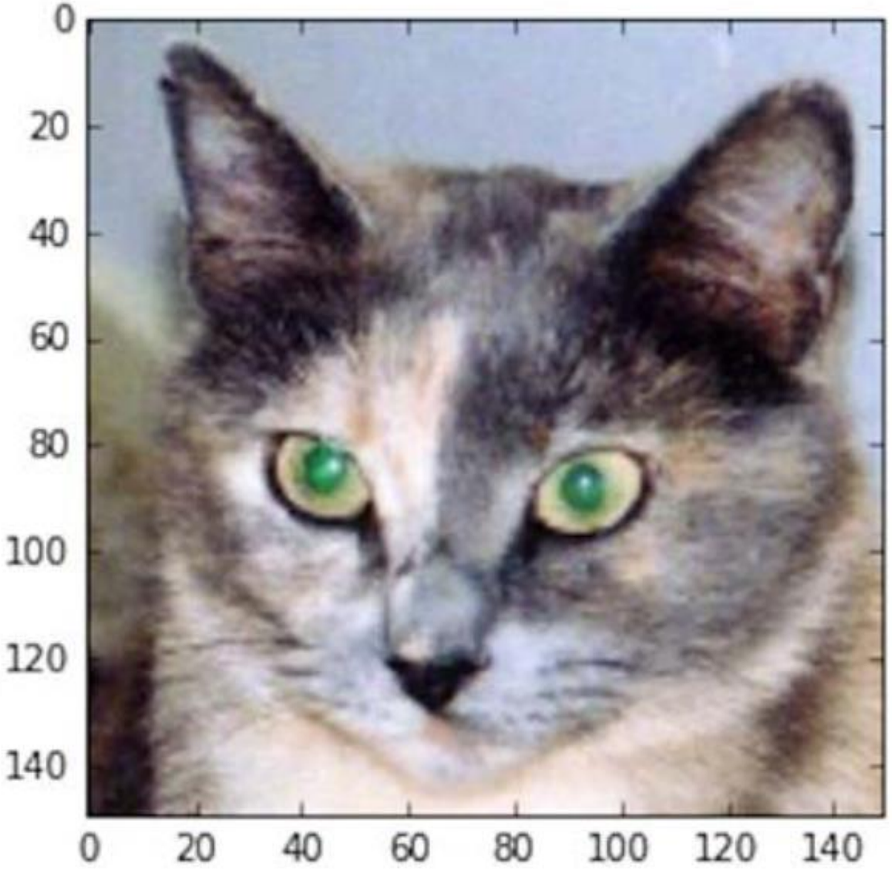
```
________________________________________________________________
Layer (type)                     Output Shape          Param #
================================================================
conv2d_5 (Conv2D)                (None, 148, 148, 32)  896
________________________________________________________________
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
```

Listing 5.25 Preprocessing 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.

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


import matplotlib.pyplot as plt

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

Preprocesses the image into a 4D tensor

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



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)
```

Extracts the outputs of the top eight layers

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

Listing 5.28 Running the model in predict mode

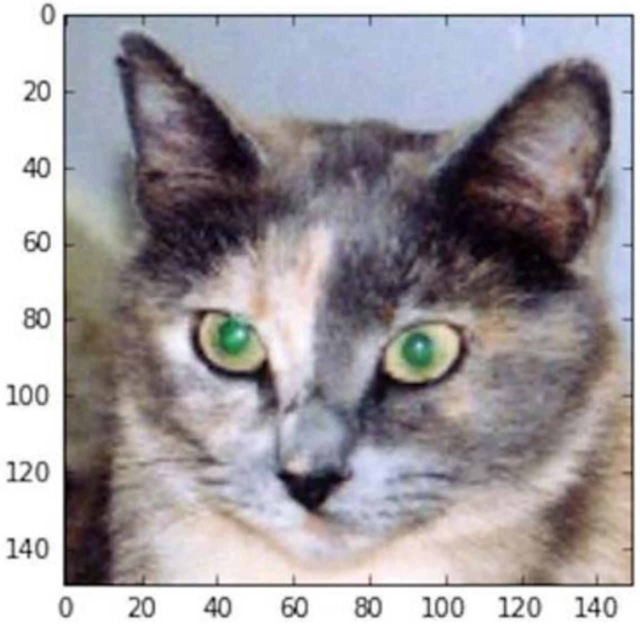
```
activations = activation_model.predict(img_tensor)
```

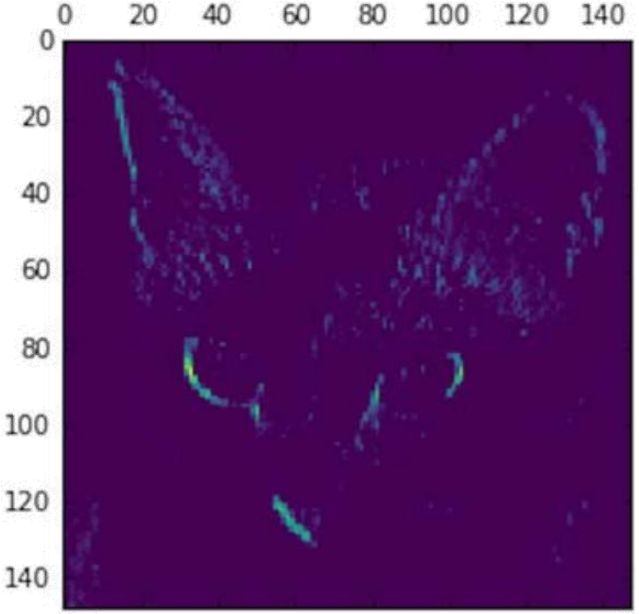
Reeight a list of five eight Numpy arrays: one array per layer activation

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

Listing 5.29 Visualizing the fourth channel

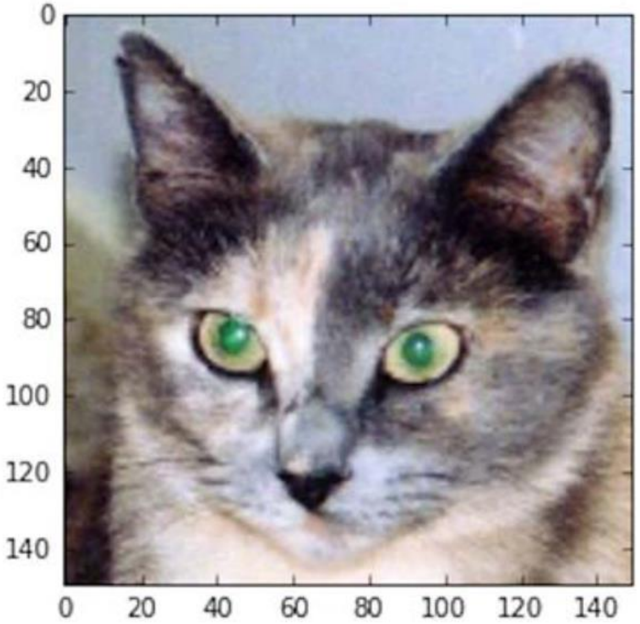
```
import matplotlib.pyplot as plt

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

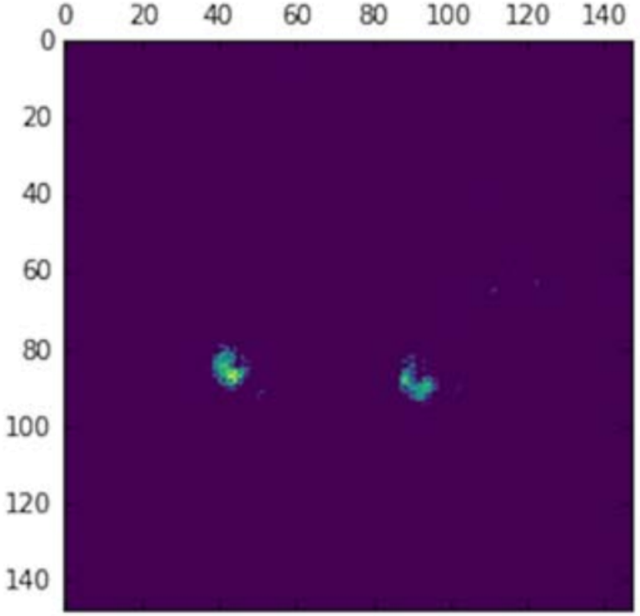




Listing 5.30 Visualizing the seventh channel

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





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')
```

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

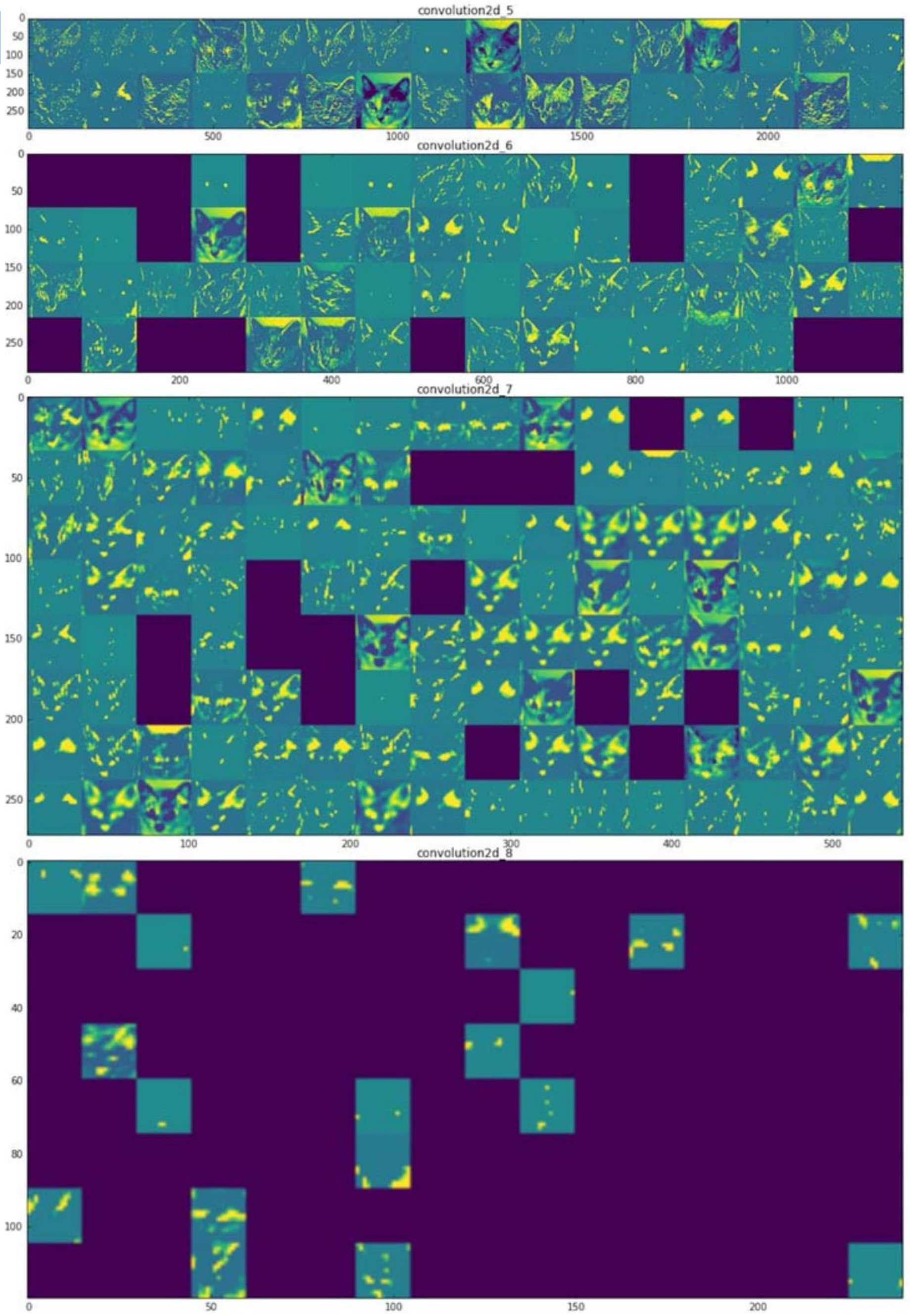
The feature map has shape (l, size, size, n_features).

Tiles the activation channels in this matrix

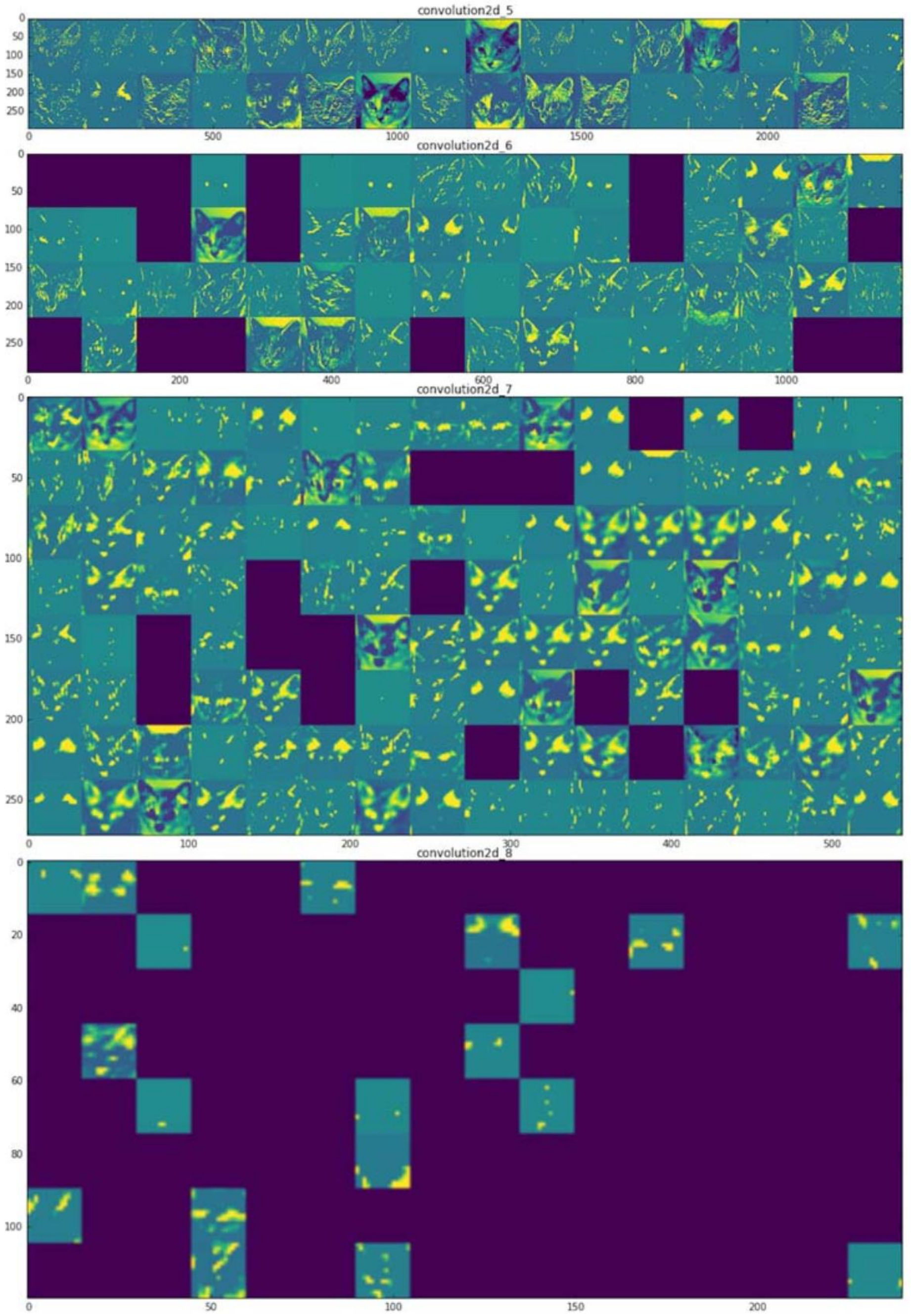
Tiles each filter into a big horizontal grid

Post-processes the feature to make it visually palatable

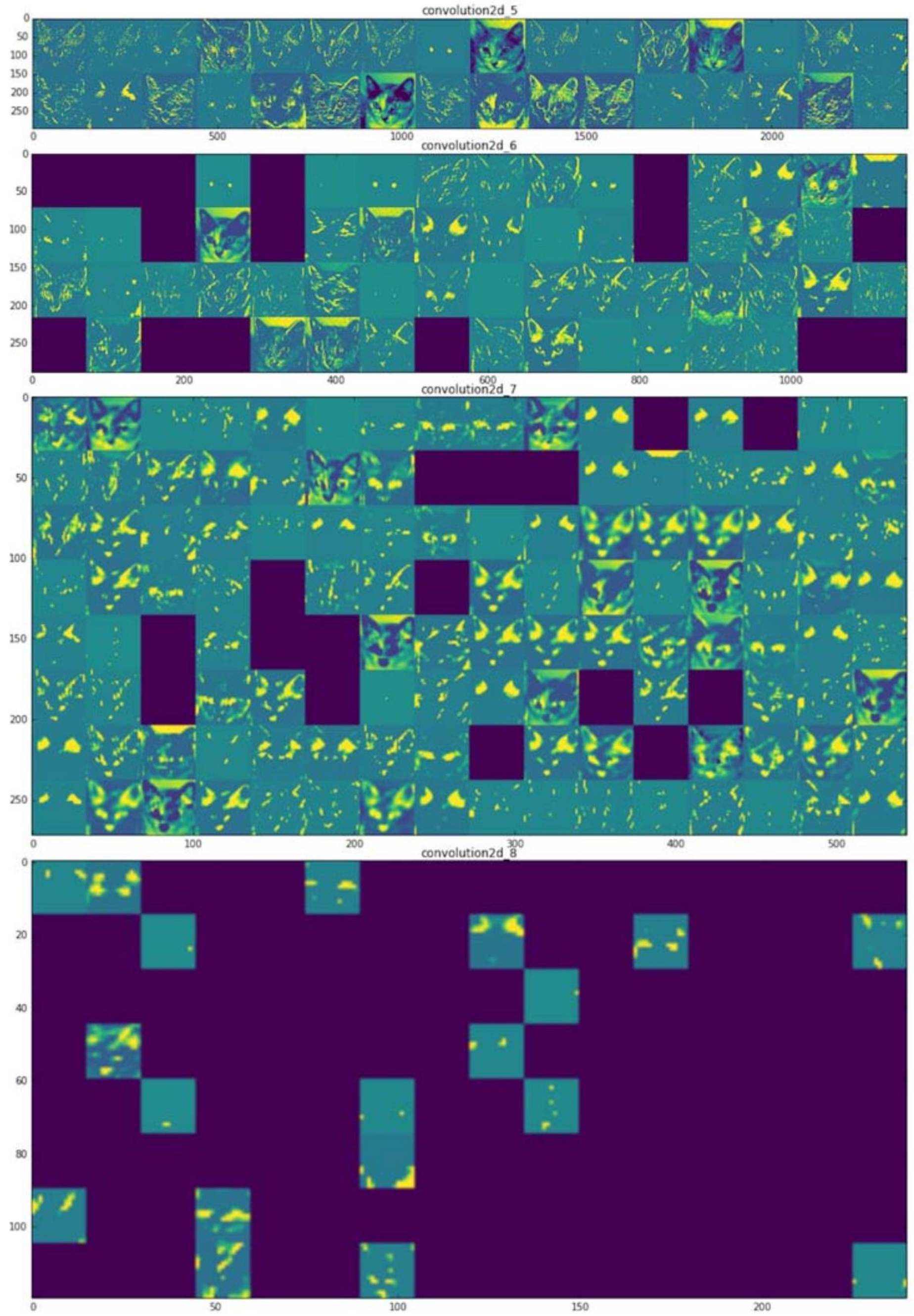
Displays the grid



- The first layer acts as a collection of various edge detectors
  - At that stage, the activations retain almost all of the information present in the initial picture
- As you go higher, the activations become increasingly abstract and less visually interpretable
  - They begin to encode higher-level concepts such as “cat ear” and “cat eye.”
  - Higher 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 increases 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 the pattern encoded by the filter isn't found in the input image

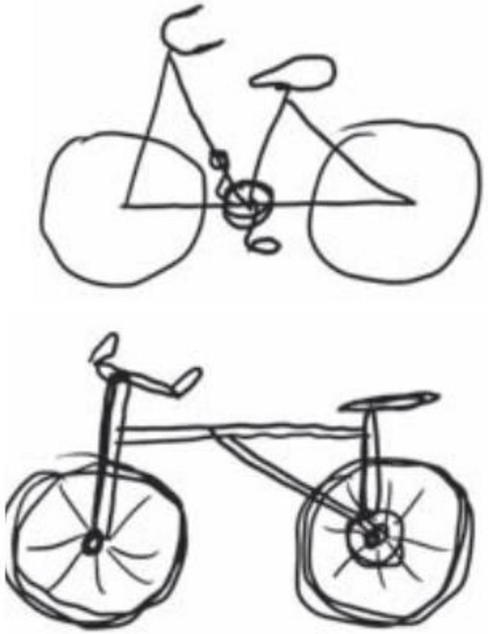


Representations

- The features extracted by a layer become increasingly abstract with the depth of the layer
  - The activations of higher layers carry less and less information about the specific input being seen, and more and more information about the target
- Information distillation pipeline
  - Raw data going in (in this case, RGB pictures) and being repeatedly transformed so that irrelevant information is filtered out (for example, the specific visual appearance of the image), and useful information is magnified and refined (for example, the class of the image)

Representations

- Analogous to the way humans and animals perceive the world
  - Brain has learned to completely abstract its visual input (to transform it into high-level visual concepts while filtering out irrelevant visual details)



Visualizing convnet filters

- Another easy way to inspect the filters learned by convnets is to display the visual pattern that each filter is meant to respond to
- This can be done with 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, starting from a blank input image
- The resulting input image will be one that the chosen filter is maximally responsive to
- Build a loss function that maximizes the value of a given filter in a given convolution layer
- Use stochastic gradient descent to adjust the values of the input image

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 5.38 Function to generate filter visualizations

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

Starts from a gray image with some noise

Runs gradient ascent for 40 steps

```
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)
```

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

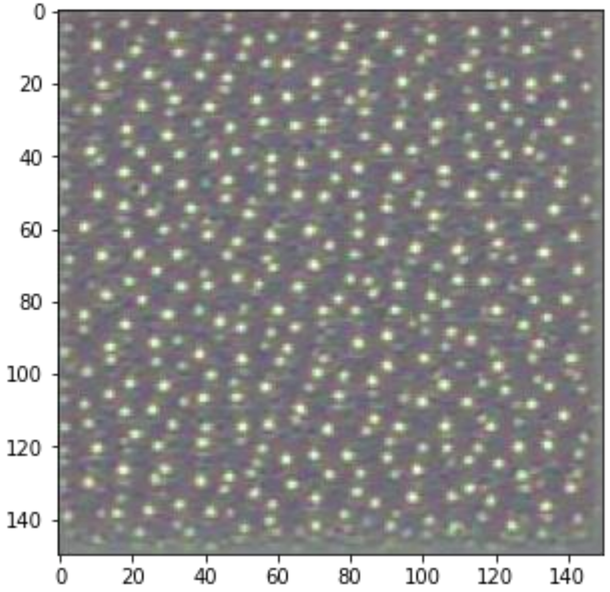
```
def deprocess_image(x):
    x -= x.mean()
    x /= (x.std() + 1e-5)
    x *= 0.1

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

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

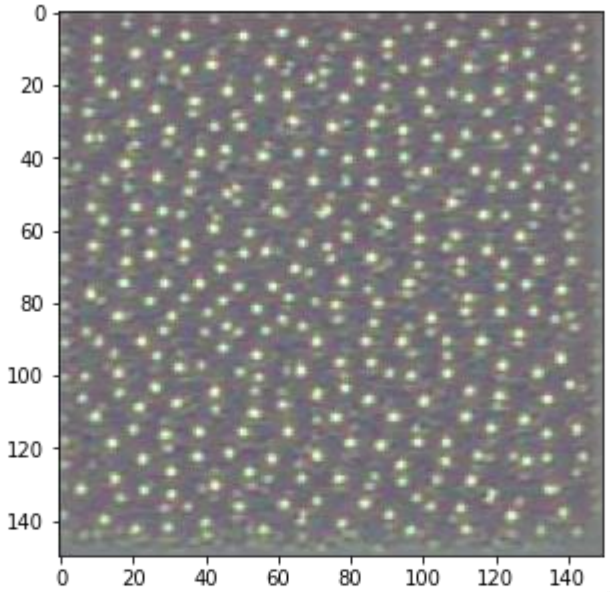
Normalizes the tensor: centers on 0, ensures that std is 0.1

Clips to [0, 1]

Converts to an RGB array

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





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)
```

Empty (black) image to store results

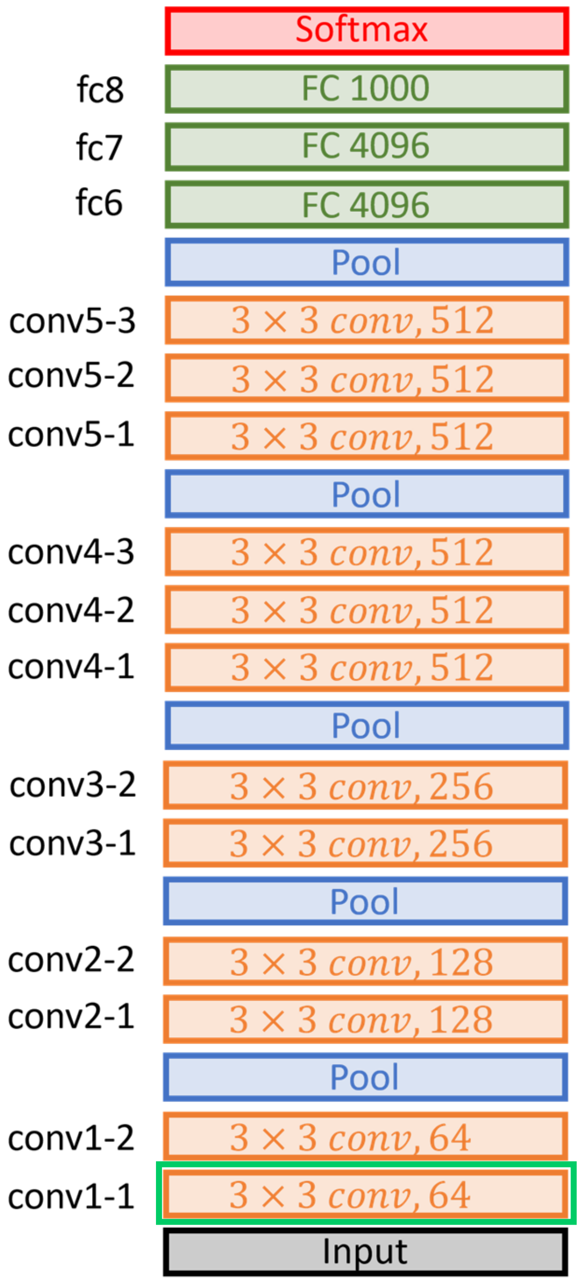
Iterates over the rows of the results grid

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

Displays the results grid



VGG16

64, 3×3×3 filters