

```
In [0]: import numpy as np
import matplotlib as mp
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import tensorflow.contrib.slim as slim
from tensorflow.examples.tutorials.mnist import input_data
import math
```

Mnist database contains images of thousands of handwritten digits, and their proper labels.

```
In [39]: mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

The CNN model we designed is a simple network with 3 sets of convolution (pooling layers) followed by a fully connected softmax layer. We use the tensorflow.contrib.slim while implementing the network as we could easily define the layer variable.

```
In [0]: tf.reset_default_graph()

x = tf.placeholder(tf.float32, [None, 784], name="x-in")
true_y = tf.placeholder(tf.float32, [None, 10], name="y-in")
keep_prob = tf.placeholder("float")

x_image = tf.reshape(x, [-1, 28, 28, 1])
hidden_1 = slim.conv2d(x_image, 5, [5, 5])
pool_1 = slim.max_pool2d(hidden_1, [2, 2])
hidden_2 = slim.conv2d(pool_1, 5, [5, 5])
pool_2 = slim.max_pool2d(hidden_2, [2, 2])
hidden_3 = slim.conv2d(pool_2, 20, [5, 5])
hidden_3 = slim.dropout(hidden_3, keep_prob)
out_y = slim.fully_connected(slim.flatten(hidden_3), 10, activation_fn=tf.nn.softmax)

cross_entropy = -tf.reduce_sum(true_y * tf.log(out_y))
correct_prediction = tf.equal(tf.argmax(out_y, 1), tf.argmax(true_y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
```

We then train the network using batch (50) gradient descent with Adam optimization.

```
In [40]: batchSize = 50
sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)
for i in range(1001):
    batch = mnist.train.next_batch(batchSize)
    sess.run(train_step, feed_dict={x:batch[0],true_y:batch[1], keep_prob:0.5})
    if i % 100 == 0 and i != 0:
        trainAccuracy = sess.run(accuracy, feed_dict={x:batch[0],true_y:batch[1], keep_prob:1.0})
        print("step %d, training accuracy %g"%(i, trainAccuracy))
```

```
step 100, training accuracy 0.14
step 200, training accuracy 0.64
step 300, training accuracy 0.8
step 400, training accuracy 0.7
step 500, training accuracy 0.82
step 600, training accuracy 0.9
step 700, training accuracy 0.86
step 800, training accuracy 0.92
step 900, training accuracy 0.88
step 1000, training accuracy 0.96
```

```
In [31]: testAccuracy = sess.run(accuracy, feed_dict={x:mnist.test.images,true_y:svhn.test.labels, keep_prob:1.0})
print("test accuracy %g"%(testAccuracy))
```

```
test accuracy 0.8939
```

We define a couple functions that will allow us to visualize the network. The first gets the activations at a given layer for a given input image. The second plots those activations in a grid.

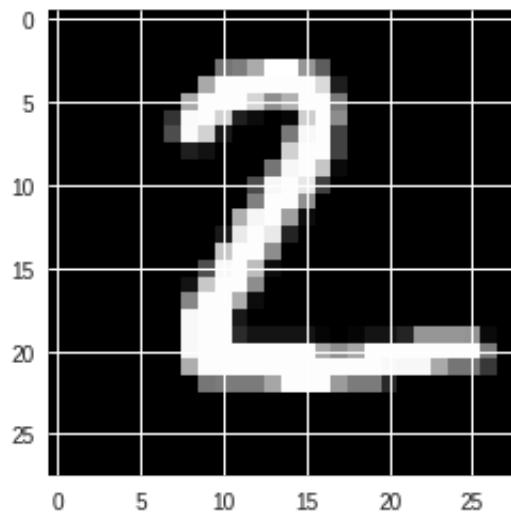
```
In [0]: def getActivations(layer,stimuli):
    units = sess.run(layer,feed_dict={x:np.reshape(stimuli,[1,784],order='F'),keep_prob:1.0})
    plotNNFilter(units)
```

```
In [0]: def plotNNFilter(units):
    filters = units.shape[3]
    plt.figure(1, figsize=(20,20))
    n_columns = 6
    n_rows = math.ceil(filters / n_columns) + 1
    for i in range(filters):
        plt.subplot(n_rows, n_columns, i+1)
        plt.title('Filter ' + str(i))
        plt.imshow(units[0,:,:,:,i], interpolation="nearest", cmap="gray")
    "
```

Choose an image from the database to visualize.

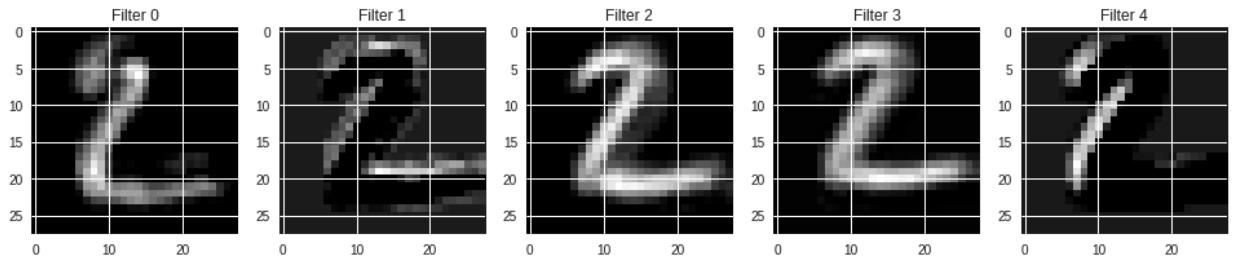
```
In [51]: imageToUse = mnist.test.images[1]
plt.imshow(np.reshape(imageToUse,[28,28]), interpolation="nearest", cmap="gray")
```

```
Out[51]: <matplotlib.image.AxesImage at 0x7f9308b0b0b8>
```

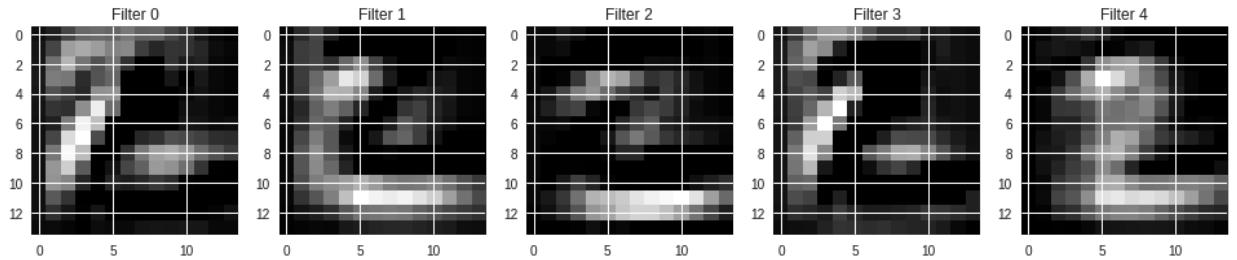


We can look at how that image activates the neurons of the first convolutional layer. We notice that each filter has learned differently to activate optimally for different features of the image. Also, as layer increases, the images which activate feature optimally will be less clear than previous layers'.

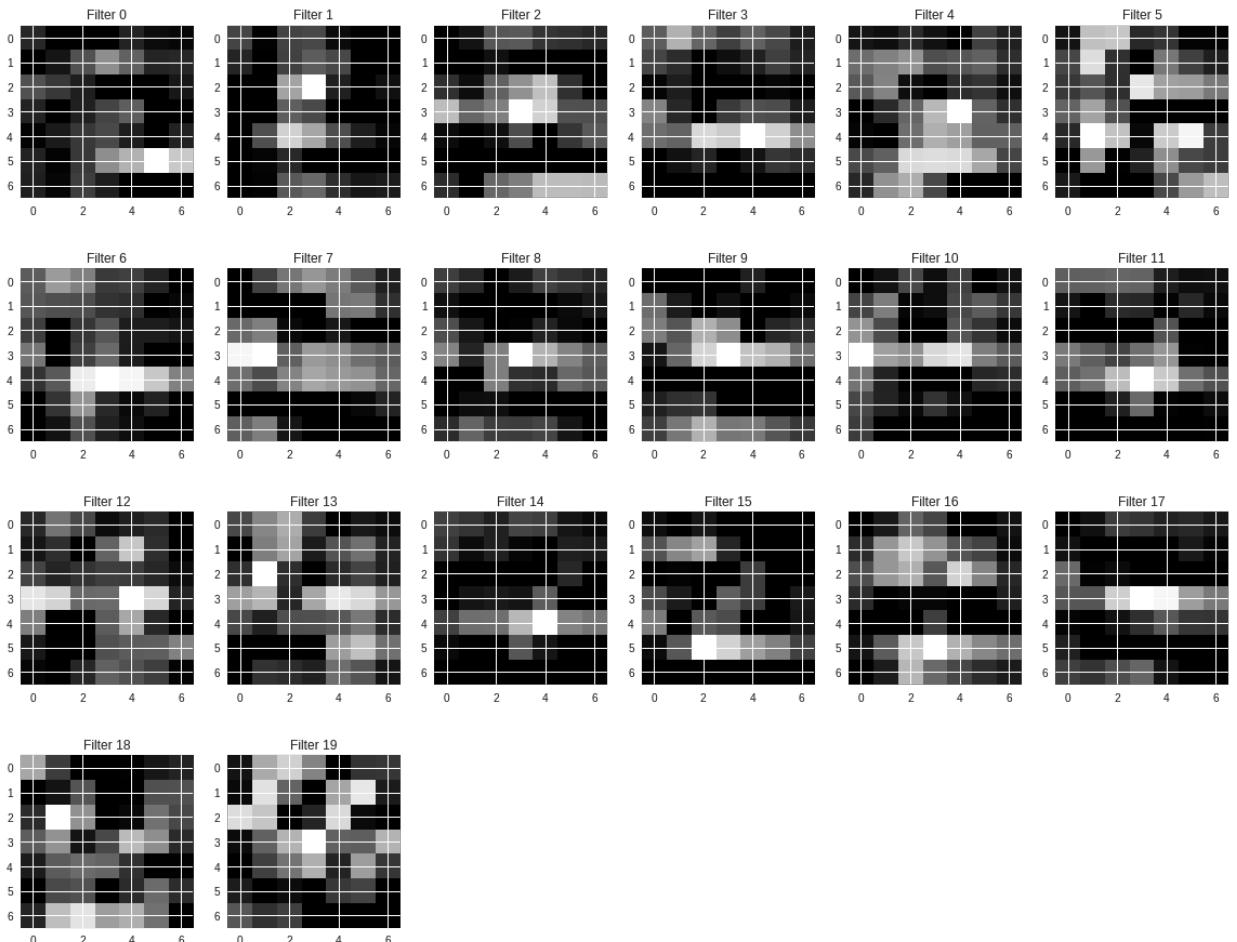
```
In [52]: getActivations(hidden_1,imageToUse)
```



```
In [53]: getActivations(hidden_2,imageToUse)
```



```
In [54]: getActivations(hidden_3,imageToUse)
```



```
In [0]: !pip install --quiet lucid==0.0.5

import numpy as np
import scipy.ndimage as nd
import tensorflow as tf

import lucid.modelzoo.vision_models as models
from lucid.misc.io import show
import lucid.optvis.objectives as objectives
import lucid.optvis.param as param
import lucid.optvis.render as render
import lucid.optvis.transform as transform
```

```
In [0]: # Let's import a model from the Lucid modelzoo!

model = models.InceptionV1()
model.load_graphdef()
```

```
In [57]: param_f = lambda: param.image(128, batch=2)
obj = objectives.channel("mixed4d_pre_relu", 479, batch=1) - objectives.
    channel("mixed4d_pre_relu", 479, batch=0)
_ = render.render_vis(model, obj, param_f)
```

512 2491.6746



```
In [3]: param_f = lambda: param.image(128, batch=2)
obj = objectives.channel("mixed4a_pre_relu", 240, batch=1) - objectives.
    channel("mixed4a_pre_relu", 240, batch=0)
_ = render.render_vis(model, obj, param_f)
```

512 2544.724



CNN Layer-Level Feature Visualization

```
In [1]: import os, shutil
```

```
In [26]: import matplotlib.pyplot as plt
```

The cats vs. dogs dataset that we will use isn't packaged with Keras. It was made available by Kaggle.com as part of a computer vision competition in late 2013, back when convnets weren't quite mainstream. You can download the original dataset at: <https://www.kaggle.com/c/dogs-vs-cats/data> (<https://www.kaggle.com/c/dogs-vs-cats/data>)

```
In [6]: train_dir = os.path.join(base_dir, 'train')
```

```
In [ ]: # The path to the directory where the original
# dataset was uncompressed
original_dataset_dir = '/Users/shilinli/Downloads/all/train'

# The directory where we will
# store our smaller dataset
base_dir = '/Users/shilinli/Downloads/cats_and_dogs_small'
os.mkdir(base_dir)

# Directories for our training,
# validation and test splits
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)

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

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

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

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

# Directory with our validation cat pictures
test_cats_dir = os.path.join(test_dir, 'cats')
os.mkdir(test_cats_dir)

# Directory with our validation dog pictures
test_dogs_dir = os.path.join(test_dir, 'dogs')
os.mkdir(test_dogs_dir)

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

# Copy next 500 cat images to validation_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)

# Copy next 500 cat images to test_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)

# Copy first 1000 dog images to train_dogs_dir
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)

# Copy next 500 dog images to validation_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)

# Copy next 500 dog images to test_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)
```

```
In [ ]: print('total validation dog images:', len(os.listdir(validation_dogs_dir)))
```

Building the Network

We've already built a small convnet for MNIST in the previous example. 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.

Since we are attacking a binary classification problem, we are ending the network with a single unit (a Dense layer of size 1) and a sigmoid activation. This unit will encode the probability that the network is looking at one class or the other.

```
In [11]: 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'))
```

```
In [12]: model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 148, 148, 32)	896
max_pooling2d_1 (MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_2 (Conv2D)	(None, 72, 72, 64)	18496
max_pooling2d_2 (MaxPooling2D)	(None, 36, 36, 64)	0
conv2d_3 (Conv2D)	(None, 34, 34, 128)	73856
max_pooling2d_3 (MaxPooling2D)	(None, 17, 17, 128)	0
conv2d_4 (Conv2D)	(None, 15, 15, 128)	147584
max_pooling2d_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
<hr/>		
Total params: 3,453,121		
Trainable params: 3,453,121		
Non-trainable params: 0		

For our compilation step, we'll go with the RMSprop optimizer as usual. Since we ended our network with a single sigmoid unit, we will use binary crossentropy as our loss.

```
In [14]: from keras import optimizers  
  
model.compile(loss='binary_crossentropy',  
              optimizer=optimizers.RMSprop(lr=1e-4),  
              metrics=['acc'])
```

Data Preprocessing

Processing Steps:

- 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).

```
In [15]: from keras.preprocessing.image import ImageDataGenerator

# 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')

Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
```

Let's take a look at the output of one of these generators: it yields batches of 150x150 RGB images (shape (20, 150, 150, 3)) and binary labels (shape (20,)). 20 is the number of samples in each batch (the batch size). Note that the generator yields these batches indefinitely: it just loops endlessly over the images present in the target folder. For this reason, we need to break the iteration loop at some point.

```
In [16]: 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,)
```

Let's fit our model to the data using the generator. We do it using the `fit_generator` method, the equivalent of `fit` for data generators like ours. It expects as first argument a Python generator that will yield batches of inputs and targets indefinitely, like ours does. Because the data is being generated endlessly, the generator needs to know example how many samples to draw from the generator before declaring an epoch over. This is the role of the `steps_per_epoch` argument: after having drawn `steps_per_epoch` batches from the generator, i.e. after having run for `steps_per_epoch` gradient descent steps, the fitting process will go to the next epoch. In our case, batches are 20-sample large, so it will take 100 batches until we see our target of 2000 samples.

When using `fit_generator`, one may pass a `validation_data` argument, much like with the `fit` method. Importantly, this argument is allowed to be a data generator itself, but it could be a tuple of Numpy arrays as well. If you pass a generator as `validation_data`, then this generator is expected to yield batches of validation data endlessly, and thus you should also specify the `validation_steps` argument, which tells the process how many batches to draw from the validation generator for evaluation.

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

```
In [18]: model.save('cats_and_dogs_small_1.h5')
```

```
In [ ]: import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(len(acc))

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```

Overfitting is caused by having too few samples to learn from, rendering us unable to train a model able to generalize to new data. Given infinite data, our model would be exposed to every possible aspect of the data distribution at hand: we would never overfit. Data augmentation takes the approach of generating more training data from existing training samples, by "augmenting" the samples via a number of random transformations that yield believable-looking images. The goal is that at training time, our model would never see the exact same picture twice. This helps the model get exposed to more aspects of the data and generalize better.

In Keras, this can be done by configuring a number of random transformations to be performed on the images read by our `ImageDataGenerator` instance. Let's get started with an example:

```
In [ ]: 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')
```

These are just a few of the options available (for more, see the Keras documentation). Let's quickly go over what we just wrote:

- 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 of the images horizontally -- relevant when there are no assumptions of horizontal asymmetry (e.g. 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.

```
In [ ]: # This is module with image preprocessing utilities
from keras.preprocessing import image

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

# We pick one image to "augment"
img_path = fnames[3]

# Read the image and resize it
img = image.load_img(img_path, target_size=(150, 150))

# Convert it to a Numpy array with shape (150, 150, 3)
x = image.img_to_array(img)

# Reshape it to (1, 150, 150, 3)
x = x.reshape((1,) + x.shape)

# The .flow() command below generates batches of randomly transformed
# images.
# It will loop indefinitely, so we need to `break` the loop at some point!
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()
```

If we train a new network using this data augmentation configuration, our network will never see twice the same input. However, the inputs that it sees are still heavily intercorrelated, since they come from a small number of original images -- we cannot produce new information, we can only remix existing information. As such, this might not be quite enough to completely get rid of overfitting. To further fight overfitting, we will also add a Dropout layer to our model, right before the densely-connected classifier:

```
In [ ]: 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'])
```

Train our network using data augmentation and dropout:

```
In [ ]: 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,)  
  
# Note that the validation data should not be augmented!  
test_datagen = ImageDataGenerator(rescale=1./255)  
  
train_generator = train_datagen.flow_from_directory(  
    # This is the target directory  
    train_dir,  
    # All images will be resized to 150x150  
    target_size=(150, 150),  
    batch_size=32,  
    # 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=32,  
    class_mode='binary')  
  
history = model.fit_generator(  
    train_generator,  
    steps_per_epoch=100,  
    epochs=100,  
    validation_data=validation_generator,  
    validation_steps=50)
```

```
In [ ]: model.save('cats_and_dogs_small_2.h5')
```

```
In [ ]: acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(len(acc))

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```

Thanks to data augmentation and dropout, we are no longer overfitting: the training curves are rather closely tracking the validation curves. We are now able to reach an accuracy of 82%, a 15% relative improvement over the non-regularized model.

By leveraging regularization techniques even further and by tuning the network's parameters (such as the number of filters per convolution layer, or the number of layers in the network), we may be able to get an even better accuracy, likely up to 86-87%. However, it would prove very difficult to go any higher just by training our own convnet from scratch, simply because we have so little data to work with. As a next step to improve our accuracy on this problem, we will have to leverage a pre-trained model, which will be the focus of the next two sections.

```
In [19]: import keras
keras.__version__
```

```
Out[19]: '2.2.4'
```

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. To do this, we will use the Keras class Model. A Model is instantiated using two arguments: an input tensor (or list of input tensors), and an output tensor (or list of output tensors).

```
In [9]: from keras.models import load_model
```

```
model = load_model('/Users/shilinli/Downloads/cats_and_dogs_small_2.h5')
model.summary() # As a reminder.
```

Layer (type)	Output Shape	Param #
<hr/>		
conv2d_5 (Conv2D)	(None, 148, 148, 32)	896
max_pooling2d_5 (MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_6 (Conv2D)	(None, 72, 72, 64)	18496
max_pooling2d_6 (MaxPooling2D)	(None, 36, 36, 64)	0
conv2d_7 (Conv2D)	(None, 34, 34, 128)	73856
max_pooling2d_7 (MaxPooling2D)	(None, 17, 17, 128)	0
conv2d_8 (Conv2D)	(None, 15, 15, 128)	147584
max_pooling2d_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
<hr/>		
Total params: 3,453,121		
Trainable params: 3,453,121		
Non-trainable params: 0		

When fed an image input, this model returns the values of the layer activations in the original model. This is the first time we encounter a multi-output model in this book: until now the models we have seen only had exactly one input and one output. 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.

```
In [23]: img_path = '/Users/shilinli/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

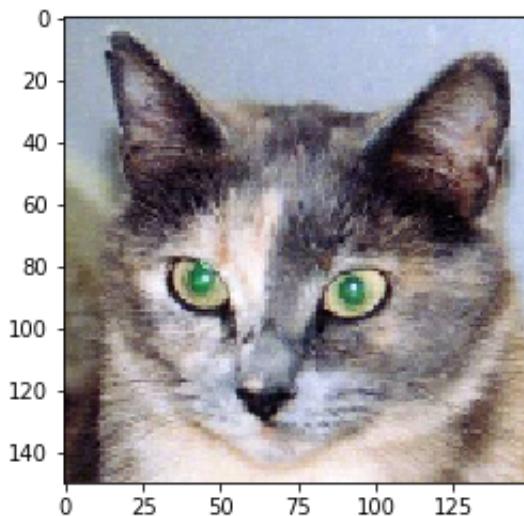
img = image.load_img(img_path, target_size=(150, 150))
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
# Remember that the model was trained on inputs
# that were preprocessed in the following way:
img_tensor /= 255.

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

```
(1, 150, 150, 3)
```

```
In [24]: import matplotlib.pyplot as plt

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



```
In [25]: from keras import models

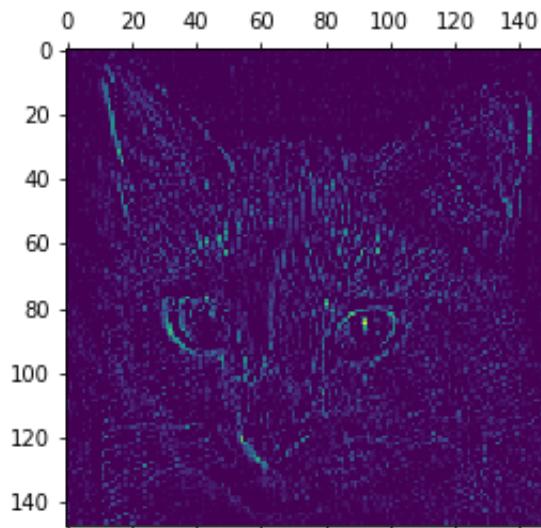
# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
In [26]: # This will return a list of 5 Numpy arrays:  
# one array per layer activation  
activations = activation_model.predict(img_tensor)
```

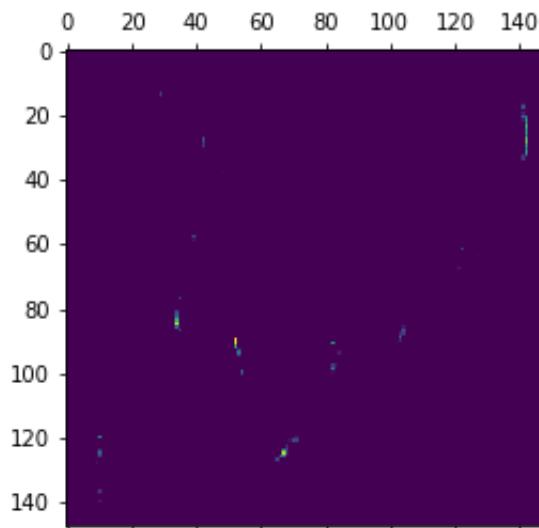
```
In [27]: first_layer_activation = activations[0]  
print(first_layer_activation.shape)  
  
(1, 148, 148, 32)
```

It's a 148x148 feature map with 32 channels. Let's try visualizing the 3rd channel:

```
In [28]: import matplotlib.pyplot as plt  
  
plt.matshow(first_layer_activation[0, :, :, 3], cmap='viridis')  
plt.show()
```



```
In [29]: plt.matshow(first_layer_activation[0, :, :, 30], cmap='viridis')
plt.show()
```



This one looks like a "bright green dot" detector, useful to encode cat eyes. At this point, let's go and plot a complete visualization of all the activations in the network. We'll extract and plot every channel in each of our 8 activation maps, and we will stack the results in one big image tensor, with channels stacked side by side.

In [30]: 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]

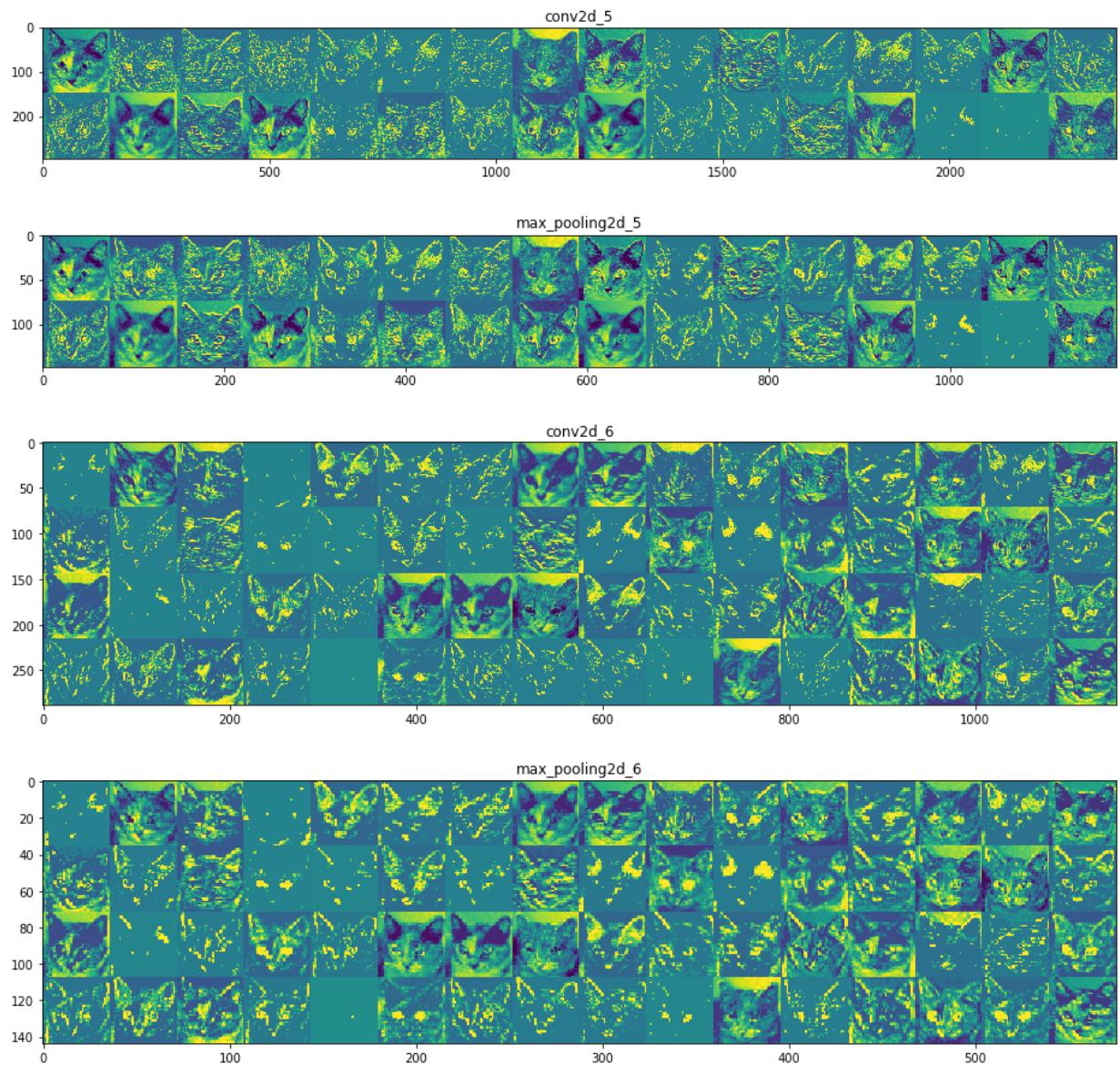
    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

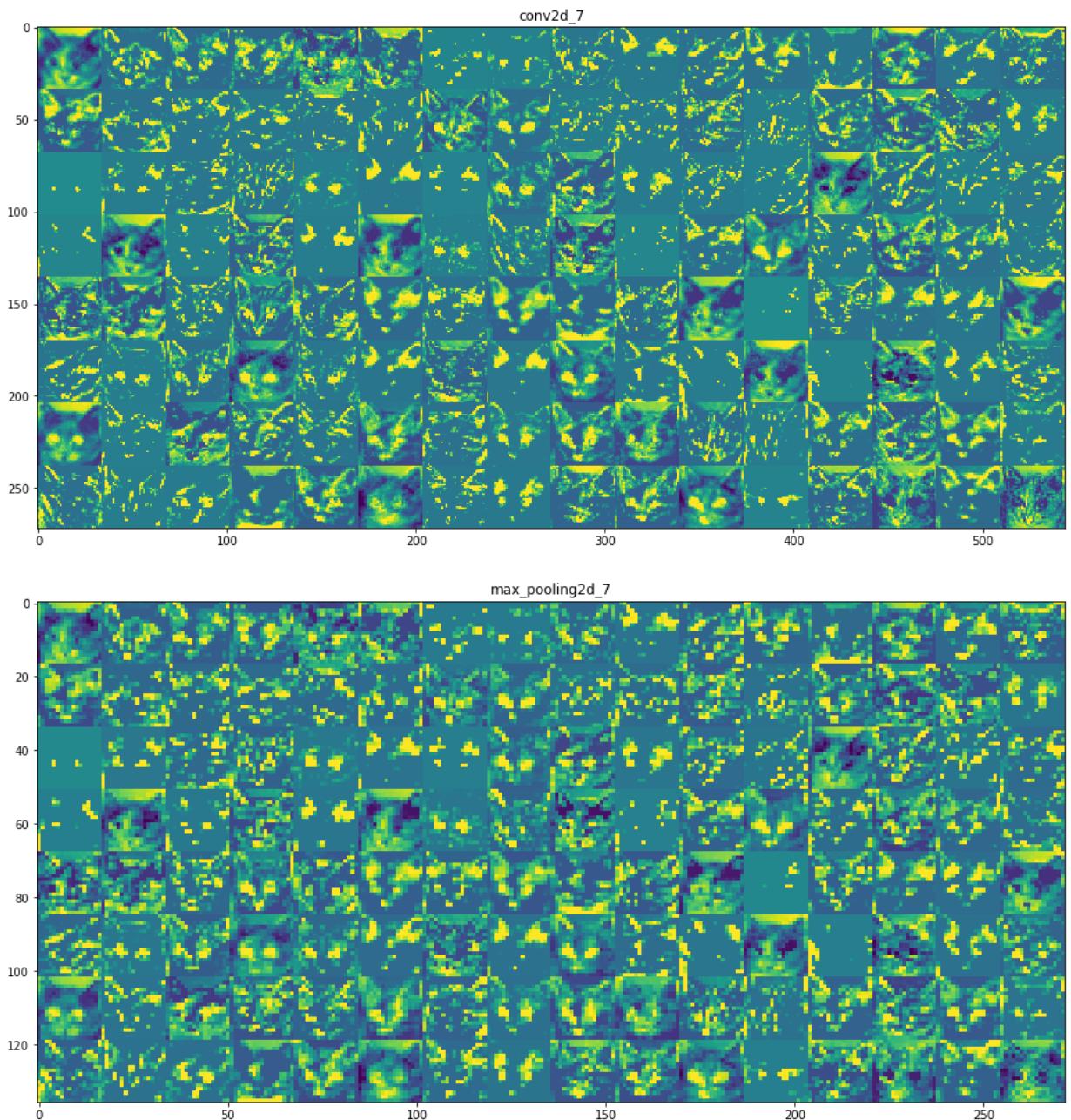
    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # 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

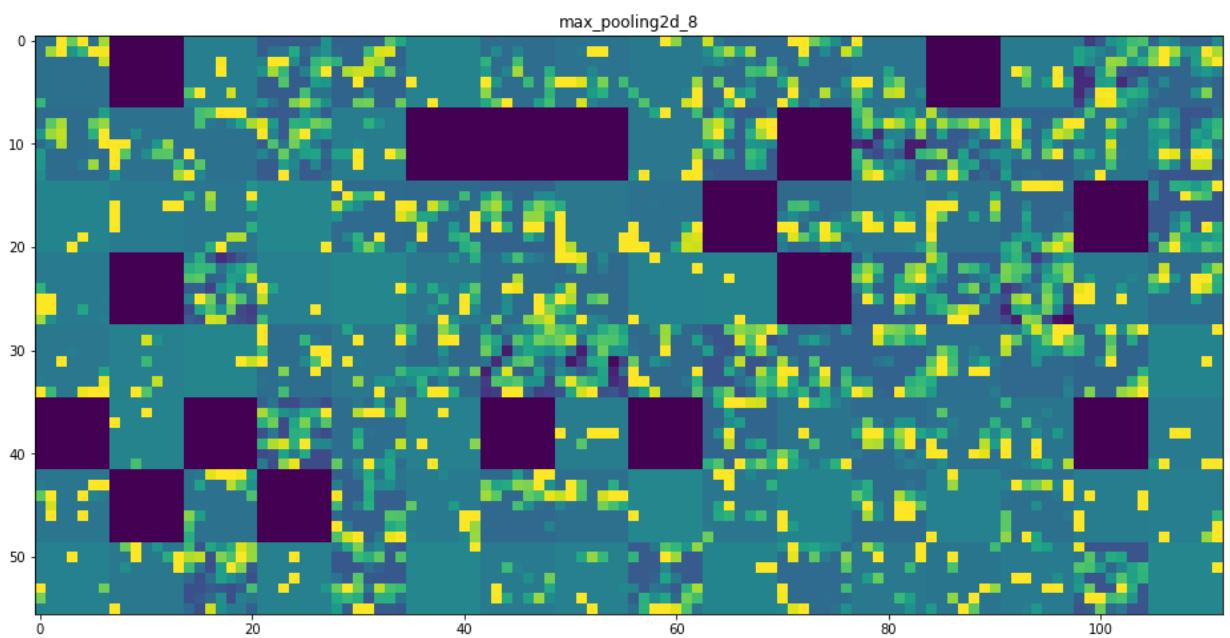
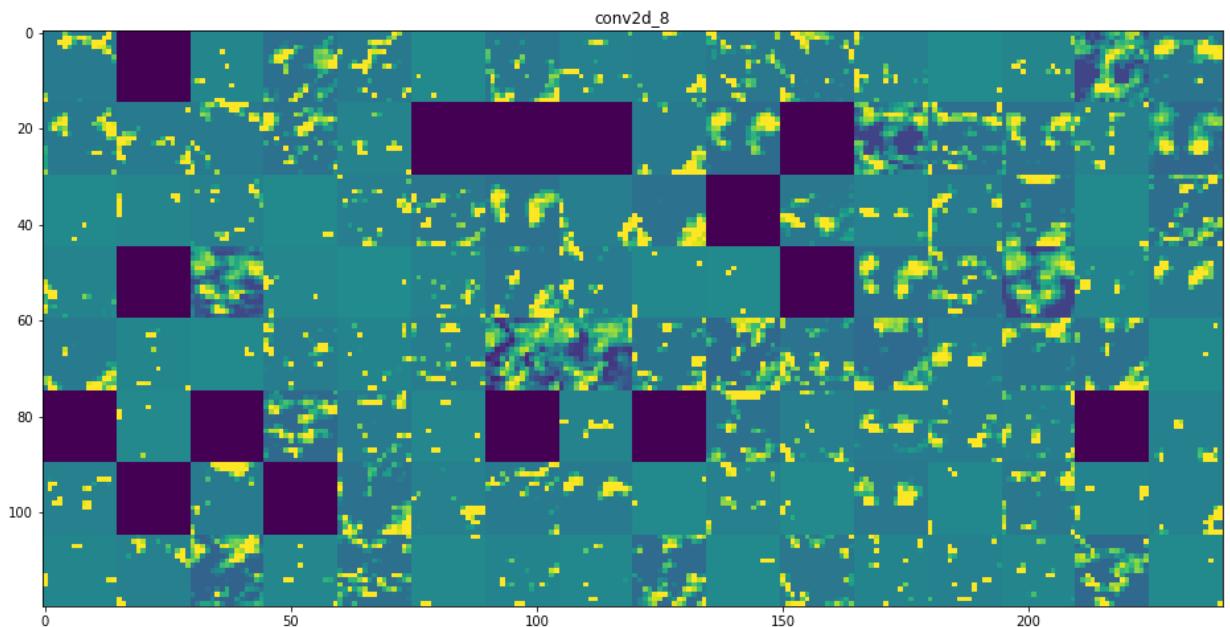
    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

plt.show()
```

/Users/shilinli/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:30: RuntimeWarning: invalid value encountered in true_divide







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.

We have just evidenced a very important universal characteristic of the representations learned by deep neural networks: the features extracted by a layer get increasingly abstract with the depth of the layer. The activations of layers higher-up carry less and less information about the specific input being seen, and more and more information about the target (in our case, the class of the image: cat or dog). A deep neural network effectively acts as an information distillation pipeline, with raw data going in (in our case, RGB 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).

This is analogous to the way humans and animals perceive the world: after observing a scene for a few seconds, a human can remember which abstract objects were present in it (e.g. bicycle, tree) but could not remember the specific appearance of these objects. In fact, if you tried to draw a generic bicycle from mind right now, chances are you could not get it even remotely right, even though you have seen thousands of bicycles in your lifetime. Try it right now: this effect is absolutely real. Your brain has learned to completely abstract its visual input, to transform it into high-level visual concepts while completely filtering out irrelevant visual details, making it tremendously difficult to remember how things around us actually look.

Visualizing Covdnet Filters

Another easy thing to do 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 would be one that the chosen filter is maximally responsive to.

The process is simple: we will build a loss function that maximizes the value of a given filter in a given convolution layer, then we will use stochastic gradient descent to adjust the values of the input image so as to maximize this activation value. For instance, here's a loss for the activation of filter 0 in the layer "block3_conv1" of the VGG16 network, pre-trained on ImageNet:

```
In [13]: 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])
```

To implement gradient descent, we will need the gradient of this loss with respect to the model's input. To do this, we will use the gradients function packaged with the backend module of Keras:

```
In [14]: # The call to `gradients` returns a list of tensors (of size 1 in this
# case)
# hence we only keep the first element -- which is a tensor.
grads = K.gradients(loss, model.input)[0]
```

A non-obvious trick to use for the gradient descent process to go smoothly is to normalize the gradient tensor, by dividing it by its L2 norm (the square root of the average of the square of the values in the tensor). This ensures that the magnitude of the updates done to the input image is always within a same range.

```
In [15]: # We add 1e-5 before dividing so as to avoid accidentally dividing by
# 0.
grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)
```

Now we need a way to compute the value of the loss tensor and the gradient tensor, given an input image. We can define a Keras backend function to do this: iterate is a function that takes a Numpy tensor (as a list of tensors of size 1) and returns a list of two Numpy tensors: the loss value and the gradient value.

```
In [16]: iterate = K.function([model.input], [loss, grads])

# Let's test it:
import numpy as np
loss_value, grads_value = iterate([np.zeros((1, 150, 150, 3))])
```

At this point we can define a Python loop to do stochastic gradient descent:

```
In [17]: # We start from a gray image with some noise
input_img_data = np.random.random((1, 150, 150, 3)) * 20 + 128.

# Run gradient ascent for 40 steps
step = 1. # this is the magnitude of each gradient update
for i in range(40):
    # Compute the loss value and gradient value
    loss_value, grads_value = iterate([input_img_data])
    # Here we adjust the input image in the direction that maximizes the loss
    input_img_data += grads_value * step
```

The resulting image tensor will be a floating point tensor of shape (1, 150, 150, 3), with values that may not be integer within [0, 255]. Hence we would need to post-process this tensor to turn it into a displayable image. We do it with the following straightforward utility function:

```
In [18]: def deprocess_image(x):
    # normalize tensor: center on 0., ensure std is 0.1
    x -= x.mean()
    x /= (x.std() + 1e-5)
    x *= 0.1

    # clip to [0, 1]
    x += 0.5
    x = np.clip(x, 0, 1)

    # convert to RGB array
    x *= 255
    x = np.clip(x, 0, 255).astype('uint8')
    return x
```

Now we have all the pieces, let's put them together into a Python function that takes as input a layer name and a filter index, and that returns a valid image tensor representing the pattern that maximizes the activation the specified filter:

```
In [19]: 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)

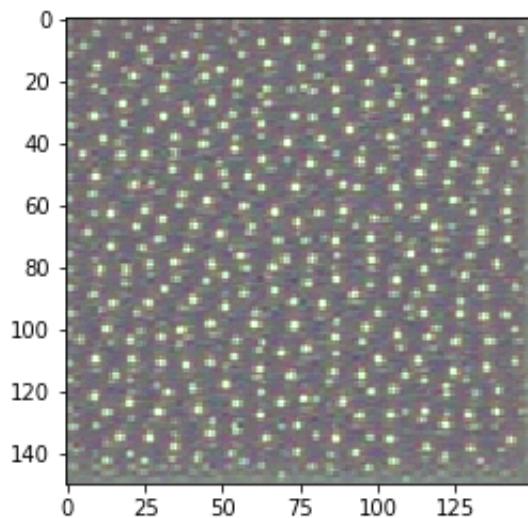
    # This function returns the loss and grads given the input picture
    iterate = K.function([model.input], [loss, grads])

    # 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)
```

```
In [38]: plt.imshow(generate_pattern('block3_conv1', 0))
plt.show()
```



It seems that filter 0 in layer block3_conv1 is responsive to a polka dot pattern.

Now the fun part: we can start visualising every single filter in every layer. For simplicity, we will only look at the first 64 filters in each layer, and will only look at the first layer of each convolution block (block1_conv1, block2_conv1, block3_conv1, block4_conv1, block5_conv1). We will arrange the outputs on a 8x8 grid of 64x64 filter patterns, with some black margins between each filter pattern.

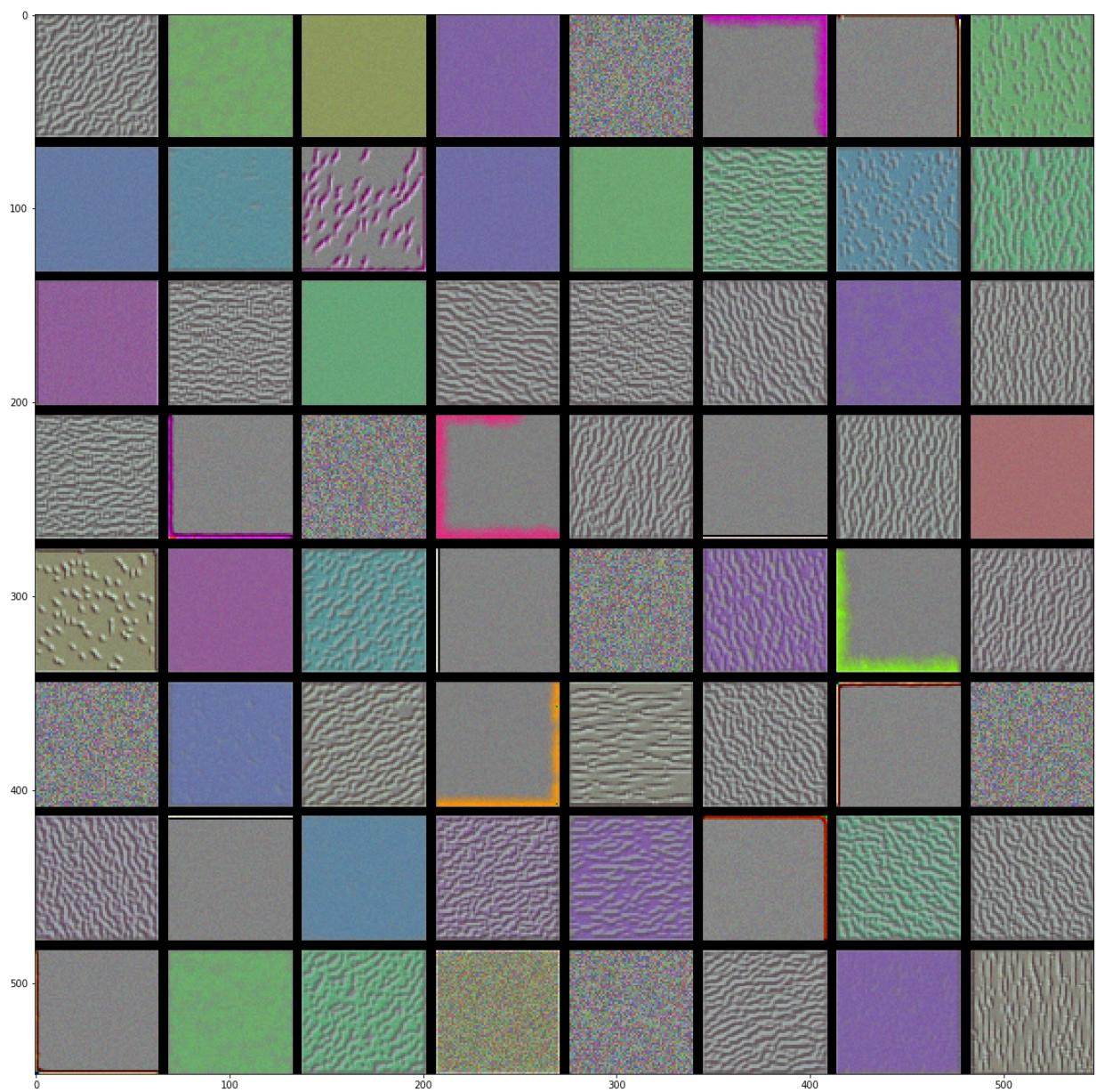
```
In [39]: for layer_name in ['block1_conv1', 'block2_conv1', 'block3_conv1', 'block4_conv1']:
    size = 64
    margin = 5

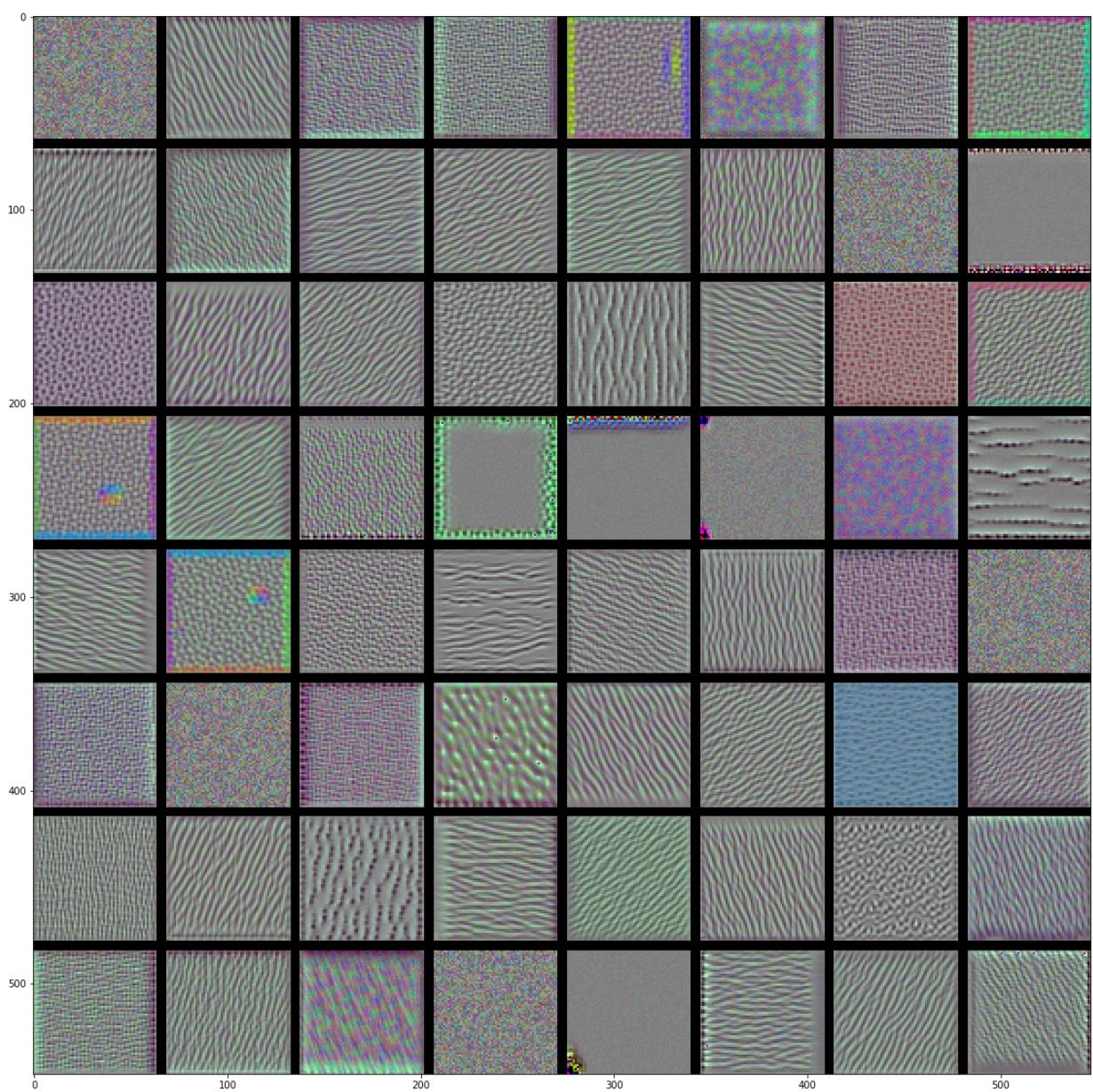
    # This is an empty (black) image where we will store our results.
    results = np.zeros((8 * size + 7 * margin, 8 * size + 7 * margin,
3))

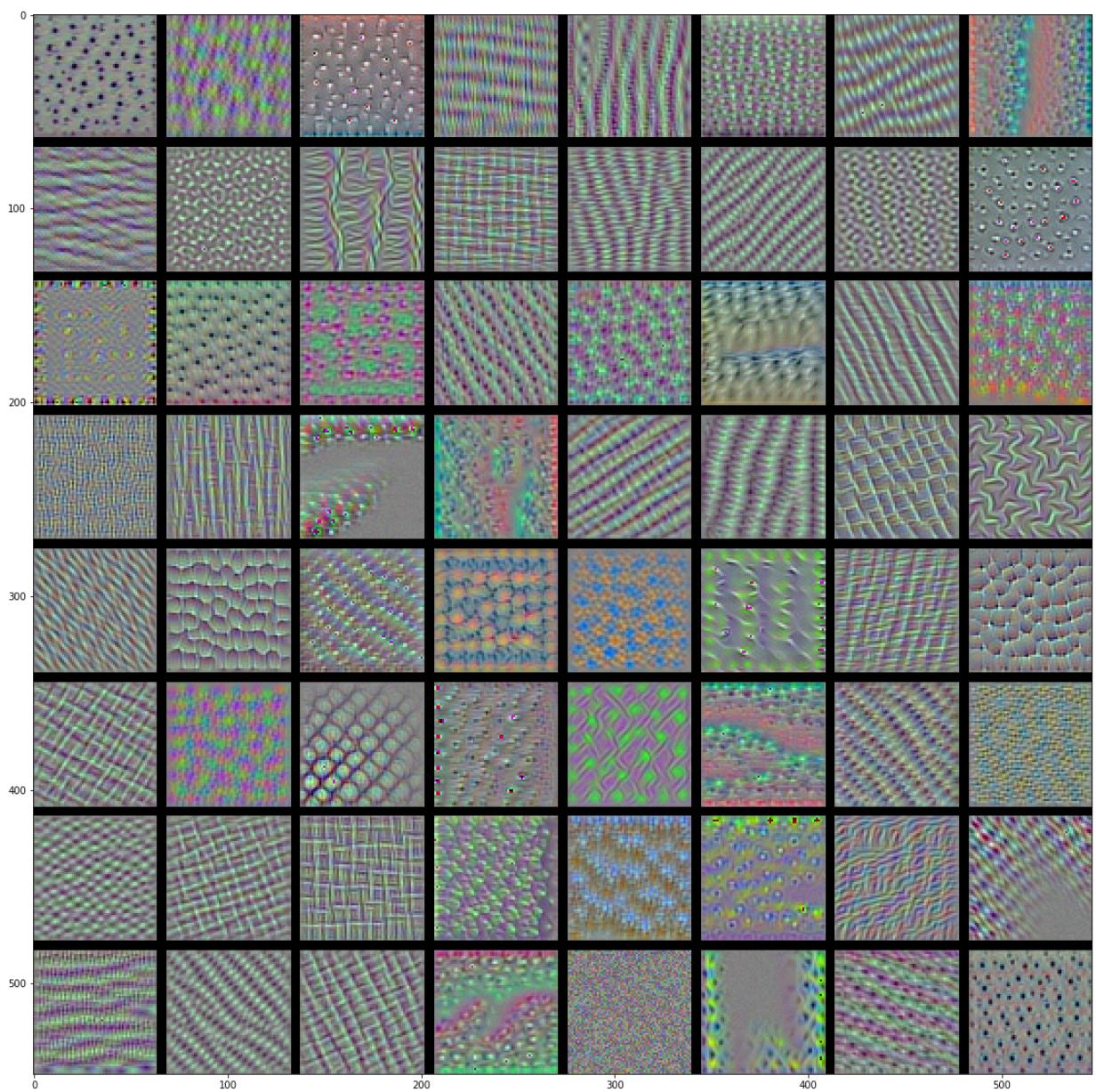
    for i in range(8): # iterate over the rows of our results grid
        for j in range(8): # iterate over the columns of our results
grid
            # Generate the pattern for filter `i + (j * 8)` in `layer_
name`
            filter_img = generate_pattern(layer_name, i + (j * 8), siz
e=size)

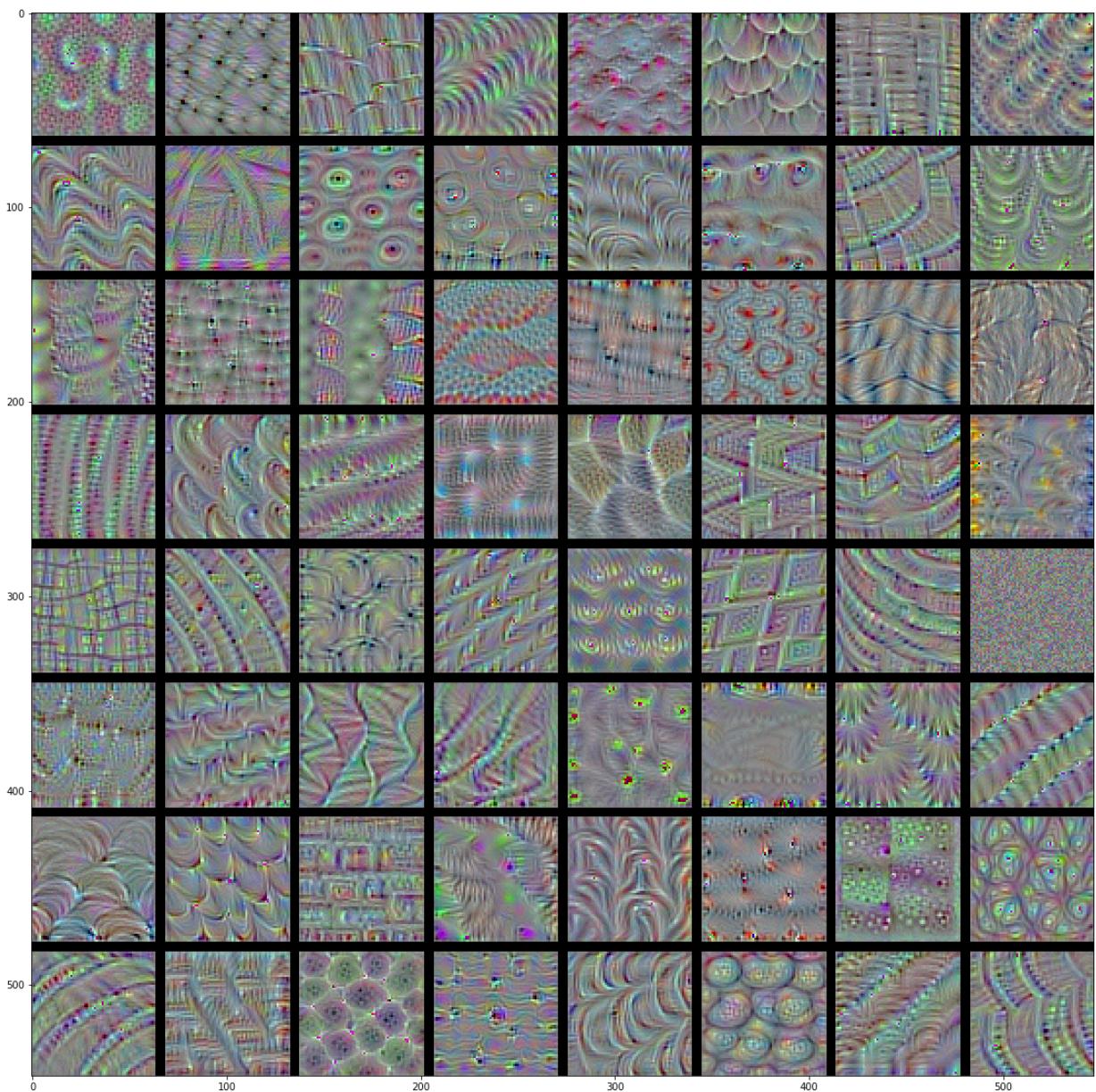
            # Put the result in the square `(i, j)` of the results gri
d
            horizontal_start = i * size + i * margin
            horizontal_end = horizontal_start + size
            vertical_start = j * size + j * margin
            vertical_end = vertical_start + size
            results[horizontal_start: horizontal_end, vertical_start: ver
tical_end, :] = filter_img

    # Display the results grid
    plt.figure(figsize=(20, 20))
    plt.imshow(results)
    plt.show()
```









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.

Visualize heatmaps of class activation

We will introduce one more visualization technique, one that is useful for understanding which parts of a given image led a convnet to its final classification decision. This is helpful for "debugging" the decision process of a convnet, in particular in case of a classification mistake. It also allows you to locate specific objects in an image.

This general category of techniques is called "Class Activation Map" (CAM) visualization, and consists in producing heatmaps of "class activation" over input images. A "class activation" heatmap is a 2D grid of scores associated with a specific output class, computed for every location in any input image, indicating how important each location is with respect to the class considered. For instance, given a image fed into one of our "cat vs. dog" convnet, Class Activation Map visualization allows us to generate a heatmap for the class "cat", indicating how cat-like different parts of the image are, and likewise for the class "dog", indicating how dog-like different parts of the image are.

The specific implementation we will use is the one described in Grad-CAM: Why did you say that? Visual Explanations from Deep Networks via Gradient-based Localization. It is very simple: 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". We will demonstrate this technique using the pre-trained VGG16 network again:

```
In [20]: from keras.applications.vgg16 import VGG16  
  
K.clear_session()  
  
# Note that we are including the densely-connected classifier on top;  
# all previous times, we were discarding it.  
model = VGG16(weights='imagenet')
```

Convert the image into something the VGG16 model can read: the model was trained on images of size 224x224, preprocessed according to a few rules that are packaged in the utility function `keras.applications.vgg16.preprocess_input`. So we need to load the image, resize it to 224x224, convert it to a Numpy float32 tensor, and apply these pre-processing rules.

```
In [21]: 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/shilinli/Downloads/creative_commons_elephant.jpeg'

# `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)
```

```
In [22]: preds = model.predict(x)
print('Predicted:', decode_predictions(preds, top=3)[0])
```

```
Predicted: [('n02504458', 'African_elephant', 0.909421), ('n01871265',
', 'tusker', 0.086182885), ('n02504013', 'Indian_elephant', 0.004354
6157)]
```

The top-3 classes predicted for this image are:

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

Thus our network has recognized our image as containing an undetermined quantity of African elephants. The entry in the prediction vector that was maximally activated is the one corresponding to the "African elephant" class, at index 386:

```
In [23]: np.argmax(preds[0])
```

```
Out[23]: 386
```

To visualize which parts of our image were the most "African elephant"-like, let's set up the Grad-CAM process:

```
In [24]: # This is the "african elephant" entry in the prediction vector
african_elephant_output = model.output[:, 386]

# This 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.out
put[0]])

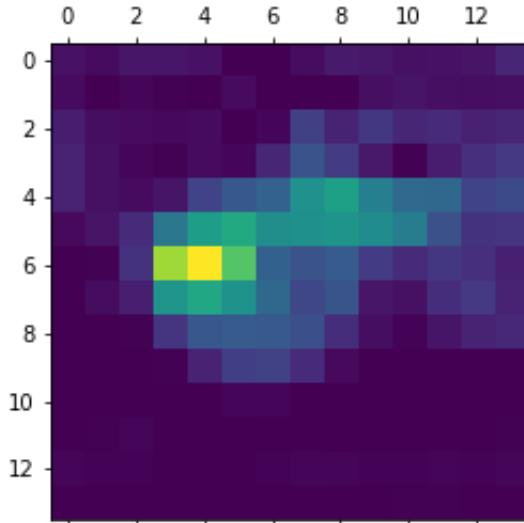
# These are the values of these two quantities, as Numpy arrays,
# given our sample image of two elephants
pooled_grads_value, conv_layer_output_value = iterate([x])

# 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)
```

For visualization purpose, we will also normalize the heatmap between 0 and 1:

```
In [27]: heatmap = np.maximum(heatmap, 0)
heatmap /= np.max(heatmap)
plt.matshow(heatmap)
plt.show()
```



Finally, we will use OpenCV to generate an image that superimposes the original image with the heatmap we just obtained:

```
In [28]: 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/shilinli/Downloads/elephant_cam.jpeg', superimposed_img)
```

Out[28]: True

```
In [3]: from IPython.display import Image  
PATH = "/Users/shilinli/Downloads/"  
Image(filename = PATH + "elephant_cam.jpeg", width=500, height=500)
```

Out[3]:



This visualisation 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?

In particular, it is interesting to note that the ears of the elephant cub are strongly activated: this is probably how the network can tell the difference between African and Indian elephants.