In [1]:

```
import keras
keras.__version__
```

```
Using TensorFlow backend.
```

Out[1]:

```
'2.2.4'
```

In [0]:

```
!git clone https://github.com/jlee1991/Python-Practice.git
```

```
Cloning into 'Python-Practice'...
remote: Enumerating objects: 37479, done.
remote: Counting objects: 100% (37479/37479), done.
remote: Compressing objects: 100% (37462/37462), done.
remote: Total 37517 (delta 21), reused 37473 (delta 15), pack-reused 3
8
Receiving objects: 100% (37517/37517), 868.28 MiB | 39.00 MiB/s, done.
Resolving deltas: 100% (26/26), done.
Checking out files: 100% (41561/41561), done.
```

Visualizing what convnets learn

This notebook contains the code sample found in Chapter 5, Section 4 of [Deep Learning with Python (https://www.manning.com/books/deep-learning-with-python?a_aid=keras&a_bid=76564dff)](https://www.manning.com/books/deep-learning-with-python?a_aid=keras&a_bid=76564dff). Note that the original text features far more content, in particular further explanations and figures: in this notebook, you will only find source code and related comments.

It is often said that deep learning models are "black boxes", learning representations that are difficult to extract and present in a human-readable form. While this is partially true for certain types of deep learning models, it is definitely not true for convnets. The representations learned by convnets are highly amenable to visualization, in large part because they are *representations of visual concepts*. Since 2013, a wide array of techniques have been developed for visualizing and interpreting these representations. We won't survey all of them, but we will cover three of the most accessible and useful ones:

- Visualizing intermediate convnet outputs ("intermediate activations"). This is useful to understand how successive convnet layers transform their input, and to get a first idea of the meaning of individual convnet filters.
- Visualizing convnets filters. This is useful to understand precisely what visual pattern or concept each filter in a convnet is receptive to.
- Visualizing heatmaps of class activation in an image. This is useful to understand which part of an image where identified as belonging to a given class, and thus allows to localize objects in images.

For the first method -- activation visualization -- we will use the small convnet that we trained from scratch on the cat vs. dog classification problem two sections ago. For the next two methods, we will use the VGG16 model that we introduced in the previous section.

Visualizing intermediate activations

Visualizing intermediate activations consists in displaying the feature maps that are output by various convolution and pooling layers in a network, given a certain input (the output of a layer is often called its "activation", the output of the activation function). This gives a view into how an input is decomposed unto the different filters learned by the network. These feature maps we want to visualize have 3 dimensions: width, height, and depth (channels). Each channel encodes relatively independent features, so the proper way to visualize these feature maps is by independently plotting the contents of every channel, as a 2D image. Let's start by loading the model that we saved in section 5.2:

In [2]:

```
from keras.models import load_model

model = load_model('cats_and_dogs_small_2.h5')
model.summary()  # As a reminder.
```

```
WARNING: Logging before flag parsing goes to stderr.
W0715 18:25:57.801321 139661108053888 deprecation_wrapper.py:119] From
```

```
/usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backen
d.py:517: The name tf.placeholder is deprecated. Please use tf.compat.
v1.placeholder instead.

W0715 18:25:57.823891 139661108053888 deprecation_wrapper.py:119] From
/usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backen

d.py:4138: The name tf.random_uniform is deprecated. Please use tf.ran
dom.uniform instead.

W0715 18:25:57.839144 139661108053888 deprecation_wrapper.py:119] From
/usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backen
d.py:3976: The name tf.nn.max_pool is deprecated. Please use tf.nn.max
_pool2d instead.

W0715 18:25:57.887421 139661108053888 deprecation_wrapper.py:119] From
/usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backen
d.py:131: The name tf.get_default_graph is deprecated. Please use tf.c
ompat.v1.get_default_graph instead.

W0715 18:25:57.889040 139661108053888 deprecation_wrapper.py:119] From
/usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backen
d.py:133: The name tf.placeholder_with_default is deprecated. Please u
se tf.compat.v1.placeholder_with_default instead.

W0715 18:25:57.899698 139661108053888 deprecation.py:506] From /usr/lo
cal/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py:34
45: calling dropout (from tensorflow.python.ops.nn_ops) with keep_prob
is deprecated and will be removed in a future version.
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `rate
= 1 - keep_prob`.
W0715 18:25:57.974103 139661108053888 deprecation_wrapper.py:119] From
/usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backen
d.py:174: The name tf.get_default_session is deprecated. Please use tf
.compat.v1.get_default_session instead.

W0715 18:25:58.887538 139661108053888 deprecation_wrapper.py:119] From
/usr/local/lib/python3.6/dist-packages/keras/optimizers.py:790: The na
me tf.train.Optimizer is deprecated. Please use tf.compat.v1.train.Opt
imizer instead.

W0715 18:25:58.901051 139661108053888 deprecation.py:323] From /usr/lo
cal/lib/python3.6/dist-packages/tensorflow/python/ops/nn_impl.py:180:
add_dispatch_support.<locals>.wrapper (from tensorflow.python.ops.arra
y_ops) is deprecated and will be removed in a future version.
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where

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

_________________________________________________________________
```

```
max_pooling2d_5 (MaxPooling2 (None, 74, 74, 32)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 72, 72, 64)        18496
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 36, 36, 64)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 34, 34, 128)       73856
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 17, 17, 128)       0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 15, 15, 128)       147584
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (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
```

This will be the input image we will use -- a picture of a cat, not part of images that the network was trained on:

In [3]:

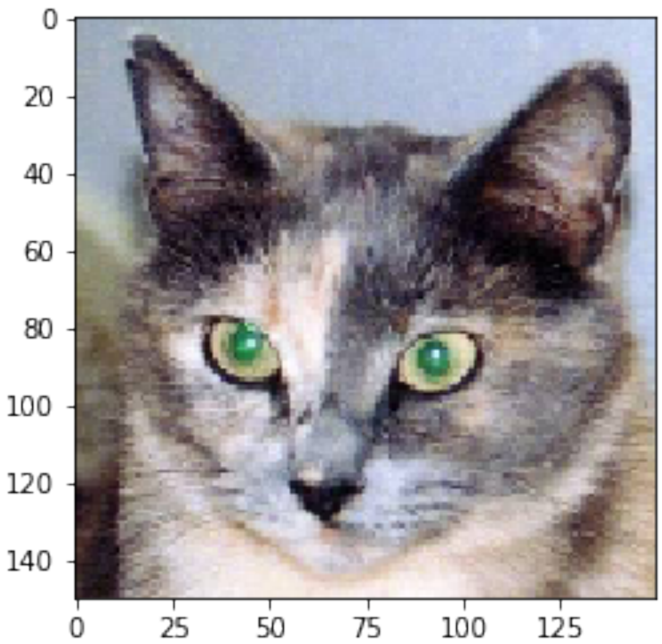
```python
img_path = '/content/Python-Practice/Harvard CSCI E-63/HW3/small/test/cats/cat.1700

# 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)
```

Let's display our picture:

In [4]:

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.imshow(img_tensor[0])
plt.show()
```



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). The resulting class is a Keras model, just like the `Sequential` models that you are familiar with, mapping the specified inputs to the specified outputs. What sets the `Model` class apart is that it allows for models with multiple outputs, unlike `Sequential`. For more information about the `Model` class, see Chapter 7, Section 1.

In [0]:

```

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)
```

When fed an image input, this model returns the values of the layer activations in the original model. This is the first time you encounter a multi-output model in this book: until now the models you 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 [0]:

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

For instance, this is the activation of the first convolution layer for our cat image input:

In [8]:

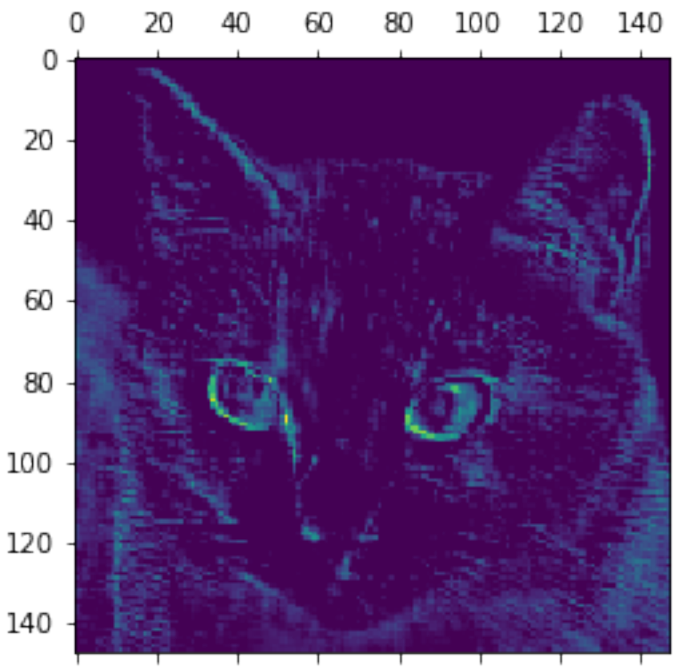
```
first_layer_activation = activations[0]
print(first_layer_activation.shape)
```

```
(1, 148, 148, 32)
```

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

In [0]:

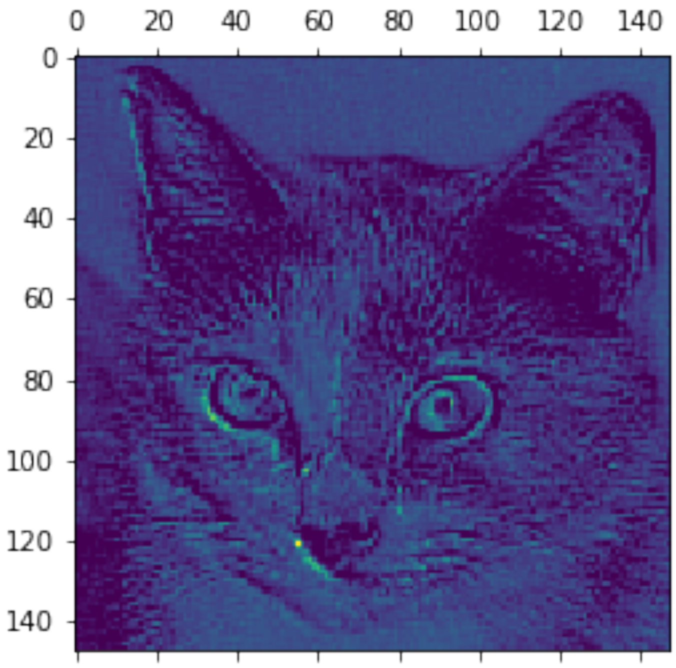
```
import matplotlib.pyplot as plt

plt.matshow(first_layer_activation[0, :, :, 3], cmap='viridis')
plt.show()
```



This channel appears to encode a diagonal edge detector. Let's try the 30th channel -- but note that your own channels may vary, since the specific filters learned by convolution layers are not deterministic.

In [0]:

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



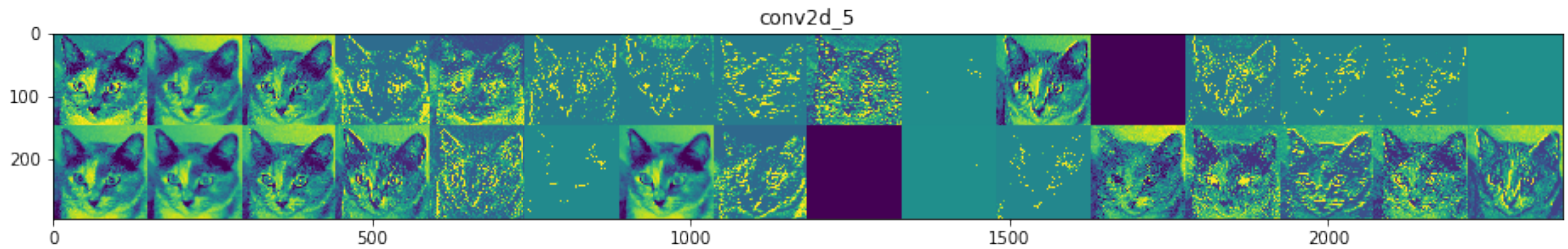
This one looks like a "bright green dot" detector, useful to encode cat eyes. At this point, let's go and plot a complete visualization of all the activations in the network. We'll extract and plot every channel in each of our 8 activation maps, and we will stack the results in one big image tensor,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

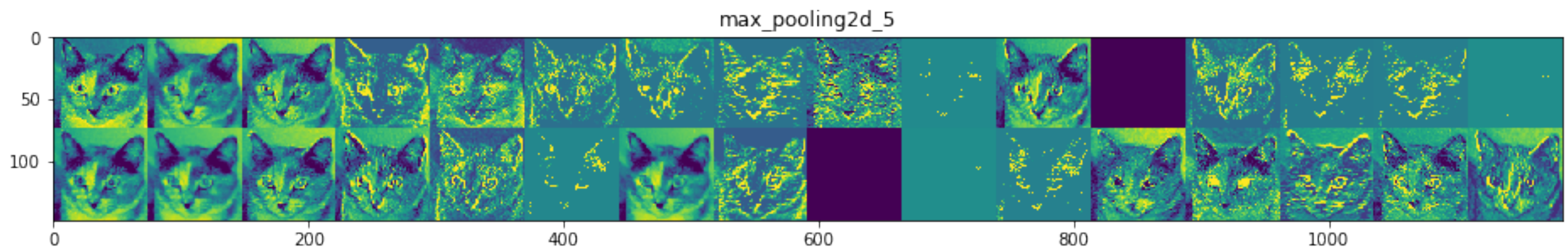
In [0]:

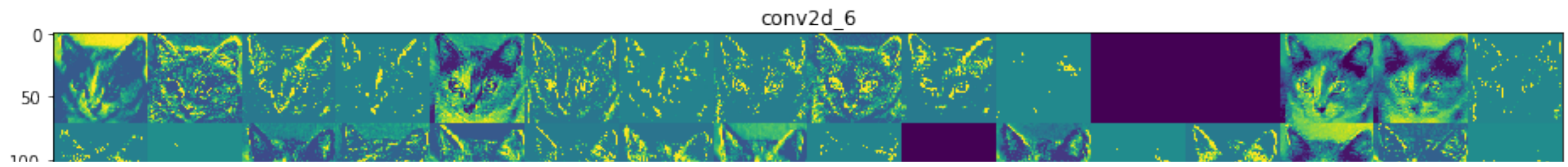
```
import keras

# 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()
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:30: Runti
meWarning: 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, 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).

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. You 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 convnet 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 [0]:

```
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])
```

In [0]:

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_2 (InputLayer)         (None, None, None, 3)     0         
_________________________________________________________________
block1_conv1 (Conv2D)        (None, None, None, 64)    1792      
_________________________________________________________________
block1_conv2 (Conv2D)        (None, None, None, 64)    36928     
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, None, None, 64)    0         
_________________________________________________________________
block2_conv1 (Conv2D)        (None, None, None, 128)   73856     
_________________________________________________________________
block2_conv2 (Conv2D)        (None, None, None, 128)   147584    
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, None, None, 128)   0         
_________________________________________________________________
block3_conv1 (Conv2D)        (None, None, None, 256)   295168    
_________________________________________________________________
block3_conv2 (Conv2D)        (None, None, None, 256)   590080    
_________________________________________________________________
block3_conv3 (Conv2D)        (None, None, None, 256)   590080    
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, None, None, 256)   0         
_________________________________________________________________
block4_conv1 (Conv2D)        (None, None, None, 512)   1180160   
_________________________________________________________________
block4_conv2 (Conv2D)        (None, None, None, 512)   2359808   
_________________________________________________________________
block4_conv3 (Conv2D)        (None, None, None, 512)   2359808   
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, None, None, 512)   0         
_________________________________________________________________
block5_conv1 (Conv2D)        (None, None, None, 512)   2359808   
_________________________________________________________________
block5_conv2 (Conv2D)        (None, None, None, 512)   2359808   
_________________________________________________________________
block5_conv3 (Conv2D)        (None, None, None, 512)   2359808   
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, None, None, 512)   0         
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
```

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 [0]:

```
# 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 [0]:

```
# 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 [0]:

```
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 [0]:

```
# 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 [0]:

```
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:

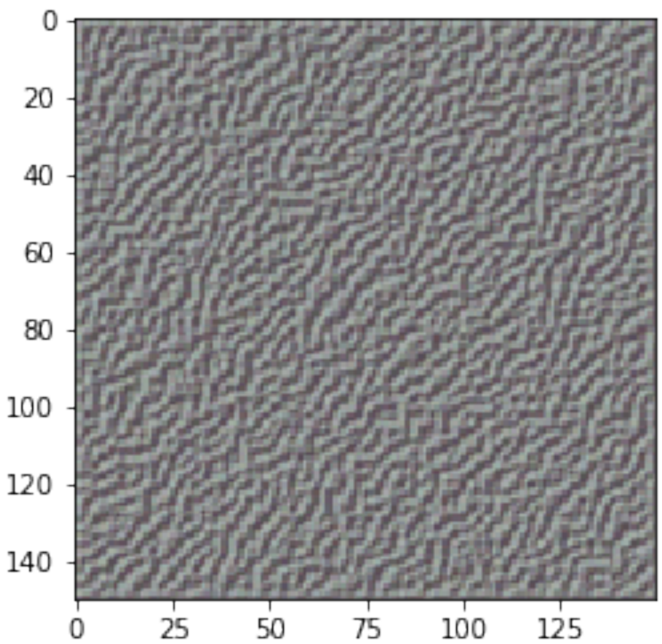
In [0]:

```
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)
```

Let's try this:

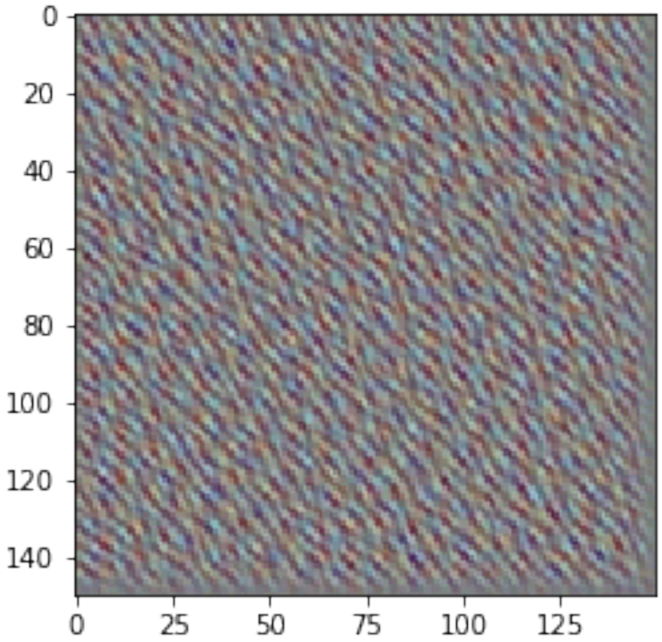
In [0]:

```
plt.imshow(generate_pattern('block1_conv1', 0))
plt.show()
```



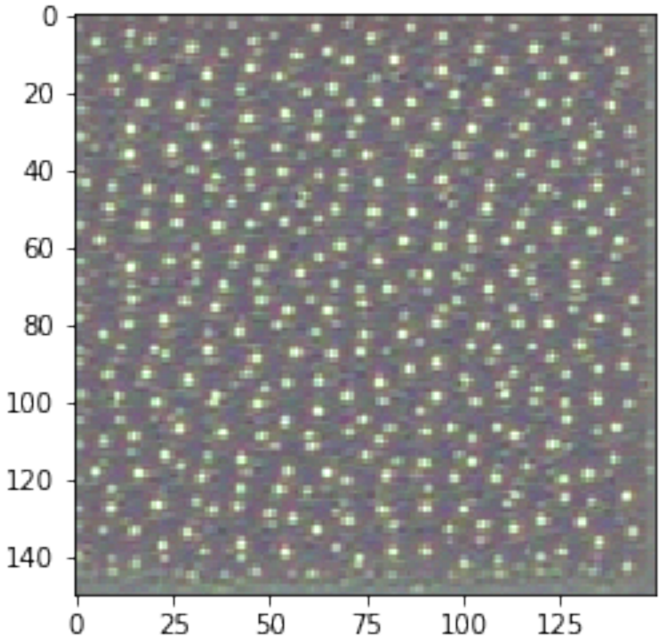
In [0]:

```
plt.imshow(generate_pattern('block3_conv1', 4))
plt.show()
```



In [0]:

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

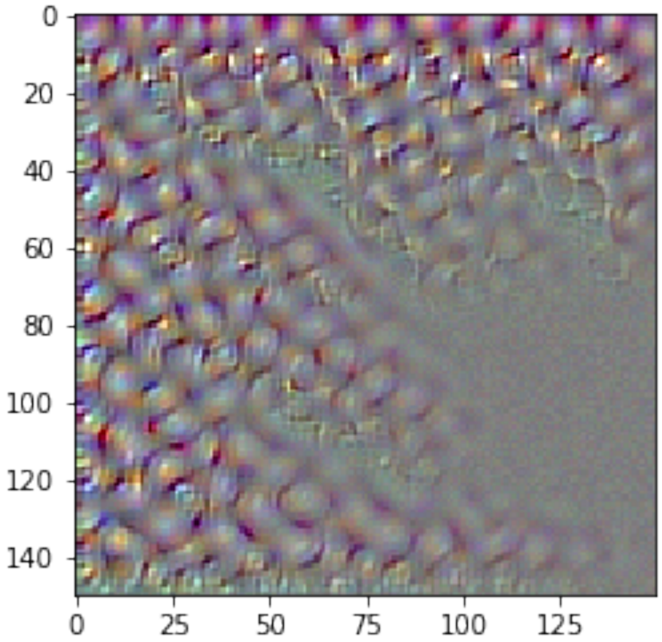


It seems that filter 0 in layer `block3_conv1` is responsive to a polka dot pattern.

Let us visualize a few more filters.

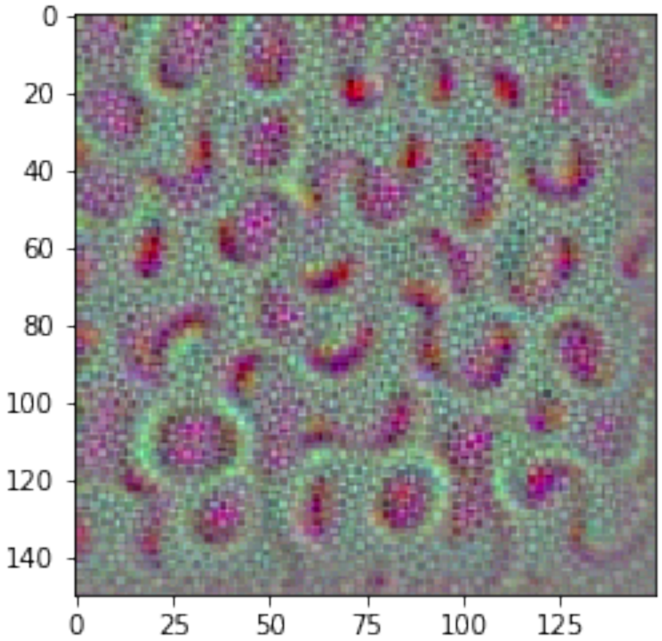
In [0]:

```
plt.imshow(generate_pattern('block3_conv2', 0))
plt.show()
```



In [0]:

```
plt.imshow(generate_pattern('block4_conv1', 0))
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.

Problem 5

Finally, consider Jupyter notebook 5.4-visualizing-what-convnetslearn00.ipynb. This notebook demonstrates how you could capture and display values of the activation (feature) maps of different layers in your network, and how you could capture and display images that pass most directly through filters in different layers. (25%)

1) Fetch an image of a tiger. Crop the image to the same size as images of cats and dogs used in the notebook.
2) Capture feature maps at different convolutions layers.
3) Select different channels (sub-layers) than the ones displayed in the notebook.
4) Continue experimentation and display for us tensors representing patterns that maximize the activation of several filters in different convolutional layers.
5) Again, experiment with channels (sublayers) different than the ones presented in the notebook. Submit a copy of the working notebook and its PDF image

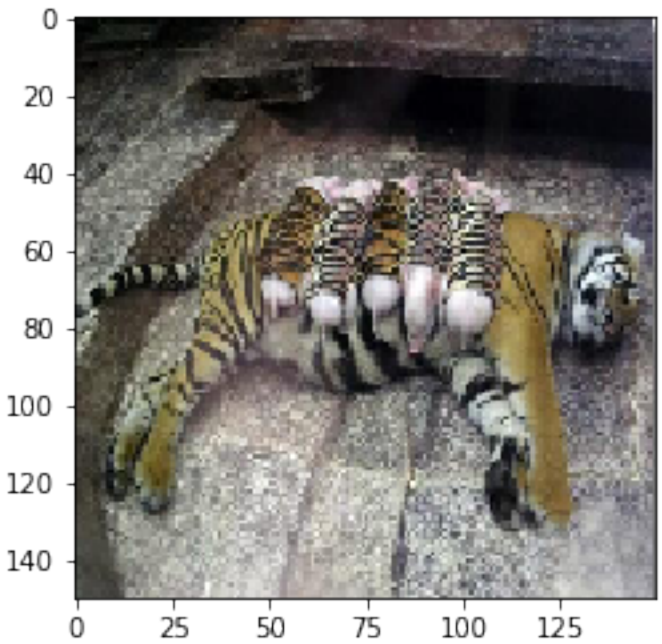
In [9]:

```python
img_path = 'tiger.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_tensor2 = image.img_to_array(img)
img_tensor2 = np.expand_dims(img_tensor2, axis=0)

# Remember that the model was trained on inputs
# that were preprocessed in the following way:
img_tensor2 /= 255.

# Its shape is (1, 150, 150, 3)
print(img_tensor2.shape)
```

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

In [10]:

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.imshow(img_tensor2[0])
plt.show()
```



In [0]:

```
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_model2 = models.Model(inputs=model.input, outputs=layer_outputs)
```

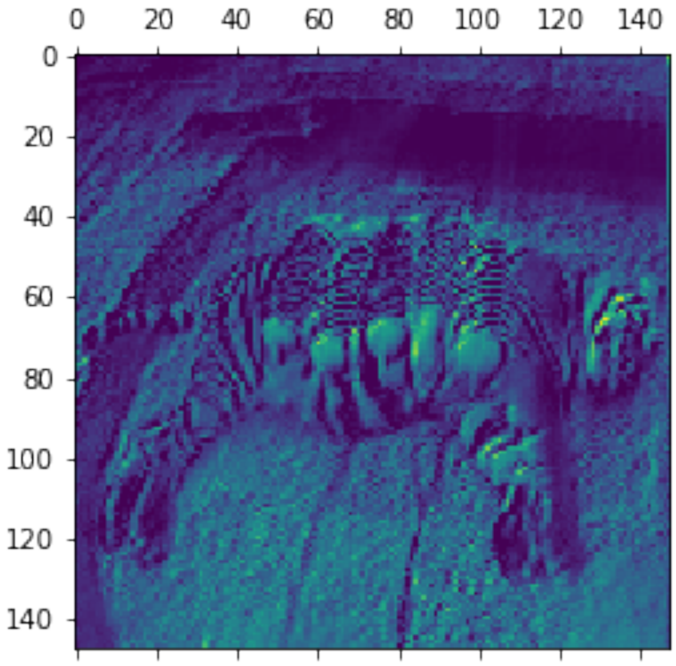
In [12]:

```
# This will return a list of 5 Numpy arrays:
# one array per layer activation
activations2 = activation_model2.predict(img_tensor2)

first_layer_activation = activations2[0]
print(first_layer_activation.shape)
```

```
(1, 148, 148, 32)
```

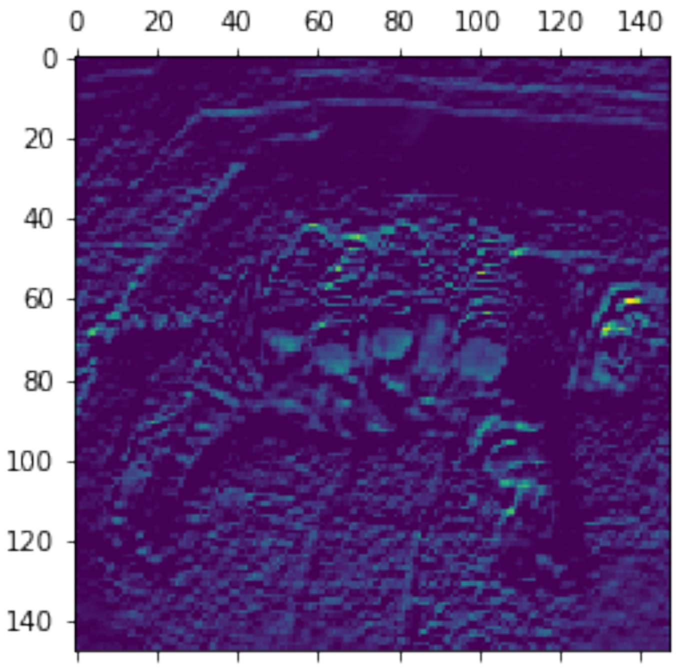
In [16]:

```
import matplotlib.pyplot as plt

plt.matshow(first_layer_activation[0, :, :, 2], cmap='viridis')
plt.show()
```



In [15]:

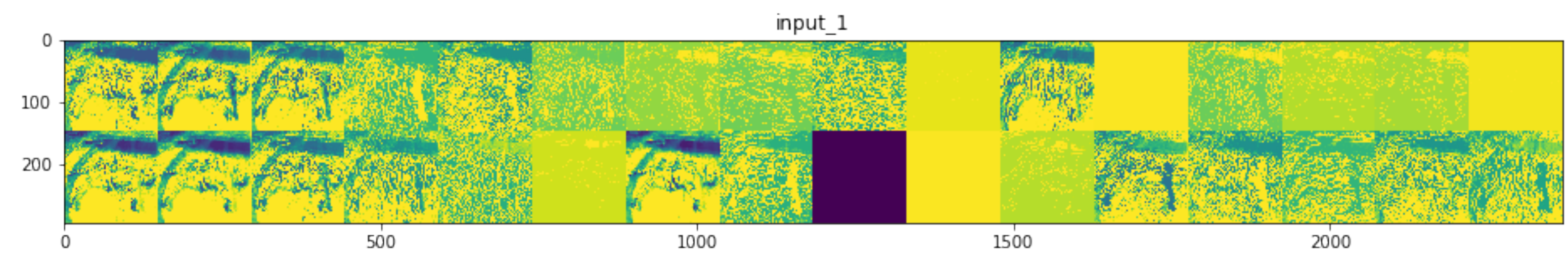
```
plt.matshow(first_layer_activation[0, :, :, 29], cmap='viridis')
plt.show()
```

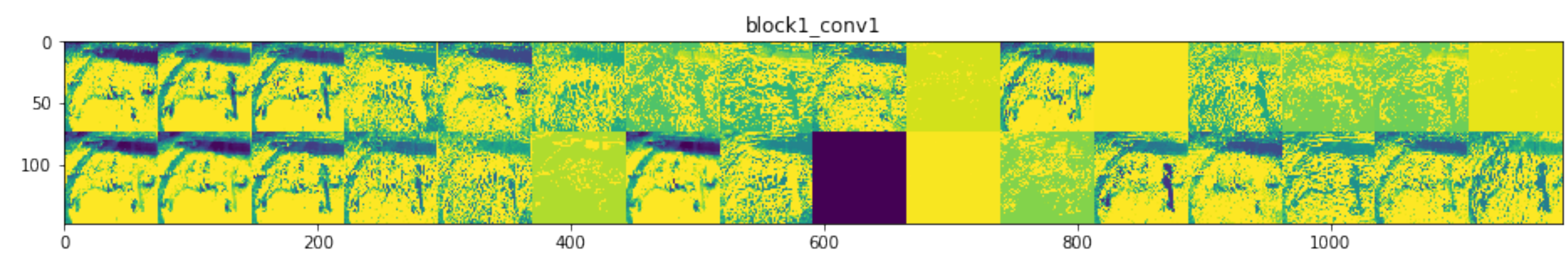


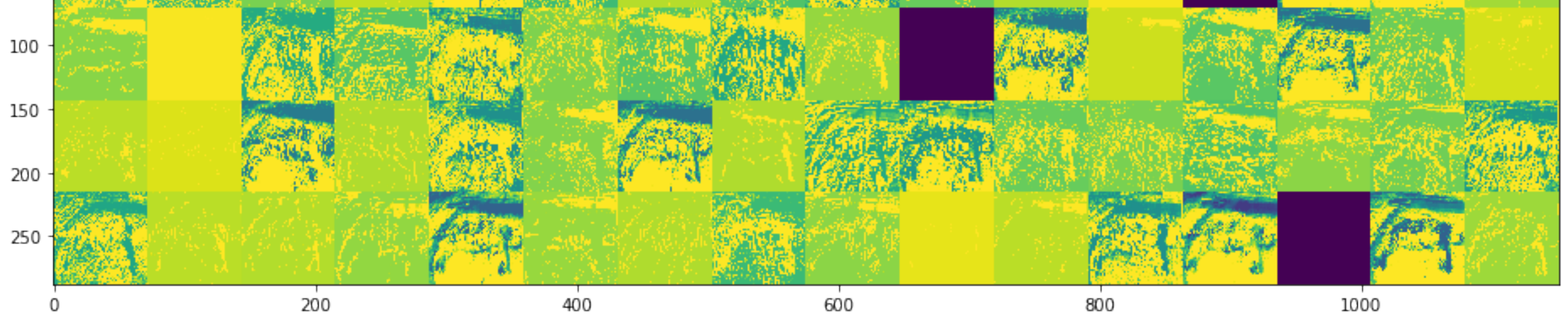
In [20]:

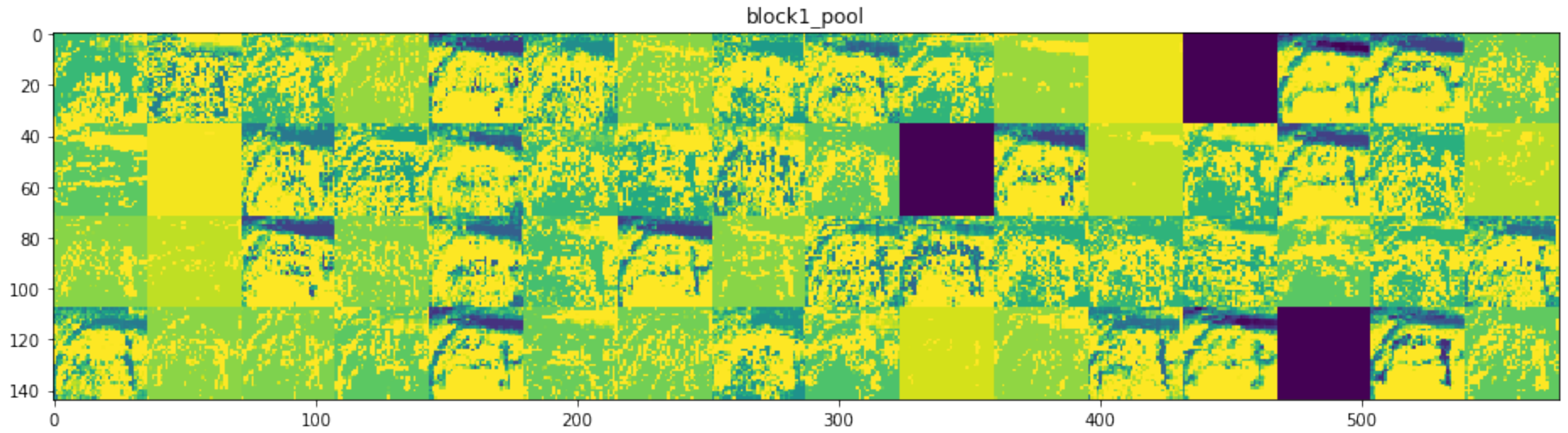
```
import keras

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:4]:
    layer_names.append(layer.name)

images_per_row = 16
```

```
images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations2):
    # 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 *= 128
            channel_image += 256
            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()
```









In [1]:

```
Using TensorFlow backend.
```

Out[1]:

```
'2.2.4'
```

In [0]:

```
Cloning into 'Python-Practice'...
remote: Enumerating objects: 37479, done.
remote: Counting objects: 100% (37479/37479), done.
remote: Compressing objects: 100% (37462/37462), done.
remote: Total 37517 (delta 21), reused 37473 (delta 15), pack-reused 3
8
Receiving objects: 100% (37517/37517), 868.28 MiB | 39.00 MiB/s, done.
Resolving deltas: 100% (26/26), done.
Checking out files: 100% (41561/41561), done.
```

Visualizing what convnets learn

This notebook contains the code sample found in Chapter 5, Section 4 of [Deep Learning with Python (https://www.manning.com/books/deep-learning-with-python?a_aid=keras&a_bid=76564dff)](https://www.manning.com/books/deep-learning-with-python?a_aid=keras&a_bid=76564dff). Note that the original text features far more content, in particular further explanations and figures: in this notebook, you will only find source code and related comments.

It is often said that deep learning models are "black boxes", learning representations that are difficult to extract and present in a human-readable form. While this is partially true for certain types of deep learning models, it is definitely not true for convnets. The representations learned by convnets are highly amenable to visualization, in large part because they are *representations of visual concepts*. Since 2013, a wide array of techniques have been developed for visualizing and interpreting these representations. We won't survey all of them, but we will cover three of the most accessible and useful ones:

- Visualizing intermediate convnet outputs ("intermediate activations"). This is useful to understand how successive convnet layers transform their input, and to get a first idea of the meaning of individual convnet filters.
- Visualizing convnets filters. This is useful to understand precisely what visual pattern or concept each filter in a convnet is receptive to.
- Visualizing heatmaps of class activation in an image. This is useful to understand which part of an image where identified as belonging to a given class, and thus allows to localize objects in images.

For the first method -- activation visualization -- we will use the small convnet that we trained from scratch on the cat vs. dog classification problem two sections ago. For the next two methods, we will use the VGG16 model that we introduced in the previous section.

Visualizing intermediate activations

Visualizing intermediate activations consists in displaying the feature maps that are output by various convolution and pooling layers in a network, given a certain input (the output of a layer is often called its "activation", the output of the activation function). This gives a view into how an input is decomposed unto the different filters learned by the network. These feature maps we want to visualize have 3 dimensions: width, height, and depth (channels). Each channel encodes relatively independent features, so the proper way to visualize these feature maps is by independently plotting the contents of every channel, as a 2D image. Let's start by loading the model that we saved in section 5.2:

In [2]:

```
WARNING: Logging before flag parsing goes to stderr.
W0715 18:25:57.801321 139661108053888 deprecation_wrapper.py:119] From
/usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backen
d.py:517: The name tf.placeholder is deprecated. Please use tf.compat.
v1.placeholder instead.

W0715 18:25:57.823891 139661108053888 deprecation_wrapper.py:119] From
/usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backen
d.py:4138: The name tf.random_uniform is deprecated. Please use tf.ran
dom.uniform instead.

W0715 18:25:57.839144 139661108053888 deprecation_wrapper.py:119] From
/usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backen
d.py:3976: The name tf.nn.max_pool is deprecated. Please use tf.nn.max
_pool2d instead.

W0715 18:25:57.887421 139661108053888 deprecation_wrapper.py:119] From
/usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backen
d.py:131: The name tf.get_default_graph is deprecated. Please use tf.c
```

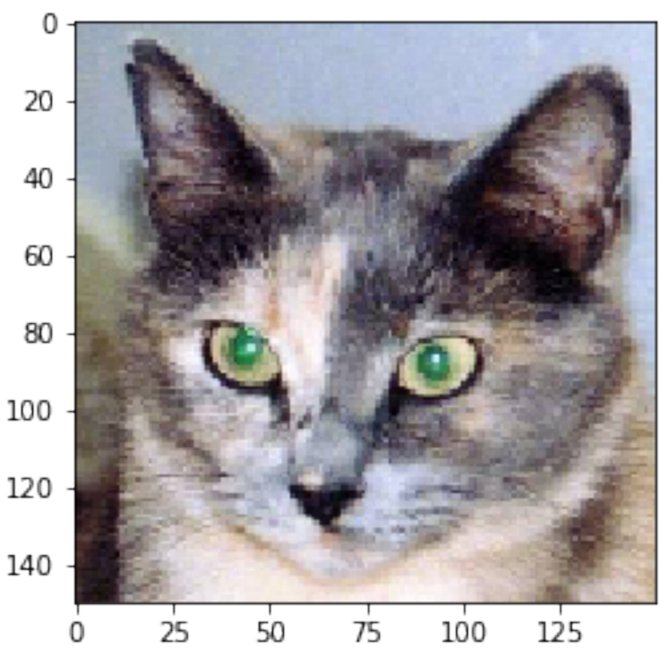
This will be the input image we will use -- a picture of a cat, not part of images that the network was trained on:

In [3]:

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

Let's display our picture:

In [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). The resulting class is a Keras model, just like the `Sequential` models that you are familiar with, mapping the specified inputs to the specified outputs. What sets the `Model` class apart is that it allows for models with multiple outputs, unlike `Sequential`. For more information about the `Model` class, see Chapter 7, Section 1.

In [0]:

```
```

When fed an image input, this model returns the values of the layer activations in the original model. This is the first time you encounter a multi-output model in this book: until now the models you 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 [0]:

```
```

For instance, this is the activation of the first convolution layer for our cat image input:

In [8]:

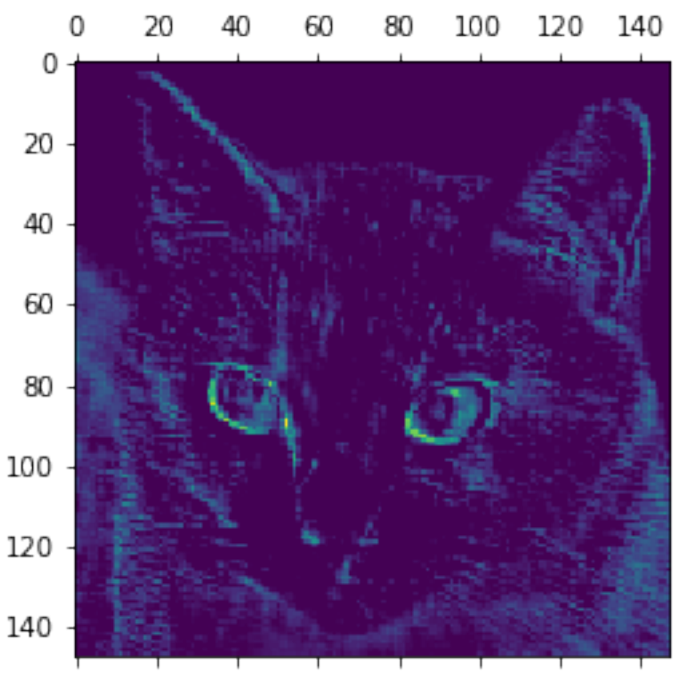
```
```

```
(1, 148, 148, 32)
```

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

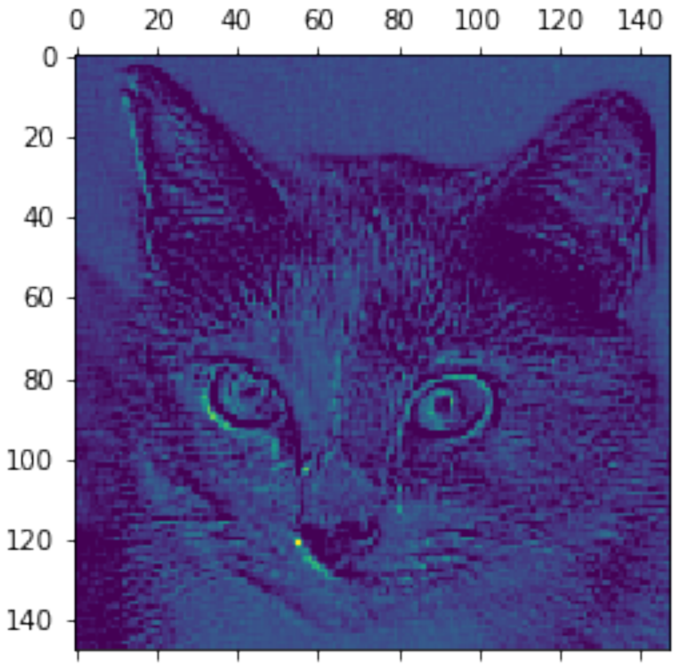
In [0]:

```
```



This channel appears to encode a diagonal edge detector. Let's try the 30th channel -- but note that your own channels may vary, since the specific filters learned by convolution layers are not deterministic.

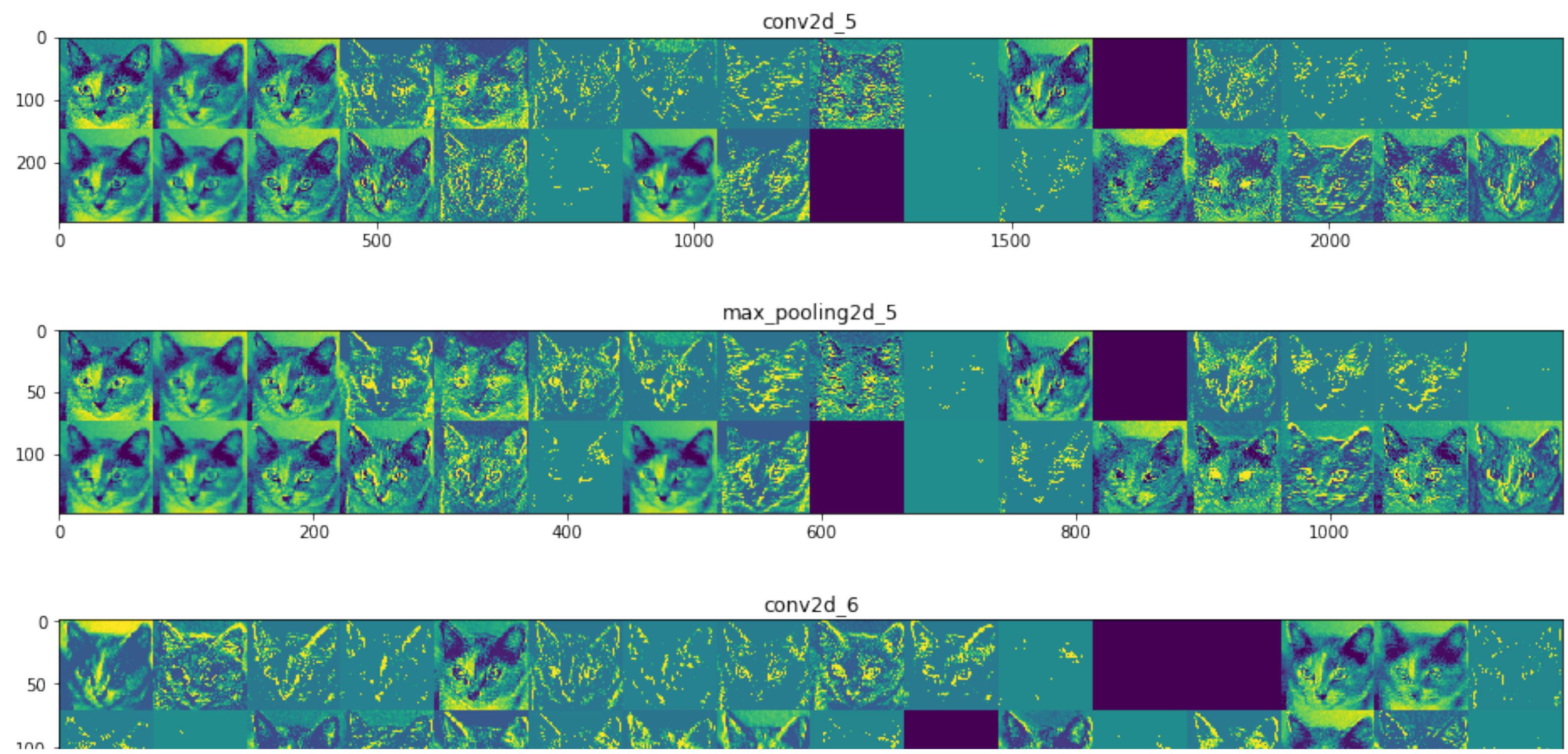
In [0]:



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 [0]:

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:30: Runti
meWarning: 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, 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).

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. You 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 convnet 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 [0]:

```
```

In [0]:

```
```

```
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, None, None, 3)     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, None, None, 64)    1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, None, None, 64)    36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, None, None, 64)    0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, None, None, 128)   73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, None, None, 128)   147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, None, None, 128)   0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, None, None, 256)   295168
_________________________________________________________________
```

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 [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 [0]:

```
```

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 [0]:

```
```

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

In [0]:

```
```

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 [0]:

```
```

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:

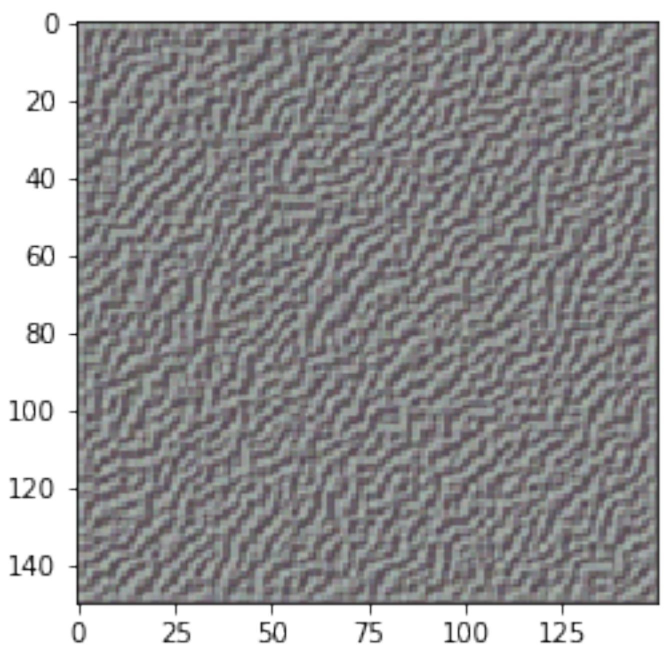
In [0]:

```
```

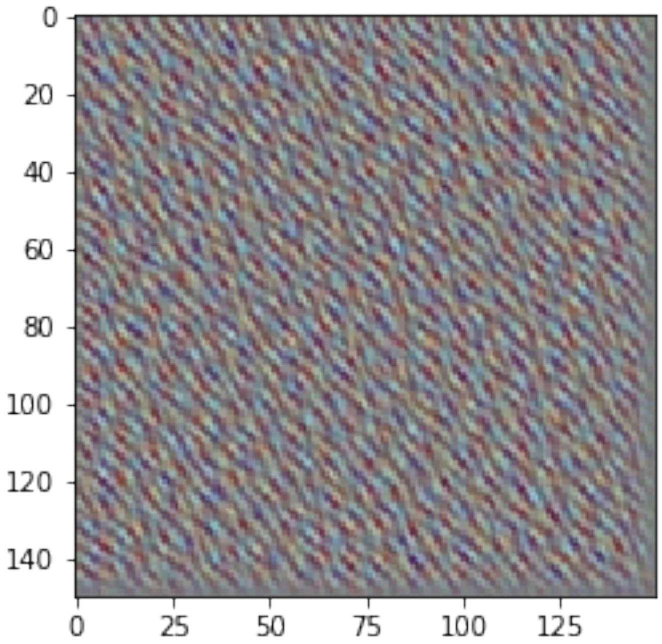
Let's try this:

In [0]:

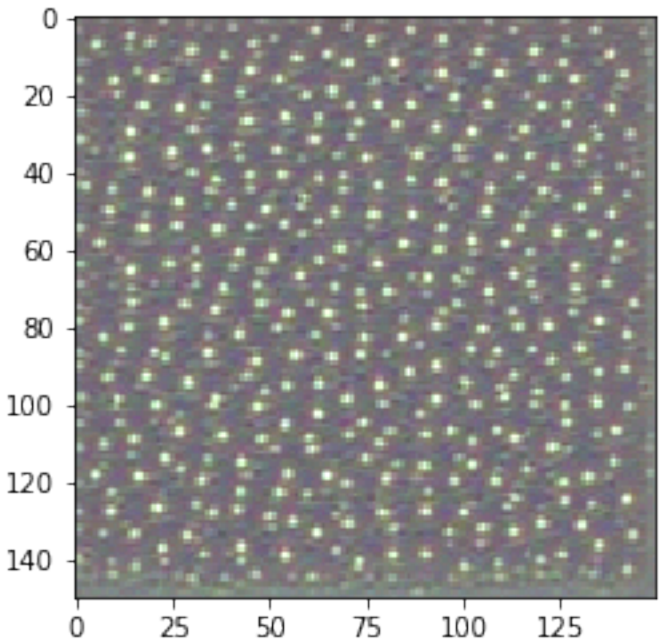
```
```



In [0]:



In [0]:

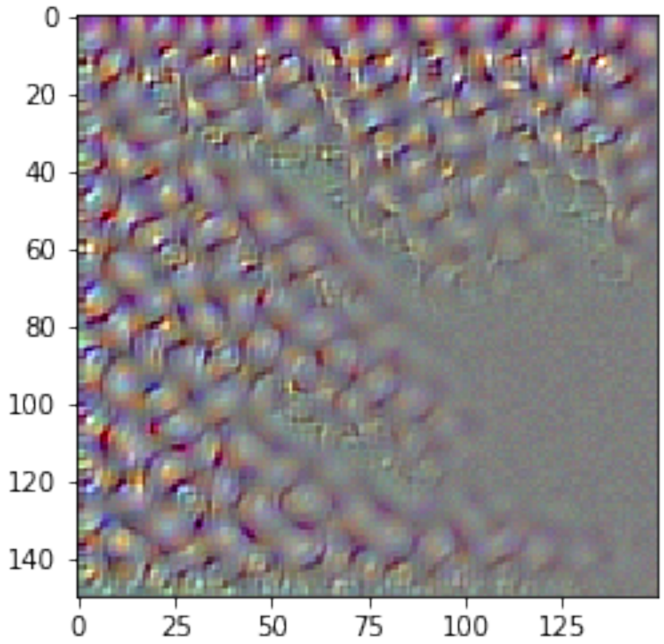


It seems that filter 0 in layer `block3_conv1` is responsive to a polka dot pattern.

Let us visualize a few more filters.

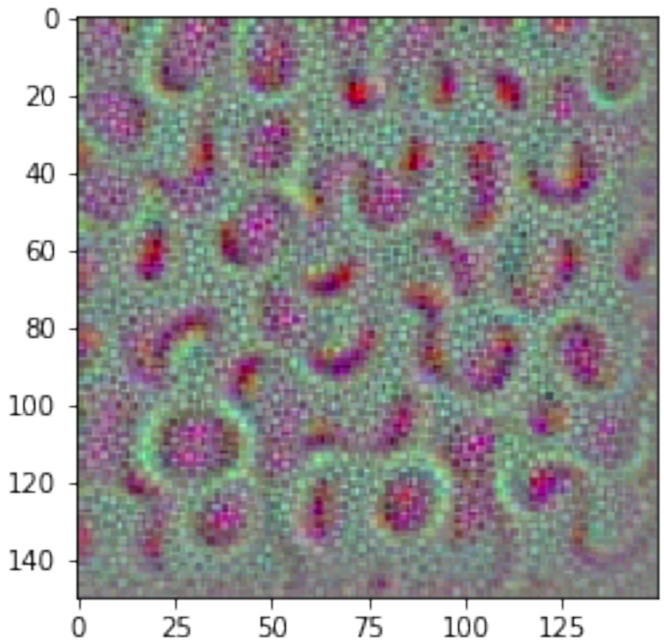
In [0]:

```

```



In [0]:

```

```



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.

Problem 5

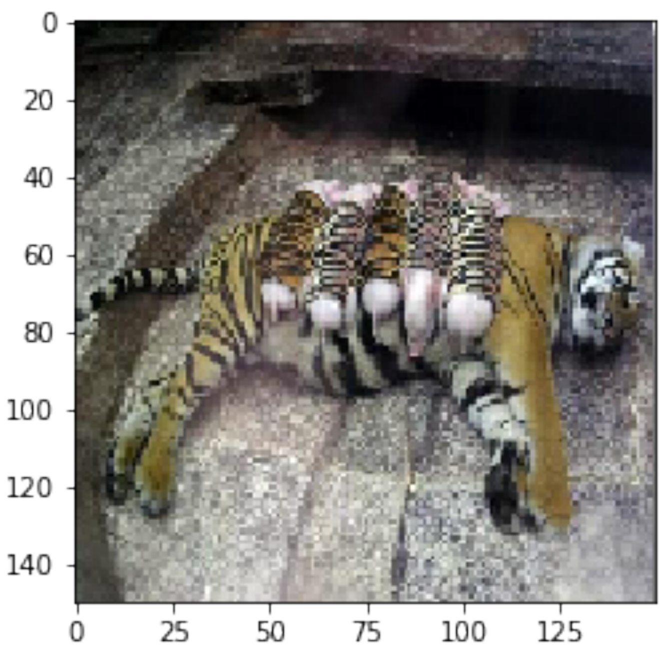
Finally, consider Jupyter notebook 5.4-visualizing-what-convnetslearn00.ipynb. This notebook demonstrates how you could capture and display values of the activation (feature) maps of different layers in your network, and how you could capture and display images that pass most directly through filters in different layers. (25%)

1) Fetch an image of a tiger. Crop the image to the same size as images of cats and dogs used in the notebook.
2) Capture feature maps at different convolutions layers.
3) Select different channels (sub-layers) than the ones displayed in the notebook.
4) Continue experimentation and display for us tensors representing patterns that maximize the activation of several filters in different convolutional layers.
5) Again, experiment with channels (sublayers) different than the ones presented in the notebook. Submit a copy of the working notebook and its PDF image

In [9]:

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

In [10]:

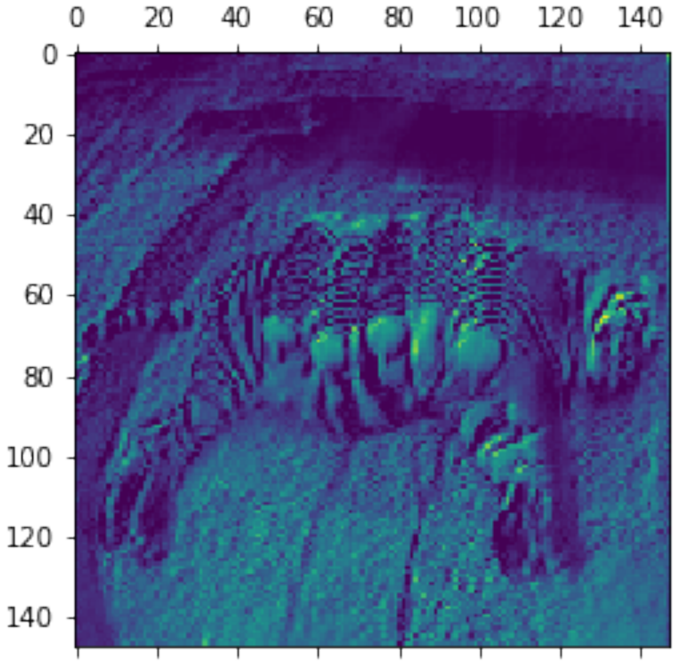


In [0]:

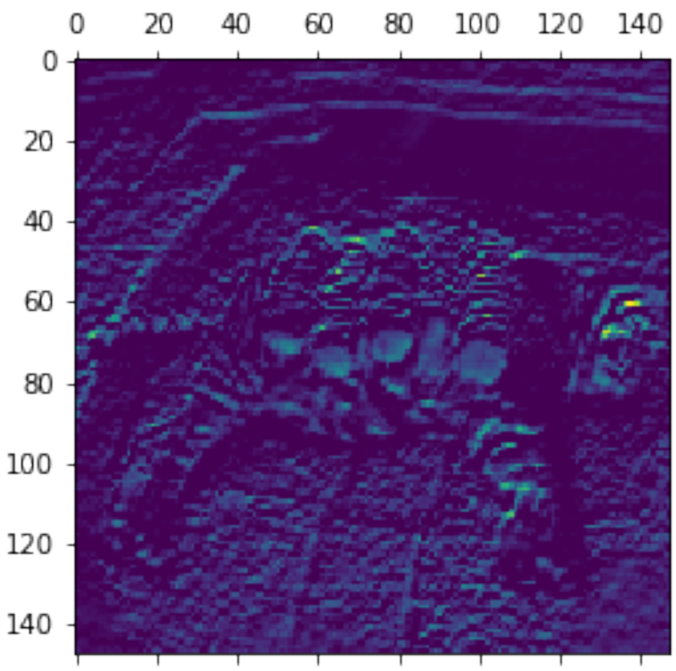
In [12]:

```
(1, 148, 148, 32)
```

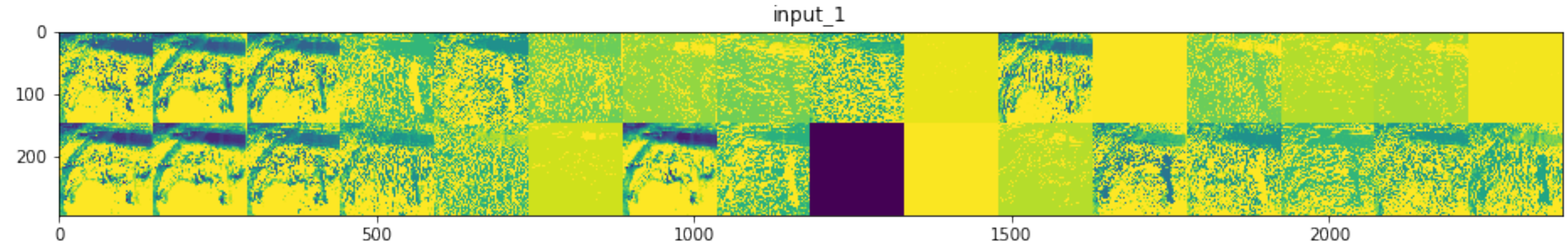
In [16]:

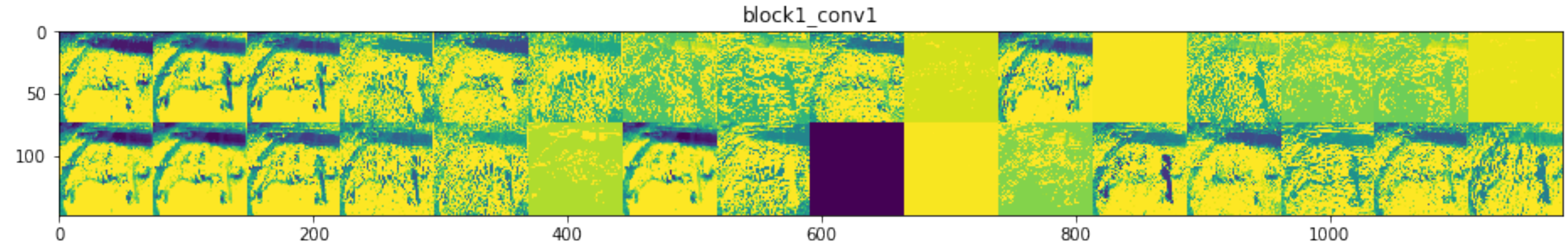


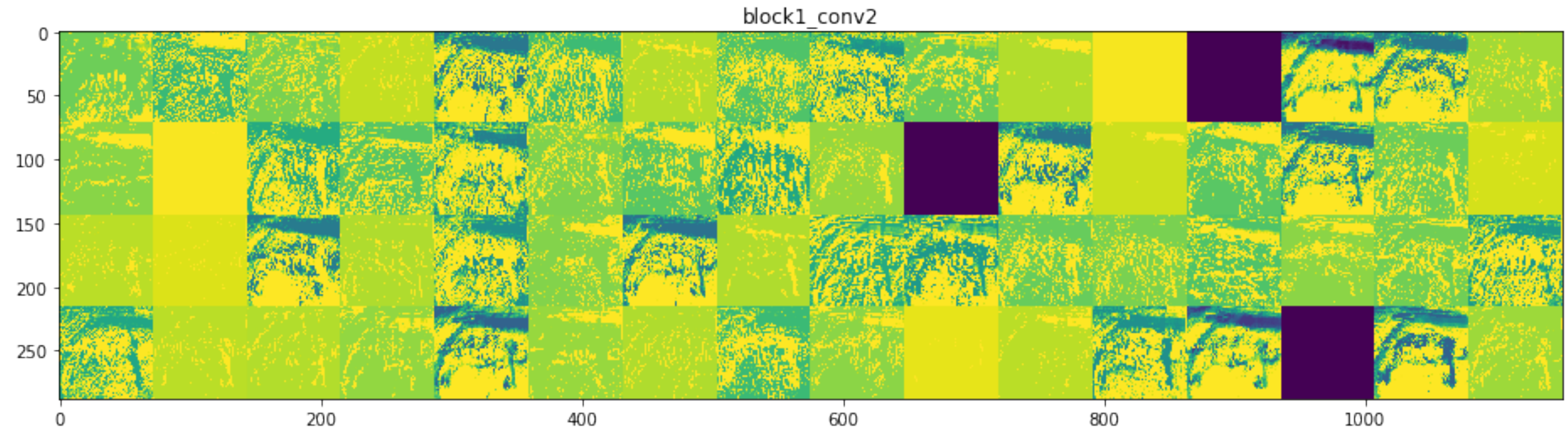
In [15]:

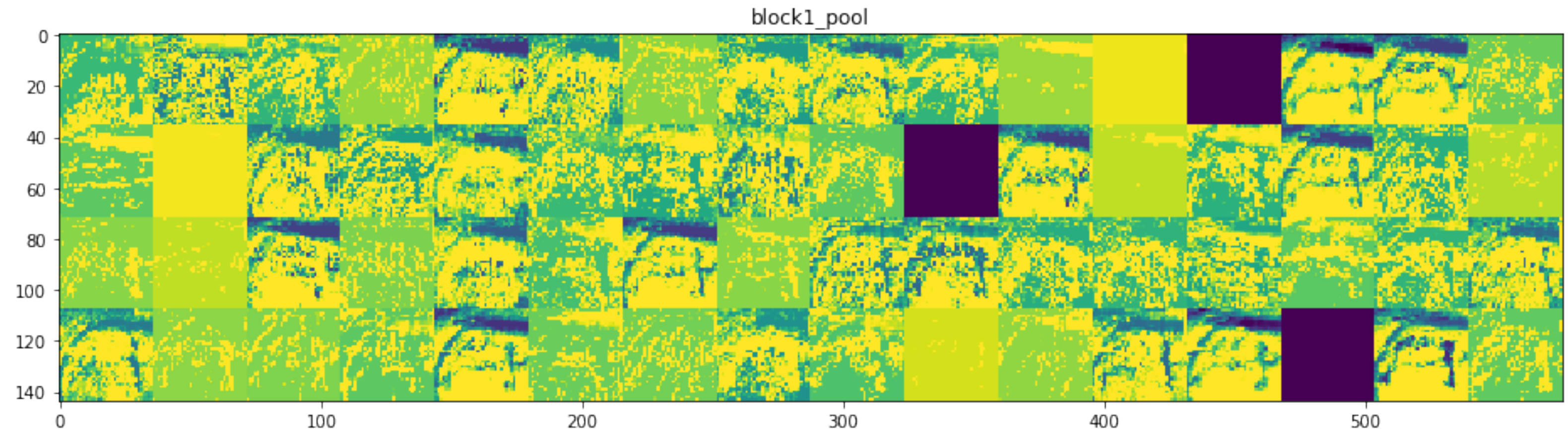


In [20]:


input_1
0
100
200
0
500
1000
1500
2000


block1_conv1
0
50
100
0
200
400
600
800
1000


block1_conv2
0
50
100
150
200
250
0
200
400
600
800
1000


block1_pool
0
20
40
60
80
100
120
140
0
100
200
300
400
500