Feature Visualization by Optimization

Fall 2018 GR5242 Final Project
GR 001 MW 16:10-17:25
Jiayi Dong (jd3416), Shilin Li (sl4261), Hengyang Lin(hl3116), Wenting Zhu (wz2401)
For More details, please check our Github Repo: https://github.com/lishilin63/CNN-Feature-Visualization (https://github.com/lishilin63/CNN-Feature-Visualization)

Project Description
We choose this project because neural networks as black box function approximators, most people are not sure how each layers are doing their jobs. Therefore, We want to explore what and how exactly different layers are extracting features. In this way, we will have a better understanding of different layers in CNN.

The project has 3 parts. The 1st part we build a straightforward 2-layer CNN and train the model by using mnist dataset. We then visualize the input image which could activate a particular unit from the final dense layer. Then we build a more complicated CNN for training cats_dogs dataset. By controling parameters, we visualize the optimized input image that activate filters for each layer, and compare the result. Also, we used the well-known VGG16 model to visualize the optimized input image and compare each layers learning characterisitcs. The 2nd part we are focusing on how a CNN learns a cat. Using the same model we have for cats_dogs dataset, we feed the network a cat picture and visualize what features the filter receives in each layer. This is another way we understand how CNN is doing classifications. The third part is an additional CNN extension of saliency maps, which are used to observe the important part of an imput picture that the CNN makes classification decision on. The model we use is ResNet50 with Grad-CAM method.

Our starting point
When applying CNN in deep learning, people can generate the model by tuning parameters in each layer, but it is hard to interprete the function or ability of each layer by human beings. Inspired by the paper written by Olah, et al. in 2017 on Distill (https://distill.pub/2017/feature-visualization/)[1], we read more articles on Feature Visualization, especially on activation maximization as optimization method. Instead of simply visualize each layer, optimization fixes parameters' in trained model and try to generate the activation maximized input image to attain the best output.

We first use the Mnist model to perform the feature visualization and then apply the same method to cat_dog_classification, which is a more complicated model but on a small scale of data, and finally move to VGG 16 model, which is a well-known trained model on a large scale of data.

The main challenge with the project
The algorithm for the Activation Maximization that can apply to differnt CNN models is very complicated if we want to code from scratch. However, fortunately, a group of Google engineers develope Keras-vis (https://raghakot.github.io/keras-vis/vis.visualization/) package[2], which can generalize Feature Visualization to all CNN models. We therefore focus on learning to utilize the Activation Maximization function and interprete different trained models.

Improvement with more time

In this project, we applied keras as the main libary for implementing activation visualizations and building CNN models. VGG16 is also imported as keras base along its feature visualizations. If we have more time, it is also encouraged to try different libraries (eg. Pytorch, Lucid) and be familiar with different packages characteristics. In addition, GoogleNet Inception V1 is also a popular CNN model trained by ImageNet. Under lucid envirnment, we could also explore the maximum acivation and minimum activation efficiently.In the future,we also plan to use other CNN visualization methods such as Deconvolutional Neural Networks and Network Inversion to explore more on how CNNs extract features.

Activation Maximization

Overview

In this project, we use the most straight-forward technique: Activation Maximization to optimize the feature visualization.

Activation Maximization is proposed to visualize the preferred inputs of neurons in each layer. The preferred input can indicate what features of a neurons has learned.[3] (https://arxiv.org/pdf/1804.11191.pdf) The learned feature is represented by a synthesized input pattern that can give rise to the highest activation of a target neuron. In order to find such input pattern, each pixel of the CNN's input is iteratively changed to get the highest activation of the target neuron.

Activation Maximization algorithm

We may pose the activation maximization problem for a unit with index j on a layer l of a network Φ as finding an image x *where: $$x^ = arg{x}max\phi{l,j}(\theta,x)$$ and θ denotes the network parameters sets (weight and bias).[4] (https://arxiv.org/pdf/1602.03616.pdf) There are four main steps in this process:[5] (https://blog.keras.io/how-convolutional-neural-networks-see-the-world.html)

(1)Build a loss function that maximizes the activation of the particular filter.

(2)Compute the gradient of the input picture with respect to this loss function and move x in the direction of this gradient.

(3)Normalize trick: we normalize the gradient of the pixels of the input image(which avoids very small and very large gradients and ensures a smooth gradient ascent process).

(4)Build a iteration function that returns the loss and gradients given the input image. We start from a noise image, then iterate the interation function for enough time. The process terminates at a certain image x* when the image without any noise.

In order to overcome the uninterpretability problem of Acitivation Maximization in the Deep Neural Network, regularization methods have been implemented to collectively improve the image quality. AM with regulation:

$$x^* = arg_x max(\phi_{l,j}(\theta, x) - \lambda(x))$$

where $\lambda(x)$ is a parameterized regularization function.[3] (https://arxiv.org/pdf/1804.11191.pdf)

In the following sections, we utilize the function of *visualize_activation* from Keras-vis package to achieve Activation Maximization. The function is referenced in [6] (https://raghakot.github.io/keras-vis/visualizations/activation_maximization/).

Part I (a) MNIST Model Visualization

The MNIST model is constructed using keras. The model is composed of 2 convolutional layers, and the test accuracy achieves ~99%.
This model refenreced Raghavendra Kotikalapudi's official example for keras-vis package (https://github.com/raghakot/keras-vis/blob/master/examples/mnist/activation_maximization.ipynb)[3].

In [1]:
```
from __future__ import print_function

import numpy as np
import keras

from keras.datasets import mnist
from keras.models import Sequential, Model
from keras.layers import Dense, Dropout, Flatten, Activation, Input
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
```

```
/Users/MichelleZhu/anaconda3/lib/python3.6/site-packages/h5py/__init__.
py:34: FutureWarning: Conversion of the second argument of issubdtype f
rom `float` to `np.floating` is deprecated. In future, it will be treat
ed as `np.float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

To save time and capacity You can skip the next two blocks and draw the trained model from our git repo.

In [3]:
```
batch_size = 128
num_classes = 10
epochs = 5

# input image dimensions
img_rows, img_cols = 28, 28

# the data, shuffled and split between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()

if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
```

```
Downloading data from https://s3.amazonaws.com/img-datasets/mnist.npz
11493376/11490434 [==============================] - 2s 0us/step
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
```

In [4]:
```
# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax', name='preds'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))

score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [==============================] - 132s 2ms/step - loss: 0.
2327 - acc: 0.9278 - val_loss: 0.0553 - val_acc: 0.9816
Epoch 2/5
60000/60000 [==============================] - 117s 2ms/step - loss: 0.
0870 - acc: 0.9744 - val_loss: 0.0388 - val_acc: 0.9869
Epoch 3/5
60000/60000 [==============================] - 119s 2ms/step - loss: 0.
0633 - acc: 0.9805 - val_loss: 0.0301 - val_acc: 0.9889
Epoch 4/5
60000/60000 [==============================] - 120s 2ms/step - loss: 0.
0512 - acc: 0.9841 - val_loss: 0.0287 - val_acc: 0.9905
Epoch 5/5
60000/60000 [==============================] - 115s 2ms/step - loss: 0.
0454 - acc: 0.9862 - val_loss: 0.0298 - val_acc: 0.9906
Test loss: 0.02977865039347744
Test accuracy: 0.9906
```

In [5]:
```
model.save('mnist_train_model.h5')
```

In [6]:
```
from keras.models import load_model

model = load_model('/Users/MichelleZhu/Documents/GitHub/CNN-Feature-Visualization/doc/mnist_train_model.h5')
```

Summary of the MNIST model

The MNIST model with 2 convolutional layer predicts the hand-written digit input picture to a most likely digit from 0 to 9. Besides of that, a max-pooling layer and two dropout layer are added after the first two convolutional layer. It was Trained on 60000 samples, validated on 10000 samples. Finally 5 epoch got 0.9908 test accuracy.

The last dense layer named "Preds" is constructed using softmax with 10 outputs, each reperents the possibility of a digit from 0 to 9.

In [7]:
```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 24, 24, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 12, 12, 64)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 9216)              0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               1179776
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
preds (Dense)                (None, 10)                1290
=================================================================
Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable params: 0
_________________________________________________________________
```

Dense Layer Visualization

The MNIST data set is in grey scale, which does not concern classifications of colors and shapes, so we only perform feature visualize by optimization on the last dense layer.

Command line to install the keras-vis package: pip install git+git://github.com/raghakot/keras-vis.git --upgrade --no-deps

As suggested by the official document (https://github.com/raghakot/keras-vis/blob/master/examples/mnist/activation_maximization.ipynb) the 'softmax' need to be converted to 'linear' at the last layer. Otherwise, there will be problem of suboptimal.

Step 1: Visualizing input that maximizes the output of node 0

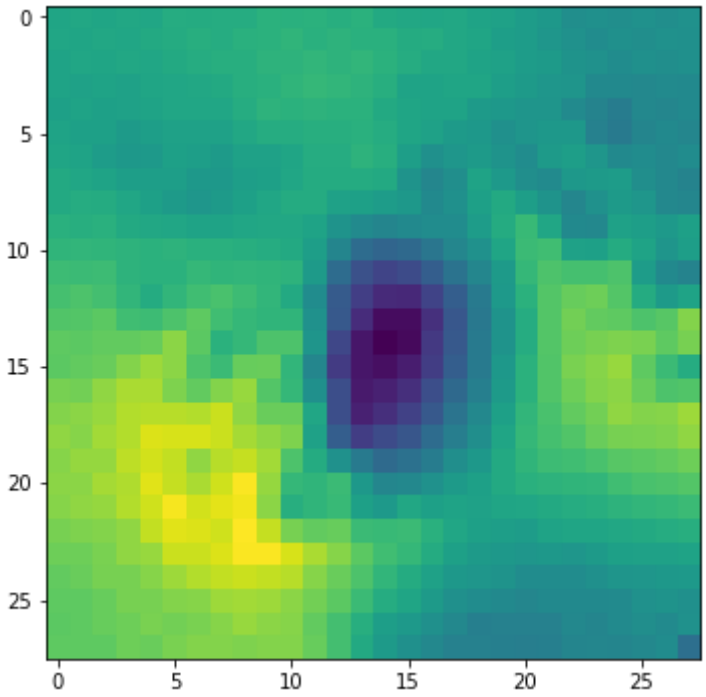
In [9]:
```
from vis.visualization import visualize_activation
from vis.visualization import get_num_filters
from vis.utils import utils
from keras import activations

from matplotlib import pyplot as plt
%matplotlib inline
```

In [27]:
```
#In Last Dense layer, get the total number of output
get_num_filters(model.layers[7])
```

Out[27]: 10

In [11]:
```
plt.rcParams['figure.figsize'] = (18, 6)

# Utility to search for layer index by name.
# Alternatively we can specify this as -1 since it corresponds to the la
st layer.
layer_idx = utils.find_layer_idx(model, 'preds')

# Swap softmax with linear
model.layers[layer_idx].activation = activations.linear
model = utils.apply_modifications(model)

# This is the output node we want to maximize.
filter_idx = 0
img = visualize_activation(model, layer_idx, filter_indices=filter_idx)
plt.imshow(img[..., 0])
```

Out[11]: <matplotlib.image.AxesImage at 0x182632e048>



In [9]:
```
print(layer_idx)
```

```
7
```

Insights on the first try of node 0 feature visualization:

By seting **filter_indices = 0**, the visualization of node 0 maximized the input so that final output. However it does not look like a zero. So we need to do the following two steps to optimize the input:

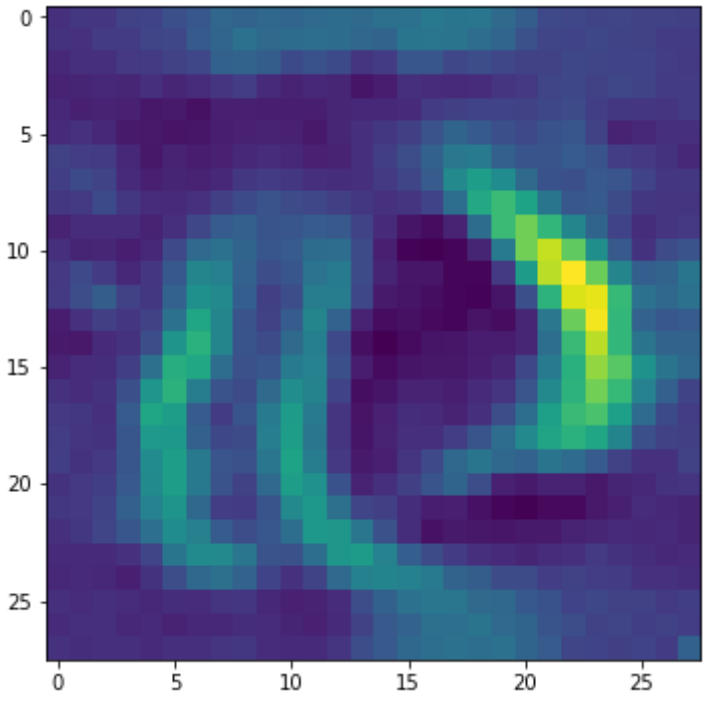
- **Input range :** the preprocess range by defualt is (0,1), we should change the input range and specify between **(0.,1.)** (float type)
- **Regularization Parameter :** The Regularization Parameter default weights might be dominating activation maximization loss weight. One way to debug this is to use **verbose=True** and examine individual loss values.

Step 2: Specifying the Input Range

Set the input_range = (0.,1.), and then visualize the node 0 input image again.

```
In [10]: img = visualize_activation(model, layer_idx, filter_indices=filter_idx,
         input_range=(0., 1.))
         plt.imshow(img[..., 0])
```

```
Out[10]: <matplotlib.image.AxesImage at 0x1c2beb8438>
```



Here we can see that the input image has much less noise than the one without specifying the input range

Step 3: Tuning regularization weights

The activation maximization function has a issue that the optimized input can go out of the traning distribution space. Total variation in the function ensures that the input images are blobber and not scattered, so that the pattern in the image can be recognized. The problem is sometimes the total variation and L-p norm can dominante the main Activation Maximization loss.

When verbose=True, we can see the activation max loss is not converging and bouncing around.

In [ ]:
```
img = visualize_activation(model, layer_idx, filter_indices=filter_idx,
input_range=(0., 1.), verbose=True)
plt.imshow(img[..., 0])
```

The Overall Loss was affected by TV loss and the ActivationMax Loss is not converging. So We would like to set the total variation loss and L-p norm weight to 0. So wee can see how is the ActivationMax Loss performing in each iteration.

Iteration: 195, named_losses: [('ActivationMax Loss', -1642.668), ('L-6.0 Norm Loss', 1.4082897), ('TV(2.0) Loss', 838.6276)], overall loss: -802.6320190429688
Iteration: 196, named_losses: [('ActivationMax Loss', -1641.0078), ('L-6.0 Norm Loss', 1.4093717), ('TV(2.0) Loss', 842.1858)], overall loss: -797.41259765625
Iteration: 197, named_losses: [('ActivationMax Loss', -1640.6877), ('L-6.0 Norm Loss', 1.4094273), ('TV(2.0) Loss', 836.90247)], overall loss: -802.3758544921875
Iteration: 198, named_losses: [('ActivationMax Loss', -1644.2861), ('L-6.0 Norm Loss', 1.4108175), ('TV(2.0) Loss', 846.14453)], overall loss: -796.7308349609375
Iteration: 199, named_losses: [('ActivationMax Loss', -1646.0762), ('L-6.0 Norm Loss', 1.4108834), ('TV(2.0) Loss', 842.84906)], overall loss: -801.8162231445312
Iteration: 200, named_losses: [('ActivationMax Loss', -1644.426), ('L-6.0 Norm Loss', 1.411845), ('TV(2.0) Loss', 846.4703)], overall loss: -796.5438842773438

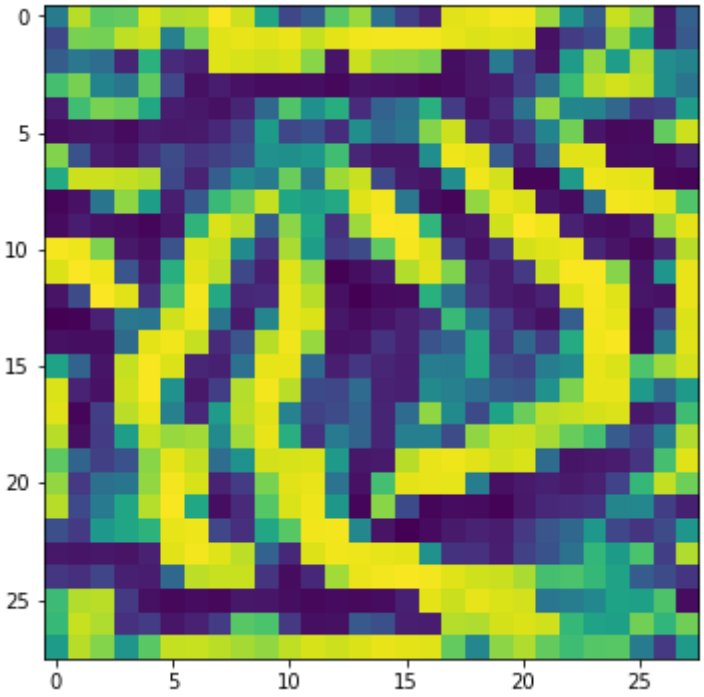
 image.png

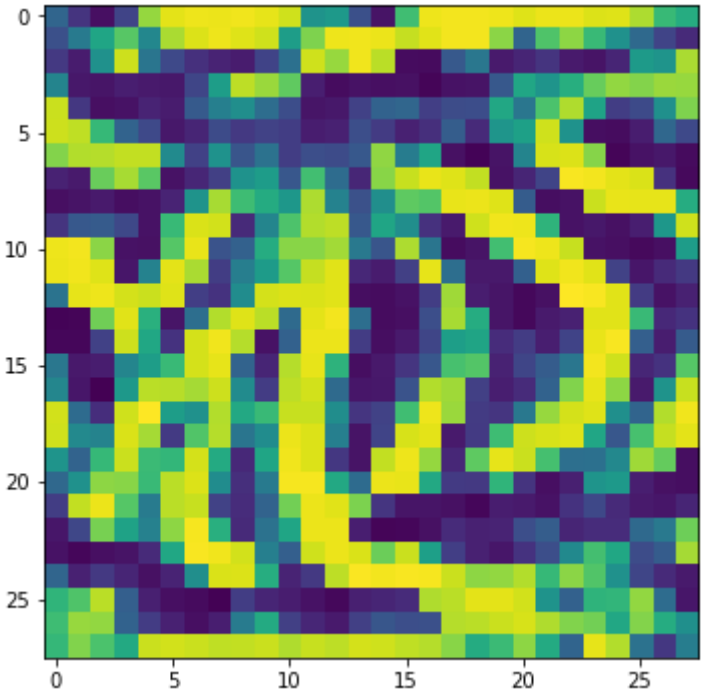
In [ ]:
```
img = visualize_activation(model, layer_idx, filter_indices=filter_idx,
input_range=(0., 1.),
                           tv_weight=0., lp_norm_weight=0., verbose=True
)
plt.imshow(img[..., 0])
```

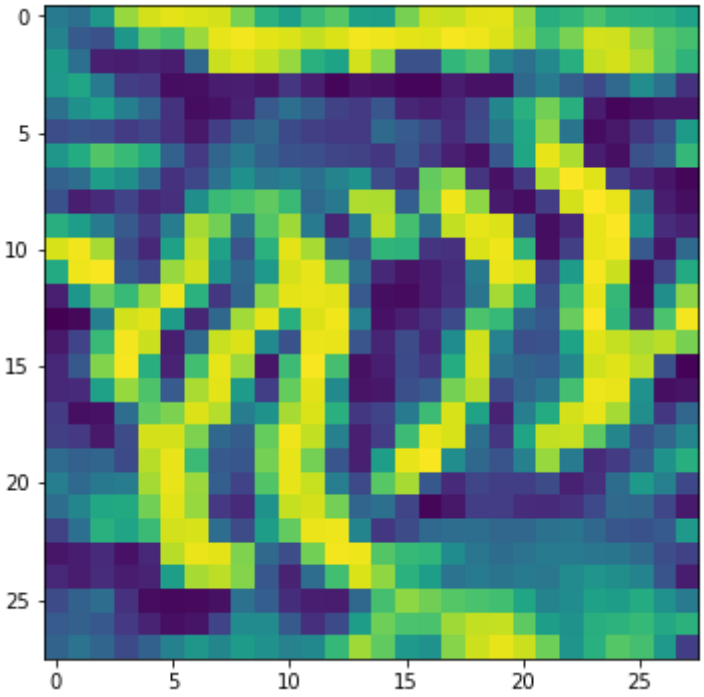
Iteration: 193, named_losses: [('ActivationMax Loss', -19257.84)], overall loss: -19257.83984375
Iteration: 194, named_losses: [('ActivationMax Loss', -19352.492)], overall loss: -19352.4921875
Iteration: 195, named_losses: [('ActivationMax Loss', -19447.11)], overall loss: -19447.109375
Iteration: 196, named_losses: [('ActivationMax Loss', -19541.332)], overall loss: -19541.33203125
Iteration: 197, named_losses: [('ActivationMax Loss', -19636.35)], overall loss: -19636.349609375
Iteration: 198, named_losses: [('ActivationMax Loss', -19731.139)], overall loss: -19731.138671875
Iteration: 199, named_losses: [('ActivationMax Loss', -19825.717)], overall loss: -19825.716796875
Iteration: 200, named_losses: [('ActivationMax Loss', -19920.195)], overall loss: -19920.1953125

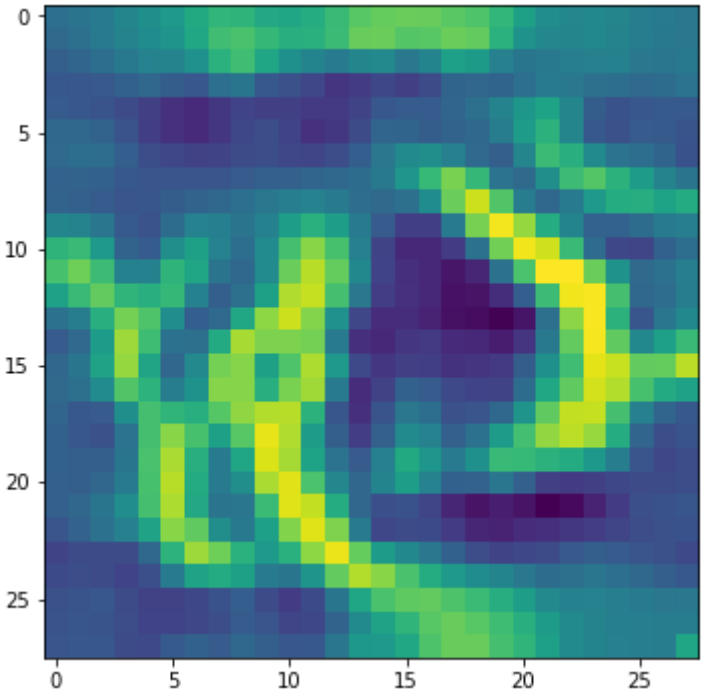
The ActivationMax Loss is converging after setting the other two loss weights to zero. However, the input picture also looks very unnatural. So we would like to set several different level of total variation weights and see which level can generate a more natural-look input.

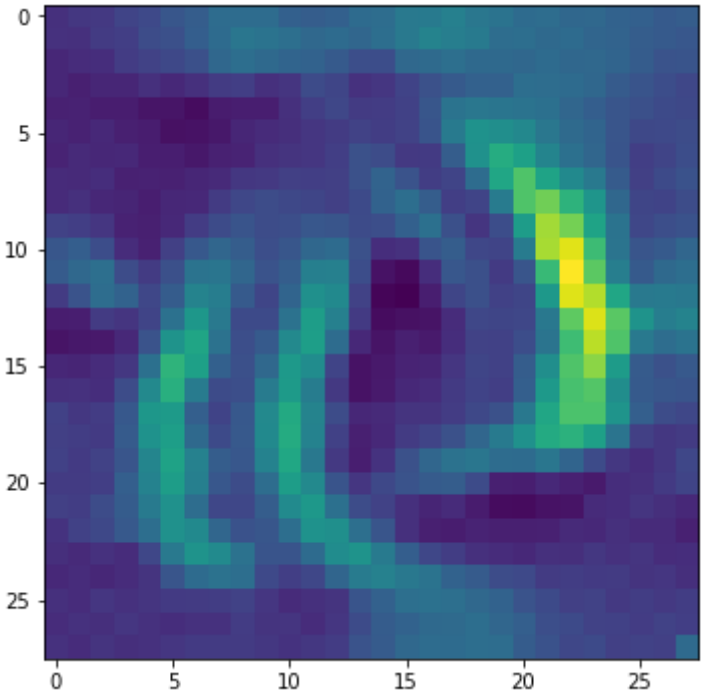
In [13]:
```
for tv_weight in [1e-3, 1e-2, 1e-1, 1, 10]:
    # Lets turn off verbose output this time to avoid clutter and just s
ee the output.
    img = visualize_activation(model, layer_idx, filter_indices=filter_i
dx, input_range=(0., 1.),
                               tv_weight=tv_weight, lp_norm_weight=0.)
    plt.figure()
    plt.imshow(img[..., 0])
```


0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25

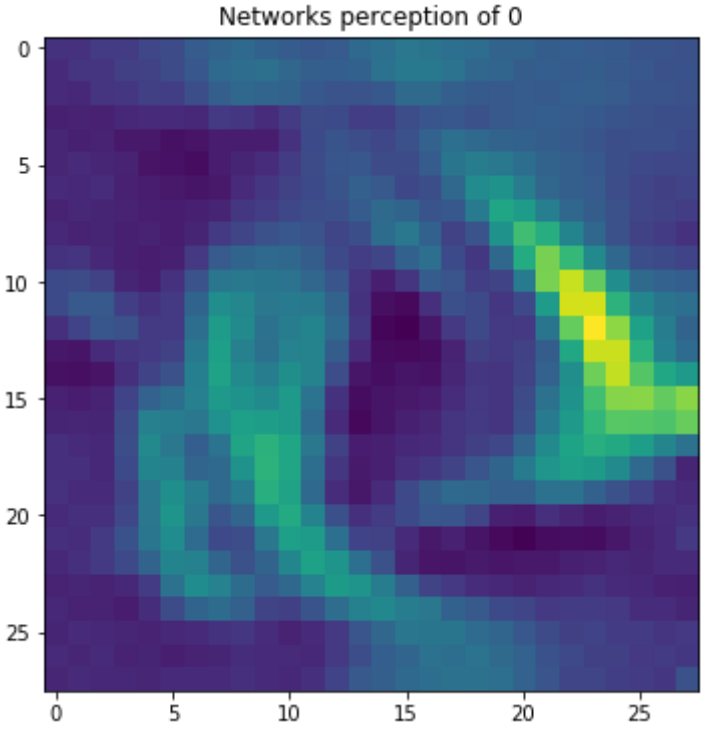

0
5
10
15
20
25
0
5
10
15
20
25

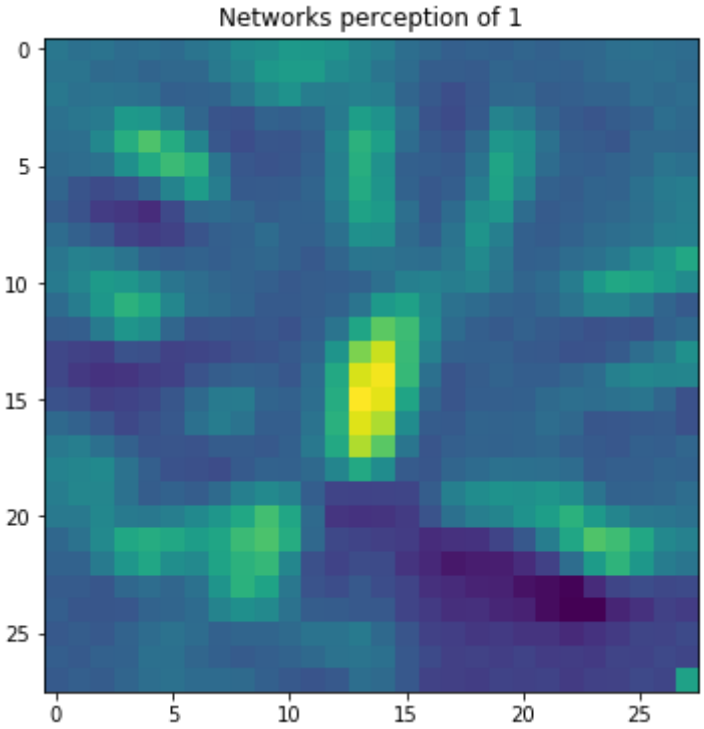

0
5
10
15
20
25
0
5
10
15
20
25

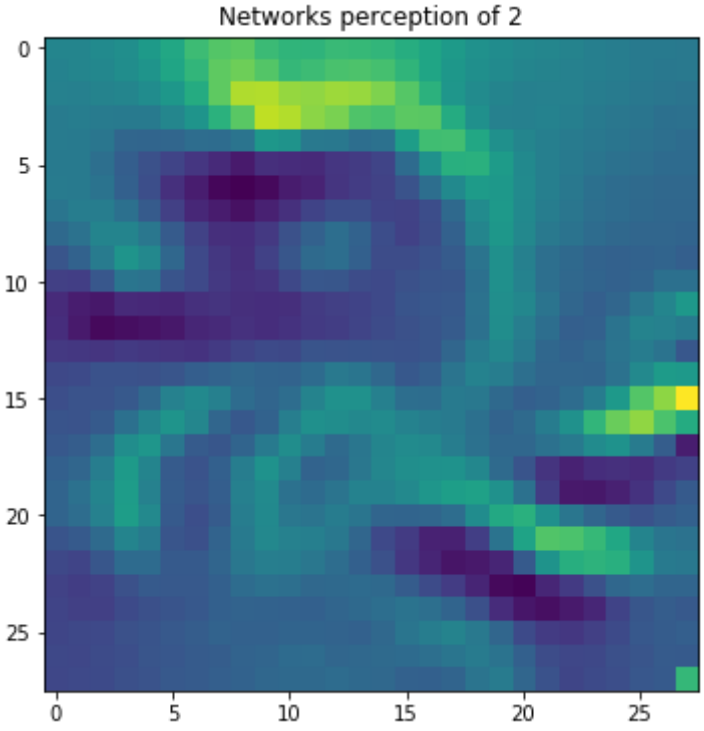


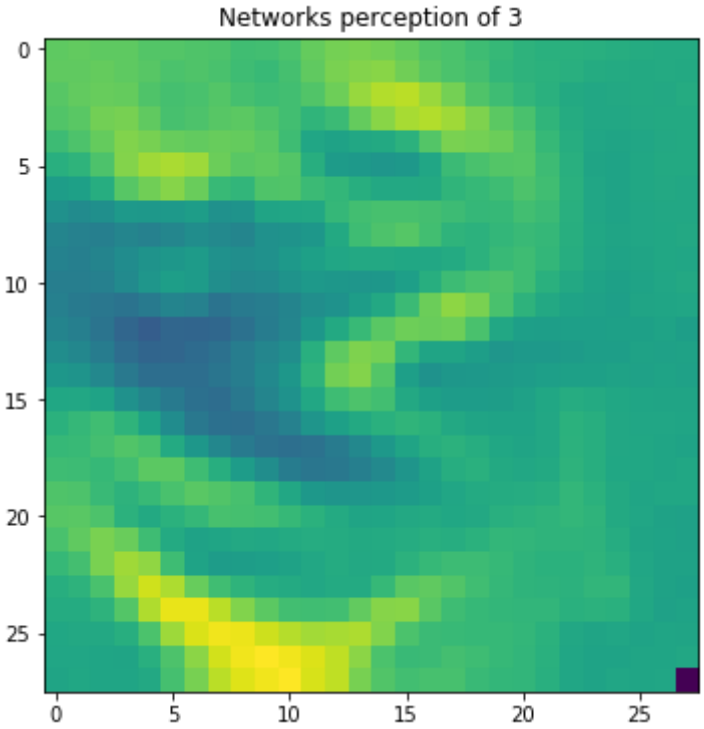
Here we can see that the default weight for Total Variation = 10 is the optimal choice. So we keep it for the rest of the 9 nodes input.

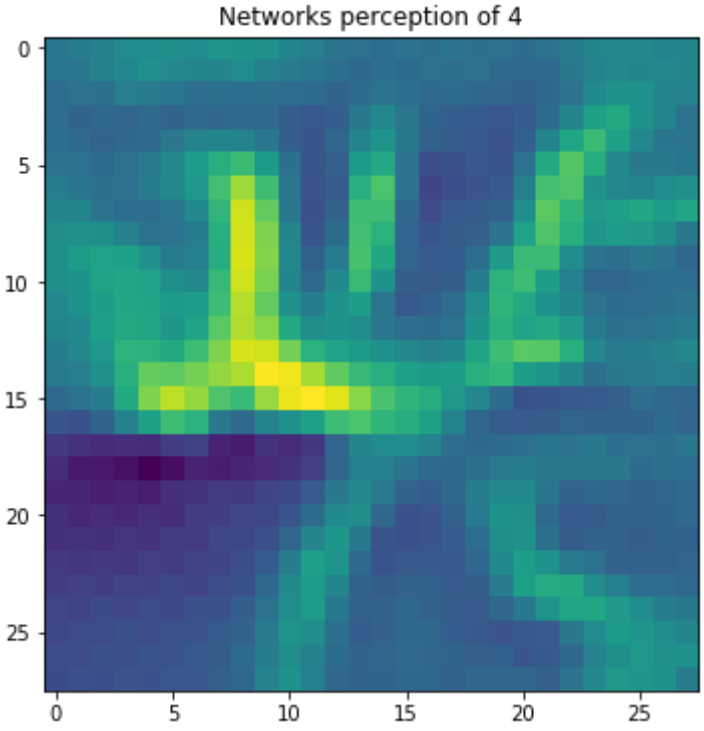
In [14]:
```
for output_idx in np.arange(10):
    # Lets turn off verbose output this time to avoid clutter and just s
ee the output.
    img = visualize_activation(model, layer_idx, filter_indices=output_i
dx, input_range=(0., 1.))
    plt.figure()
    plt.title('Networks perception of {}'.format(output_idx))
    plt.imshow(img[..., 0])
```

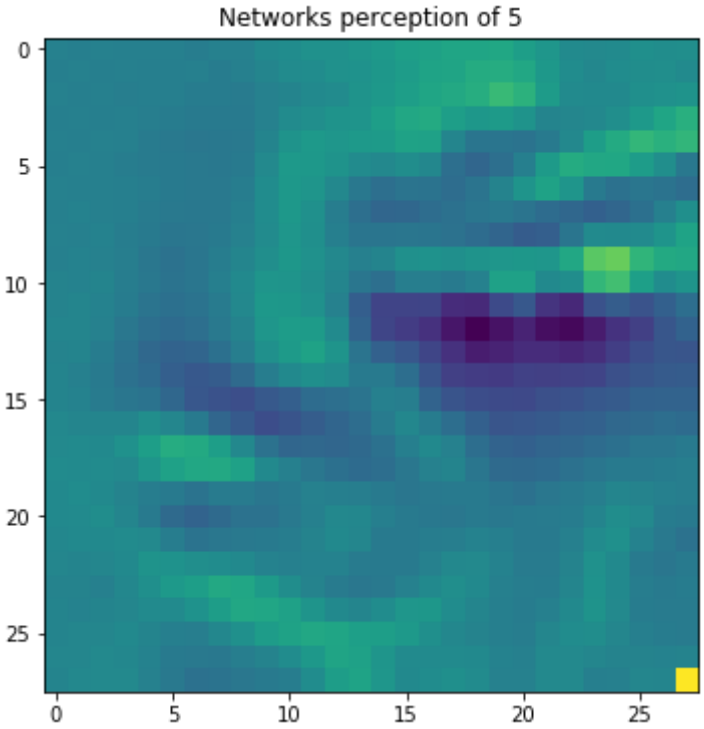

Networks perception of 0
0
5
10
15
20
25
0
5
10
15
20
25

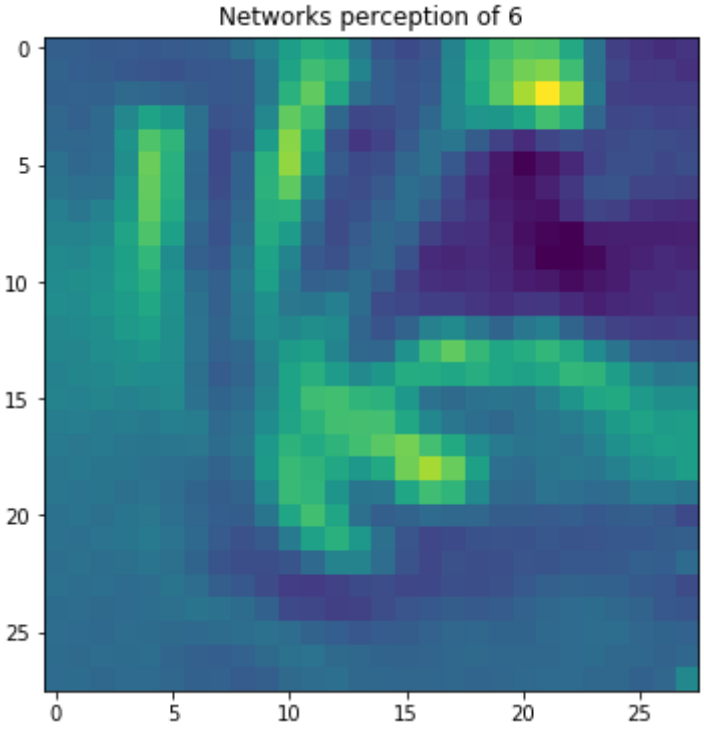

Networks perception of 1
0
5
10
15
20
25
0
5
10
15
20
25

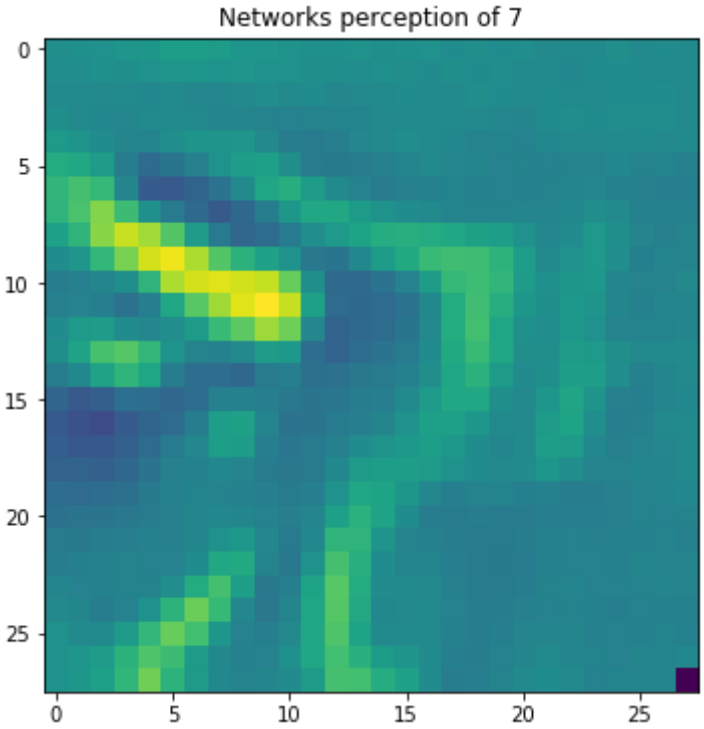

Networks perception of 2
0
5
10
15
20
25
0
5
10
15
20
25

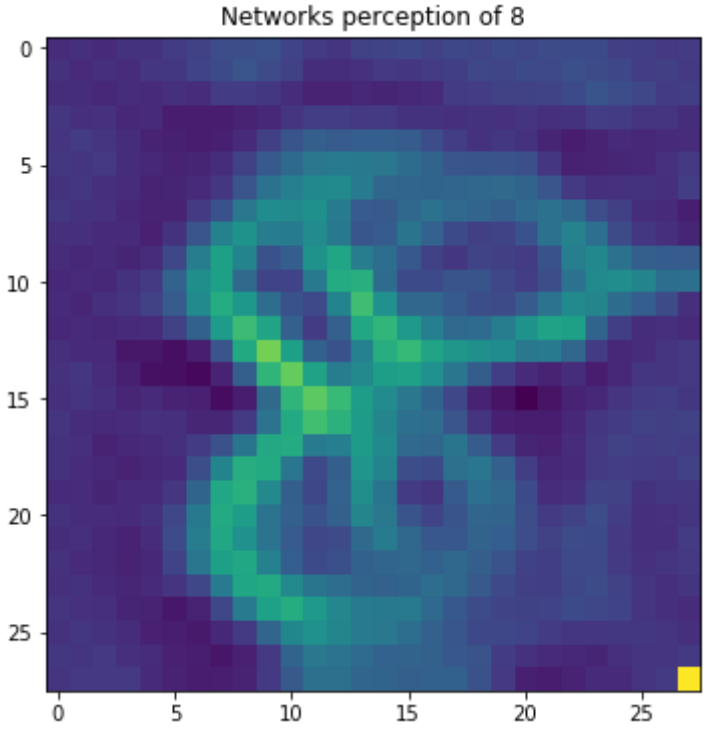

Networks perception of 3
0
5
10
15
20
25
0
5
10
15
20
25

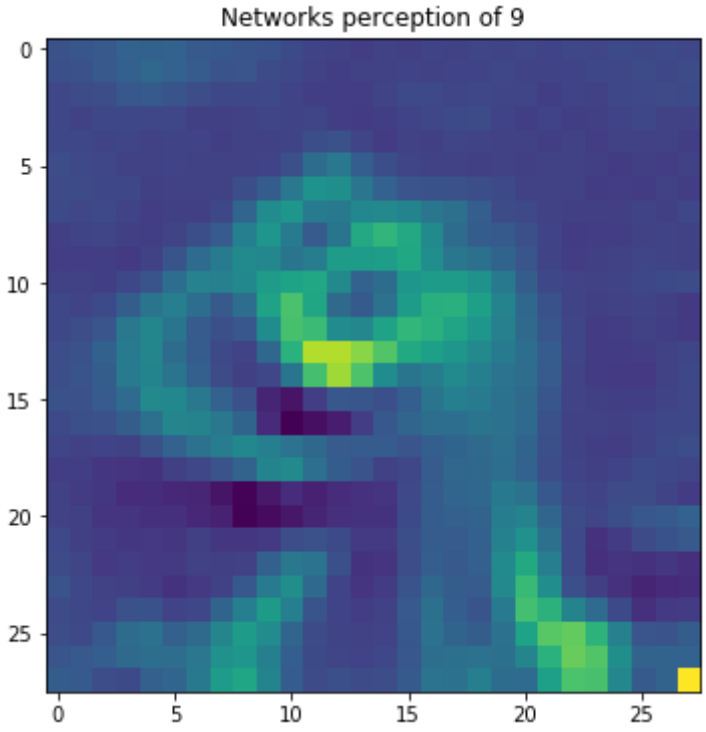

Networks perception of 4
0
5
10
15
20
25
0
5
10
15
20
25


Networks perception of 5
0
5
10
15
20
25
0
5
10
15
20
25


Networks perception of 6
0
5
10
15
20
25
0
5
10
15
20
25


Networks perception of 7
0
5
10
15
20
25
0
5
10
15
20
25





Here we can see that activation maximized input images can still be recognized to what it originally looks like.

Part I (b) Cats_Dog_Model_Visualization

In this part, we visualize the trained cats_dogs model in which the training process is shown in Part II. The way of interpreting filters is to generate an input image that maximizes the filter output. This allows us to generate an input that activates the filter. We continue to use the visualize_activation function from vis.visualization as the activation maximization visualization method. The final dense layer's activation is maximized and the input image from the second last dense is shown. We also plot input images which maximally optimize the fliters (we chose 10 filters randomly) from each convolutional layer. This part of code is referenced [8] (https://github.com/raghakot/keras-vis/blob/master/examples/vggnet/activation_maximization.ipynb).

In [1]:
```
from __future__ import print_function

import numpy as np
import keras

from keras.datasets import mnist
from keras.models import Sequential, Model
from keras.layers import Dense, Dropout, Flatten, Activation, Input
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K

from vis.visualization import visualize_activation
from vis.utils import utils
from keras import activations
from vis.utils import utils
from vis.visualization import get_num_filters

from matplotlib import pyplot as plt
%matplotlib inline
```

```
Using TensorFlow backend.
```

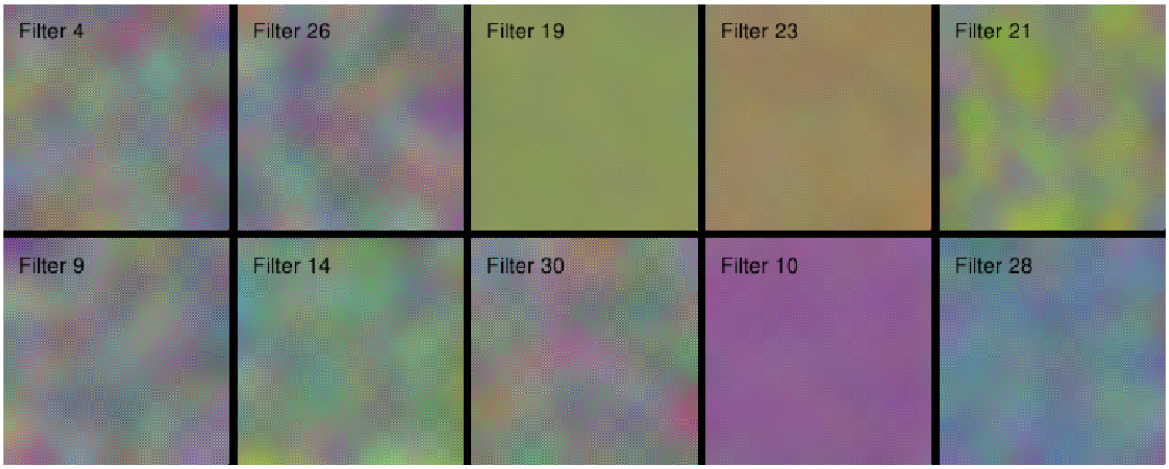
In [2]:
```
from keras.models import load_model

model = load_model('/GitHub/CNN-Feature-Visualization/output/cats_and_dogs_small_2.h5')
model.summary()
```

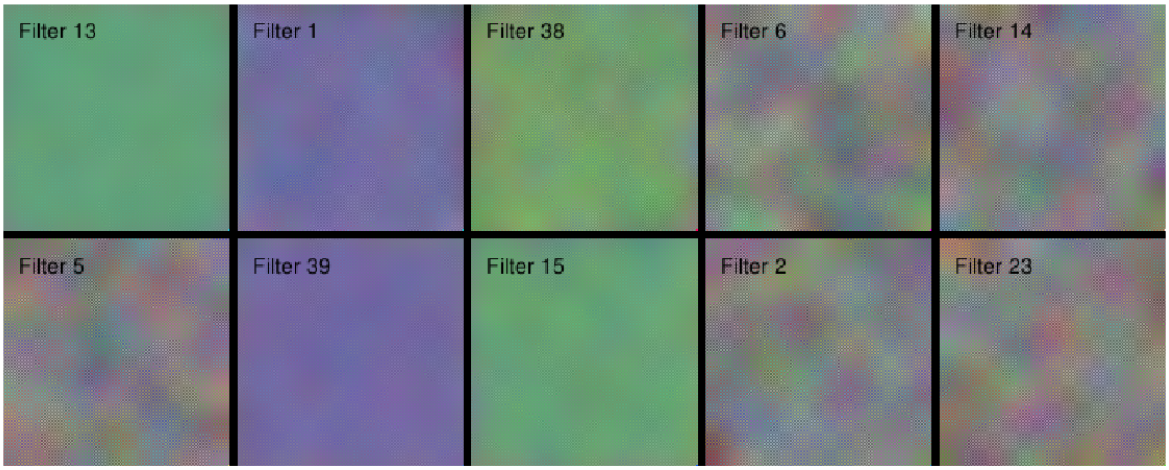
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 148, 148, 32)      896
_________________________________________________________________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

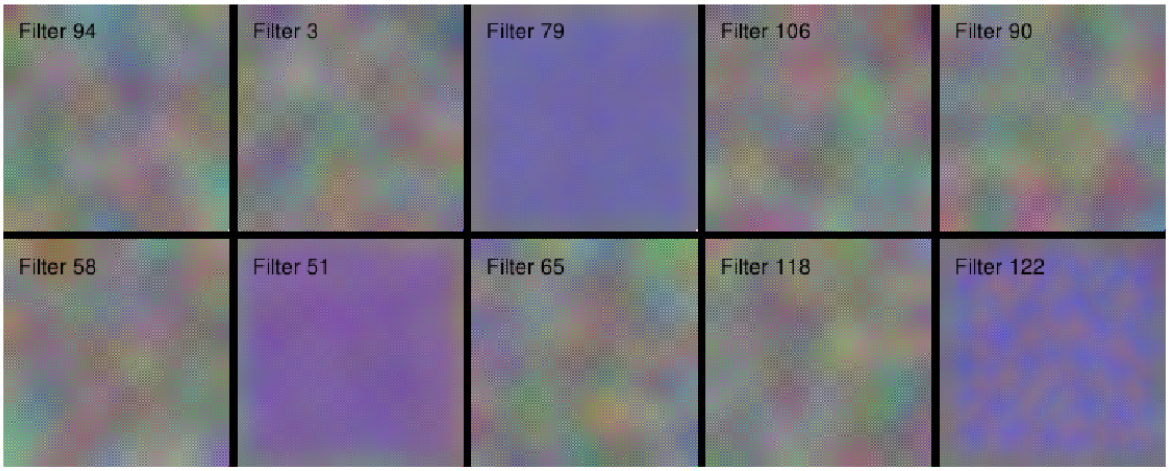
There are total of 4 convolutional layers in which 10 filters are randomly selected as our activation optimization object. From the plot, the difference is not quite obvious, but we can still observe that the convolutional layer 3 and 4 have a deeper color comparing with layer 1 and 2. Also, for conv2d_8 the filters focus more on in detail where the color concentration is centered while conv2d_5 and conv2d_6 color seems spread equally. As a result, there is difference between input images that maximally activate differet layers.

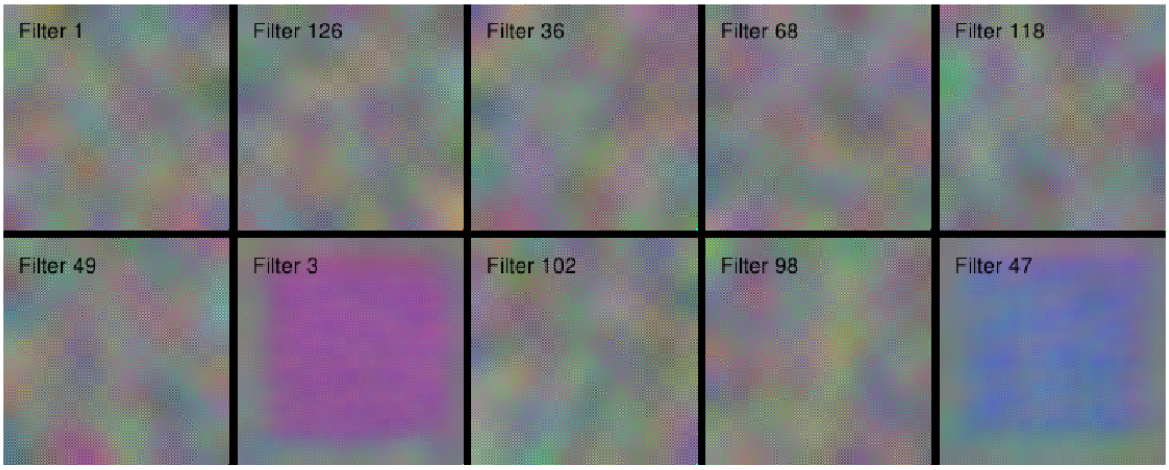
Since our model is trained within a relative small training set, the input images which optimize the activation for layers do not demostrate detailed information. Therefore, we apply the popular VGG16 model in part (3) to discover more about the input images which optimize the activation.

In [3]:
```
selected_indices = []
for layer_name in ['conv2d_5', 'conv2d_6', 'conv2d_7', 'conv2d_8']:
    layer_idx = utils.find_layer_idx(model, layer_name)

    # Visualize all filters in this layer.
    filters = np.random.permutation(get_num_filters(model.layers[layer_i
dx]))[:10]
    selected_indices.append(filters)

    # Generate input image for each filter.
    vis_images = []
    for idx in filters:
        img = visualize_activation(model, layer_idx, filter_indices=idx)

        # Utility to overlay text on image.
        img = utils.draw_text(img, 'Filter {}'.format(idx))
        vis_images.append(img)

    # Generate stitched image palette with 5 cols so we get 2 rows.
    stitched = utils.stitch_images(vis_images, cols=5)
    plt.figure(figsize=(20, 20))
    plt.axis('off')
    plt.imshow(stitched)
    plt.show()
```


Filter 4
Filter 26
Filter 19
Filter 23
Filter 21
Filter 9
Filter 14
Filter 30
Filter 10
Filter 28


Filter 13
Filter 1
Filter 38
Filter 6
Filter 14
Filter 5
Filter 39
Filter 15
Filter 2
Filter 23

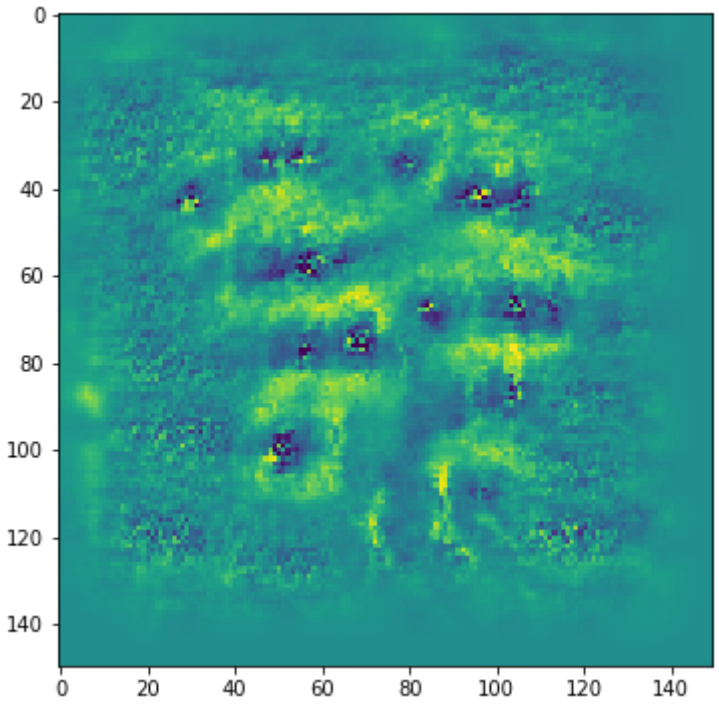

Filter 94
Filter 3
Filter 79
Filter 106
Filter 90
Filter 58
Filter 51
Filter 65
Filter 118
Filter 122



The following two graphs shows the input images that maximally activate the sigmoid dense layer. Since we are doing a binary classification with cats and dogs, the last layer is a sigmoid function instead of softmax. We could observe that both images are quite similar to each other with its color and some feature patterns. However, the two images completely activate different node so that the final output will be 0 for cats and 1 for dogs. As a result, these are two completely different imput images that the CNN classification based on.

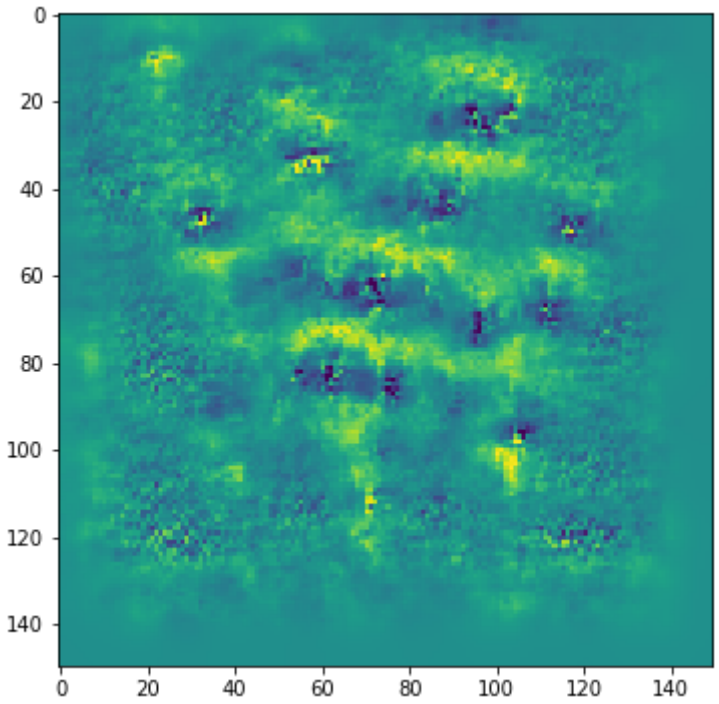
In [12]:
```
plt.rcParams['figure.figsize'] = (18, 6)

# Utility to search for layer index by name.
# Alternatively we can specify this as -1 since it corresponds to the la
st layer.
#layer_idx = utils.find_layer_idx(model, 'preds')
layer_idx = 11


# Swap softmax with linear
model.layers[layer_idx].activation = activations.linear
model = utils.apply_modifications(model)

# This is the output node we want to maximize.
filter_idx = 0
img = visualize_activation(model, layer_idx, filter_indices=filter_idx)
plt.imshow(img[..., 0])
```

Out[12]: <matplotlib.image.AxesImage at 0x1c33d7f940>



In [13]:
```
plt.rcParams['figure.figsize'] = (18, 6)

# Utility to search for layer index by name.
# Alternatively we can specify this as -1 since it corresponds to the la
st layer.
#layer_idx = utils.find_layer_idx(model, 'preds')
layer_idx = 11


# Swap softmax with linear
model.layers[layer_idx].activation = activations.linear
model = utils.apply_modifications(model)

# This is the output node we want to maximize.
filter_idx = -1
img = visualize_activation(model, layer_idx, filter_indices=filter_idx)
plt.imshow(img[..., 0])
```

Out[13]: <matplotlib.image.AxesImage at 0x1c349e00b8>



Part I (c) Visualization of a typical model trained by ImageNet

we try to visualize what deep convolutional neural networks really learn, and how they understand the images we feed them.

We will use Keras and related extension to visualize inputs that maximize the activation of the filters in different layers of the VGG-16 architecture, trained on ImageNet.

In [1]:
```
from keras.applications import VGG16
from vis.utils import utils
from keras import activations
import numpy as np
from vis.visualization import visualize_activation

from matplotlib import pyplot as plt
%matplotlib inline

# Build the VGG16 network with ImageNet weights
model = VGG16(weights='imagenet', include_top=True)
```

```
Using TensorFlow backend.
```

Model architecture

VGG-16 (also called OxfordNet) is a convolutional neural network architecture named after the Visual Geometry Group from Oxford, who developed it. It was used to win the ILSVR (ImageNet) competition in 2014. To this day is it still considered to be an excellent vision model, although it has been somewhat outperformed by more revent advances such as Inception and ResNet. [5] (https://blog.keras.io/how-convolutional-neural-networks-see-the-world.html)

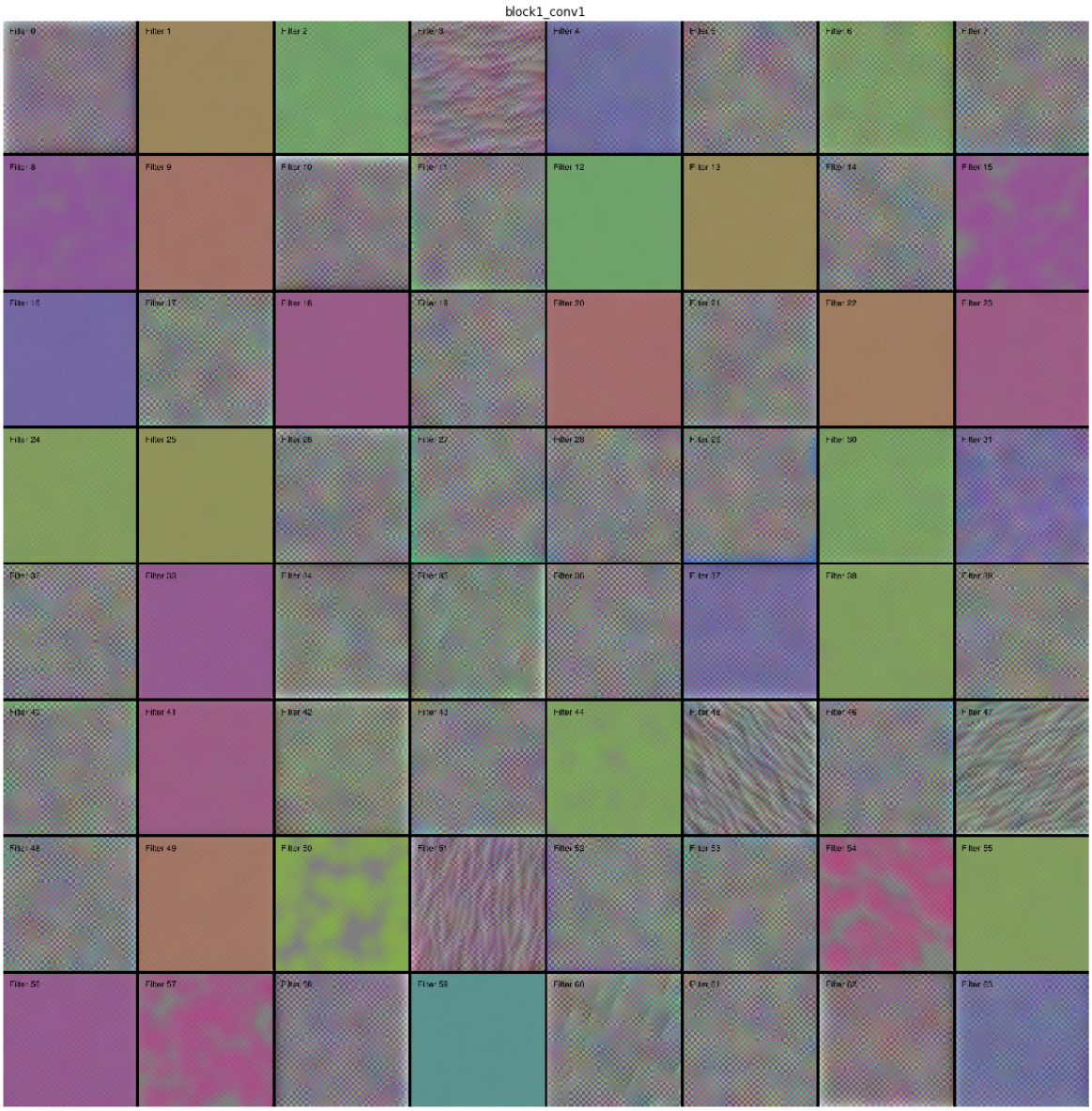
In [1]:
```
model.summary()
```

```
---------------------------------------------------------------------
----
NameError                                 Traceback (most recent call l
ast)
<ipython-input-1-9a7881f870d4> in <module>()
----> 1 model.summary()

NameError: name 'model' is not defined
```

Besides fully-connected layers, VGG-16 model has 18 layers in total, which could be divided into 5 blocks. Each block contains several filters and 1 maxpooling layer. In this report we aim to explore what filters have learned. We wish to systematically display what sort of input maximizes each filter in each layer, giving us a neat visualization of the convnet's modular-hierarchical decomposition of its visual space.

Visualization of the first layer

In [15]:
```
from vis.visualization import get_num_filters

# The name of the layer we want to visualize
# You can see this in the model definition.
layer_name = 'block1_conv1'
layer_idx = utils.find_layer_idx(model, layer_name)

# Visualize all filters in this layer.
filters = np.arange(get_num_filters(model.layers[layer_idx]))

# Generate input image for each filter.
vis_images = []
for idx in filters:
    img = visualize_activation(model, layer_idx, filter_indices=idx)

    # Utility to overlay text on image.
    img = utils.draw_text(img, 'Filter {}'.format(idx))
    vis_images.append(img)

# Generate stitched image palette with 8 cols.
stitched = utils.stitch_images(vis_images, cols=8)
plt.figure(figsize=(20, 20))
plt.axis('off')
plt.imshow(stitched)
plt.title(layer_name)
plt.show()
```

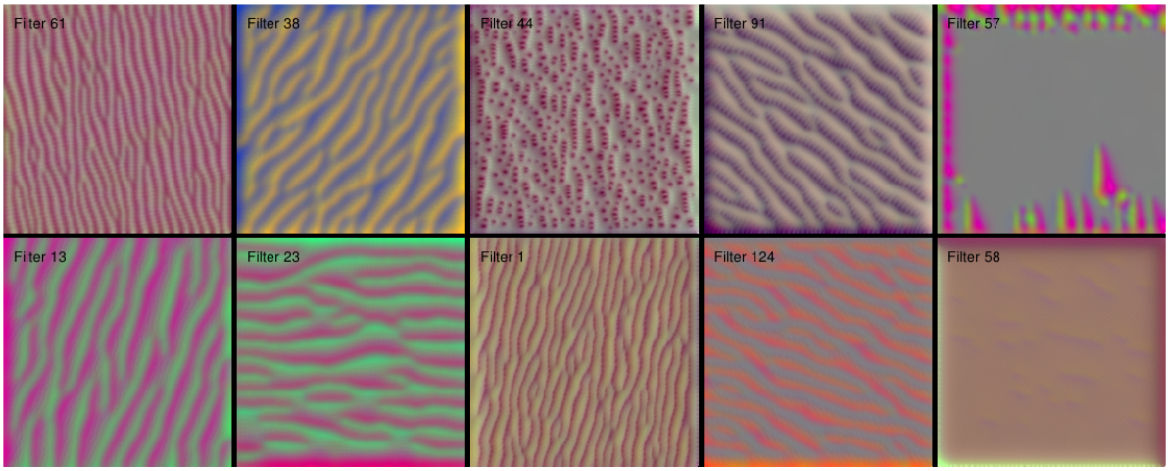


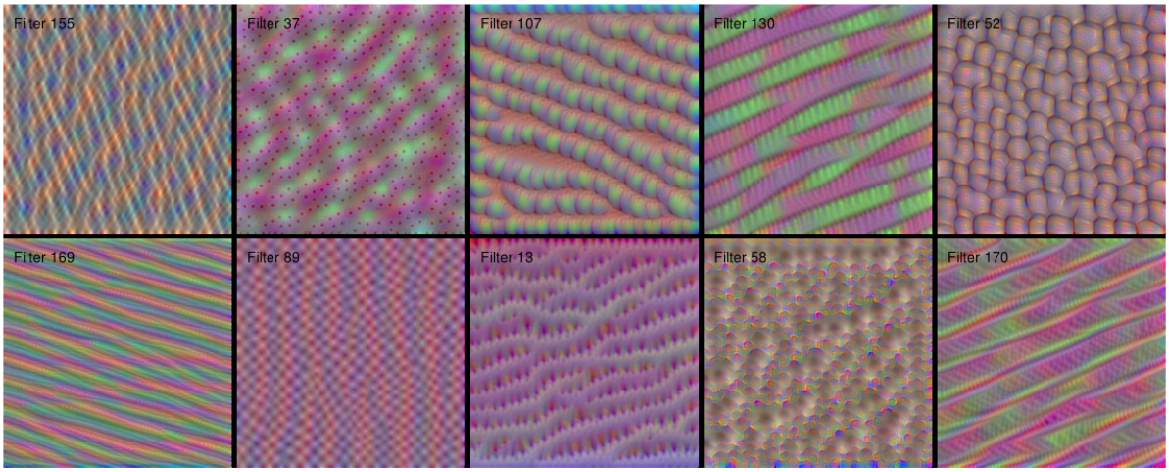
As we can see, the first layer basically just encode direction and color. We observed that some filters in the first layer were visualized as approximately pure colors and others, which indicates that these filters were serving as color filters. Meanwhile, some other visualized filters displayed significant stripes, which means that they have learned features relating to directions. In fact, customized VGG-16 model may learn different types of features in the first layer. Some researchers had shown that their visualization of first layer of customized VGG-16 model would only display colors [9] (https://lhncbc.nlm.nih.gov/publication/pub9809). So in general, we would make a conculusion that the first layer.

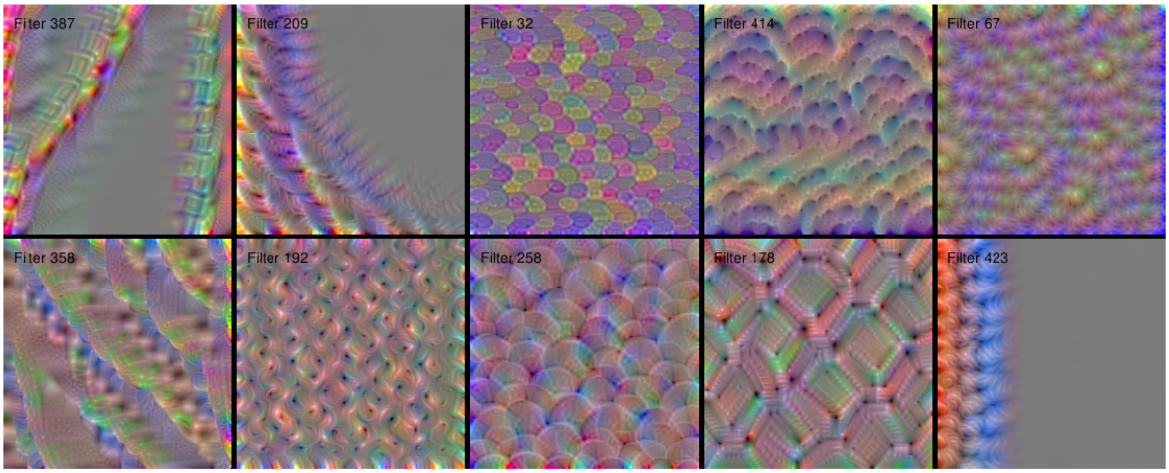
Visualization of higher level layers

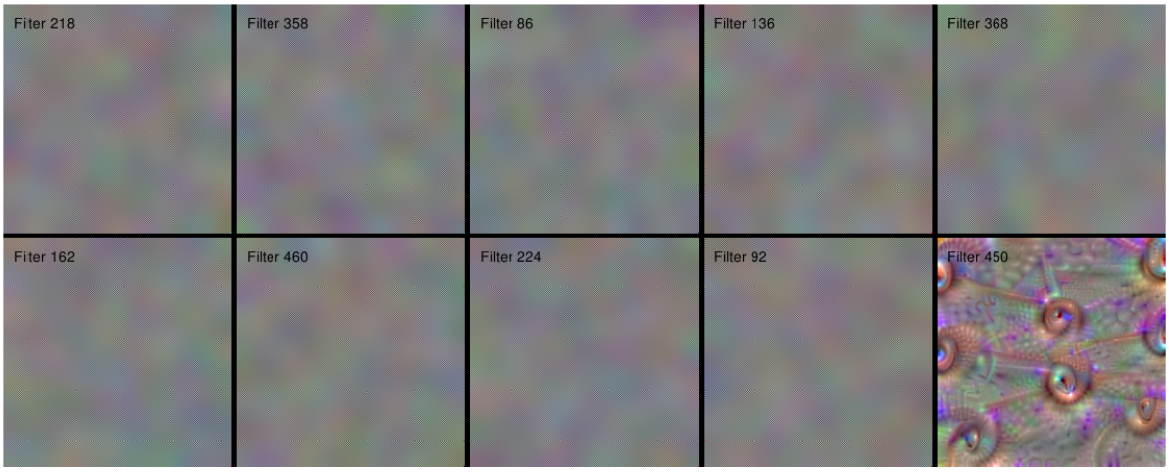
We chose to visualize 10 randomly selected filters in layers "block2_conv2", "block3_conv3", "block4_conv3", "block5_conv3".

In [18]:
```
from vis.visualization import get_num_filters

selected_indices = []
for layer_name in ['block2_conv2', 'block3_conv3', 'block4_conv3', 'bloc
k5_conv3']:
    layer_idx = utils.find_layer_idx(model, layer_name)

    # Visualize 10 random filters in this layer.
    filters = np.random.permutation(get_num_filters(model.layers[layer_i
dx]))[:10]
    selected_indices.append(filters)

    # Generate input image for each filter.
    vis_images = []
    for idx in filters:
        img = visualize_activation(model, layer_idx, filter_indices=idx)

        # Utility to overlay text on image.
        img = utils.draw_text(img, 'Filter {}'.format(idx))
        vis_images.append(img)

    # Generate stitched image palette with 5 cols so we get 2 rows.
    stitched = utils.stitch_images(vis_images, cols=5)
    plt.figure(figsize=(20, 20))
    plt.axis('off')
    plt.imshow(stitched)
    plt.show()
```


Filter 61
Filter 38
Filter 44
Filter 91
Filter 57
Filter 13
Filter 23
Filter 1
Filter 124
Filter 58


Filter 155
Filter 37
Filter 107
Filter 130
Filter 52
Filter 169
Filter 89
Filter 13
Filter 58
Filter 170


Filter 387
Filter 209
Filter 32
Filter 414
Filter 67
Filter 358
Filter 192
Filter 258
Filter 178
Filter 423



The low-level features were abstracted to construct complex, high-level features in the deeper convolutional layers.

In details, in block2_conv2, the filters begin to learn feature of edge information, direction, etc. However, the feature patterns are still quite simple without too much specific objects displayed. In block3_conv3, the patterns become more complex with more kinds of shapes. Not just points and stripes, but circles and polygons have emerged. In layer block4_conv3, the partterns become more intricately and colourfully piled up, with different sorts of overlap, stack and duplicate. Here part of the filters have suffered from unconvergence. In the last convolutional layer block5_conv3, unconvergence has become a serious problem. We would deal with this problem on the following.

Visualization of unconverged layer

From the visualization shown above, we notice that some of the filters in block5_conv3 failed to converge. This is usually because regularization losses (total variation and LP norm) are overtaking activation maximization loss [3] (https://arxiv.org/pdf/1804.11191.pdf). We could set Verbose to TRUE to have a look.

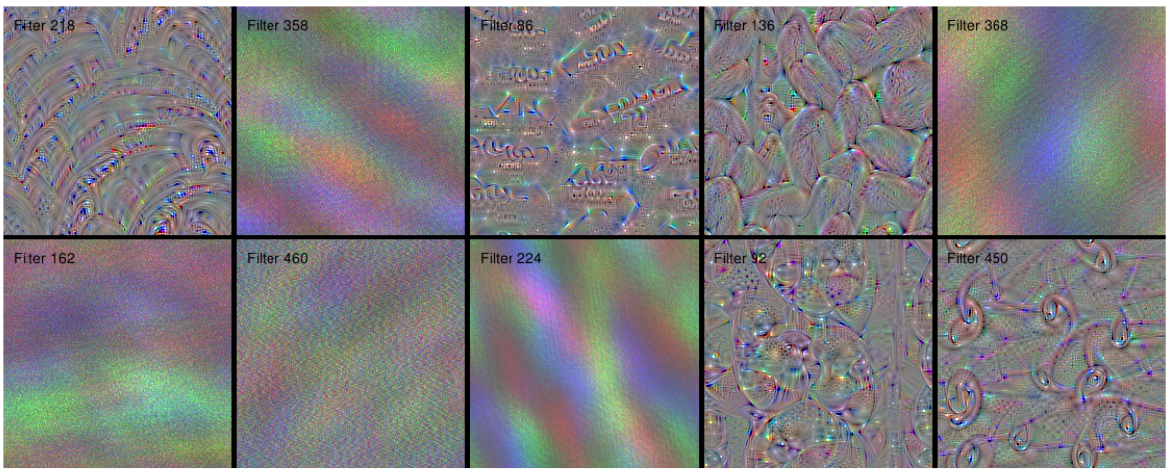
According to the user documents of keras-vis, we have several options to ecourage convergence if unconvergence happens: [10] (https://raghakot.github.io/keras-vis/vis.visualization/#visualize_activation)

- 1.Different regularization weights.
- 2.Increase number of iterations.
- 3.Add Jitter input_modifier.
- 4.Try with 0 regularization weights, generate a converged image and use that as seed_input with regularization enabled.

First move: we add jitter and disable total variation.

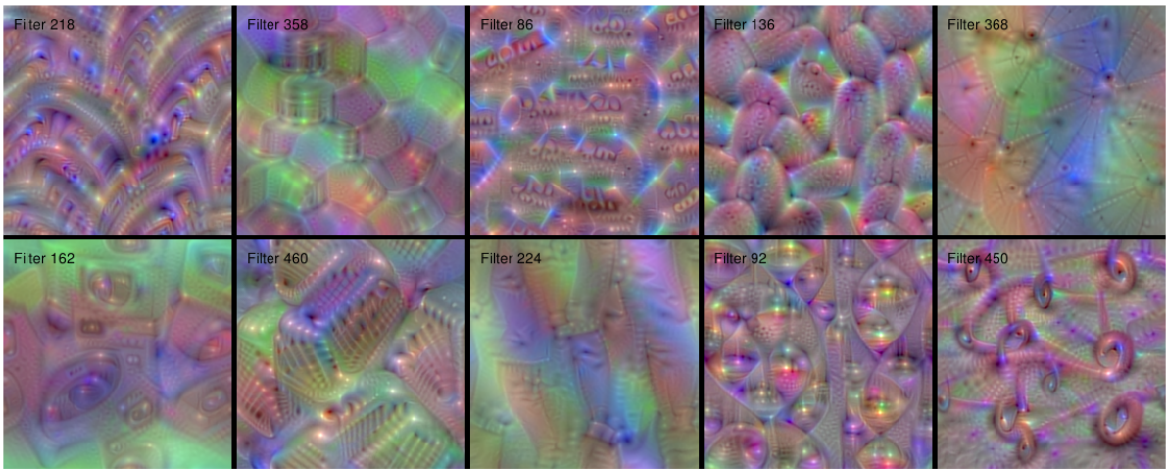
Adding jitter would bring us some small improvement.

In [27]:
```
from vis.input_modifiers import Jitter
layer_idx = utils.find_layer_idx(model, 'block5_conv3')

# We need to select the same random filters in order to compare the resu
lts.
filters = selected_indices[-1]
selected_indices.append(filters)

# Generate input image for each filter.
vis_images = []
for idx in filters:
    # We will jitter 5% relative to the image size.
    img = visualize_activation(model, layer_idx, filter_indices=idx,
                               tv_weight=0.,
                               input_modifiers=[Jitter(0.05)])

    # Utility to overlay text on image.
    img = utils.draw_text(img, 'Filter {}'.format(idx))
    vis_images.append(img)

# Generate stitched image palette with 5 cols so we get 2 rows.
stitched = utils.stitch_images(vis_images, cols=5)
plt.figure(figsize=(20, 20))
plt.axis('off')
plt.imshow(stitched)
plt.show()
```



Second move: We take a specific output from here and use it as a seed_input with total_variation enabled, making further improvement based on previous work.

In [28]:
```
# Generate input image for each filter.
new_vis_images = []
for i, idx in enumerate(filters):
    # We will seed with optimized image this time.
    img = visualize_activation(model, layer_idx, filter_indices=idx,
                               seed_input=vis_images[i],
                               input_modifiers=[Jitter(0.05)])
    # Utility to overlay text on image.
    img = utils.draw_text(img, 'Filter {}'.format(idx))
    new_vis_images.append(img)

# Generate stitched image palette with 5 cols so we get 2 rows.
stitched = utils.stitch_images(new_vis_images, cols=5)
plt.figure(figsize=(20, 20))
plt.axis('off')
plt.imshow(stitched)
plt.show()
```



As we can see here, the visualizations have converged very well without increasing iterations. Compared to previous visualizations which fail to show any patterns or textures, the improvement here is very significant.

In the highest convolutional layer (block5_conv3), we start to recognize textures similar to those found in objects of training images such as feathers, eyes, polygons, etc. However, the features learned here are still not close to "figures" in the meaninng of human sense.

If we would like to view the final classification decision makers, we should view the last layer "prediction", which contains 1000 categories. In the prediction layer, we definitely would be able to use our eyes to recoginize quite more accurate textures corresponding to their categories just as we did for 3-layer CNN model trained by MNIST in the previous chapter.

Up to now, the whole work above reveals some basic visualizations of filters for a typical and sophisticated CNN model, which is both complicated on model architecture as well as on training dataset. Apart from that, we succeed to deal with unconvergence problem.

In conclusion, the filters of CNN model have learn two main points: first, they understand a decomposition of their visual input space as a hierarchical-modular network of convolution filters; second, they understand a probabilitistic mapping between certain combinations of these filters and a set of arbitrary labels. But naturally, we can not qualify this method of learning as "seeing" in human sense.

Part II Learning a Cat

Summary

This part will show intermediate visualization of convolution and pooling layers in a network (the output of a layer is often called its "activation", the output of the activation function). This gives another view of how an input is decomposed by different filters learned from the network. The input is a [1,150,150,3] image of cat. The model is trained through 2000 cats and dogs with sigmoid activation in the final dense layer. 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. The final trained model has training and testing error of 95% and 82% respectively. The model we use is referenced in "Visualizing what convnets learn" [11] (https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/5.4-visualizing-what-convnets-learn.ipynb).

Data

The CNN model we designed is for a binary classification problem (cats and dogs). The small dataset we used is from Kaggle [12] (https://www.kaggle.com/c/dogs-vs-cats/data). The original dataset contains 25,000 images of dogs and cats (12,500 from each class) and is 543MB large (compressed). We then use only 10% portion of this dataset to train our model to save time. The final manipulated dataset contains 4000 pictures of cats and dogs (2000 cats, 2000 dogs) with 2000 pictures for training, 1000 for validation, and finally 1000 for testing. This section references with "Using convnets with small datasets"[13] (https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/5.2-using-convnets-with-small-datasets.ipynb).

In [1]:
```
import keras
keras.__version__
```

```
/anaconda3/lib/python3.6/site-packages/h5py/__init__.py:36: FutureWarnin
g: Conversion of the second argument of issubdtype from `float` to `np.fl
oating` is deprecated. In future, it will be treated as `np.float64 == n
p.dtype(float).type`.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

Out[1]:
```
'2.2.4'
```

In [4]:
```
from keras import optimizers
from keras import layers
from keras import models
```

Since we are visualizing an input that comes from the same training set which the input images are heavily intercorrelated -- 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 add a Dropout layer to our model, right before the densely-connected classifier:

In [5]:
```
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'])
```

In [9]:
```
base_dir = '/Users/shilinli/Documents/GitHub/CNN-Feature-Visualization/data/
train_dir = '/Users/shilinli/Documents/GitHub/CNN-Feature-Visualization/data
validation_dir = '/Users/shilinli/Documents/GitHub/CNN-Feature-Visualizatio
test_dir = '/Users/shilinli/Documents/GitHub/CNN-Feature-Visualization/data/
```

In [10]:
```
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.
```

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)
```

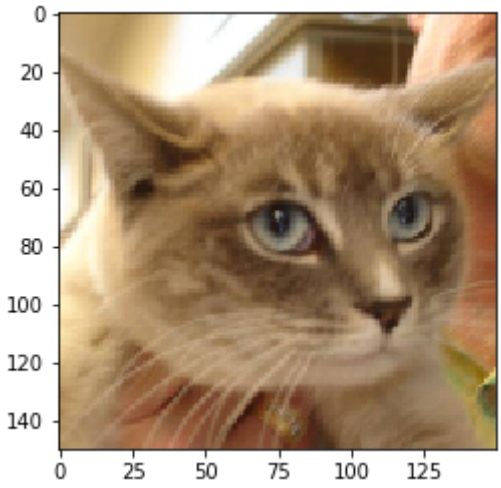
In [ ]:
```
model.save('cats_and_dogs_small_2.h5')
```

In [15]:
```
from keras.models import load_model

model = load_model('/Users/shilinli/Documents/GitHub/CNN-Feature-Visualizati
```

In [21]:
```
img_path = '/Users/shilinli/Documents/GitHub/CNN-Feature-Visualization/data/

# 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 [22]:
```
import matplotlib.pyplot as plt

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



In order to extract the feature maps we want to look at, we 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 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.

In [23]:
```
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)
```

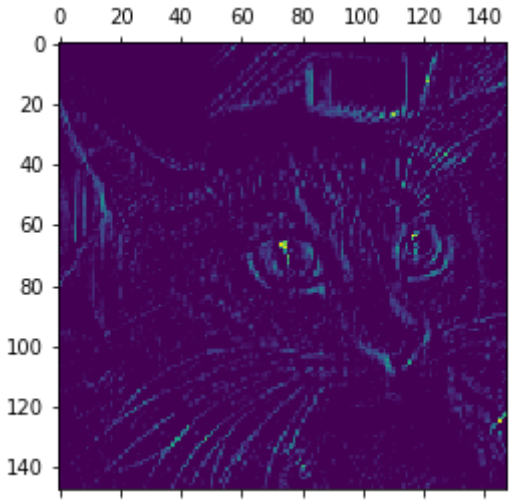
In [24]:
```
# This will return a list of 5 Numpy arrays:
# one array per layer activation
activations = activation_model.predict(img_tensor)
```

The activation of the first convolutional layer transfers the dimension [1,150,150,3] to [1,148,148,32] in which it becomes 32 channels. We visualize the 3rd channel below. It seems the activation has a clear shape of the cats that it learns the edge or shape of the input image. From here, we can conclude that the activation contains almost all the information from the input image.

In [25]:
```
first_layer_activation = activations[0]
print(first_layer_activation.shape)
```

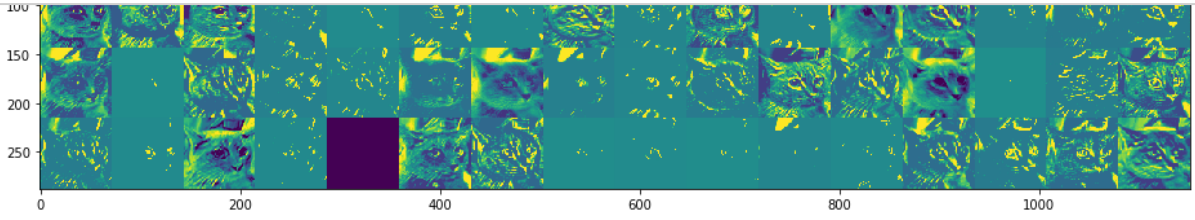
```
(1, 148, 148, 32)
```

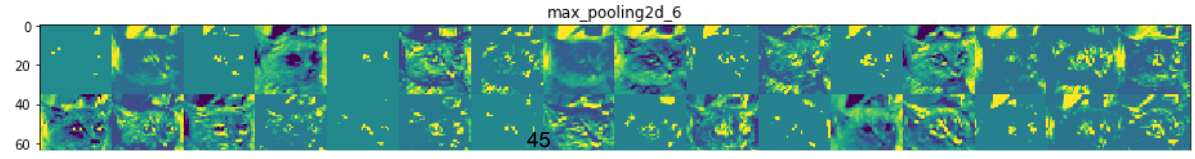
In [26]:
```
import matplotlib.pyplot as plt

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

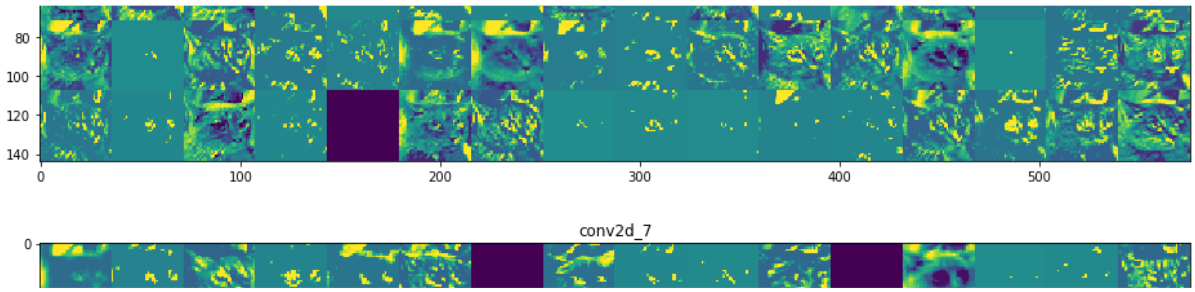


Next, we are visualizing every convolutional layer and pooling layer with all the channels displayed. In this way, we could observe the information learned by each layer and compare the difference between the activations. Before doing this experiment, we believe that each channel learned a small portion of the cats characteristics. The CNN in the training process records these characteristics. When an image of cat is fed, it activate different channel with the cats characteristics instead of dogs, which in return will drive the model to classify it as a cat.

In [27]:
```
import keras

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:8]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

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

    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                             :, :,
                                             col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

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

plt.show()
```







A few remarkable findings:

- 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, when 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. Our 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.

Part III Saliency and grad-CAM (ResNet50)

In this section, we are reviewing the saliency map for CNN model which is used to make classification decision, referenced in [14] (https://github.com/raghakot/keras-vis/blob/master/examples/resnet/attention.ipynb). To visualize activation over the final dense layer outputs, we need to switch the softmax activation out for linear since gradient of output node will depend on all the other node activations. We want to make the CNN model more transparent by visualizing the regions of input (two ouzels) that are 'important' for predictions from these models or visual explanations. The method Grad-CAM is refereced in this paper [15] (https://arxiv.org/pdf/1610.02391v1.pdf).

Grad-CAM Review

Given an image, we foward propagate the image through the model to obtain the raw class scores before softmax. The gradients are set to zero for all classes except the desired class, which is set to 1. This signal is then backpropagated to the rectified convolutional feature map of interest, where we can compute the coarse Grad-CAM localization (blue heatmap). Finally, we pointwise multiply the heatmap with guided backpropagation to get Guided Grad-CAM visualizations which are both high-resolution and class-discriminative.

ResNet50

The saliency map visualization is applied from model ResNet50, a convolutional neural network that is trained on more than a million images from the ImageNet database. The network is 50 layers deep and can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. The general framework of design is as follows:

- Use 3*3 filters mostly
- Down sampling with CNN layers with stride 2
- Global average pooling layer and a 1000-way fully-connected layer with Softmax in the end

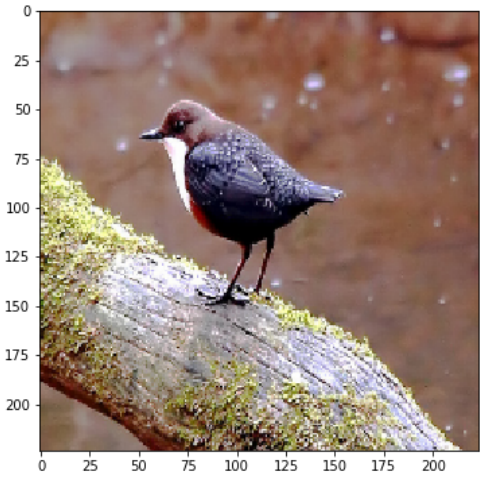
In [9]:
```
from keras.applications import ResNet50
from vis.utils import utils
from keras import activations

# Hide warnings on Jupyter Notebook
import warnings
warnings.filterwarnings('ignore')

# Build the ResNet50 network with ImageNet weights
model = ResNet50(weights='imagenet', include_top=True)

# Utility to search for layer index by name.
# Alternatively we can specify this as -1 since it corresponds to the last l
layer_idx = utils.find_layer_idx(model, 'fc1000')

# Swap softmax with linear
model.layers[layer_idx].activation = activations.linear
model = utils.apply_modifications(model)
```

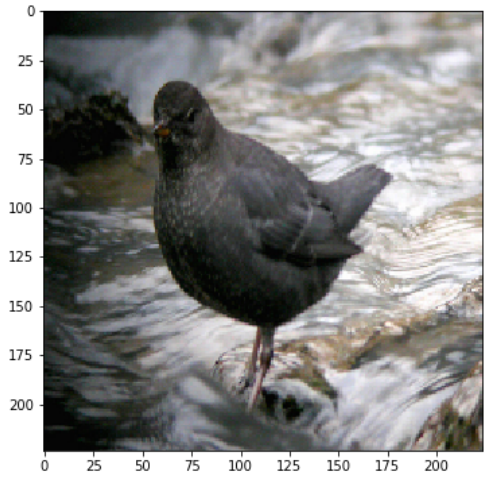
In [13]:
```
model.summary()
```

```
_____________________________
res3c_branch2b (Conv2D)         (None, 28, 28, 128)  147584      activa
tion_66[0][0]
_____________________________________________________________________
_____________________________
bn3c_branch2b (BatchNormalizati (None, 28, 28, 128)  512         res3c_
branch2b[0][0]
_____________________________________________________________________
_____________________________
activation_67 (Activation)      (None, 28, 28, 128)  0           bn3c_b
ranch2b[0][0]
_____________________________________________________________________
_____________________________
res3c_branch2c (Conv2D)         (None, 28, 28, 512)  66048       activa
tion_67[0][0]
_____________________________________________________________________
_____________________________
bn3c_branch2c (BatchNormalizati (None, 28, 28, 512)  2048        res3c_
branch2c[0][0]
_____________________________________________________________________
```

In [10]:
```
rom vis.utils import utils
rom matplotlib import pyplot as plt
atplotlib inline
t.rcParams['figure.figsize'] = (18, 6)

ng1 = utils.load_img('/Users/shilinli/Documents/GitHub/CNN-Feature-Visualizat
ng2 = utils.load_img('/Users/shilinli/Documents/GitHub/CNN-Feature-Visualizat

 ax = plt.subplots(1, 2)
[0].imshow(img1)
[1].imshow(img2)
```

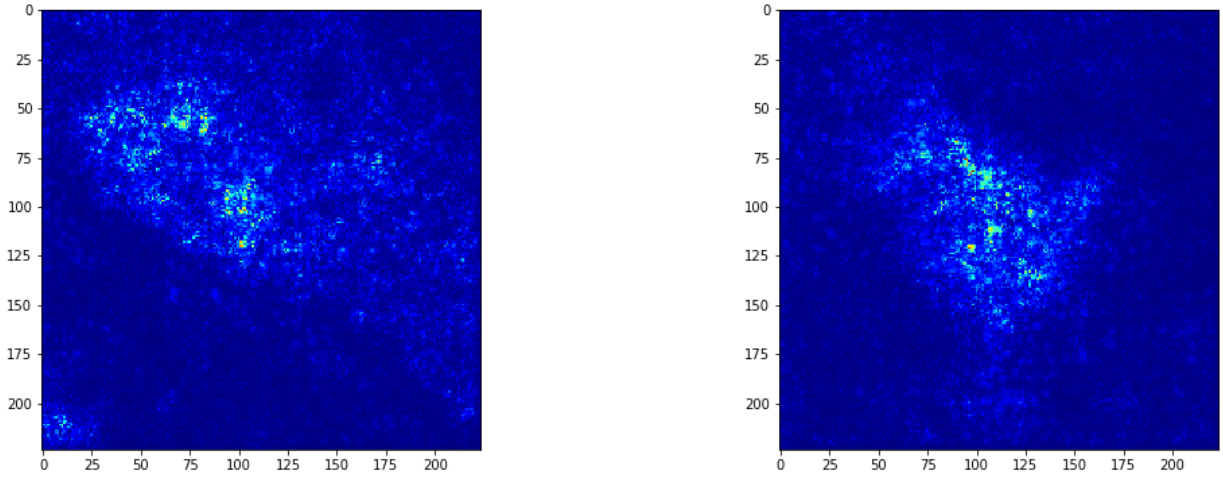
Out[10]: <matplotlib.image.AxesImage at 0x1c6b254d68>





After swapping the last dense layer to linear, we visualize the saliency map for the two blackbirds. Although the pictures are vague in detail texture, we could stil observe there is roughly a contour of the bird in which its color is lighter than the purple background. It seems the CNN correctly uses the lighter area for the classification problem.

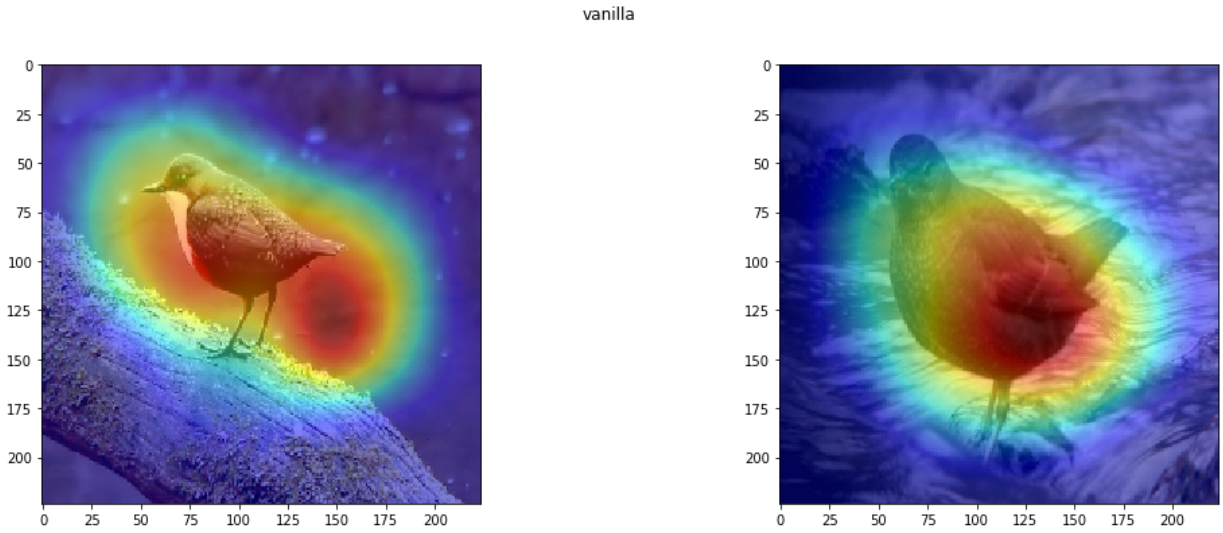
In [11]:
```
from vis.visualization import visualize_saliency, overlay
from vis.utils import utils
from keras import activations

# Utility to search for layer index by name.
# Alternatively we can specify this as -1 since it corresponds to the last
layer_idx = utils.find_layer_idx(model, 'fc1000')

f, ax = plt.subplots(1, 2)
for i, img in enumerate([img1, img2]):
    # 20 is the imagenet index corresponding to `ouzel`
    grads = visualize_saliency(model, layer_idx, filter_indices=20, seed_inp

    # visualize grads as heatmap
    ax[i].imshow(grads, cmap='jet')
```



In order to visualize clearer for detailed saliency map, we apply the visualization for layer 'res5c_branch2c'. The Grad-CAM method contains 'vanilla', 'guided' and 'relu' in its backpropogation process. In this visualization, we only focus on the vanilla propogation. From the pictures, we could directly see the yellow area covers mostly the birds bodies that the CNN identifies the class. For models that are trained using a relevant small dataset, the network might learned the information using the background. For example, the model could classify football because of the grass and sky whereas identify a pinpong because of the table and human hands. As a result, the learning process may yield unprecise predictions and the learning process is unsuccessful. Thus, we conclude that a relative large training samples are essential for a CNN model.

In [12]:
```
import numpy as np
import matplotlib.cm as cm
from vis.visualization import visualize_cam

penultimate_layer = utils.find_layer_idx(model, 'res5c_branch2c')

for modifier in [None]:
    plt.figure()
    f, ax = plt.subplots(1, 2)
    plt.suptitle("vanilla" if modifier is None else modifier)
    for i, img in enumerate([img1, img2]):
        # 20 is the imagenet index corresponding to `ouzel`
        grads = visualize_cam(model, layer_idx, filter_indices=20,
                              seed_input=img, penultimate_layer_idx=penultir
                              backprop_modifier=modifier)
        # Lets overlay the heatmap onto original image.
        jet_heatmap = np.uint8(cm.jet(grads)[..., :3] * 255)
        ax[i].imshow(overlay(jet_heatmap, img))
```

```
<Figure size 1296x432 with 0 Axes>
```



Reference List

[1] Olah, et al., "Feature Visualization", Distill, 2017. Referenced from https://distill.pub/2017/feature-visualization/ (https://distill.pub/2017/feature-visualization/)
[2] Kotikalapudi, et al., " Keras Visualization Toolkit", Keras-vis Documentation, 2018. Referenced from https://raghakot.github.io/keras-vis/vis.visualization/ (https://raghakot.github.io/keras-vis/vis.visualization/)
[3] Qin, Z., Yu, F., Liu, C., & Chen, X., "HOW CONVOLUTIONAL NEURAL NETWORKS SEE THE WORLD — A SURVEY OF CONVOLUTIONAL NEURAL
NETWORK VISUALIZATION METHODS", Mathematical Foundations of Computing, 2018. Referenced from arXiv:1804.11191v2 (https://arxiv.org/abs/1804.11191) [cs.CV]
[4] Nguyen, Y., Yosinski, J., & CluneReferenced, J.,"Multifaceted Feature Visualization: Uncovering the Different Types of Features Learned By Each Neuron in Deep Neural Networks", 2016, Referenced from arXiv:1602.03616v2 (https://arxiv.org/abs/1602.03616) [cs.NE]
[5] Chollet, F., "How convolutional neural networks see the world", Demo, 2016. Referenced from https://blog.keras.io/how-convolutional-neural-networks-see-the-world.html (https://blog.keras.io/how-convolutional-neural-networks-see-the-world.html)
[6] Kotikalapudi, R.,"What is Activation Maximization", 2017, Referenced from https://raghakot.github.io/keras-vis/visualizations/activation_maximization/ (https://raghakot.github.io/keras-vis/visualizations/activation_maximization/)
[7] Kotikalapudi, R., "Activation Maximization on MNIST", 2017. Referenced from https://github.com/raghakot/keras-vis/blob/master/examples/mnist/activation_maximization.ipynb (https://github.com/raghakot/keras-vis/blob/master/examples/mnist/activation_maximization.ipynb)
[8] Kotikalapudi, R.,"Activation Maximization on VGGNet", 2017. Referenced from https://github.com/raghakot/keras-vis/blob/master/examples/vggnet/activation_maximization.ipynb (https://github.com/raghakot/keras-vis/blob/master/examples/vggnet/activation_maximization.ipynb)
[9] Rajaraman, S., Silamut, K., Hossain, M.A., Ersoy, I., Maude, R.J., Jaeger, S., Thoma, G.R., Antani, S.K. "Understanding the learned behavior of customized convolutional neural networks toward malaria parasite detection in thin blood smear images", J Med Imaging (Bellingham), 2018 Jul;5(3):034501. doi: 10.1117/1.JMI.5.3.034501. Epub 2018 Jul 18. Referenced from https://lhncbc.nlm.nih.gov/publication/pub9809 (https://lhncbc.nlm.nih.gov/publication/pub9809)
[10] Kotikalapudi, R.,"visualize_activation", 2017. Referenced from https://raghakot.github.io/keras-vis/vis.visualization/#visualize_activation (https://raghakot.github.io/keras-vis/vis.visualization/#visualize_activation)
[11] Chollet, F., "Visualizing what convnets learn",deep-learning-with-python-notebooks, 2017. Feferenced from https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/5.4-visualizing-what-convnets-learn.ipynb (https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/5.4-visualizing-what-convnets-learn.ipynb) [12] Kaggle, "Dogs vs. Cats", 2013. Referenced from https://www.kaggle.com/c/dogs-vs-cats/data (https://www.kaggle.com/c/dogs-vs-cats/data)
[13] Chollet, F., "Using convnets with small datasets", 2017. Referened from https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/5.2-using-convnets-with-small-datasets.ipynb (https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/5.2-using-convnets-with-small-datasets.ipynb)
[14] Kotikalapudi, R., "Attention on ResNet50", 2018. Referenced from https://github.com/raghakot/keras-vis/blob/master/examples/resnet/attention.ipynb (https://github.com/raghakot/keras-vis/blob/master/examples/resnet/attention.ipynb)

[15] Selvaraju, R.R., Das, A., Vedantam, R., Cogswell, M., Parikh, D., Batra, D., "Grad-CAM: Why did you say that? Visual Explanations from Deep Networks via Gradient-based Localization" 2017, Referenced from