Using more sophisticated images with Convolutional Neural

In the previous lesson you saw how to use a CNN to make your recognition of the handwriting dig to the next level, recognizing real images of Cats and Dogs in order to classify an incoming image handwriting recognition made your life a little easier by having all the images be the same size an Real-world images aren't like that -- they're in different shapes, aspect ratios etc, and they're usuall

So, as part of the task you need to process your data -- not least resizing it to be uniform in shape

You'll follow these steps:

1. Explore the Example Data of Cats and Dogs
2. Build and Train a Neural Network to recognize the difference between the two
3. Evaluate the Training and Validation accuracy

Explore the Example Data

Let's start by downloading our example data, a .zip of 2,000 JPG pictures of cats and dogs, and e

NOTE: The 2,000 images used in this exercise are excerpted from the "Dogs vs. Cats" dataset ava images. Here, we use a subset of the full dataset to decrease training time for educational purpos

```
!wget --no-check-certificate \
  https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip \
  -O /tmp/cats_and_dogs_filtered.zip
```

```
--2020-03-15 18:41:48--  https://storage.googleapis.com/mledu-datasets/cats_a
Resolving storage.googleapis.com (storage.googleapis.com)... 74.125.20.128, 2
Connecting to storage.googleapis.com (storage.googleapis.com)|74.125.20.128|:
HTTP request sent, awaiting response... 200 OK
Length: 68606236 (65M) [application/zip]
Saving to: '/tmp/cats_and_dogs_filtered.zip'

/tmp/cats_and_dogs_ 100%[===================>]  65.43M  75.6MB/s    in 0.9s

2020-03-15 18:41:49 (75.6 MB/s) - '/tmp/cats_and_dogs_filtered.zip' saved [68
```

The following python code will use the OS library to use Operating System libraries, giving you acc allowing you to unzip the data.

```
import os
import zipfile

local_zip = '/tmp/cats_and_dogs_filtered.zip'

zip_ref = zipfile.ZipFile(local_zip, 'r')
```

```

zip_ref.extractall('/tmp')
zip_ref.close()
```

The contents of the .zip are extracted to the base directory `/tmp/cats_and_dogs_filtered`, whi subdirectories for the training and validation datasets (see the [Machine Learning Crash Course]() fo sets), which in turn each contain `cats` and `dogs` subdirectories.

In short: The training set is the data that is used to tell the neural network model that 'this is what etc. The validation data set is images of cats and dogs that the neural network will not see as par how badly it does in evaluating if an image contains a cat or a dog.

One thing to pay attention to in this sample: We do not explicitly label the images as cats or dogs. example earlier, we had labelled 'this is a 1', 'this is a 7' etc. Later you'll see something called an Im to read images from subdirectories, and automatically label them from the name of that subdirect directory containing a 'cats' directory and a 'dogs' one. ImageGenerator will label the images appr

Let's define each of these directories:

```
base_dir = '/tmp/cats_and_dogs_filtered'

train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')

# Directory with our training cat/dog pictures
train_cats_dir = os.path.join(train_dir, 'cats')
train_dogs_dir = os.path.join(train_dir, 'dogs')

# Directory with our validation cat/dog pictures
validation_cats_dir = os.path.join(validation_dir, 'cats')
validation_dogs_dir = os.path.join(validation_dir, 'dogs')

```

Now, let's see what the filenames look like in the `cats` and `dogs` `train` directories (file naming c directory):

```
train_cat_fnames = os.listdir( train_cats_dir )
train_dog_fnames = os.listdir( train_dogs_dir )

print(train_cat_fnames[:10])
print(train_dog_fnames[:10])
```

```
['cat.590.jpg', 'cat.658.jpg', 'cat.948.jpg', 'cat.226.jpg', 'cat.573.jpg', '
['dog.828.jpg', 'dog.168.jpg', 'dog.609.jpg', 'dog.446.jpg', 'dog.892.jpg', '
```

Let's find out the total number of cat and dog images in the `train` and `validation` directories:

```
print('total training cat images :', len(os.listdir(      train_cats_dir ) ))
print('total training dog images :', len(os.listdir(      train_dogs_dir ) ))

```

```
print('total validation cat images :', len(os.listdir( validation_cats_dir )
print('total validation dog images :', len(os.listdir( validation_dogs_dir )
```

```
total training cat images : 1000
total training dog images : 1000
total validation cat images : 500
total validation dog images : 500
```

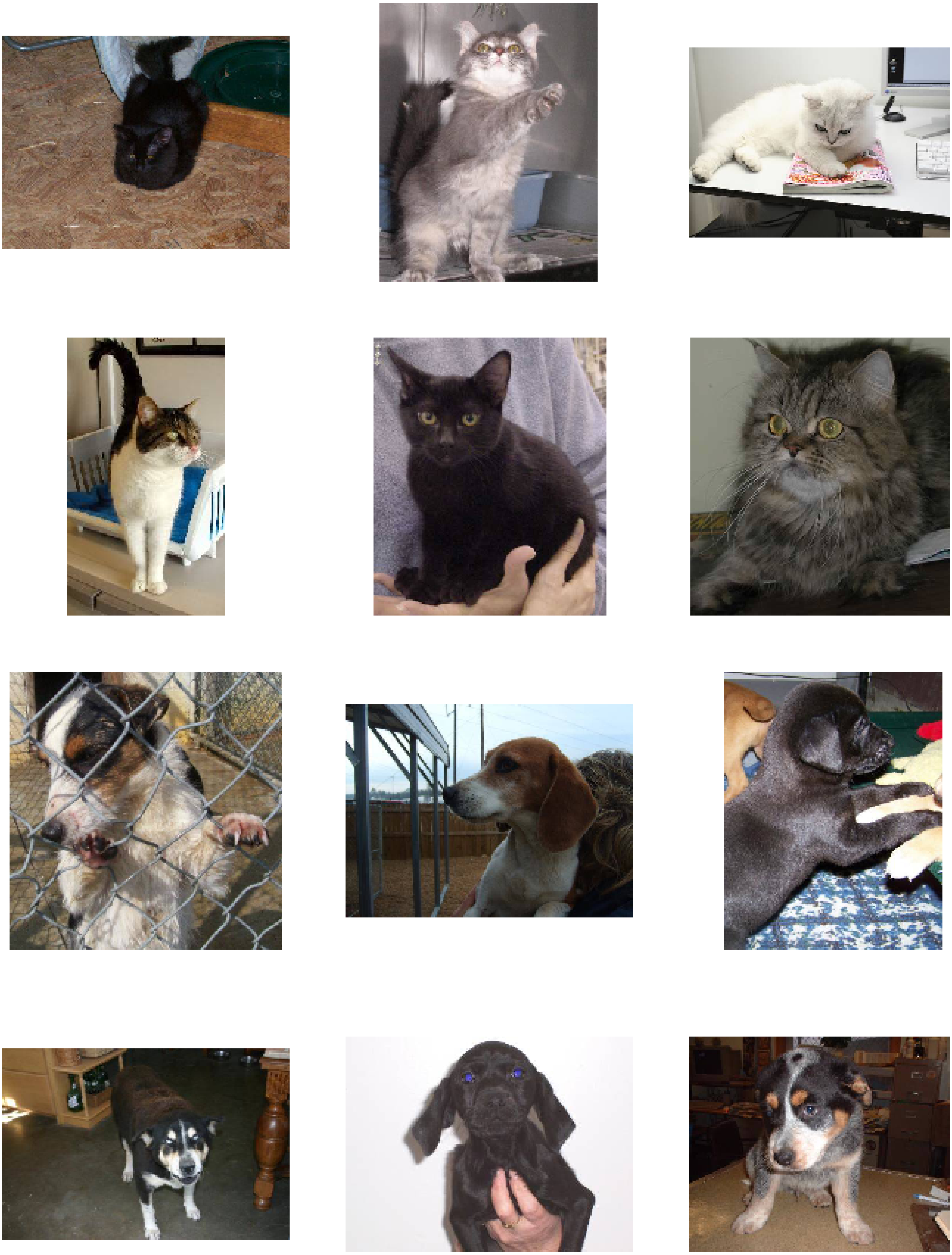
For both cats and dogs, we have 1,000 training images and 500 validation images.

Now let's take a look at a few pictures to get a better sense of what the cat and dog datasets look

```
%matplotlib inline

import matplotlib.image as mpimg
import matplotlib.pyplot as plt

# Parameters for our graph; we'll output images in a 4x4 configuration
nrows = 4
ncols = 4

pic_index = 0 # Index for iterating over images
```

Now, display a batch of 8 cat and 8 dog pictures. You can rerun the cell to see a fresh batch each

```
# Set up matplotlib fig, and size it to fit 4x4 pics
fig = plt.gcf()
fig.set_size_inches(ncols*4, nrows*4)

pic_index+=8

next_cat_pix = [os.path.join(train_cats_dir, fname)
                for fname in train_cat_fnames[ pic_index-8:pic_index]
               ]

next_dog_pix = [os.path.join(train_dogs_dir, fname)
                for fname in train_dog_fnames[ pic_index-8:pic_index]
               ]

for i, img_path in enumerate(next_cat_pix+next_dog_pix):
  # Set up subplot; subplot indices start at 1
  sp = plt.subplot(nrows, ncols, i + 1)
  sp.axis('Off') # Don't show axes (or gridlines)

  img = mpimg.imread(img_path)
  plt.imshow(img)

plt.show()

```



It may not be obvious from looking at the images in this grid, but an important note here, and a sig
is that these images come in all shapes and sizes. When you did the handwriting recognition exar
with. These are color and in a variety of shapes. Before training a Neural network with them you'll
the next section.

Ok, now that you have an idea for what your data looks like, the next step is to define the model th from these images

Building a Small Model from Scratch to Get to ~72% Accuracy

In the previous section you saw that the images were in a variety of shapes and sizes. In order to need them to be in a uniform size. We've chosen 150x150 for this, and you'll see the code that pre

But before we continue, let's start defining the model:

Step 1 will be to import tensorflow.

```
import tensorflow as tf
```

The default version of TensorFlow in Colab will soon switch to TensorFlow 2.x.
We recommend you [upgrade](https://) now or ensure your notebook will continue to use TensorFlow 1.x via the `%t`

Next we will define a Sequential layer as before, adding some convolutional layers first. Note the i example it was 28x28x1, because the image was 28x28 in greyscale (8 bits, 1 byte for color depth (24 bits, 3 bytes) for the color depth.

We then add a couple of convolutional layers as in the previous example, and flatten the final resu

Finally we add the densely connected layers.

Note that because we are facing a two-class classification problem, i.e. a *binary classification pro* [activation](https://), so that the output of our network will be a single scalar between 0 and 1, encoding the (as opposed to class 0).

```
model = tf.keras.models.Sequential([
    # Note the input shape is the desired size of the image 150x150 with 3 by
    tf.keras.layers.Conv2D(16, (3,3), activation='relu', input_shape=(150, 15
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    # Flatten the results to feed into a DNN
    tf.keras.layers.Flatten(),
    # 512 neuron hidden layer
    tf.keras.layers.Dense(512, activation='relu'),
    # Only 1 output neuron. It will contain a value from 0-1 where 0 for 1 cl
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow_cor
Instructions for updating:
If using Keras pass *_constraint arguments to layers.
```

The model.summary() method call prints a summary of the NN

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d (Conv2D)              (None, 148, 148, 16)      448       
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 74, 74, 16)        0         
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 72, 72, 32)        4640      
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 36, 36, 32)        0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 34, 34, 64)        18496     
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 17, 17, 64)        0         
_________________________________________________________________
flatten (Flatten)            (None, 18496)             0         
_________________________________________________________________
dense (Dense)                (None, 512)               9470464   
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 513       
=================================================================
Total params: 9,494,561
Trainable params: 9,494,561
Non-trainable params: 0
_________________________________________________________________
```

The "output shape" column shows how the size of your feature map evolves in each successive la the feature maps by a bit due to padding, and each pooling layer halves the dimensions.

Next, we'll configure the specifications for model training. We will train our model with the `binary` classification problem and our final activation is a sigmoid. (For a refresher on loss metrics, see tl use the `rmsprop` optimizer with a learning rate of `0.001`. During training, we will want to monitor

NOTE: In this case, using the [RMSprop optimization algorithm] is preferable to [stochastic gradient] learning-rate tuning for us. (Other optimizers, such as [Adam] and [Adagrad], also automatically ada work equally well here.)

```
from tensorflow.keras.optimizers import RMSprop

model.compile(optimizer=RMSprop(lr=0.001),
              loss='binary_crossentropy',
              metrics = ['acc'])
```

Data Preprocessing

Let's set up data generators that will read pictures in our source folders, convert them to `float32` our network. We'll have one generator for the training images and one for the validation images. O of size 150x150 and their labels (binary).

As you may already know, data that goes into neural networks should usually be normalized in so processing by the network. (It is uncommon to feed raw pixels into a convnet.) In our case, we wil pixel values to be in the `[0, 1]` range (originally all values are in the `[0, 255]` range).

In Keras this can be done via the `keras.preprocessing.image.ImageDataGenerator` class usi `ImageDataGenerator` class allows you to instantiate generators of augmented image batches (a `.flow_from_directory(directory)`. These generators can then be used with the Keras model inputs: `fit_generator`, `evaluate_generator`, and `predict_generator`.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# All images will be rescaled by 1./255.
train_datagen = ImageDataGenerator( rescale = 1.0/255. )
test_datagen  = ImageDataGenerator( rescale = 1.0/255. )

# --------------------
# Flow training images in batches of 20 using train_datagen generator
# --------------------
train_generator = train_datagen.flow_from_directory(train_dir,
                                                    batch_size=20,
                                                    class_mode='binary',
                                                    target_size=(150, 150))
# --------------------
# Flow validation images in batches of 20 using test_datagen generator
# --------------------
validation_generator =  test_datagen.flow_from_directory(validation_dir,
                                                         batch_size=20,
                                                         class_mode  = 'binar
                                                         target_size = (150,

```

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

Training

Let's train on all 2,000 images available, for 15 epochs, and validate on all 1,000 test images. (This

Do note the values per epoch.

You'll see 4 values per epoch -- Loss, Accuracy, Validation Loss and Validation Accuracy.

The Loss and Accuracy are a great indication of progress of training. It's making a guess as to the measuring it against the known label, calculating the result. Accuracy is the portion of correct gue measurement with the data that has not been used in training. As expected this would be a bit low section on overfitting later in this course.

```
history = model.fit_generator(train_generator,
                              validation_data=validation_generator,
                              steps_per_epoch=100,
                              epochs=15,
                              validation_steps=50,
                              verbose=2)
```

```
Epoch 1/15
Epoch 1/15
100/100 - 15s - loss: 0.8980 - acc: 0.5570 - val_loss: 0.6601 - val_acc: 0.60
Epoch 2/15
Epoch 1/15
100/100 - 8s - loss: 0.6553 - acc: 0.6590 - val_loss: 0.6102 - val_acc: 0.680
Epoch 3/15
Epoch 1/15
100/100 - 8s - loss: 0.5711 - acc: 0.7035 - val_loss: 0.5886 - val_acc: 0.707
Epoch 4/15
Epoch 1/15
100/100 - 8s - loss: 0.4989 - acc: 0.7645 - val_loss: 0.5833 - val_acc: 0.719
Epoch 5/15
Epoch 1/15
100/100 - 8s - loss: 0.4263 - acc: 0.7965 - val_loss: 0.6079 - val_acc: 0.712
Epoch 6/15
Epoch 1/15
100/100 - 8s - loss: 0.3507 - acc: 0.8520 - val_loss: 0.6109 - val_acc: 0.717
Epoch 7/15
Epoch 1/15
100/100 - 8s - loss: 0.2767 - acc: 0.8875 - val_loss: 0.7339 - val_acc: 0.717
Epoch 8/15
Epoch 1/15
100/100 - 8s - loss: 0.1884 - acc: 0.9240 - val_loss: 0.8298 - val_acc: 0.713
Epoch 9/15
Epoch 1/15
100/100 - 8s - loss: 0.1268 - acc: 0.9540 - val_loss: 1.1452 - val_acc: 0.718
Epoch 10/15
Epoch 1/15
100/100 - 8s - loss: 0.1003 - acc: 0.9735 - val_loss: 1.0673 - val_acc: 0.719
Epoch 11/15
Epoch 1/15
100/100 - 8s - loss: 0.0640 - acc: 0.9815 - val_loss: 1.3476 - val_acc: 0.713
Epoch 12/15
Epoch 1/15
100/100 - 8s - loss: 0.1177 - acc: 0.9820 - val_loss: 1.4038 - val_acc: 0.706
Epoch 13/15
Epoch 1/15
100/100 - 8s - loss: 0.0407 - acc: 0.9845 - val_loss: 1.9012 - val_acc: 0.713
Epoch 14/15
Epoch 1/15
100/100 - 8s - loss: 0.0803 - acc: 0.9845 - val_loss: 1.7132 - val_acc: 0.707
Epoch 15/15
Epoch 1/15
100/100 - 8s - loss: 0.0505 - acc: 0.9865 - val_loss: 2.2358 - val_acc: 0.703
```

Running the Model

Let's now take a look at actually running a prediction using the model. This code will allow you to will then upload them, and run them through the model, giving an indication of whether the object

```
import numpy as np

from google.colab import files
from keras.preprocessing import image

uploaded=files.upload()

for fn in uploaded.keys():

  # predicting images
  path='/content/' + fn
  img=image.load_img(path, target_size=(150, 150))

  x=image.img_to_array(img)
  x=np.expand_dims(x, axis=0)
  images = np.vstack([x])

  classes = model.predict(images, batch_size=10)

  print(classes[0])

  if classes[0]>0:
    print(fn + " is a dog")

  else:
    print(fn + " is a cat")

```

Choose Files download.jpeg

- **download.jpeg**(image/jpeg) - 6581 bytes, last modified: 16/03/2020 - 100% done

```
Saving download.jpeg to download.jpeg
[0.]
download.jpeg is a cat
```

Visualizing Intermediate Representations

To get a feel for what kind of features our convnet has learned, one fun thing to do is to visualize through the convnet.

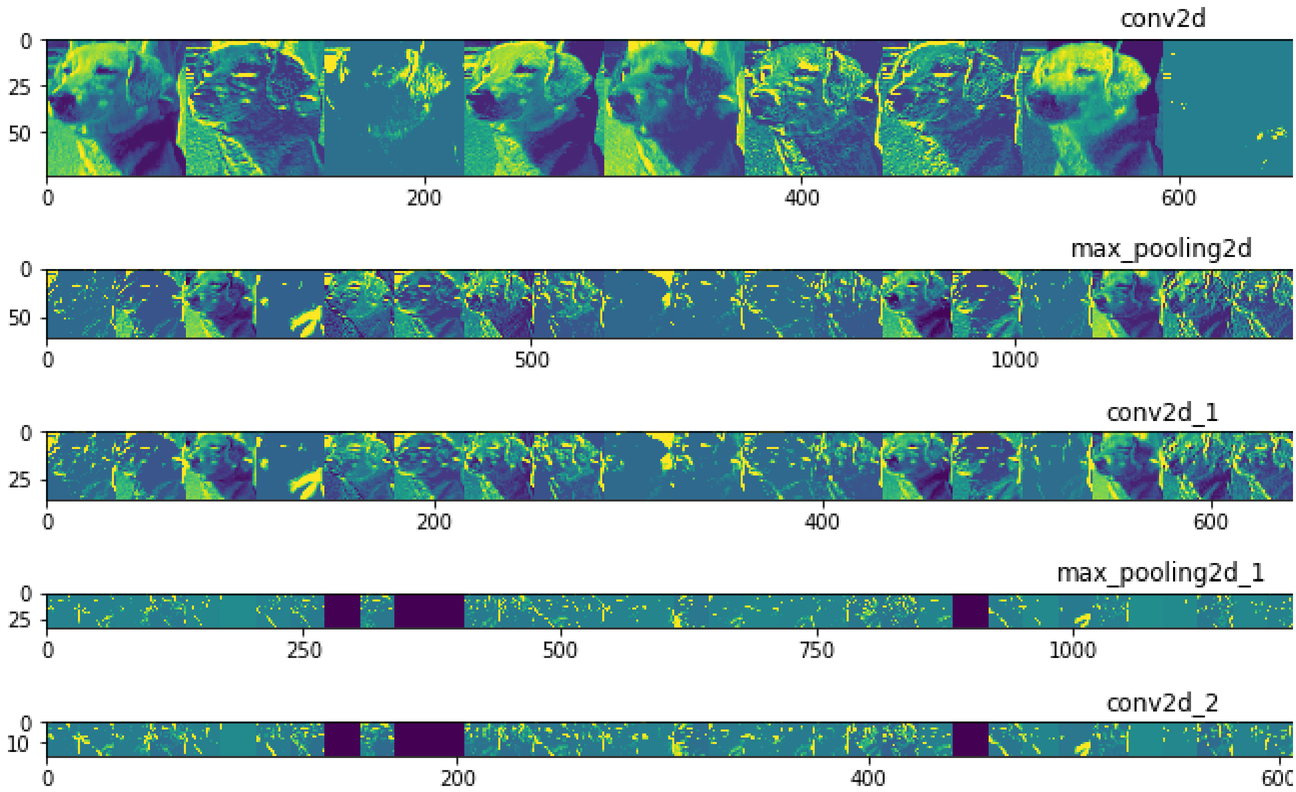
Let's pick a random cat or dog image from the training set, and then generate a figure where each the row is a specific filter in that output feature map. Rerun this cell to generate intermediate repre

```
import numpy as np
import random
from   tensorflow.keras.preprocessing.image import img_to_array, load_img

# Let's define a new Model that will take an image as input, and will output
# intermediate representations for all layers in the previous model after
# the first
```

```
# the first.
successive_outputs = [layer.output for layer in model.layers[1:]]

#visualization_model = Model(img_input, successive_outputs)
visualization_model = tf.keras.models.Model(inputs = model.input, outputs = s

# Let's prepare a random input image of a cat or dog from the training set.
cat_img_files = [os.path.join(train_cats_dir, f) for f in train_cat_fnames]
dog_img_files = [os.path.join(train_dogs_dir, f) for f in train_dog_fnames]

img_path = random.choice(cat_img_files + dog_img_files)
img = load_img(img_path, target_size=(150, 150))  # this is a PIL image

x   = img_to_array(img)                           # Numpy array with shape (1
x   = x.reshape((1,) + x.shape)                   # Numpy array with shape (1

# Rescale by 1/255
x /= 255.0

# Let's run our image through our network, thus obtaining all
# intermediate representations for this image.
successive_feature_maps = visualization_model.predict(x)

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

# -----------------------------------------------------------------------
# Now let's display our representations
# -----------------------------------------------------------------------
for layer_name, feature_map in zip(layer_names, successive_feature_maps):

  if len(feature_map.shape) == 4:

    #-------------------------------------------
    # Just do this for the conv / maxpool layers, not the fully-connected lay
    #-------------------------------------------
    n_features = feature_map.shape[-1]  # number of features in the feature m
    size       = feature_map.shape[ 1]  # feature map shape (1, size, size, n

    # We will tile our images in this matrix
    display_grid = np.zeros((size, size * n_features))

    #-------------------------------------------------
    # Postprocess the feature to be visually palatable
    #-------------------------------------------------
    for i in range(n_features):
      x  = feature_map[0, :, :, i]
      x -= x.mean()
      x /= x.std ()
      x *=  64
      x += 128
      x  = np.clip(x, 0, 255).astype('uint8')
      display_grid[:, i * size : (i + 1) * size] = x # Tile each filter into

    #-----------------
    # Display the grid
```

```
    # Display the grid
    #-----------------

    scale = 20. / n_features
    plt.figure( figsize=(scale * n_features, scale) )
    plt.title ( layer_name )
    plt.grid  ( False )
    plt.imshow( display_grid, aspect='auto', cmap='viridis' )
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:55: RuntimeWarni
```



As you can see we go from the raw pixels of the images to increasingly abstract and compact rep downstream start highlighting what the network pays attention to, and they show fewer and fewer zero. This is called "sparsity." Representation sparsity is a key feature of deep learning.

These representations carry increasingly less information about the original pixels of the image, b class of the image. You can think of a convnet (or a deep network in general) as an information di

Evaluating Accuracy and Loss for the Model

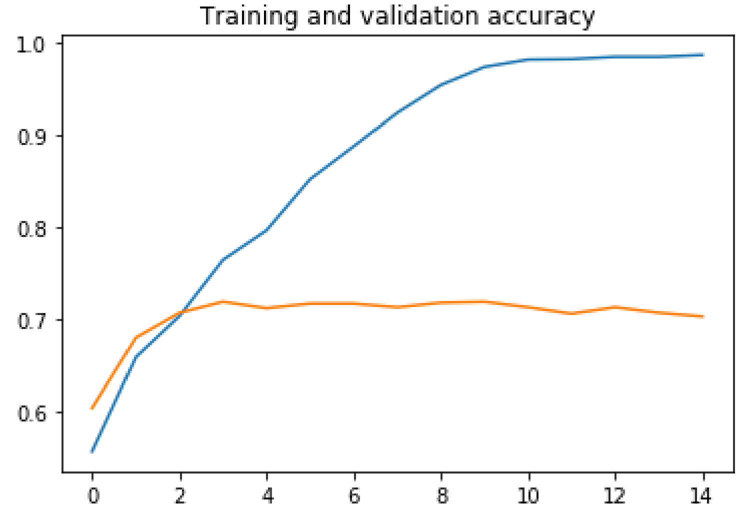
Let's plot the training/validation accuracy and loss as collected during training:

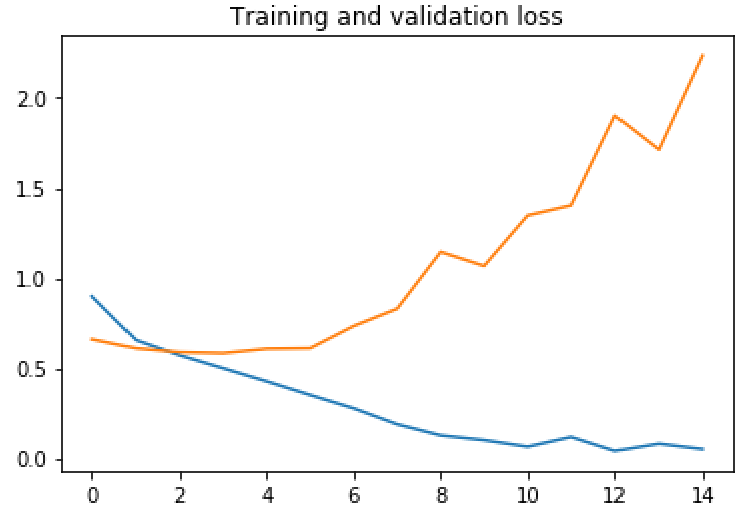
```
#-----------------------------------------------------------
# Retrieve a list of list results on training and test data
# sets for each training epoch
#-----------------------------------------------------------
acc      = history.history[     'acc' ]
val_acc  = history.history[ 'val_acc' ]
loss     = history.history[    'loss' ]
val_loss = history.history['val_loss' ]

epochs   = range(len(acc)) # Get number of epochs
```

```
epochs   = range(len(acc)) # Get number of epochs

#------------------------------------------------
# Plot training and validation accuracy per epoch
#------------------------------------------------
plt.plot  ( epochs,     acc )
plt.plot  ( epochs, val_acc )
plt.title ('Training and validation accuracy')
plt.figure()

#------------------------------------------------
# Plot training and validation loss per epoch
#------------------------------------------------
plt.plot  ( epochs,     loss )
plt.plot  ( epochs, val_loss )
plt.title ('Training and validation loss'   )
```

--VISUAL--

```
Text(0.5, 1.0, 'Training and validation loss')
```





As you can see, we are **overfitting** like it's getting out of fashion. Our training accuracy (in blue) ge accuracy (in green) stalls as 70%. Our validation loss reaches its minimum after only five epochs.

Since we have a relatively small number of training examples (2000), overfitting should be our nur model exposed to too few examples learns patterns that do not generalize to new data, i.e. when making predictions. For instance, if you, as a human, only see three images of people who are lun sailors, and among them the only person wearing a cap is a lumberjack, you might start thinking t lumberjack as opposed to a sailor. You would then make a pretty lousy lumberjack/sailor classifie

Overfitting is the central problem in machine learning: given that we are fitting the parameters of c make sure that the representations learned by the model will be applicable to data never seen bef specific to the training data?

In the next exercise, we'll look at ways to prevent overfitting in the cat vs. dog classification model

Clean Up

Before running the next exercise, run the following cell to terminate the kernel and free memory re

```
import os, signal

os.kill(     os.getpid() ,
           signal.SIGKILL
         )
```