```
!wget --no-check-certificate \
    https://storage.googleapis.com/laurencemoroney-blog.appspot.com/horse-or-
    -O /tmp/horse-or-human.zip
```

```
--2020-03-12 17:02:29--  https://storage.googleapis.com/laurencemoroney-blog.
Resolving storage.googleapis.com (storage.googleapis.com)... 172.217.214.128,
Connecting to storage.googleapis.com (storage.googleapis.com)|172.217.214.128
HTTP request sent, awaiting response... 200 OK
Length: 149574867 (143M) [application/zip]
Saving to: '/tmp/horse-or-human.zip'

/tmp/horse-or-human 100%[===================>] 142.65M  42.1MB/s    in 3.6s

2020-03-12 17:02:33 (39.8 MB/s) - '/tmp/horse-or-human.zip' saved [149574867/
```

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/horse-or-human.zip'
zip_ref = zipfile.ZipFile(local_zip, 'r')
zip_ref.extractall('/tmp/horse-or-human')
zip_ref.close()
```

The contents of the .zip are extracted to the base directory `/tmp/horse-or-human`, which in turn subdirectories.

In short: The training set is the data that is used to tell the neural network model that 'this is what like' etc.

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

Let's define each of these directories:

```
# Directory with our training horse pictures
train_horse_dir = os.path.join('/tmp/horse-or-human/horses')

# Directory with our training human pictures
train_human_dir = os.path.join('/tmp/horse-or-human/humans')
```

Your session crashed for an unknown reason. View runtime logs

Now, let's see what the filenames look like in the `horses` and `humans` training directories:

```
train_horse_names = os.listdir(train_horse_dir)
print(train_horse_names[:10])
```

```
train_human_names = os.listdir(train_human_dir)
print(train_human_names[:10])
```

```
['horse11-2.png', 'horse27-3.png', 'horse07-1.png', 'horse43-5.png', 'horse01
['human13-14.png', 'human03-20.png', 'human08-26.png', 'human15-28.png', 'hum
```

Let's find out the total number of horse and human images in the directories:

```
print('total training horse images:', len(os.listdir(train_horse_dir)))
print('total training human images:', len(os.listdir(train_human_dir)))
```

```
total training horse images: 500
total training human images: 527
```

Now let's take a look at a few pictures to get a better sense of what they look like. First, configure

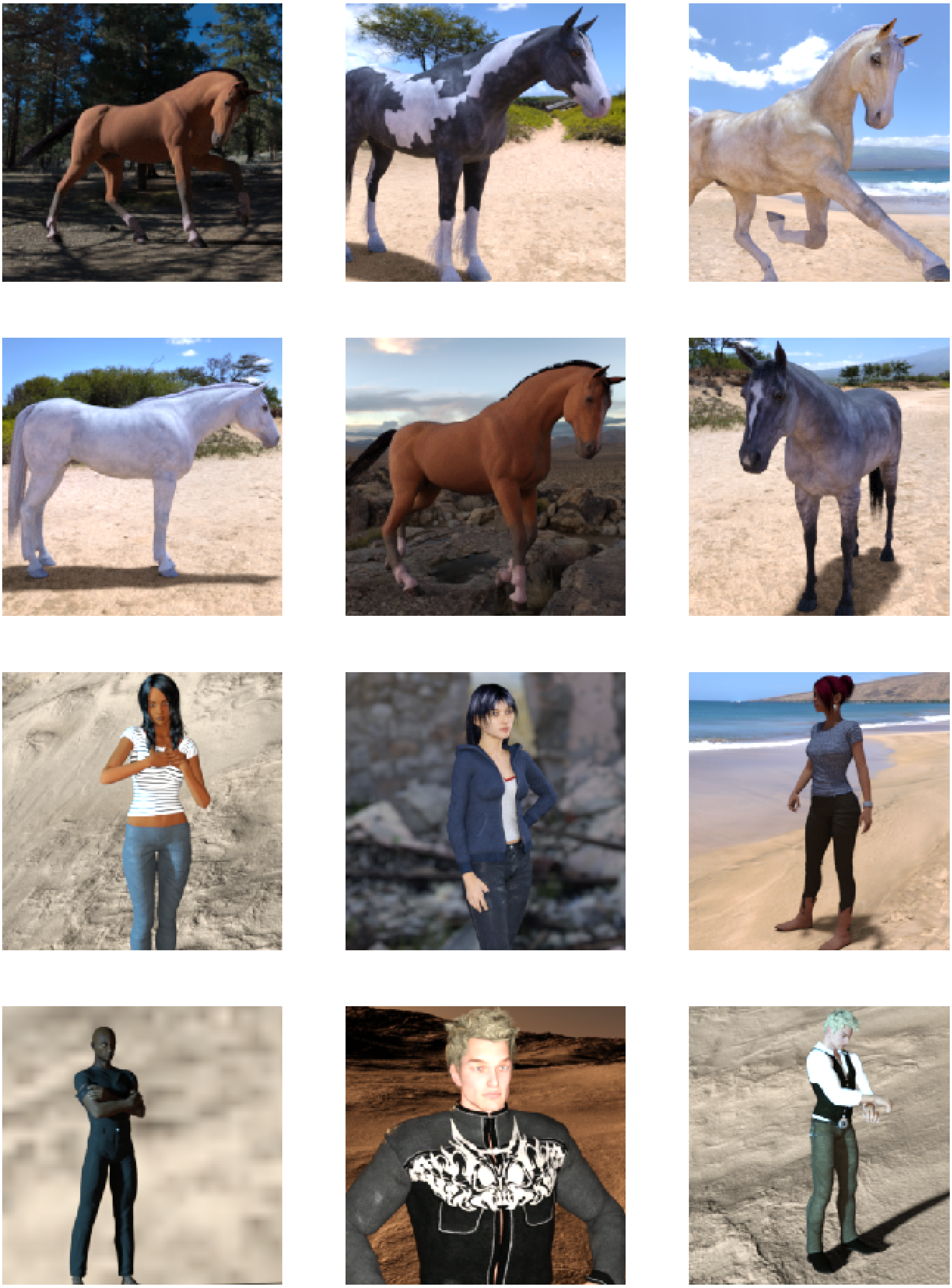
```
%matplotlib inline

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

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

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

Now, display a batch of 8 horse and 8 human pictures. You can rerun the cell to see a fresh batch

```
# 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_horse_pix = [os.path.join(train_horse_dir, fname)
                for fname in train_horse_names[pic_index-8:pic_index]]
next_human_pix = [os.path.join(train_human_dir, fname)
                for fname in train_human_names[pic_index-8:pic_index]]

for i, img_path in enumerate(next_horse_pix+next_human_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)
```

Your session crashed for an unknown reason. View runtime logs

```
  plt.imshow(img)

plt.show()
```



Your session crashed for an unknown reason. View runtime logs

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 now or ensure your notebook will continue to use TensorFlow 1.x via the `%tensorflow_version 1.x` magic: more info.

We then add convolutional layers as in the previous example, and flatten the final result to feed int

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, 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 300x300 with 3 by
    # This is the first convolution
    tf.keras.layers.Conv2D(16, (3,3), activation='relu', input_shape=(300, 30
    tf.keras.layers.MaxPooling2D(2, 2),
    # The second convolution
    tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    # The third convolution
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    # The fourth convolution
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    # The fifth convolution
    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

Your session crashed for an unknown reason. View runtime logs

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d (Conv2D)              (None, 298, 298, 16)      448       
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 149, 149, 16)      0         
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 147, 147, 32)      4640      
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 73, 73, 32)        0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 71, 71, 64)        18496     
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 35, 35, 64)        0         
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 33, 33, 64)        36928     
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 16, 16, 64)        0         
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 14, 14, 64)        36928     
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 7, 7, 64)          0         
_________________________________________________________________
flatten (Flatten)            (None, 3136)              0         
_________________________________________________________________
dense (Dense)                (None, 512)               1606144   
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 513       
=================================================================
Total params: 1,704,097
Trainable params: 1,704,097
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

                 metrics=['acc'])
```

Your session crashed for an unknown reason. View runtime logs

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow_cor
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
```

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 size 300x300 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=1/255)

# Flow training images in batches of 128 using train_datagen generator
train_generator = train_datagen.flow_from_directory(
        '/tmp/horse-or-human/',  # This is the source directory for training
        target_size=(300, 300),  # All images will be resized to 150x150
        batch_size=128,
        # Since we use binary_crossentropy loss, we need binary labels
        class_mode='binary')

```

```
Found 1027 images belonging to 2 classes.
```

Training

Let's train for 15 epochs -- this may take a few minutes to run.

Do note the values per epoch.

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

Your session crashed for an unknown reason. View runtime logs

```
history = model.fit_generator(
      train_generator,
      steps_per_epoch=8,
      epochs=15,
      verbose=1)
```

```
Epoch 1/15
8/8 [==============================] - 9s 1s/step - loss: 0.6901 - acc: 0.564
Epoch 2/15
8/8 [==============================] - 5s 654ms/step - loss: 0.7063 - acc: 0.
Epoch 3/15
8/8 [==============================] - 6s 743ms/step - loss: 0.7360 - acc: 0.
Epoch 4/15
8/8 [==============================] - 6s 739ms/step - loss: 0.5188 - acc: 0.
Epoch 5/15
8/8 [==============================] - 6s 741ms/step - loss: 0.2582 - acc: 0.
Epoch 6/15
8/8 [==============================] - 7s 849ms/step - loss: 0.1777 - acc: 0.
Epoch 7/15
8/8 [==============================] - 6s 735ms/step - loss: 0.1574 - acc: 0.
Epoch 8/15
8/8 [==============================] - 5s 641ms/step - loss: 0.2443 - acc: 0.
Epoch 9/15
8/8 [==============================] - 7s 843ms/step - loss: 0.2698 - acc: 0.
Epoch 10/15
8/8 [==============================] - 6s 732ms/step - loss: 0.0935 - acc: 0.
Epoch 11/15
8/8 [==============================] - 6s 758ms/step - loss: 0.3006 - acc: 0.
Epoch 12/15
8/8 [==============================] - 6s 746ms/step - loss: 0.1476 - acc: 0.
Epoch 13/15
8/8 [==============================] - 7s 848ms/step - loss: 0.0371 - acc: 0.
Epoch 14/15
8/8 [==============================] - 6s 750ms/step - loss: 0.0352 - acc: 0.
Epoch 15/15
8/8 [==============================] - 5s 637ms/step - loss: 0.0783 - acc: 0.
```

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=(300, 300))
  x = image.img_to_array(img)
```

Your session crashed for an unknown reason. View runtime logs

```
  classes = model.predict(images, batch_size=10)
  print(classes[0])
  if classes[0]>0.5:
    print(fn + " is a human")
```

```
  else:
    print(fn + " is a horse")

```

```
Using TensorFlow backend.
```

Choose Files finall.jpg

- **finall.jpg**(image/jpeg) - 7249 bytes, last modified: 28/11/2019 - 100% done

```
Saving finall.jpg to finall.jpg
[0.9988802]
finall.jpg is a human
```

Visualizing Intermediate Representations

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

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

```
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.
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 from the training set.
horse_img_files = [os.path.join(train_horse_dir, f) for f in train_horse_name
human_img_files = [os.path.join(train_human_dir, f) for f in train_human_name
img_path = random.choice(horse_img_files + human_img_files)

img = load_img(img_path, target_size=(300, 300))  # this is a PIL image
x = img_to_array(img)  # Numpy array with shape (150, 150, 3)
x = x.reshape((1,) + x.shape)  # Numpy array with shape (1, 150, 150, 3)

# Rescale by 1/255
x /= 255

# 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]
```

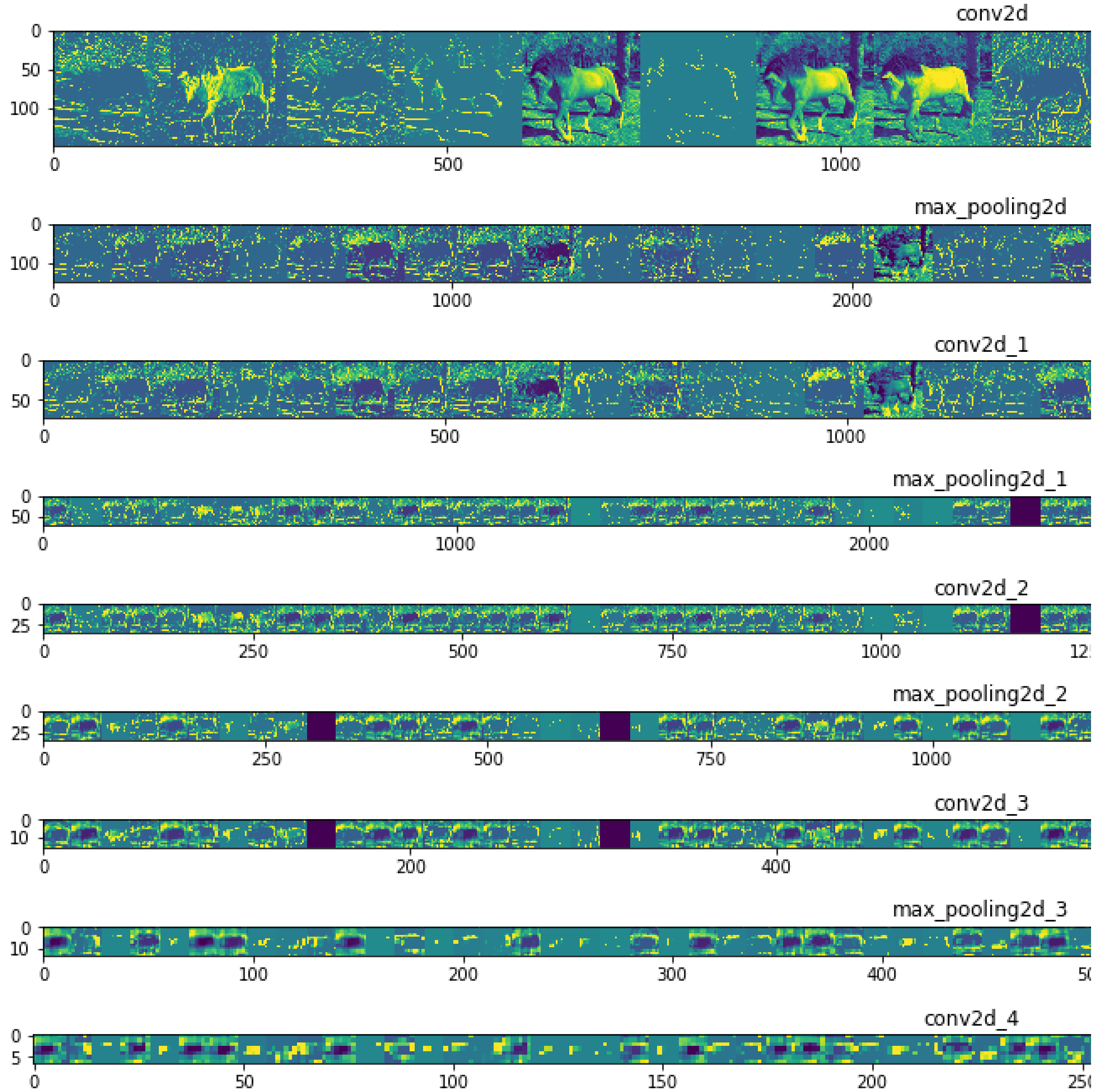
Your session crashed for an unknown reason. View runtime logs

```
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 feature map
    # The feature map has shape (1, size, size, n_features)
```

```
    size = feature_map.shape[1]
    # We will tile our images in this matrix
    display_grid = np.zeros((size, size * n_features))
    for i in range(n_features):
      # Postprocess the feature to make it visually palatable
      x = feature_map[0, :, :, i]
      x -= x.mean()
      x /= x.std()
      x *= 64
      x += 128
      x = np.clip(x, 0, 255).astype('uint8')
      # We'll tile each filter into this big horizontal grid
      display_grid[:, i * size : (i + 1) * size] = x
    # 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')
```

Your session crashed for an unknown reason. View runtime logs

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:43: 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

Clean Up

Your session crashed for an unknown reason. View runtime logs

ate the kernel and free memory re

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

--NORMAL--

Your session crashed for an unknown reason. View runtime logs