Lecture 05

CNNs, Visualizing Feature Maps

CSCI E-89

cscie-89 Deep Learning, Fall 2024

Zoran B. Djordjević

Objectives

- Learned features of neural networks are hardly interpretable.
- Several approaches for understanding and visualizing Convolutional Neural Networks (ConvNets) are developed. We briefly survey some of these approaches and related work.
- This lecture follows closely the second part of Chapter 5 of “Deep Learning with Python” by Francois Chollet, 1st Edition. Very similar content is presented in Chapter 9 of the 2nd Edition of the same book.
- Description of Functional API follows a section Chapter 10 of Aurelien Geron’s book. Similar text is presented in Chapter 7 of Chollet’s 2nd Edition.
- For more details about OpenCV, please consult Lab notes

Brief Introduction to OpenCV

OpenCV

- OpenCV (Open Source Computer Vision Library) is an open-source computer vision and machine learning software library. https://opencv.org
- OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products. OpenCV's license, allow businesses to utilize and modify the code.
- The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, extract 3D models of objects, produce 3D point clouds from stereo cameras, and stitch images together to produce a high-resolution image of an entire scene
- OpenCV can find similar images from an image database, remove red eyes from images taken using flash, follow eye movements, recognize scenery and establish markers to overlay with augmented reality, etc.
- OpenCV has more than 47-thousand-member user community. An estimated number of downloads is above 18 million. The library is used extensively in industry, research and by government.
- OpenCV is used by Google, Yahoo, Microsoft, Intel, IBM, Sony, Honda, Toyota and many other large corporation as well by many startups.
- OpenCV's uses span the range od applications including stitching street view images together, detecting intrusions in surveillance video, monitoring mine equipment, helping robots navigate and pick up objects, detection of swimming pool drowning accidents, running interactive art, checking runways, inspecting labels on products in factories, rapid face detection and many others.
- OpenCV has C++, Python, Java and MATLAB interfaces.
- OpenCV supports Windows, Linux, Android and Mac OS operating systems.

Installation of OpenCV

- Try first:

```
C:..\>  pip install --upgrade opencv-python
```

- If this does not work,
- On Windows, please follow:

https://www.solarianprogrammer.com/2016/09/17/install-opencv-3-with-python-3-on-windows/

- On Mac OS, please follow:

https://www.learnopencv.com/install-opencv3-on-macos/

- On Ubuntu, please follow:

https://www.pyimagesearch.com/2015/07/20/install-opencv-3-0-and-python-3-4-on-ubuntu/

Alternative Installation on Windows

- If `pip install opencv-python` command worked, you could do the following
- Verify your Python version. I have 3.10.0 on my machine.

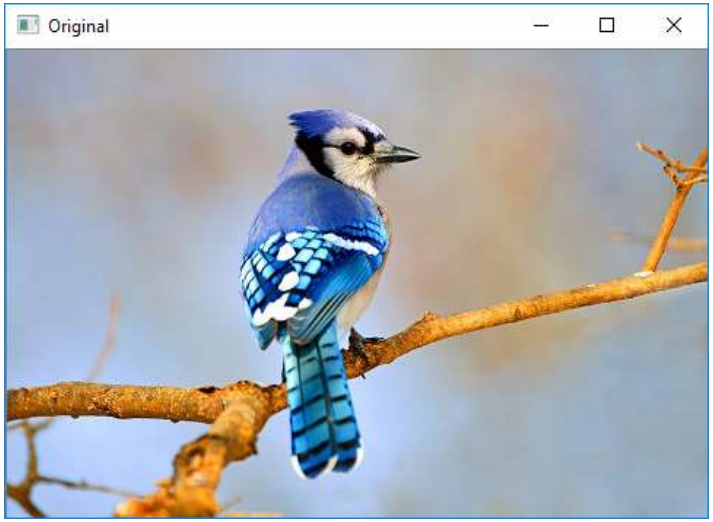
- Once installation is finished, open Python session and type:

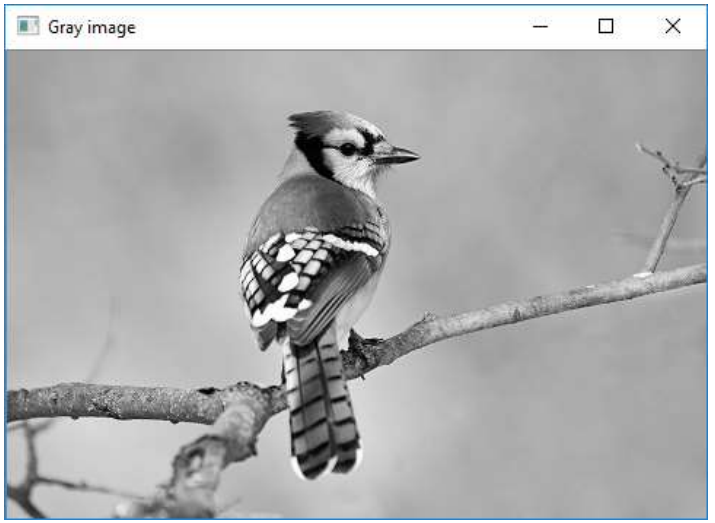
```
>>> import cv2
>>> print(cv2.__version__)
4.10.0
```

Small demo

```
import cv2
print(cv2.__version__)
image = cv2.imread("bluejay.jpg", 1)
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
cv2.imshow("Original", image)
cv2.imshow("Gray image", gray_image)
cv2.imwrite('gray_jay.jpg', gray_image)
cv2.waitKey(0) & 0xFF     # this is needed on a 64 bit machine
cv2.destroyAllWindows()
```

- If originally you had a color image of a blue jay, you will see two images. Gray image will be saved to your directory.





A few OpenCV commands

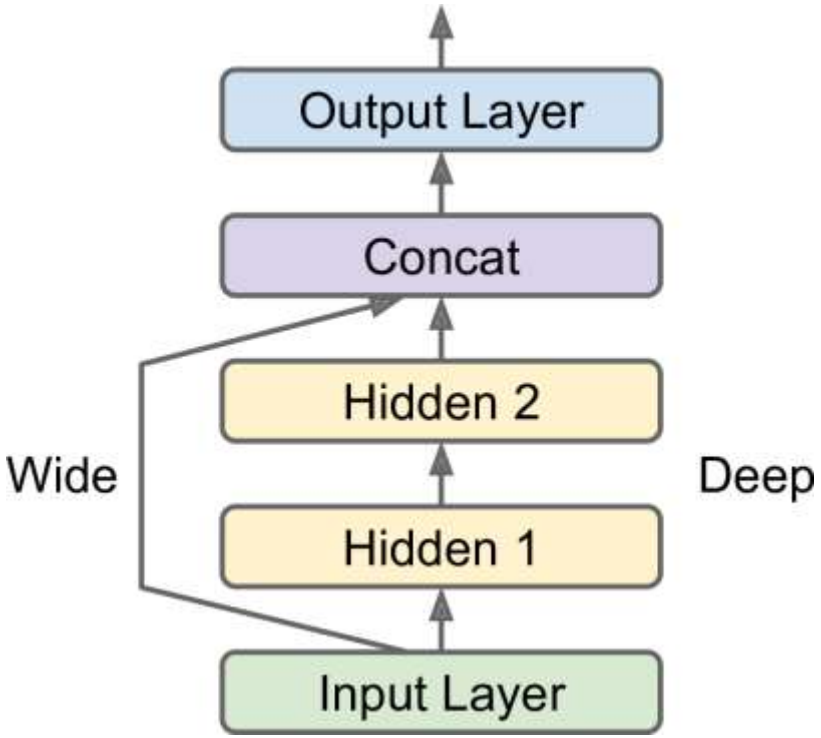
- `cv2.imread()` reads an image. The image should be in the working directory or a full path of image should be given.
- Second argument is a flag which specifies the way image should be read.
  - `cv2.IMREAD_COLOR`: Loads a color image. Any transparency of image will be neglected. It is the default flag. (1)
  - `cv2.IMREAD_GRAYSCALE`: Loads an image in grayscale mode (0)
  - `cv2.IMREAD_UNCHANGED` (-1): Loads an image as such including alpha channel
  - Instead of these three flags, simply pass integers: 0, 1, or -1 respectively.
- `cv2.imshow()` displays an image in a window. The window fits to the image size.
- First argument is a window name which is a string. Second argument is our image. You can create as many windows as you wish, but with different window names
- `cv2.waitKey() &0xFF` is a keyboard binding function. Its argument is the time in milliseconds. The function waits for specified milliseconds for any keyboard event. If you press any key in that time, the program continues. If **0** is passed, it waits indefinitely for a key stroke.
- `cv2.destroyAllWindows()` simply destroys all the windows we created. If you want to destroy any specific window, use the function `cv2.destroyWindow()` where you pass the exact window name as the argument.
- `cv2.imwrite('grayim.jpg', grayim)` saves the image in JPG format to the working directory
- `cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)` converts a color image to grayscale.

Functional API

Need for More Versatile Model

- As we have seen, the Sequential API is quite easy to use. However, although Sequential models are extremely common, it is sometimes useful to build neural networks with more complex topologies, or with multiple inputs or outputs. For this purpose, Keras offers the Functional API.
- One example of a nonsequential neural network is a Wide & Deep neural network.

- One such neural network architecture was introduced in a 2016 paper by Heng-Tze Cheng et al. (https://arxiv.org/abs/1606.07792)
- It connects all or part of the inputs directly to the output layer, as shown in Figure to the right.
- This architecture makes it possible for the neural network to learn both deep patterns (using the deep path) and simple rules (through the short path).
- In contrast, a regular Sequential network forces all the data to flow through the full stack of layers.



Wide & Deep neural network

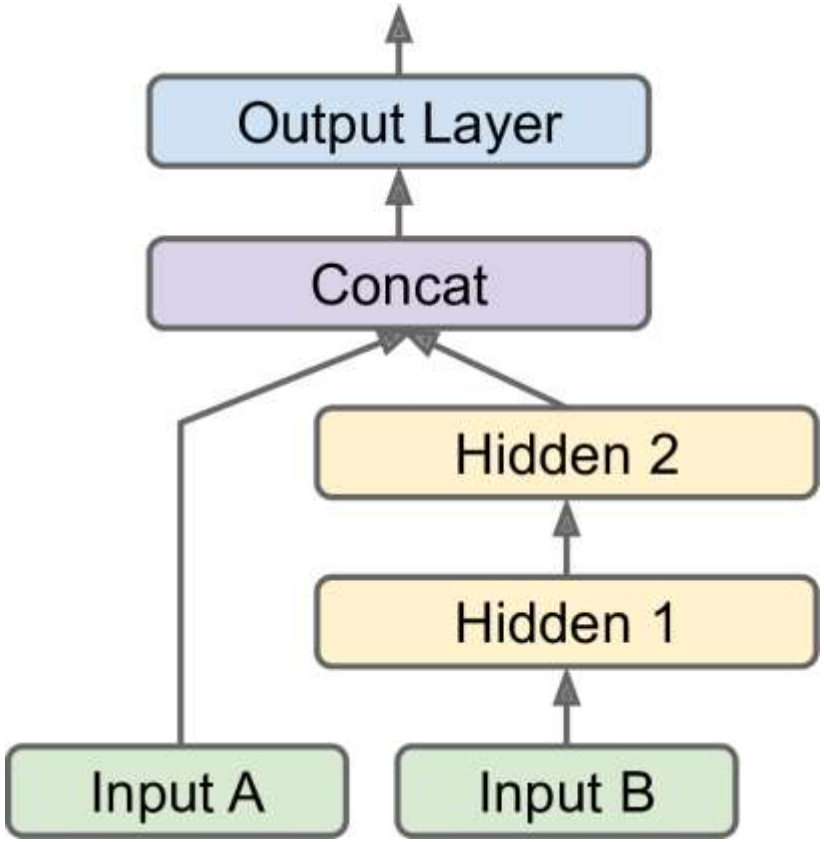
- Wide & Deep NN can be build using Keras Functional API in the following manner:

```
input_ = keras.layers.Input(shape=X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation="relu")(input_)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.Concatenate()([input_, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.Model(inputs=[input_], outputs=[output])
```

- First, we need to create an `Input` object. This is a specification of the `shape` and `dtype` of the inputs. A model may have multiple inputs. The name `input_` is used to avoid overshadowing Python's built-in `input()` function.
- Next, we create a `Dense` layer with 30 neurons, with the `ReLU` activation function. Notice that we call that layer like a function, passing `input_` as it's argument. This is why the model is called the Functional API.
- Note that we are just telling Keras how it should connect the layers together; no actual data is being processed yet.
- Then, we create a second hidden layer, `hidden2`, and again we call it as a function. We pass to layer `hidden2` the output of the first hidden layer, `hidden1`.
- Next, we create a `Concatenate` layer, and once again we use it like a function, to concatenate the original input, `input_`, and the output of the second hidden layer, `hidden2`.
- Then we create the `output` layer, with a single neuron and no activation function, and we call it like a function, passing to that layer the result of the concatenation.
- Lastly, we create a Keras `Model`, specifying which inputs and outputs it should use.

Model with two or more Inputs

- You might want to send a subset of the features through the wide path and a different subset (possibly overlapping) through the deep path (see Figure below)?
- In this case, one solution is to use multiple inputs. For example, we could send five features through the wide path, and six features through the deep path.



Model with two or more Inputs in Functional API

- Code using Functional API that implements previous architecture reads like:

```
input_A = keras.layers.Input(shape=[5], name="wide_input")
input_B = keras.layers.Input(shape=[6], name="deep_input")
hidden1 = keras.layers.Dense(30, activation="relu")(input_B)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_A, hidden2])
output = keras.layers.Dense(1, name="output")(concat)
model = keras.Model(inputs=[input_A, input_B], outputs=[output])
```

- You should name at least the most important layers, especially when the model gets a bit complex like this. Note that we specified `inputs=[input_A, input_B]` when creating the model. Now we can compile the model as usual, but when we call the `fit()` method, instead of passing a single input matrix `X_train`, we must pass a pair of matrices (`X_train_A, X_train_B`): one element per input. The same is true for `X_valid`, and also for `X_test` and `X_new` when calling `evaluate()| predict()`:

```
model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))
X_train_A, X_train_B = X_train[:, :5], X_train[:, 2:]
X_valid_A, X_valid_B = X_valid[:, :5], X_valid[:, 2:]
X_test_A, X_test_B = X_test[:, :5], X_test[:, 2:]
X_new_A, X_new_B = X_test_A[:3], X_test_B[:3]
history = model.fit((X_train_A, X_train_B), y_train, epochs=20,
validation_data=((X_valid_A, X_valid_B), y_valid))
mse_test = model.evaluate((X_test_A, X_test_B), y_test)
y_pred = model.predict((X_new_A, X_new_B)
```

- Inputs to `fit(), evaluate() & predict()` could be passed as dictionaries mapping the input names to the input values, like `{"wide_input": X_train_A, "deep_input": X_train_B}`.

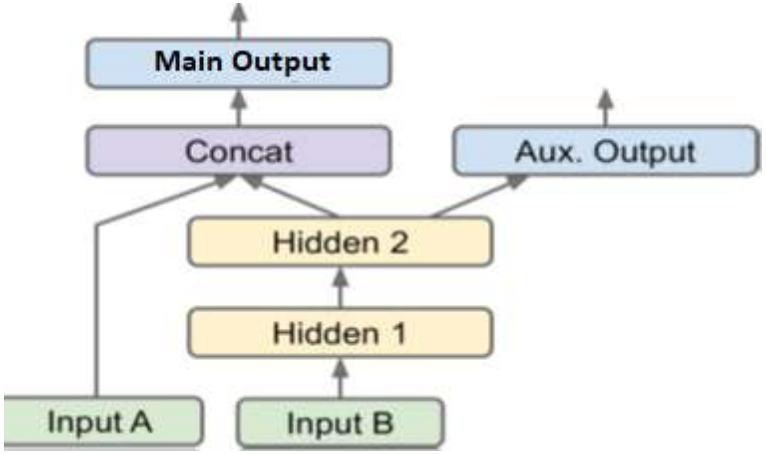
Use cases with multiple outputs

There are many use cases in which you may want to have multiple outputs:

- For instance, you may want to locate and classify the main object in a picture. This is both a regression task (finding the coordinates of the object's center, as well as its width and height) and a classification task (what object is discovered).
- Similarly, you may have multiple independent tasks based on the same data. You could train one neural network per task, but in many cases you will get better results on all tasks by training a single neural network with one output per task. This is because the neural network can learn features in the data that are useful across tasks. For example, you could perform *multitask classification* on pictures of faces, using one output to classify the person's facial expression (smiling, surprised, etc.) and another output to identify whether they are wearing glasses or not.
- Another use case is as a regularization technique (i.e., a training constraint whose objective is to reduce overfitting and thus improve the model's ability to generalize). For example, you may want to add some auxiliary outputs in a neural network architecture to ensure that the underlying part of the network learns something useful on its own, without relying on the rest of the network.

Handling Multiple Outputs

- In the architecture to the right, we are adding and auxiliary output by connecting it to the appropriate layer and by adding the output to our model's list of outputs.
- The following code builds this network




```
# Same as above, up to the main output layer
output = keras.layers.Dense(1, name="main_output")(concat)
aux_output = keras.layers.Dense(1, name="aux_output")(hidden2)
model = keras.Model(inputs=[input_A, input_B], outputs=[output, aux_output])
```

- Each output will need its own loss function. When we compile the model, we should pass a list of losses (if we pass a single loss, Keras will assume that the same loss must be used for all outputs).
- By default, Keras will compute all these losses and simply add them up to get the final loss used for training. If we want to give the main output's loss a greater weight, we pass `loss_weights` which weights outputs differently when compiling the model.

```
model.compile(loss=["mse", "mse"], loss_weights=[0.9, 0.1],
optimizer="sgd")
```

Labels and Losses for Multiple outputs

- When we train the model, we need to provide labels for each output. Instead of passing `y_train`, we need to pass `(y_train_A, y_train_B)` (and the same goes for `y_valid` and `y_test`):
- If the main output and the auxiliary output predict the same targets (labels), they should use the same labels.

```
history = model.fit([X_train_A, X_train_B], [y_train, y_train], epochs=20,
        validation_data=([X_valid_A, X_valid_B], [y_valid, y_valid]))
```

- When we evaluate the model, Keras will return the total loss, as well as all the individual losses:

```
total_loss, main_loss, aux_loss = model.evaluate(
    [X_test_A, X_test_B], [y_test, y_test])
```

- Similarly, the `predict()` method will return predictions for each output:

```
y_pred_main, y_pred_aux = model.predict([X_new_A, X_new_B])
```

- As you can see, you can build easily different architectures with the Functional API.

Saving and Restoring Models

Saving and Restoring a Model

- In Keras, either using Sequential API or Functional API, saving a trained model is simple

```
model = keras.models.Sequential([...]) # or keras.Model([...])
model.compile([...])
history=model.fit([...])
model.save("my_keras_model.h5")
```

- Keras uses the HDF5 format to save both the model's architecture (including every layer's hyperparameters) and the values of all the model's trainable parameters for every layer (e.g., connection weights and biases). Method `model.save()` also saves the optimizer (including its hyperparameters and any state it may have).
- You typically operate a script that trains a model and saves it, and one or more scripts (or web services) that load the model and use it to make predictions.
- Loading the model is just as easy:

```
model = keras.models.load_model("my_keras_model.h5")
```

- This works whether using the Sequential API or the Functional API.
- Once you load a saved model, you can use it to make predictions or to continue training.

Callbacks

Callbacks

- What if training lasts several hours? This is quite common, especially when training on large datasets. In this case, you should not only save your model at the end of training, but also save checkpoints at regular intervals during training, to avoid losing everything if your code or computer crashes.
- We use `callbacks` to tell the `fit()` method to save `checkpoints`?

Using Callbacks

- The `fit()` method accepts a callbacks argument that lets you specify a list of objects that Keras will call at the start and at the end of training, at the start and end of each epoch, and even before and after processing of each batch.
- For example, the `ModelCheckpoint` callback saves checkpoints of your model at regular intervals during training, by default at the end of each epoch. We build and compile the model, then define callbacks and register them with `fit()` method

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5")
history = model.fit(X_train, y_train, epochs=10, callbacks=[checkpoint_cb])
```

- If you use a validation set during training, you can set `save_best_only=True` when creating the `ModelCheckpoint`. In this case, the callback will only save your model when its performance on the validation set is the best so far. This way, you do not need to worry about training for too long and overfitting the training set: simply restore the last model saved after training, and this will be the best model on the validation set.
- The following code is a simple way to save and restore the best model

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5",
                                                save_best_only=True)
history = model.fit(X_train, y_train, epochs=10,
                            validation_data=(X_valid, y_valid),
                            callbacks=[checkpoint_cb])
model = keras.models.load_model("my_keras_model.h5") # roll back to best model
```

`EarlyStopping` callback

- We implement early stopping with the `EarlyStopping` callback.
- `EarlyStopping` callback interrupts the training when it measures no progress on the validation set for a specified number of epochs (argument `patience`). It optionally rolls back to the best model. You can combine both callbacks to save checkpoints of your model (in case your computer crashes) and interrupt training early when there is no more progress (to avoid wasting time and resources):

```
checkpoint_cb =
keras.callbacks.ModelCheckpoint("my_keras_model.h5",
                                save_best_only=True)
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10,
restore_best_weights=True)
history = model.fit(X_train, y_train, epochs=100,
validation_data=(X_valid, y_valid),
callbacks=[checkpoint_cb, early_stopping_cb])
```

- The number of epochs can be set to a large value since training will stop automatically when there is no more progress. In this case, there is no need to restore the best model saved because the `EarlyStopping` callback will keep track of the best weights and restore them for you at the end of training.
- If you need extra control, you can write your own custom callbacks

Visualization of Feature Maps

Objectives

- We will learn how to train a new model from scratch on small data set.

- We will start by naively training a small convnet on 2000 training samples, without any regularization, to set a baseline for what can be achieved. This will get us to a classification accuracy of 71%. At that point, our main issue will be overfitting.
- In next lecture, we will introduce *Data Augmentation*, a powerful technique for mitigating overfitting in computer vision and DL training. By leveraging data augmentation, we will improve our network to reach an accuracy of 82%.
- Also, in the next lecture, will review two more essential techniques for applying deep learning to small datasets: *Feature Extraction with a Pre-trained network* (this will get us to an accuracy of 90% to 93%) and *Fine-tuning a Pre-trained network* (this will get us to our final accuracy of 95%).

- All together, we will examine four strategies:
  1. Training a small model from scratch (this lecture)
  2. Data-augmentation, (next lecture)
  3. Feature Extraction from a pre-trained model, (next lecture), and
  4. Fine-tuning a Pre-trained model (next lecture).

- These techniques will constitute your toolbox for training DL models on small datasets.

- In this lecture, we will implement strategy #1 learn how to visualize intermediate activations. We will demonstrate strategies 2, 3 and 4 in the next lecture.

Downloading Data

- We will use Kaggle's dogs vs. cats dataset from https://www.kaggle.com/c/dogs-vs-cats/data
- Data come in a `test.zip` and `train.zip` archives. Once opened, folders contains some 25,000 images of cute dogs and cats. Total volume of the dataset is some 880 MB
- After downloading and uncompressing the zip files, we will start with `train` images and create a new dataset containing three subsets: a `training` set with 1000 samples of each class, a `validation` set with 500 samples of each class, and finally a `test` set with 500 samples of each class.
- Code for creation of this new dataset follows:

```
import os, shutil
# The directory where the original dataset was uncompressed
original_dataset_dir = 'dogs-vs-cats\\train'
# The directory for the smaller dataset
base_dir = 'cats_and_dogs_small'
os.mkdir(base_dir)
# Directories for small training,validation and test splits
train_dir = os.path.join(base_dir, 'train')
os.mkdir(train_dir)
validation_dir=os.path.join(base_dir,'validation')
os.mkdir(validation_dir)
test_dir = os.path.join(base_dir, 'test')
os.mkdir(test_dir)
```

Building Directories for Small Dataset

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

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

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

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

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

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

Populating a Small Dataset

```
# Copy first 1000 cat images to train_cats_dir
fnames = ['cat.{}.jpg'.format(i) for i in range(1000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(train_cats_dir, fname)
    shutil.copyfile(src, dst)
# Copy next 500 cat images to validation_cats_dir
fnames = ['cat.{}.jpg'.format(i) for i in range(1000, 1500)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(validation_cats_dir, fname)
    shutil.copyfile(src, dst)
# Copy next 500 cat images to test_cats_dir
fnames = ['cat.{}.jpg'.format(i) for i in range(1500, 2000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(test_cats_dir, fname)
    shutil.copyfile(src, dst)
# Copy first 1000 dog images to train_dogs_dir
fnames = ['dog.{}.jpg'.format(i) for i in range(1000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(train_dogs_dir, fname)
    shutil.copyfile(src, dst)
# Copy next 500 dog images to validation_dogs_dir
fnames = ['dog.{}.jpg'.format(i) for i in range(1000, 1500)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(validation_dogs_dir, fname)
    shutil.copyfile(src, dst)
# Copy next 500 dog images to test_dogs_dir
fnames = ['dog.{}.jpg'.format(i) for i in range(1500, 2000)]
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(test_dogs_dir, fname)
    shutil.copyfile(src, dst)
```

Structure of CNN

- We will use a typical structure of CNNs, i.e., a stack of alternated Conv2D (with `Relu` activation) and `MaxPooling2D` layers.
- We are dealing with bigger images (150x150) and we will make our network accordingly larger: it will have one more `Conv2D` + `MaxPooling2D` stage. This serves both to augment the capacity of the network and to reduce the size of the feature maps, so that they are not overly large when we reach the Flatten layer. We will end up with feature maps of size 7x7 right before the Flatten layer.
- The depths of the feature maps are progressively increased, from 32 to 128, while the size of the feature maps is decreasing (from 148x148 to 7x7).

```
from tensorflow.keras import layers
from tensorflow.keras import models
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
        input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Dimensions of the Feature Map

- Model summary gives us the dimensions of the feature maps:

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 148, 148, 32)      896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 74, 74, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 72, 72, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 36, 36, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 34, 34, 128)       73856
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 17, 17, 128)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 15, 15, 128)       147584
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 7, 7, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 6272)              0
_________________________________________________________________
dense (Dense)                (None, 512)               3211776
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 513
=================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
```

Compilation

- In the compilation step, we use the `RMSprop` optimizer.
- Since we ended our network with a single sigmoid unit, we use `binary_crossentropy` as the loss function.
- We are performing a binary classification. The result is either a cat or a dog?

```
from tensorflow.keras import optimizers

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(learning_rate=1e-4),
              metrics=['acc'])
```

Python Generators

Side note: Python Generators

- Generator is a function which `yield`-s items rather than `return`-ing a list. Generator behaves like an iterator. A generator can be used in a loop.

```
def infinite_sequence():
        num = 0
        while True:
                yield num
                print("restart here")
                num += 1
>>> gen = infinite_sequence()
>>> next(gen)
0
>>> next(gen)
restart here
1
>>> next(gen)
restart here
2
>>> next(gen)
restart here
3
```

```
def firstn(n=10):
        num = 0
        while num < n:
                yield num
                num += 1

>>for i in firstn(6):
      print(i)

0
1
2
3
4
5
```

- Python generators are defined as normal functions, but with a `yield` statement instead of a `return` statement. Function containing at least one `yield` statement (it may contain other `yield` or `return` statements), is a generator function.
- Both `yield` and `return` will return some value from a function.
- While a `return` statement terminates a function entirely, `yield` statement pauses the function saving all its states and later, on subsequent calls, continues from where it stopped.

Data Preprocessing
and Training a Small Model from Scratch

Data Preprocessing

- Data should be formatted into appropriately pre-processed floating-point tensors before fed into the network.
- Currently, the data is a collection of JPEG files. The steps for getting data into the network are roughly:
  - Read the picture files.
  - Decode the JPEG content to RBG grids of pixels.
  - Convert these into floating point tensors.
  - Rescale the pixel values (between 0 and 255) to the [0, 1] interval (Neural networks prefer to deal with small input values).
- Keras has utilities to perform those steps automatically.
- Keras has a module with image processing helper tools, located at `tf.keras.preprocessing.image`.
- This module contains the class `ImageDataGenerator` which allows a quick set up Python generators that can automatically turn image files on a disk into batches of pre-processed tensors.

Transforming JPEGs to batches of RGB Tensors

```
from tensorflow.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.
```

- This code yields batches of 150x150 RGB images (shape (20, 150, 150, 3)) and binary labels (shape (20,)).

Limit Batch Creation, `break`

- 20 is the number of samples in each batch (the batch size).

```
for data_batch, labels_batch in train_generator:
    print('data batch shape:', data_batch.shape)
    print('labels batch shape:', labels_batch.shape)
    break  # this break takes you out of the endless loop

data batch shape: (20, 150, 150, 3)
labels batch shape: (20,).
```

Fitting `model` to `data`, `fit()`

- We will fit our model to the data using the `fit()` method.
- `fit()` method expects as its first argument a Python generator that yields mini-batches of inputs and targets indefinitely, like the `train_generator` or `validation_generator`.
- Because the data is being generated endlessly, the Keras `model` needs to know how many samples to draw from the generator before declaring an epoch over.
- This is the role of the `steps_per_epoch` argument: after having drawn `steps_per_epoch` batches from the generator—that is, after having run for `steps_per_epoch` gradient descent steps—the fitting process will go to the next epoch. In this case, batches are 20 samples, so it will take 100 batches until you see your target of 2,000 samples.

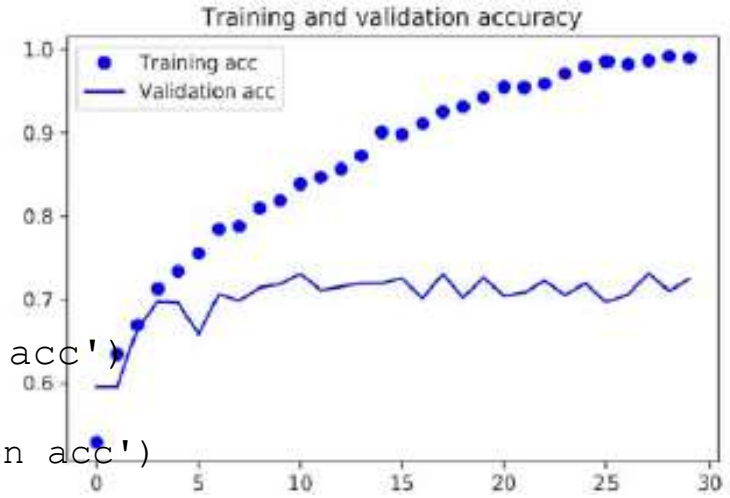
Fitting `model` to `data`, method `fit()`

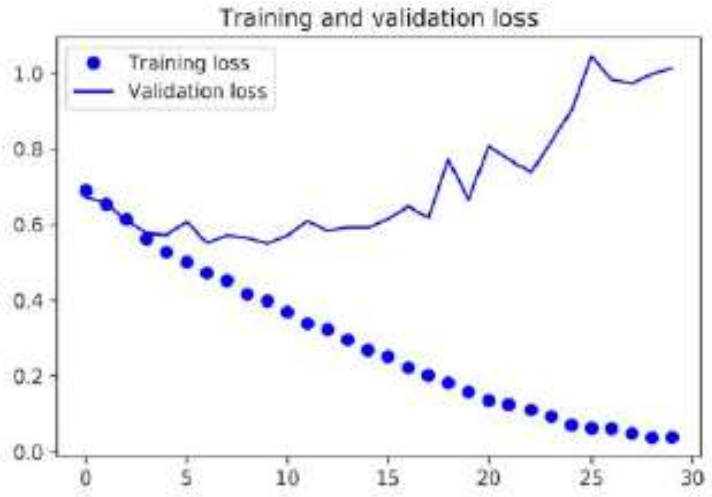
- When using `fit`, you can pass a `validation_data` argument.
- It's important to note that this argument is allowed to be a data generator, but it could also be a tuple of Numpy arrays.
- If you pass a generator as `validation_data`, then this generator is expected to yield batches of validation data endlessly; thus, you should also specify the `validation_steps` argument, which tells the process how many batches to draw from the validation generator for evaluation.
- Because the data is being generated endlessly, the generator needs to know how many samples to draw from the generator before declaring an epoch over. This is the role of the `steps_per_epoch` argument. After having drawn `steps_per_epoch` batches from the generator, i.e. after having run for `steps_per_epoch` gradient descent steps, the fitting process will go to the next epoch. In our case, batches are 20-sample large, so it will take 100 batches until we see the entire target of 2000 samples.

```
history = model.fit(
        train_generator,
        steps_per_epoch=100,
        epochs=30,
        validation_data=validation_generator,
        validation_steps=50)
```

Accuracy over training and validation data

```
import matplotlib.pyplot as plt
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, 'bo', label='Training acc')

plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Valida
plt.title('Training and validation loss')
plt.legend()
plt.show()
```





- These plots are characteristic of overfitting.
- We have relatively few training samples (2,000)
- Overfitting is the number-one concern.
- You should always save your models

```
model.save('cats_and_dogs_small_1.h5')
```

Visualizing Intermediate Activations

Immediate Objectives

- The representations learned by CNNs are moderately easy to visualization and understand, in large part because they are *representations of visual concepts*.

We will cover three of the most accessible and useful visualization techniques.

- **Visualizing intermediate CNN outputs** ("intermediate activations"). Activation values of CNN layers help us understand how successive convnet layers transform their inputs and discover the meaning/purpose of the individual convnet filters.
- **Visualizing CNN filters**. This tells us what visual patterns or concepts each filter in a convnet is receptive to.
- **Visualizing heatmaps of class activation in an image**. Heat maps help us understand which part of an image is responsible for that image being classified as belonging to a given class. These heat maps are one way of localizing objects in images.
- Note: Extensive analysis of CNNs require GPU, TPU or at least multi-CPU power. The following examples should be done in Google Colab or on a GPU equipped local machine.

Visualizing intermediate activations

- To visualize intermediate activations, we display the feature maps produced by various convolution and pooling layers, given a certain input
- The output of a layer is often called its "activation“, The activation is the collection of outputs of all activation functions of all neurons in a layer. Activations provide a view into how an input is transformed by different filters learned by the network.
- The feature maps have 3 dimensions: `width`, `height`, and `depth` (the number of channels). Each channel encodes relatively independent features. To visualize those feature maps, we independently plot the contents of every channel, as a 2D image.
- We will start by loading the model we saved in previous experiments:

```
import tensorflow as tf
from tensorflow import keras
from keras.models import load_model

model = load_model('cats_and_dogs_small_1.h5')
```

- By loading a saved model, we are bringing an entire trained network back to life.

Model Content

To remind ourselves of the content of the network, we do: `model.summary()`:

```
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
```

Examining the results, i.e., feature maps, on one image input

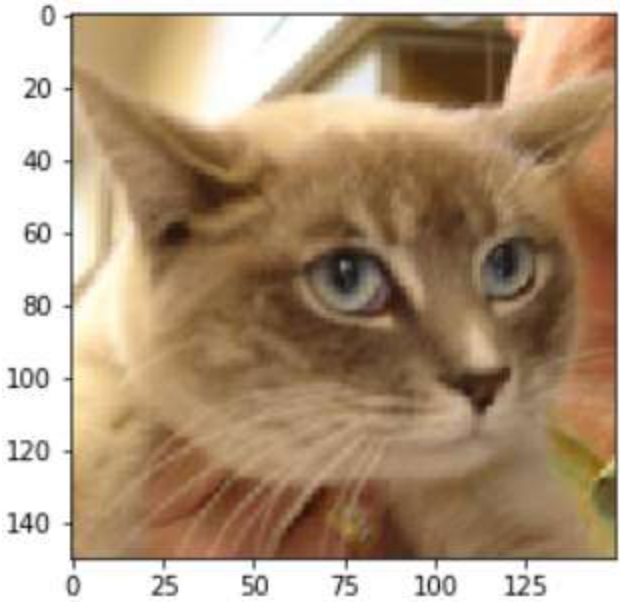
- We will take one of the test images, a cat that was not part of the training or validation set and preprocess that image into a 4 D tensor:

```
img_path = 'cats_and_dogs_small/test/cats/cat.1512.jpg'

# We preprocess the image into a 4D tensor
from tensorflow.keras.preprocessing import image
from tensorflow.keras import utils
import numpy as np
img = utils.load_img(img_path, target_size=(150, 150))
img_tensor = utils.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)
# Remember that the model was trained on inputs
# that were preprocessed by rescaling with 255:
img_tensor /= 255.0
# Its shape is (1, 150, 150, 3)
print(img_tensor.shape)
(1, 150, 150, 3)
```

(1 image in a batch, 150x150 pixels and 3 colors)

- We could actually display cat's image:

```
import matplotlib.pyplot as plt

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



Class tf.keras.Model

- Class `tf.keras.Model` is part of `Keras functional API`.
- Given input tensor(s) & output tensor(s), we instantiate a `Model` by passing `input` and `output` tensors. For example:

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
a = Input(shape=(32,))
b = Dense(32)(a)
model = Model(inputs=a, outputs=b)
```

- This `model` includes all layers required in the computation of tensor `b` given tensor `a`.
- In the case of multi-input or multi-output models, you can use lists as well:

```
model = Model(inputs=[a1, a2], outputs=[b1, b3, b3])
```

Useful attributes of Model are:

- `model.layers`, a flattened list of the layers comprising the model graph.
- `model.inputs`, the list of input tensors.
- `model.outputs`, the list of output tensors.

Method `compile()` of class `Model` configures the model for training. Its signature reads

```
compile(self, optimizer, loss=None, metrics=None,
loss_weights=None, sample_weight_mode=None,
weighted_metrics=None, target_tensors=None)
```

Methods of class `Model`

- Other methods of class `Model` are:

`fit()`, trains the model for a fixed number of epochs (iterations over the dataset). Method signature reads:

```
fit(self, x=None, y=None, batch_size=None, epochs=1, verbose=1,
callbacks=None, validation_split=0.0, validation_data=None, shuffle=True,
class_weight=None, sample_weight=None, initial_epoch=0,
steps_per_epoch=None, validation_steps=None)
```

`evaluate()` returns the loss & accuracy values for the model in test mode. Method's signature is:

```
evaluate(self, x=None, y=None, batch_size=None, verbose=1,
sample_weight=None, steps=None)
```

`predict()` generates output predictions for the input samples. Computation is done in batches. The signature of this method reads:

```
predict(self, x, batch_size=None, verbose=0, steps=None)
```

`train_on_batch()` runs a single gradient update on a single batch of data with signature:

```
train_on_batch(self, x, y, sample_weight=None, class_weight=None)
```

- Full detailed description of class Model can be found at: https://www.tensorflow.org/api_docs/python/tf/keras/Model

Extracting Feature Maps

- To extract feature maps, we create a Keras model that takes batches of images (or an individual image) as input, and outputs the activations of all convolution and pooling layers.
- 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 object is a Keras `model`, similar to the `Sequential` models, mapping the specified inputs to the specified outputs.
- Class `Model` allows models with multiple inputs and multiple outputs, unlike `Sequential`.

```
from tensorflow.keras import Model
# Extracts the outputs of the first 8 (conv and max_pooling layers)
layer_outputs = [layer.output for layer in model.layers[:8]]
# Creates a model that returns these outputs, given the model's input:
activation_model = Model(inputs=model.input, outputs=layer_outputs)
```

- When fed an image input, this model returns the values of activations of first 8 layers in the original model (those are `conv` and `max_pooling` layers).
- So far, the models we have seen had exactly one input and one output.
- In the general case, a model could have any number of inputs and outputs.
- This particular model has one input and 8 outputs, one output per activation generated by each layer.

Activations in the first Layer, `cat.1512/1700.jpg`

- The following command returns a list of 8 Numpy arrays:

```
# one array per layer activation
activations = activation_model.predict(img_tensor)
len(activations) # will give 8
```

- The activation of the first convolution layer for the cat image input

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

- This is a 148 × 148 feature map with 32 channels. We plot the activation of the 5th channel of the first layer of the original model, for `cat.1512 &1700.jpg` input:

```
import matplotlib.pyplot as plt
plt.matshow(first_layer_activation[0, :, :, 4], cmap='viridis')
```

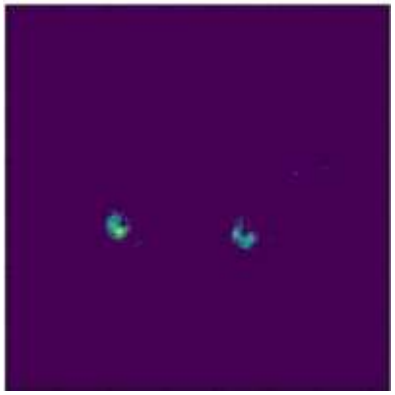
- This channel appears to encode a diagonal edge detector.



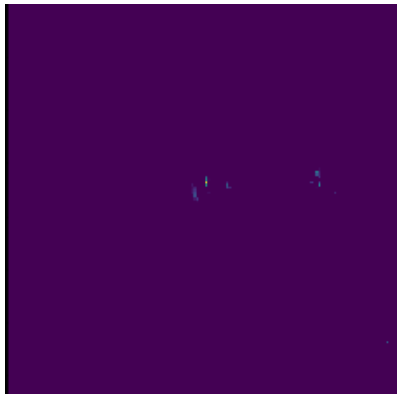

Activation of the first layer, 8th Channel

- Let's try the 32nd channel of the activation of the first layer on cat.1700.jpg.
- Note that your own channels may vary, because the specific filters learned by convolution layers aren't deterministic.

```
plt.matshow(first_layer_activation[0, :, :, 31], cmap='viridis')
```



The filter for `cat.1700` looks like a “bright green dot” detector, useful for encoding cat eyes.



The filter for `cat.1512` could also be interpreted as eye detector?

Visualizing every channel in every intermediate activation

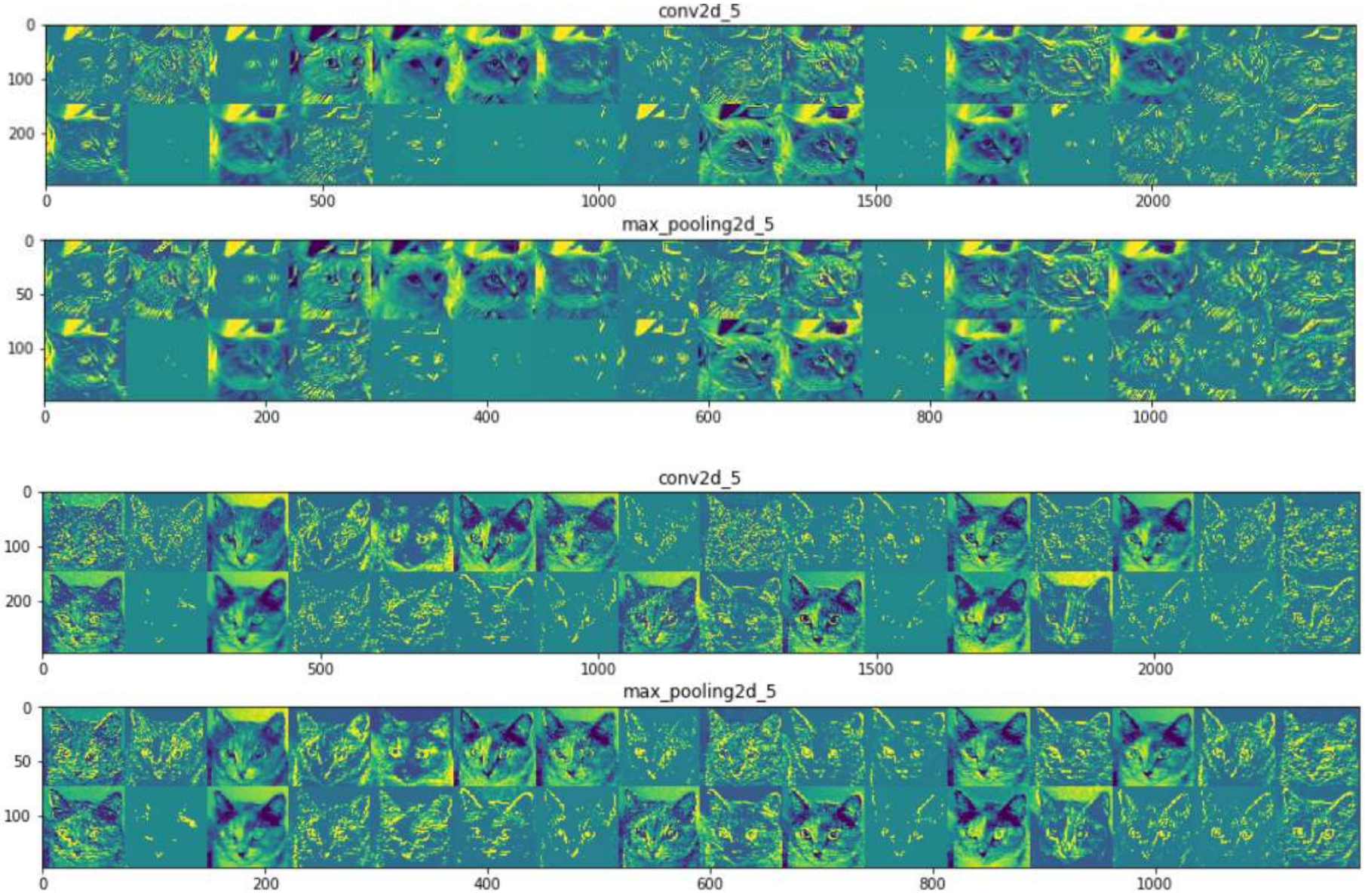
- Next, we will plot complete visualization of all the activations in the network.
- We extract and plot every channel in each of 8 activation maps, and we stack the results in one big image tensor, with channels stacked side by side

```
from tensorflow import keras
layer_names = []  # Names of layers, will be part of the plot
for layer in model.layers[:8]:
    layer_names.append(layer.name)

images_per_row = 16    # Display the feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    n_features = layer_activation.shape[-1] # No of features in feature map

    size = layer_activation.shape[1] #Feature map shape (1, size, size, n_features)

    n_cols = n_features // images_per_row   # We tile channels in this matrix
    display_grid = np.zeros((size * n_cols, images_per_row * size))
    for col in range(n_cols):  # We tile each filter into this big horizontal grid
        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

    scale = 1. / size     # Display the grid
    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()
```

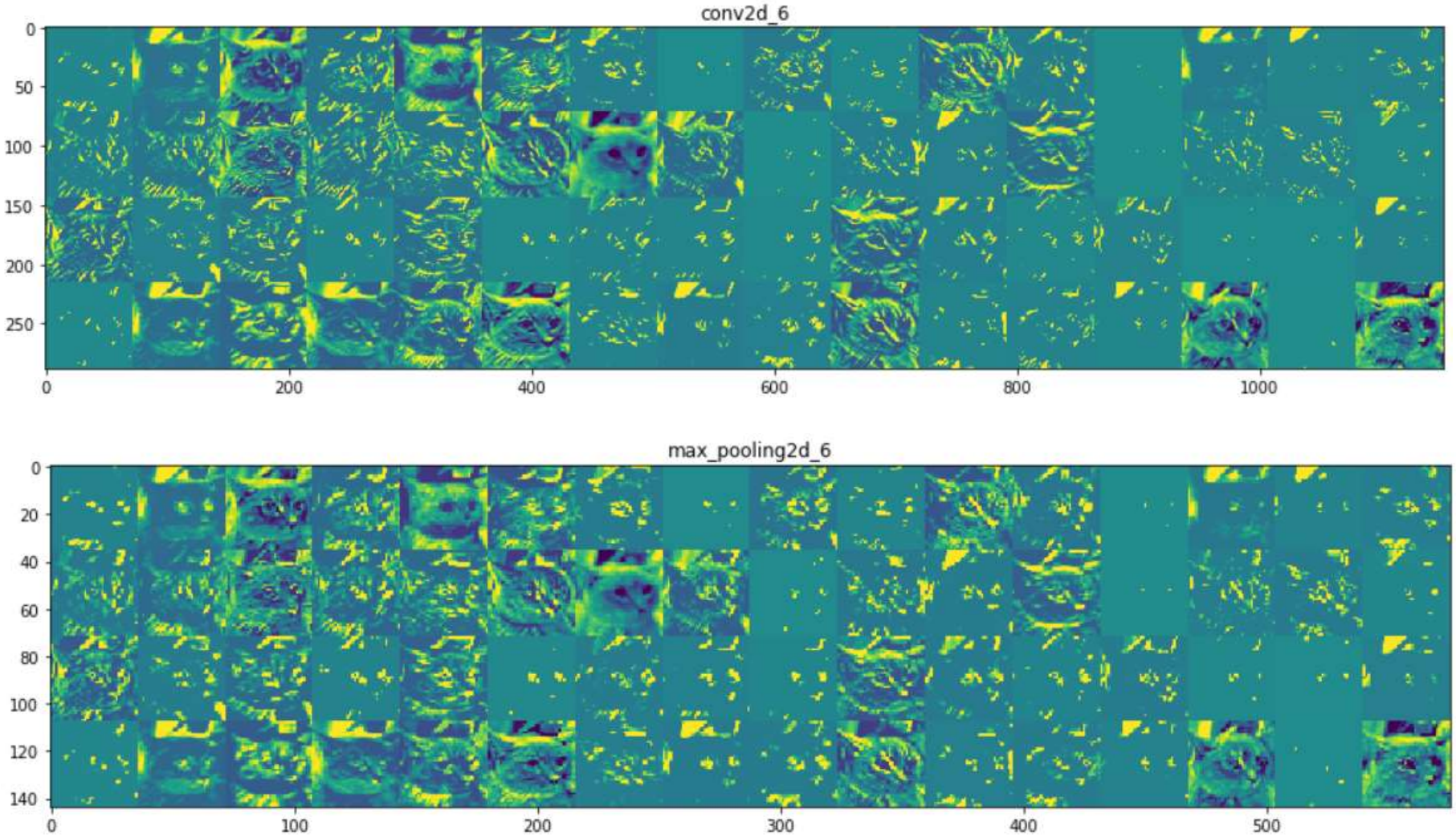
Every channel in intermediate activations

- In the following slides we presented channels for every activation layer for cat.1512 and cat.1700.
- There obviously are differences between activations generated by two images.
- Blank channels, the ones which have no content for a particular cat, are different as well.
- We went through this somewhat unusual exercise of placing so many images into a set of Power Points slides in order to underline the differences and similarities of the fate of any image as it propagate through the CNN stack of layers.

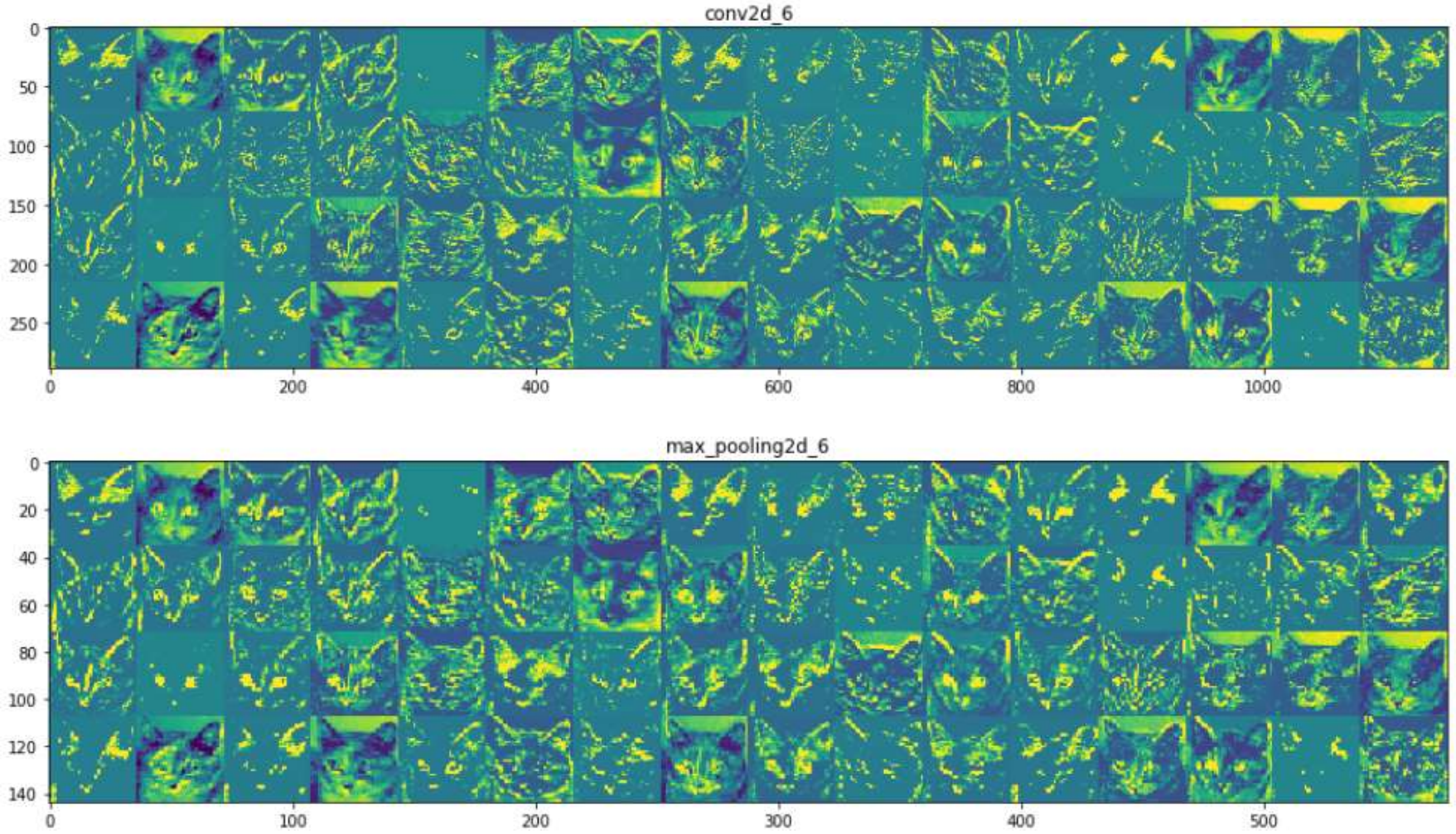
Feature Maps, cat.1512 (top), cat.1700 (bottom)



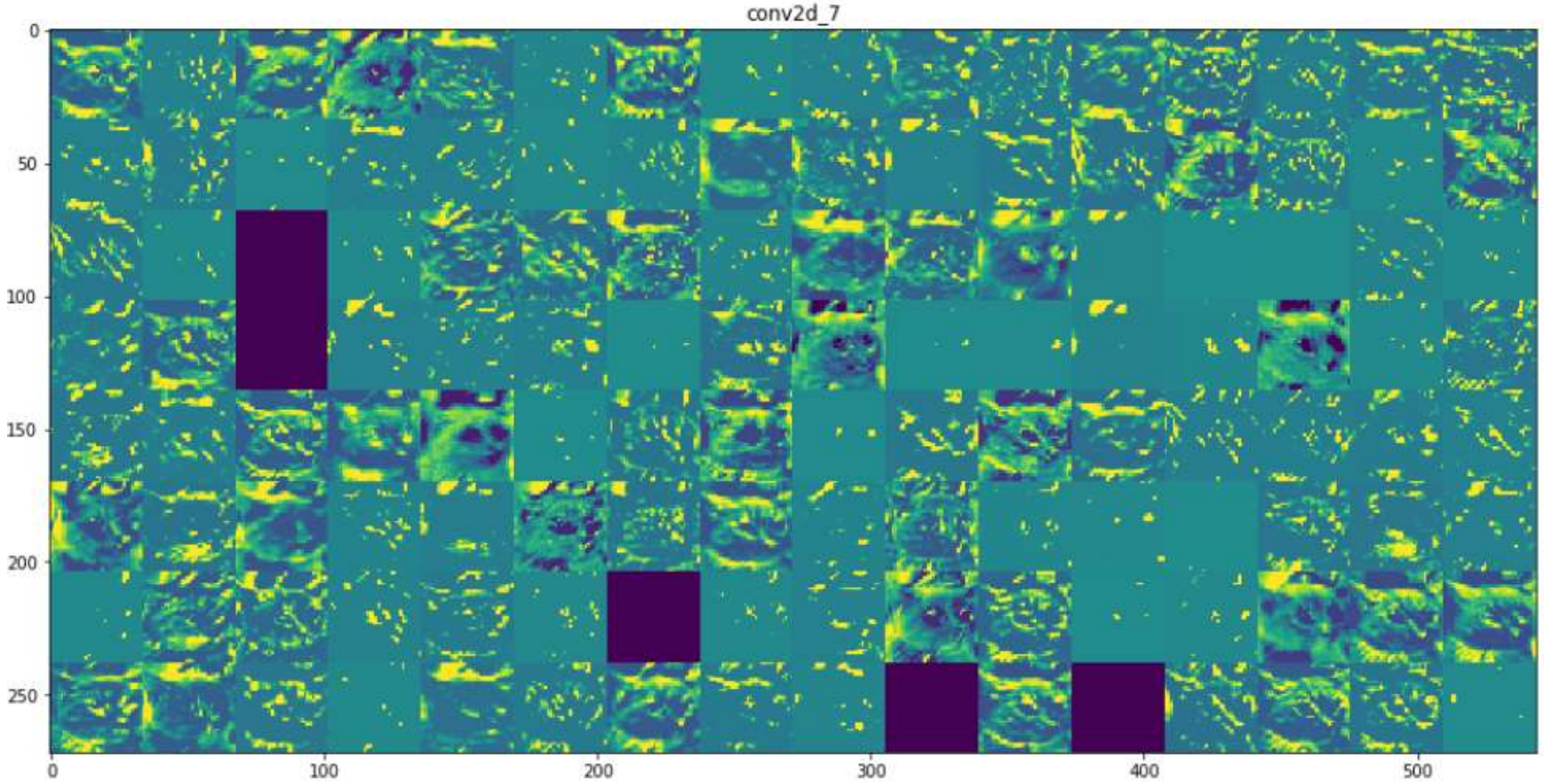
Feature Maps, cat.1512



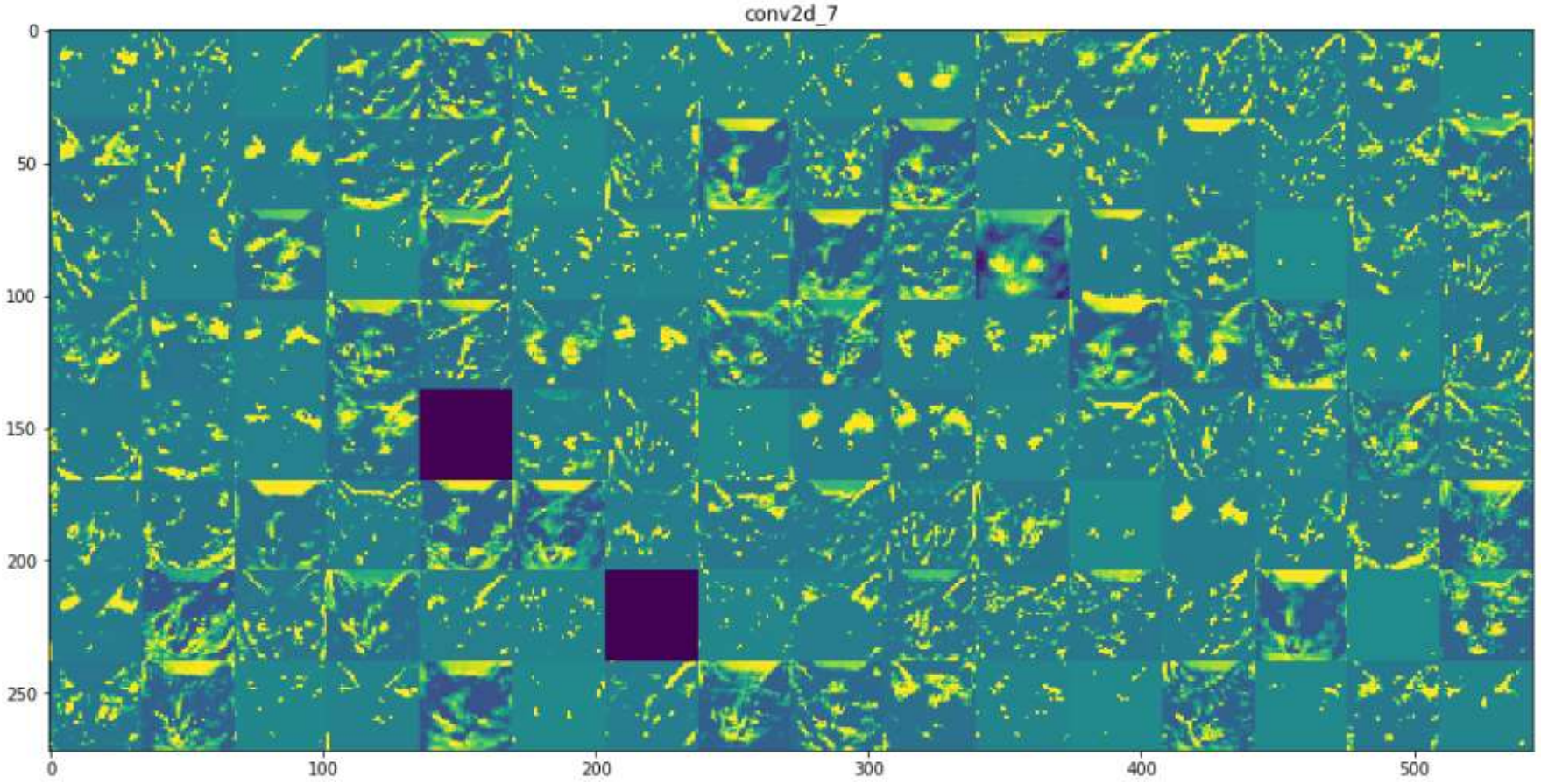
Feature Maps, cat.1700



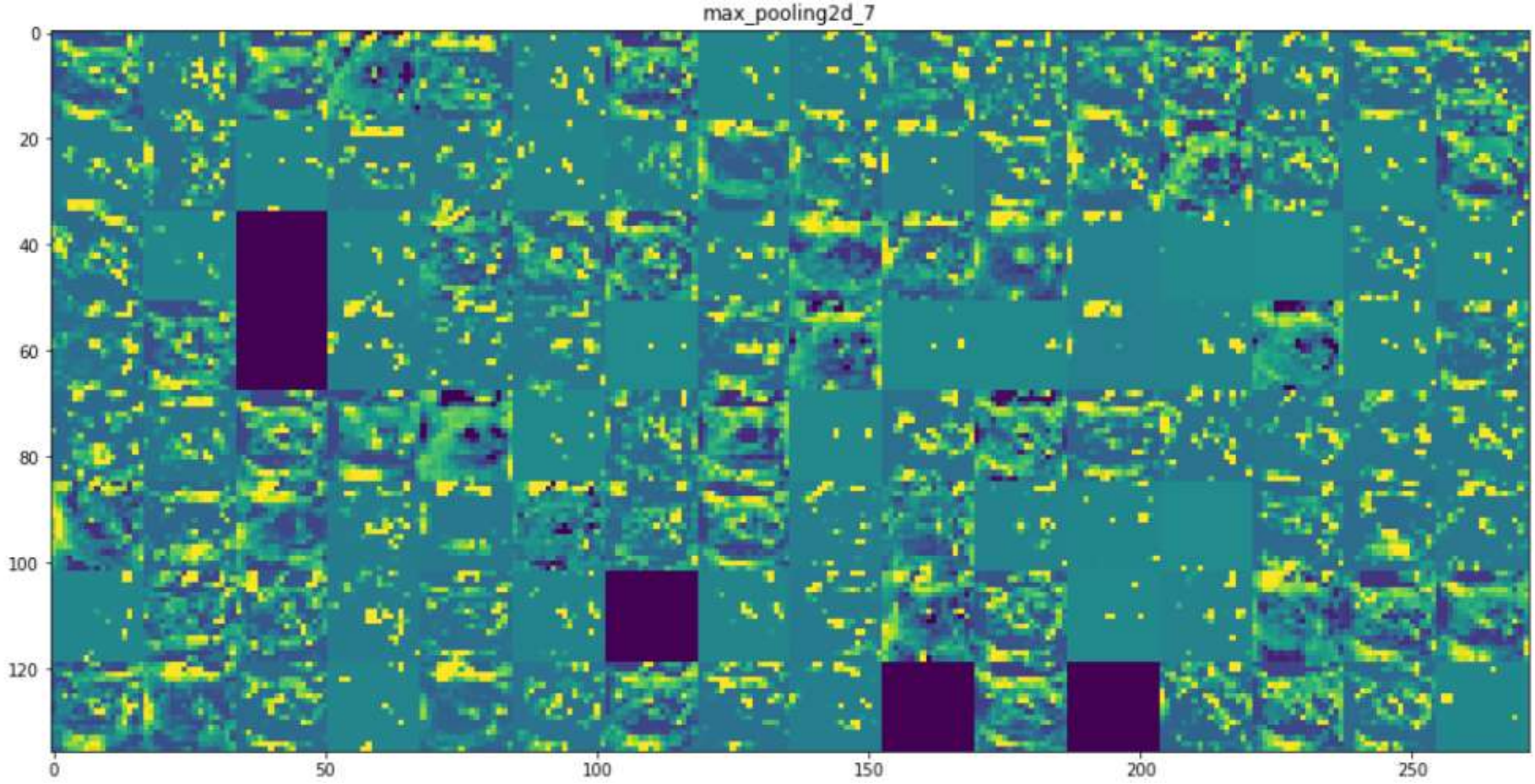
Feature Maps, cat.1512



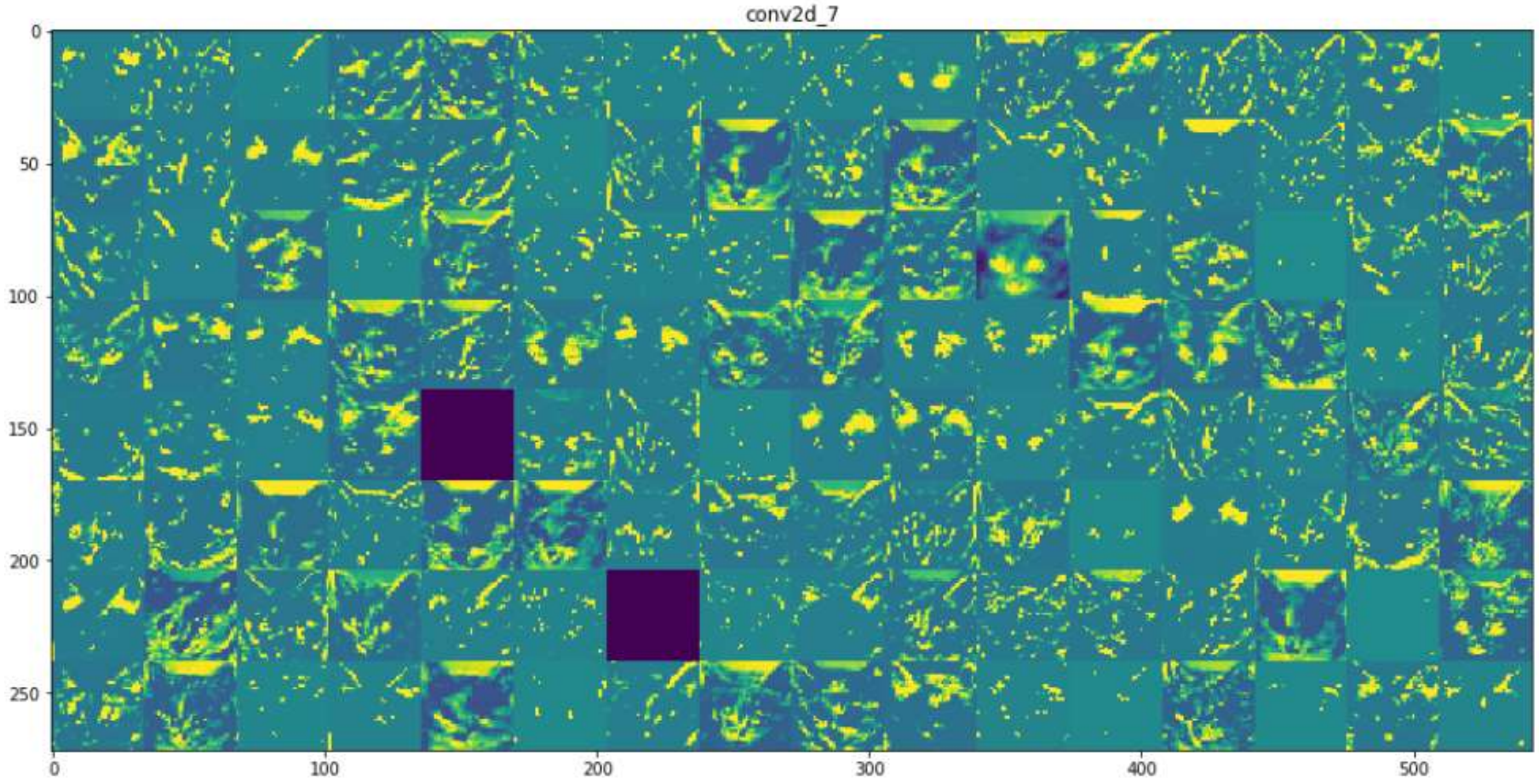
Feature Maps, cat.1700



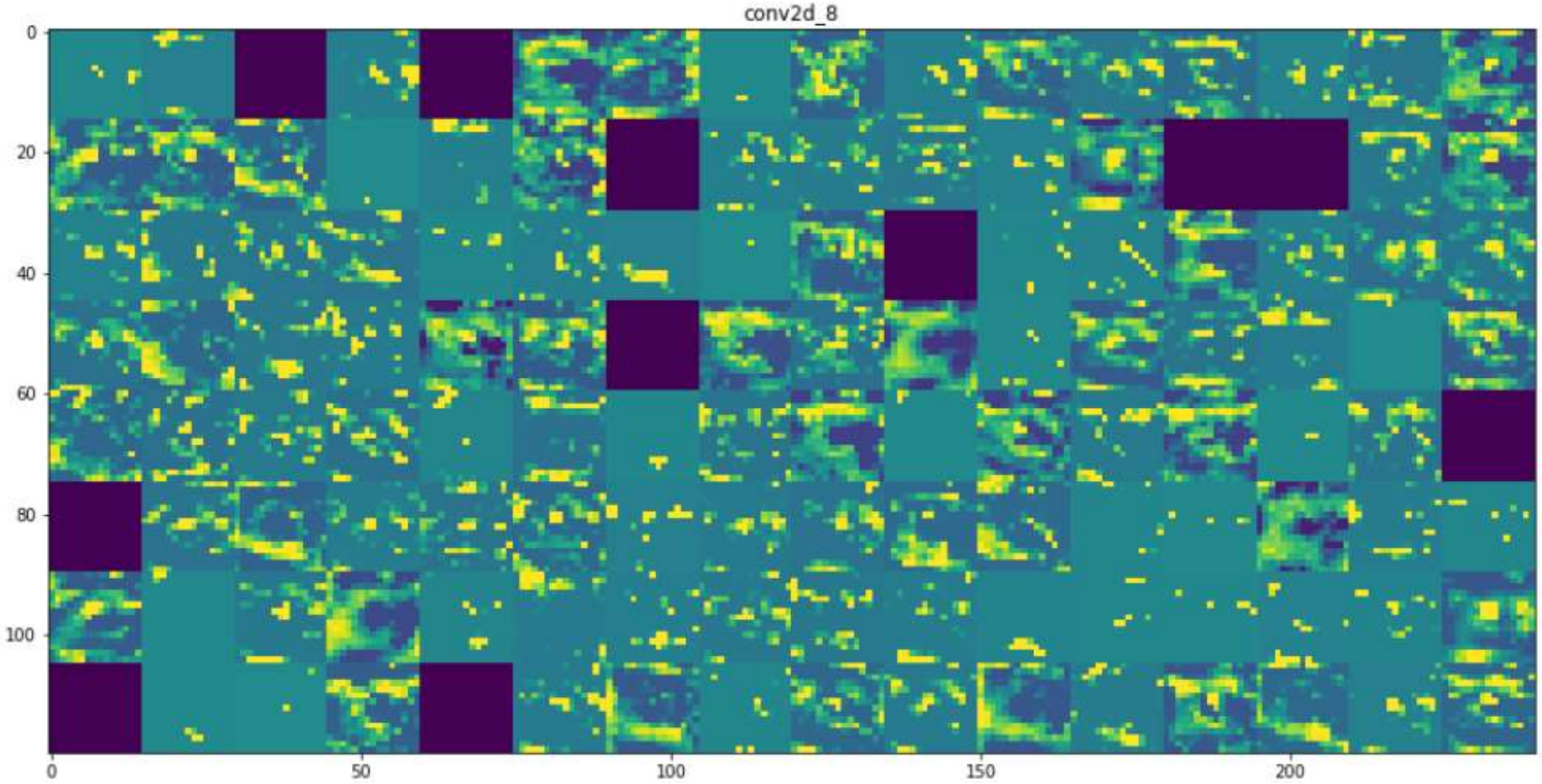
Feature Maps, cat.1512



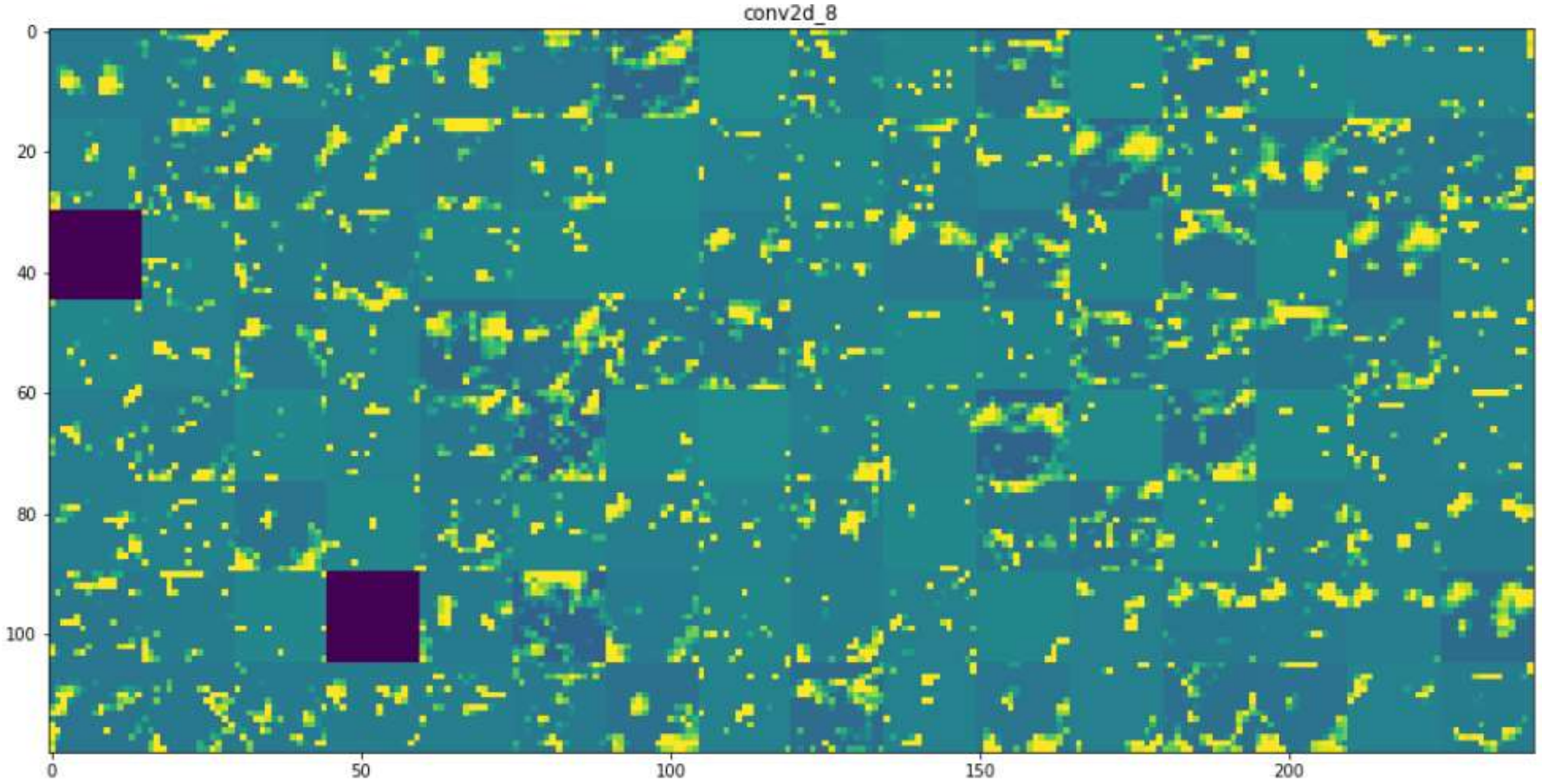
Feature Maps, cat.1700



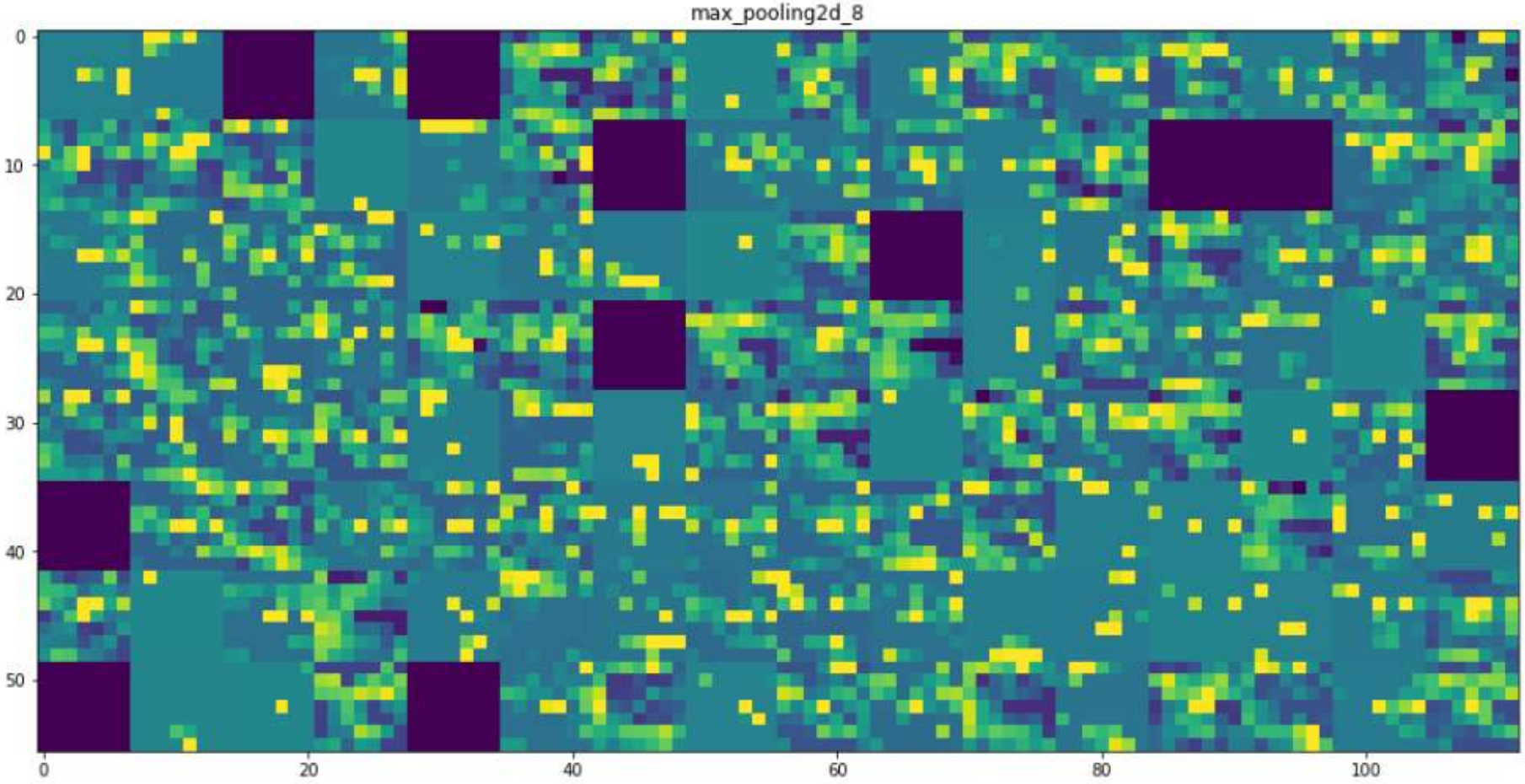
Feature Maps, cat.1512



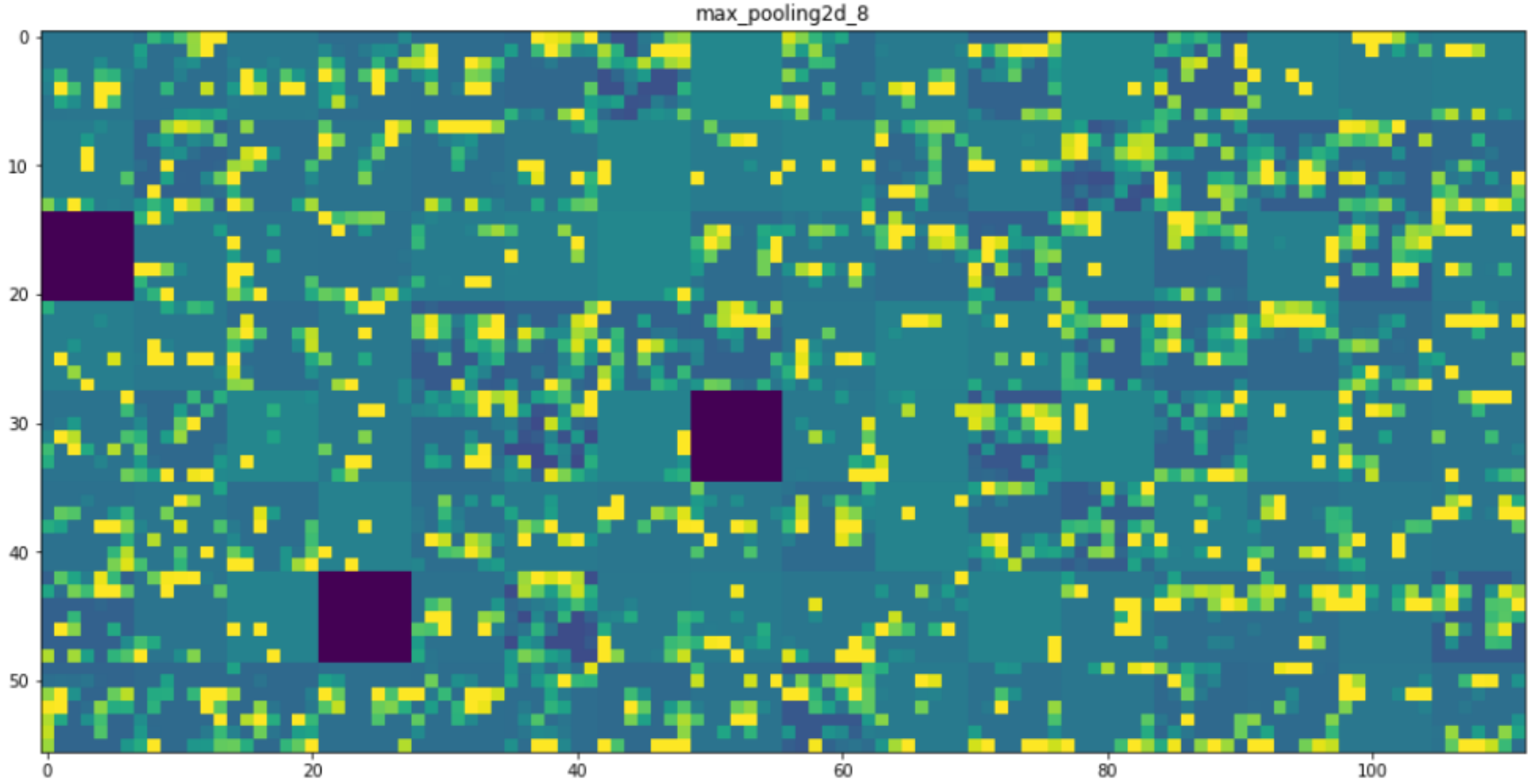
Feature Maps, cat.1700



Feature Maps, cat.1512



Feature Maps, cat.1700



Interpretation of Feature Maps

- The first layer acts as a collection of various edge detectors. At that stage, the activations are still retaining almost all 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.

Interpretation of Feature Maps

- We have seen a very important universal characteristic of the representations learned by deep neural networks: the features extracted by a layer get increasingly abstract with the depth of the layer.
- The activations of layers higher-up carry less and less information about the specific input being seen, and more and more information about the target (in our case, the class of the image: cat or dog).
- A deep neural network effectively acts as an **information distillation pipeline**, with raw data going in (in our case, RBG pictures), and getting repeatedly transformed so that irrelevant information gets filtered out (e.g., the specific visual appearance of the image) while useful information get magnified and refined (e.g. ,the class of the image).
- This is analogous to the way humans and animals perceive the world: after observing a scene for a few seconds, a human can remember which abstract objects were present in it (e.g., bicycle, tree) but could not remember the specific appearance of these objects. In fact, if you tried to draw a generic bicycle from mind right now, chances are you could not get it even remotely right, even though you have seen thousands of bicycles in your lifetime.
- This effect is real. Your brain has learned to completely abstract its visual input, to transform it into high-level visual concepts while completely filtering out irrelevant visual details, making it tremendously difficult to remember how things around us actually look.

Stored Architectures

Stored Architectures with Trained Weights

- Keras module `tf.keras.applications` contains several, historically most important models, i.e. canned architectures with pre-trained weights.
- We can import those trained models, use them for inference (predictions) or use them for examination and study. Some of the available architectures are:
  - `densenet` module: DenseNet models for Keras.
  - `efficientnet` module: EfficientNet models for Keras.
  - `imagenet_utils` module: Utilities for ImageNet data preprocessing & prediction decoding.
  - `inception_resnet_v2` module: Inception-ResNet V2 model for Keras.
  - `inception_v3` module: Inception V3 model for Keras.
  - `mobilenet` module: MobileNet v1 models for Keras.
  - `mobilenet_v2` module: MobileNet v2 models for Keras.
  - `nasnet` module: NASNet-A models for Keras.
  - `resnet` module: ResNet models for Keras.
  - `resnet50` module: Public API for tf.keras.applications.resnet50 namespace.
  - `resnet_v2` module: ResNet v2 models for Keras.
  - `vgg16` module: VGG16 model for Keras.
  - `vgg19` module: VGG19 model for Keras.
  - `xception` module: Xception V1 model for Keras.
- In the following slides we will start working with one such architecture, VGG16.

Examining Filter Weights

Download VGG16 Trained Model

- We will work with VGG16 network, pre-trained on ImageNet:

```
from tensorflow.keras.applications import VGG16

model = VGG16(weights='imagenet',include_top=False)
```

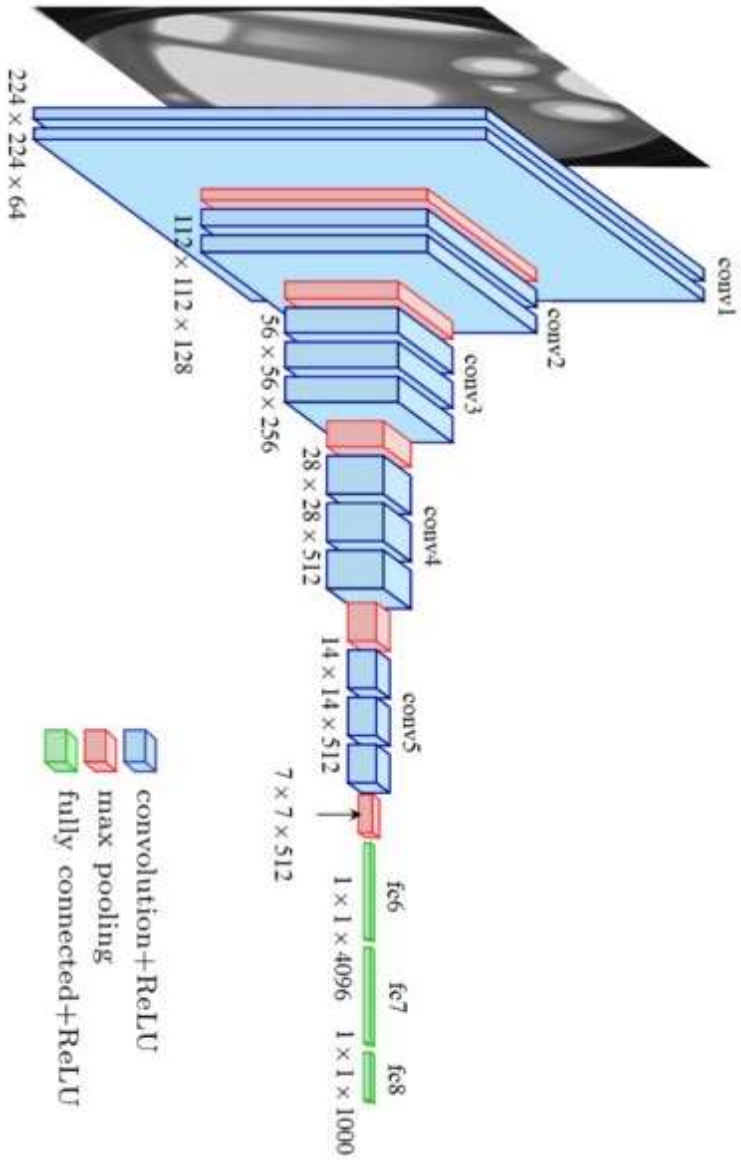
- The classification layers are irrelevant for this use case, so we specify `include_top=False` stage of the model.

```
model.summary()
```

- Blocks and layers and their output shape are presented on the next slide.
- Notice that the architecture involves 5 blocks with varying number of layers.

VGG16 Architecture

```
Model: "vgg16"_________________________________________________________
Layer (type)                 Output Shape              Param #
=========================================================
input_1 (InputLayer)         [(None, None, None, 3)]   0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, None, None, 64)    1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, None, None, 64)    36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, None, None, 64)    0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, None, None, 128)   73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, None, None, 128)   147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, None, None, 128)   0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, None, None, 256)   295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, None, None, 256)   590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, None, None, 256)   590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, None, None, 256)   0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, None, None, 512)   1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, None, None, 512)   0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, None, None, 512)   0
==============
Trainable params: 14,714,688
```



How to Visualize Filters

- Perhaps the simplest visualization to perform is to plot the learned filters directly.
- In neural networks, the learned filters are simply weights, yet because of the specialized two-dimensional structure of the filters, the weight values have a spatial relationship to each other and plotting each filter as a two-dimensional image is meaningful (or could be).
- The model summary printed above summarizes the output shape of each layer, e.g., the shape of the resulting feature maps. It does not give any idea of the shape of the filters (weights) in the network, only the total number of weights per layer.
- We can access all layers of the model via the `model.layers` property.
- Each layer has a `layer.name` property, where the convolutional layers have a naming convention like `block#_conv#`, where the `'#'` is an integer.
- We can check the name of each layer and skip any without the string `'conv'`.
- Each convolutional layer has two sets of weights.
- One is the block of filters and the other is the block of bias values. These are accessible via the `layer.get_weights()` function.
- We retrieve these weights and then summarize their shape.

Summarize Filters in Each Layer

```
from keras.applications.vgg16 import VGG16
from matplotlib import pyplot
# load the model
model = VGG16()
# summarize filter shapes
for layer in model.layers:
    # check for convolutional layer
    if 'conv' not in layer.name:
    continue
    # get filter weights
    filters, biases = layer.get_weights()
    print(layer.name, filters.shape)
```

```
block1_conv1 (3, 3, 3, 64)
block1_conv2 (3, 3, 64, 64)
block2_conv1 (3, 3, 64, 128)
block2_conv2 (3, 3, 128, 128)
block3_conv1 (3, 3, 128, 256)
block3_conv2 (3, 3, 256, 256)
block3_conv3 (3, 3, 256, 256)
block4_conv1 (3, 3, 256, 512)
block4_conv2 (3, 3, 512, 512)
block4_conv3 (3, 3, 512, 512)
block5_conv1 (3, 3, 512, 512)
block5_conv2 (3, 3, 512, 512)
block5_conv3 (3, 3, 512, 512)
```

- We can see that all convolutional layers use 3×3 filters (kernels), which are small and perhaps easy (or difficult) to interpret.
- An architectural concern with a convolutional neural network is that the depth of a filter must match the depth of the input for the filter (e.g. the number of channels).
- We can see that for the input image with three channels for red, green and blue, that each filter has a depth of three (here we are working with a channel-last format). We could visualize one filter as a plot with three images, one for each channel, or compress all three down to a single color image, or even just look at the first channel and assume the other channels will look the same. The problem is, we then have 63 other filters that we might like to visualize.

Retrieving Filters

- We can retrieve the filters from the first layer as follows:

```
# retrieve weights from the second hidden layer
filters, biases = model.layers[1].get_weights()
```

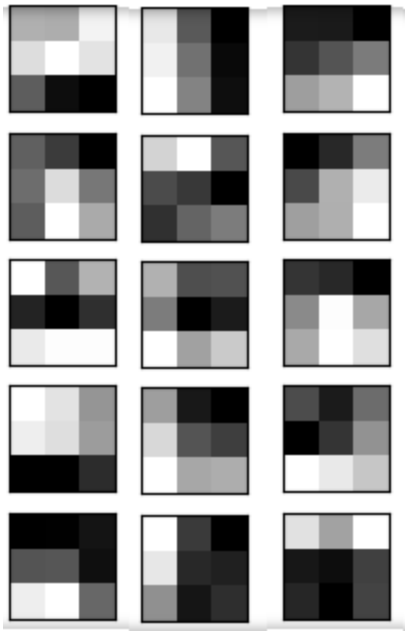
- The weight values will likely be small values centered around 0.0.
- We can normalize their values to the range 0-1 to make them easy to visualize.

```
f_min, f_max = filters.min(), filters.max()
filters = (filters - f_min) / (f_max - f_min)
```

- Now we can enumerate the first six filters out of the 64 in the block and plot each of the three channels of each filter.
- We use the matplotlib library and plot each filter as a new row of subplots, and each filter channel or depth as a new column.

Display filters as 3X3 gray images

```
# plot first few filters
n_filters, ix = 6, 1
for i in range(n_filters):
    # get the filter
    f = filters[:, :, :, i]
    # plot each channel separately
        for j in range(3):
        # specify subplot and turn of axis

        ax = pyplot.subplot(n_filters, 3, ix)

        ax.set_xticks([])
        ax.set_yticks([])
        # plot filter channel in grayscale

        pyplot.imshow(f[:, :, j], cmap='gray')

        ix += 1
# show the figure
pyplot.show()
```



Interpreting Filter Weights

- Running the example creates a figure with six rows of three images, or 18 images, one row for each filter and one column for each channel

- We can see that in some cases, the filter is the same across the channels (the first row), and in others, the filters differ (the last row).

- The dark squares indicate small or inhibitory weights and the light squares represent large or excitatory weights. Using this intuition, we can see that the filters on the first row detect a gradient from light in the top left to dark in the bottom right.

- Although we have a visualization, we only see the first six of the 64 filters in the first convolutional layer. Visualizing all 64 filters in one image is feasible.

- Sadly, this does not scale; if we wish to start looking at filters in the second convolutional layer, we can see that again we have 64 filters, but each has 64 channels to match the input feature maps. To see all 64 channels in a row for all 64 filters would require (64×64) 4,096 subplots in which it may be challenging to see any detail.

Visualizing Activations of Convnet Filters

Visualizing CNN Filters

- We could understand filters learned by convnets by displaying the visual pattern that each filter is meant to respond to.
- This can be done with **gradient ascent in input space** by applying a gradient descent like algorithm to the value of the input image so that CNN maximizes the response of a specific filter, starting from a blank input image. We call this technique *gradient ascent* since we want to maximize not minimize response.
- The resulting input image would be one that the chosen filter is maximally responsive to.
- This is the process:
  1. We build a loss function that maximizes the value of a given filter in a given convolution layer,
  2. Use *stochastic gradient ascent* to adjust the values of the input image to maximize this activation value.
- For instance, we will try to maximize the activation of filter 0 in the layer `"block3_conv1"` of the VGG16 network, pre-trained on ImageNet:

What are we doing, again

- Layers in VGG16 are all named something like
- `block2_conv1, block5_conv3`, etc..
- VGG16 is structured into blocks, each containing several convolutional layers.

- We determine the inputs that maximize the activation of the filters in different layers of the VGG16 architecture, trained on ImageNet.
- In our analysis, we only go up to the last convolutional layer. We do not include fully-connected layers.
- The reason is that adding the fully connected layers would forces us to use a fixed input size for the model (224x224, the original ImageNet format).
- By only keeping the convolutional modules, our model can be adapted to arbitrary input sizes.
- The model loads a set of weights pre-trained on ImageNet.

“Feature extractor" returns the output of a specific layer

- We create a new model that returns the output of a specific layer—a "feature extractor“ model. This is a Functional API model and is inspectable. We can query the output of any one of its layers and reuse it in a new model.
- We choose a specific layer:

```
layer_name = "block2_conv1"
layer = model.get_layer(name=layer_name)
feature_extractor = keras.Model(inputs=model.input, outputs=layer.output)
```

- We could replace `block2_conv1` with any layer in VGG16 convolutional base.
- Object `layer` is the layer we are interested in.
- We use `model.input` and `layer.output` to create a `model` that, given an input image, returns the output of the target layer.
- To the model, call it on some input data. VGG16 requires inputs to be preprocessed via the `keras.applications.vgg16.preprocess_input` function, which converts images from RGB to BGR, then each color channel is zero-centered with respect to the ImageNet dataset, without scaling.

```
activation = feature_extractor(
    keras.applications.vgg16.preprocess_input(img_tensor)
)
```

Define the Loss Function

- We define a function that returns a scalar value quantifying how much a given input image "activates" a given filter in the layer. This is the "loss function“ that we maximize during the gradient ascent process.

```
import tensorflow as tf
def compute_loss(image, filter_index):
    activation = feature_extractor(image)
    filter_activation = activation[:, 2:-2, 2:-2, filter_index]
    return tf.reduce_mean(filter_activation)
```

- The loss function takes an image tensor, and the intiger index of the filter we analyze.
- Note that we avoid border artifacts by only involving non-border pixels in the loss: we discard the first 2 pixels along the sides of the activation.
- Loss function `compute_loss` returns the mean of the activation values for the filter.

The difference between `model.predict(x)` and `model(x)`

- In an earlier analysis, we used `predict(x)` for feature extraction. Here, we use `model(x)`. Why?
- Both `y = model.predict(x)` and `y = model(x)` (where `x` is an array of input data) mean "run the model on `x` and retrieve the output `y`." Yet, those two methods are not exactly the same.
- `predict()` loops over the data in batches (in fact, you can specify the batch size via `predict(x, batch_size=64))`, and extracts the NumPy value of the outputs. It's schematically equivalent to:

```
def predict(x):
    y_batches = []
    for x_batch in get_batches(x):
        y_batch = model(x).numpy()
        y_batches.append(y_batch)
    return np.concatenate(y_batches)
```

- This means that `predict()` calls can scale to very large arrays. Meanwhile, `model(x)` happens in-memory and doesn't scale. On the other hand, `predict()` is not differentiable: you cannot retrieve its gradient if you call it in a `GradientTape` scope.
- You should use `model(x)` when you need to retrieve the gradients of the model call.
- You should use `predict()` if you just need predicted values.
- In short, always use `predict()`, unless you are writing a low-level gradient descent loop (as we are now).

Loss maximization via stochastic gradient ascent

- Let's set up the gradient ascent step function, using the `GradientTape`. Note that we'll use a `@tf.function` decorator to speed it up.
- A non-obvious trick to help the gradient-descent process go smoothly is to normalize the gradient tensor by dividing it by its L2 norm (the square root of the average of the square of the values in the tensor). This ensures that after the updates done to the input image, its magnitude is always within the same range.

```
@tf.function
def gradient_ascent_step(image, filter_index, learning_rate):
    with tf.GradientTape() as tape:
      tape.watch(image)                                  #1
      loss = compute_loss(image, filter_index)           #2
    grads = tape.gradient(loss, image)                   #3
    grads = tf.math.l2_normalize(grads)                  #4
    image += learning_rate * grads                       #5
   return image                                         #6
```

1. We explicitly watch the image tensor, since it is not a TensorFlow Variable.
2. Compute the loss scalar, indicating how much the current image activates the filter.
3. Computes the gradients of the loss with respect to the image.
4. Apply the "gradient normalization trick".
5. Move the image a little bit in a direction that activate our target filter more strongly.
6. Return the updated image, so we can run the step function in a loop.

Function to generate filter visualizations

- Next, we create a function that takes as input a layer name and a filter index, and returns a valid image tensor representing the pattern that maximizes the activation of the specified filter:

```
img_width = 150
img_height = 150
def generate_filter_pattern(filter_index):
    iterations = 30                  #1
    learning_rate = 10.              #2
    image = tf.random.uniform(
        minval=0.4,
        maxval=0.6,
        shape=(1, img_width, img_height, 3))   #3
    for i in range(iterations):                #4
        image = gradient_ascent_step(image, filter_index,
learning_rate)
    return image[0].numpy()
```

1. Number of gradient ascent steps to apply.
2. Amplitude of a single step.
3. Initialize an image tensor with random values (the Xception model expects input values in the [0, 1] range, so here we pick a range centered on 0.5).
4. Repeatedly update the values of the image tensor so as to maximize our loss function.

Utility function to convert a tensor into a valid image

- The resulting image tensor is a floating-point array of shape (150, 150, 3), with values that may not be integers within [0, 255]. Hence, you need to post-process this tensor to turn it into a displayable image. You do so with the following straightforward utility function.

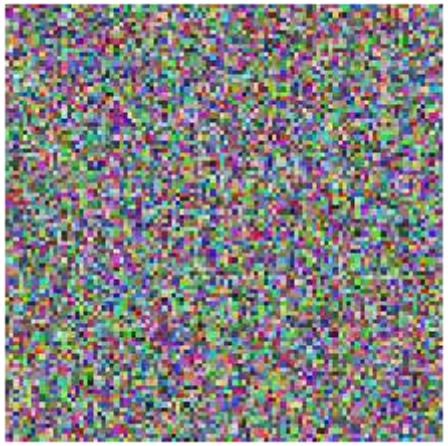
```
def deprocess_image(image):
    image -= image.mean()                                  #1
    image /= image.std()                                   #1
    image *= 64                                            #1
    image += 128                                           #1
    image = np.clip(image, 0, 255).astype("uint8")         #1
    image = image[25:-25, 25:-25, :]                       #2
    return image
```

1. Normalize image values within the [0, 255] range.
2. Center crop to avoid border artifacts.

Discovered pattern

- If we call the function on the previous slide and then show it as an image using `matplotlib.imshow()`, we would get the following:

```
plt.axis("off")
plt.imshow(deprocess_image(generate_filter_pattern(filter_index=0))
```

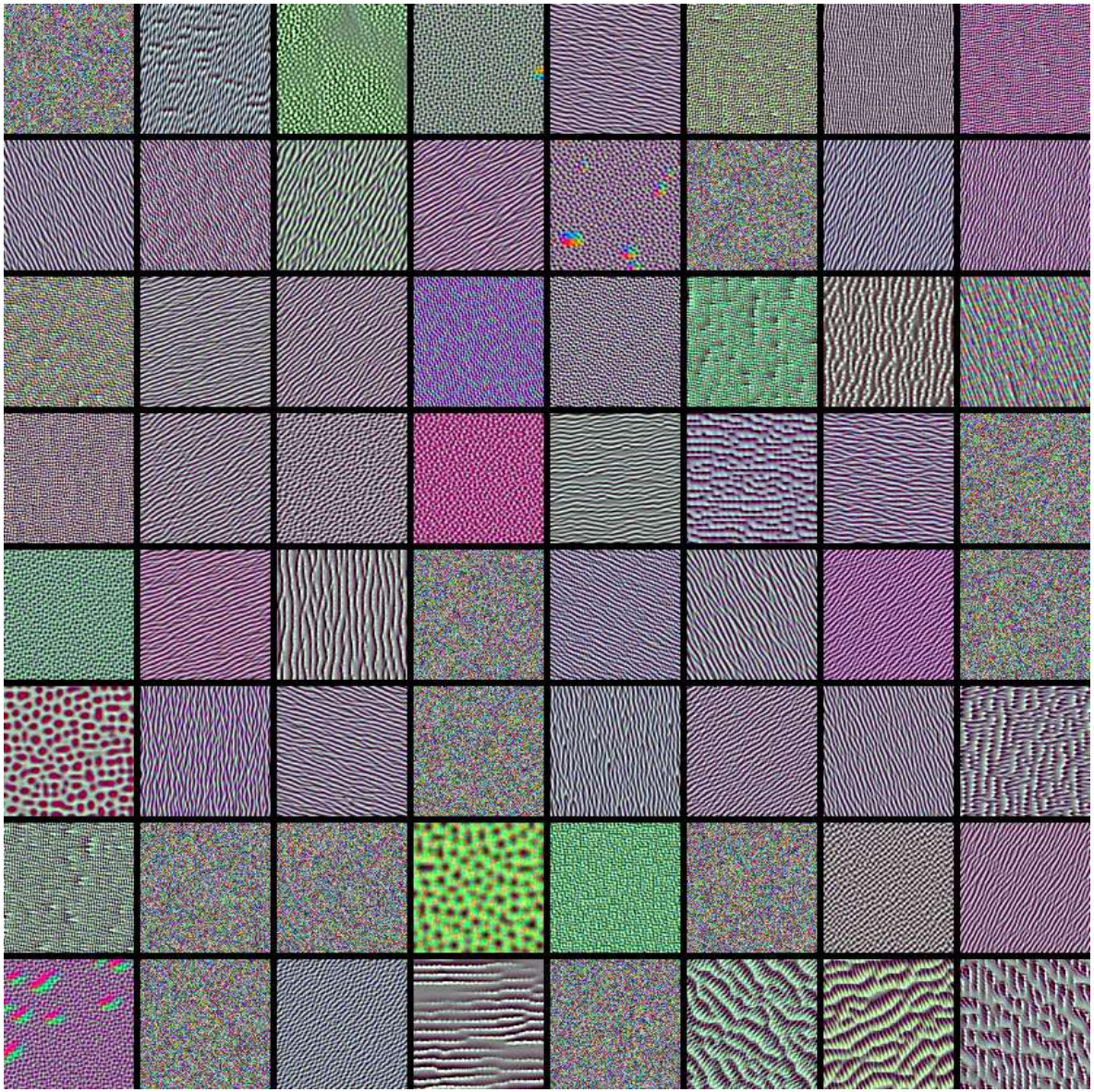


- It seems that filter 0 in layer `block3_conv1` is responsive to a polka dot pattern.
- We can visualize every single filter in every layer.

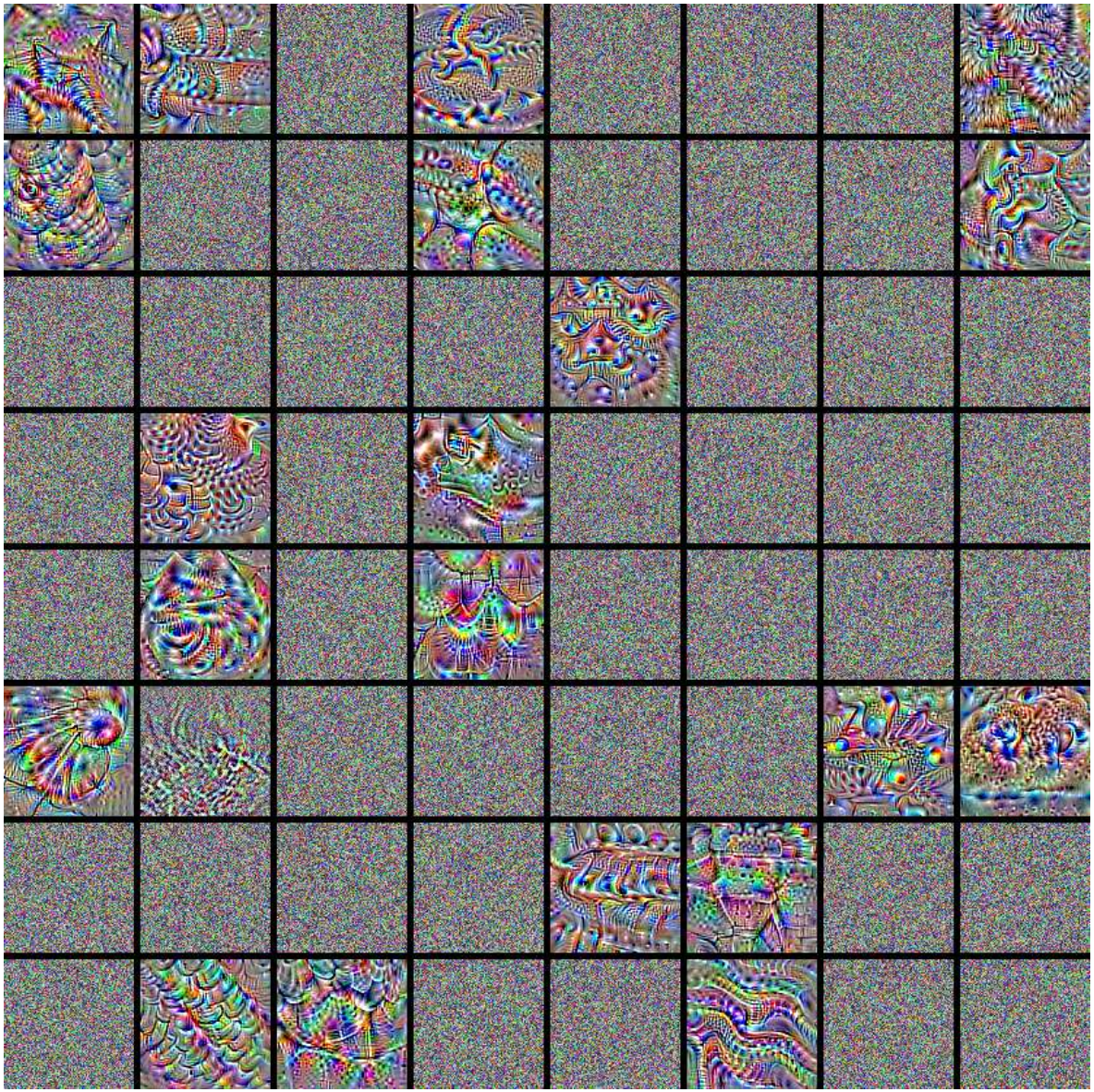
Filter Visualization

- Each layer in a convnet simply learns a collection of filters such that their inputs can be expressed as a combination of those filters. This is like how the Fourier transform decomposes signals onto a set of cosine functions.
- The filters in these convnet filter banks get increasingly complex and refined as we go higher-up in the model.
- These filter visualizations tell us a lot about how convnet layers see the world: each layer in a convnet simply learns a collection of filters such that their inputs can be expressed as a combination of the filters.
- The filters from the first layer in the model (block1_conv1) encode simple directional edges and colors (or colored edges in some cases).
- The filters from block2_conv1 encode simple textures made from combinations of edges and colors.
- The filters in higher-up layers start resembling textures found in natural images: feathers, eyes, leaves, etc.

1st 64 Filters in `block2_conv1`



1st 64 Filters in `block5_conv2`



Additional Observation

- A lot of these filters are identical but rotated by some non-random factor (typically 90 degrees). This means that we could potentially compress the number of filters used in a convnet by a large factor by finding a way to make the convolution filters rotation-invariant.
- The rotation observation holds true even for relatively high-level filters, such as those in block4_conv1.

Visualizing Heatmaps

Visualizing Heat-maps, Locating Objects in Images

- Heatmaps are useful in understanding which parts of a given image led a convnet to its final classification decision.
- This is helpful for "debugging" the decision process of a CNN, in particular in case of a classification mistake. Heat-maps allow you to locate specific objects in an image.
- This general category of techniques is called "Class Activation Map" (CAM) visualization and produces heat-maps of "class activation" over input images.
- A "class activation" heat-map is a 2D grid of scores associated with a specific output class, computed for every location in any input image, indicating how important each location is with respect to the identifying that class as network prediction.
- Given an image fed into one of our "cat vs. dog" convnet, Class Activation Map visualization allows us to generate a heat-map for the class "cat", indicating how cat-like different parts of the image are, and likewise for the class "dog", indicating how dog-like different parts of the image are.
- The specific implementation we will use is called Grad-CAM and was presented in the following paper:

  Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization by Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, Dhruv Batra (https://arxiv.org/abs/1610.02391)

Heatmaps

- The technique consists in taking the output feature map of a convolution layer given an input image and weighing every channel in that feature map by the gradient of the class with respect to the channel.
- Intuitively, one way to understand this trick is that we are weighting a spatial map of "how intensely the input image activates different channels" by "how important each channel is with regard to the class", resulting in a spatial map of "how intensely the input image activates the class".
- We will demonstrate this technique using the pre-trained VGG16 network:

```
from tensorflow.keras.applications.vgg16 import VGG16

# Note that we are including the densely-connected classifier on top;
# all previous times, we were discarding it.
model = VGG16(weights='imagenet')
```

model.summary

```
Model: "vgg16"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 input_1 (InputLayer)         [(None, 224, 224, 3)]     0

 block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792

 block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928

 block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0

 block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856

 block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584

 block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0

 block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168

 block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080

 block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080

 block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0

 block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160

 block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808

 block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808

 block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0

 block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808

 block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808

 block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808

 block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0

 flatten (Flatten)            (None, 25088)             0

 fc1 (Dense)                  (None, 4096)              102764544

 fc2 (Dense)                  (None, 4096)              16781312

 predictions (Dense)          (None, 1000)              4097000

=================================================================
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
_________________________________
```

Safari in savanna

- Consider the image of an African elephant shown in figure below. Let's convert this image into something the VGG16 model can read:

- We need to convert this image into something the VGG16 model can read: the model was trained on images of size 224x244, preprocessed according to a few rules that are packaged in the utility function `keras.applications.vgg16.preprocess_input`.
- So we need to load the image, resize it to 224x224, convert it to a Numpy float32 tensor, and apply these pre-processing rules.

Preprocess the Image

```
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.vgg16 import preprocess_input, decode_predictions
import numpy as np

# The local path to our target image
img_path = 'single-muddy-elephant.jpg'

# `img` is a PIL image of size 224x224
img = image.load_img(img_path, target_size=(224, 224))

# `img_array` is a float32 Numpy array of shape (224, 224, 3)
Img_array = image.img_to_array(img)

# We add a dimension to transform our array into a "batch"
# of size (1, 224, 224, 3)
img_array = np.expand_dims(img_array, axis=0)

# Finally we preprocess the batch
# (this does channel-wise color normalization)
img_array = preprocess_input(img_array)
```

Ask the model to categorize the image, `predict()`

- We will pass the preprocessed image to `model.predict()` method and than decode prediction, i.e., present readable class labels.

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

```
Predicted: [('n02504458', 'African_elephant', 0.46741), ('n01871265',
'tusker', 0.36958), ('n02504013', 'Indian_elephant', 0.16315)]
```

- The output presents 3 most probable classes and suggests that this image contains an "`African_elephant`" with probability of 47%.

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

```
np.argmax(preds[0])
```

- To visualize which parts of our image were the most "African elephant"-like, we will use the Grad-CAM process.

Model that returns the last convolutional output

- Setting up a model that returns the last convolutional output
- To visualize which parts of the image are the most African-elephant–like, let's set up the Grad-CAM process.
- First, we create a model that maps the input image to the activations of the last convolutional layer.

```
# VGG16, the last_conv_layer_name = "block5_conv3"
last_conv_layer_name = "block5_conv3"
classifier_layer_names = [
    "block5_pool",
    "flatten",
    "fc1",
    fc2",
    "predictions",]
last_conv_layer = model.get_layer(last_conv_layer_name)
last_conv_layer_model = keras.Model(model.inputs, last_conv_layer.output)
```

Model: the last convolutional output to the final predictions

- Setting up a model that goes from the last convolutional output to the final predictions
- Second, we create a model that maps the activations of the last convolutional layer to the final class predictions
- Reapplying the classifier on top of the last convolutional output.

```
classifier_input = keras.Input(shape=last_conv_layer.output.shape[1:])
x = classifier_input
for layer_name in classifier_layer_names:
        x = model.get_layer(layer_name)(x)

classifier_model = keras.Model(classifier_input, x)
```

Gradients of the top predicted class vs. convolutional output

- Retrieving the gradients of the top predicted class with respect to the last convolutional output

```
import tensorflow as tf
with tf.GradientTape() as tape:
    # Compute activations of the last conv layer and make the tape watch it.
    last_conv_layer_output = last_conv_layer_model(img_array)
    tape.watch(last_conv_layer_output)
    # Retrieve the activation channel corresponding to he top predicted class.
    preds = classifier_model(last_conv_layer_output)
    top_pred_index = tf.argmax(preds[0])
    top_class_channel = preds[:, top_pred_index]

grads = tape.gradient(top_class_channel, last_conv_layer_output)
```

- `grads` above is the gradient of the top predicted class with respect to the output feature map of the last convolutional layer.
- Now we can apply pooling and importance weighting to the gradient tensor to obtain the heatmap of class activation.

Gradient pooling and channel importance weighting

- `pooled_grads` below is a vector where each entry is the mean intensity of the gradient for a given channel. It quantifies the importance of each channel regarding the top predicted class.

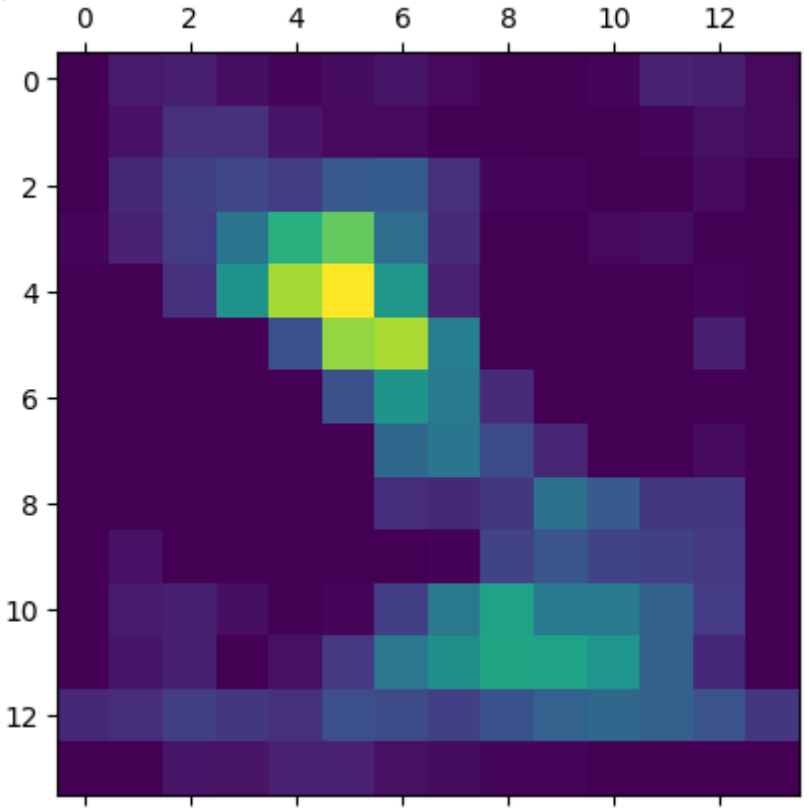
```
pooled_grads = tf.reduce_mean(grads, axis=(0, 1, 2)).numpy()
last_conv_layer_output = last_conv_layer_output.numpy()[0]

# Multiply each channel in the output of the last convolutional layer by
# how important this channell is“
for i in range(pooled_grads.shape[-1]):
    last_conv_layer_output[:, :, i] *= pooled_grads[i]

# The channel-wise mean of the resulting feature map is the heat-map of
class activation

heatmap = np.mean(last_conv_layer_output, axis=-1)
```

- The heat-map needs to be post-processed:

Normalized Heat-map

- For visualization purpose, we will also normalize the heat-map between 0 and 1:

```
heatmap = np.maximum(heatmap, 0)
heatmap /= np.max(heatmap)
plt.matshow(heatmap)
plt.show()
```



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

Superimpose Heat-map using OpenCV

- OpenCv can literarily add images. We rescale the heat map first:

```
import cv2

# We use cv2 to load the original image
img = cv2.imread(img_path)

# We resize the heatmap to have the same size as the original image
heatmap = cv2.resize(heatmap, (img.shape[1], img.shape[0]))

# We convert the heatmap to RGB
heatmap = np.uint8(255 * heatmap)

# We apply the heatmap to the original image
heatmap = cv2.applyColorMap(heatmap, cv2.COLORMAP_JET)

# 0.4 here is a heatmap intensity factor
superimposed_img = heatmap * 0.4 + img

# Save the image to disk
cv2.imwrite('elephant_cam.jpg', superimposed_img)
```

Superimposing the heatmap with the original picture

```
import matplotlib
img = keras.utils.load_img(img_path)
img = keras.utils.img_to_array(img)
heatmap = np.uint8(255 * heatmap)
# Use the "jet" colormap to recolorize the heatmap.
jet = matplotlib.colormaps.get_cmap("jet")
jet_colors = jet(np.arange(256))[:, :3]
jet_heatmap = jet_colors[heatmap]
# Create an image that contains the recolorized heatmap
jet_heatmap = keras.utils.array_to_img(jet_heatmap)
jet_heatmap = jet_heatmap.resize((img.shape[1], img.shape[0]))
jet_heatmap = keras.utils.img_to_array(jet_heatmap)
# Superimpose the heatmap and the original image, with 40% heatmap opacity.
superimposed_img = jet_heatmap * 0.4 + img
superimposed_img = keras.utils.array_to_img(superimposed_img)
# save the superimposed image
save_path = "elephant_cam.jpg"
superimposed_img.save(save_path)
```

Superimposed Image and Heatmap

- The result of operations is a new image.

- This visualization technique answers two important questions:
  - Why network thinks this image contained an African elephant. It found one?
  - Where is the African elephant located in the picture?
- It is interesting to note that the head and the ears of the elephant are strongly activated: this is probably how the network can tell the difference between African and Indian elephants.

Summary

- A fundamental problem when building a computer vision application is that of interpretability: why did your classifier think a particular image contained a fridge, when all you can see is a truck? This is especially relevant to use cases where deep learning is used in complement to human expertise, such as medical imaging use cases.
- We covered some techniques for visualizing what convnets learn and understanding the decisions they make.
- It’s often said that deep-learning models are "black boxes": they learn representations that are difficult to extract and present in a human-readable form. Although this is partially true for certain types of deep-learning models, it’s definitely not true for convnets. The representations learned by convnets are highly amenable to visualization, in large part because they’re representations of visual concepts.

Appendix: Keras Backend

Keras backend, a note

- Today, Keras is only used with TensorFlow. Comments on this and the next slide are important if you are dealing with some older code.

- Keras is a model-level library, providing high-level building blocks for developing deep learning models. It does not handle itself low-level operations such as tensor products, convolutions and so on. Instead, it relies on a specialized, well-optimized tensor manipulation library to do so, serving as the "backend engine" of Keras.
- Two most important backend implementations available are **Theano** backend and the **TensorFlow** backend.
- If you have run Keras at least once, you will find the Keras configuration file at:

```
~/.keras/keras.json
```

- If it isn't there, you can create it. It probably looks like this:

```
{"epsilon": 1e-07, "floatx": "float32", "backend": "theano"}
```

- Simply change the field backend to either "`theano`" or "`tensorflow`", and Keras will use the new configuration next time you run any Keras code.
- You can also define the environment variable `KERAS_BACKEND` and this will override what is defined in your config file :

```
KERAS_BACKEND=tensorflow
python -c "from keras import backend; print backend._BACKEND"
Using TensorFlow backend.
```

-

Using the abstract Keras backend to write new code

- If you want the Keras code you write to be compatible with both Theano and TensorFlow, you must write it via the abstract Keras backend API. Here's an intro.
- You can import the backend module via:

```
from keras import backend as K
```

- The code below instantiates an input placeholder. It's equivalent to `tf.placeholder()` or `T.matrix()`, `T.tensor3()`, etc.

```
input = K.placeholder(shape=(2, 4, 5))
# also works:
input = K.placeholder(shape=(None, 4, 5))
# also works:
input = K.placeholder(ndim=3)
```

```
The code below instantiates a shared variable. It's equivalent to tf.variable()
or theano.shared().
val = np.random.random((3, 4, 5))
var = K.variable(value=val)
```

- Most tensor operations you will need can be done as you would in TensorFlow or Theano:

```
a = b + c * K.abs(d)
c = K.dot(a, K.transpose(b))
a = K.sum(b, axis=2)
a = K.softmax(b)
a = concatenate([b, c], axis=-1)
# etc...
```