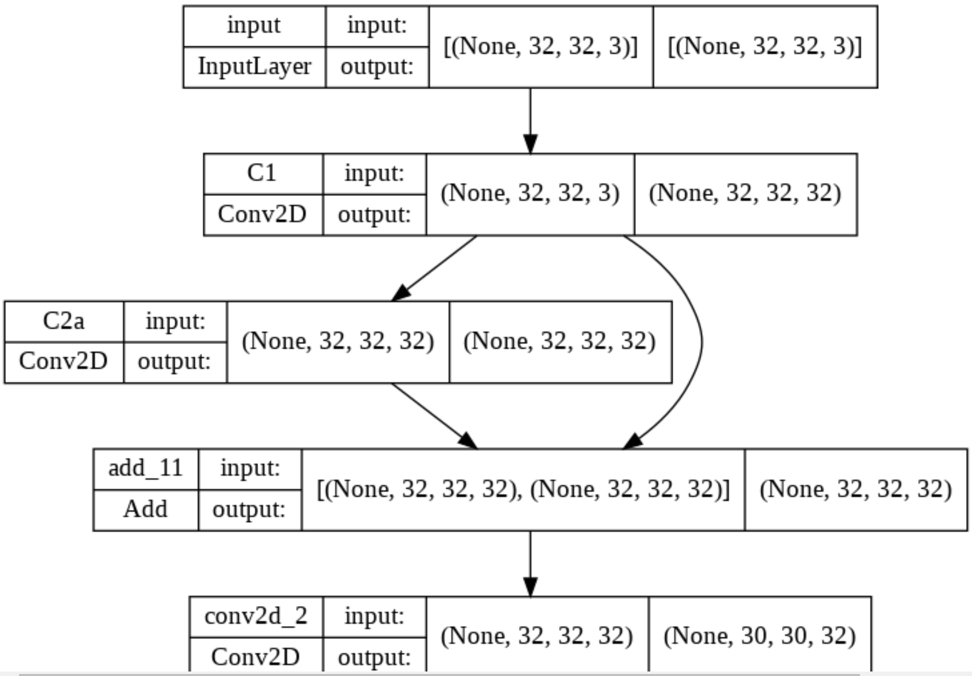
```
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.utils import plot_model
```

Implementing Convolutional Neural Networks

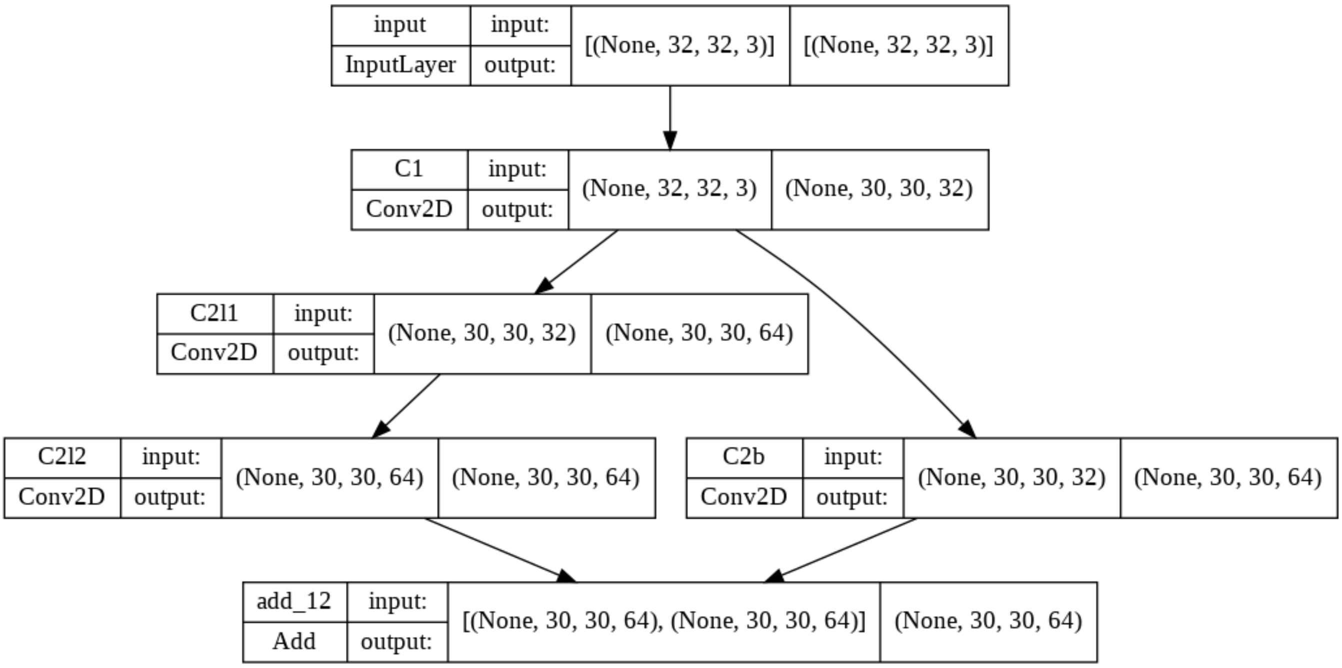
by Sanjeev Gupta

```
# Build model with residual connection

inputs = keras.Input(shape=(32, 32, 3), name="input")
x = layers.Conv2D(32, 3, activation="relu", padding="same", name="C1")(inputs)  # Q: No. of filters and kernel size? A:
residual = x
x = layers.Conv2D(32, 3, activation="relu", padding="same", name="C2a")(x)
x = layers.add([x, residual]   )   # Q: Do x and residual have the same shape? A:
x = layers.Conv2D(32, 3)(x)
model = keras.Model(inputs=inputs, outputs=x)
plot_model(model, show_shapes=True)
```



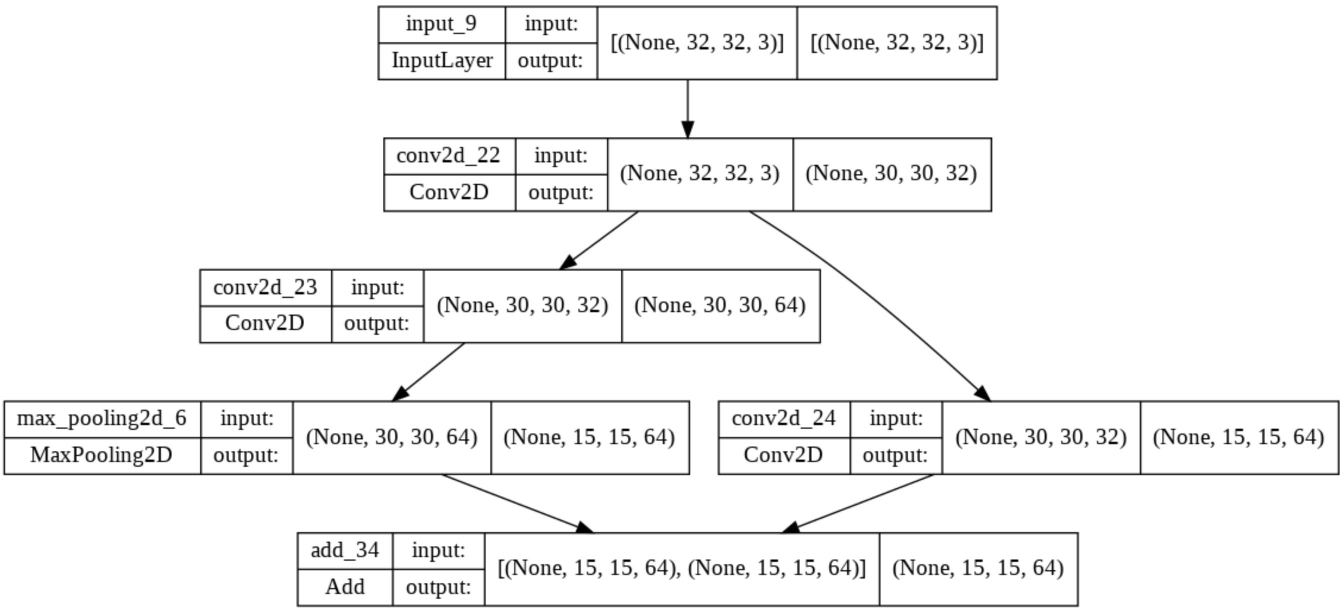
Residual branch may contain 1 layer to make sure addition is possible, i.e. accomodate sizes.

```
# Build model with residual connection - layer in the residual branch

inputs = keras.Input(shape=(32, 32, 3), name="input")
x = layers.Conv2D(32, 3, activation="relu", name="C1")(inputs)  # Q: No. of filters and kernel size?
residual = x
x = layers.Conv2D(64, 3, activation="relu", padding="same", name="C2l1")(x)
x = layers.Conv2D(64, 3, activation="relu", padding="same", name="C2l2")(x)

residual = layers.Conv2D(64, 1, name="C2b")(residual)          # Q: Why 1? A:
x = layers.add([x, residual]   )                      # Q: Do x and residual have the same shape?
model = keras.Model(inputs=inputs, outputs=x)


plot_model(model, show_shapes=True)
```



Important: Add layers of **same** shape !

```
# model with residual connections and a max pool layer in between
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Conv2D(32, 3, activation="relu")(inputs)
residual = x
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
x = layers.MaxPooling2D(2, padding="same")(x)  # Max pooling layer reduces the dimsion. #Q: By how much ? A:
residual = layers.Conv2D(64, 1, strides=2)(residual) #Need a stride=2 to accomodate for the maxpool downsampling in the other branch
x = layers.add([x, residual])
model = keras.Model(inputs=inputs, outputs=x)
plot_model(model, show_shapes=True)
```



With residual connections, you can build networks of arbitrary depth, without having to worry about vanishing gradients.

We will see an example later.

Intuitions on why residual blocks work:

- Shorter path for gradients

Batch Normalization

- Adaptively normalize data even as the mean and variance change over time during training
- During training, it uses the mean and variance of the current batch of data to normalize samples
- During inference (when a big enough batch of representative data may not be available), it uses an exponential moving average of the batch-wise mean and variance of the data seen during training.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\dots m}\}$;
Parameters to be learned: γ, β

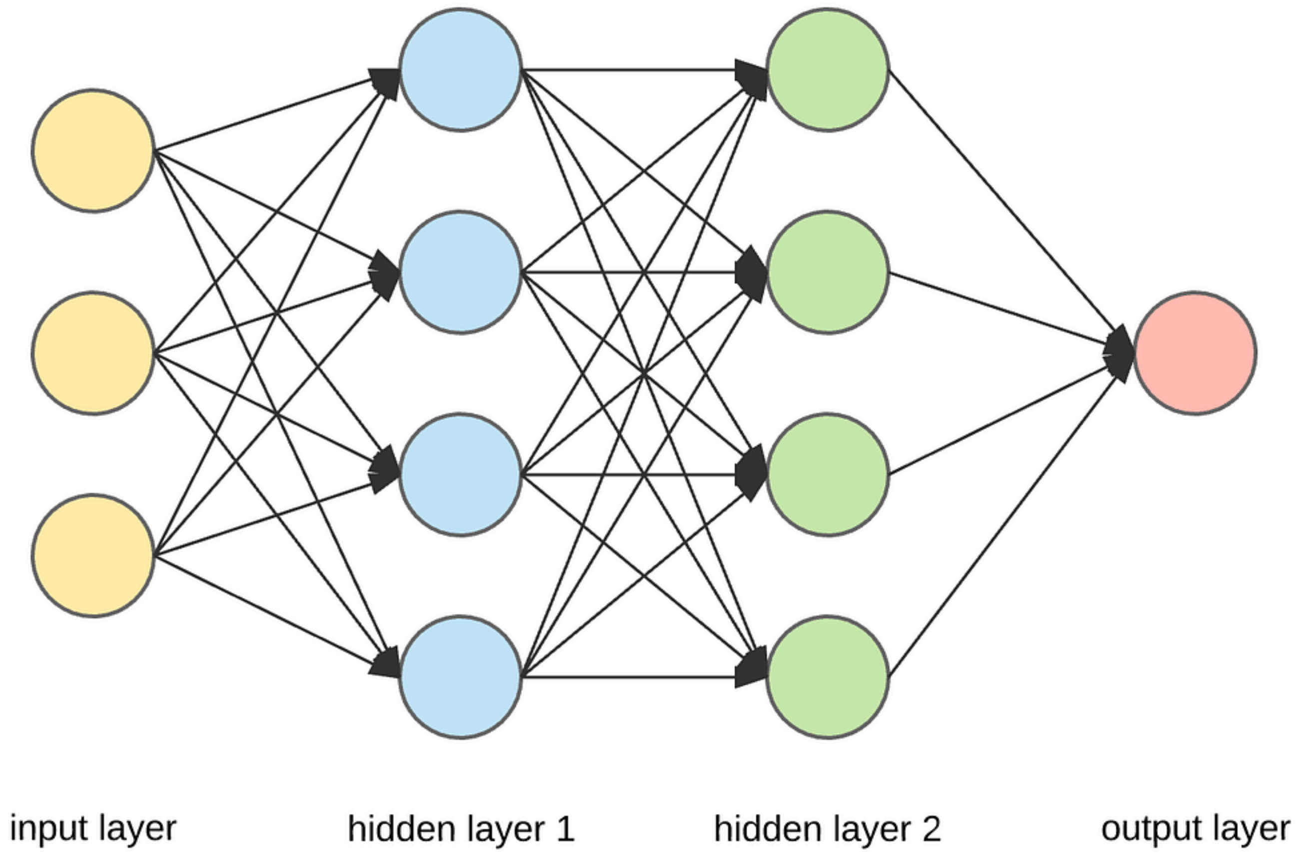
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

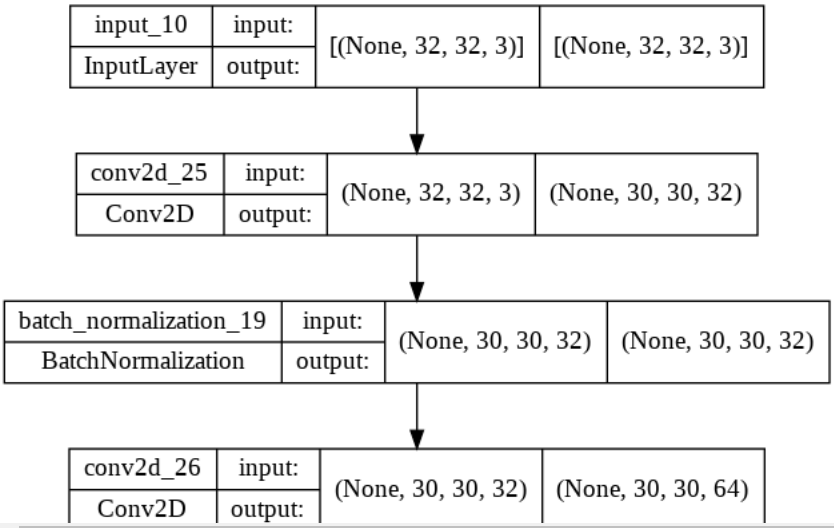


Now let's try to calculate the no. of params introduced because of batch normalization

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_3 (Flatten)          (None, 784)               0
_________________________________________________________________
batch_normalization_v2 (Batc (None, 784)               3136
_________________________________________________________________
dense_50 (Dense)             (None, 300)               235500
_________________________________________________________________
batch_normalization_v2_1 (Ba (None, 300)               1200
_________________________________________________________________
dense_51 (Dense)             (None, 100)               30100
_________________________________________________________________
batch_normalization_v2_2 (Ba (None, 100)               400
_________________________________________________________________
dense_52 (Dense)             (None, 10)                1010
=================================================================
Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368
```

```
# Model with a batch normalization layer
inputs = keras.Input(shape=(32, 32, 3))
# Because the output of the Conv2D layer gets normalized, the layer doesn’t need its own bias vector
```

```
x = layers.Conv2D(32, 3, activation="relu", use_bias=False)(inputs)
x = layers.BatchNormalization()(x)
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
model = keras.Model(inputs=inputs, outputs=x)
plot_model(model, show_shapes=True)
```




```
model.summary()
```

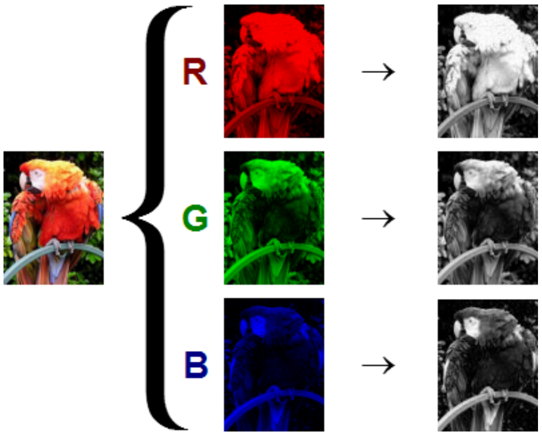
```
Model: "model_17"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_10 (InputLayer)       [(None, 32, 32, 3)]       0

 conv2d_25 (Conv2D)          (None, 30, 30, 32)        864

 batch_normalization_19 (Bat  (None, 30, 30, 32)       128
 chNormalization)

 conv2d_26 (Conv2D)          (None, 30, 30, 64)        18496

=================================================================
Total params: 19,488
Trainable params: 19,424
Non-trainable params: 64
_________________________________________________________________
```

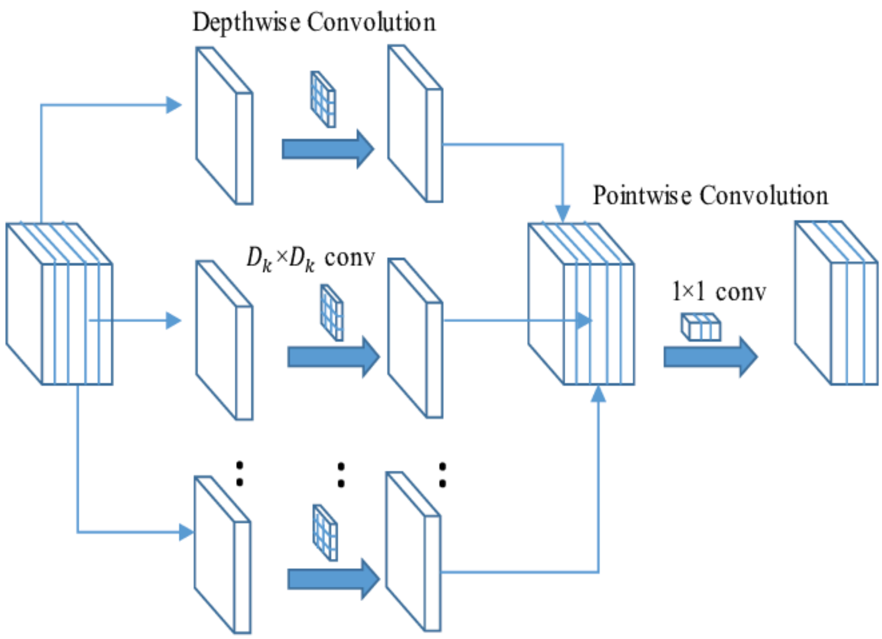
Intuitions:

- Batch Normalization is also a (weak) regularization method.
  - increases no. of params
  - but also adds noise ~ data augmentation ~ dropout

Depthwise separable convolutions

- This layer performs a spatial convolution on each channel of its input, independently, before mixing output channels via a pointwise convolution
- This makes your model smaller and acts as a strong prior
- We impose a strong prior by assuming that spatial patterns and cross-channel patterns can be modelled separately.
- This is equivalent to separating the learning of spatial features and the learning of channel-wise features.
- depthwise separable convolution relies on the assumption that spatial locations in intermediate activations are highly correlated, but different channels are highly independent.
- So we never use depthwise separable convolution after the input layer. Because RGB channels are **highly corelated**





Let's quickly look at the code first

```
# Building a model with Separable Conv layer
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Conv2D(32, 3, activation="relu")(inputs)
x = layers.SeparableConv2D(64, 3, activation="relu", padding="same")(x)
sep_model = keras.Model(inputs=inputs, outputs=x)
plot_model(sep_model, show_shapes=True)
sep_model.summary()
# Q: Verify the no. of params in the separable_conv2D layer A: 2400 = (32*9) + (32*1*1*64) + 64
```

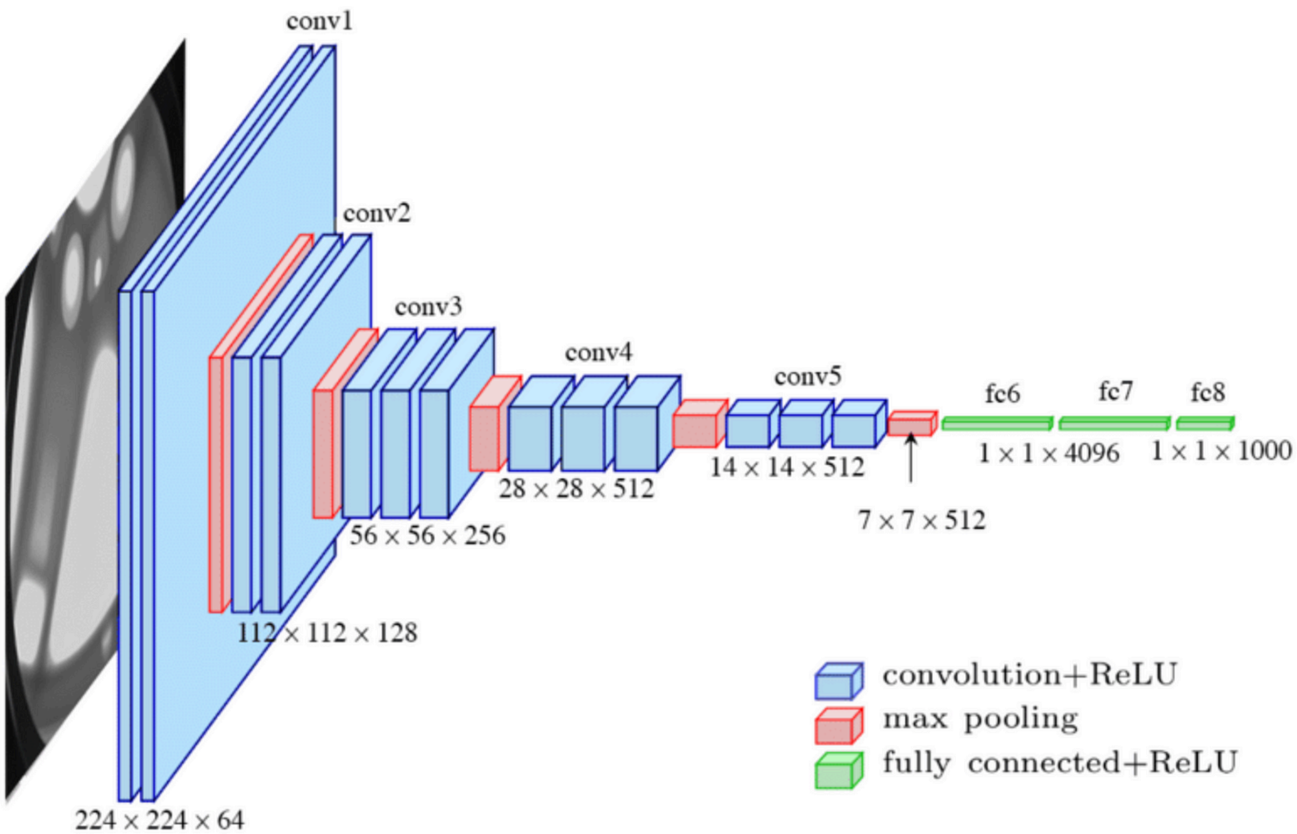
```
Model: "model_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 32, 32, 3)]       0

 conv2d_5 (Conv2D)           (None, 30, 30, 32)        896

 separable_conv2d (Separable  (None, 30, 30, 64)       2400
 Conv2D)

=================================================================
Total params: 3,296
Trainable params: 3,296
Non-trainable params: 0
_________________________________________________________________
```

Let's compoare the above with a model where we replace the SeparableConv2D with a Conv2D layer

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Conv2D(32, 3, activation="relu")(inputs)
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
model = keras.Model(inputs=inputs, outputs=x)
```

```
plot_model(model, show_shapes=True)
model.summary()
# Q: Why does sep_model have much less params? A: Depthwise and pointwise convs are done independently
```

```
Model: "model_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_4 (InputLayer)        [(None, 32, 32, 3)]       0

 conv2d_6 (Conv2D)           (None, 30, 30, 32)        896

 conv2d_7 (Conv2D)           (None, 30, 30, 64)        18496

=================================================================
Total params: 19,392
Trainable params: 19,392
Non-trainable params: 0
_________________________________________________________________
```

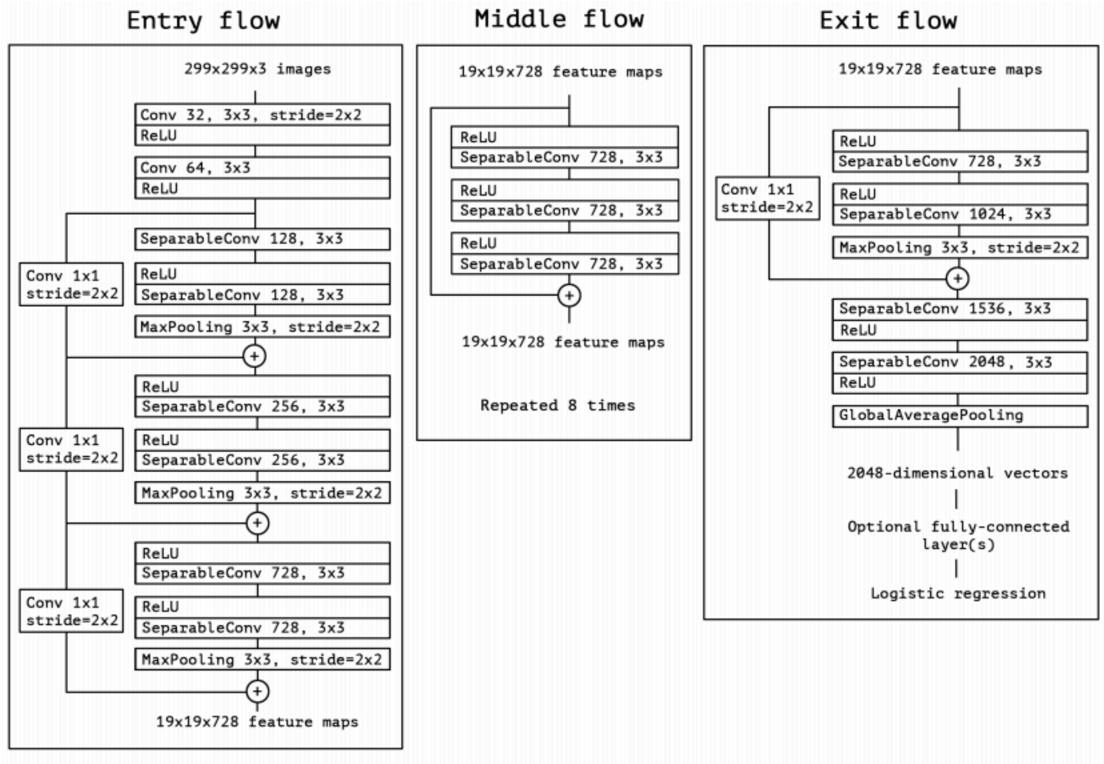


Start coding or generate with AI.

A mini Xception-like model

We'll build a model like the Xception model, but a smaller version.

But first let's see what the actual Xception model looks like.



Q: In middle- flow blocks, what arguments do you give to the SepConv layer ?

A: HW question

Let's download the cats-vs-dogs data again and create datasets.

```
# Download the dataset
!git clone -b master https://github.com/sumanthk94/DA_225-o.git
```

```
Cloning into 'DA_225-o'...
remote: Enumerating objects: 19060, done.
remote: Counting objects: 100% (5024/5024), done.
remote: Compressing objects: 100% (5021/5021), done.
remote: Total 19060 (delta 0), reused 5024 (delta 0), pack-reused 14036
Receiving objects: 100% (19060/19060), 570.44 MiB | 25.44 MiB/s, done.
```

```
# defining path names for futur use
data_dir = 'DA_225-o/cats_vs_dogs_small'

train_path = data_dir + '/train'
validation_path = data_dir + '/validation'
test_path = data_dir + '/test'
```

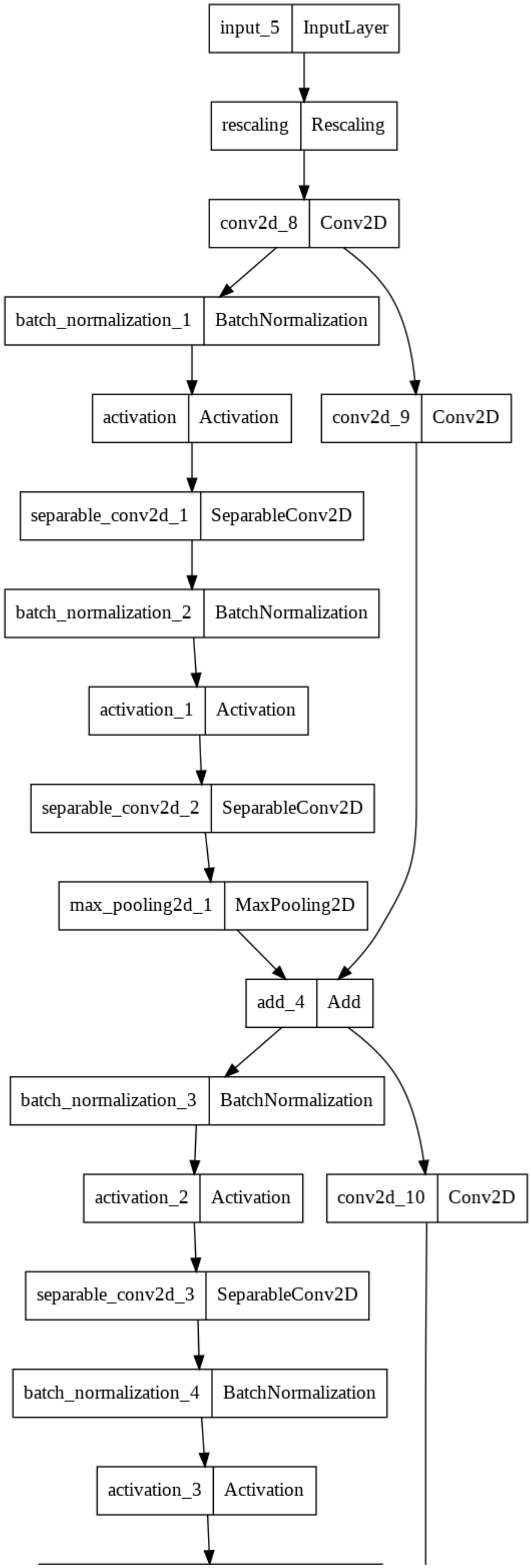
```
# creating datasets using utility
from tensorflow.keras.utils import image_dataset_from_directory
train_dataset = image_dataset_from_directory(
               train_path,
                image_size=(180, 180), # Resize the images to (180,180)
                batch_size=32)
validation_dataset = image_dataset_from_directory(
                      validation_path,
                      image_size=(180, 180),
                      batch_size=32)
test_dataset = image_dataset_from_directory(
                test_path,
                image_size=(180, 180),
                batch_size=32)
```

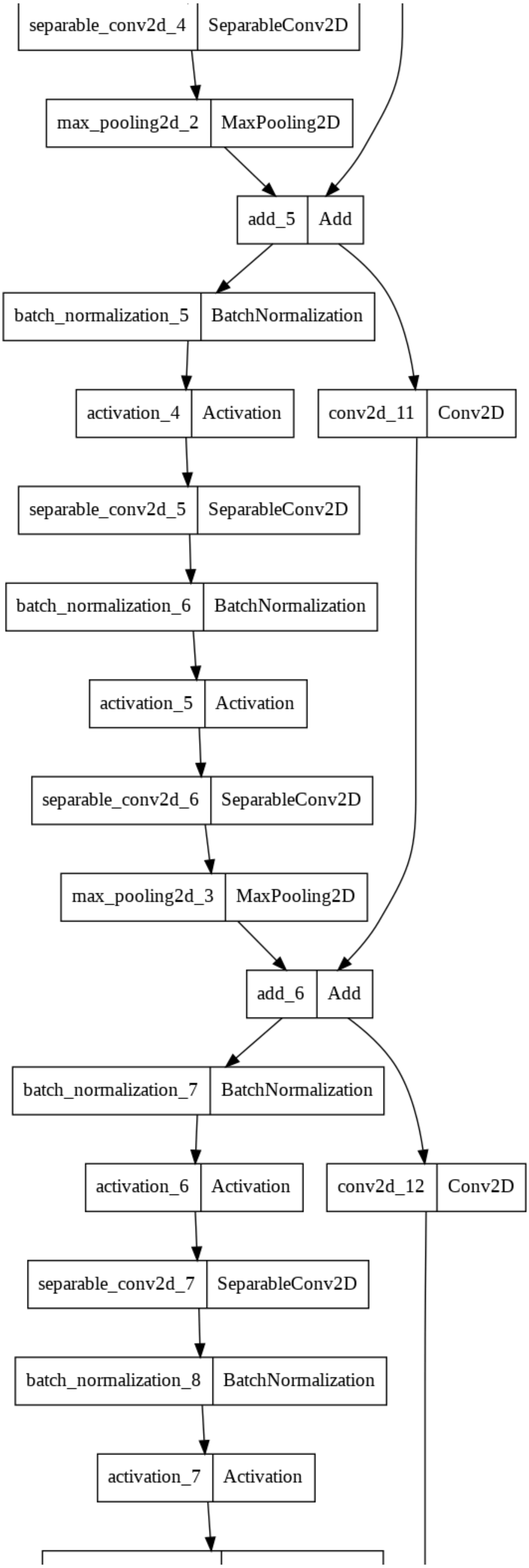
```
Found 2000 files belonging to 2 classes.
Found 1000 files belonging to 2 classes.
Found 2000 files belonging to 2 classes.
```

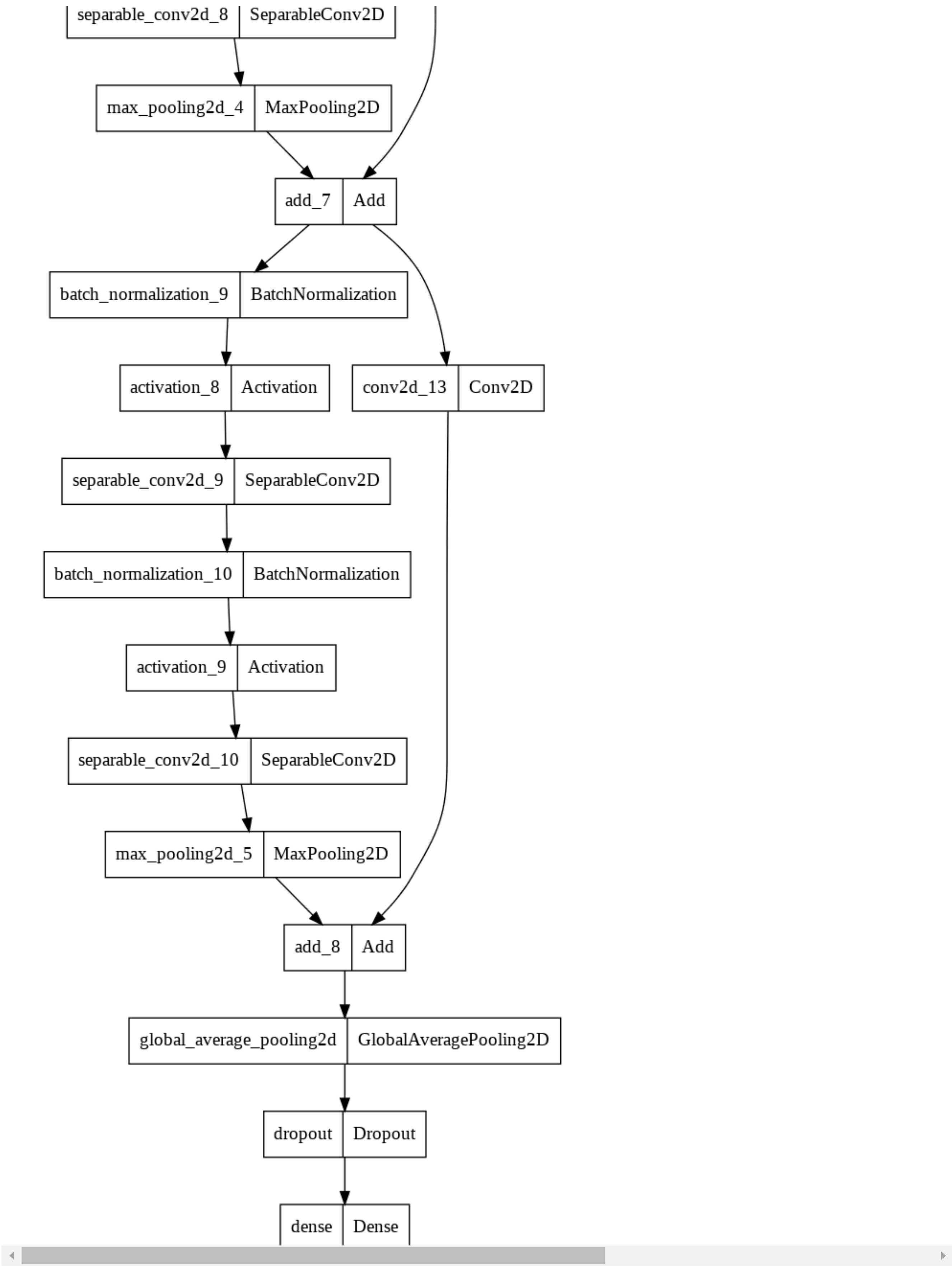
```
import keras
from keras import layers
inputs = keras.Input(shape=(180, 180, 3))
# x = data_augmentation(inputs)
```

```
x = layers.Rescaling(1./255)(inputs)
x = layers.Conv2D(filters=32, kernel_size=5, use_bias=False)(x) # Q: why not use depth-wise sep conv here? A: RGB channels in input imag

for size in [32, 64, 128, 256, 512]:                # Repeated block. Very common practice
    residual = x

    x = layers.BatchNormalization()(x)              # We can also apply BN just before the activation
    x = layers.Activation("relu")(x)
    x = layers.SeparableConv2D(size, 3, padding="same", use_bias=False)(x)

    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)
    x = layers.SeparableConv2D(size, 3, padding="same", use_bias=False)(x)

    x = layers.MaxPooling2D(3, strides=2, padding="same")(x)

    residual = layers.Conv2D(
        size, 1, strides=2, padding="same", use_bias=False)(residual)
    x = layers.add([x, residual])

x = layers.GlobalAveragePooling2D()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

```
plot_model(model)
```


input_5 | InputLayer
rescaling | Rescaling
conv2d_8 | Conv2D
batch_normalization_1 | BatchNormalization
activation | Activation
conv2d_9 | Conv2D
separable_conv2d_1 | SeparableConv2D
batch_normalization_2 | BatchNormalization
activation_1 | Activation
separable_conv2d_2 | SeparableConv2D
max_pooling2d_1 | MaxPooling2D
add_4 | Add
batch_normalization_3 | BatchNormalization
activation_2 | Activation
conv2d_10 | Conv2D
separable_conv2d_3 | SeparableConv2D
batch_normalization_4 | BatchNormalization
activation_3 | Activation

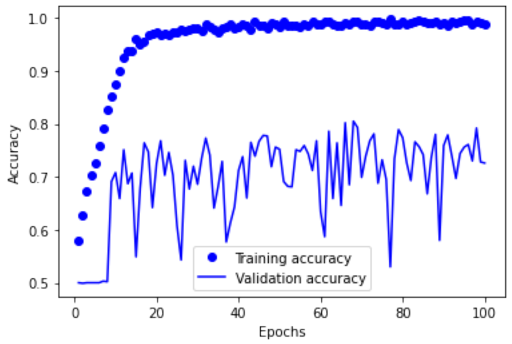

separable_conv2d_4 | SeparableConv2D
max_pooling2d_2 | MaxPooling2D
add_5 | Add
batch_normalization_5 | BatchNormalization
activation_4 | Activation
conv2d_11 | Conv2D
separable_conv2d_5 | SeparableConv2D
batch_normalization_6 | BatchNormalization
activation_5 | Activation
separable_conv2d_6 | SeparableConv2D
max_pooling2d_3 | MaxPooling2D
add_6 | Add
batch_normalization_7 | BatchNormalization
activation_6 | Activation
conv2d_12 | Conv2D
separable_conv2d_7 | SeparableConv2D
batch_normalization_8 | BatchNormalization
activation_7 | Activation

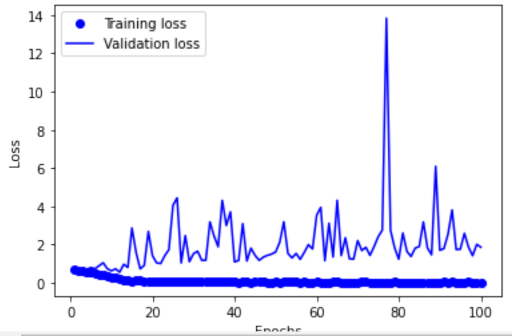

separable_conv2d_8 | SeparableConv2D
max_pooling2d_4 | MaxPooling2D
add_7 | Add
batch_normalization_9 | BatchNormalization
activation_8 | Activation
conv2d_13 | Conv2D
separable_conv2d_9 | SeparableConv2D
batch_normalization_10 | BatchNormalization
activation_9 | Activation
separable_conv2d_10 | SeparableConv2D
max_pooling2d_5 | MaxPooling2D
add_8 | Add
global_average_pooling2d | GlobalAveragePooling2D
dropout | Dropout
dense | Dense

```
model.compile(loss="binary_crossentropy",
              optimizer="rmsprop",
              metrics=["accuracy"])
history = model.fit( train_dataset,
                     epochs=100,
                     validation_data=validation_dataset)
```

```
Epoch 72/100
63/63 [==============================] - 16s 254ms/step - loss: 0.0436 - accuracy: 0.9855 - val_loss: 1.8592 - val_accuracy: 0.76
Epoch 73/100
63/63 [==============================] - 16s 259ms/step - loss: 0.0213 - accuracy: 0.9930 - val_loss: 1.4377 - val_accuracy: 0.78
Epoch 74/100
63/63 [==============================] - 16s 256ms/step - loss: 0.0262 - accuracy: 0.9925 - val_loss: 1.8999 - val_accuracy: 0.68
Epoch 75/100
63/63 [==============================] - 16s 259ms/step - loss: 0.0238 - accuracy: 0.9915 - val_loss: 2.4129 - val_accuracy: 0.73
Epoch 76/100
63/63 [==============================] - 16s 255ms/step - loss: 0.0348 - accuracy: 0.9890 - val_loss: 2.7587 - val_accuracy: 0.69
Epoch 77/100
63/63 [==============================] - 16s 256ms/step - loss: 0.0099 - accuracy: 0.9975 - val_loss: 13.8351 - val_accuracy: 0.5
Epoch 78/100
63/63 [==============================] - 16s 261ms/step - loss: 0.0286 - accuracy: 0.9890 - val_loss: 2.7523 - val_accuracy: 0.73
Epoch 79/100
63/63 [==============================] - 16s 256ms/step - loss: 0.0458 - accuracy: 0.9885 - val_loss: 1.8347 - val_accuracy: 0.78
Epoch 80/100
63/63 [==============================] - 16s 257ms/step - loss: 0.0238 - accuracy: 0.9930 - val_loss: 1.2373 - val_accuracy: 0.77
Epoch 81/100
63/63 [==============================] - 16s 255ms/step - loss: 0.0357 - accuracy: 0.9885 - val_loss: 2.6036 - val_accuracy: 0.72
Epoch 82/100
63/63 [==============================] - 16s 260ms/step - loss: 0.0276 - accuracy: 0.9915 - val_loss: 1.6421 - val_accuracy: 0.69
Epoch 83/100
63/63 [==============================] - 16s 260ms/step - loss: 0.0207 - accuracy: 0.9940 - val_loss: 1.3811 - val_accuracy: 0.76
Epoch 84/100
63/63 [==============================] - 16s 260ms/step - loss: 0.0252 - accuracy: 0.9945 - val_loss: 1.8127 - val_accuracy: 0.75
Epoch 85/100
63/63 [==============================] - 16s 258ms/step - loss: 0.0171 - accuracy: 0.9930 - val_loss: 1.8998 - val_accuracy: 0.74
Epoch 86/100
63/63 [==============================] - 16s 258ms/step - loss: 0.0357 - accuracy: 0.9900 - val_loss: 3.1893 - val_accuracy: 0.66
Epoch 87/100
63/63 [==============================] - 16s 258ms/step - loss: 0.0345 - accuracy: 0.9910 - val_loss: 1.8249 - val_accuracy: 0.73
Epoch 88/100
63/63 [==============================] - 16s 256ms/step - loss: 0.0132 - accuracy: 0.9935 - val_loss: 1.4656 - val_accuracy: 0.78
Epoch 89/100
63/63 [==============================] - 16s 260ms/step - loss: 0.0320 - accuracy: 0.9875 - val_loss: 6.0967 - val_accuracy: 0.58
Epoch 90/100
63/63 [==============================] - 16s 258ms/step - loss: 0.0264 - accuracy: 0.9915 - val_loss: 1.7044 - val_accuracy: 0.75
Epoch 91/100
63/63 [==============================] - 16s 260ms/step - loss: 0.0490 - accuracy: 0.9865 - val_loss: 1.7897 - val_accuracy: 0.77
Epoch 92/100
63/63 [==============================] - 16s 257ms/step - loss: 0.0293 - accuracy: 0.9920 - val_loss: 2.5363 - val_accuracy: 0.73
Epoch 93/100
63/63 [==============================] - 16s 259ms/step - loss: 0.0407 - accuracy: 0.9900 - val_loss: 3.8167 - val_accuracy: 0.69
Epoch 94/100
63/63 [==============================] - 16s 254ms/step - loss: 0.0279 - accuracy: 0.9920 - val_loss: 1.7427 - val_accuracy: 0.74
Epoch 95/100
63/63 [==============================] - 16s 258ms/step - loss: 0.0120 - accuracy: 0.9965 - val_loss: 1.7606 - val_accuracy: 0.75
Epoch 96/100
63/63 [==============================] - 16s 257ms/step - loss: 0.0105 - accuracy: 0.9960 - val_loss: 2.5897 - val_accuracy: 0.76
Epoch 97/100
63/63 [==============================] - 16s 259ms/step - loss: 0.0490 - accuracy: 0.9885 - val_loss: 1.8709 - val_accuracy: 0.73
Epoch 98/100
63/63 [==============================] - 16s 256ms/step - loss: 0.0205 - accuracy: 0.9935 - val_loss: 1.4192 - val_accuracy: 0.79
Epoch 99/100
63/63 [==============================] - 16s 256ms/step - loss: 0.0311 - accuracy: 0.9905 - val_loss: 2.0145 - val_accuracy: 0.72
Epoch 100/100
```

```
data = pd.DataFrame(history.history)
plt.plot(range(1,len(data)+1),data['accuracy'],'bo',label="Training accuracy")
plt.plot(range(1,len(data)+1),data['val_accuracy'],'b',label="Validation accuracy")
plt.legend()
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.show()
plt.figure()
plt.plot(range(1,len(data)+1),data['loss'],'bo',label="Training loss")
plt.plot(range(1,len(data)+1),data['val_loss'],'b',label="Validation loss")
plt.legend()
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.show()
```



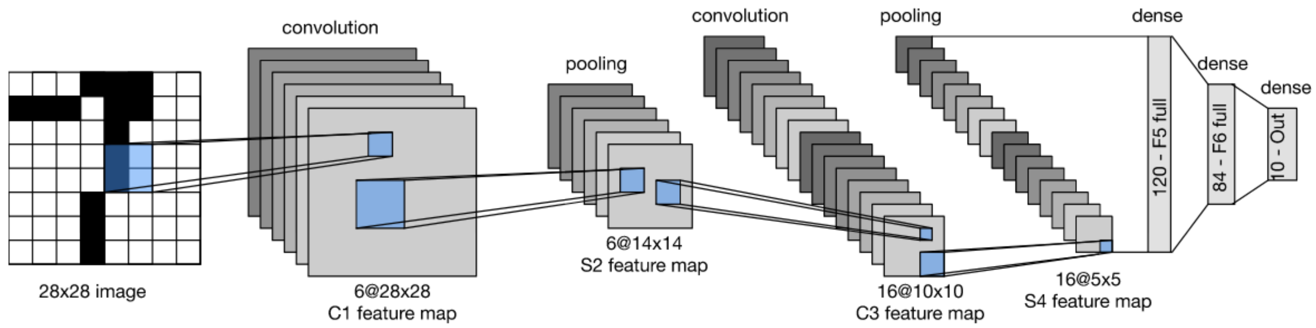



```
Start coding or generate with AI.
```

Visualizing intermediate activations

The output of a layer is called its 'activation'. (It's the output of the activation function)

These activations can be visualized by plotting the feature maps.

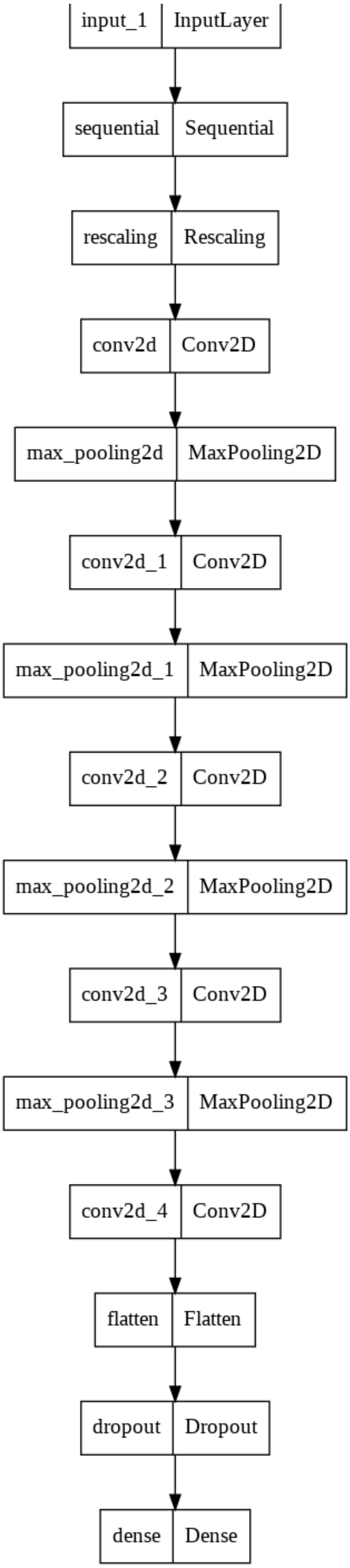


We will plot each feature map independently as a 2D image, since they encode relatively indepent features

```
from tensorflow import keras
from tensorflow.keras.utils import plot_model
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras import layers  # <----- Note this


from google.colab import drive
drive.mount('/content/drive')
%cd "/content/drive/My Drive/"
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
/content/drive/My Drive
```

```
model = keras.models.load_model('/content/drive/MyDrive/Colab Notebooks/convnet_from_scratch_with_augmentation.keras')
plot_model(model)
```



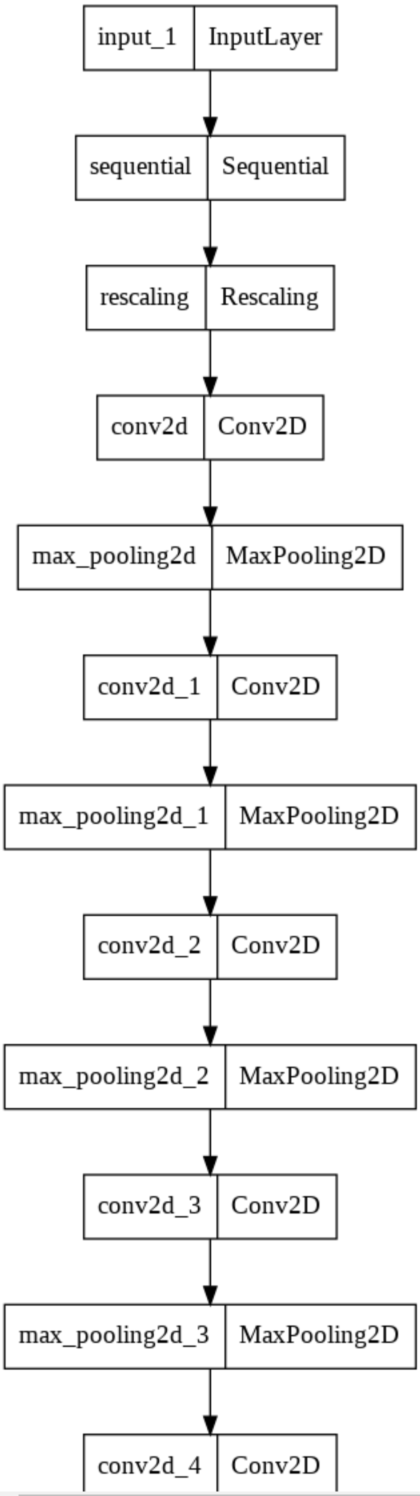

```
# test_loss, test_acc = model.evaluate(test_dataset) #Q: Why don't we have 2 arguments
# print(f"Test accuracy is:{test_acc:.3f}")
```

```
# Preprocessing a single image
```

```
img_path = keras.utils.get_file(fname="cat.jpg",
                                origin="https://img-datasets.s3.amazonaws.com/cat.jpg")

def get_img_array(img_path, target_size):
    img = keras.utils.load_img(img_path, target_size=target_size)
    array = keras.utils.img_to_array(img) # converts image to np array
    # Add a dimension to transform the array intoa “batch” of a single sample.
    array = np.expand_dims(array, axis=0)     #Its shape is now (1, 180, 180, 3)
    return array

img_tensor = get_img_array(img_path, target_size=(180, 180)) #resize image

plt.axis("off")
plt.imshow(img_tensor[0].astype("uint8"))
plt.show()
```

```
Downloading data from https://img-datasets.s3.amazonaws.com/cat.jpg
81920/80329 [==============================] - 0s 2us/step
90112/80329 [=================================] - 0s 1us/step
```

```
# Focus here
# Instantiating a model that returns "layer activations"
layer_outputs = []
layer_names = []
for layer in model.layers:                                      #Q: what does model.layers return ? A: lis of layers of the model
    if isinstance(layer, (layers.Conv2D, layers.MaxPooling2D)): #Q: differnce between layer and layers? A: layer is a layer object in tl
        layer_outputs.append(layer.output)
        # layer_outputs.append(layers.Dense(1)(layer.output))  # To visualise the output in plot_model()
        layer_names.append(layer.name)
activation_model = keras.Model(inputs=model.input, outputs=layer_outputs) #Q: what does layer_outputs contain? A: output of conv2d and r
plot_model(activation_model)
```

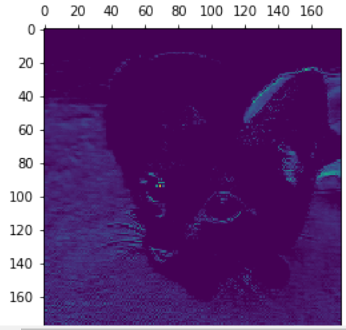



```
# Compute layer activations
activations = activation_model.predict(img_tensor)
print(f"No. of outputs= {len(activations)}")

first_layer_feature_maps = activations[0]    # Q: why is this the first layer's activation? A: index 0
print(f"first_layer_activation.shape= {first_layer_feature_maps.shape}")
```

```
No. of outputs= 9
first_layer_activation.shape= (1, 178, 178, 32)
```

```
# Visualise activation
import matplotlib.pyplot as plt
plt.matshow(first_layer_feature_maps[0, :, :, 0], cmap="viridis")  # Q: which (1st/2nd/..)feature map are we visualing? A: 1st (0 in the
```

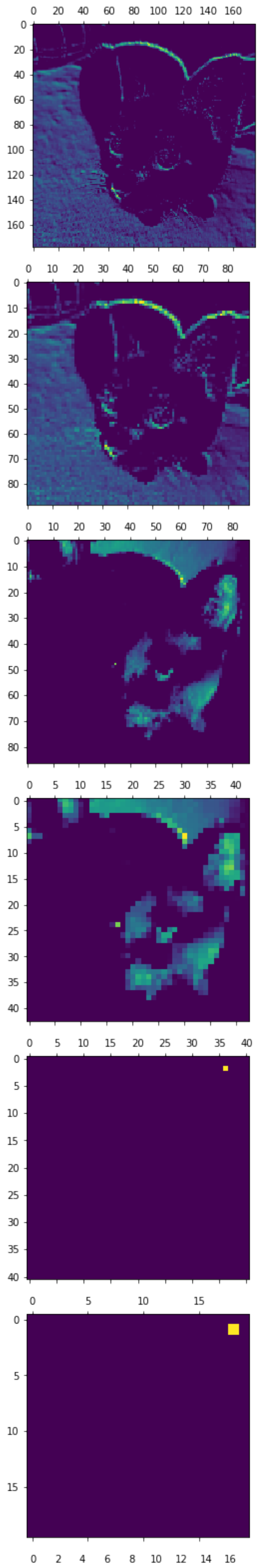
```
<matplotlib.image.AxesImage at 0x7f71509bf9d0>
```

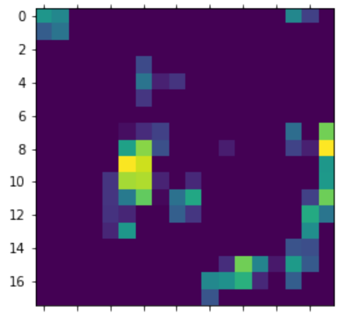


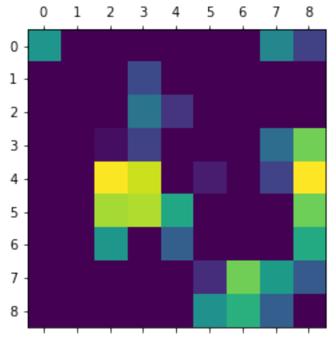
It seems that the filter has detected __.

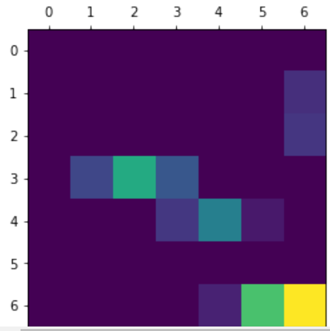
Let's look at a feature map after each layer.

```
for i in range(9):
  plt.matshow(activations[i][0, :, :, 2], cmap="viridis")
```







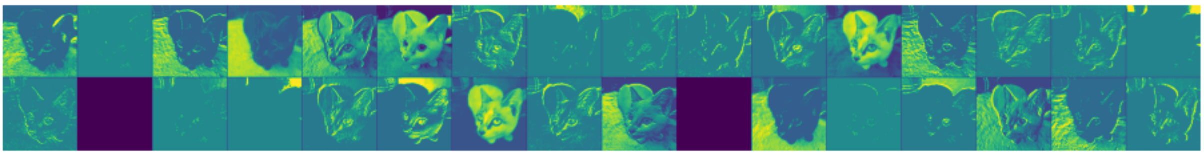


Note the dimensions on the above images. Successive feature maps are actually of smaller dimensions but scaled to be the same size during visualization.

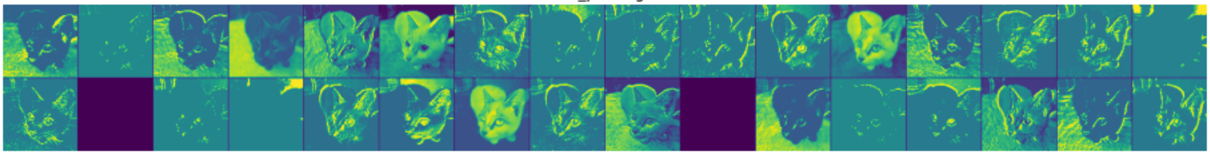
Now let's visualise all the feature maps of all the layers

```
# Post-processing code - only visualizaton
# Visualizing every channel in every intermediate activation
images_per_row = 16
for layer_name, layer_activation in zip(layer_names, activations):
    n_features = layer_activation.shape[-1]
    size = layer_activation.shape[1]
    n_cols = n_features // images_per_row
    display_grid = np.zeros(((size + 1) * n_cols - 1,
                             images_per_row * (size + 1) - 1))
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_index = col * images_per_row + row
            channel_image = layer_activation[0, :, :, channel_index].copy()
            if channel_image.sum() != 0:
                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 + 1): (col + 1) * size + col,
                row * (size + 1) : (row + 1) * size + row] = channel_image
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                        scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.axis("off")
    plt.imshow(display_grid, aspect="auto", cmap="viridis")
```

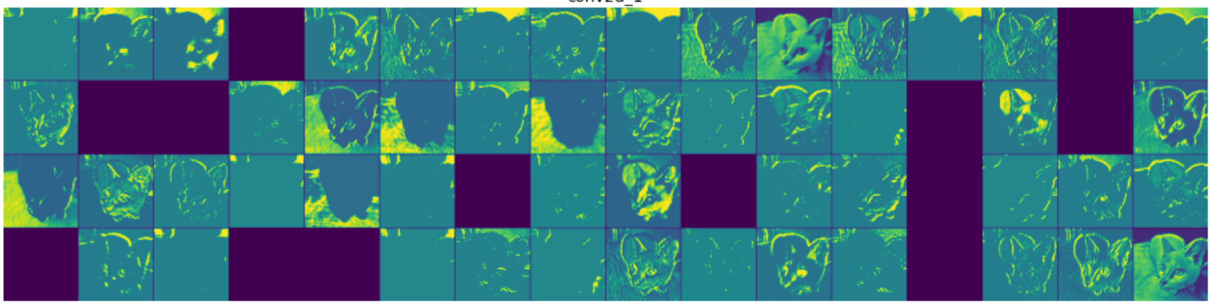
conv2d



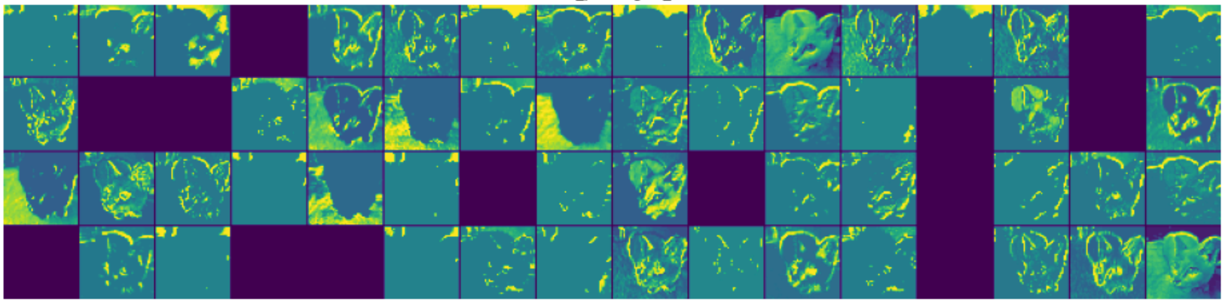
max_pooling2d



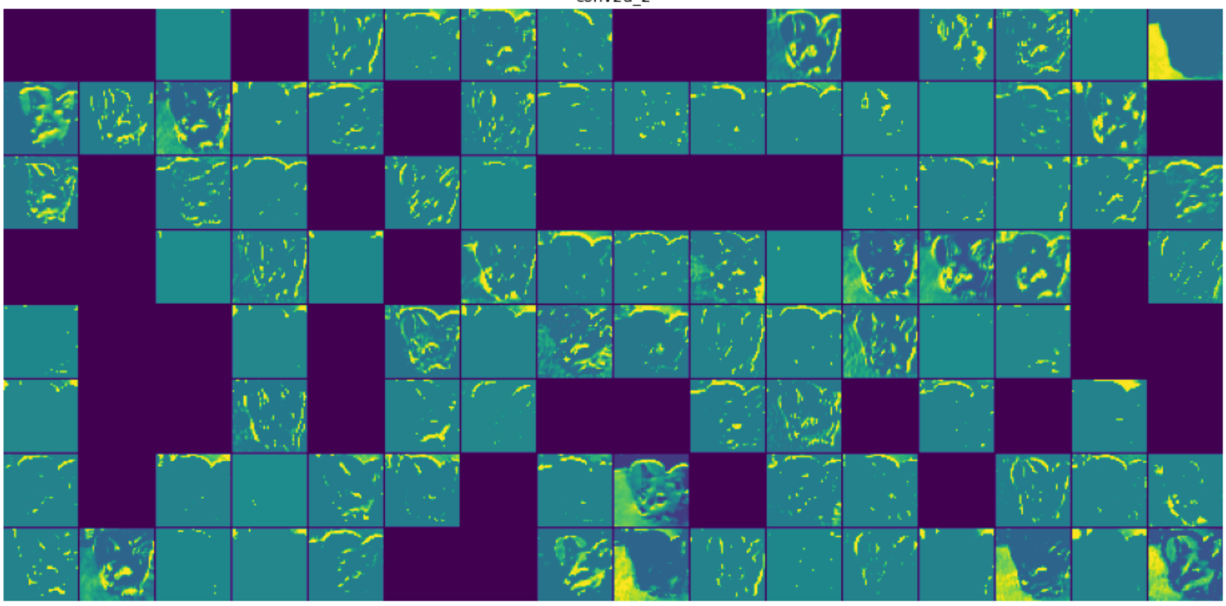
conv2d_1



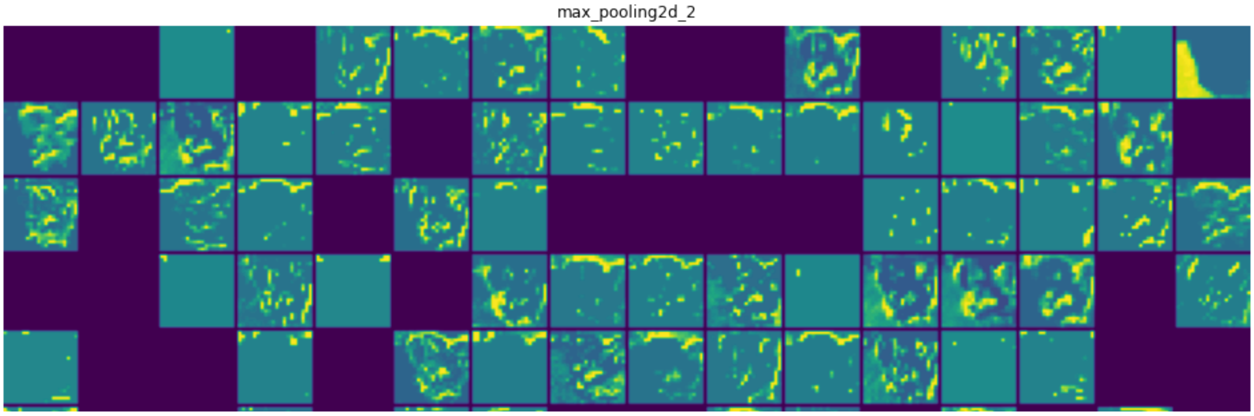
max_pooling2d_1

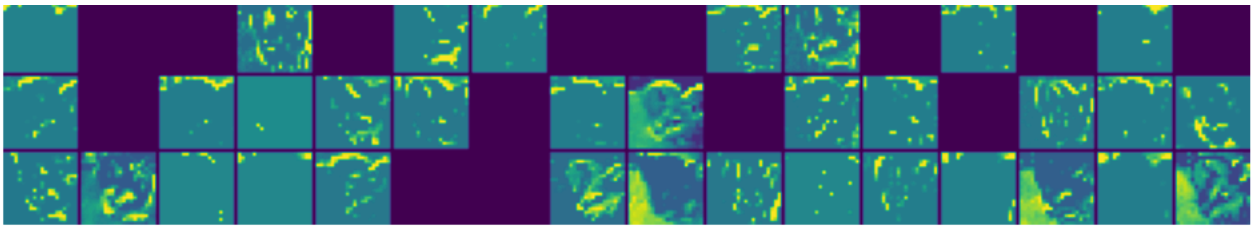


conv2d_2

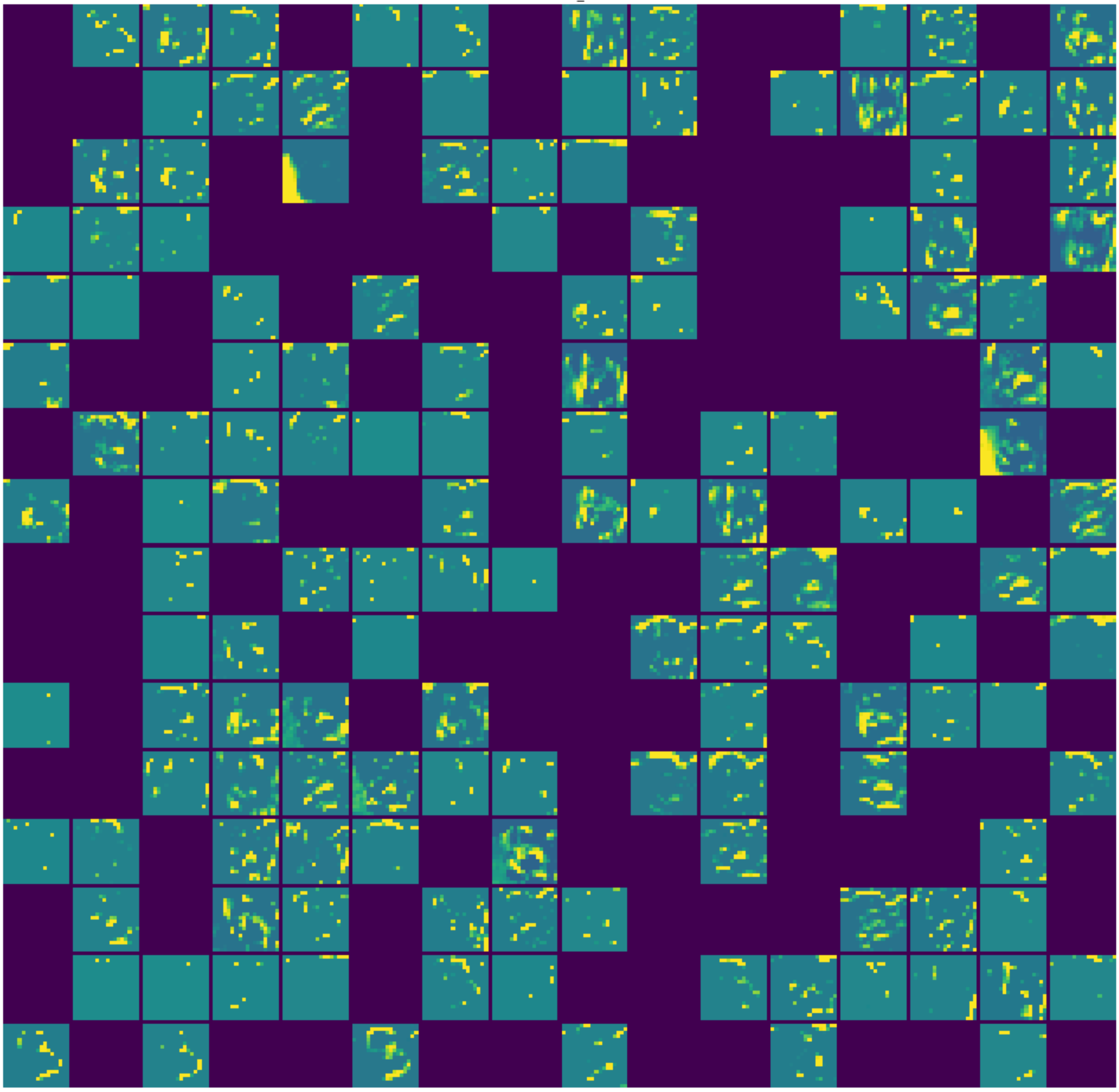


max_pooling2d_2

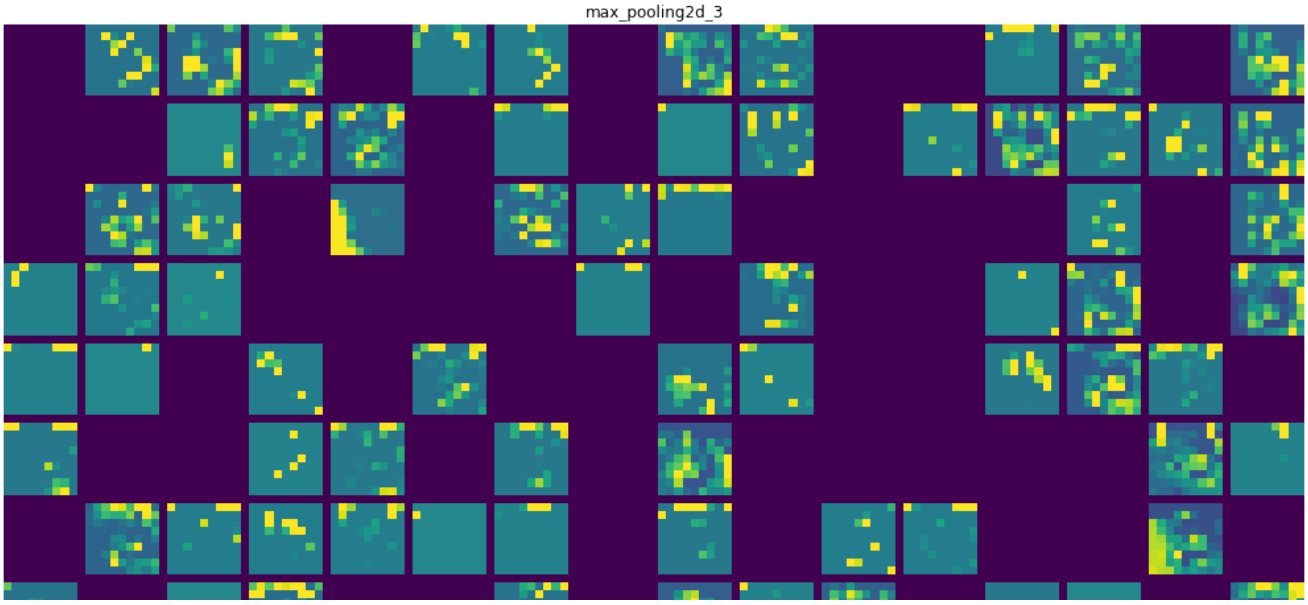


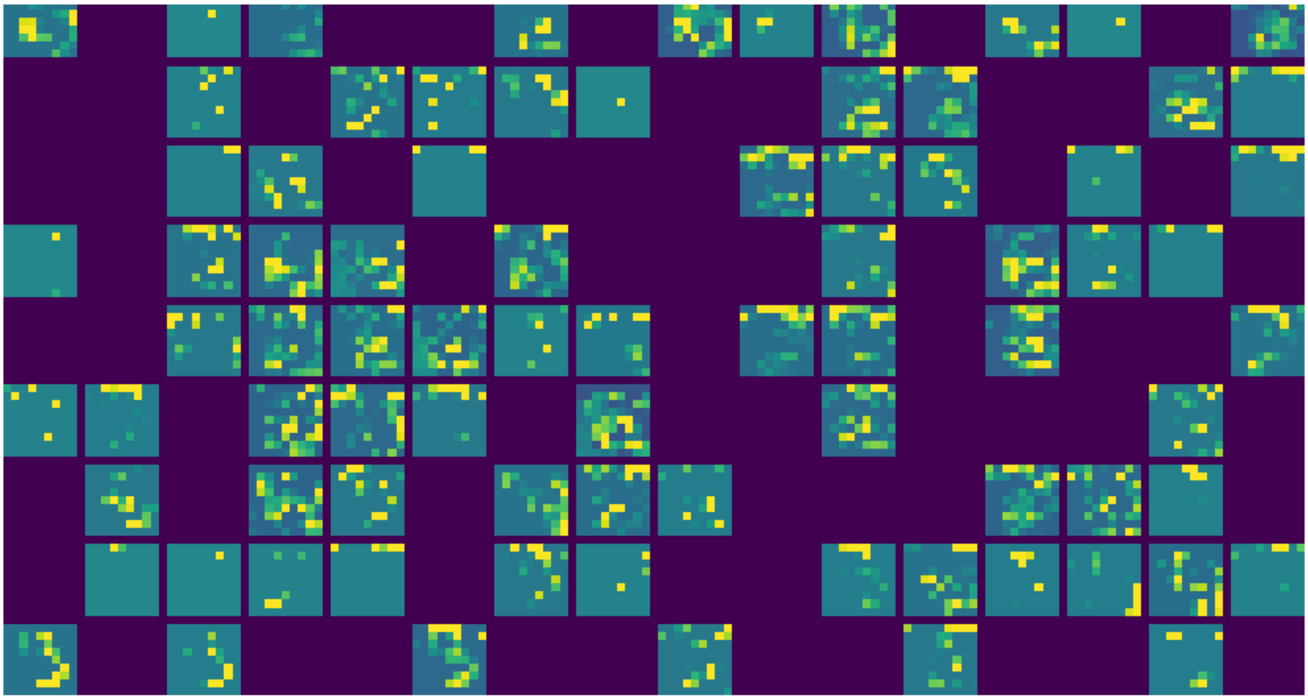


conv2d_3

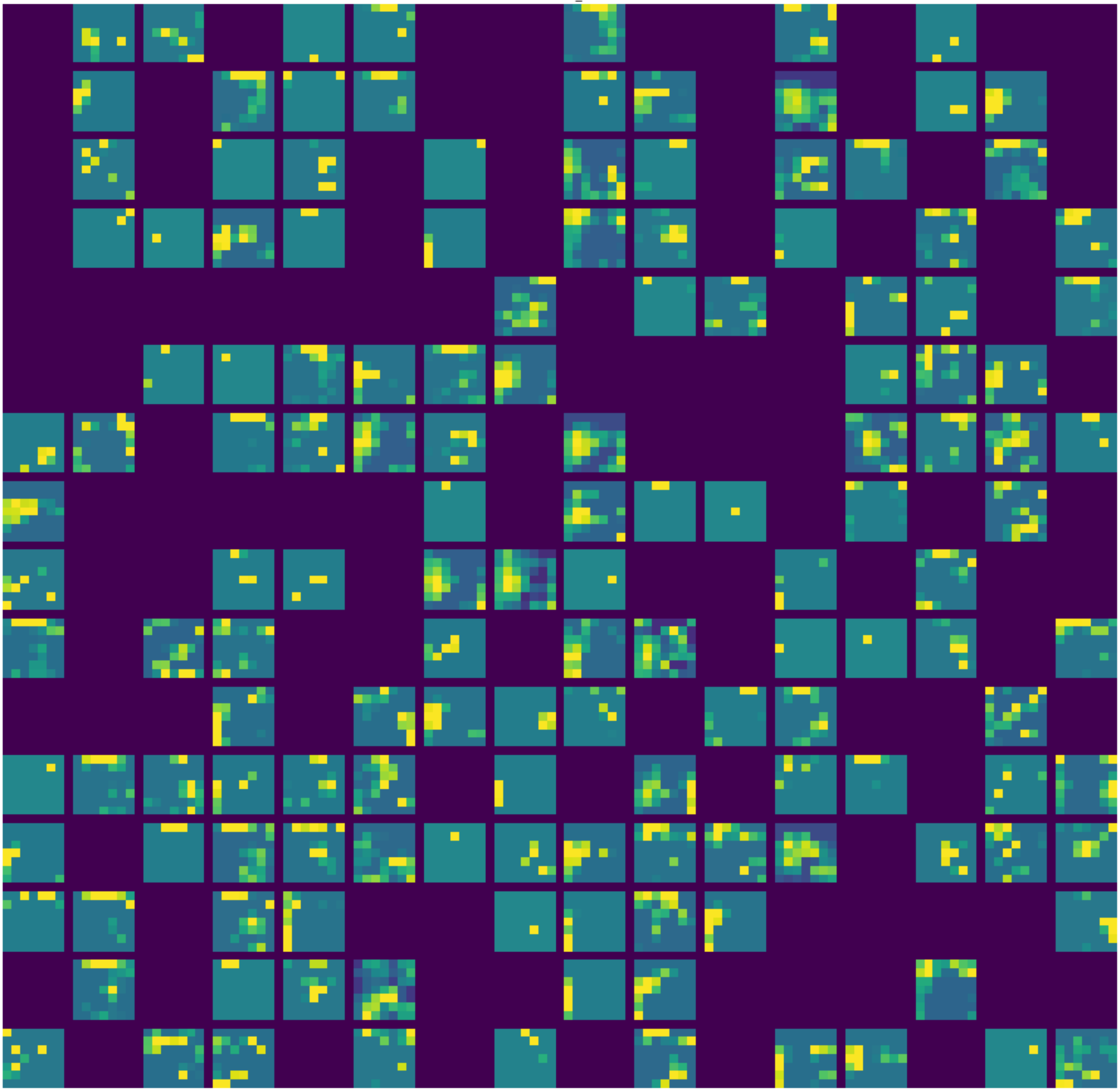


max_pooling2d_3





conv2d_4



- The first layer acts as a collection of various edge detectors.
- As you go deeper, the activations become increasingly abstract and less visually interpretable. They begin to encode higher-level concepts such as “cat ear” and “cat eye.”
- The sparsity of the activations increases with the depth of the layer: in the first layer, almost all filters are activated by the input image, but in the following layers, more and more filters are blank. This means the pattern encoded by the filter isn’t found in the input image

Visualising convnet filters

- Pick a filter
- Ask the question: What kind of an input image will excite the filter?
- What should the input image be so that you see a (yellow) feature map ?
- In other words, we want to visualize those patterns in the input image that filter picks up and results in high (yellow) values in the feature map.

```
# Instantiating the Xception convolutional base

model = keras.applications.xception.Xception(
    weights="imagenet",
    include_top=False)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/xception/xception_weights_tf_dim_ordering_tf_kern
83689472/83683744 [==============================] - 0s 0us/step
83697664/83683744 [==============================] - 0s 0us/step
```

```
# Q: Printing the names of  conv and sepConv layers in Xception

for layer in model.layers:
    if isinstance(layer, (keras.layers.Conv2D, keras.layers.SeparableConv2D)):
        print(layer.name)
```

```
block1_conv1
block1_conv2
block2_sepconv1
block2_sepconv2
conv2d_14
block3_sepconv1
block3_sepconv2
conv2d_15
block4_sepconv1
block4_sepconv2
conv2d_16
block5_sepconv1
block5_sepconv2
block5_sepconv3
block6_sepconv1
block6_sepconv2
block6_sepconv3
block7_sepconv1
block7_sepconv2
block7_sepconv3
block8_sepconv1
block8_sepconv2
block8_sepconv3
block9_sepconv1
block9_sepconv2
block9_sepconv3
block10_sepconv1
block10_sepconv2
block10_sepconv3
block11_sepconv1
block11_sepconv2
block11_sepconv3
block12_sepconv1
block12_sepconv2
block12_sepconv3
block13_sepconv1
block13_sepconv2
conv2d_17
block14_sepconv1
block14_sepconv2
```

```
# Creating a feature extractor model


layer_name = "block3_sepconv1"
layer = model.get_layer(name=layer_name)
```