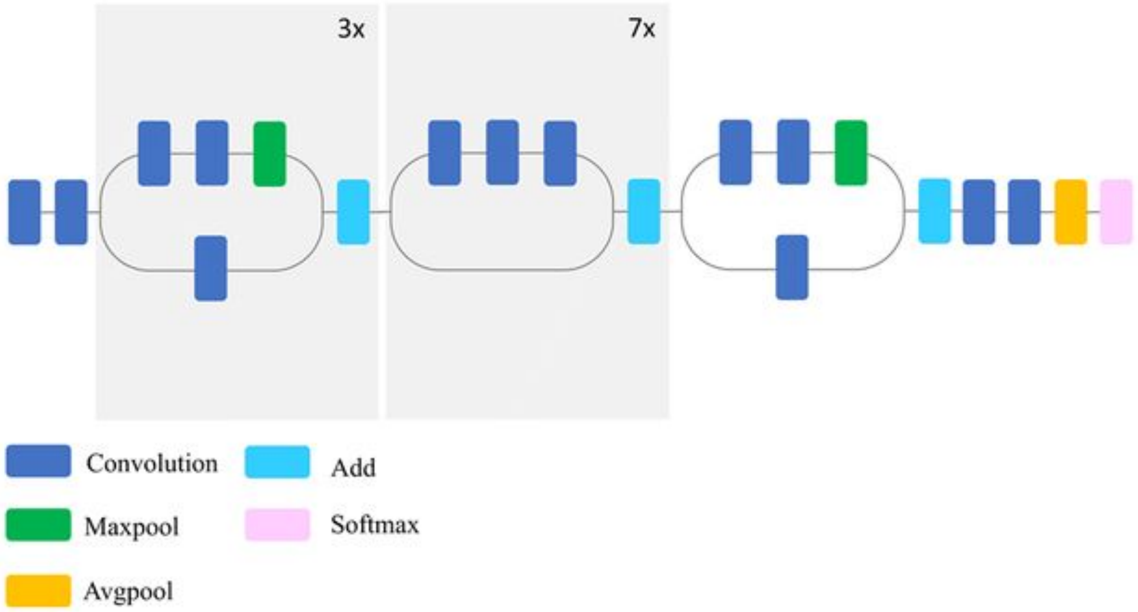
# Modern Convnet Architecture




------

# OBJECTIVES

- Understanding the important building blocks of modern CNNs

  - Residual connections
  - Batch normalisation
  - Depthwise separable convolution

- Building a Mini-Xception Model

- Interpret what CNNs learn

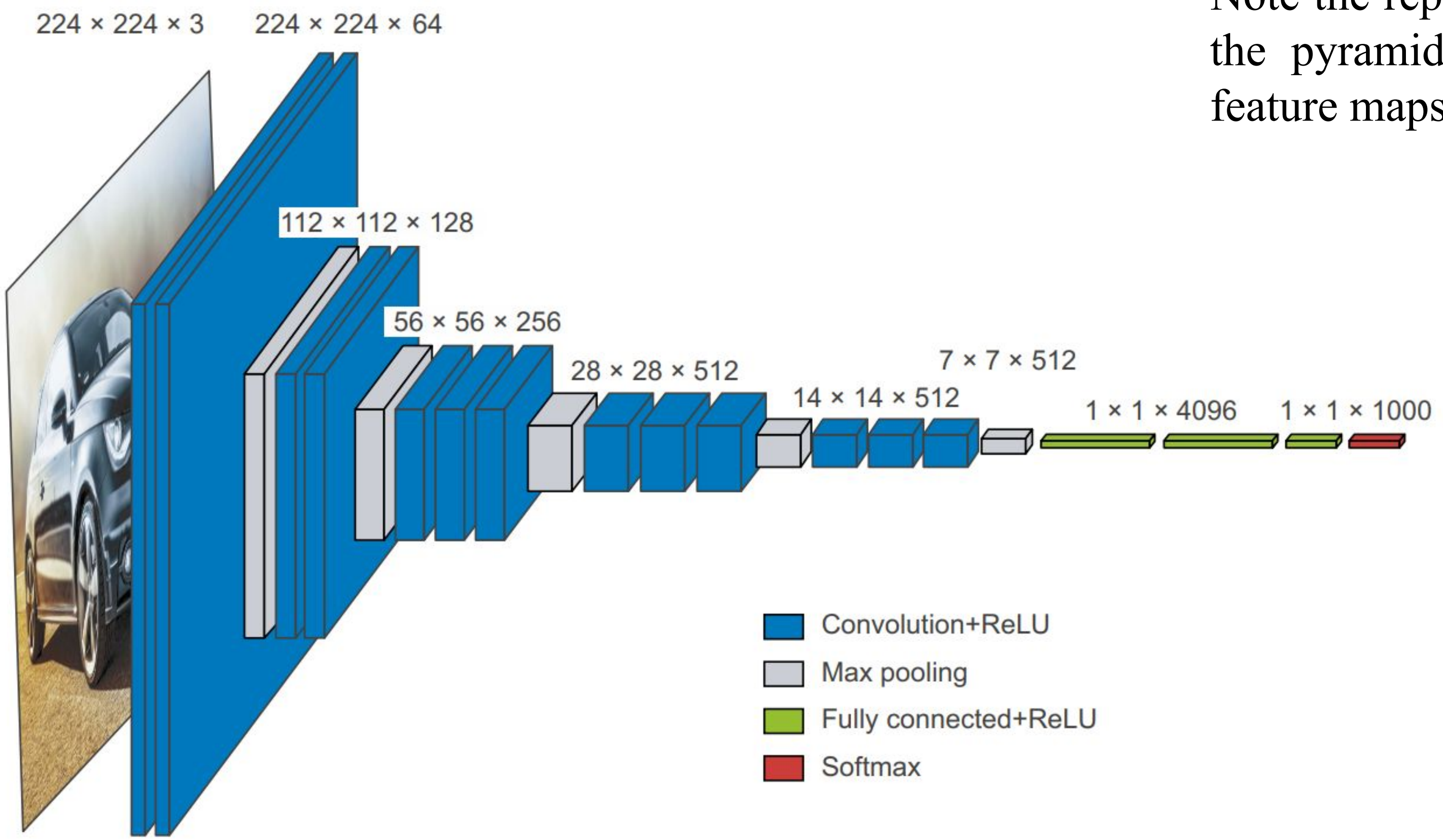
  1. Visualising activations
  2. Visualising filters
  3. Visualising heatmaps

For
1) Use : Small Convnet from last AST: Cat/Dog classification with Augmentation
2) & 3) Use: Pretrained Xception model

# PART : A

## Understanding the important building blocks of modern CNNs

# VGG16 Architecture

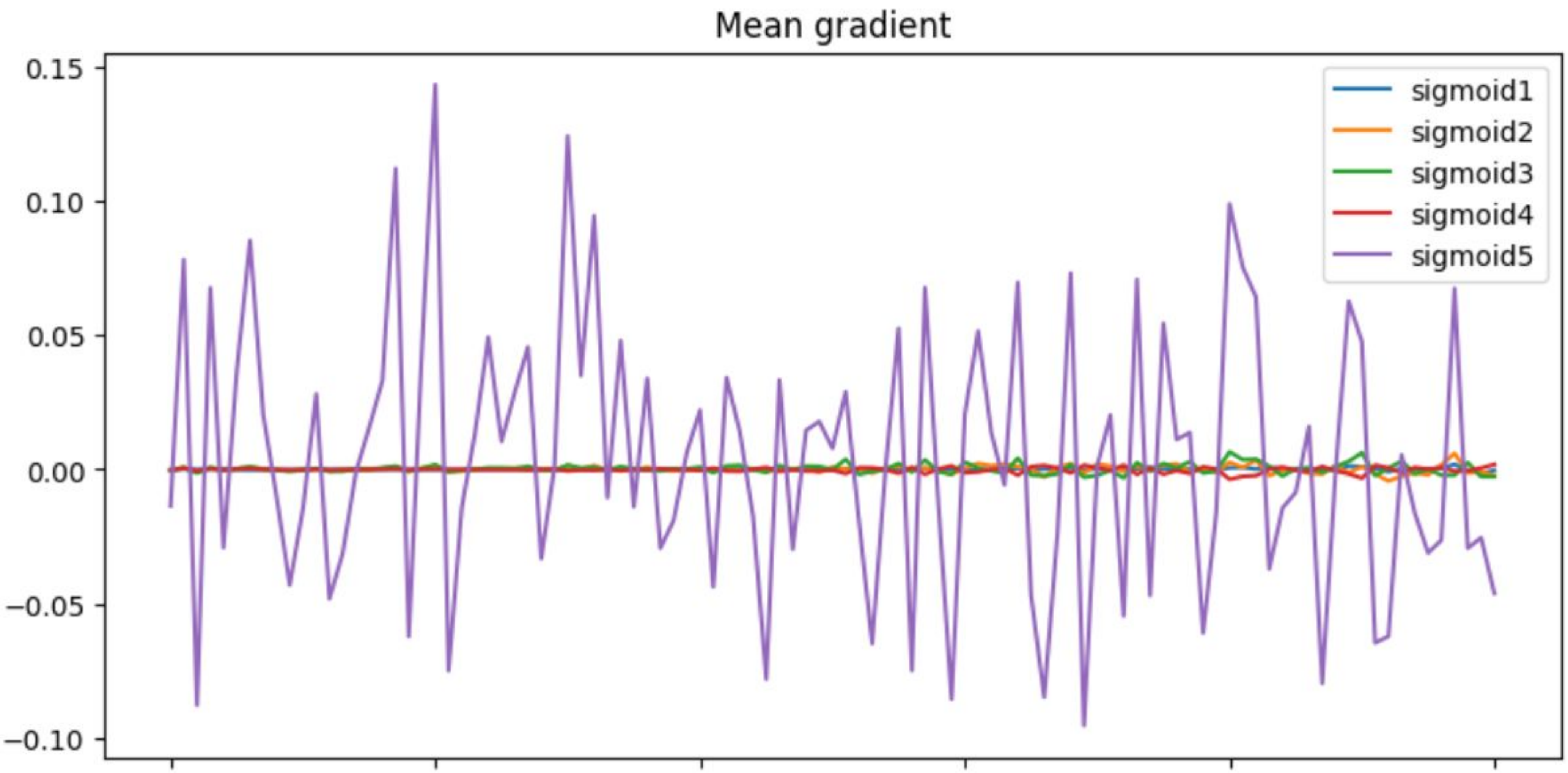



- Note the repeated layer blocks and the pyramid-like structure of the feature maps.

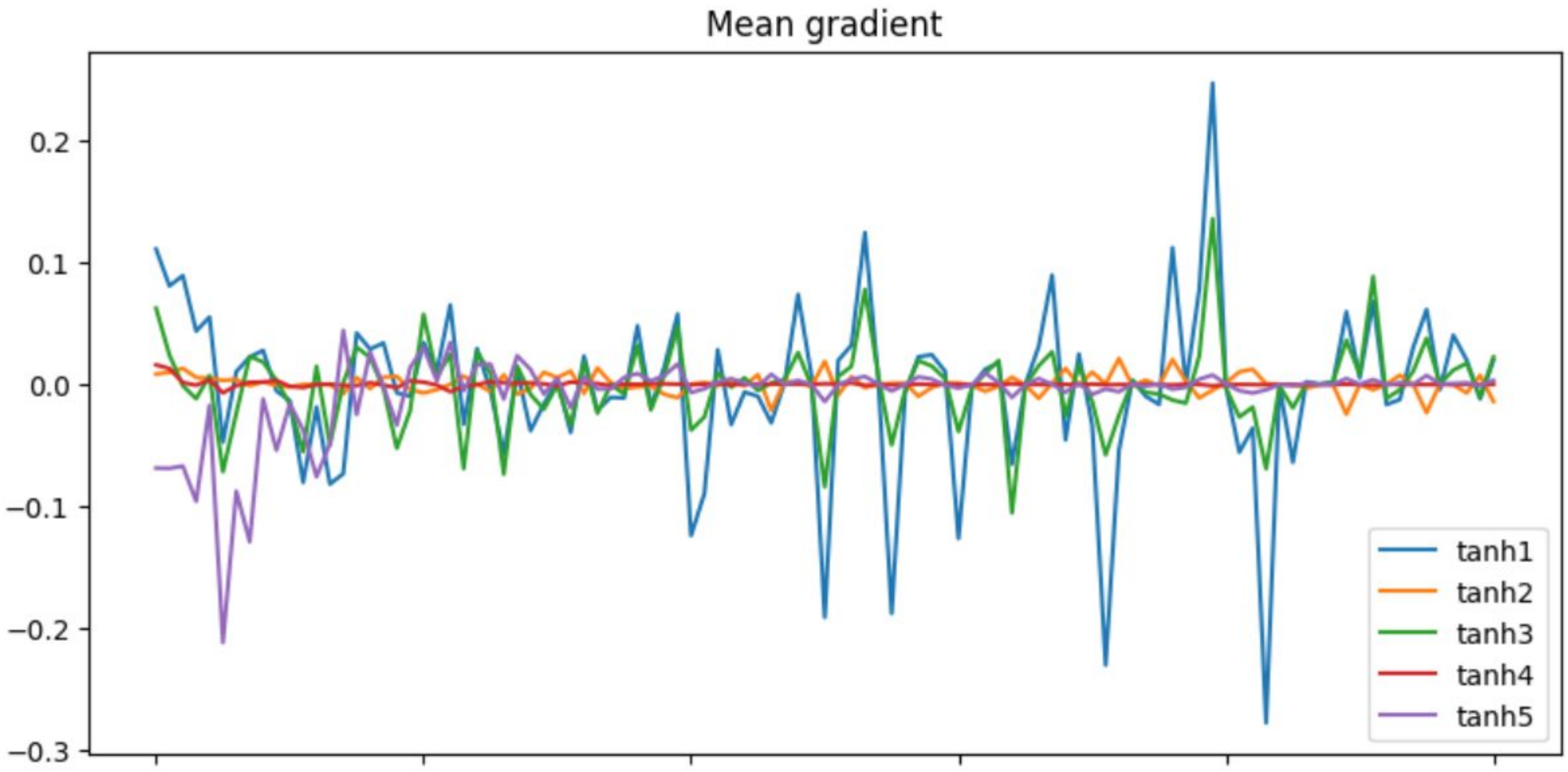
- Deeper hierarchies are intrinsically good because they encourage feature reuse, and therefore abstraction. In general, a deep stack of narrow layers performs better than a shallow stack of large layers. However, there's a limit to how deep you can stack layers, due to the problem of vanishing gradients.

□ **Residual connection**

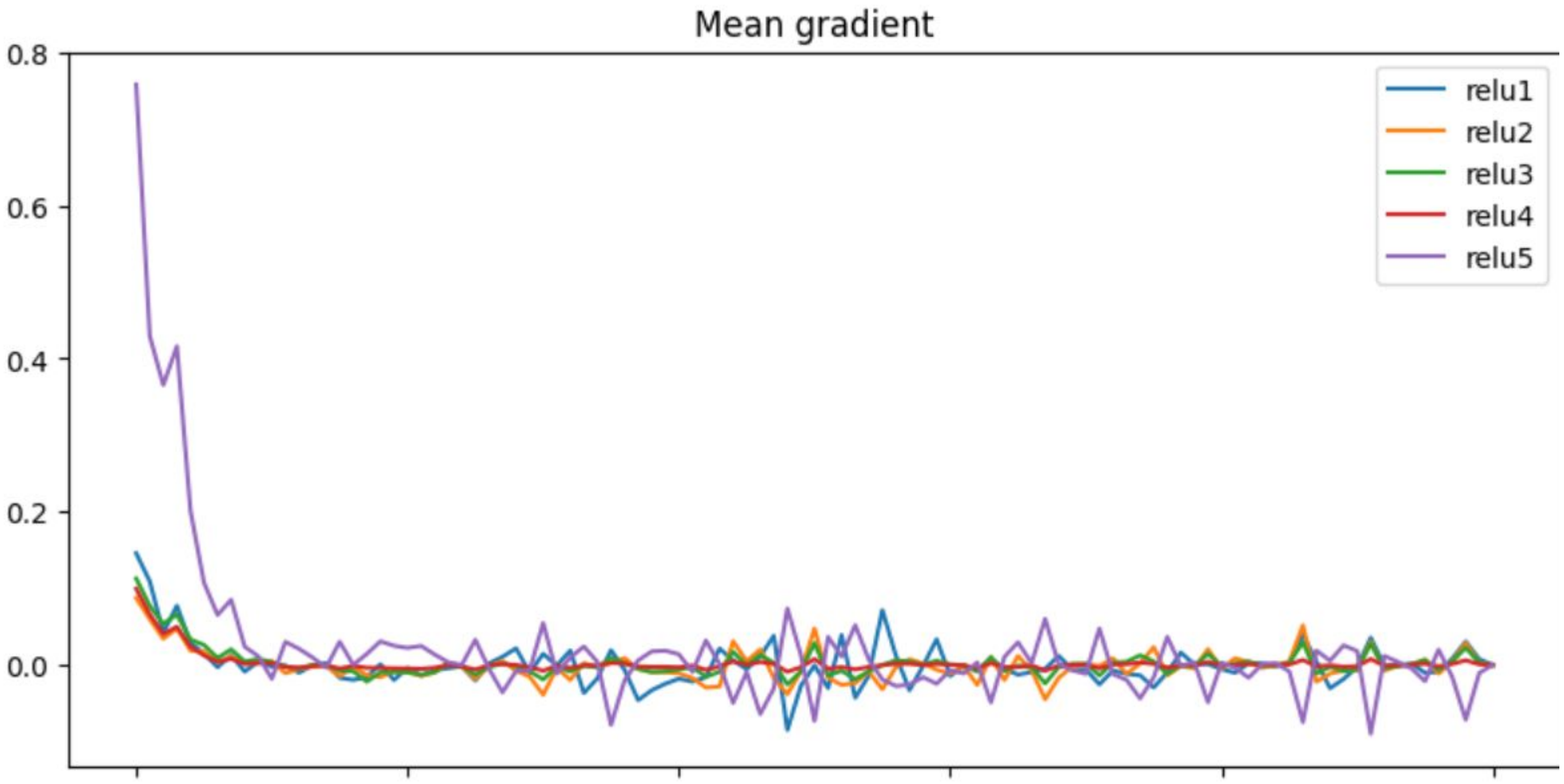
# Vanishing gradients - explained



# Vanishing gradients



# Vanishing gradients



# Vanishing gradients

Input gradients are calculated from output gradients using the formula

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot f'(x)$$

where:

$$\left(\frac{\partial L}{\partial x}\right)$$

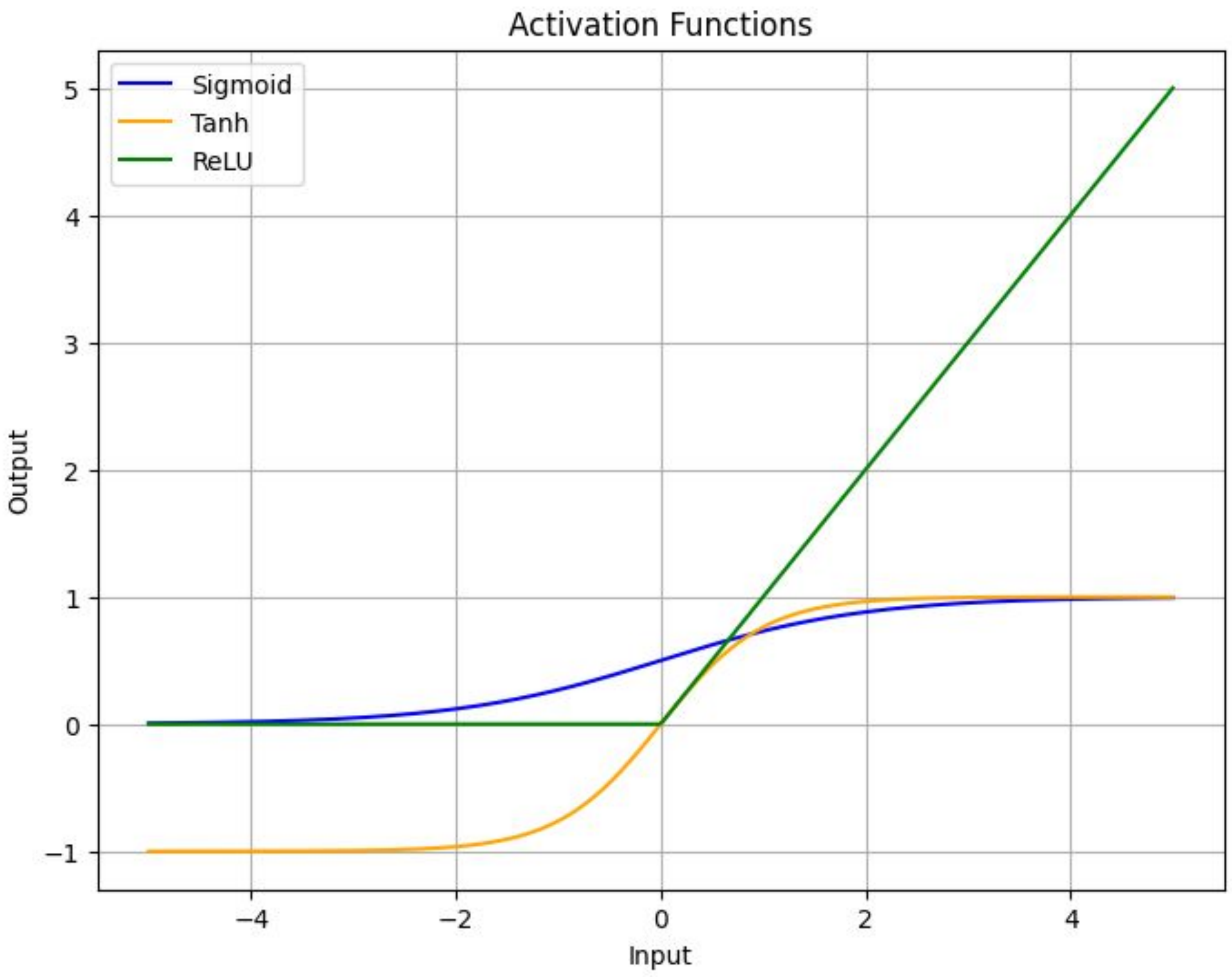
is the gradient of the loss with respect to the input ( x ),

$$\left(\frac{\partial L}{\partial y}\right)$$

is the gradient of the loss with respect to the output ( y ),

$$f'(x)$$

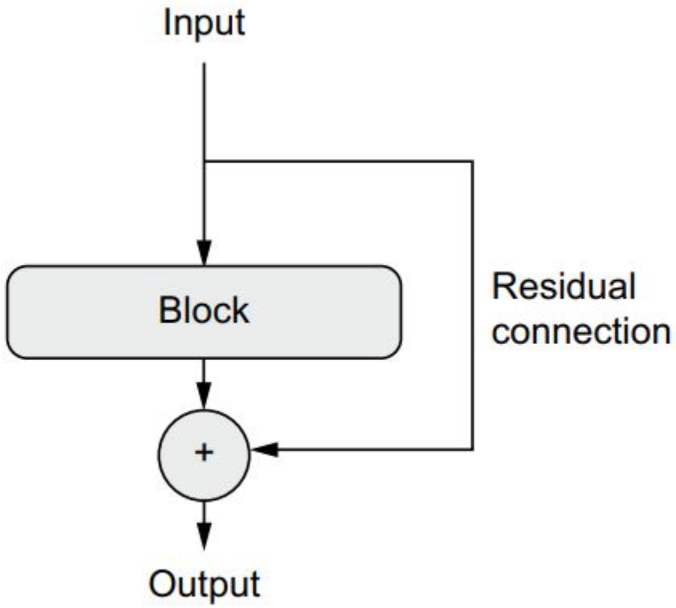
is the derivative of the activation function with respect to x.



# Vanishing gradients - solutions

- Weight initialization
- Residual connections
- Batch normalization

A residual connection




A residual connection in pseudocode

Some input tensor

Save a pointer to the original input. This is called the residual.

```
x = ...
residual = x
x = block(x)
x = add([x, residual])
```

This computation block can potentially be destructive or noisy, and that's fine.

Add the original input to the layer's output: the final output will thus always preserve full information about the original input.

# Case 1: Residual block where the number of filters changes

```
# 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)
residual = x
x = layers.Conv2D(64, 3, activation="relu", padding="same", name="C211")(x)
x = layers.Conv2D(64, 3, activation="relu", padding="same", name="C212")(x)
residual = layers.Conv2D(64, 1, name="C2b")(residual)
x = layers.add([x, residual]    )
model2 = keras.Model(inputs=inputs, outputs=x)
```

| Layer (type) | Output Shape | Param # |
|---|---|---|
| input (InputLayer) | [(None, 32, 32, 3)] | 0 |
| C1 (Conv2D) | (None, 30, 30, 32) | 896 |
| C211 (Conv2D) | (None, 30, 30, 64) | 18496 |
| C212 (Conv2D) | (None, 30, 30, 64) | 36928 |
| C2b (Conv2D) | (None, 30, 30, 64) | 2112 |
| add_2 (Add) | (None, 30, 30, 64) | 0 |

# Case 2: Where the target block includes a max pooling layer

```
# 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)
residual = layers.Conv2D(64, 1, strides=2)(residual)
x = layers.add([x, residual])
model3 = keras.Model(inputs=inputs, outputs=x)
plot_model(model3, show_shapes=True)
```

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 32, 32, 3)]       0

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

 conv2d_6 (Conv2D)           (None, 30, 30, 64)        18496

 max_pooling2d_1 (MaxPooling2D)  (None, 15, 15, 64)    0

 conv2d_7 (Conv2D)           (None, 15, 15, 64)        2112

 add_4 (Add)                 (None, 15, 15, 64)        0
```

# Batch Normalization

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: $\gamma$, $\beta$

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

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

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

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

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

The BatchNormalization layer can be used after any layer—Dense, Conv2D, etc.:

```
x = ...
x = layers.Conv2D(32, 3, use_bias=False)(x)
x = layers.BatchNormalization()(x)
```

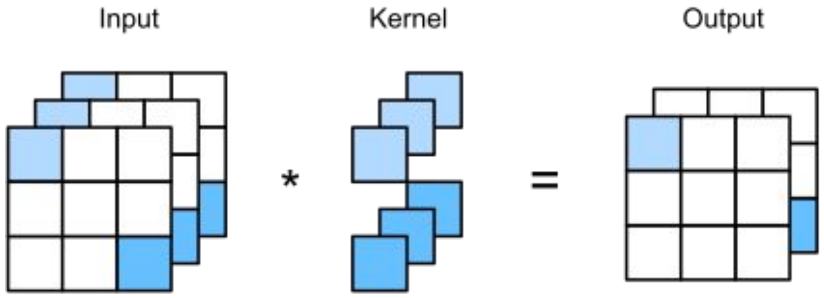
**Because the output of the Conv2D layer gets normalized, the layer doesn't need its own bias vector.**

**Parameters** : [gamma weights, beta weights, moving_mean (non-trainable), moving_variance(non-trainable)] = 32 x 4 = 128

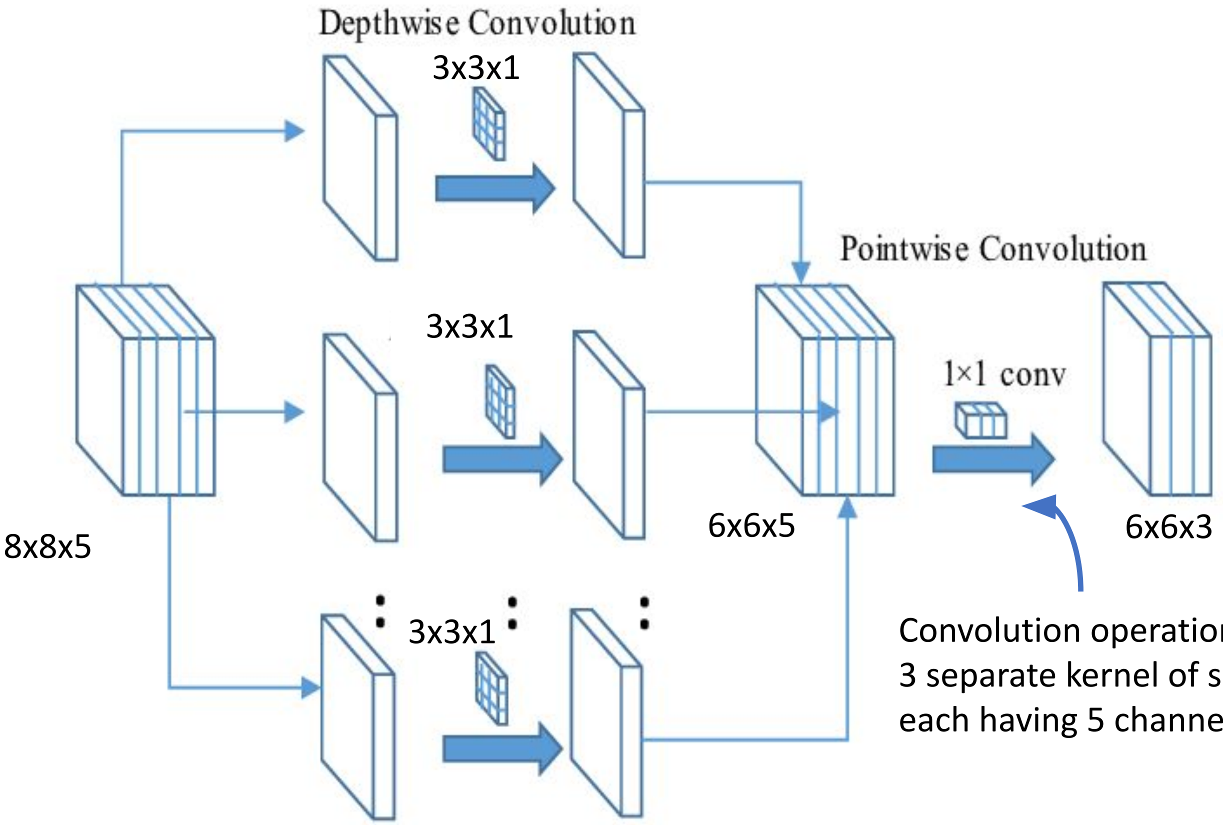
# Depthwise separable convolution

## Depthwise separable convolutions = Depthwise Conv. + Pointwise Conv.

### Revisiting Convolution Operation




$$Output\ shape\ after\ Conv = \frac{n+2p-f}{s}+1$$




Convolution operation:
3 separate kernel of size 1x1,
each having 5 channel

5 separate kernel, each having only one channel

## Depthwise separable convolution

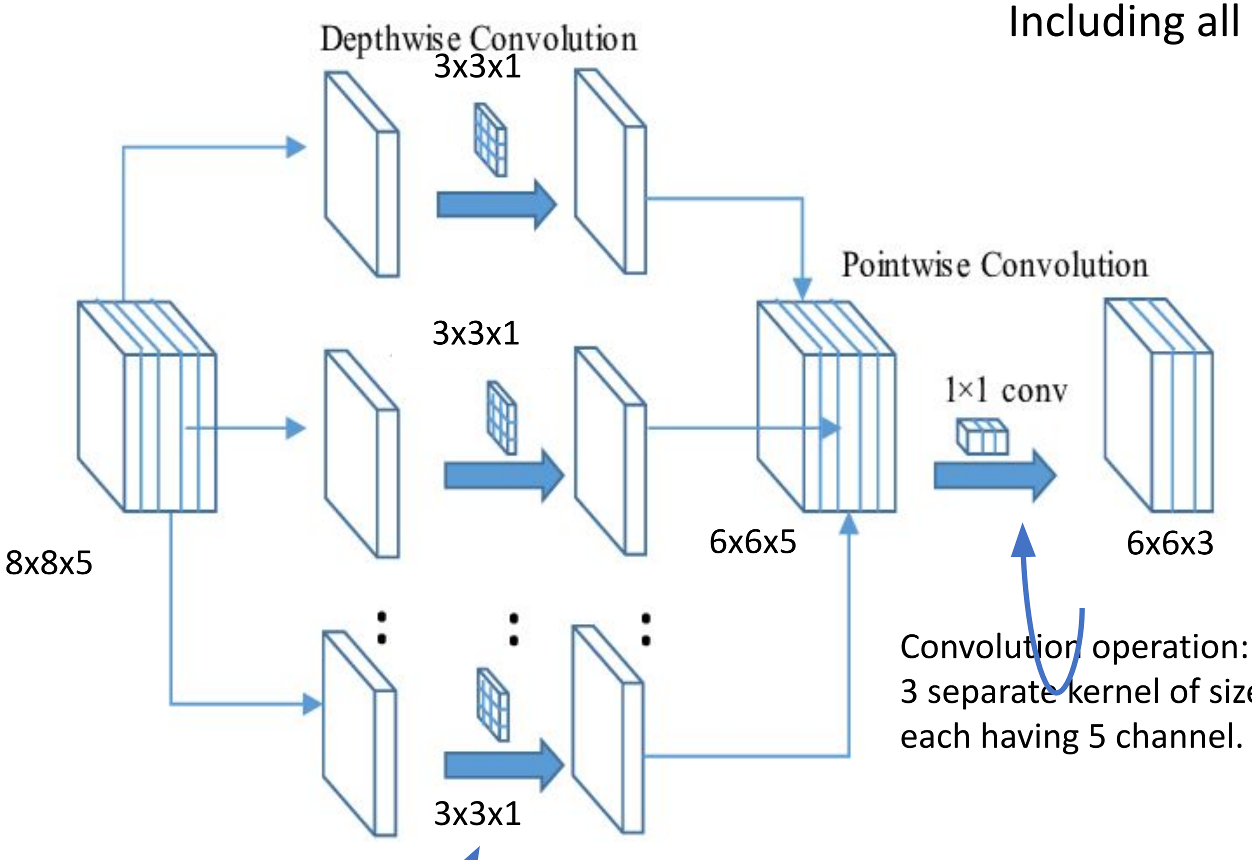
Bias

Parameter Calculation : $\underbrace{5 \times (3 \times 3) + 5}_{\text{Depthwise C.}} + \underbrace{(5 \times 1 \times 1 \times 3) + 3}_{\text{Pointwise C.}} = 68$

Including all bias

$\boxed{\text{Depthwise C. = N. of in channel x (f x f) + N. of in channel}}$

$\boxed{\text{Pointwise C. = N. of in channel x N. of filter + N. of filter}}$




Convolution operation:
3 separate kernel of size 1x1,
each having 5 channel.

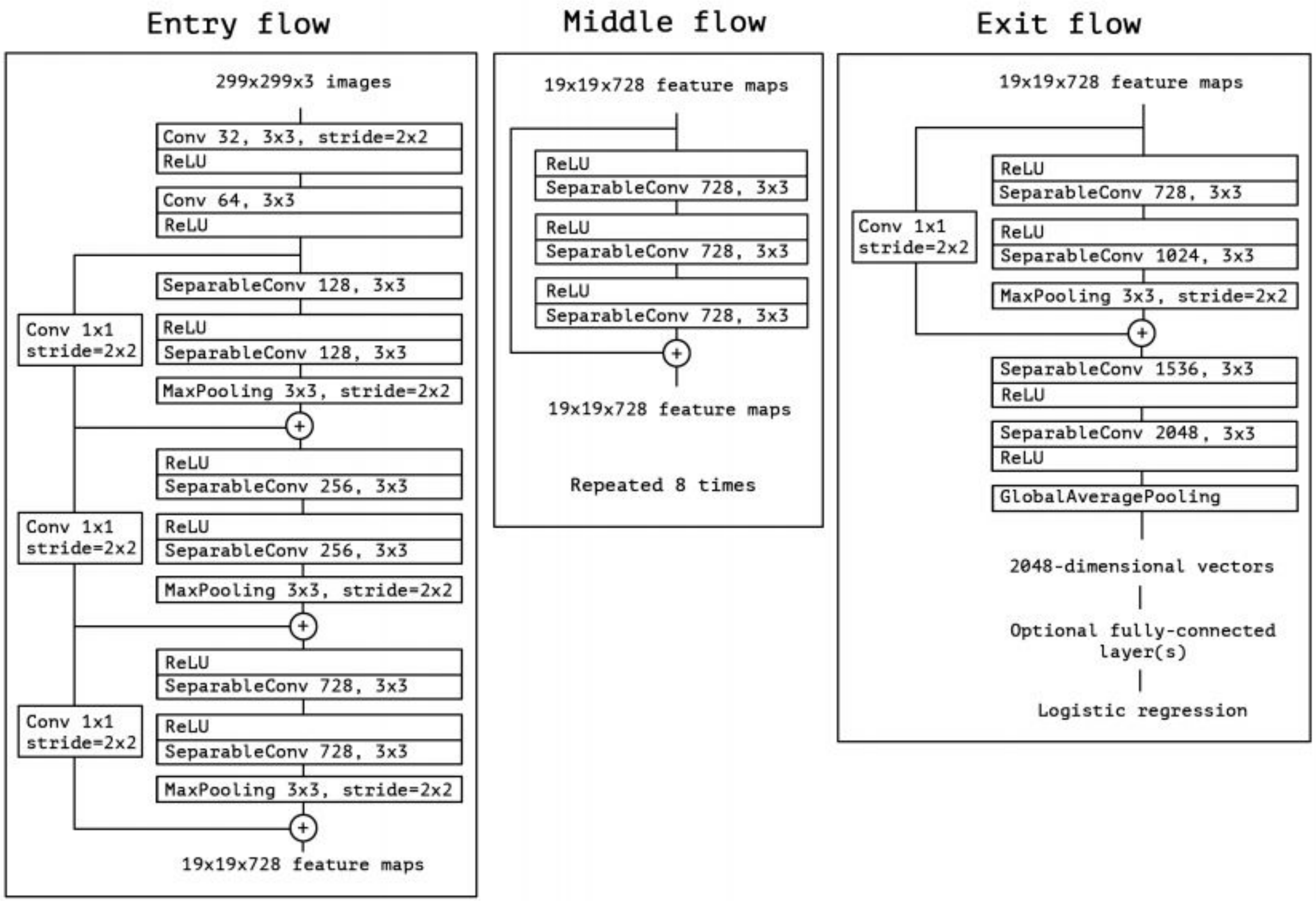
5 separate kernel, each having only one channel.

Keras

```
x = layers.SeparableConv2D(64, 3, activation="relu", padding="same")(x)
```

In keras, the bias for Depthwise C. is not implemented.

# Xception Model



# Creating a Mini-Xception Model (a smaller version)

```
inputs = keras.Input(shape=(180, 180, 3))
x = data_augmentation(inputs)

x = layers.Rescaling(1./255)(x)
x = layers.Conv2D(filters=32, kernel_size=5, use_bias=False)(x)

for size in [32, 64, 128, 256, 512]:
    residual = x

    x = layers.BatchNormalization()(x)
    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)
```

We use the same data augmentation configuration as before.

Don't forget input rescaling!

Note that the assumption that underlies separable convolution, "feature channels are largely independent," does not hold for RGB images! Red, green, and blue color channels are actually highly correlated in natural images. As such, the first layer in our model is a regular Conv2D layer. We'll start using SeparableConv2D afterwards.

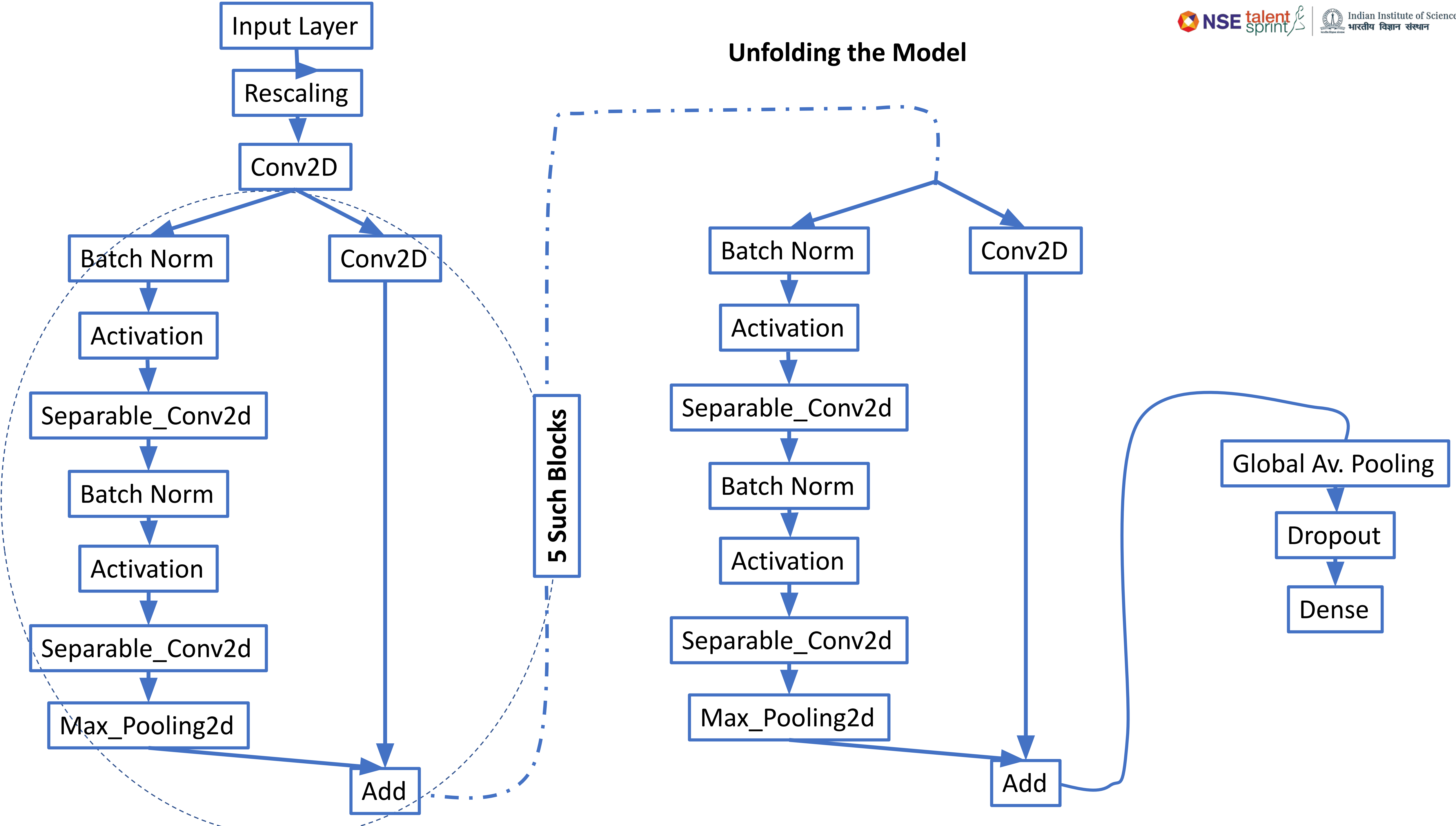
We apply a series of convolutional blocks with increasing feature depth. Each block consists of two batch-normalized depthwise separable convolution layers and a max pooling layer, with a residual connection around the entire block.

In the original model, we used a Flatten layer before the Dense layer. Here, we go with a GlobalAveragePooling2D layer.

Like in the original model, we add a dropout layer for regularization.

NSE talent sprint | Indian Institute of Science भारतीय विज्ञान संस्थान

## Unfolding the Model

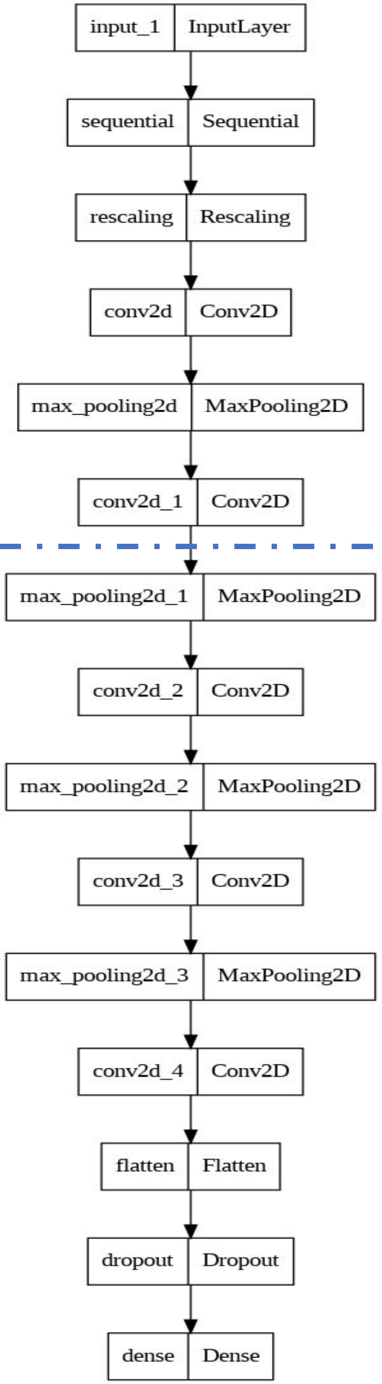


# PART : B

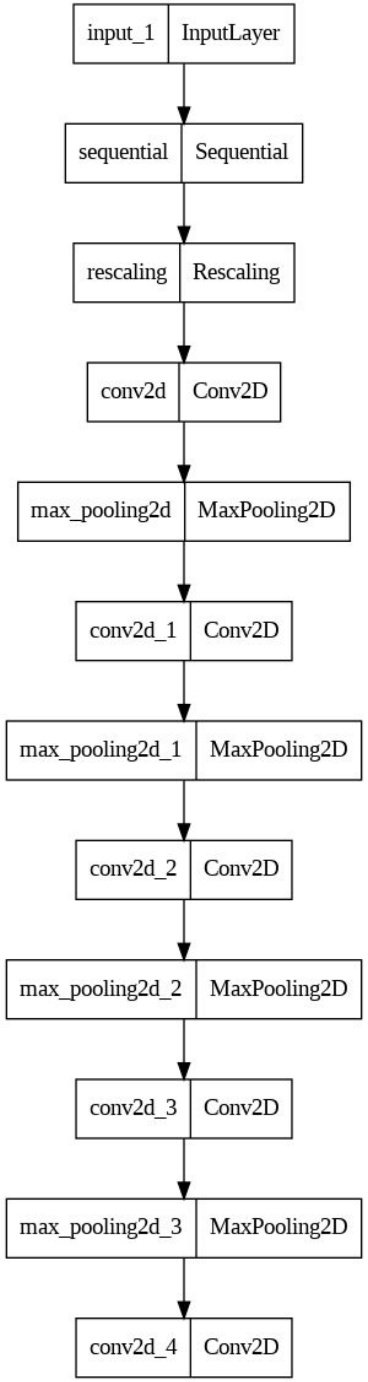
## Interpret what CNN's learn

# 1. Visualizing intermediate activations

```
path='/content/convnet_from_scratch_with_augmentation.keras'
model = keras.models.load_model(path)

plot_model(model)
```




```python
from tensorflow.keras import layers

layer_outputs = []
layer_names = []

for layer in model.layers:
    if isinstance(layer, (layers.Conv2D, layers.MaxPooling2D)):
        layer_outputs.append(layer.output)
        layer_names.append(layer.name)
activation_model = keras.Model(inputs=model.input, outputs=layer_outputs)

plot_model(activation_model)
```




Pre-processing the image and feeding into the model :

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

first_layer_feature_maps = activations[0]
print(f"first_layer_activation.shape= {first_layer_feature_maps.shape}")
```

```
1/1 [==============================] - 0s 420ms/step
No. of outputs= 9
first_layer_activation.shape= (1, 178, 178, 32)
```

```
layer_outputs
```

```
[<KerasTensor: shape=(None, 178, 178, 32) dtype=float32 (created by layer 'conv2d')>,
 <KerasTensor: shape=(None, 89, 89, 32) dtype=float32 (created by layer 'max_pooling2d')>,
 <KerasTensor: shape=(None, 87, 87, 64) dtype=float32 (created by layer 'conv2d_1')>,
 <KerasTensor: shape=(None, 43, 43, 64) dtype=float32 (created by layer 'max_pooling2d_1')>,
 <KerasTensor: shape=(None, 41, 41, 128) dtype=float32 (created by layer 'conv2d_2')>,
 <KerasTensor: shape=(None, 20, 20, 128) dtype=float32 (created by layer 'max_pooling2d_2')>,
 <KerasTensor: shape=(None, 18, 18, 256) dtype=float32 (created by layer 'conv2d_3')>,
 <KerasTensor: shape=(None, 9, 9, 256) dtype=float32 (created by layer 'max_pooling2d_3')>,
 <KerasTensor: shape=(None, 7, 7, 256) dtype=float32 (created by layer 'conv2d_4')>]
```

```
layer_names
```

```
['conv2d',
 'max_pooling2d',
 'conv2d_1',
 'max_pooling2d_1',
 'conv2d_2',
 'max_pooling2d_2',
 'conv2d_3',
 'max_pooling2d_3',
 'conv2d_4']
```

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

first_layer_feature_maps = activations[0]
print(f"first_layer_activation.shape= {first_layer_feature_maps.shape}")
```

```
1/1 [==============================] - 0s 420ms/step
No. of outputs= 9
first_layer_activation.shape= (1, 178, 178, 32)
```

```
print(type(activations))
print(len(activations ))
activations[0].shape
```

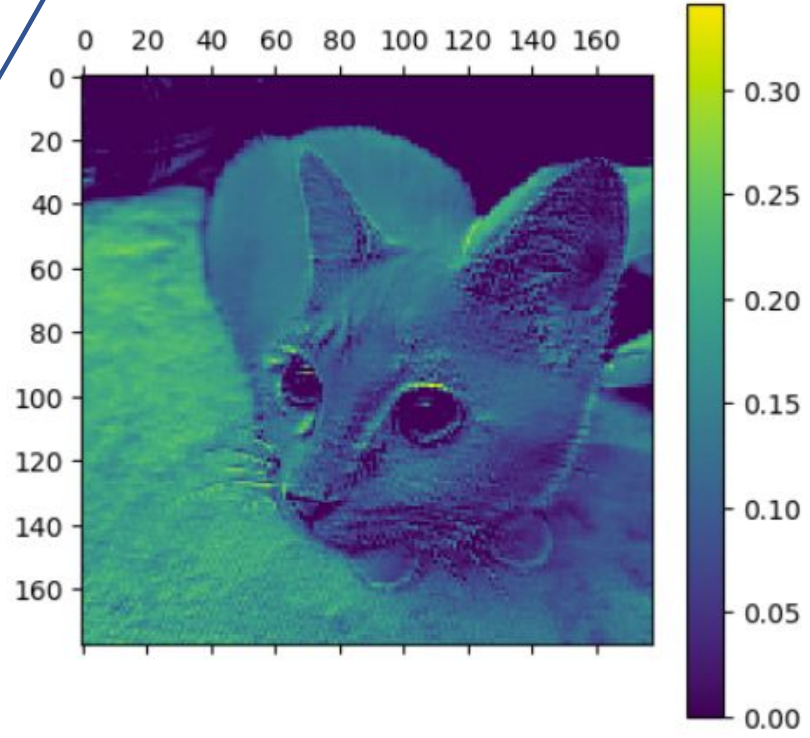
```
<class 'list'>
9
(1, 178, 178, 32)
```

```
activations[0][0,:,:,0].shape
```

```
(178, 178)
```

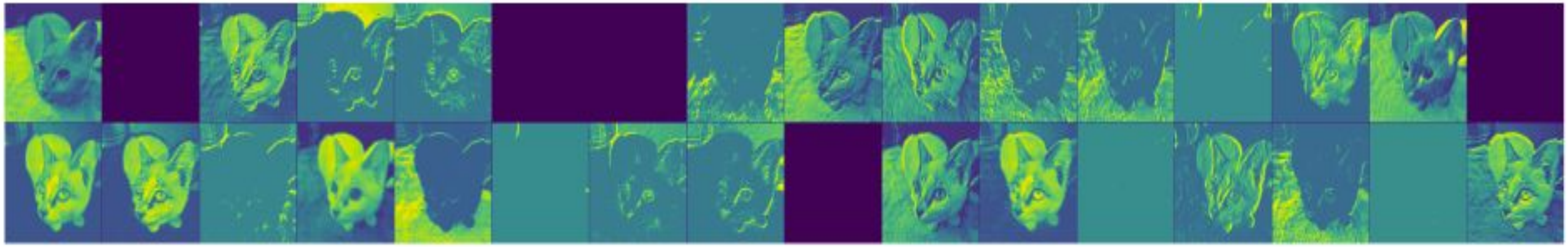
```
import matplotlib.pyplot as plt
plt.matshow(first_layer_feature_maps[0, :, :, 0], cmap="viridis")
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar at 0x7f38bc463280>
```




Colormaps link

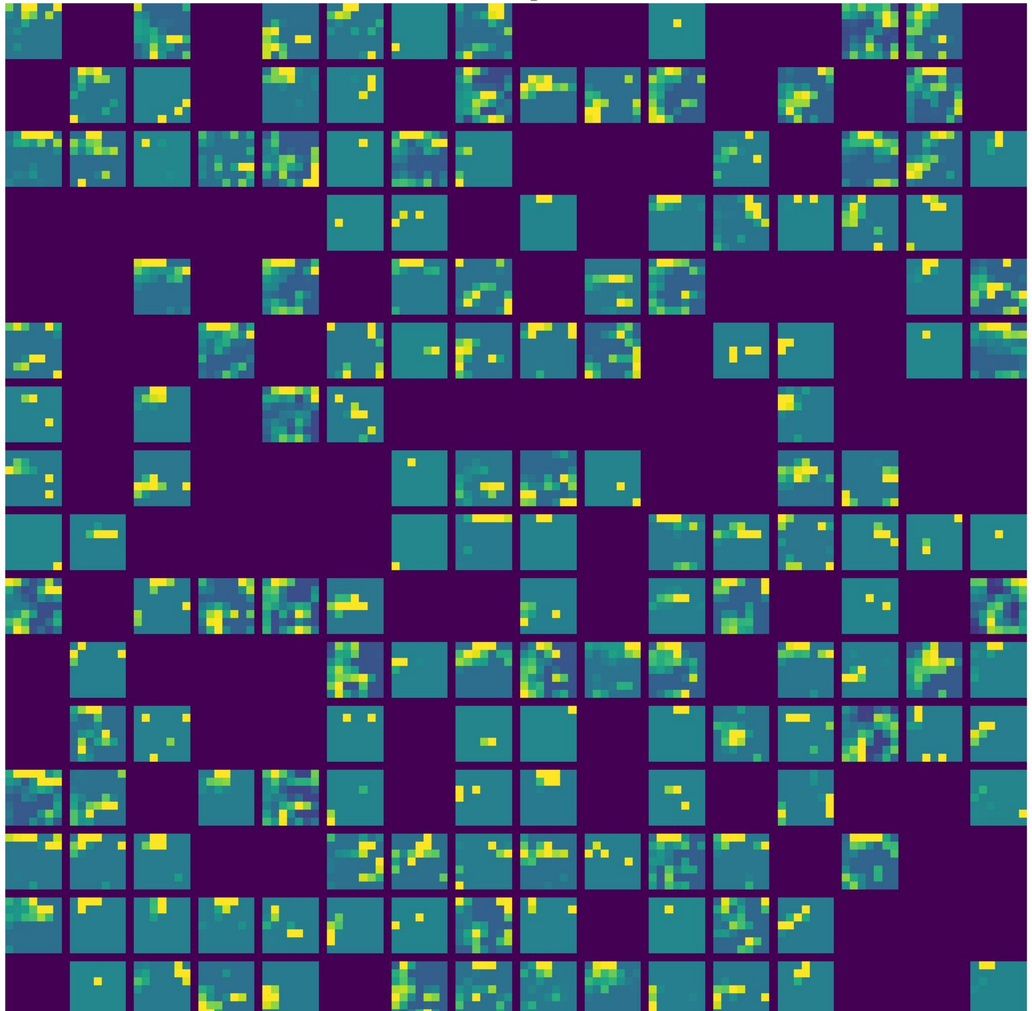
```
conv2d (Conv2D)              (None, 178, 178, 32)
```




```
layer_names
```

```
['conv2d',
 'max_pooling2d',
 'conv2d_1',
 'max_pooling2d_1',
 'conv2d_2',
 'max_pooling2d_2',
 'conv2d_3',
 'max_pooling2d_3',
 'conv2d_4']
```

16x16=256




```
conv2d_4 (Conv2D)           (None, 7, 7, 256)
```

## Findings :

- The first layer acts as a collection of various edge detectors.
- As we go deeper, the activations become increasingly abstract and less visually interpretable. They begin to encode higher-level concepts.
- 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

## 2. Visualising ConvNet filters

- Ask the question: What kind of an input image will excite the filter?
- What should the input image be so that we see a (yellow) feature map?

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

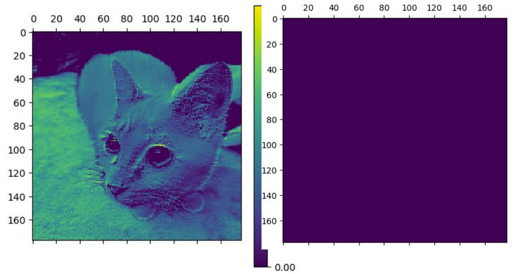
```
for layer in model.layers:
    if isinstance(layer, (keras.layers.Conv2D, keras.layers.SeparableConv2D)):
        print(layer.name)
```

```
block1_conv1
block1_conv2
block2_sepconv1
block2_sepconv2
conv2d_4
block3_sepconv1
block3_sepconv2
```

```
# Creating a feature extractor model
layer_name = "block3_sepconv1"
layer = model.get_layer(name=layer_name)
feature_extractor = keras.Model(inputs=model.input, outputs=layer.output)
feature_extractor.summary()
```

```
==================================================================================================
 input_2 (InputLayer)           [(None, None, None,  0           []
                                 3)]

 block1_conv1 (Conv2D)          (None, None, None,   864         ['input_2[0][0]']
                                32)

 block1_conv1_bn (BatchNormaliz  (None, None, None,   128         ['block1_conv1[0][0]']
 ation)                         32)

 block1_conv1_act (Activation)  (None, None, None,   0           ['block1_conv1_bn[0][0]']
                                32)

 block1_conv2 (Conv2D)          (None, None, None,   18432       ['block1_conv1_act[0][0]']
                                64)

 block1_conv2_bn (BatchNormaliz  (None, None, None,   256         ['block1_conv2[0][0]']
 ation)                         64)

 block1_conv2_act (Activation)  (None, None, None,   0           ['block1_conv2_bn[0][0]']
                                64)

 block2_sepconv1 (SeparableConv  (None, None, None,   8768        ['block1_conv2_act[0][0]']
 2D)                            128)

 block2_sepconv1_bn (BatchNorma  (None, None, None,   512         ['block2_sepconv1[0][0]']
 lization)                      128)

 block2_sepconv2_act (Activatio  (None, None, None,   0           ['block2_sepconv1_bn[0][0]']
 n)                             128)

 block2_sepconv2 (SeparableConv  (None, None, None,   17536       ['block2_sepconv2_act[0][0]']
 2D)                            128)

 block2_sepconv2_bn (BatchNorma  (None, None, None,   512         ['block2_sepconv2[0][0]']
 lization)                      128)

 conv2d_4 (Conv2D)              (None, None, None,   8192        ['block1_conv2_act[0][0]']
                                128)

 block2_pool (MaxPooling2D)     (None, None, None,   0           ['block2_sepconv2_bn[0][0]']
                                128)

 batch_normalization_4 (BatchNo  (None, None, None,   512         ['conv2d_4[0][0]']
 rmalization)                   128)

 add_12 (Add)                   (None, None, None,   0           ['block2_pool[0][0]',
                                128)                              'batch_normalization_4[0][0]']

 block3_sepconv1_act (Activatio  (None, None, None,   0           ['add_12[0][0]']
 n)                             128)

 block3_sepconv1 (SeparableConv  (None, None, None,   33920       ['block3_sepconv1_act[0][0]']
 2D)                            256)

==================================================================================================
```

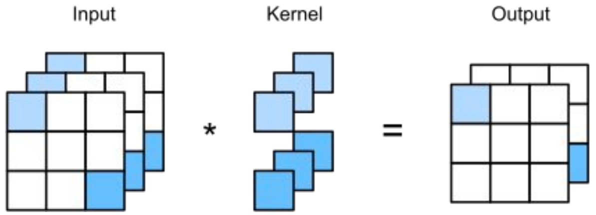
# Problem formulation:



We want that image as an input to the filter which excites the filter most. What does it mean?

Different input images are passed through the filter and the mean of all the pixel values is observed in output from the filter.

Initialize with a random image and calculate the mean of the activation values from the filter (loss function).
Calculate gradient of loss wr.t to image . Apply gradient ascent. Get the updated image. Iterate…




```
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)
```

Note that we avoid border artifacts by only involving non-border pixels in the loss;
we discard the first two pixels along the sides of the activation.

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

```
img_width = 200
img_height = 200

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

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

plt.axis("off")
plt.imshow(deprocess_image(generate_filter_pattern(filter_index=2)))
```

# Thanks !