Deep Learning

Lecture Topic:
Advanced Deep Learning for Computer Vision

Anxiao (Andrew) Jiang

Learning Objectives:

1. Understand image segmentation.

2. Understand patterns in CNN architectures

3. Understand how to visualize what CNN learns

Roadmap of this lecture:

1. Image segmentation.
2. CNN architecture patterns
    - 2.1 Residual connection
    - 2.2 Batch normalization and depthwise separable convolution
    - 2.3 Put them together for an Xception-like model
3. Visualize and interpret what CNN learns
    - 3.1 Visualize intermediate activations
    - 3.2 Visualize convolution filters
    - 3.3 Visualize heatmaps of class activation

- The different branches of computer vision: image classification, image segmentation, object detection
- Modern convnet architecture patterns: residual connections, batch normalization, depthwise separable convolutions
- Techniques for visualizing and interpreting what convnets learn

Three essential computer vision tasks

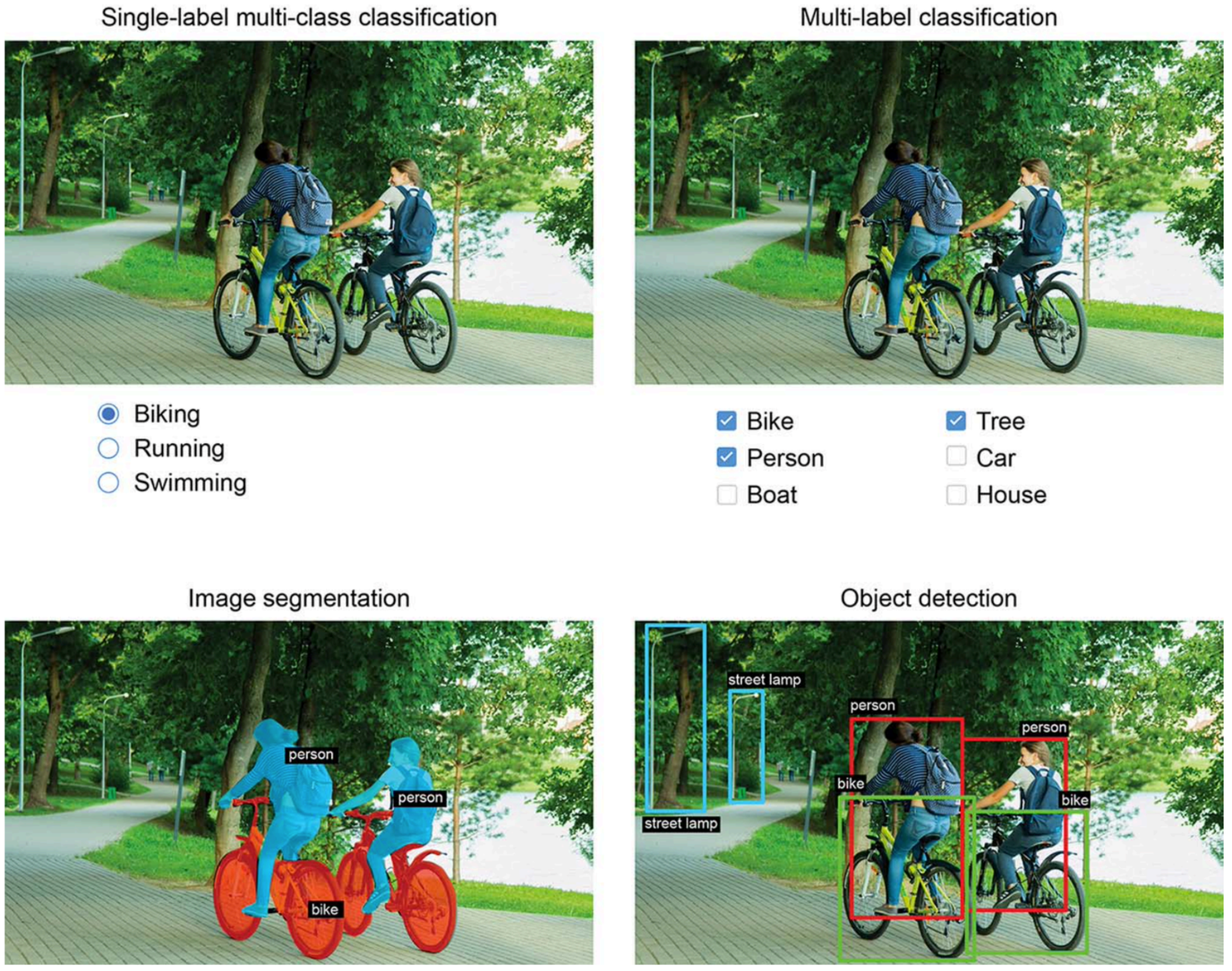


Figure 9.1 The three main computer vision tasks: classification, segmentation, detection

Image segmentation

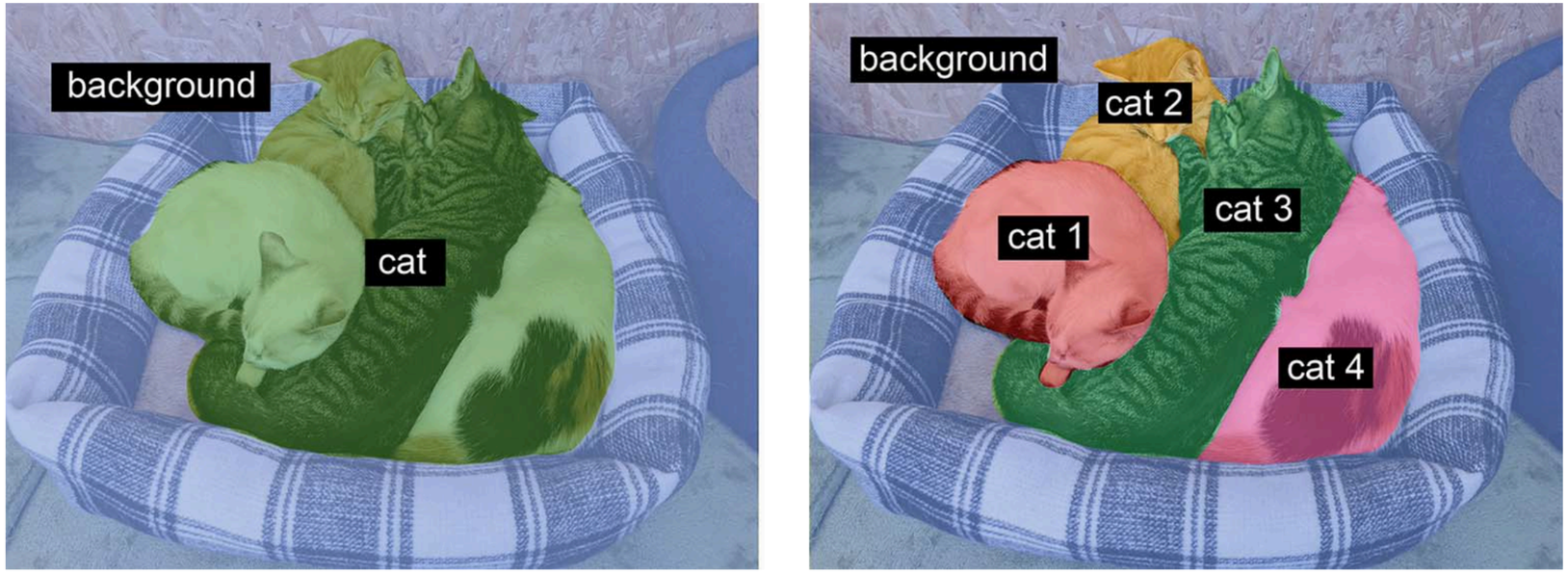


Figure 9.2 Semantic segmentation vs. instance segmentation

Image segmentation

We'll work with the Oxford-IIIT Pets dataset (www.robots.ox.ac.uk/~vgg/data/pets/), which contains 7,390 pictures of various breeds of cats and dogs, together with foreground-background segmentation masks for each picture. A *segmentation mask* is the image-segmentation equivalent of a label: it's an image the same size as the input image, with a single color channel where each integer value corresponds to the class of the corresponding pixel in the input image. In our case, the pixels of our segmentation masks can take one of three integer values:

- 1 (foreground)
- 2 (background)
- 3 (contour)

Image segmentation: Get the data

Let's start by downloading and uncompressing our dataset, using the `wget` and `tar` shell utilities:

```
!wget http://www.robots.ox.ac.uk/~vgg/data/pets/data/images.tar.gz
!wget http://www.robots.ox.ac.uk/~vgg/data/pets/data/annotations.tar.gz
!tar -xf images.tar.gz
!tar -xf annotations.tar.gz
```

The input pictures are stored as JPG files in the images/ folder (such as images/Abyssinian_1.jpg), and the corresponding segmentation mask is stored as a PNG file with the same name in the annotations/trimaps/ folder (such as annotations/trimaps/Abyssinian_1.png).

Image segmentation: Get the data

Let's prepare the list of input file paths, as well as the list of the corresponding mask file paths:

```
import os

input_dir = "images/"
target_dir = "annotations/trimaps/"

input_img_paths = sorted(
    [os.path.join(input_dir, fname)
     for fname in os.listdir(input_dir)
     if fname.endswith(".jpg")])


target_paths = sorted(
    [os.path.join(target_dir, fname)
     for fname in os.listdir(target_dir)
     if fname.endswith(".png") and not fname.startswith(".")])
```

Image segmentation: Get the data

Now, what does one of these inputs and its mask look like? Let's take a quick look. Here's a sample image (see figure 9.3):

```
import matplotlib.pyplot as plt
from tensorflow.keras.utils import load_img, img_to_array

plt.axis("off")
plt.imshow(load_img(input_img_paths[9]))
```

Display input image number 9.

Figure 9.3 An example image

Image segmentation: Get the data

And here is its corresponding target (see figure 9.4):

The original labels are 1, 2, and 3. We subtract 1 so that the labels range from 0 to 2, and then we multiply by 127 so that the labels become 0 (black), 127 (gray), 254 (near-white).

```
def display_target(target_array):
    normalized_array = (target_array.astype("uint8") - 1) * 127
    plt.axis("off")
    plt.imshow(normalized_array[:, :, 0])

img = img_to_array(load_img(target_paths[9], color_mode="grayscale"))
display_target(img)
```

We use color_mode="grayscale" so that the image we load is treated as having a single color channel.

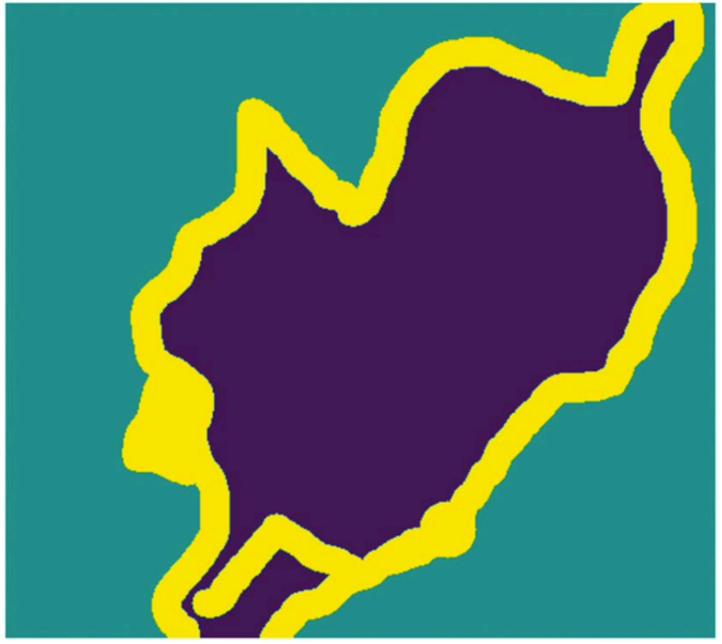


Figure 9.4 The corresponding target mask

Image segmentation: Get the data

Next, let's load our inputs and targets into two NumPy arrays, and let's split the arrays into a training and a validation set. Since the dataset is very small, we can just load everything into memory:

```
import numpy as np
import random

img_size = (200, 200)
num_imgs = len(input_img_paths)

random.Random(1337).shuffle(input_img_paths)
random.Random(1337).shuffle(target_paths)

def path_to_input_image(path):
    return img_to_array(load_img(path, target_size=img_size))

def path_to_target(path):
    img = img_to_array(
        load_img(path, target_size=img_size, color_mode="grayscale"))
    img = img.astype("uint8") - 1
    return img

input_imgs = np.zeros((num_imgs,) + img_size + (3,), dtype="float32")
targets = np.zeros((num_imgs,) + img_size + (1,), dtype="uint8")
for i in range(num_imgs):
    input_imgs[i] = path_to_input_image(input_img_paths[i])
    targets[i] = path_to_target(target_paths[i])

num_val_samples = 1000
train_input_imgs = input_imgs[:-num_val_samples]
train_targets = targets[:-num_val_samples]
val_input_imgs = input_imgs[-num_val_samples:]
val_targets = targets[-num_val_samples:]
```

We resize everything to 200 × 200.

Total number of samples in the data

Shuffle the file paths (they were originally sorted by breed). We use the same seed (1337) in both statements to ensure that the input paths and target paths stay in the same order.

Subtract 1 so that our labels become 0, 1, and 2.

Load all images in the input_imgs float32 array and their masks in the targets uint8 array (same order). The inputs have three channels (RBG values) and the targets have a single channel (which contains integer labels).

Reserve 1,000 samples for validation.

Split the data into a training and a validation set.

Image segmentation: Build the model

```
from tensorflow import keras
from tensorflow.keras import layers

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))
    x = layers.Rescaling(1./255)(inputs)

    x = layers.Conv2D(64, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(256, 3, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(256, 3, activation="relu", padding="same")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        256, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        128, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        64, 3, activation="relu", padding="same", strides=2)(x)

    outputs = layers.Conv2D(num_classes, 3, activation="softmax",
     padding="same")(x)

    model = keras.Model(inputs, outputs)
    return model

model = get_model(img_size=img_size, num_classes=3)
model.summary()
```

Don't forget to rescale input images to the [0-1] range.

Note how we use padding="same" everywhere to avoid the influence of border padding on feature map size.

inputs

We end the model with a per-pixel three-way softmax to classify each output pixel into one of our three categories.

Image segmentation: Build the model

```
from tensorflow import keras
from tensorflow.keras import layers

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))
    x = layers.Rescaling(1./255)(inputs)

    x = layers.Conv2D(64, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(256, 3, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(256, 3, activation="relu", padding="same")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        256, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        128, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        64, 3, activation="relu", padding="same", strides=2)(x)

    outputs = layers.Conv2D(num_classes, 3, activation="softmax",
     padding="same")(x)

    model = keras.Model(inputs, outputs)
    return model

model = get_model(img_size=img_size, num_classes=3)
model.summary()
```

Don't forget to rescale input images to the [0-1] range.

Note how we use padding="same" everywhere to avoid the influence of border padding on feature map size.

We end the model with a per-pixel three-way softmax to classify each output pixel into one of our three categories.

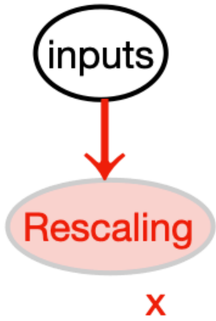


Image segmentation: Build the model

```
from tensorflow import keras
from tensorflow.keras import layers

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))
    x = layers.Rescaling(1./255)(inputs)

    x = layers.Conv2D(64, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(256, 3, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(256, 3, activation="relu", padding="same")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        256, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        128, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        64, 3, activation="relu", padding="same", strides=2)(x)

    outputs = layers.Conv2D(num_classes, 3, activation="softmax",
     padding="same")(x)

    model = keras.Model(inputs, outputs)
    return model

model = get_model(img_size=img_size, num_classes=3)
model.summary()
```

Don't forget to rescale input images to the [0-1] range.

Note how we use padding="same" everywhere to avoid the influence of border padding on feature map size.

We end the model with a per-pixel three-way softmax to classify each output pixel into one of our three categories.

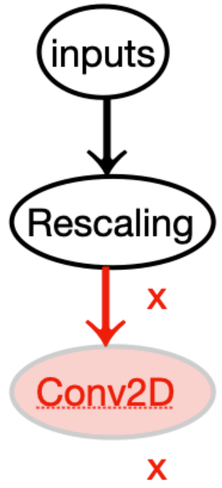


Image segmentation: Build the model

```
from tensorflow import keras
from tensorflow.keras import layers

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))
    x = layers.Rescaling(1./255)(inputs)

    x = layers.Conv2D(64, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(256, 3, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(256, 3, activation="relu", padding="same")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        256, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        128, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        64, 3, activation="relu", padding="same", strides=2)(x)

    outputs = layers.Conv2D(num_classes, 3, activation="softmax",
     padding="same")(x)

    model = keras.Model(inputs, outputs)
    return model

model = get_model(img_size=img_size, num_classes=3)
model.summary()
```

Don't forget to rescale input images to the [0-1] range.

Note how we use padding="same" everywhere to avoid the influence of border padding on feature map size.

We end the model with a per-pixel three-way softmax to classify each output pixel into one of our three categories.

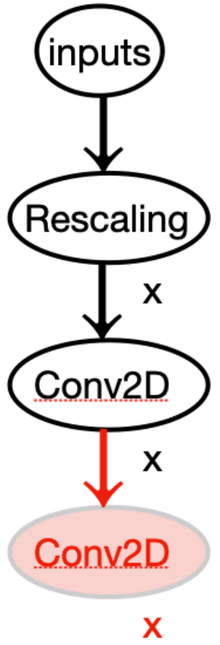


Image segmentation: Build the model

```
from tensorflow import keras
from tensorflow.keras import layers

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))
    x = layers.Rescaling(1./255)(inputs)

    x = layers.Conv2D(64, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(256, 3, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(256, 3, activation="relu", padding="same")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        256, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        128, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        64, 3, activation="relu", padding="same", strides=2)(x)

    outputs = layers.Conv2D(num_classes, 3, activation="softmax",
     padding="same")(x)

    model = keras.Model(inputs, outputs)
    return model

model = get_model(img_size=img_size, num_classes=3)
model.summary()
```

Don't forget to rescale input images to the [0-1] range.

Note how we use padding="same" everywhere to avoid the influence of border padding on feature map size.

We end the model with a per-pixel three-way softmax to classify each output pixel into one of our three categories.

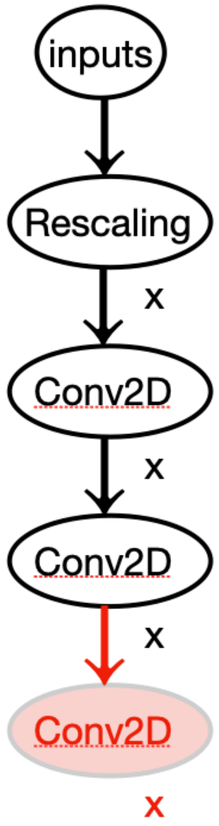


Image segmentation: Build the model

```
from tensorflow import keras
from tensorflow.keras import layers

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))
    x = layers.Rescaling(1./255)(inputs)

    x = layers.Conv2D(64, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(256, 3, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(256, 3, activation="relu", padding="same")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        256, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        128, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        64, 3, activation="relu", padding="same", strides=2)(x)

    outputs = layers.Conv2D(num_classes, 3, activation="softmax",
     padding="same")(x)

    model = keras.Model(inputs, outputs)
    return model

model = get_model(img_size=img_size, num_classes=3)
model.summary()
```

Don't forget to rescale input images to the [0-1] range.

Note how we use padding="same" everywhere to avoid the influence of border padding on feature map size.

We end the model with a per-pixel three-way softmax to classify each output pixel into one of our three categories.

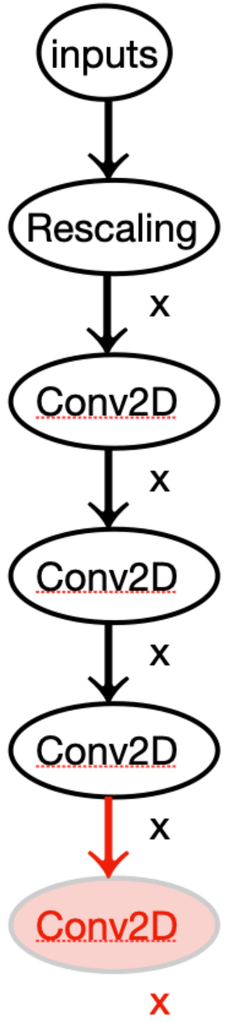


Image segmentation: Build the model

```
from tensorflow import keras
from tensorflow.keras import layers

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))
    x = layers.Rescaling(1./255)(inputs)

    x = layers.Conv2D(64, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(256, 3, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(256, 3, activation="relu", padding="same")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        256, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        128, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        64, 3, activation="relu", padding="same", strides=2)(x)

    outputs = layers.Conv2D(num_classes, 3, activation="softmax",
     padding="same")(x)

    model = keras.Model(inputs, outputs)
    return model

model = get_model(img_size=img_size, num_classes=3)
model.summary()
```

Don't forget to rescale input images to the [0-1] range.

Note how we use padding="same" everywhere to avoid the influence of border padding on feature map size.

We end the model with a per-pixel three-way softmax to classify each output pixel into one of our three categories.

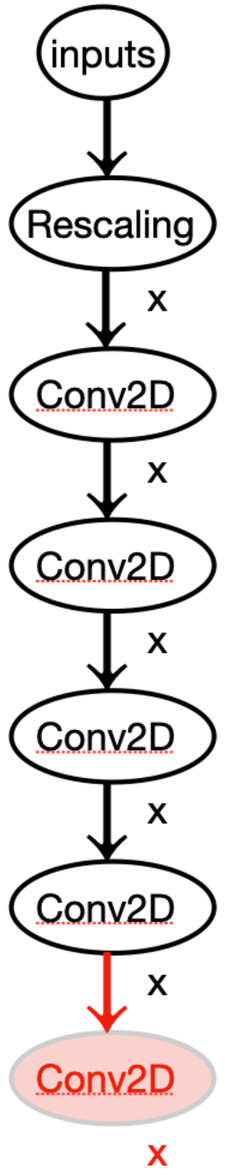


Image segmentation: Build the model

```
from tensorflow import keras
from tensorflow.keras import layers

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))
    x = layers.Rescaling(1./255)(inputs)

    x = layers.Conv2D(64, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(256, 3, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(256, 3, activation="relu", padding="same")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        256, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        128, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        64, 3, activation="relu", padding="same", strides=2)(x)

    outputs = layers.Conv2D(num_classes, 3, activation="softmax",
     padding="same")(x)

    model = keras.Model(inputs, outputs)
    return model

model = get_model(img_size=img_size, num_classes=3)
model.summary()
```

Don't forget to rescale input images to the [0-1] range.

Note how we use padding="same" everywhere to avoid the influence of border padding on feature map size.

We end the model with a per-pixel three-way softmax to classify each output pixel into one of our three categories.

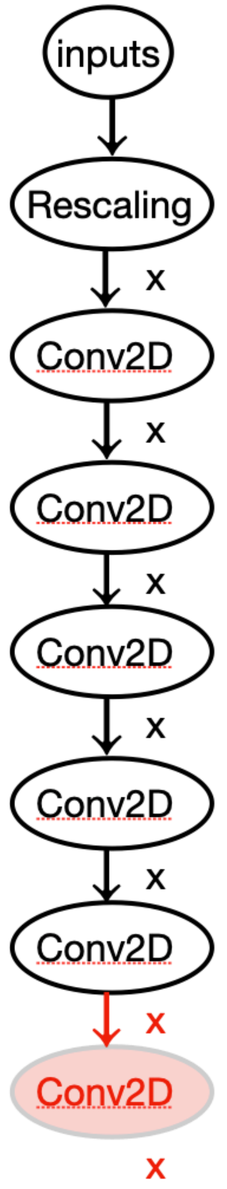


Image segmentation: Build the model

```python
from tensorflow import keras
from tensorflow.keras import layers

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))
    x = layers.Rescaling(1./255)(inputs)

    x = layers.Conv2D(64, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(256, 3, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(256, 3, activation="relu", padding="same")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        256, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        128, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        64, 3, activation="relu", padding="same", strides=2)(x)

    outputs = layers.Conv2D(num_classes, 3, activation="softmax",
     padding="same")(x)

    model = keras.Model(inputs, outputs)
    return model

model = get_model(img_size=img_size, num_classes=3)
model.summary()
```

Don't forget to rescale input images to the [0-1] range.

Note how we use padding="same" everywhere to avoid the influence of border padding on feature map size.

We end the model with a per-pixel three-way softmax to classify each output pixel into one of our three categories.

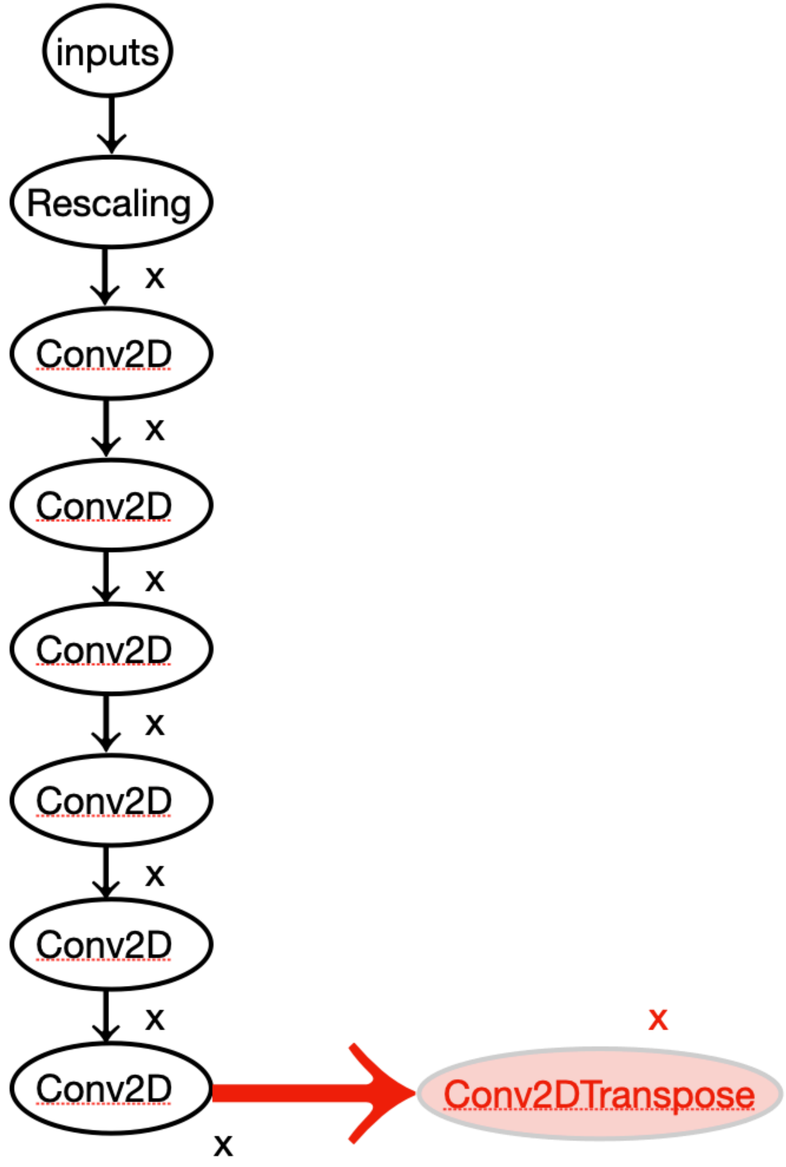


Image segmentation: Build the model

```python
from tensorflow import keras
from tensorflow.keras import layers

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))
    x = layers.Rescaling(1./255)(inputs)

    x = layers.Conv2D(64, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(256, 3, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(256, 3, activation="relu", padding="same")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        256, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        128, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        64, 3, activation="relu", padding="same", strides=2)(x)

    outputs = layers.Conv2D(num_classes, 3, activation="softmax",
     padding="same")(x)

    model = keras.Model(inputs, outputs)
    return model

model = get_model(img_size=img_size, num_classes=3)
model.summary()
```

Don't forget to rescale input images to the [0-1] range.

Note how we use padding="same" everywhere to avoid the influence of border padding on feature map size.

We end the model with a per-pixel three-way softmax to classify each output pixel into one of our three categories.

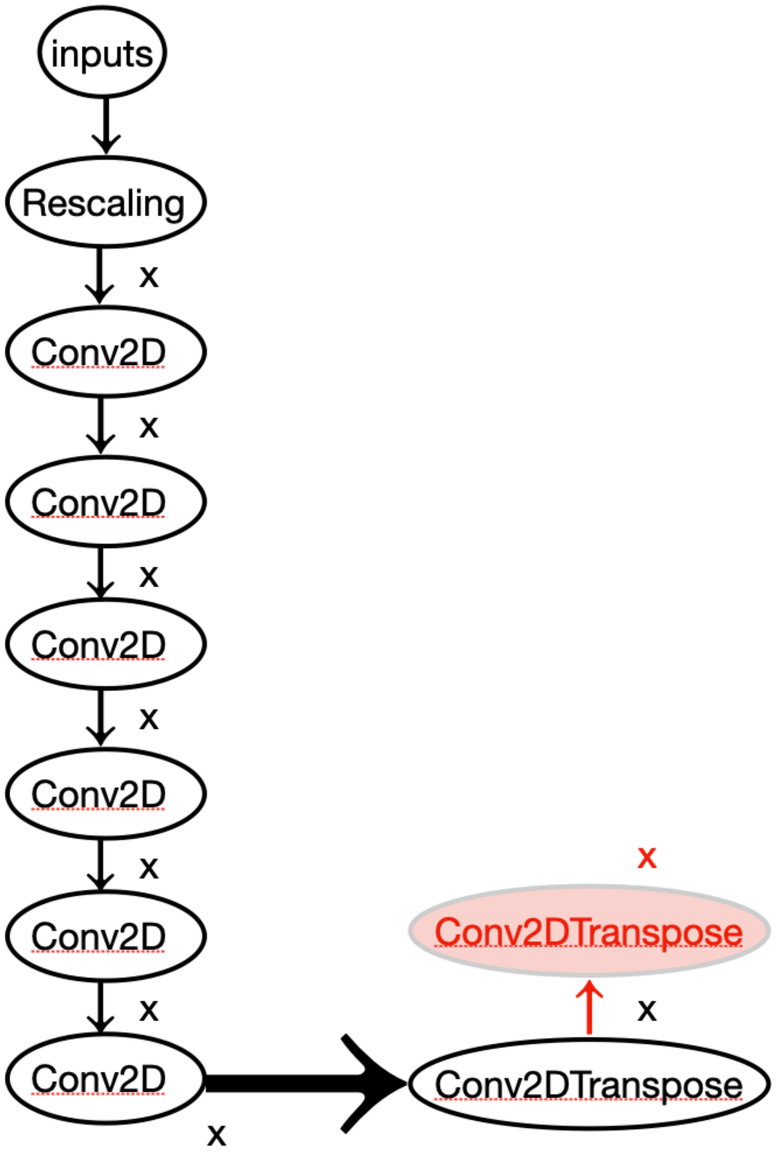


Image segmentation: Build the model

```python
from tensorflow import keras
from tensorflow.keras import layers

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))
    x = layers.Rescaling(1./255)(inputs)

    x = layers.Conv2D(64, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(256, 3, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(256, 3, activation="relu", padding="same")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        256, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        128, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        64, 3, activation="relu", padding="same", strides=2)(x)

    outputs = layers.Conv2D(num_classes, 3, activation="softmax",
     padding="same")(x)

    model = keras.Model(inputs, outputs)
    return model

model = get_model(img_size=img_size, num_classes=3)
model.summary()
```

Don't forget to rescale input images to the [0-1] range.

Note how we use padding="same" everywhere to avoid the influence of border padding on feature map size.

We end the model with a per-pixel three-way softmax to classify each output pixel into one of our three categories.

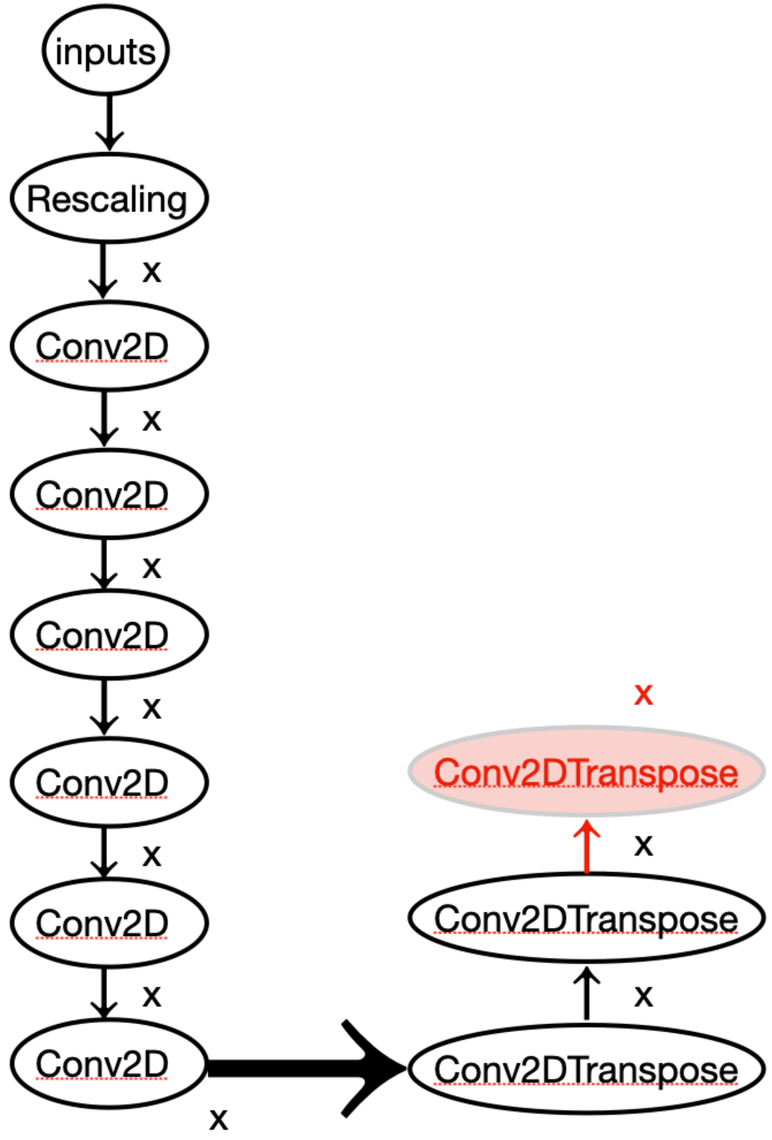


Image segmentation: Build the model

```
from tensorflow import keras
from tensorflow.keras import layers

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))
    x = layers.Rescaling(1./255)(inputs)

    x = layers.Conv2D(64, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(256, 3, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(256, 3, activation="relu", padding="same")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        256, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        128, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        64, 3, activation="relu", padding="same", strides=2)(x)

    outputs = layers.Conv2D(num_classes, 3, activation="softmax",
     padding="same")(x)

    model = keras.Model(inputs, outputs)
    return model

model = get_model(img_size=img_size, num_classes=3)
model.summary()
```

Don't forget to rescale input images to the [0-1] range.

Note how we use padding="same" everywhere to avoid the influence of border padding on feature map size.

We end the model with a per-pixel three-way softmax to classify each output pixel into one of our three categories.

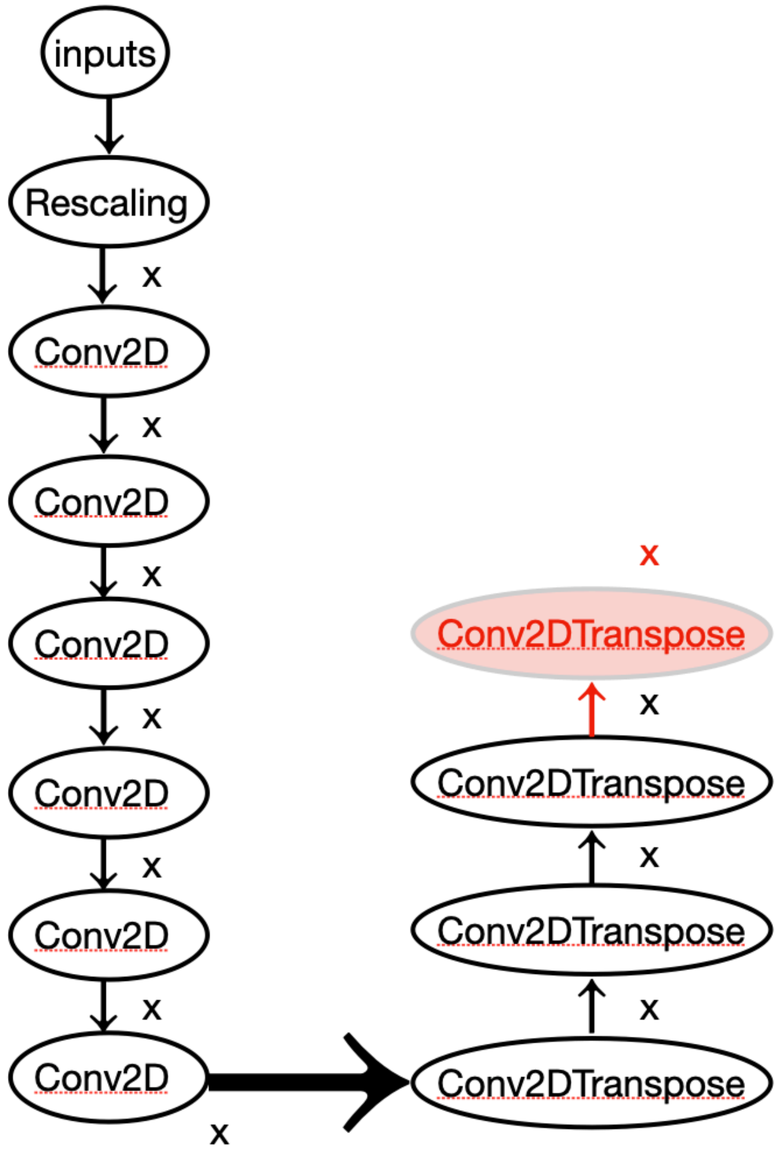


Image segmentation: Build the model

```python
from tensorflow import keras
from tensorflow.keras import layers

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))
    x = layers.Rescaling(1./255)(inputs)

    x = layers.Conv2D(64, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(256, 3, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(256, 3, activation="relu", padding="same")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        256, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        128, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        64, 3, activation="relu", padding="same", strides=2)(x)

    outputs = layers.Conv2D(num_classes, 3, activation="softmax",
     padding="same")(x)

    model = keras.Model(inputs, outputs)
    return model

model = get_model(img_size=img_size, num_classes=3)
model.summary()
```

Don't forget to rescale input images to the [0-1] range.

Note how we use padding="same" everywhere to avoid the influence of border padding on feature map size.

We end the model with a per-pixel three-way softmax to classify each output pixel into one of our three categories.

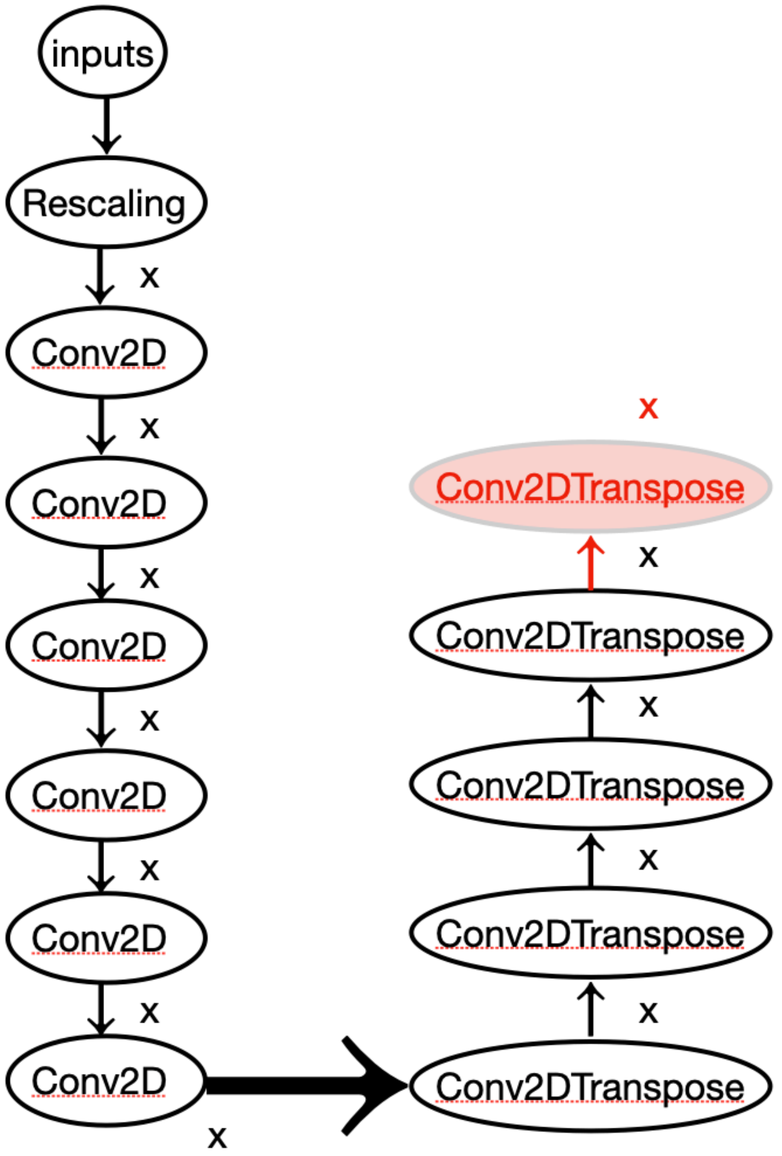


Image segmentation: Build the model

```python
from tensorflow import keras
from tensorflow.keras import layers

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))
    x = layers.Rescaling(1./255)(inputs)

    x = layers.Conv2D(64, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(256, 3, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(256, 3, activation="relu", padding="same")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        256, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        128, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        64, 3, activation="relu", padding="same", strides=2)(x)

    outputs = layers.Conv2D(num_classes, 3, activation="softmax",
     padding="same")(x)

    model = keras.Model(inputs, outputs)
    return model

model = get_model(img_size=img_size, num_classes=3)
model.summary()
```

Don't forget to rescale input images to the [0-1] range.

Note how we use padding="same" everywhere to avoid the influence of border padding on feature map size.

We end the model with a per-pixel three-way softmax to classify each output pixel into one of our three categories.

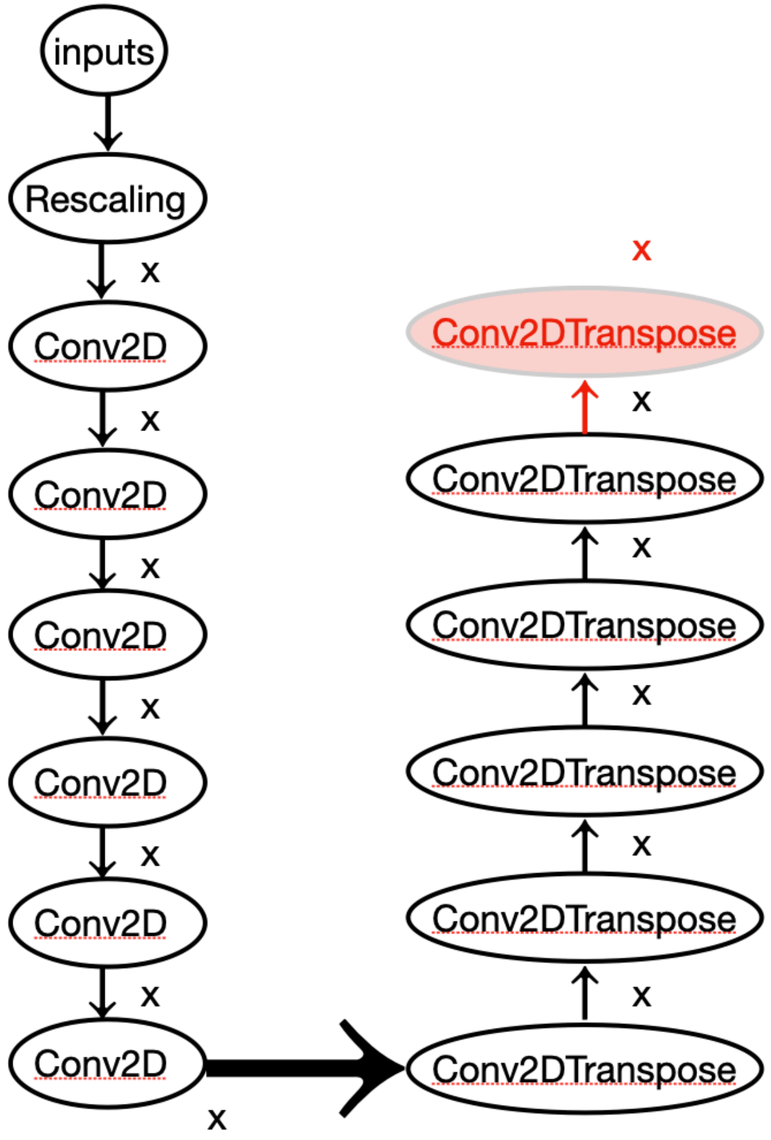


Image segmentation: Build the model

```python
from tensorflow import keras
from tensorflow.keras import layers

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))
    x = layers.Rescaling(1./255)(inputs)

    x = layers.Conv2D(64, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(256, 3, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(256, 3, activation="relu", padding="same")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        256, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        128, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        64, 3, activation="relu", padding="same", strides=2)(x)

    outputs = layers.Conv2D(num_classes, 3, activation="softmax",
     padding="same")(x)

    model = keras.Model(inputs, outputs)
    return model

model = get_model(img_size=img_size, num_classes=3)
model.summary()
```

Don't forget to rescale input images to the [0-1] range.

Note how we use padding="same" everywhere to avoid the influence of border padding on feature map size.

We end the model with a per-pixel three-way softmax to classify each output pixel into one of our three categories.

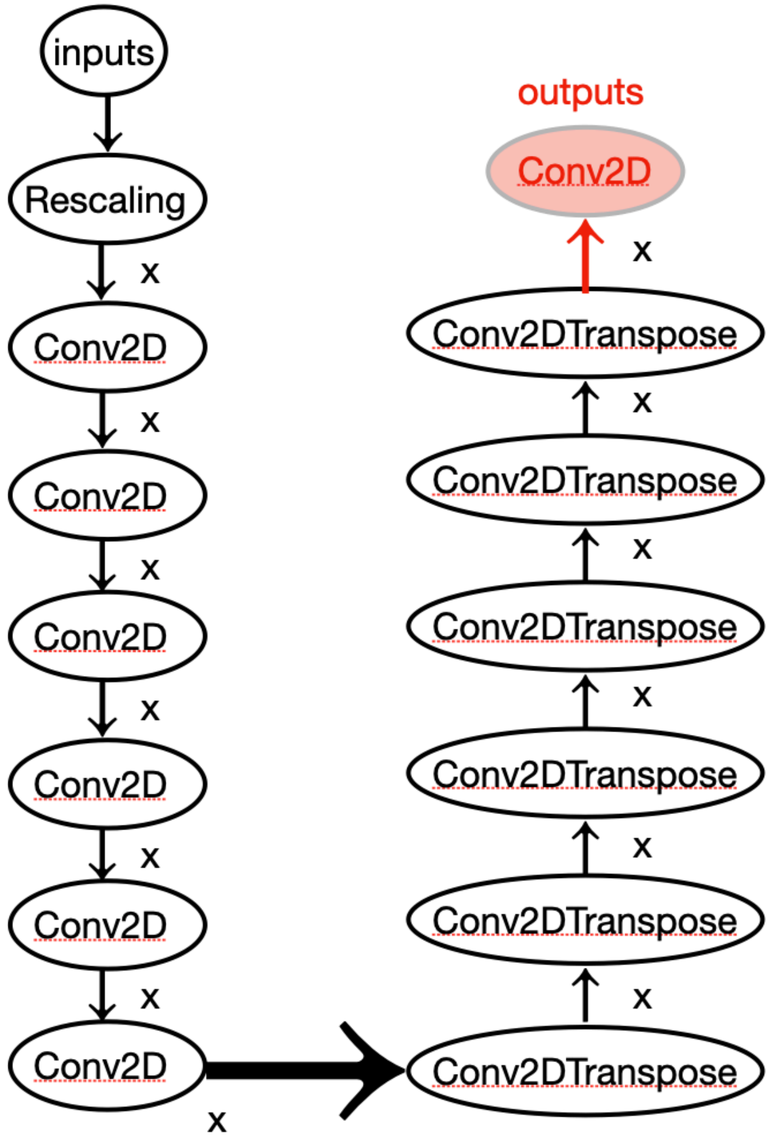


Image segmentation: Build the model

```
from tensorflow import keras
from tensorflow.keras import layers

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))
    x = layers.Rescaling(1./255)(inputs)

    x = layers.Conv2D(64, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(256, 3, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(256, 3, activation="relu", padding="same")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        256, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        128, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        64, 3, activation="relu", padding="same", strides=2)(x)

    outputs = layers.Conv2D(num_classes, 3, activation="softmax",
     padding="same")(x)

    model = keras.Model(inputs, outputs)
    return model

model = get_model(img_size=img_size, num_classes=3)
model.summary()
```

Don't forget to rescale input images to the [0-1] range.

Note how we use padding="same" everywhere to avoid the influence of border padding on feature map size.

We end the model with a per-pixel three-way softmax to classify each output pixel into one of our three categories.

model

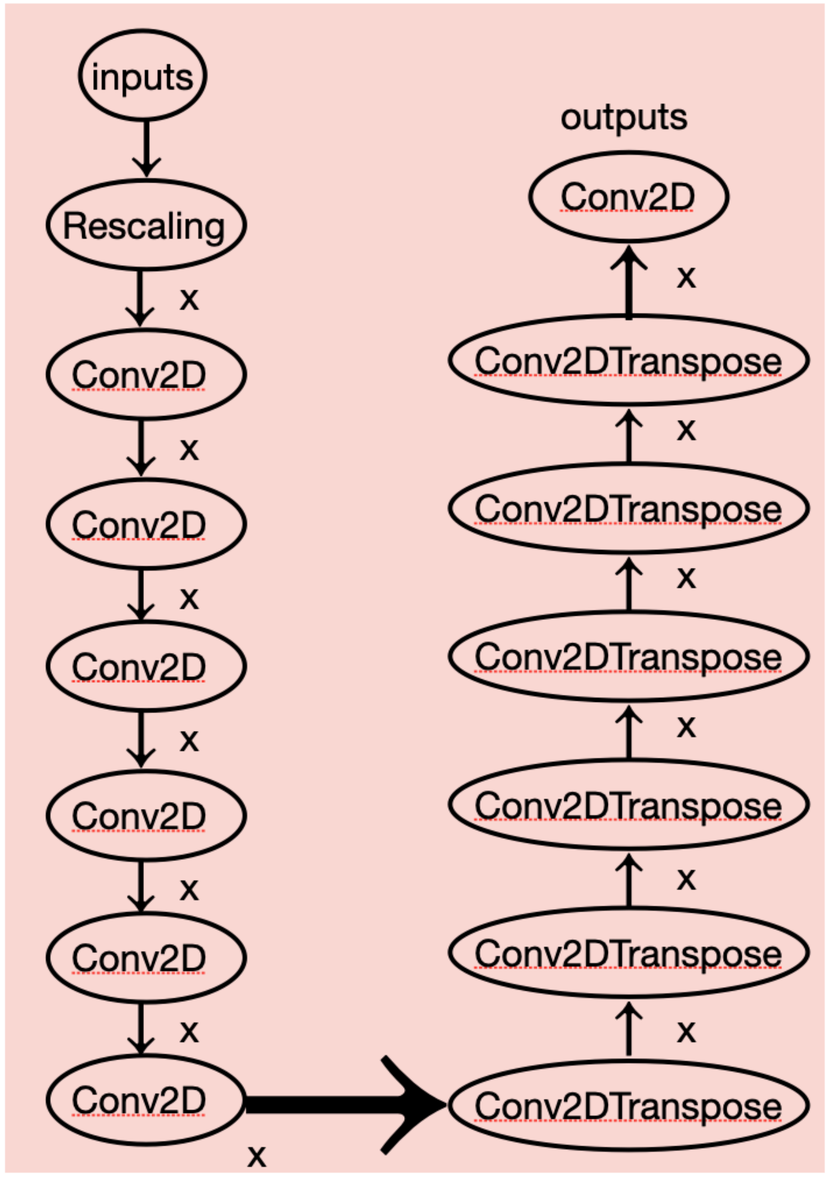


Image segmentation: Build the model

```
from tensorflow import keras
from tensorflow.keras import layers

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))
    x = layers.Rescaling(1./255)(inputs)

    x = layers.Conv2D(64, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(256, 3, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(256, 3, activation="relu", padding="same")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        256, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        128, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(
        64, 3, activation="relu", padding="same", strides=2)(x)

    outputs = layers.Conv2D(num_classes, 3, activation="softmax",
     padding="same")(x)

    model = keras.Model(inputs, outputs)
    return model

model = get_model(img_size=img_size, num_classes=3)
model.summary()
```

Don't forget to rescale input images to the [0-1] range.

Note how we use padding="same" everywhere to avoid the influence of border padding on feature map size.

We end the model with a per-pixel three-way softmax to classify each output pixel into one of our three categories.

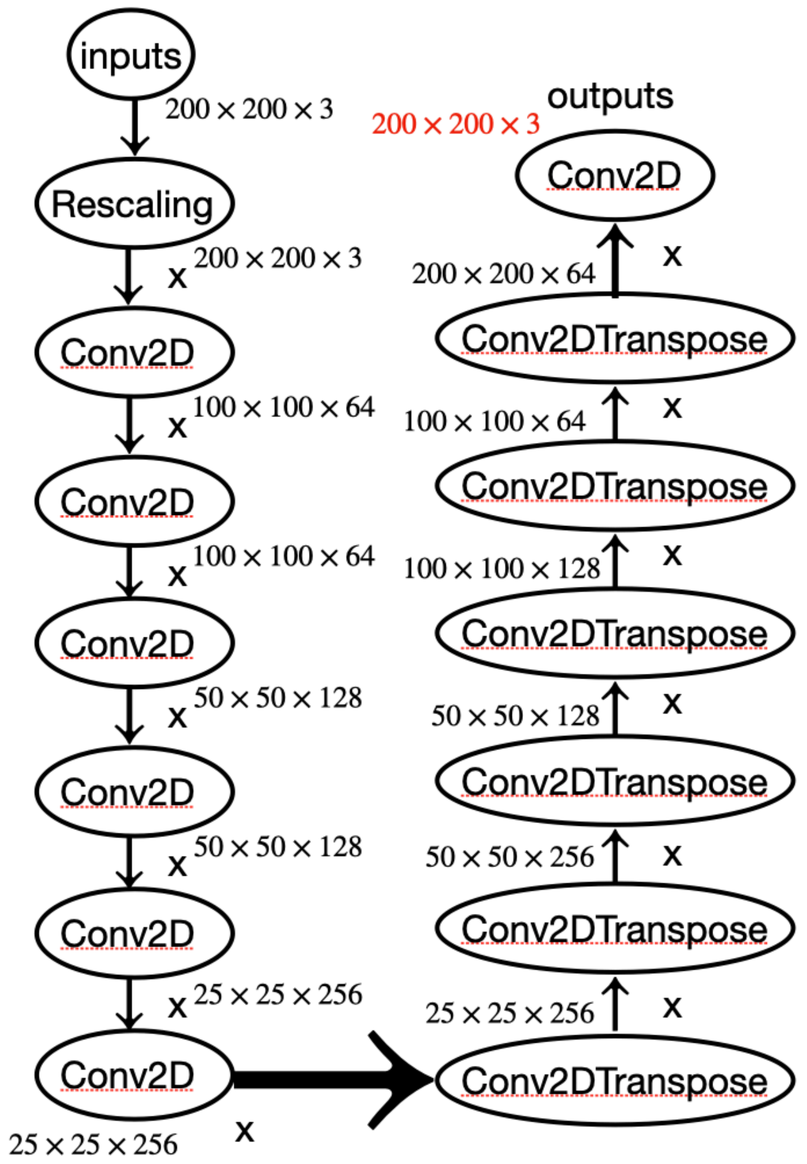


Image segmentation: Build the model

Reason we used “stride” instead of “maxpooling”:

Preserve spatial location of information, because we need to “reconstruct” the image in a sense.

“Convolution” often downsamples the image, and
“Deconvolution” often upsamples the image.

Image segmentation: Compile and Train the model

```
model.compile(optimizer="rmsprop", loss="sparse_categorical_crossentropy")

callbacks = [
    keras.callbacks.ModelCheckpoint("oxford_segmentation.keras",
                                    save_best_only=True)
]

history = model.fit(train_input_imgs, train_targets,
                    epochs=50,
                    callbacks=callbacks,
                    batch_size=64,
                    validation_data=(val_input_imgs, val_targets))
```

Image segmentation: Check training/validation performance

```
epochs = range(1, len(history.history["loss"]) + 1)
loss = history.history["loss"]
val_loss = history.history["val_loss"]
plt.figure()
plt.plot(epochs, loss, "bo", label="Training loss")
plt.plot(epochs, val_loss, "b", label="Validation loss")
plt.title("Training and validation loss")
plt.legend()
```

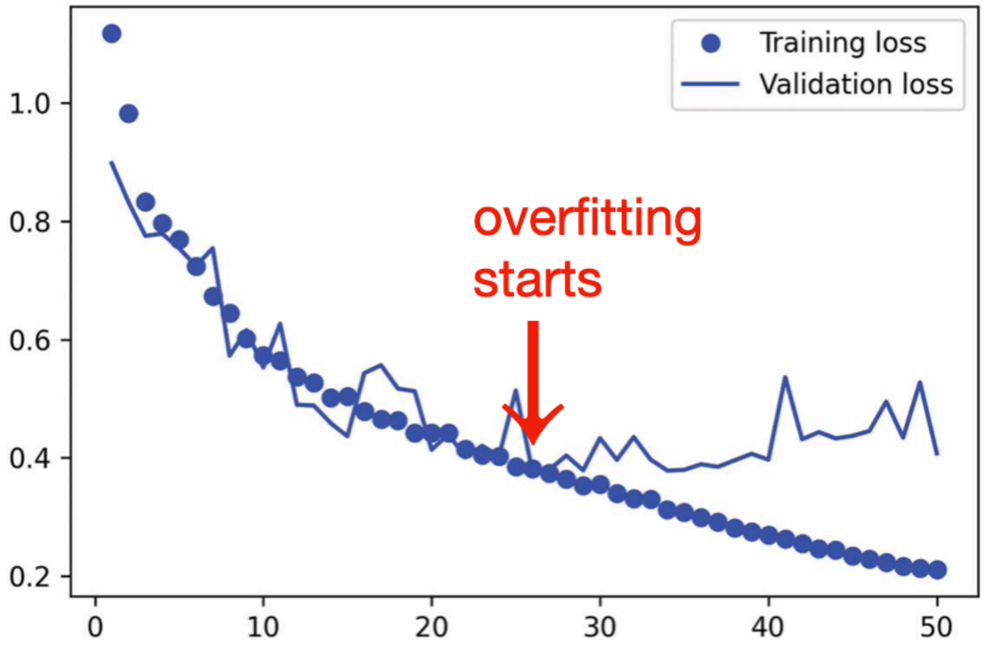


Figure 9.5 Displaying training and validation loss curves

Image segmentation: use (best saved) trained model for prediction

```
from tensorflow.keras.utils import array_to_img

model = keras.models.load_model("oxford_segmentation.keras")

i = 4
test_image = val_input_imgs[i]
plt.axis("off")
plt.imshow(array_to_img(test_image))

mask = model.predict(np.expand_dims(test_image, 0))[0]

def display_mask(pred):
    mask = np.argmax(pred, axis=-1)
    mask *= 127
    plt.axis("off")
    plt.imshow(mask)

display_mask(mask)
```

Utility to display a model's prediction

Image segmentation: use (best saved) trained model for prediction

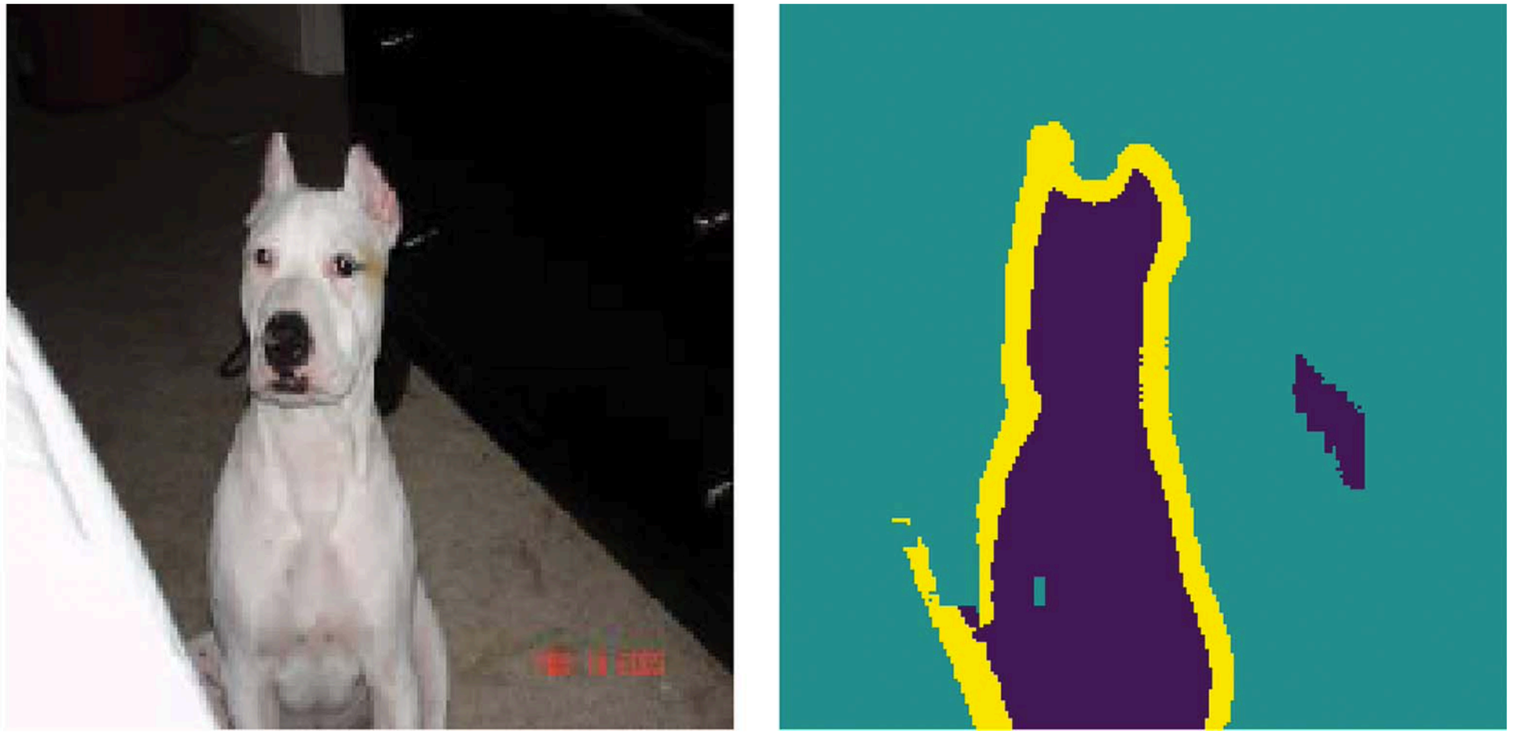


Figure 9.6 A test image and its predicted segmentation mask

Quiz questions:

1. How to build an image segmentation network?

2. What is the difference between semantic segmentation and instance segmentation?

Roadmap of this lecture:

1. Image segmentation.
2. CNN architecture patterns
    - 2.1 Residual connection
    - 2.2 Batch normalization and depthwise separable convolution
    - 2.3 Put them together for an Xception-like model
3. Visualize and interpret what CNN learns
    - 3.1 Visualize intermediate activations
    - 3.2 Visualize convolution filters
    - 3.3 Visualize heatmaps of class activation

A few important techniques for CNNs

1. Residual connections

2. Batch normalization

3. Separable convolution

Architecture design for models:
Modularity, hierarchy, and reuse

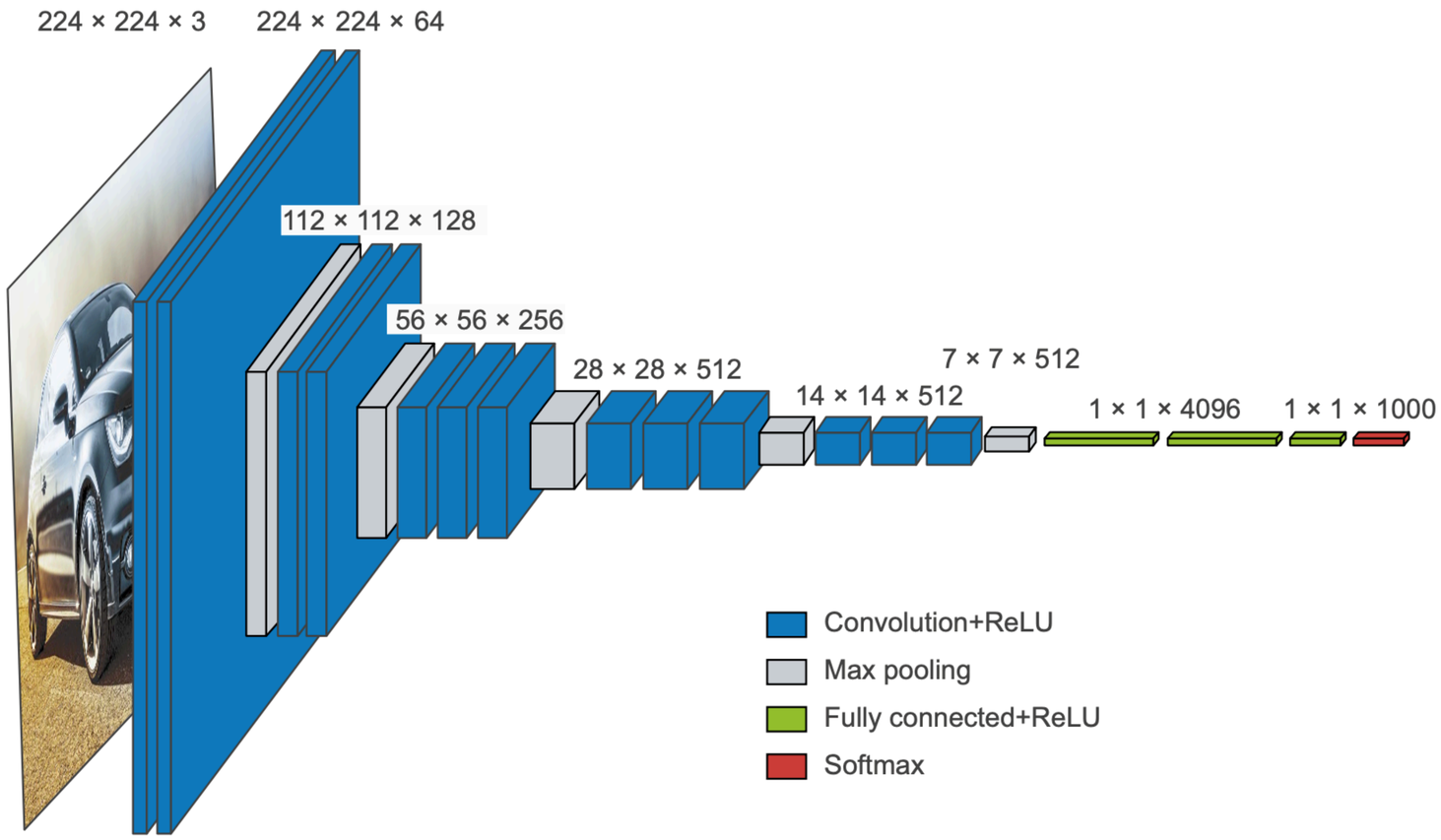
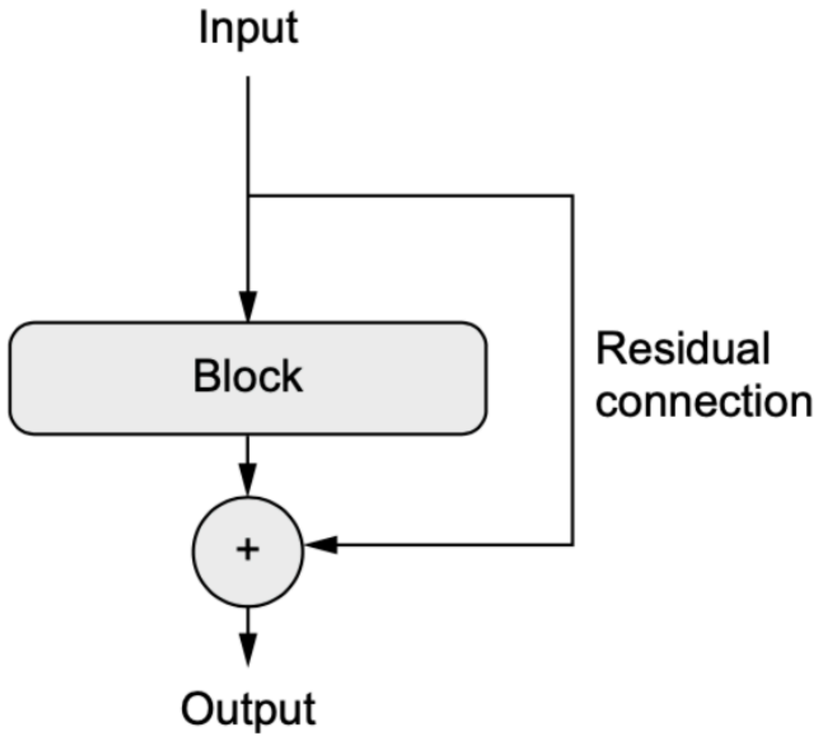


Figure 9.8 The VGG16 architecture: note the repeated layer blocks and the pyramid-like structure of the feature maps

Residual Connections

Residual connections: reduce “vanishing gradients” problem, allow us to train very deep networks.



Residual Connections

In practice, you'd implement a residual connection as follows.

Listing 9.1 A residual connection in pseudocode

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

Some input tensor

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

Residual Connections

In practice, you'd implement a residual connection as follows.

Listing 9.1 A residual connection in pseudocode

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

Some input tensor

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

Residual Connections

In practice, you'd implement a residual connection as follows.

Listing 9.1 A residual connection in pseudocode

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

Some input tensor

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

Residual Connections

In practice, you'd implement a residual connection as follows.

Listing 9.1 A residual connection in pseudocode

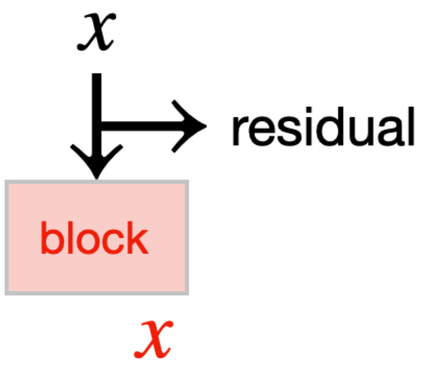
```
x = ...
residual = x
x = block(x)
x = add([x, residual])
```

Some input tensor

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



Residual Connections

In practice, you'd implement a residual connection as follows.

Listing 9.1 A residual connection in pseudocode

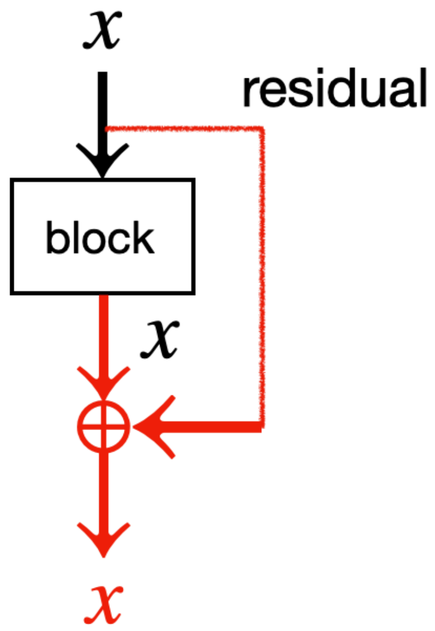
```
x = ...
residual = x
x = block(x)
x = add([x, residual])
```

Some input tensor

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



Residual Connections: keep shapes the same (for adding)

Listing 9.2 Residual block where the number of filters changes

```
from tensorflow import keras
from tensorflow.keras import layers

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)
residual = layers.Conv2D(64, 1)(residual)
x = layers.add([x, residual])
```

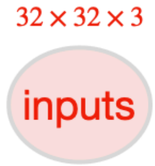
Set aside the residual.

This is the layer around which we create a residual connection: it increases the number of output filers from 32 to 64. Note that we use padding="same" to avoid downsampling due to padding.

The residual only had 32 filters, so we use a 1 × 1 Conv2D to project it to the correct shape.

Now the block output and the residual have the same shape and can be added.

Residual Connections: keep shapes the same (for adding)



Listing 9.2 Residual block where the number of filters changes

```
from tensorflow import keras
from tensorflow.keras import layers

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)
residual = layers.Conv2D(64, 1)(residual)
x = layers.add([x, residual])
```

Set aside the residual.

This is the layer around which we create a residual connection: it increases the number of output filers from 32 to 64. Note that we use padding="same" to avoid downsampling due to padding.

The residual only had 32 filters, so we use a 1 × 1 Conv2D to project it to the correct shape.

Now the block output and the residual have the same shape and can be added.

Residual Connections: keep shapes the same (for adding)

Listing 9.2 Residual block where the number of filters changes

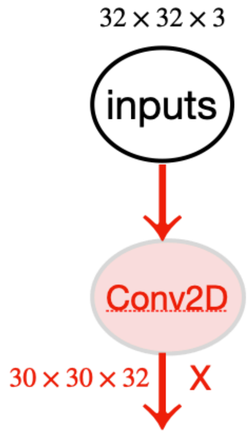
```
from tensorflow import keras
from tensorflow.keras import layers

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)
residual = layers.Conv2D(64, 1)(residual)
x = layers.add([x, residual])
```

Set aside the residual.

This is the layer around which we create a residual connection: it increases the number of output filers from 32 to 64. Note that we use padding="same" to avoid downsampling due to padding.

The residual only had 32 filters, so we use a 1 × 1 Conv2D to project it to the correct shape.

Now the block output and the residual have the same shape and can be added.



Residual Connections: keep shapes the same (for adding)

Listing 9.2 Residual block where the number of filters changes

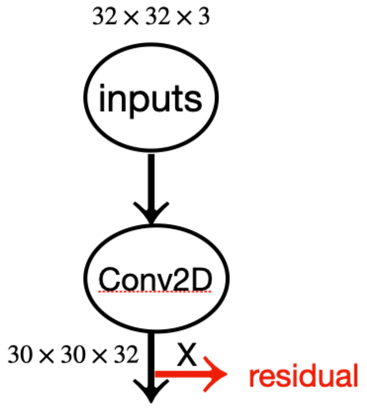
```
from tensorflow import keras
from tensorflow.keras import layers

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)
residual = layers.Conv2D(64, 1)(residual)
x = layers.add([x, residual])
```

Set aside the residual.

This is the layer around which we create a residual connection: it increases the number of output filers from 32 to 64. Note that we use padding="same" to avoid downsampling due to padding.

The residual only had 32 filters, so we use a 1 × 1 Conv2D to project it to the correct shape.

Now the block output and the residual have the same shape and can be added.



Residual Connections: keep shapes the same (for adding)

Listing 9.2 Residual block where the number of filters changes

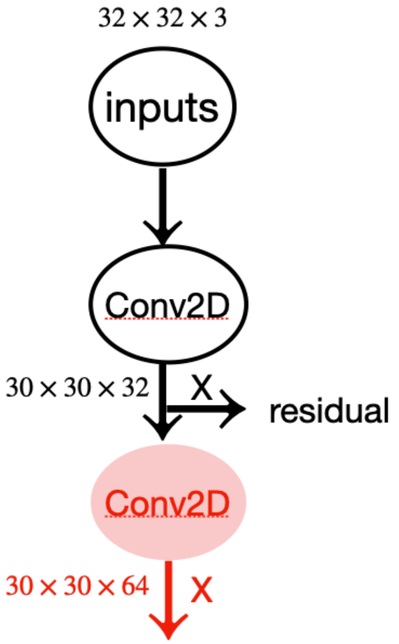
```
from tensorflow import keras
from tensorflow.keras import layers

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)
residual = layers.Conv2D(64, 1)(residual)
x = layers.add([x, residual])
```

Set aside the residual.

This is the layer around which we create a residual connection: it increases the number of output filers from 32 to 64. Note that we use padding="same" to avoid downsampling due to padding.

The residual only had 32 filters, so we use a 1 × 1 Conv2D to project it to the correct shape.

Now the block output and the residual have the same shape and can be added.



Residual Connections: keep shapes the same (for adding)

Listing 9.2 Residual block where the number of filters changes

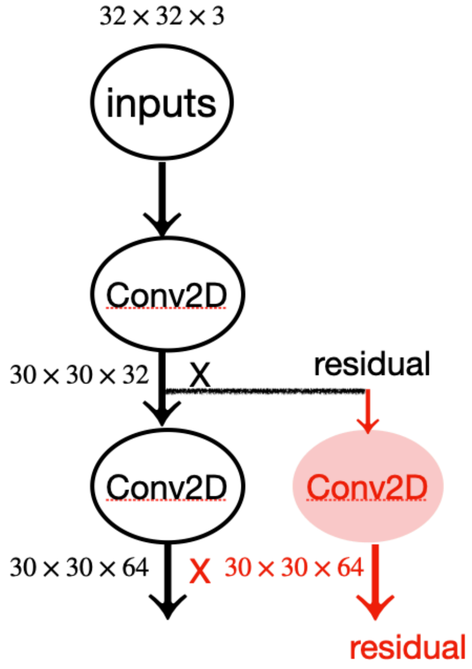
```
from tensorflow import keras
from tensorflow.keras import layers

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)
residual = layers.Conv2D(64, 1)(residual)
x = layers.add([x, residual])
```

Set aside the residual.

This is the layer around which we create a residual connection: it increases the number of output filers from 32 to 64. Note that we use padding="same" to avoid downsampling due to padding.

The residual only had 32 filters, so we use a 1 × 1 Conv2D to project it to the correct shape.

Now the block output and the residual have the same shape and can be added.



Residual Connections: keep shapes the same (for adding)

Listing 9.2 Residual block where the number of filters changes

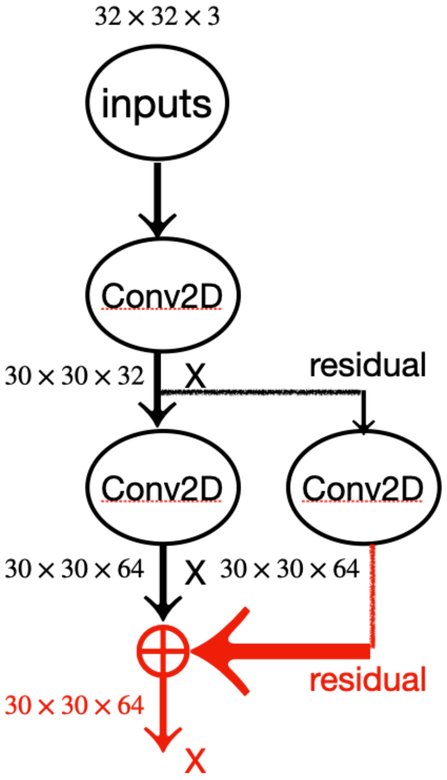
```
from tensorflow import keras
from tensorflow.keras import layers

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)
residual = layers.Conv2D(64, 1)(residual)
x = layers.add([x, residual])
```

Set aside the residual.

This is the layer around which we create a residual connection: it increases the number of output filers from 32 to 64. Note that we use padding="same" to avoid downsampling due to padding.

The residual only had 32 filters, so we use a 1 × 1 Conv2D to project it to the correct shape.

Now the block output and the residual have the same shape and can be added.



Residual Connections: keep shapes the same (for adding)

Listing 9.3 Case where the target block includes a max pooling layer

```
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])
```

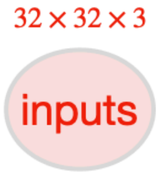
Set aside the residual.

Now the block output and the residual have the same shape and can be added.

We use strides=2 in the residual projection to match the downsampling created by the max pooling layer.

This is the block of two layers around which we create a residual connection: it includes a 2 × 2 max pooling layer. Note that we use padding="same" in both the convolution layer and the max pooling layer to avoid downsampling due to padding.

Residual Connections: keep shapes the same (for adding)



Listing 9.3 Case where the target block includes a max pooling layer

```
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])
```

Set aside the residual.

Now the block output and the residual have the same shape and can be added.

We use strides=2 in the residual projection to match the downsampling created by the max pooling layer.

This is the block of two layers around which we create a residual connection: it includes a 2 × 2 max pooling layer. Note that we use padding="same" in both the convolution layer and the max pooling layer to avoid downsampling due to padding.

Residual Connections: keep shapes the same (for adding)

Listing 9.3 Case where the target block includes a max pooling layer

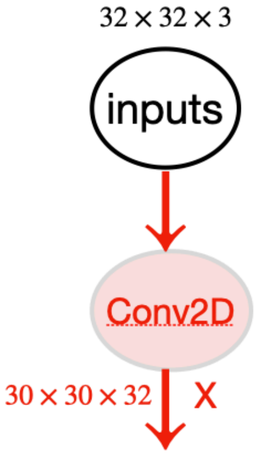
```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Conv2D(32, 3, activation="relu")(inputs)
residual = x
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
x = layers.MaxPooling2D(2, padding="same")(x)
residual = layers.Conv2D(64, 1, strides=2)(residual)
x = layers.add([x, residual])
```

Set aside the residual.

Now the block output and the residual have the same shape and can be added.

We use strides=2 in the residual projection to match the downsampling created by the max pooling layer.

This is the block of two layers around which we create a residual connection: it includes a 2 × 2 max pooling layer. Note that we use padding="same" in both the convolution layer and the max pooling layer to avoid downsampling due to padding.



Residual Connections: keep shapes the same (for adding)

Listing 9.3 Case where the target block includes a max pooling layer

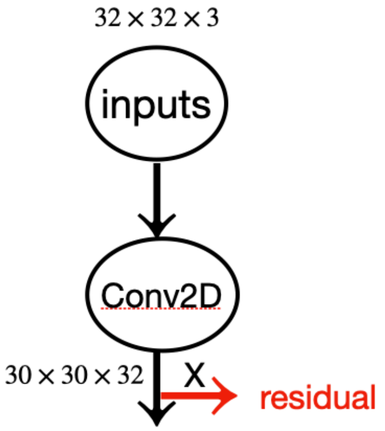
```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Conv2D(32, 3, activation="relu")(inputs)
residual = x
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
x = layers.MaxPooling2D(2, padding="same")(x)
residual = layers.Conv2D(64, 1, strides=2)(residual)
x = layers.add([x, residual])
```

Set aside the residual.

Now the block output and the residual have the same shape and can be added.

We use strides=2 in the residual projection to match the downsampling created by the max pooling layer.

This is the block of two layers around which we create a residual connection: it includes a 2 × 2 max pooling layer. Note that we use padding="same" in both the convolution layer and the max pooling layer to avoid downsampling due to padding.



Residual Connections: keep shapes the same (for adding)

Listing 9.3 Case where the target block includes a max pooling layer

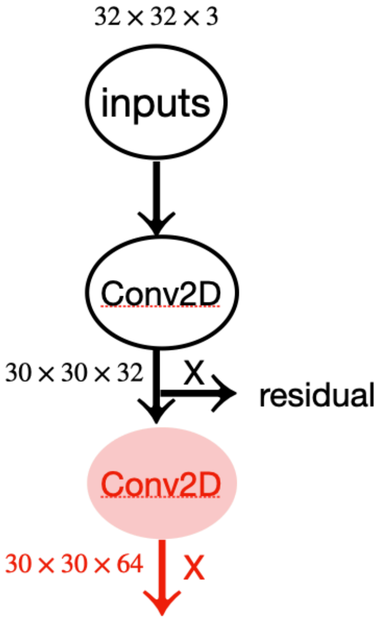
```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Conv2D(32, 3, activation="relu")(inputs)
residual = x
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
x = layers.MaxPooling2D(2, padding="same")(x)
residual = layers.Conv2D(64, 1, strides=2)(residual)
x = layers.add([x, residual])
```

Set aside the residual.

Now the block output and the residual have the same shape and can be added.

We use strides=2 in the residual projection to match the downsampling created by the max pooling layer.

This is the block of two layers around which we create a residual connection: it includes a 2 × 2 max pooling layer. Note that we use padding="same" in both the convolution layer and the max pooling layer to avoid downsampling due to padding.



Residual Connections: keep shapes the same (for adding)

Listing 9.3 Case where the target block includes a max pooling layer

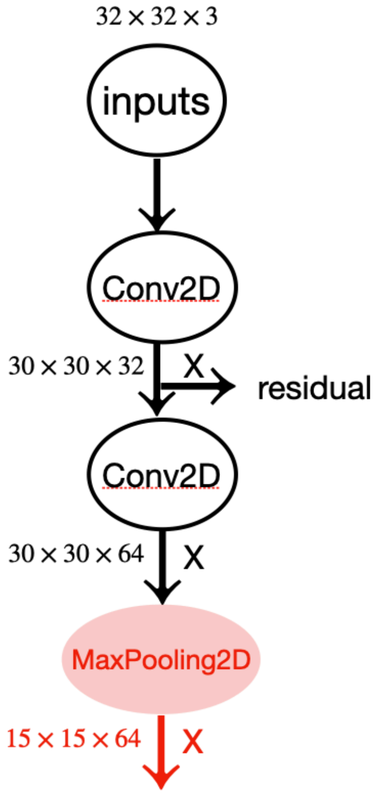
```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Conv2D(32, 3, activation="relu")(inputs)
residual = x
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
x = layers.MaxPooling2D(2, padding="same")(x)
residual = layers.Conv2D(64, 1, strides=2)(residual)
x = layers.add([x, residual])
```

Set aside the residual.

Now the block output and the residual have the same shape and can be added.

We use strides=2 in the residual projection to match the downsampling created by the max pooling layer.

This is the block of two layers around which we create a residual connection: it includes a 2 × 2 max pooling layer. Note that we use padding="same" in both the convolution layer and the max pooling layer to avoid downsampling due to padding.



Residual Connections: keep shapes the same (for adding)

Listing 9.3 Case where the target block includes a max pooling layer

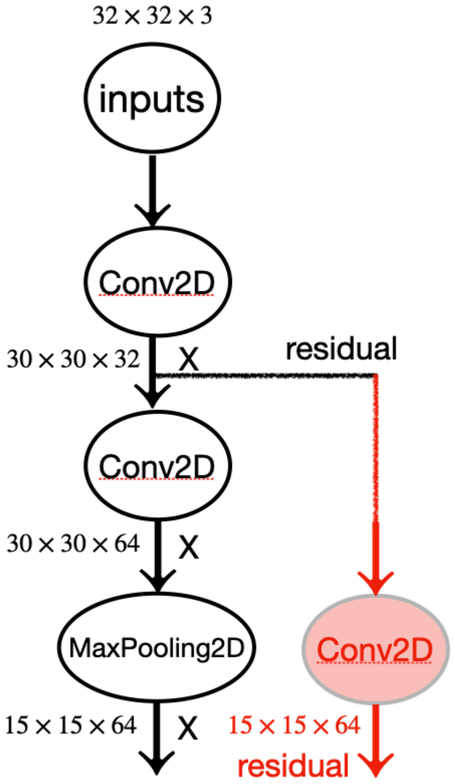
```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Conv2D(32, 3, activation="relu")(inputs)
residual = x
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
x = layers.MaxPooling2D(2, padding="same")(x)
residual = layers.Conv2D(64, 1, strides=2)(residual)
x = layers.add([x, residual])
```

Set aside the residual.

Now the block output and the residual have the same shape and can be added.

We use strides=2 in the residual projection to match the downsampling created by the max pooling layer.

This is the block of two layers around which we create a residual connection: it includes a 2 × 2 max pooling layer. Note that we use padding="same" in both the convolution layer and the max pooling layer to avoid downsampling due to padding.



Residual Connections: keep shapes the same (for adding)

Listing 9.3 Case where the target block includes a max pooling layer

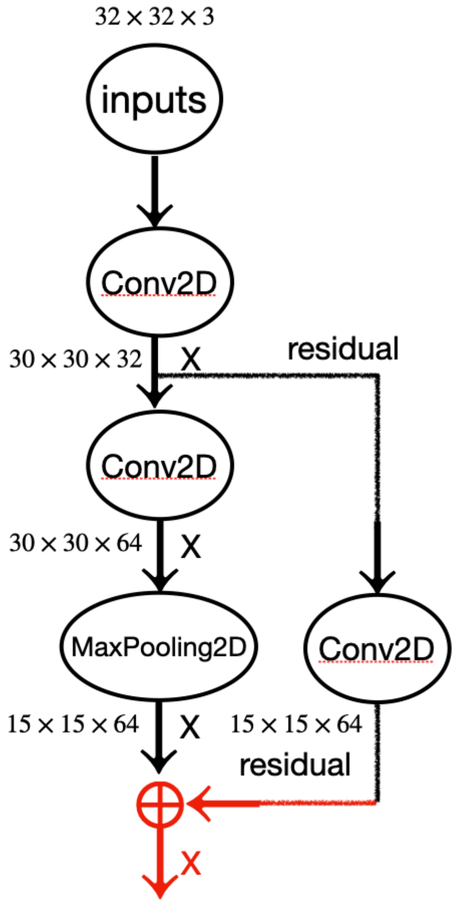
```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Conv2D(32, 3, activation="relu")(inputs)
residual = x
x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
x = layers.MaxPooling2D(2, padding="same")(x)
residual = layers.Conv2D(64, 1, strides=2)(residual)
x = layers.add([x, residual])
```

Set aside the residual.

Now the block output and the residual have the same shape and can be added.

We use strides=2 in the residual projection to match the downsampling created by the max pooling layer.

This is the block of two layers around which we create a residual connection: it includes a 2 × 2 max pooling layer. Note that we use padding="same" in both the convolution layer and the max pooling layer to avoid downsampling due to padding.



Residual Connections: example of a simple CNN

To make these ideas more concrete, here is an example of a simple CNN structured into a series of blocks, each made of two convolution layers and one optional max pooling layer, with a residual connection around each block.

Residual Connections: example of a simple CNN

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

def residual_block(x, filters, pooling=False):
    residual = x
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)

    if pooling:
        x = layers.MaxPooling2D(2, padding="same")(x)
        residual = layers.Conv2D(filters, 1, strides=2)(residual)
    elif filters != residual.shape[-1]:
        residual = layers.Conv2D(filters, 1)(residual)
    x = layers.add([x, residual])
    return x

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

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

Utility function to apply a convolutional block with a residual connection, with an option to add max pooling

If we use max pooling, we add a strided convolution to project the residual to the expected shape.

If we don't use max pooling, we only project the residual if the number of channels has changed.

First block

Second block; note the increasing filter count in each block.

The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.

Residual Connections: example of a simple CNN

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

def residual_block(x, filters, pooling=False):
    residual = x
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)

    if pooling:
        x = layers.MaxPooling2D(2, padding="same")(x)
        residual = layers.Conv2D(filters, 1, strides=2)(residual)
    elif filters != residual.shape[-1]:
        residual = layers.Conv2D(filters, 1)(residual)
    x = layers.add([x, residual])
    return x

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

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

Utility function to apply a convolutional block with a residual connection, with an option to add max pooling

$32 \times 32 \times 3$

inputs

If we use max pooling, we add a strided convolution to project the residual to the expected shape.

If we don't use max pooling, we only project the residual if the number of channels has changed.

First block

Second block; note the increasing filter count in each block.

The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.

Residual Connections: example of a simple CNN

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

def residual_block(x, filters, pooling=False):
    residual = x
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)

    if pooling:
        x = layers.MaxPooling2D(2, padding="same")(x)
        residual = layers.Conv2D(filters, 1, strides=2)(residual)
    elif filters != residual.shape[-1]:
        residual = layers.Conv2D(filters, 1)(residual)
    x = layers.add([x, residual])
    return x

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

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

Utility function to apply a convolutional block with a residual connection, with an option to add max pooling

32 × 32 × 3

inputs

If we use max pooling, we add a strided convolution to project the residual to the expected shape.

First block

If we don't use max pooling, we only project the residual if the number of channels has changed.

Second block; note the increasing filter count in each block.

The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.

Residual Connections: example of a simple CNN

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

def residual_block(x, filters, pooling=False):
    residual = x
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)

    if pooling:
        x = layers.MaxPooling2D(2, padding="same")(x)
        residual = layers.Conv2D(filters, 1, strides=2)(residual)
    elif filters != residual.shape[-1]:
        residual = layers.Conv2D(filters, 1)(residual)
    x = layers.add([x, residual])
    return x

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

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

Utility function to apply a convolutional block with a residual connection, with an option to add max pooling

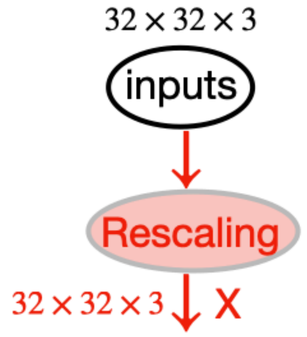
First block

If we use max pooling, we add a strided convolution to project the residual to the expected shape.

If we don't use max pooling, we only project the residual if the number of channels has changed.

Second block; note the increasing filter count in each block.

The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.



Residual Connections: example of a simple CNN

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

def residual_block(x, filters, pooling=False):
    residual = x
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)

    if pooling:
        x = layers.MaxPooling2D(2, padding="same")(x)
        residual = layers.Conv2D(filters, 1, strides=2)(residual)
    elif filters != residual.shape[-1]:
        residual = layers.Conv2D(filters, 1)(residual)
    x = layers.add([x, residual])
    return x

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

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

Utility function to apply a convolutional block with a residual connection, with an option to add max pooling

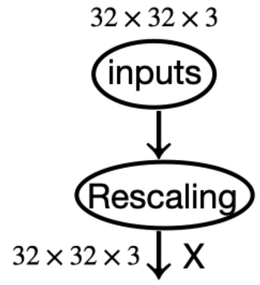
If we use max pooling, we add a strided convolution to project the residual to the expected shape.

If we don't use max pooling, we only project the residual if the number of channels has changed.

First block

Second block; note the increasing filter count in each block.

The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.



residual_block

x↓

Residual Connections: example of a simple CNN

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

def residual_block(x, filters, pooling=False):
    residual = x
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)

    if pooling:
        x = layers.MaxPooling2D(2, padding="same")(x)
        residual = layers.Conv2D(filters, 1, strides=2)(residual)
    elif filters != residual.shape[-1]:
        residual = layers.Conv2D(filters, 1)(residual)
    x = layers.add([x, residual])
    return x

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

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

Utility function to apply a convolutional block with a residual connection, with an option to add max pooling

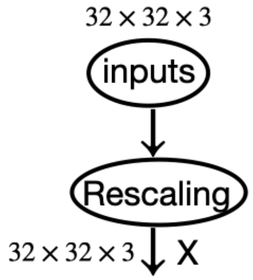
If we use max pooling, we add a strided convolution to project the residual to the expected shape.

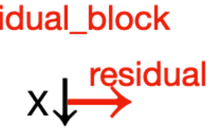
If we don't use max pooling, we only project the residual if the number of channels has changed.

First block

Second block; note the increasing filter count in each block.

The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.





Residual Connections: example of a simple CNN

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

def residual_block(x, filters, pooling=False):
    residual = x
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)

    if pooling:
        x = layers.MaxPooling2D(2, padding="same")(x)
        residual = layers.Conv2D(filters, 1, strides=2)(residual)
    elif filters != residual.shape[-1]:
        residual = layers.Conv2D(filters, 1)(residual)
    x = layers.add([x, residual])
    return x

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

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

Utility function to apply a convolutional block with a residual connection, with an option to add max pooling

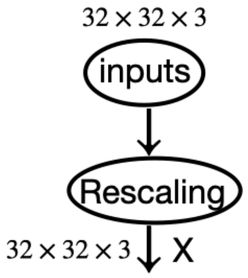
If we use max pooling, we add a strided convolution to project the residual to the expected shape.

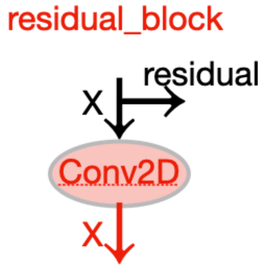
If we don't use max pooling, we only project the residual if the number of channels has changed.

First block

Second block; note the increasing filter count in each block.

The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.





Residual Connections: example of a simple CNN

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

def residual_block(x, filters, pooling=False):
    residual = x
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)

    if pooling:
        x = layers.MaxPooling2D(2, padding="same")(x)
        residual = layers.Conv2D(filters, 1, strides=2)(residual)
    elif filters != residual.shape[-1]:
        residual = layers.Conv2D(filters, 1)(residual)
    x = layers.add([x, residual])
    return x

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

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

Utility function to apply a convolutional block with a residual connection, with an option to add max pooling

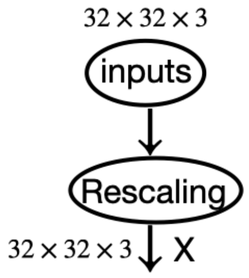
If we use max pooling, we add a strided convolution to project the residual to the expected shape.

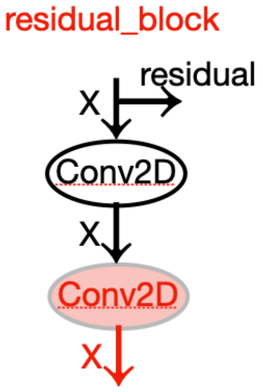
If we don't use max pooling, we only project the residual if the number of channels has changed.

First block

Second block; note the increasing filter count in each block.

The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.





Residual Connections: example of a simple CNN

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

def residual_block(x, filters, pooling=False):
    residual = x
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)

    if pooling:
        x = layers.MaxPooling2D(2, padding="same")(x)
        residual = layers.Conv2D(filters, 1, strides=2)(residual)
    elif filters != residual.shape[-1]:
        residual = layers.Conv2D(filters, 1)(residual)
    x = layers.add([x, residual])
    return x

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

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

Utility function to apply a convolutional block with a residual connection, with an option to add max pooling

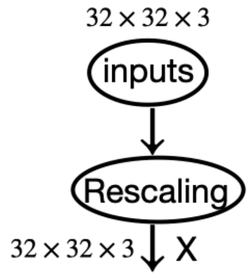
If we use max pooling, we add a strided convolution to project the residual to the expected shape.

If we don't use max pooling, we only project the residual if the number of channels has changed.

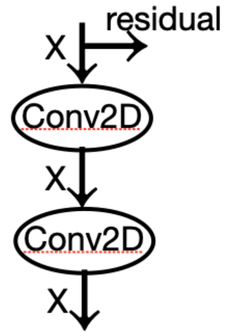
First block

Second block; note the increasing filter count in each block.

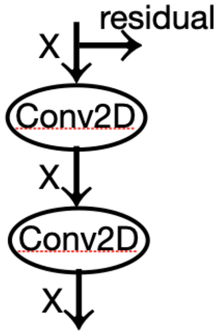
The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.



residual_block
pooling==True



residual_block
pooling==False



Residual Connections: example of a simple CNN

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

def residual_block(x, filters, pooling=False):
    residual = x
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)

    if pooling:
        x = layers.MaxPooling2D(2, padding="same")(x)
        residual = layers.Conv2D(filters, 1, strides=2)(residual)
    elif filters != residual.shape[-1]:
        residual = layers.Conv2D(filters, 1)(residual)
    x = layers.add([x, residual])
    return x

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

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

Utility function to apply a convolutional block with a residual connection, with an option to add max pooling

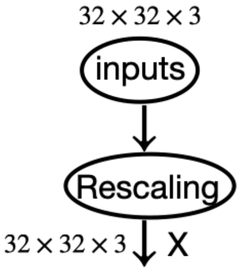
If we use max pooling, we add a strided convolution to project the residual to the expected shape.

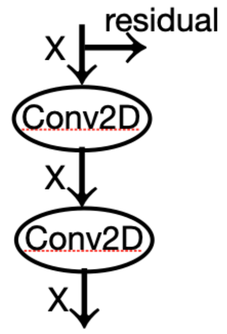
If we don't use max pooling, we only project the residual if the number of channels has changed.

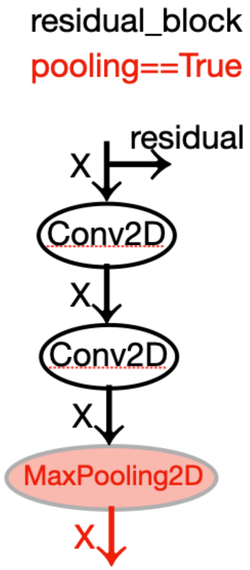
First block

Second block; note the increasing filter count in each block.

The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.







Residual Connections: example of a simple CNN

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

def residual_block(x, filters, pooling=False):
    residual = x
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)

    if pooling:
        x = layers.MaxPooling2D(2, padding="same")(x)
        residual = layers.Conv2D(filters, 1, strides=2)(residual)
    elif filters != residual.shape[-1]:
        residual = layers.Conv2D(filters, 1)(residual)
    x = layers.add([x, residual])
    return x

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

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

Utility function to apply a convolutional block with a residual connection, with an option to add max pooling

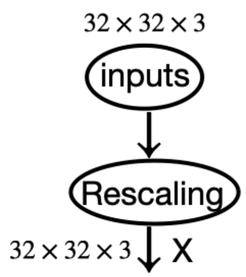
If we use max pooling, we add a strided convolution to project the residual to the expected shape.

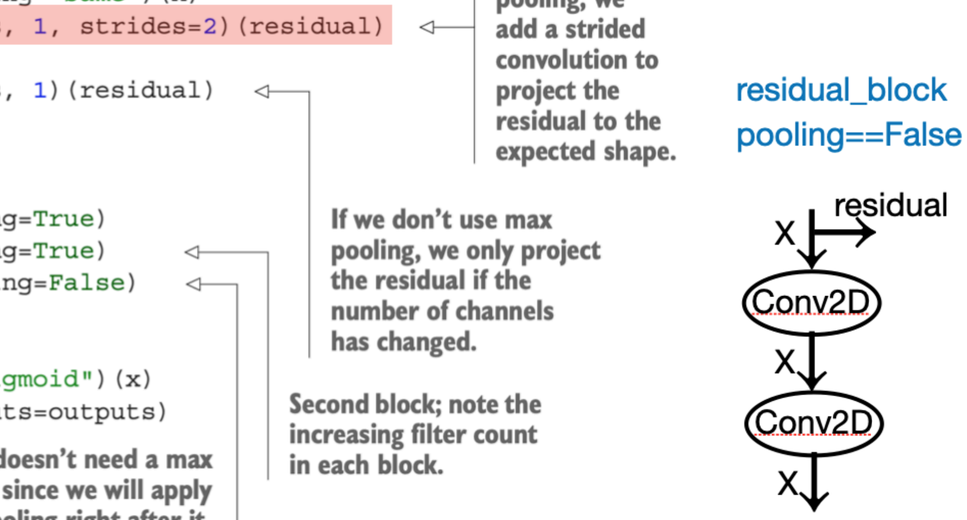
If we don't use max pooling, we only project the residual if the number of channels has changed.

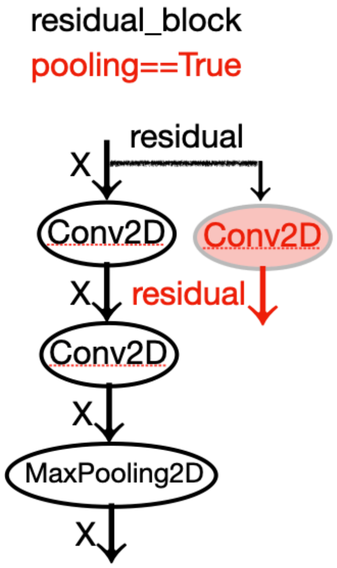
First block

Second block; note the increasing filter count in each block.

The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.







Residual Connections: example of a simple CNN

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

def residual_block(x, filters, pooling=False):
    residual = x
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)

    if pooling:
        x = layers.MaxPooling2D(2, padding="same")(x)
        residual = layers.Conv2D(filters, 1, strides=2)(residual)
    elif filters != residual.shape[-1]:
        residual = layers.Conv2D(filters, 1)(residual)
    x = layers.add([x, residual])
    return x

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

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

Utility function to apply a convolutional block with a residual connection, with an option to add max pooling

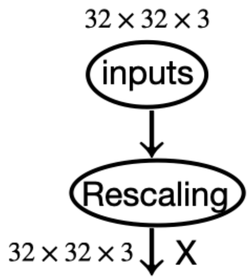
If we use max pooling, we add a strided convolution to project the residual to the expected shape.

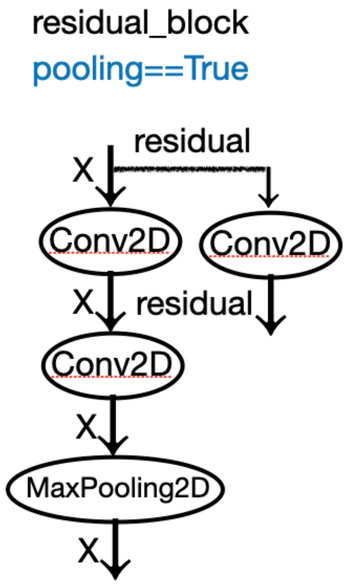
If we don't use max pooling, we only project the residual if the number of channels has changed.

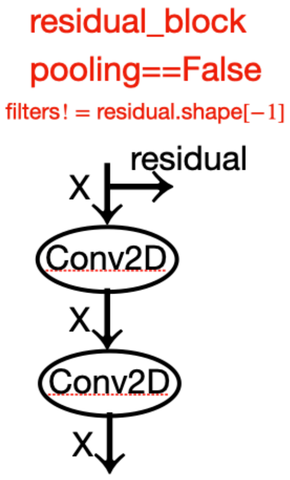
First block

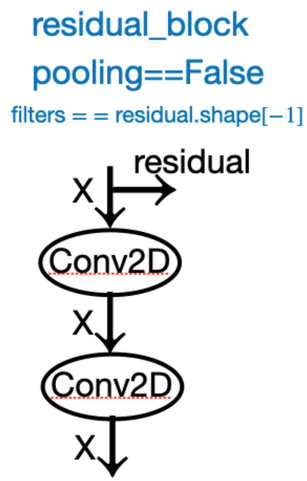
Second block; note the increasing filter count in each block.

The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.









Residual Connections: example of a simple CNN

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

def residual_block(x, filters, pooling=False):
    residual = x
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)

    if pooling:
        x = layers.MaxPooling2D(2, padding="same")(x)
        residual = layers.Conv2D(filters, 1, strides=2)(residual)
    elif filters != residual.shape[-1]:
        residual = layers.Conv2D(filters, 1)(residual)
    x = layers.add([x, residual])
    return x

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

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

Utility function to apply a convolutional block with a residual connection, with an option to add max pooling

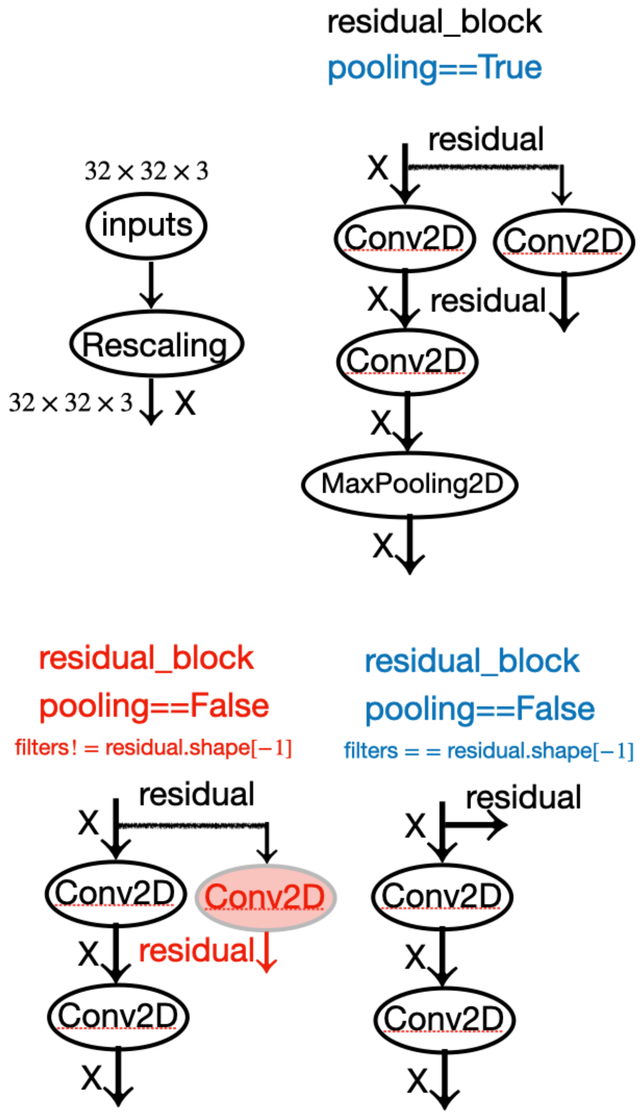
If we use max pooling, we add a strided convolution to project the residual to the expected shape.

If we don't use max pooling, we only project the residual if the number of channels has changed.

First block

Second block; note the increasing filter count in each block.

The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.



Residual Connections: example of a simple CNN

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

def residual_block(x, filters, pooling=False):
    residual = x
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)

    if pooling:
        x = layers.MaxPooling2D(2, padding="same")(x)
        residual = layers.Conv2D(filters, 1, strides=2)(residual)
    elif filters != residual.shape[-1]:
        residual = layers.Conv2D(filters, 1)(residual)
    x = layers.add([x, residual])
    return x

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

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

Utility function to apply a convolutional block with a residual connection, with an option to add max pooling

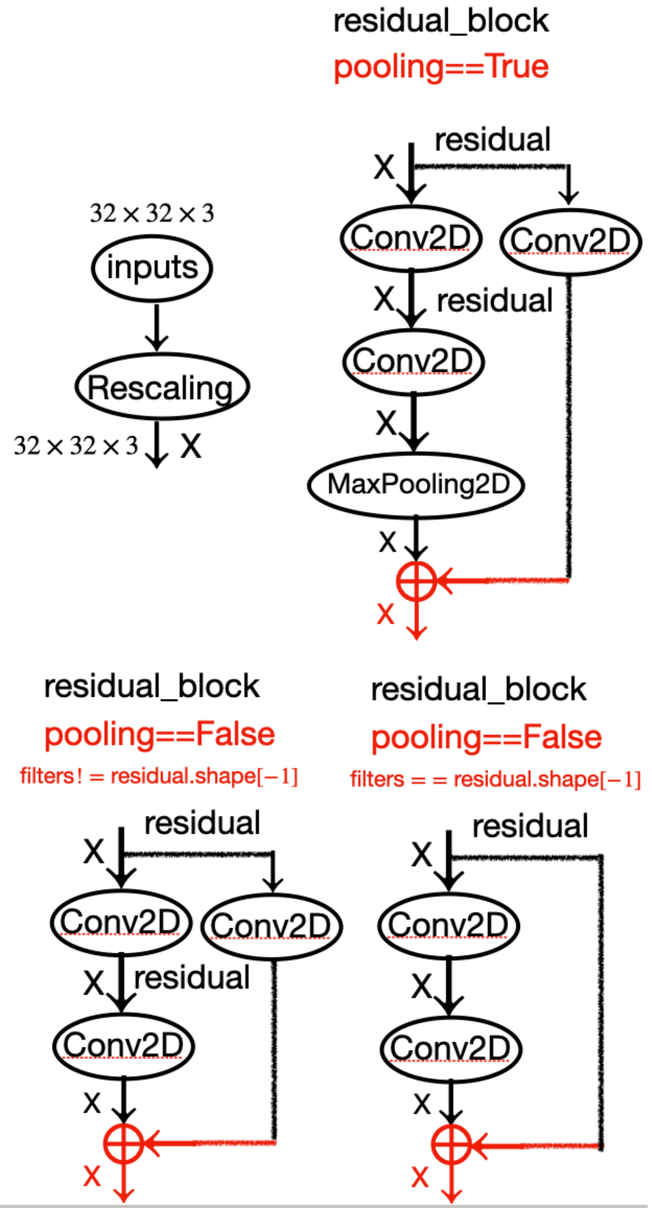
If we use max pooling, we add a strided convolution to project the residual to the expected shape.

If we don't use max pooling, we only project the residual if the number of channels has changed.

First block

Second block; note the increasing filter count in each block.

The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.



Residual Connections: example of a simple CNN

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

def residual_block(x, filters, pooling=False):
    residual = x
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(filters, 3, activation="relu", padding="same")(x)

    if pooling:
        x = layers.MaxPooling2D(2, padding="same")(x)
        residual = layers.Conv2D(filters, 1, strides=2)(residual)
    elif filters != residual.shape[-1]:
        residual = layers.Conv2D(filters, 1)(residual)
    x = layers.add([x, residual])
    return x

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

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

Utility function to apply a convolutional block with a residual connection, with an option to add max pooling

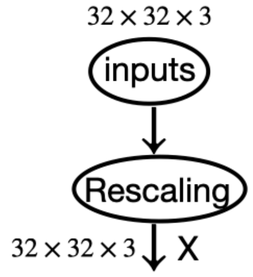
If we use max pooling, we add a strided convolution to project the residual to the expected shape.

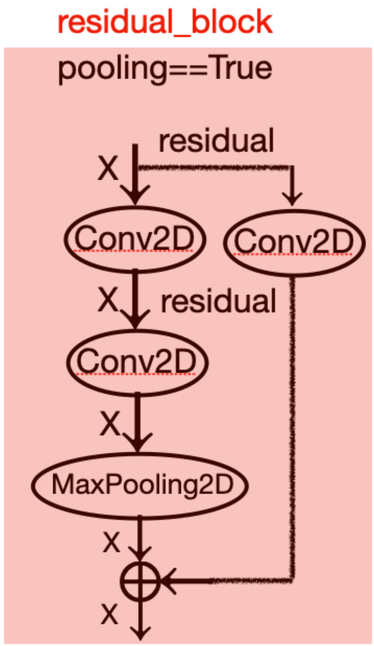
If we don't use max pooling, we only project the residual if the number of channels has changed.

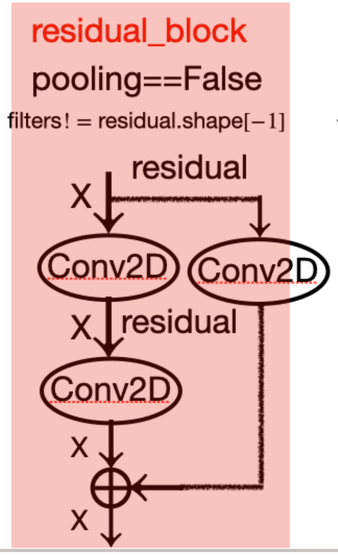
First block

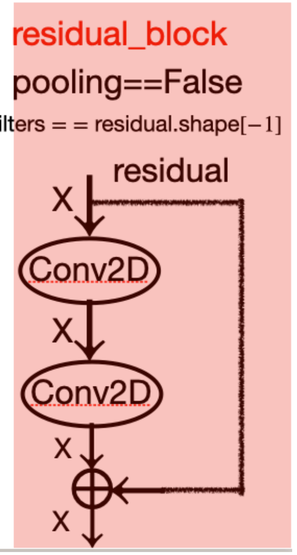
Second block; note the increasing filter count in each block.

The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.









Residual Connections: example of a simple CNN

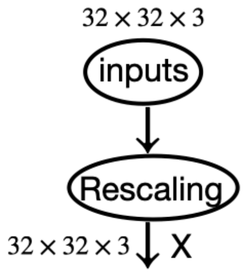
```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

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

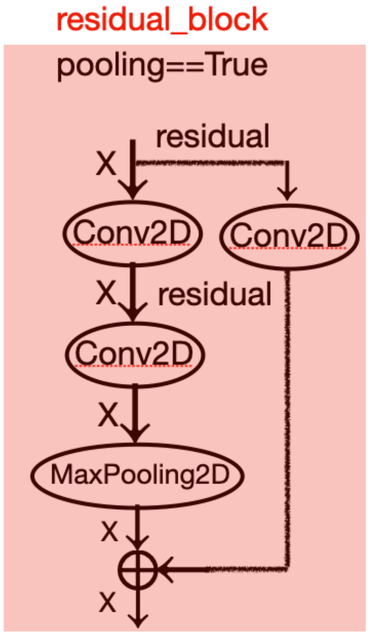
First block

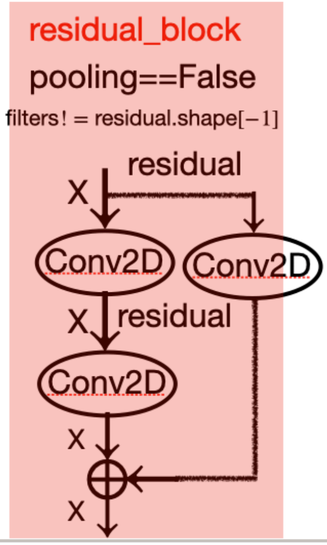
If we don't use max pooling, we only project the residual if the number of channels has changed.

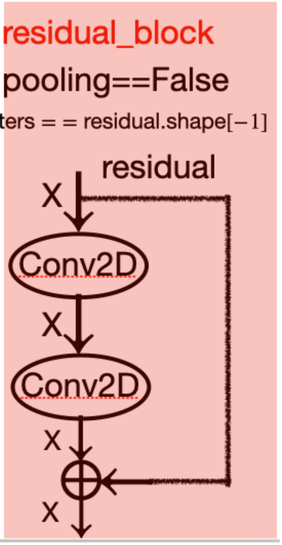
Second block; note the increasing filter count in each block.

The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.









Residual Connections: example of a simple CNN

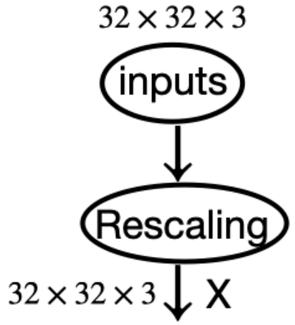
```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

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

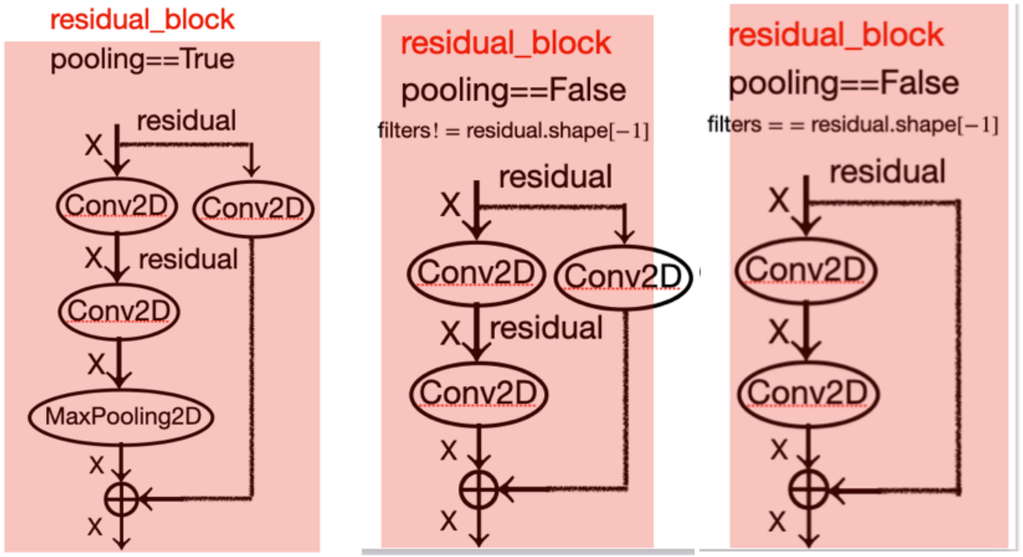
First block

If we don't use max pooling, we only project the residual if the number of channels has changed.

Second block; note the increasing filter count in each block.

The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.





Residual Connections: example of a simple CNN

```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)
```

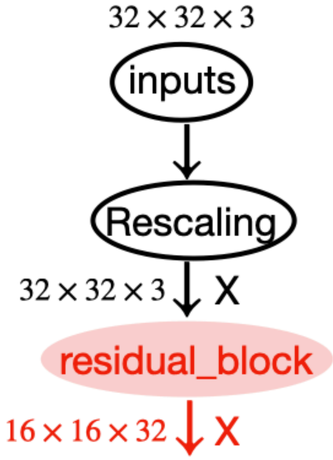
First block

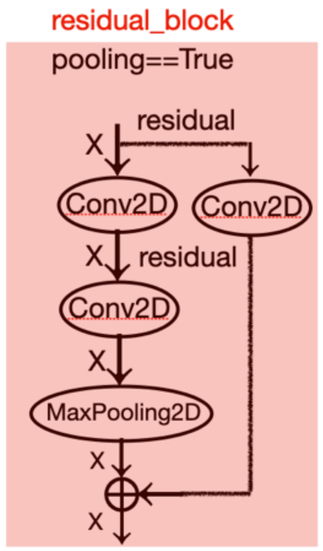
```
x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

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

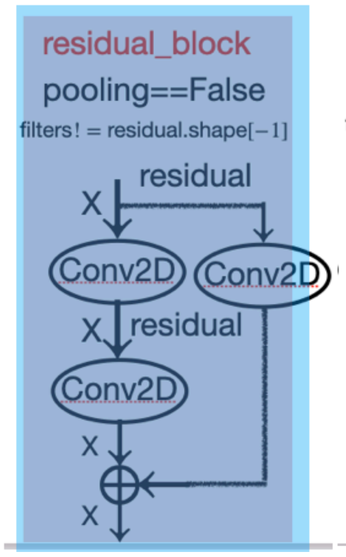
The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.

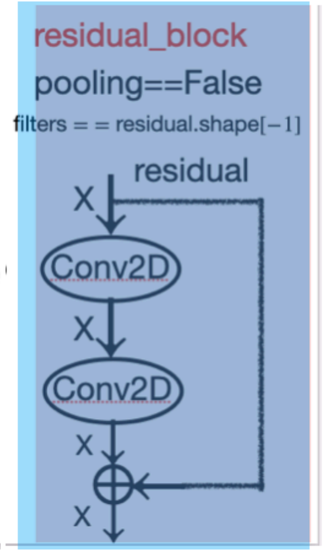
Second block; note the increasing filter count in each block.

If we don't use max pooling, we only project the residual if the number of channels has changed.









Residual Connections: example of a simple CNN

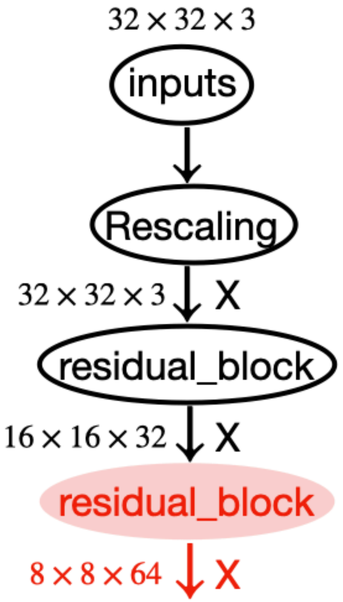
```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

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

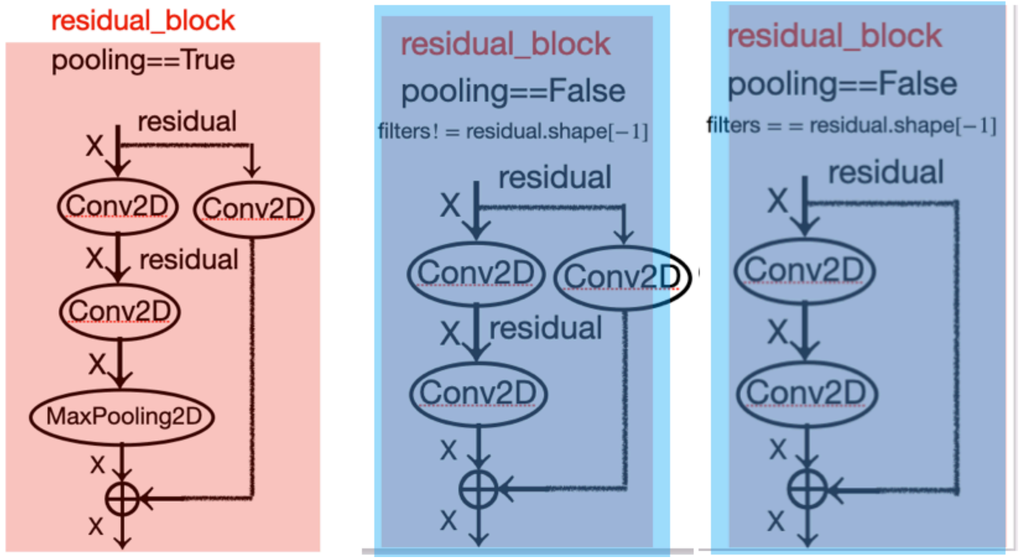
First block

If we don't use max pooling, we only project the residual if the number of channels has changed.

Second block; note the increasing filter count in each block.

The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.





Residual Connections: example of a simple CNN

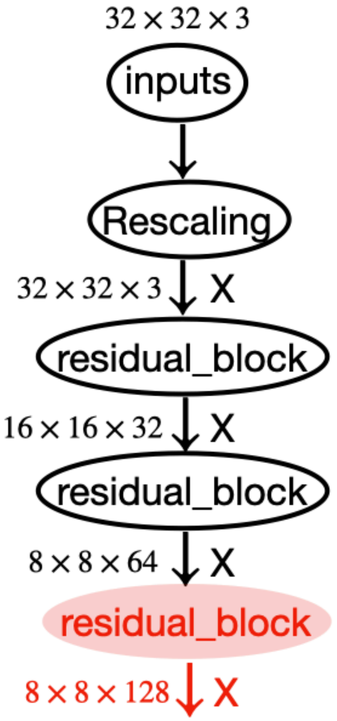
```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

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

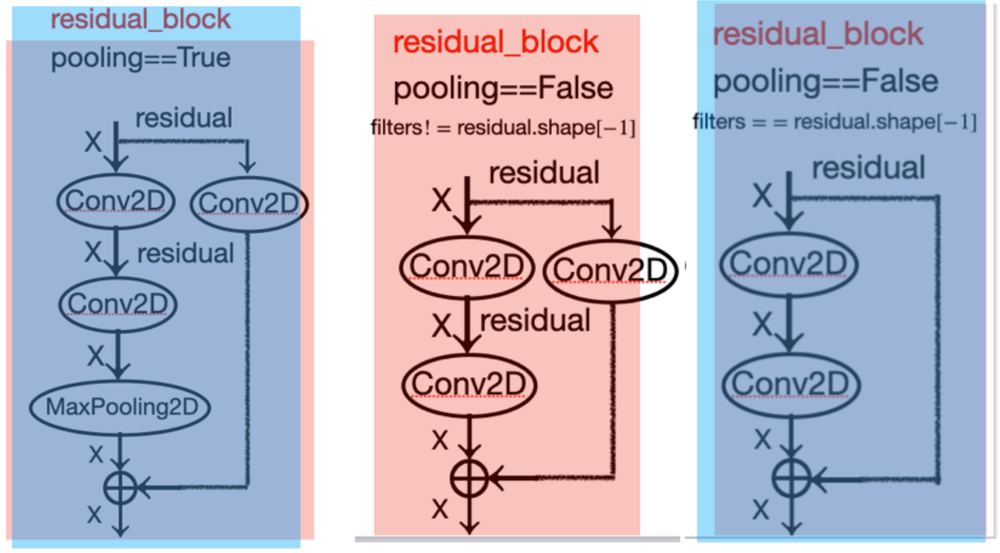
First block

If we don't use max pooling, we only project the residual if the number of channels has changed.

Second block; note the increasing filter count in each block.

The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.





Residual Connections: example of a simple CNN

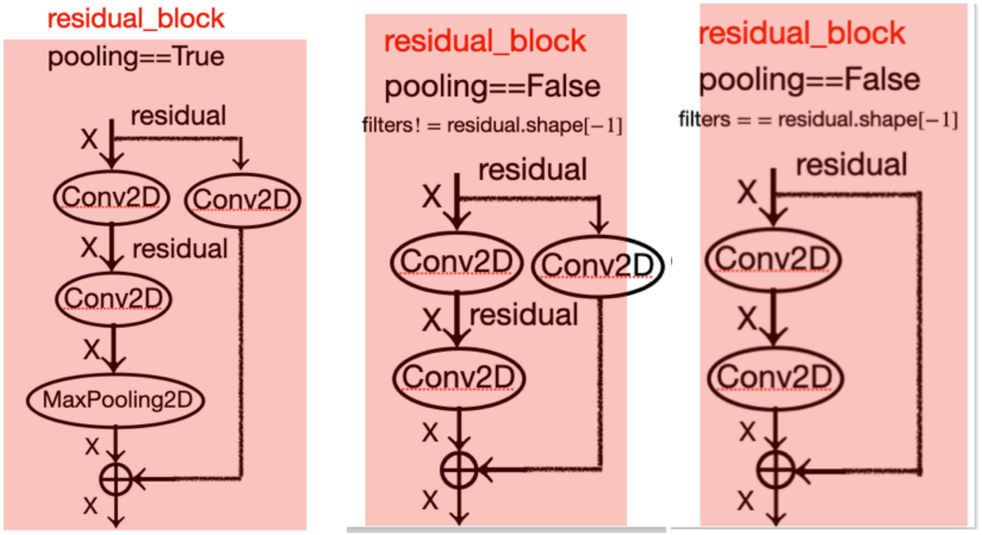
```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

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

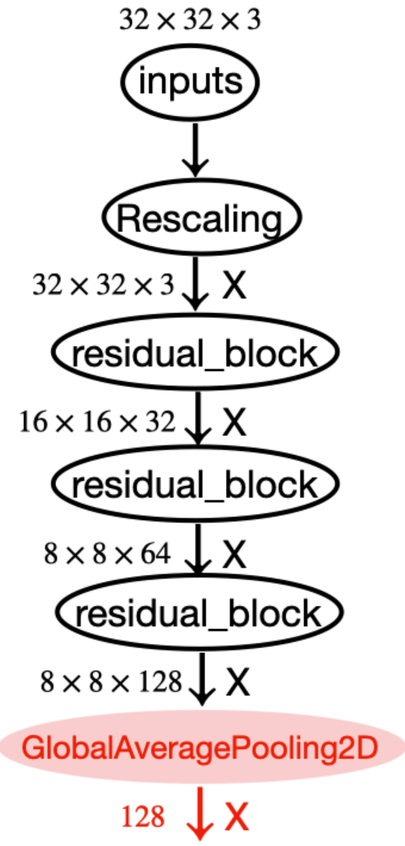
First block

If we don't use max pooling, we only project the residual if the number of channels has changed.

Second block; note the increasing filter count in each block.

The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.





Residual Connections: example of a simple CNN

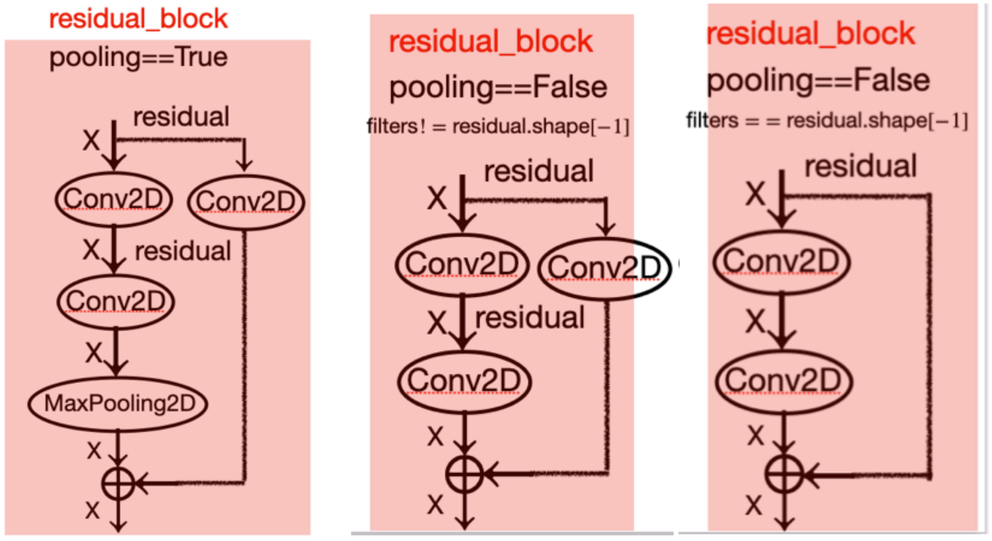
```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

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

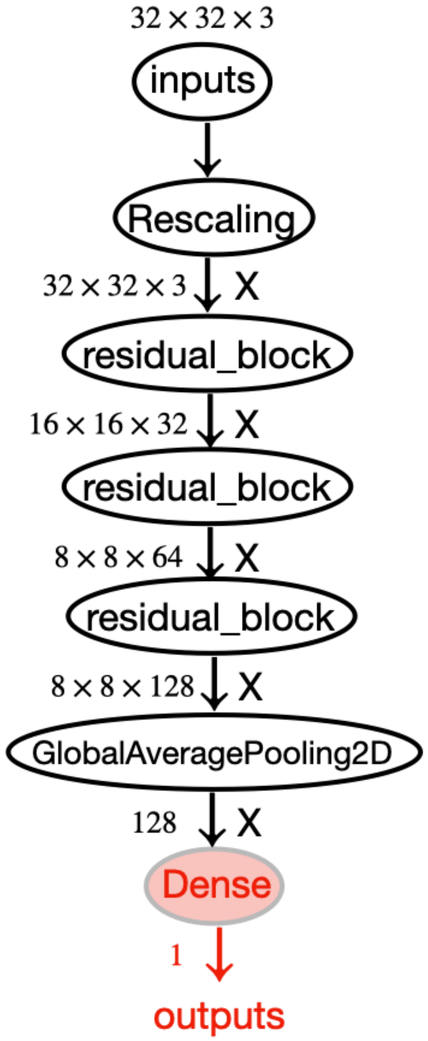
First block

If we don't use max pooling, we only project the residual if the number of channels has changed.

Second block; note the increasing filter count in each block.

The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.





Residual Connections: example of a simple CNN

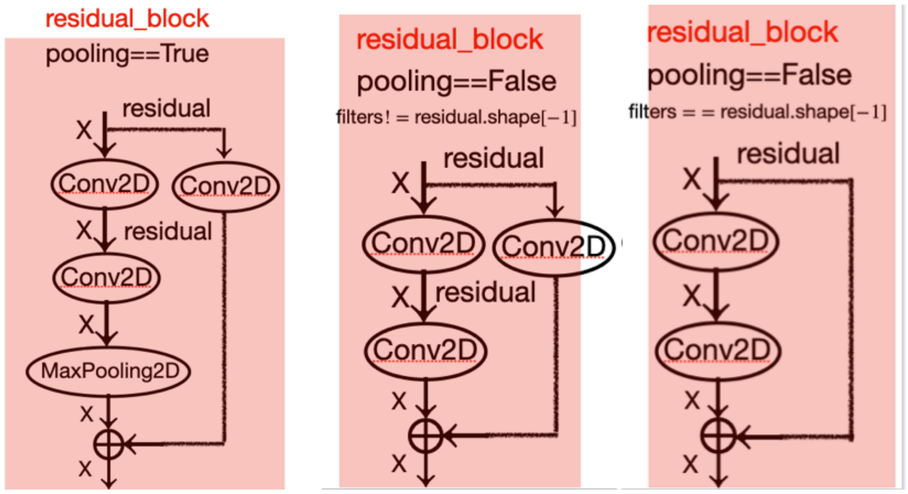
```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Rescaling(1./255)(inputs)

x = residual_block(x, filters=32, pooling=True)
x = residual_block(x, filters=64, pooling=True)
x = residual_block(x, filters=128, pooling=False)

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

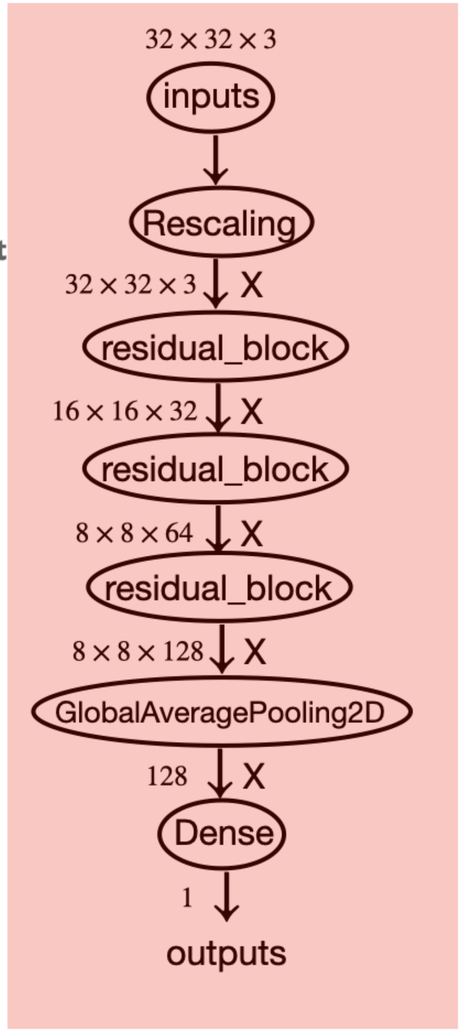
First block

If we don't use max pooling, we only project the residual if the number of channels has changed.

Second block; note the increasing filter count in each block.

The last block doesn't need a max pooling layer, since we will apply global average pooling right after it.





model

Quiz questions:

1. Why are residual connections important for CNNs?

2. How to build a network with residual connections?

Roadmap of this lecture:

1. Image segmentation.
2. CNN architecture patterns
    - 2.1 Residual connection
    - 2.2 Batch normalization and depthwise separable convolution
    - 2.3 Put them together for an Xception-like model
3. Visualize and interpret what CNN learns
    - 3.1 Visualize intermediate activations
    - 3.2 Visualize convolution filters
    - 3.3 Visualize heatmaps of class activation

Batch Normalization

Can we normalize data for intermediate layers (not just input data)?

Benefit: help training, and generalize to data with different distributions.

Batch normalization does just that. It's a type of layer (`BatchNormalization` in Keras) introduced in 2015 by Ioffe and Szegedy;[2] it can 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, and 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.

Batch Normalization

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.

Batch Normalization

Importantly, I would generally recommend placing the previous layer's activation *after* the batch normalization layer (although this is still a subject of debate). So instead of doing what is shown in listing 9.4, you would do what's shown in listing 9.5.

Listing 9.4 How not to use batch normalization

```
x = layers.Conv2D(32, 3, activation="relu")(x)
x = layers.BatchNormalization()(x)
```

Listing 9.5 How to use batch normalization: the activation comes last

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

Note the lack of activation here.

We place the activation after the BatchNormalization layer.

Batch Normalization

On batch normalization and fine-tuning

Batch normalization has many quirks. One of the main ones relates to fine-tuning: when fine-tuning a model that includes `BatchNormalization` layers, I recommend leaving these layers frozen (set their `trainable` attribute to `False`). Otherwise they will keep updating their internal mean and variance, which can interfere with the very small updates applied to the surrounding `Conv2D` layers.

Depthwise Separable Convolution

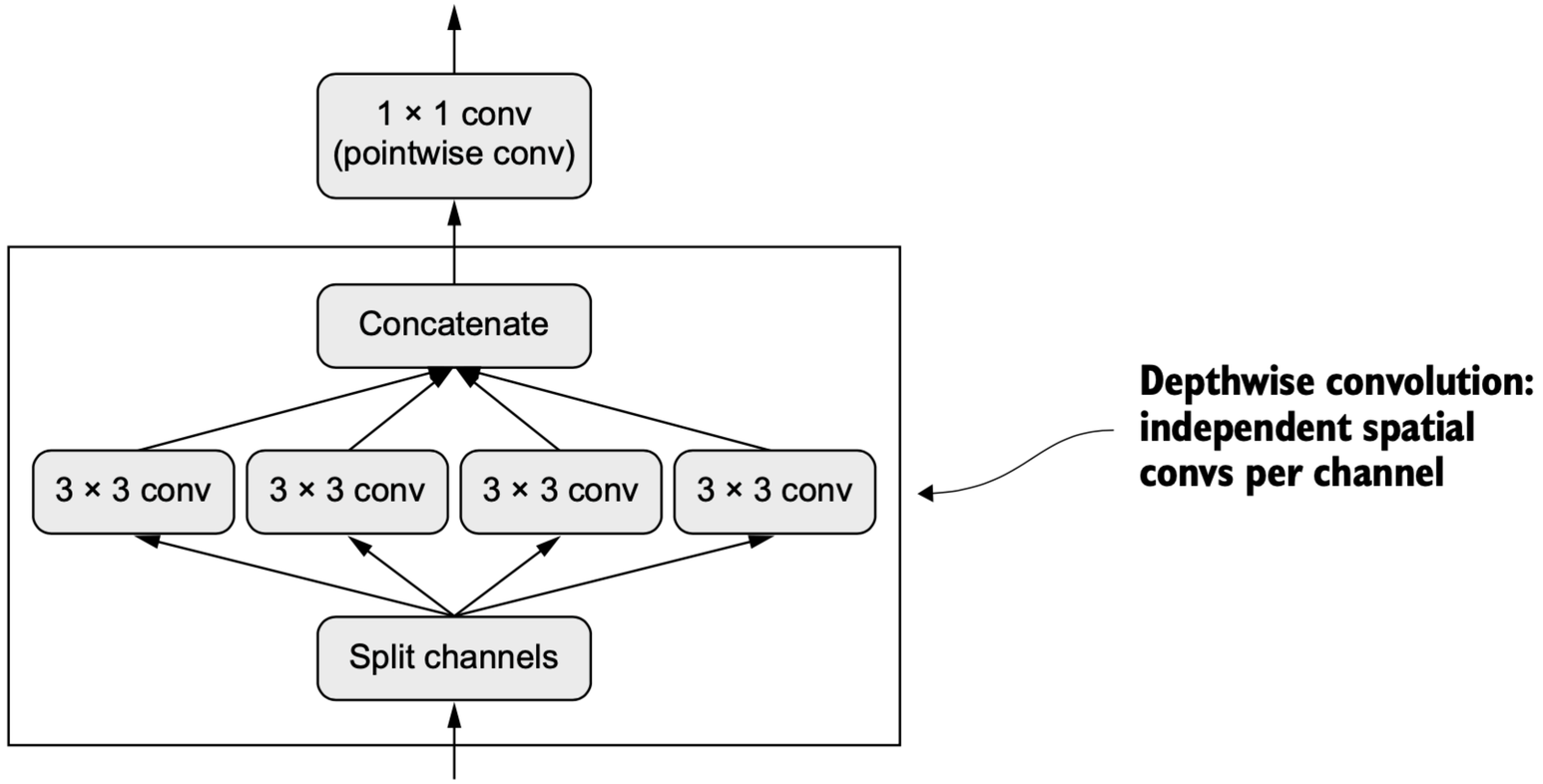


Figure 9.10 Depthwise separable convolution: a depthwise convolution followed by a pointwise convolution

Quiz questions:

1. What is batch normalization? Why is it important?

2. How is depthwise separable convolution different from ordinary convolution?

Roadmap of this lecture:

1. Image segmentation.
2. CNN architecture patterns
    - 2.1 Residual connection
    - 2.2 Batch normalization and depthwise separable convolution
    - 2.3 Put them together for an Xception-like model
3. Visualize and interpret what CNN learns
    - 3.1 Visualize intermediate activations
    - 3.2 Visualize convolution filters
    - 3.3 Visualize heatmaps of class activation

Putting it together: A mini Xception-like model

As a reminder, here are the convnet architecture principles you've learned so far:

- Your model should be organized into repeated *blocks* of layers, usually made of multiple convolution layers and a max pooling layer.
- The number of filters in your layers should increase as the size of the spatial feature maps decreases.
- Deep and narrow is better than broad and shallow.
- Introducing residual connections around blocks of layers helps you train deeper networks.
- It can be beneficial to introduce batch normalization layers after your convolution layers.
- It can be beneficial to replace `Conv2D` layers with `SeparableConv2D` layers, which are more parameter-efficient.

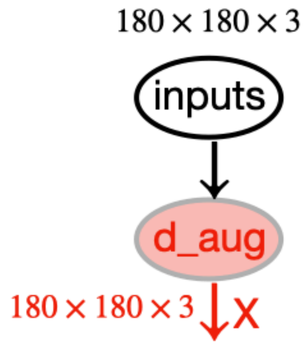
Let's bring these ideas together into a single model. Its architecture will resemble a smaller version of Xception, and we'll apply it to the dogs vs. cats task from the last chapter. For data loading and model training, we'll simply reuse the setup we used in section 8.2.5, but we'll replace the model definition with the following convnet:


inputs = keras.Input(shape=(180, 180, 3))
x = data_augmentation(inputs)
We use the same data augmentation configuration as before.
Don't forget input rescaling!
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)
In the original model, we used a Flatten layer before the Dense layer. Here, we go with a GlobalAveragePooling2D layer.
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
Like in the original model, we add a dropout layer for regularization.
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.
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.
180 × 180 × 3
inputs

```
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)
```

Don't forget input rescaling!

We use the same data augmentation configuration as before.



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.

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.

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.

```
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)
```

Don't forget input rescaling!

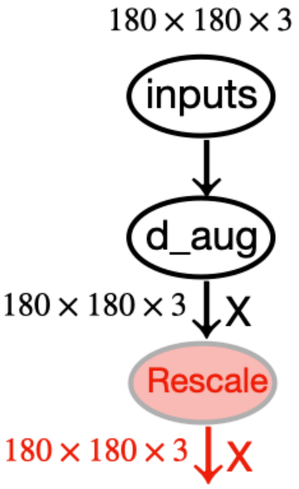
We use the same data augmentation configuration as before.

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.

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.

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.



```
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.

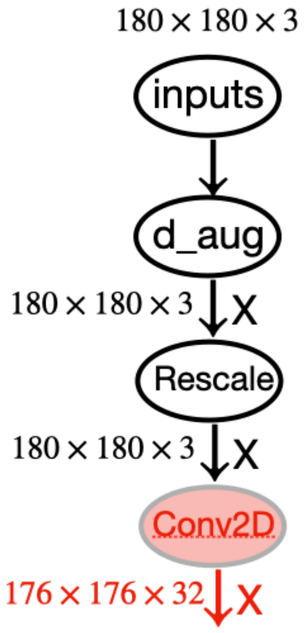
Don't forget input rescaling!

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.

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.

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.



```
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)
```

Don't forget input rescaling!

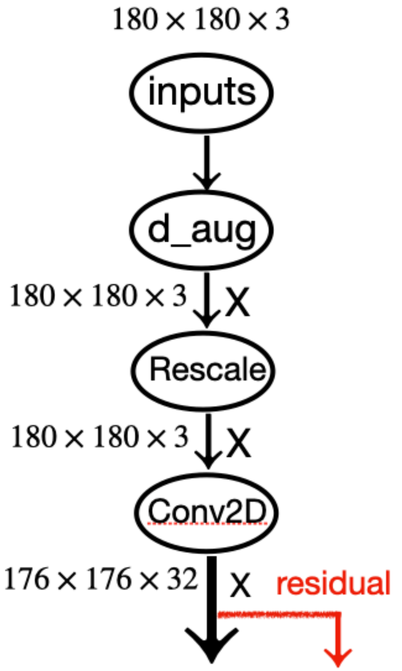
We use the same data augmentation configuration as before.

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.

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.

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.



```
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)
```

Don't forget input rescaling!

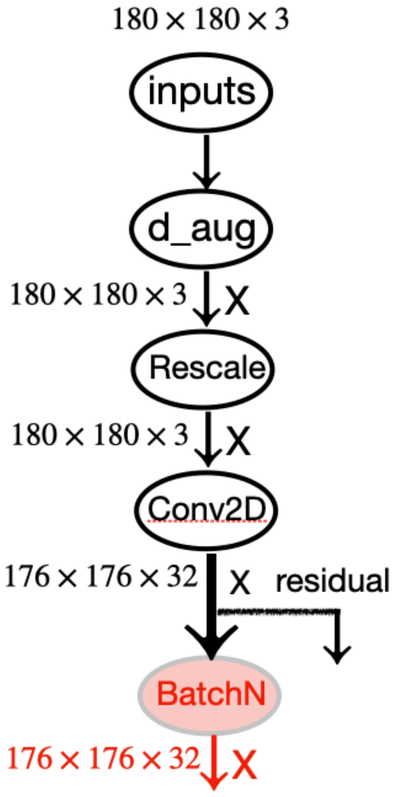
We use the same data augmentation configuration as before.

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.

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.

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.



```
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)
```

Don't forget input rescaling!

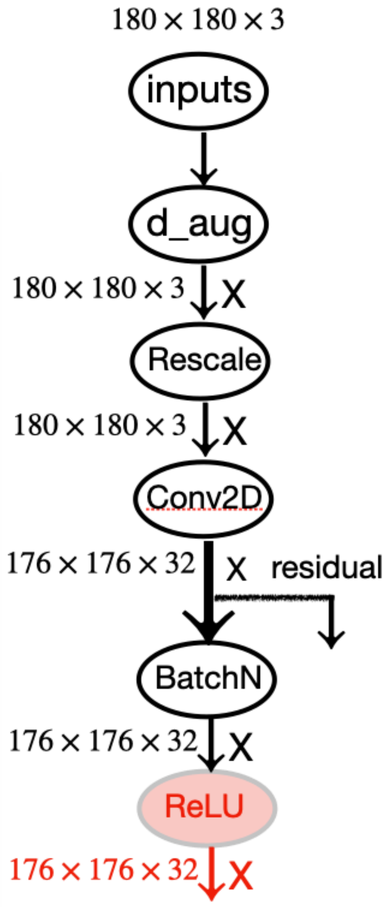
We use the same data augmentation configuration as before.

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.

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.

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.



```
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)
```

Don't forget input rescaling!

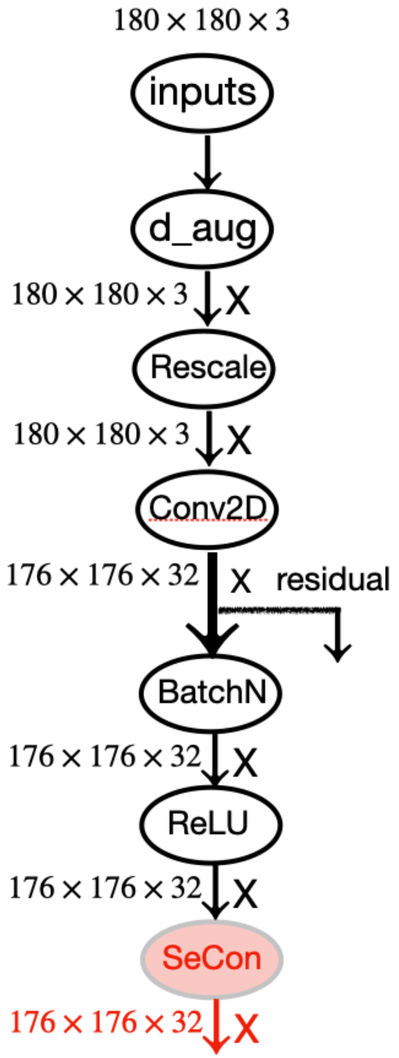
We use the same data augmentation configuration as before.

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.

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.

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.



```
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)
```

Don't forget input rescaling!

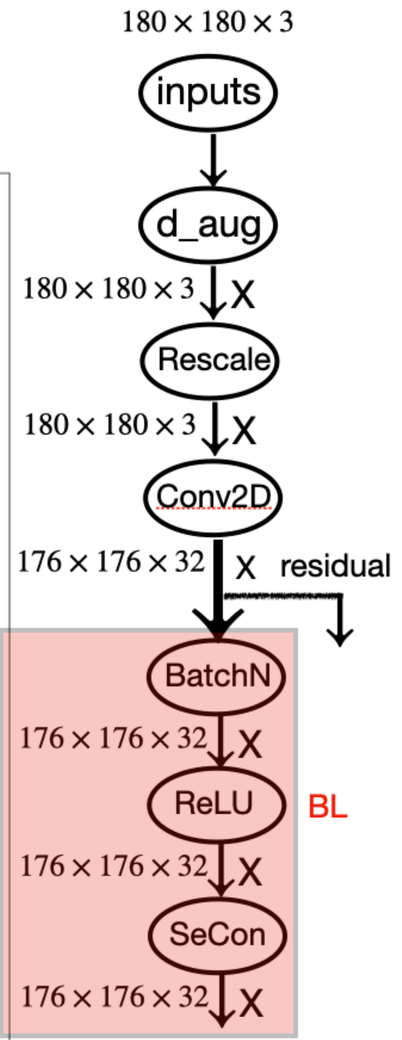
We use the same data augmentation configuration as before.

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.

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.

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.



```
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)
```

Don't forget input rescaling!

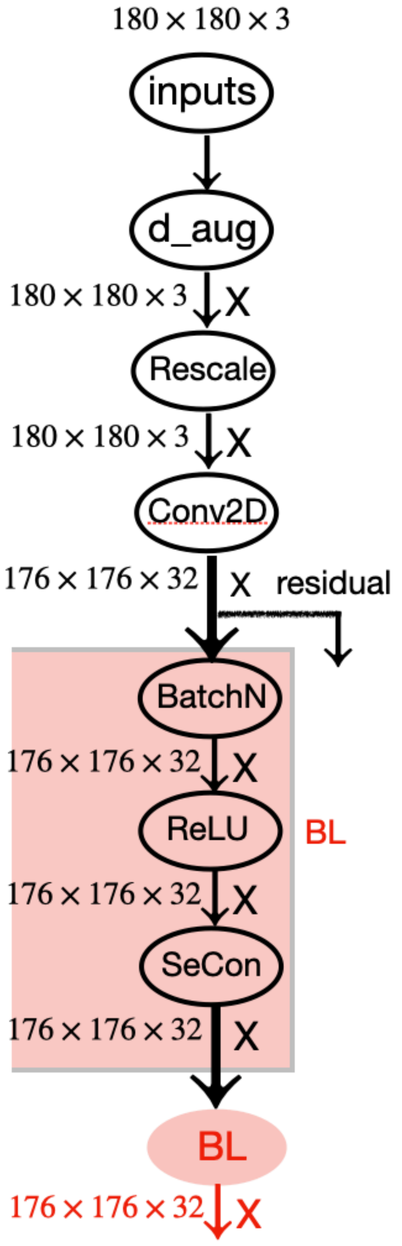
We use the same data augmentation configuration as before.

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.

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.

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.



```
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)
```

Don't forget input rescaling!

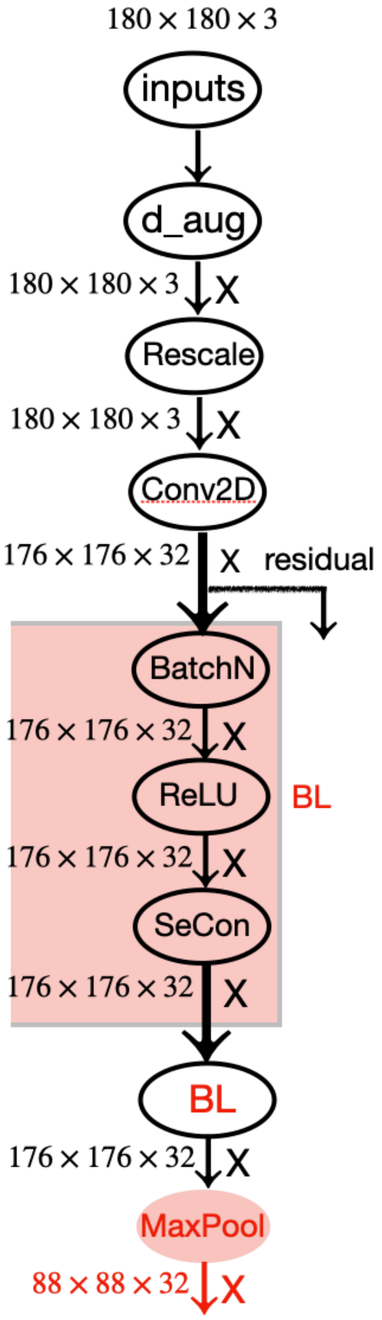
We use the same data augmentation configuration as before.

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.

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.

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.



```
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)
```

Don't forget input rescaling!

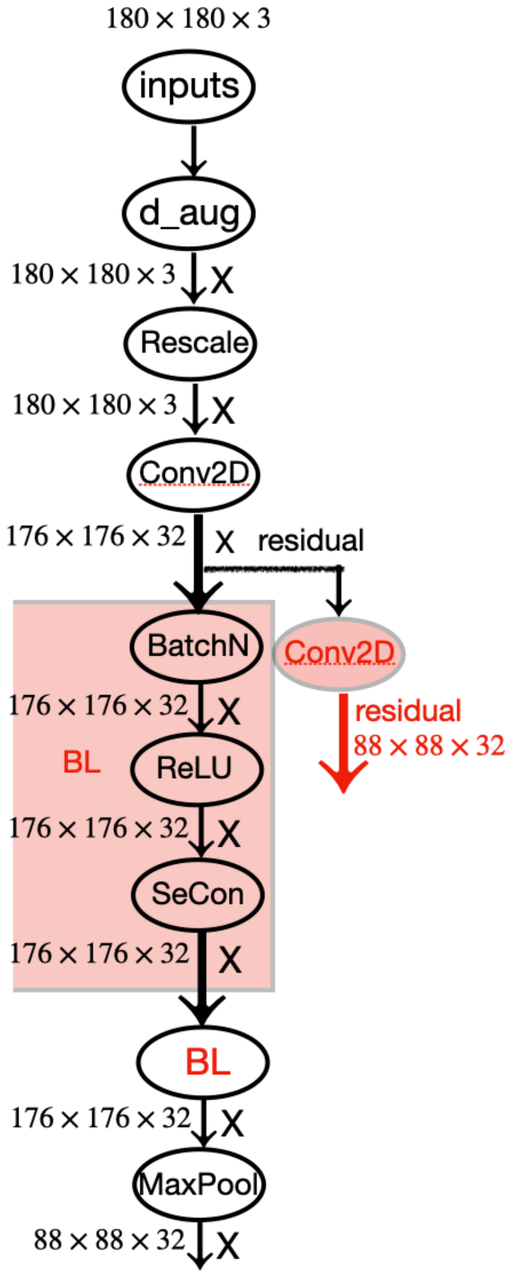
We use the same data augmentation configuration as before.

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.

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.

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.



```
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)
```

Don't forget input rescaling!

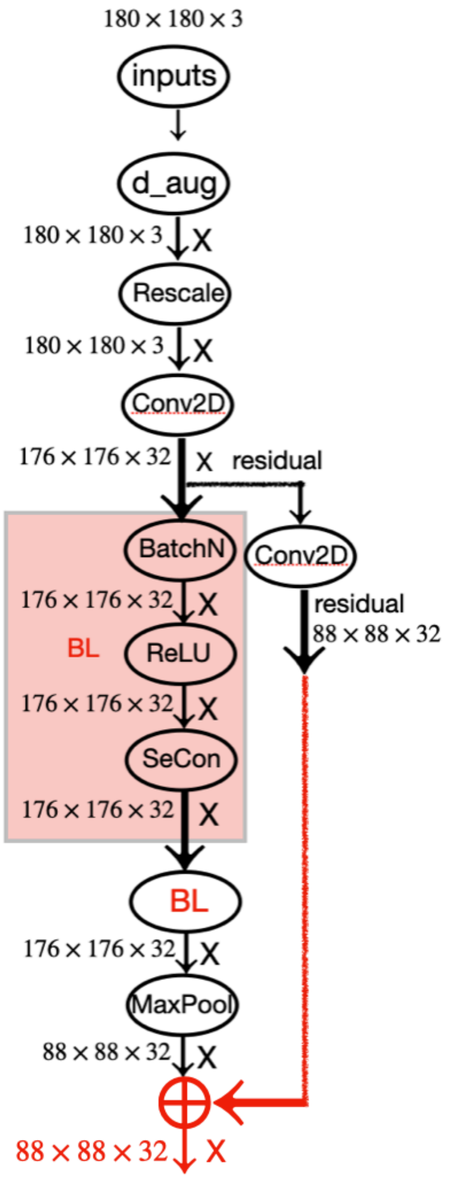
We use the same data augmentation configuration as before.

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.

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.

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.



```
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)
```

Don't forget input rescaling!

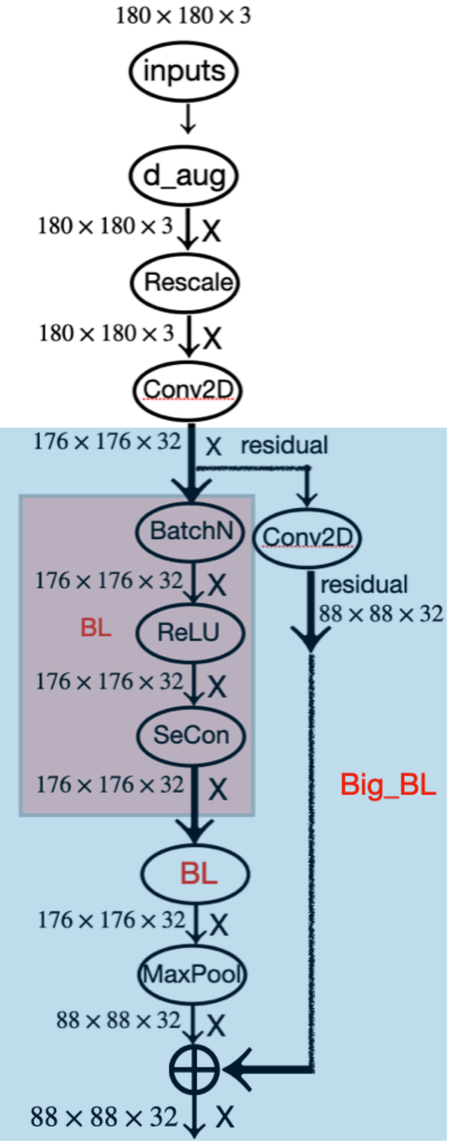
We use the same data augmentation configuration as before.

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.

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.

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.



```
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)
```

Don't forget input rescaling!

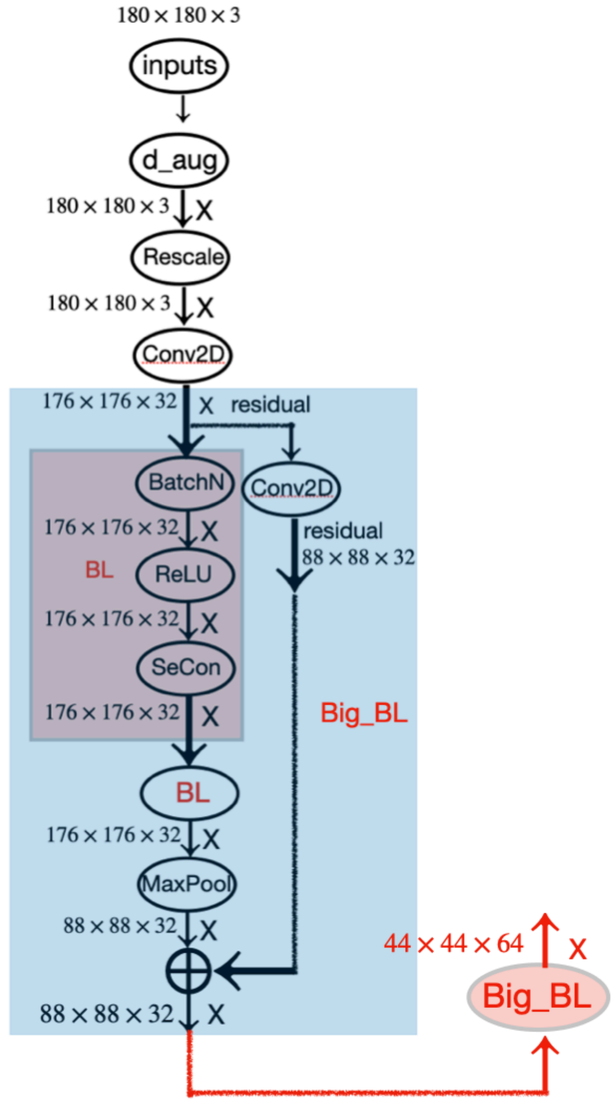
We use the same data augmentation configuration as before.

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.

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.

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.



```
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)
```

Don't forget input rescaling!

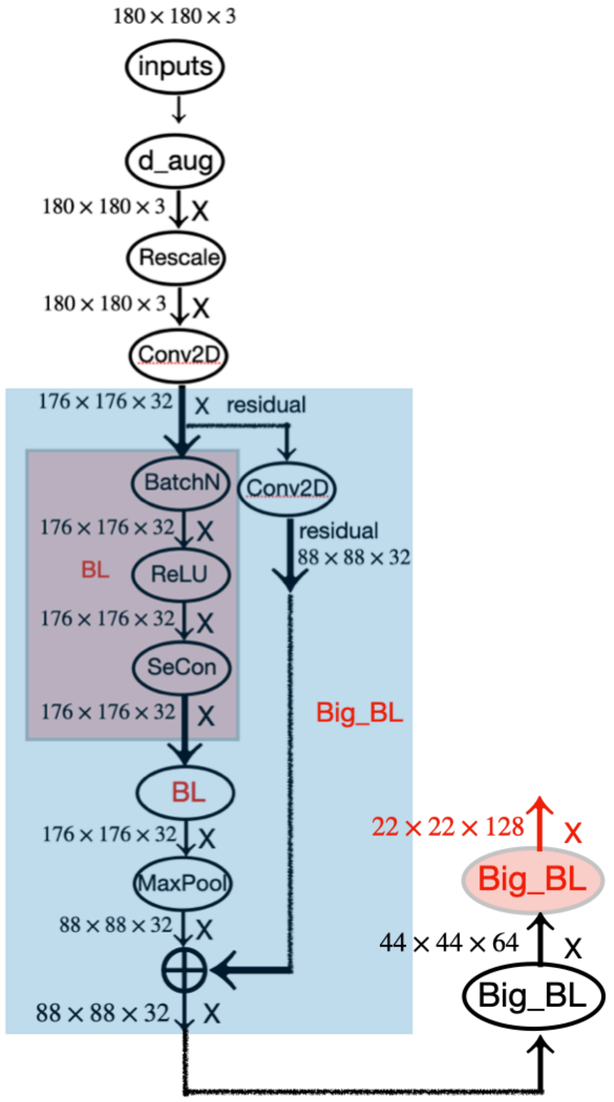
We use the same data augmentation configuration as before.

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.

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.

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.



```
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)
```

Don't forget input rescaling!

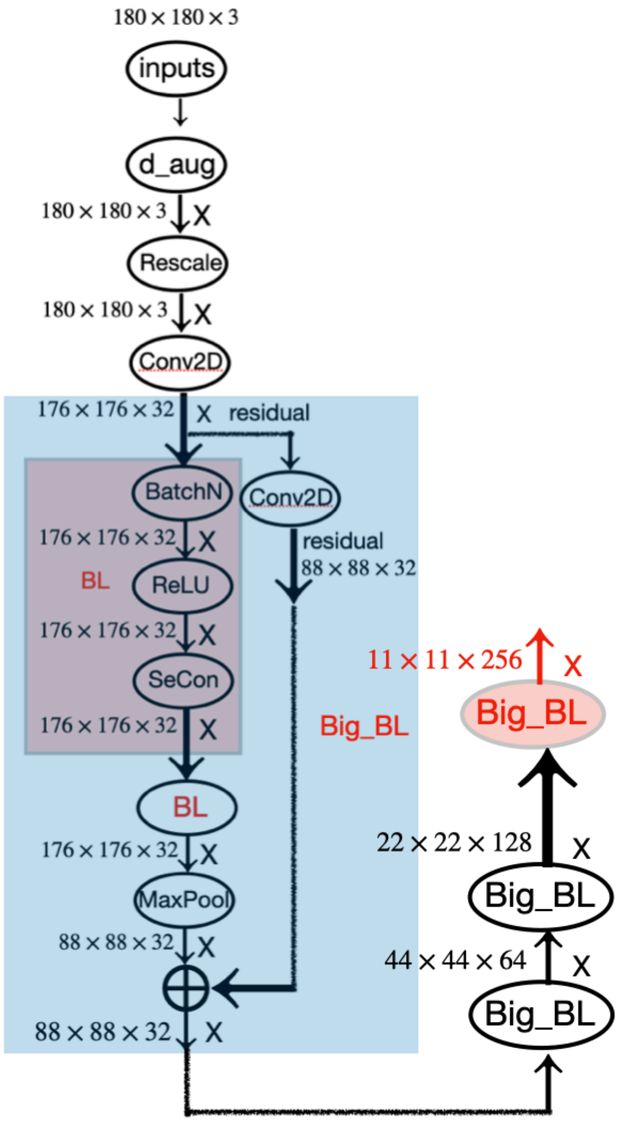
We use the same data augmentation configuration as before.

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.

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.

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.



```
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)
```

Don't forget input rescaling!

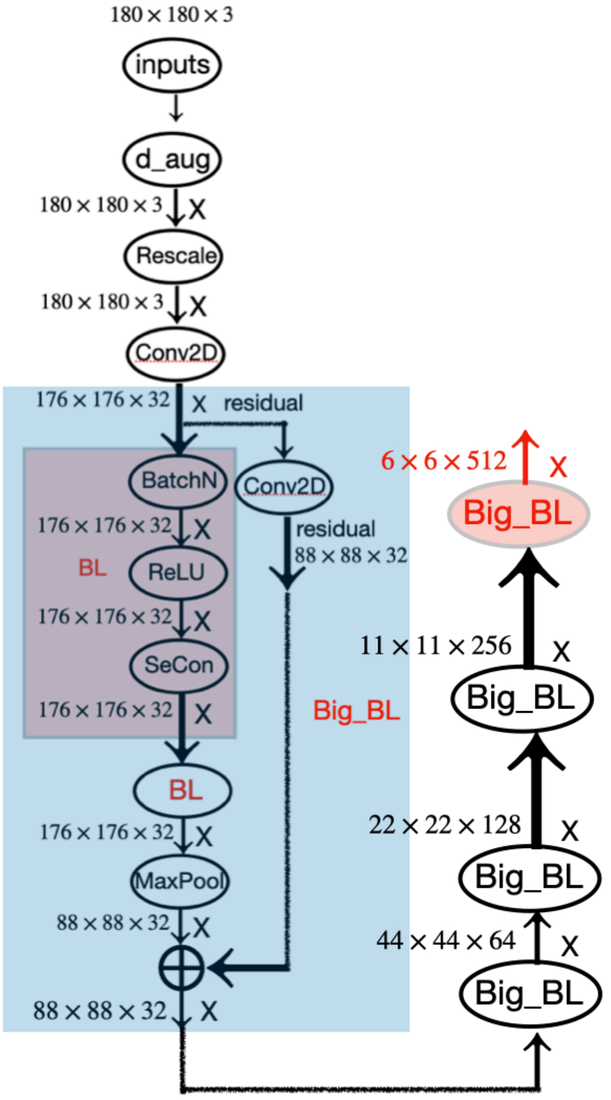
We use the same data augmentation configuration as before.

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.

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.

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.



```
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)
```

Don't forget input rescaling!

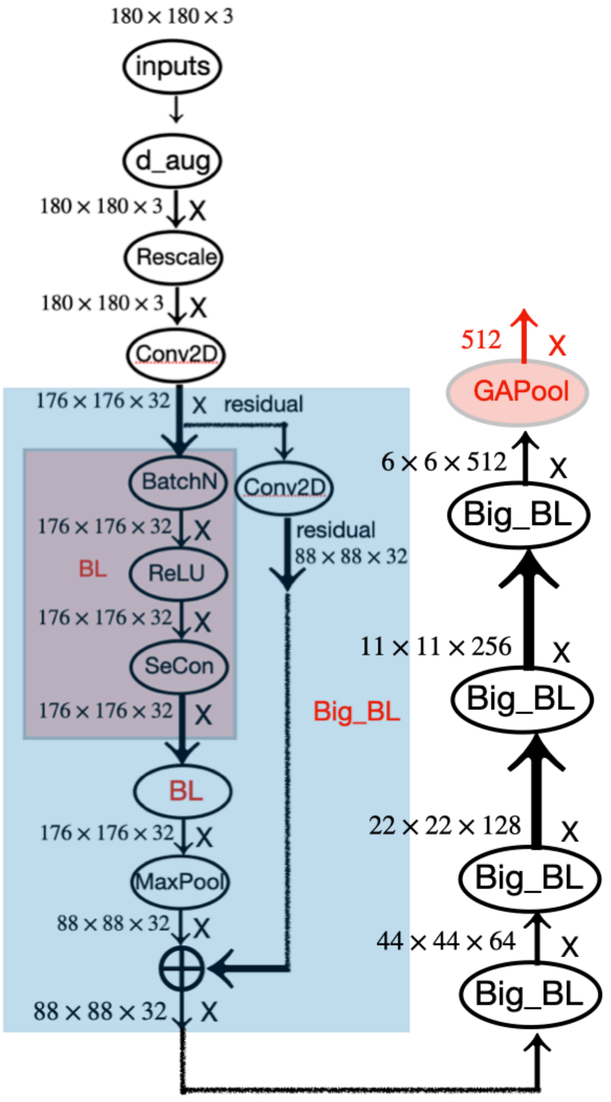
We use the same data augmentation configuration as before.

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.

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.

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.



```
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)
```

Don't forget input rescaling!

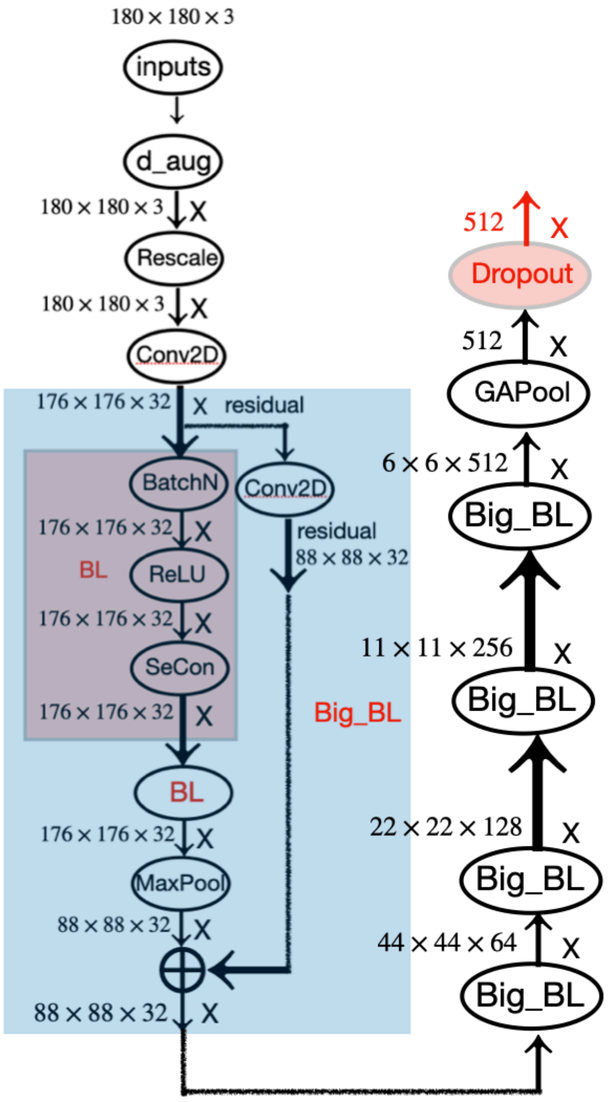
We use the same data augmentation configuration as before.

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.

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.

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.



```
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)
```

Don't forget input rescaling!

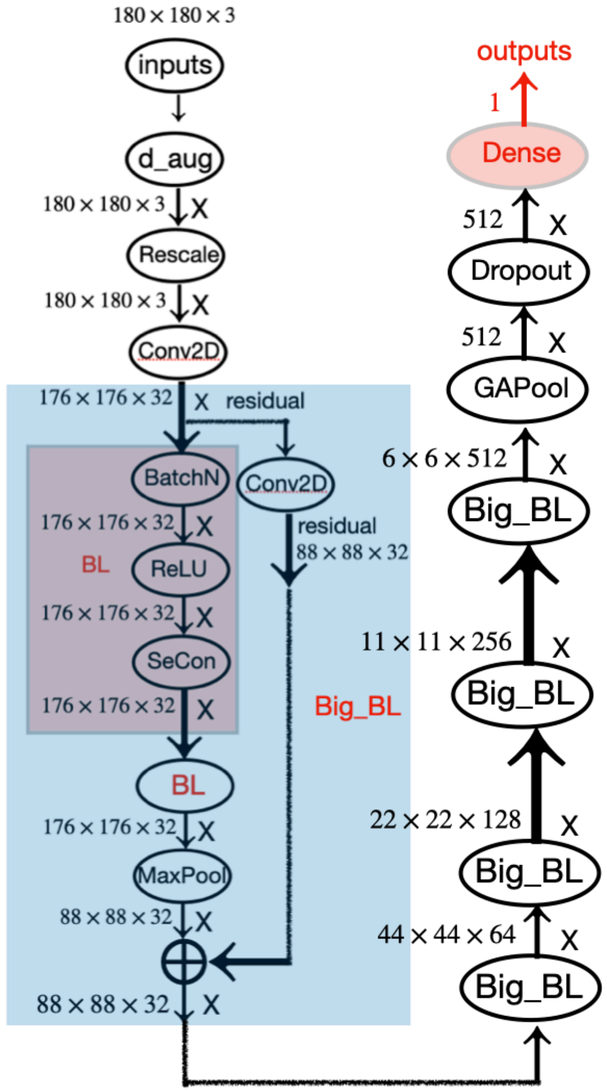
We use the same data augmentation configuration as before.

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.

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.

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.



Putting it together: A mini Xception-like model

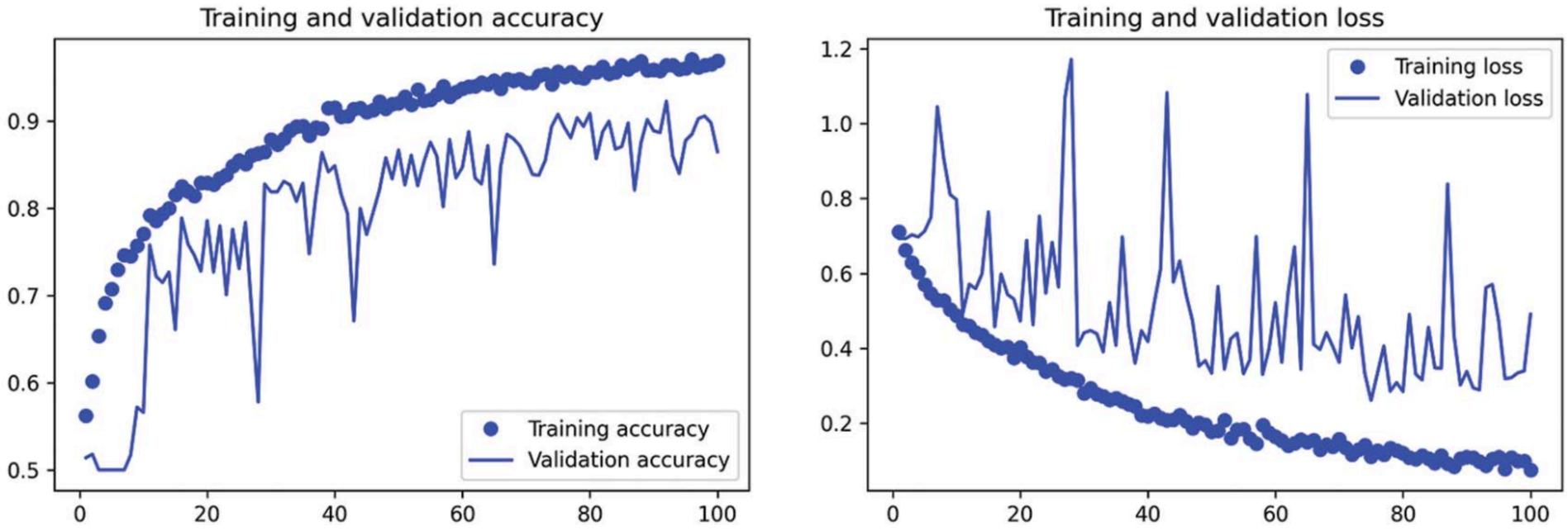


Figure 9.11 Training and validation metrics with an Xception-like architecture

Quiz question:

1. How to use modules to build a complex network?

Roadmap of this lecture:

1. Image segmentation.
2. CNN architecture patterns
    - 2.1 Residual connection
    - 2.2 Batch normalization and depthwise separable convolution
    - 2.3 Put them together for an Xception-like model
3. Visualize and interpret what CNN learns
    - 3.1 Visualize intermediate activations
    - 3.2 Visualize convolution filters
    - 3.3 Visualize heatmaps of class activation

Interpret what CNNs Learn

- *Visualizing intermediate convnet outputs (intermediate activations)*—Useful for understanding how successive convnet layers transform their input, and for getting a first idea of the meaning of individual convnet filters
- *Visualizing convnet filters*—Useful for understanding precisely what visual pattern or concept each filter in a convnet is receptive to
- *Visualizing heatmaps of class activation in an image*—Useful for understanding which parts of an image were identified as belonging to a given class, thus allowing you to localize objects in images

Visualizing Intermediate Activations

Visualizing intermediate activations consists of displaying the values returned by various convolution and pooling layers in a model, given a certain input (the output of a layer is often called its *activation*, the output of the activation function). This gives a view into how an input is decomposed into the different filters learned by the network. We want to visualize feature maps with three dimensions: width, height, and depth (channels). Each channel encodes relatively independent features, so the proper way to visualize these feature maps is by independently plotting the contents of every channel as a 2D image. Let's start by loading the model that you saved in section 8.2:

```
>>> from tensorflow import keras
>>> model = keras.models.load_model(
    "convnet_from_scratch_with_augmentation.keras")
>>> model.summary()
```

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 180, 180, 3)]	0
sequential (Sequential)	(None, 180, 180, 3)	0
rescaling_1 (Rescaling)	(None, 180, 180, 3)	0
conv2d_5 (Conv2D)	(None, 178, 178, 32)	896
max_pooling2d_4 (MaxPooling2	(None, 89, 89, 32)	0
conv2d_6 (Conv2D)	(None, 87, 87, 64)	18496
max_pooling2d_5 (MaxPooling2	(None, 43, 43, 64)	0
conv2d_7 (Conv2D)	(None, 41, 41, 128)	73856
max_pooling2d_6 (MaxPooling2	(None, 20, 20, 128)	0
conv2d_8 (Conv2D)	(None, 18, 18, 256)	295168
max_pooling2d_7 (MaxPooling2	(None, 9, 9, 256)	0
conv2d_9 (Conv2D)	(None, 7, 7, 256)	590080
flatten_1 (Flatten)	(None, 12544)	0
dropout (Dropout)	(None, 12544)	0
dense_1 (Dense)	(None, 1)	12545

Total params: 991,041
Trainable params: 991,041
Non-trainable params: 0

Listing 9.6 Preprocessing a single image

```
from tensorflow import keras
import numpy as np

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)
    array = np.expand_dims(array, axis=0)
    return array

img_tensor = get_img_array(img_path, target_size=(180, 180))
```

Download a test image.

Open the image file and resize it.

Turn the image into a float32 NumPy array of shape (180, 180, 3).

Add a dimension to transform the array into a "batch" of a single sample. Its shape is now (1, 180, 180, 3).

Let's display the picture (see figure 9.12).

Listing 9.7 Displaying the test picture

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

Figure 9.12

Visualizing Intermediate Activations

In order to extract the feature maps we want to look at, we'll create a Keras model that takes batches of images as input, and that outputs the activations of all convolution and pooling layers.

Listing 9.8 Instantiating a model that returns layer activations

```
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)
```

Extract the outputs of all Conv2D and MaxPooling2D layers and put them in a list.

Save the layer names for later.

Create a model that will return these outputs, given the model input.

When fed an image input, this model returns the values of the layer activations in the original model, as a list. This is the first time you've encountered a multi-output model in this book in practice since you learned about them in chapter 7; until now, the models you've seen have had exactly one input and one output. This one has one input and nine outputs: one output per layer activation.

Listing 9.9 Using the model to compute layer activations

```
activations = activation_model.predict(img_tensor)
```

Return a list of nine NumPy arrays: one array per layer activation.

For instance, this is the activation of the first convolution layer for the cat image input:

```
>>> first_layer_activation = activations[0]
>>> print(first_layer_activation.shape)
(1, 178, 178, 32)
```

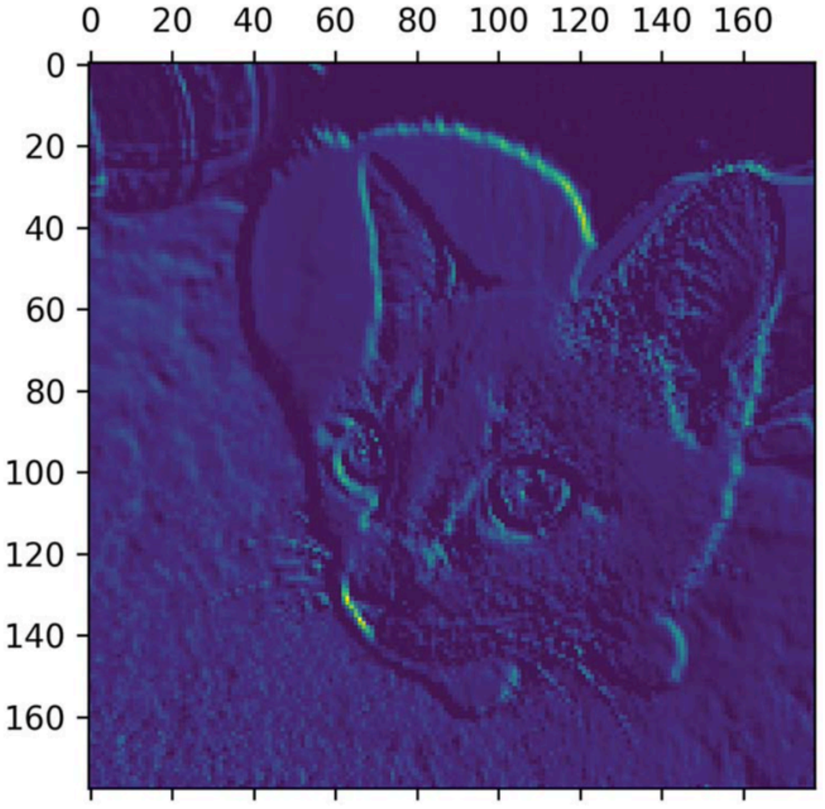
Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 180, 180, 3)]	0
sequential (Sequential)	(None, 180, 180, 3)	0
rescaling_1 (Rescaling)	(None, 180, 180, 3)	0
conv2d_5 (Conv2D)	(None, 178, 178, 32)	896
max_pooling2d_4 (MaxPooling2	(None, 89, 89, 32)	0
conv2d_6 (Conv2D)	(None, 87, 87, 64)	18496
max_pooling2d_5 (MaxPooling2	(None, 43, 43, 64)	0
conv2d_7 (Conv2D)	(None, 41, 41, 128)	73856
max_pooling2d_6 (MaxPooling2	(None, 20, 20, 128)	0
conv2d_8 (Conv2D)	(None, 18, 18, 256)	295168
max_pooling2d_7 (MaxPooling2	(None, 9, 9, 256)	0
conv2d_9 (Conv2D)	(None, 7, 7, 256)	590080
flatten_1 (Flatten)	(None, 12544)	0
dropout (Dropout)	(None, 12544)	0
dense_1 (Dense)	(None, 1)	12545

```
Total params: 991,041
Trainable params: 991,041
Non-trainable params: 0
```

It's a 178 × 178 feature map with 32 channels. Let's try plotting the fifth channel of the activation of the first layer of the original model (see figure 9.13).

Listing 9.10 Visualizing the fifth channel

```
import matplotlib.pyplot as plt
plt.matshow(first_layer_activation[0, :, :, 5], cmap="viridis")
```



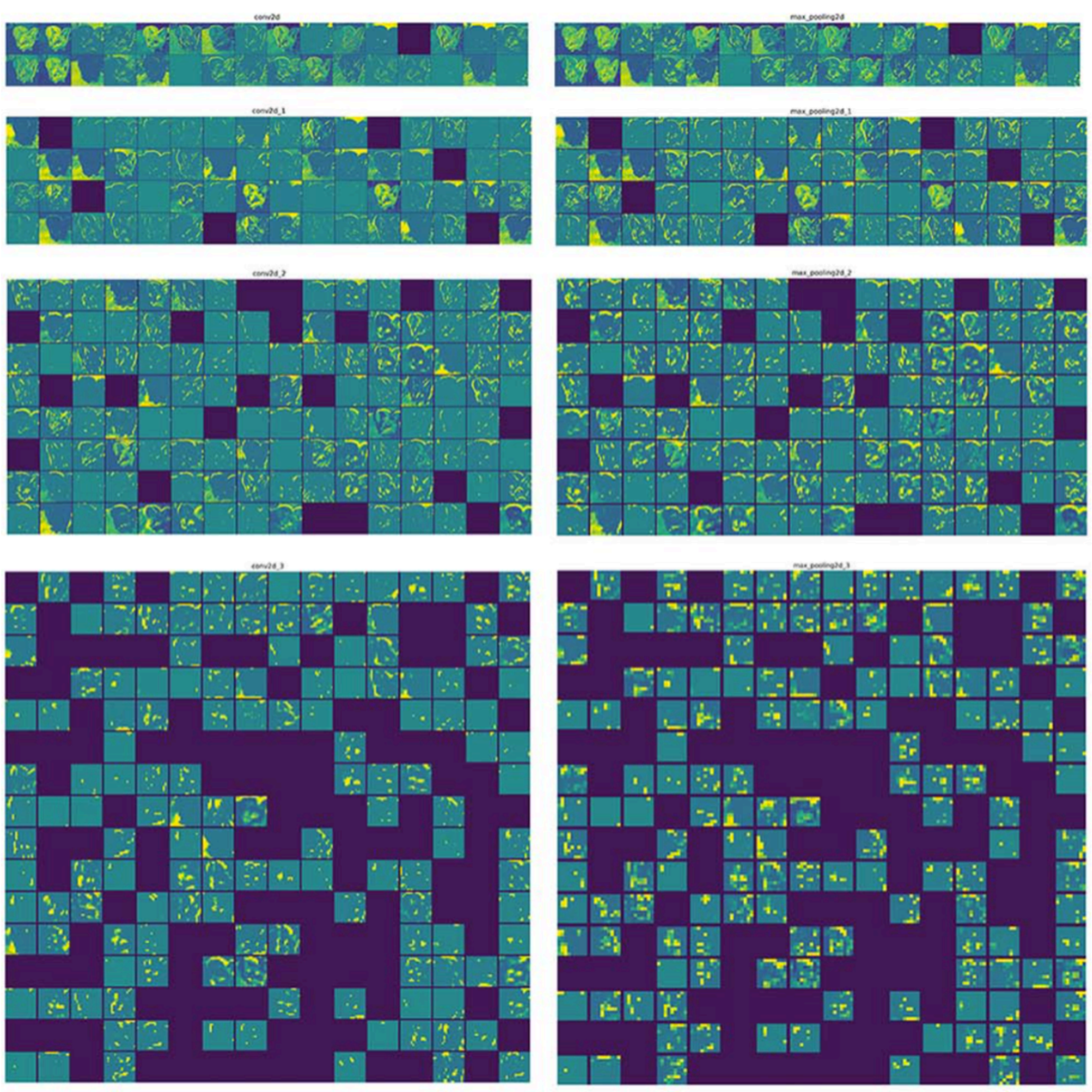
This channel appears to encode a diagonal edge detector.

Figure 9.13 Fifth channel of the activation of the first layer on the test cat picture

Visualizing Intermediate Activations

Let’s plot a complete visualization of all the activations in the network. We will extract and plot every channel in each of the layer activations, and we’ll stack the results in one big grid, with channels stacked side by side.

Visualizing Intermediate Activations



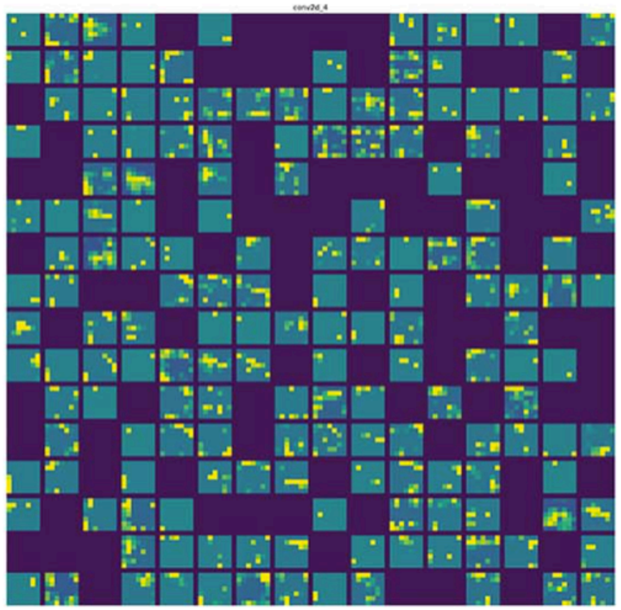
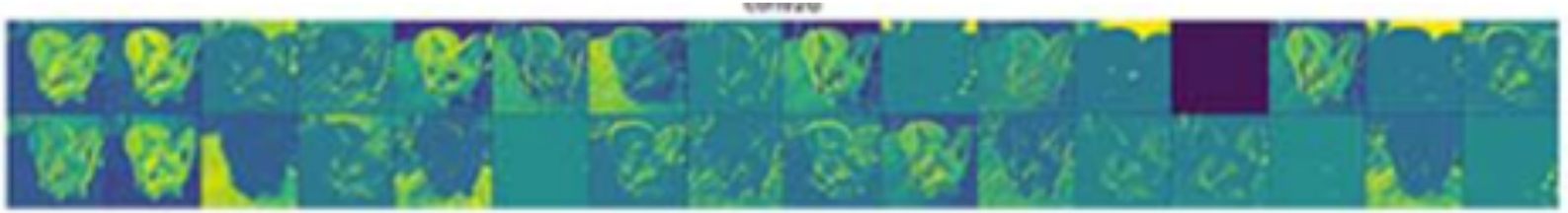
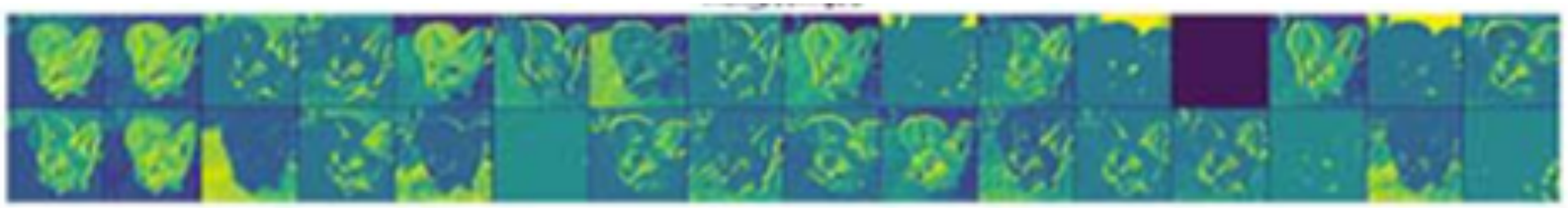
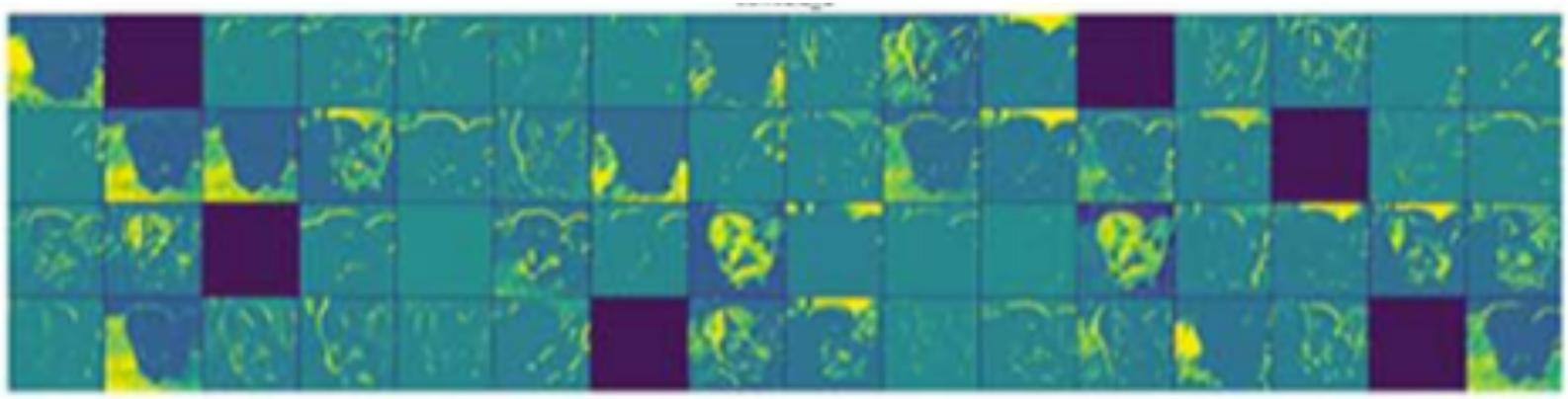
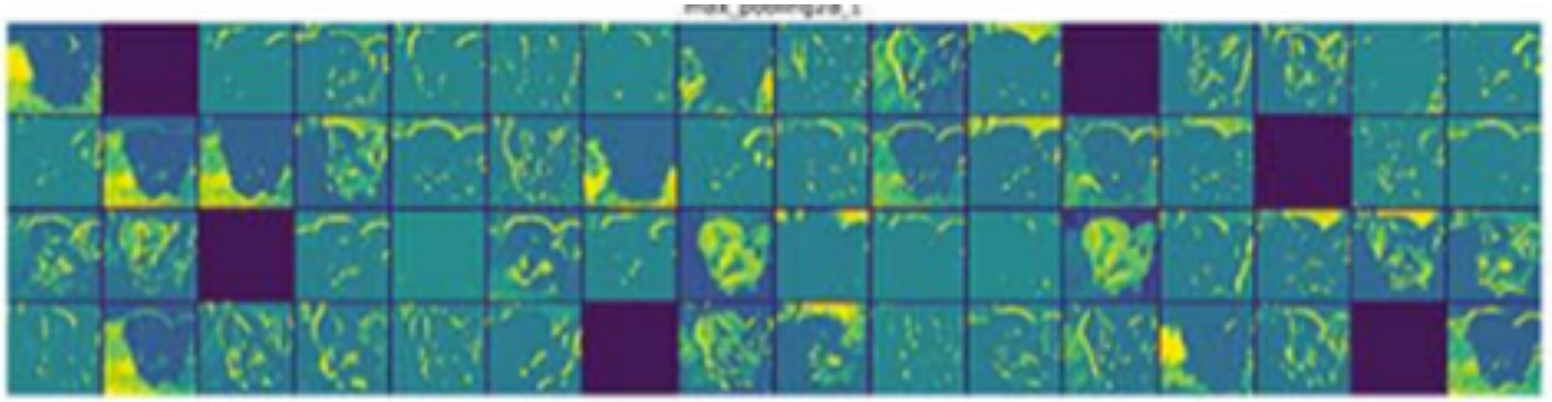


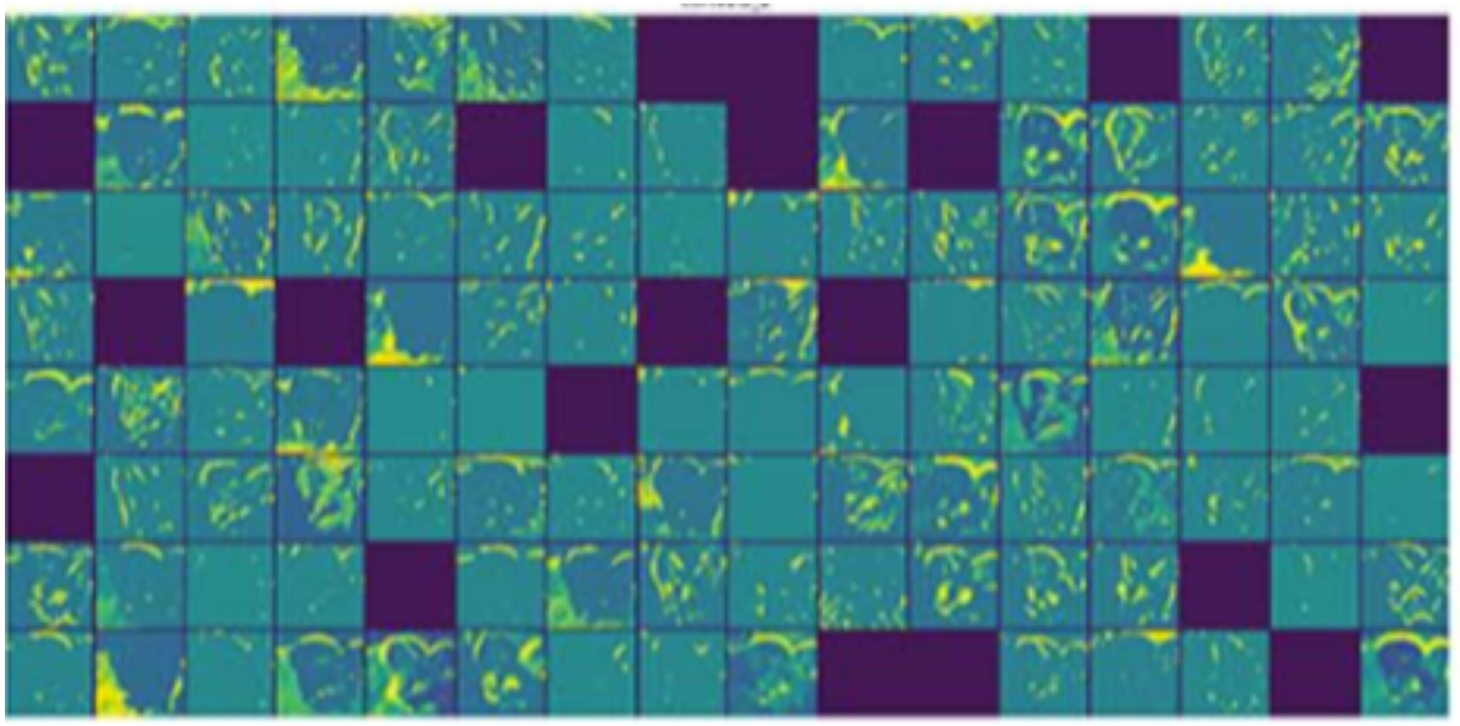
Figure 9.14 Every channel of every layer activation on the test cat picture

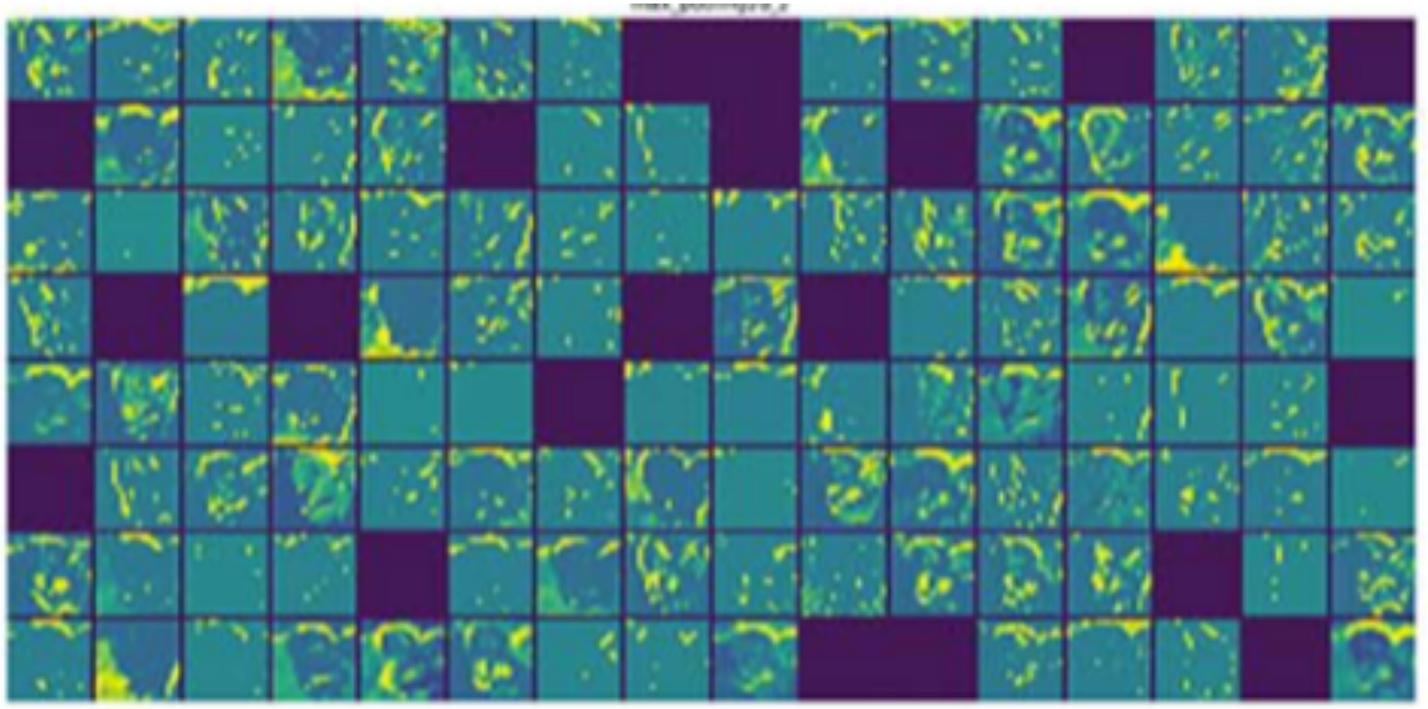


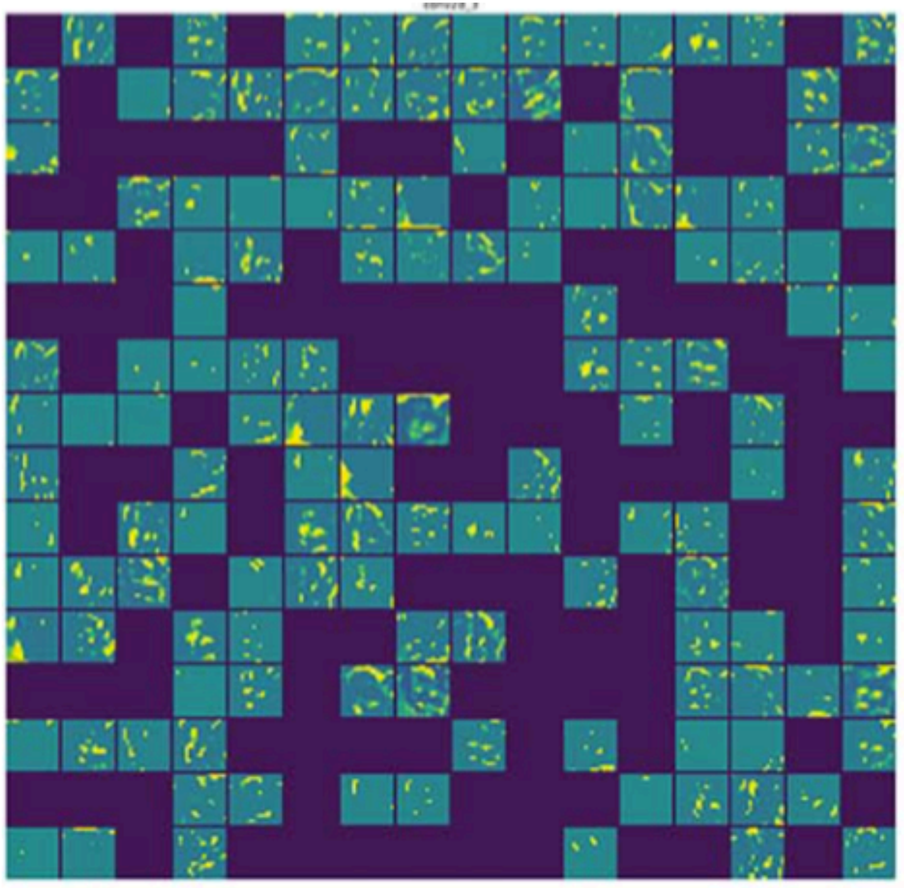


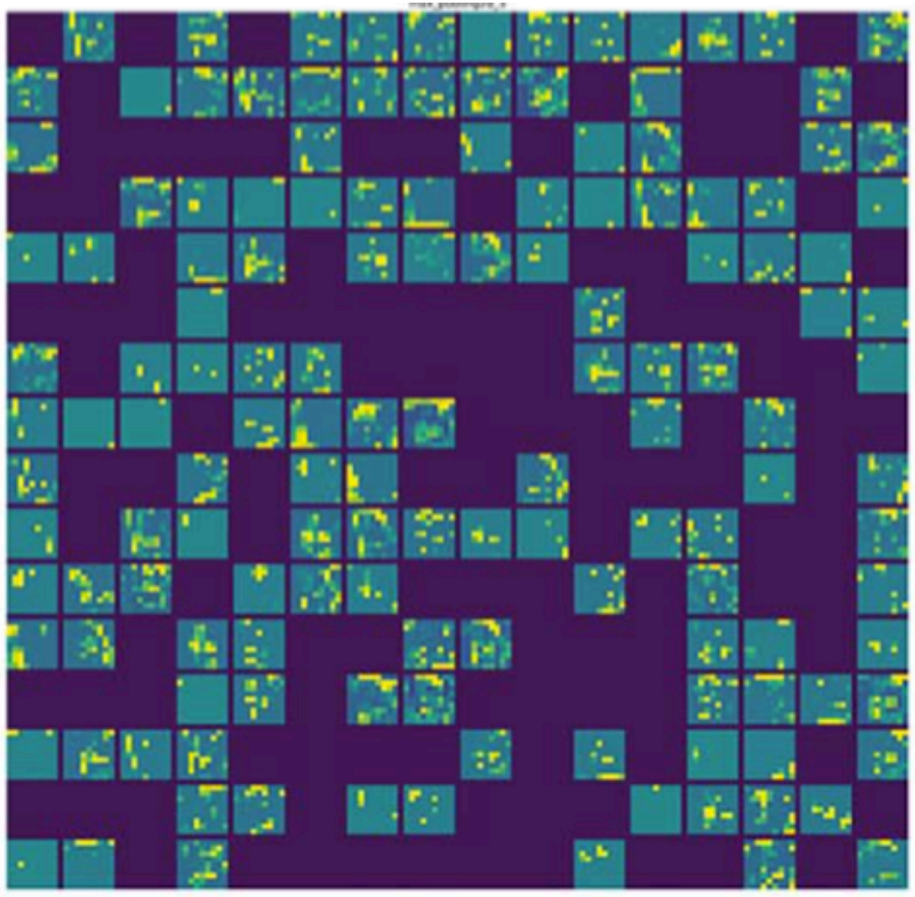



max_pooling2d_1

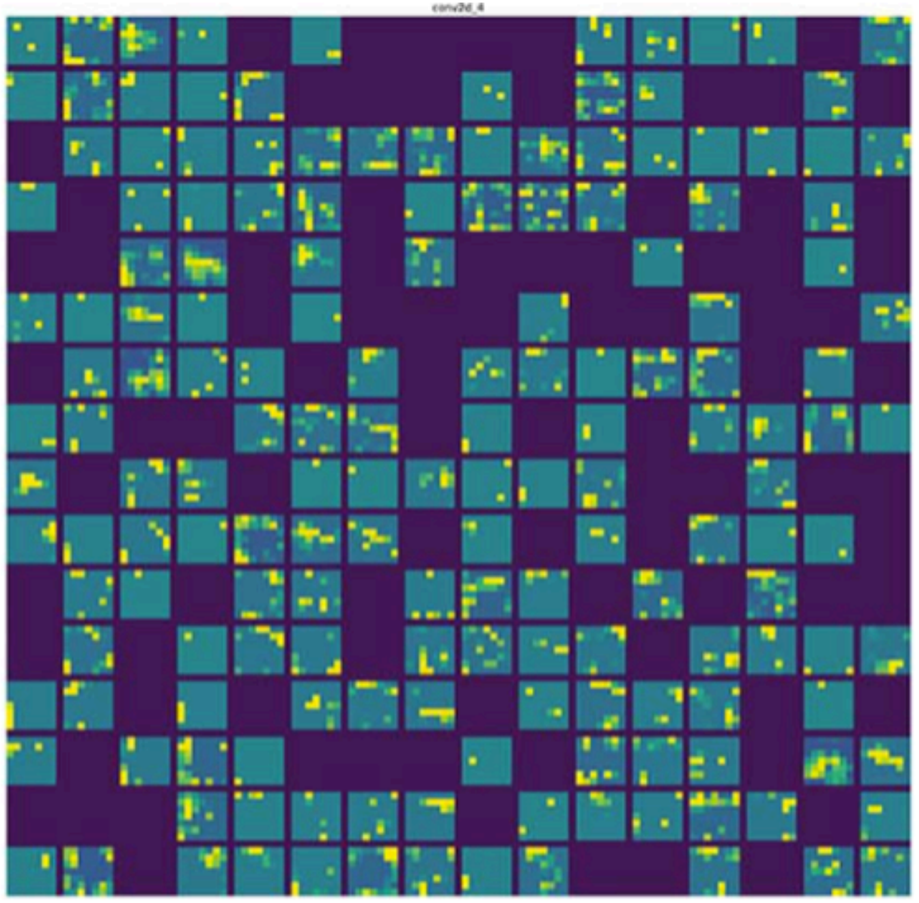








conv2d_4



Visualizing Intermediate Activations

There are a few things to note here:

- The first layer acts as a collection of various edge detectors. At that stage, the activations retain almost all of the information present in the initial picture.
- 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.” Deeper 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 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.

Quiz questions:

1. How to visualize intermediate activations?

2. How do the activations change from lower layers to higher layers?

Roadmap of this lecture:

1. Image segmentation.
2. CNN architecture patterns
    - 2.1 Residual connection
    - 2.2 Batch normalization and depthwise separable convolution
    - 2.3 Put them together for an Xception-like model
3. Visualize and interpret what CNN learns
    - 3.1 Visualize intermediate activations
    - 3.2 Visualize convolution filters
    - 3.3 Visualize heatmaps of class activation

Visualizing CNN Filters

Another easy way to inspect the filters learned by convnets is to display the visual pattern that each filter is meant to respond to. This can be done with *gradient ascent in input space*: applying *gradient descent* to the value of the input image of a convnet so as to *maximize* the response of a specific filter, starting from a blank input image. The resulting input image will be one that the chosen filter is maximally responsive to.

Visualizing CNN Filters

Let's try this with the filters of the Xception model, pretrained on ImageNet. The process is simple: we'll build a loss function that maximizes the value of a given filter in a given convolution layer, and then we'll use stochastic gradient descent to adjust the values of the input image so as to maximize this activation value. This will be our second example of a low-level gradient descent loop leveraging the `GradientTape` object (the first one was in chapter 2).

First, let's instantiate the Xception model, loaded with weights pretrained on the ImageNet dataset.

Listing 9.12 Instantiating the Xception convolutional base

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

The classification layers are irrelevant for this use case, so we don't include the top stage of the model.

Visualizing CNN Filters

We're interested in the convolutional layers of the model—the `Conv2D` and `SeparableConv2D` layers. We'll need to know their names so we can retrieve their outputs. Let's print their names, in order of depth.

Listing 9.13 Printing the names of all convolutional layers in Xception

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

You'll notice that the `SeparableConv2D` layers here are all named something like `block6_sepconv1`, `block7_sepconv2`, etc. Xception is structured into blocks, each containing several convolutional layers.

Visualizing CNN Filters

Now, let's create a second model that returns the output of a specific layer—a *feature extractor* model. Because our model is a Functional API model, it is inspectable: we can query the `output` of one of its layers and reuse it in a new model. No need to copy the entire Xception code.

Listing 9.14 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)
```

You could replace this with the name of any layer in the Xception convolutional base.

This is the layer object we're interested in.

We use model.input and layer.output to create a model that, given an input image, returns the output of our target layer.

Visualizing CNN Filters

To use this model, simply call it on some input data (note that Xception requires inputs to be preprocessed via the `keras.applications.xception.preprocess_input` function).

Listing 9.15 Using the feature extractor

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

Visualizing CNN Filters

Let's use our feature extractor model to 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'll 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 index of the filter we are considering (an integer).

Return the mean of the activation values for the filter.

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.

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 the magnitude of the updates done to the input image is always within the same range.

Listing 9.16 Loss maximization via stochastic gradient ascent

Explicitly watch the image tensor, since it isn't a TensorFlow Variable (only Variables are automatically watched in a gradient tape).

```
@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
```

Compute the loss scalar, indicating how much the current image activates the filter.

Compute the gradients of the loss with respect to the image.

Apply the "gradient normalization trick."

Move the image a little bit in a direction that activates our target filter more strongly.

Return the updated image so we can run the step function in a loop.

Now we have all the pieces. Let's put them together into a Python function that takes as input a layer name and a filter index, and returns a tensor representing the pattern that maximizes the activation of the specified filter.

Listing 9.17 Function to generate filter visualizations

```
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()
```

Amplitude of a single step

Number of gradient ascent steps to apply

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).

Repeatedly update the values of the image tensor so as to maximize our loss function.

Visualizing CNN Filters

The resulting image tensor is a floating-point array of shape `(200, 200, 3)`, with values that may not be integers within `[0, 255]`. Hence, we need to post-process this tensor to turn it into a displayable image. We do so with the following straightforward utility function.

Listing 9.18 Utility function to convert a tensor into a valid image

```
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
```

Normalize image values within the [0, 255] range.

Center crop to avoid border artifacts.

Visualizing CNN Filters

Let's try it (see figure 9.16):

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

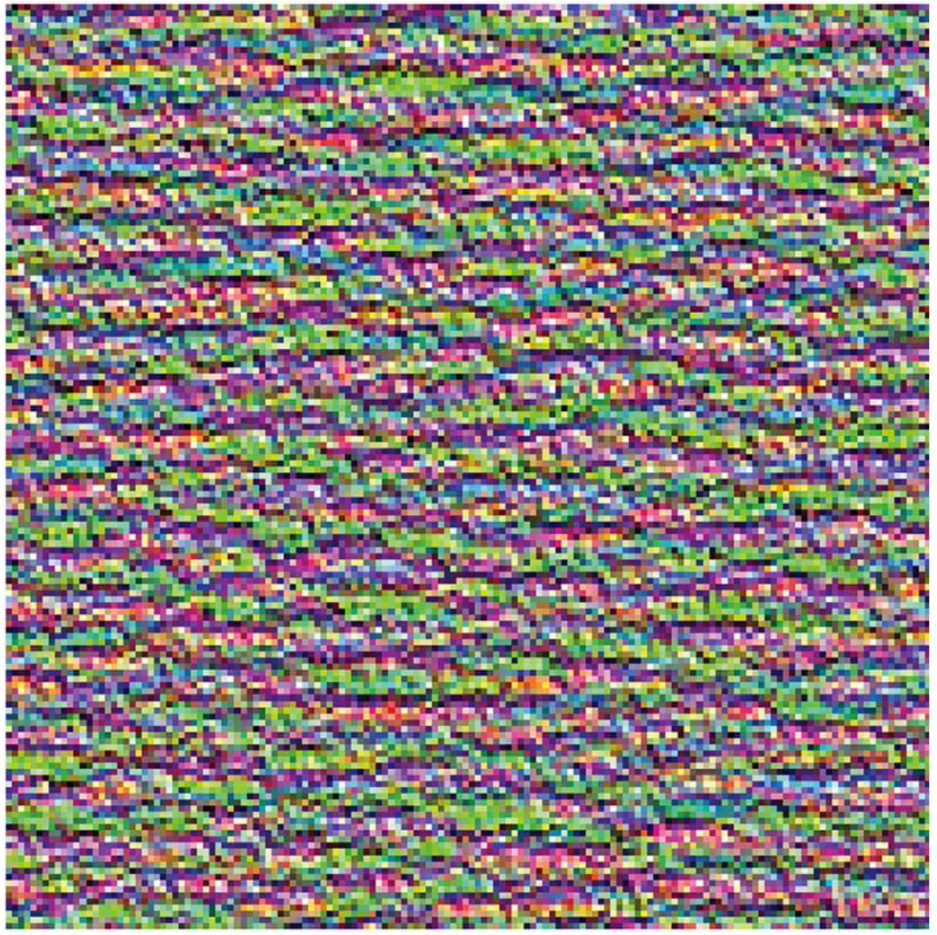


Figure 9.16 Pattern that the second channel in layer `block3_sepconv1` responds to maximally

Now the fun part: you can start visualizing every filter in the layer, and even every filter in every layer in the model.

Listing 9.19 Generating a grid of all filter response patterns in a layer

```
all_images = []
for filter_index in range(64):
    print(f"Processing filter {filter_index}")
    image = deprocess_image(
        generate_filter_pattern(filter_index)
    )
    all_images.append(image)

margin = 5
n = 8
cropped_width = img_width - 25 * 2
cropped_height = img_height - 25 * 2
width = n * cropped_width + (n - 1) * margin
height = n * cropped_height + (n - 1) * margin
stitched_filters = np.zeros((width, height, 3))

for i in range(n):
    for j in range(n):
        image = all_images[i * n + j]
        stitched_filters[
            row_start = (cropped_width + margin) * i
            row_end = (cropped_width + margin) * i + cropped_width
            column_start = (cropped_height + margin) * j
            column_end = (cropped_height + margin) * j + cropped_height

            stitched_filters[
                row_start: row_end,
                column_start: column_end, :] = image

keras.utils.save_img(
    f"filters_for_layer_{layer_name}.png", stitched_filters)
```

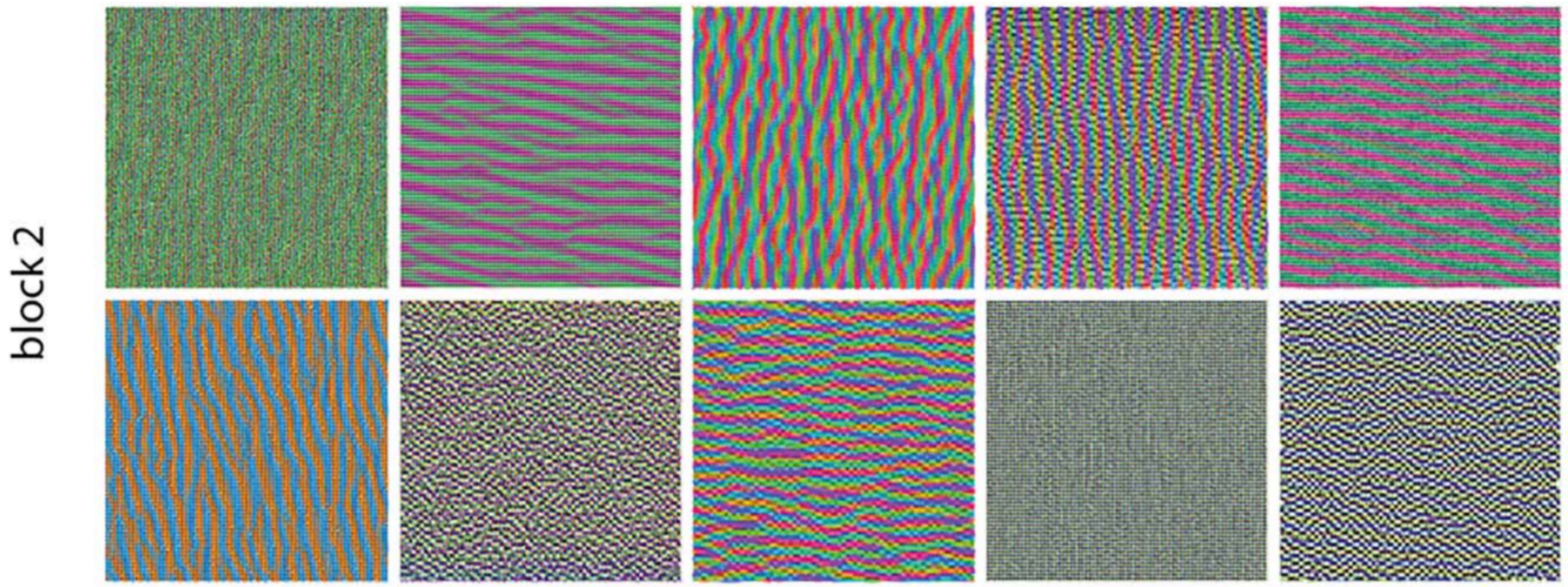
Generate and save visualizations for the first 64 filters in the layer.

Prepare a blank canvas for us to paste filter visualizations on.

Fill the picture with the saved filters.

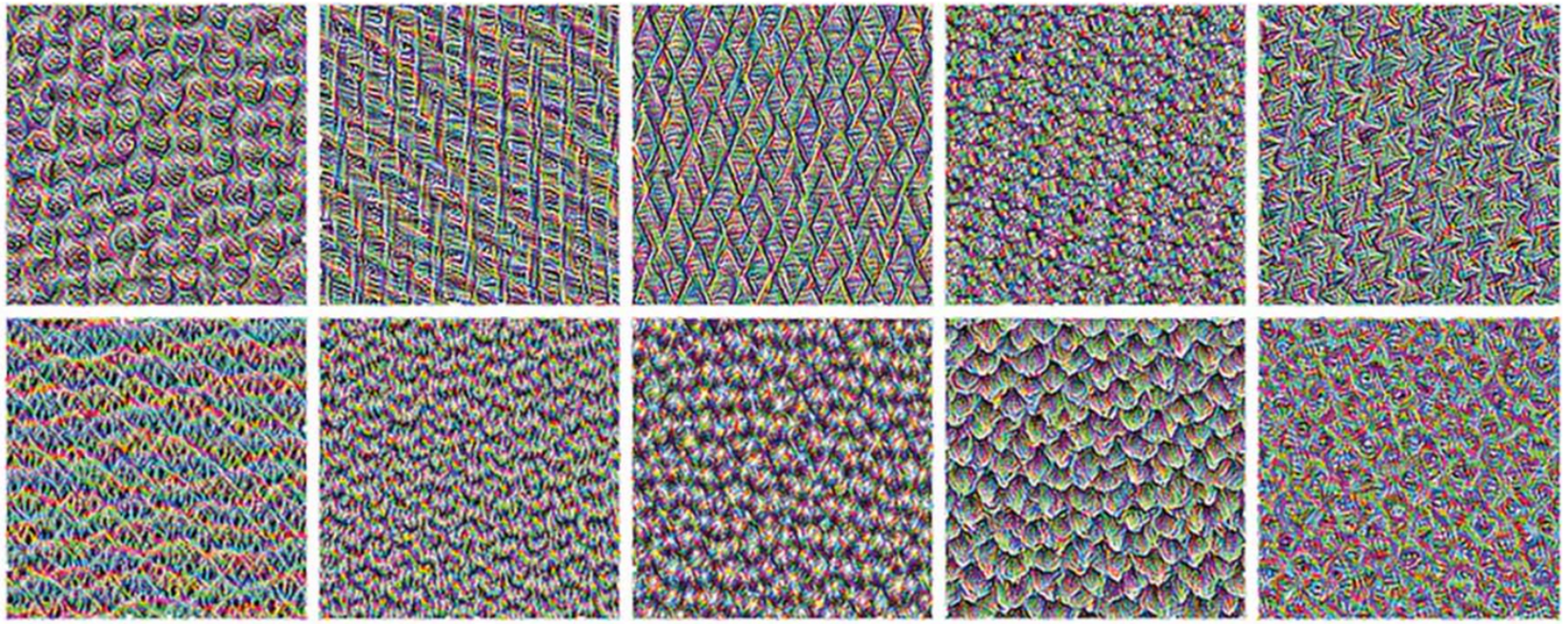
Save the canvas to disk.

Visualizing CNN Filters



Visualizing CNN Filters

block 4



Visualizing CNN Filters

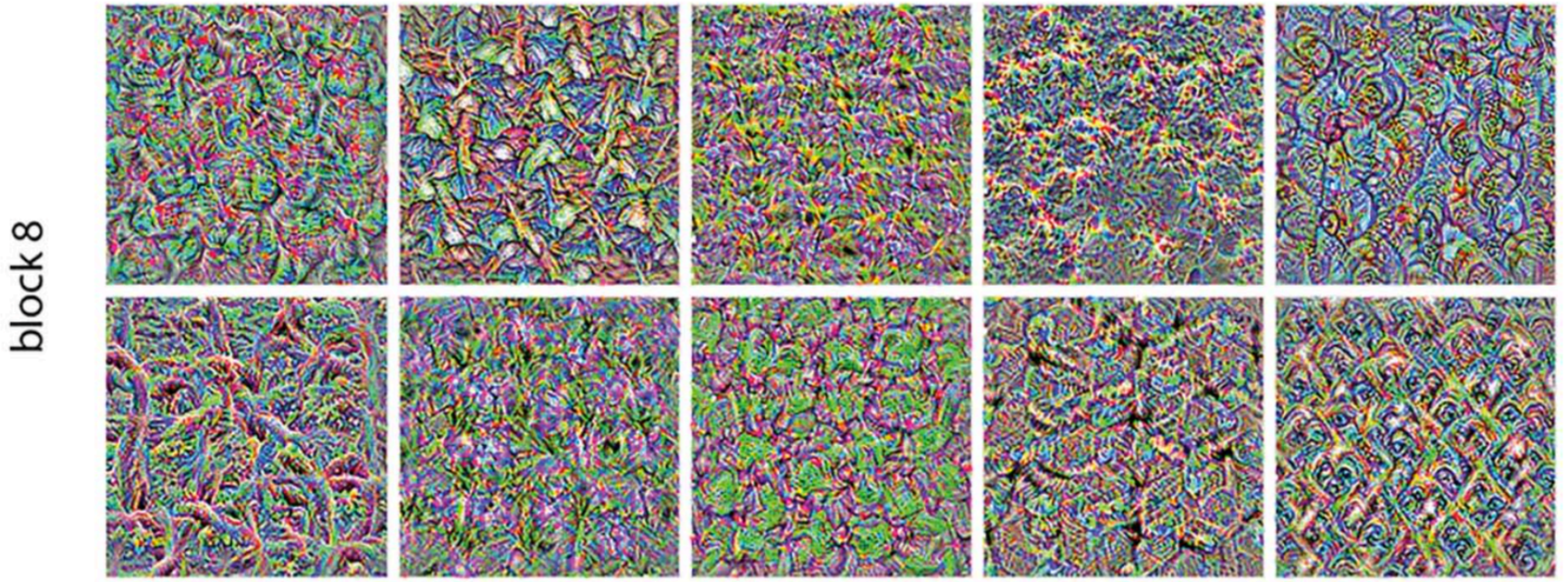


Figure 9.17 Some filter patterns for layers `block2_sepconv1`, `block4_sepconv1`, and `block8_sepconv1`

Quiz questions:

1. How to visualize CNN filters?

2. How is visualizing CNN filters different from visualizing intermediate activations?

Roadmap of this lecture:

1. Image segmentation.
2. CNN architecture patterns
    - 2.1 Residual connection
    - 2.2 Batch normalization and depthwise separable convolution
    - 2.3 Put them together for an Xception-like model
3. Visualize and interpret what CNN learns
    - 3.1 Visualize intermediate activations
    - 3.2 Visualize convolution filters
    - 3.3 Visualize heatmaps of class activation

Visualizing Heatmaps of Class Activation

We'll introduce one last visualization technique—one that is useful for 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 convnet, particularly in the case of a classification mistake (a problem domain called *model interpretability*). It can also allow you to locate specific objects in an image.

Figure 9.20 African elephant class activation heatmap over the test picture

Visualizing Heatmaps of Class Activation

The specific implementation we'll use is the one described in an article titled "Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization."[5]

Grad-CAM consists of 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 to imagine that you're 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."

Visualizing Heatmaps of Class Activation

Let's demonstrate this technique using the pretrained Xception model.

Listing 9.20 Loading the Xception network with pretrained weights

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

Note that we include the densely connected classifier on top; in all previous cases, we discarded it.

Visualizing Heatmaps of Class Activation

Consider the image of two African elephants shown in figure 9.18, possibly a mother and her calf, strolling on the savanna. Let's convert this image into something the Xcep-

Figure 9.18 Test picture of African elephants

Visualizing Heatmaps of Class Activation

Let's convert this image into something the Xception model can read: the model was trained on images of size 299 × 299, preprocessed according to a few rules that are packaged in the `keras.applications.xception.preprocess_input` utility function. So we need to load the image, resize it to 299 × 299, convert it to a NumPy `float32` tensor, and apply these preprocessing rules.

Listing 9.21 Preprocessing an input image for Xception

```
img_path = keras.utils.get_file(
    fname="elephant.jpg",
    origin="https://img-datasets.s3.amazonaws.com/elephant.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)
    array = np.expand_dims(array, axis=0)
    array = keras.applications.xception.preprocess_input(array)
    return array

img_array = get_img_array(img_path, target_size=(299, 299))
```

Download the image and store it locally under the path img_path.

Return a Python Imaging Library (PIL) image of size 299 × 299.

Return a float32 NumPy array of shape (299, 299, 3).

Add a dimension to transform the array into a batch of size (1, 299, 299, 3).

Preprocess the batch (this does channel-wise color normalization).

Visualizing Heatmaps of Class Activation

You can now run the pretrained network on the image and decode its prediction vector back to a human-readable format:

```
>>> preds = model.predict(img_array)
>>> print(keras.applications.xception.decode_predictions(preds, top=3)[0])
[("n02504458", "African_elephant", 0.8699266),
 ("n01871265", "tusker", 0.076968715),
 ("n02504013", "Indian_elephant", 0.02353728)]
```

The top three classes predicted for this image are as follows:

- African elephant (with 87% probability)
- Tusker (with 7% probability)
- Indian elephant (with 2% probability)

Visualizing Heatmaps of Class Activation

The network has recognized the image as containing an undetermined quantity of African elephants. 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])
386
```

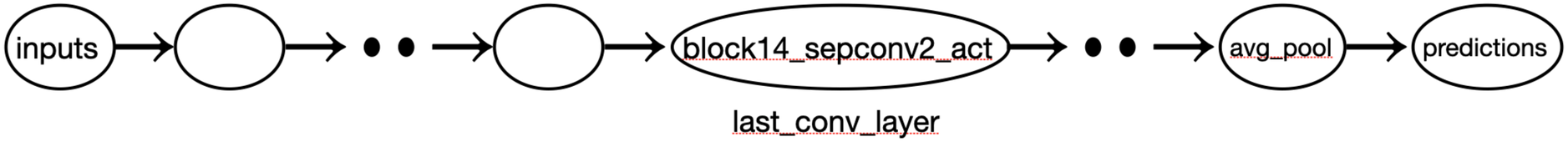
To visualize which parts of the image are the most African-elephant–like, let's set up the Grad-CAM process.

Visualizing Heatmaps of Class Activation

First, we create a model that maps the input image to the activations of the last convolutional layer.

Listing 9.22 Setting up a model that returns the last convolutional output

```
last_conv_layer_name = "block14_sepconv2_act"
classifier_layer_names = [
    "avg_pool",
    "predictions",
]
last_conv_layer = model.get_layer(last_conv_layer_name)
last_conv_layer_model = keras.Model(model.inputs, last_conv_layer.output)
```

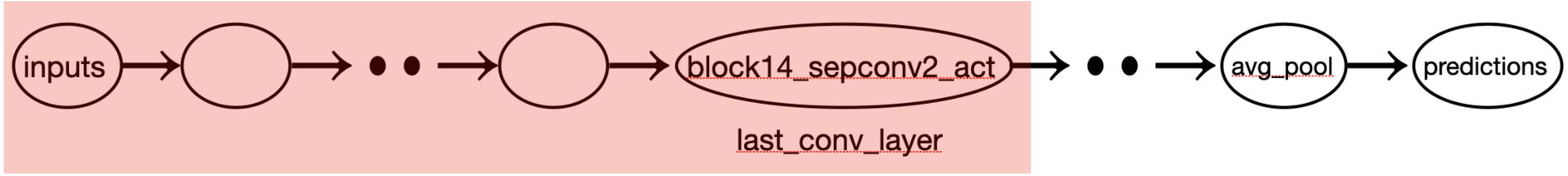


Visualizing Heatmaps of Class Activation

First, we create a model that maps the input image to the activations of the last convolutional layer.

Listing 9.22 Setting up a model that returns the last convolutional output

```
last_conv_layer_name = "block14_sepconv2_act"
classifier_layer_names = [
    "avg_pool",
    "predictions",
]
last_conv_layer = model.get_layer(last_conv_layer_name)
last_conv_layer_model = keras.Model(model.inputs, last_conv_layer.output)
```

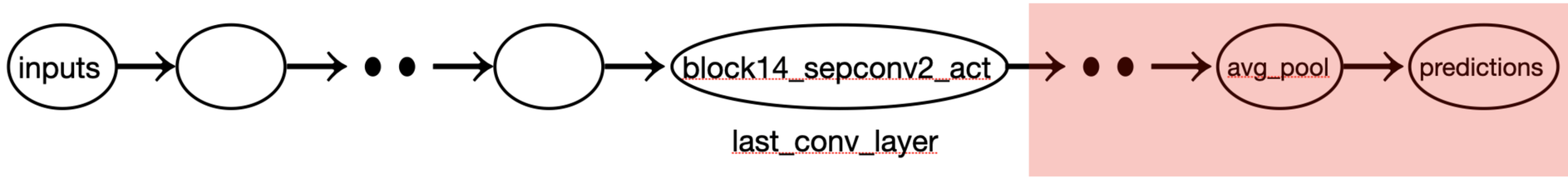


Visualizing Heatmaps of Class Activation

Second, we create a model that maps the activations of the last convolutional layer to the final class predictions.

Listing 9.23 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)
```



Visualizing Heatmaps of Class Activation

Then we compute the gradient of the top predicted class for our input image with respect to the activations of the last convolution layer.

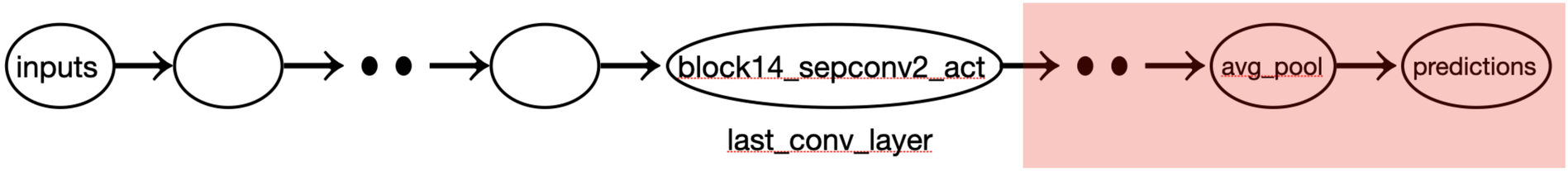
Listing 9.24 Retrieving the gradients of the top predicted class

```
import tensorflow as tf

with tf.GradientTape() as tape:
    last_conv_layer_output = last_conv_layer_model(img_array)
    tape.watch(last_conv_layer_output)
    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)
```

Compute activations of the last conv layer and make the tape watch it.

Retrieve the activation channel corresponding to the top predicted class.

This is the gradient of the top predicted class with regard to the output feature map of the last convolutional layer.



Visualizing Heatmaps of Class Activation

Now we apply pooling and importance weighting to the gradient tensor to obtain our heatmap of class activation.

Listing 9.25 Gradient pooling and channel-importance weighting

This is a vector where each entry is the mean intensity of the gradient for a given channel. It quantifies the importance of each channel with regard to 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]
for i in range(pooled_grads.shape[-1]):
    last_conv_layer_output[:, :, i] *= pooled_grads[i]
heatmap = np.mean(last_conv_layer_output, axis=-1)
```

Multiply each channel in the output of the last convolutional layer by "how important this channel is."

The channel-wise mean of the resulting feature map is our heatmap of class activation.

Visualizing Heatmaps of Class Activation

For visualization purposes, we'll also normalize the heatmap between 0 and 1. The result is shown in figure 9.19.

Listing 9.26 Heatmap post-processing

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

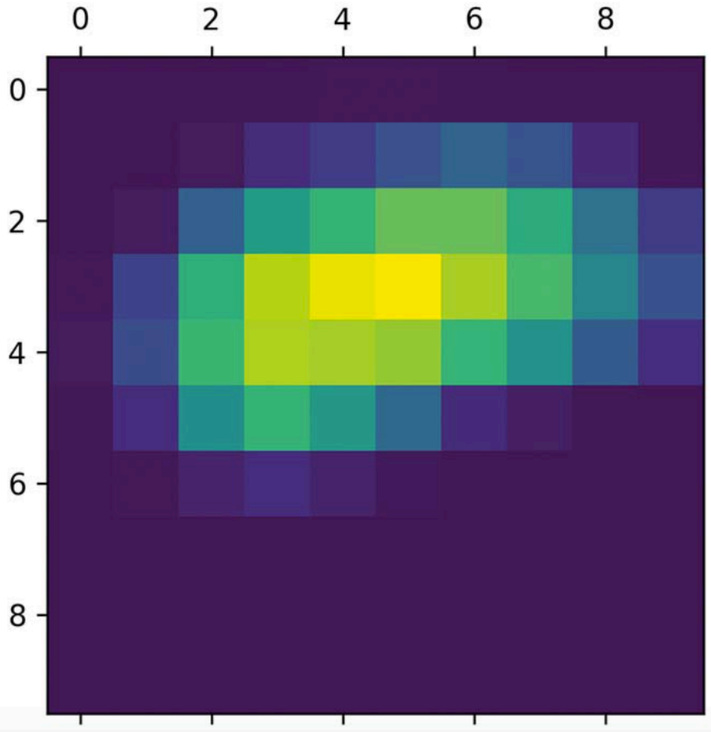


Figure 9.19 Standalone class activation heatmap

Visualizing Heatmaps of Class Activation

Finally, let's generate an image that superimposes the original image on the heatmap we just obtained (see figure 9.20).

Listing 9.27 Superimposing the heatmap on the original picture

```
import matplotlib.cm as cm

img = keras.utils.load_img(img_path)
img = keras.utils.img_to_array(img)

heatmap = np.uint8(255 * heatmap)

jet = cm.get_cmap("jet")
jet_colors = jet(np.arange(256))[:, :3]
jet_heatmap = jet_colors[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)

superimposed_img = jet_heatmap * 0.4 + img
superimposed_img = keras.utils.array_to_img(superimposed_img)

save_path = "elephant_cam.jpg"
superimposed_img.save(save_path)
```

Load the original image.

Rescale the heatmap to the range 0–255.

Use the "jet" colormap to recolorize the heatmap.

Create an image that contains the recolorized heatmap.

Superimpose the heatmap and the original image, with the heatmap at 40% opacity.

Save the superimposed image.

Visualizing Heatmaps of Class Activation

Figure 9.20 African elephant class activation heatmap over the test picture

Quiz question:

1. How to visualize heatmaps of class activation?