Deep Learning

Week 5: Convolutional neural networks

Contents

Introduction

The last week of the module focused on optimisation and regularisation of deep learning models. You learned about several techniques that are essential for successfully training large deep learning models in practice, including optimiser algorithms, weight regularisation, dropout and early stopping.

You saw how callbacks can be used in Keras to perform actions depending on how the training progresses, such as early stopping or model saving. You also learned how to implement custom callbacks in a model.

This week of the module focuses on another important model architecture. The multilayer perceptron is a general deep learning model architecture, which makes use of several fully connected (dense) layers. However it is often not the optimal choice, and in this week we will see how to build convolutional neural networks (CNNs).

Convolutional neural networks

A convolutional neural network (CNN) is a type of neural network with a special structure. It can be seen as a special case of the multilayer perceptron architecture that builds certain assumptions into the design of the model, in particular using **local connectivity** and **equivariance**.

An important motivating application for CNNs is **computer vision**, as the architectural design of these networks mimics the visual system, where neurons respond to stimulus in a restricted region of the visual field (Hubel 1959). This concept led initially to the development of the neocognitron (Fukushima 1980), and later to the modern convolutional neural network trained by backpropagation (LeCun et al 1989).

In this exposition, we will focus on CNNs for image processing, using 2-D convolutions. However, convolutional neural networks have also been very successful when applied to time series data,

using 1-D convolutions. They can also be applied to 3-D image processing tasks, or video analysis, with 3-D convolutions.

The convolution operation

The convolution operation for two (Lebesgue integrable) functions h and k is defined as

$$(h * k)(t) = \int_{-\infty}^{\infty} h(\tau)k(t-\tau)d\tau.$$

It can be described as the weighted average of the function h according to the weighting function (or **kernel**) k at each point in time t. As t changes, the weighting function emphasises different parts of the input function h.

In practice (and in the context of CNNs), we need to discretise the data, and work instead with discrete convolutions:

$$(h * k)(t) = \sum_{\tau=-\infty}^{\infty} h(\tau)k(t-\tau).$$

In this case, we can assume that both h and k (and the convolution $(h * k)$) take integer arguments. In addition, when the kernel function k has finite support, we can write the above as a finite summation.

In convolutional neural networks with image inputs, a 2-D discrete convolution is used:

$$(\mathbf{h} * \mathbf{k})(i,j) = \sum_{m,n} h(m,n)k(i-m, j-n).$$

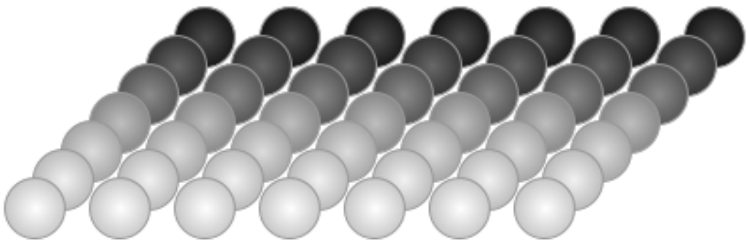
In the above, we will consider $h(i,j)$ and $k(i,j)$ to denote the (i,j)-th elements of the matrices $\mathbf{h} \in \mathbb{R}^{n_h \times n_w}$ and $\mathbf{k} \in \mathbb{R}^{k_h \times k_w}$ respectively, where n_h and n_w are the image height and width in pixels, and k_h and k_w are the kernel height and width in pixels. We will use bold lower case for the matrices $\mathbf{h}$ and $\mathbf{k}$ for consistency of notation with previous sections.

In practice, many libraries implement the **cross-correlation** operation, which is the same as above but changing the orientation of the arguments:

$$(\mathbf{h} * \mathbf{k})(i,j) = \sum_{m,n} h(i+m, j+n)k(m,n). \tag{1}$$

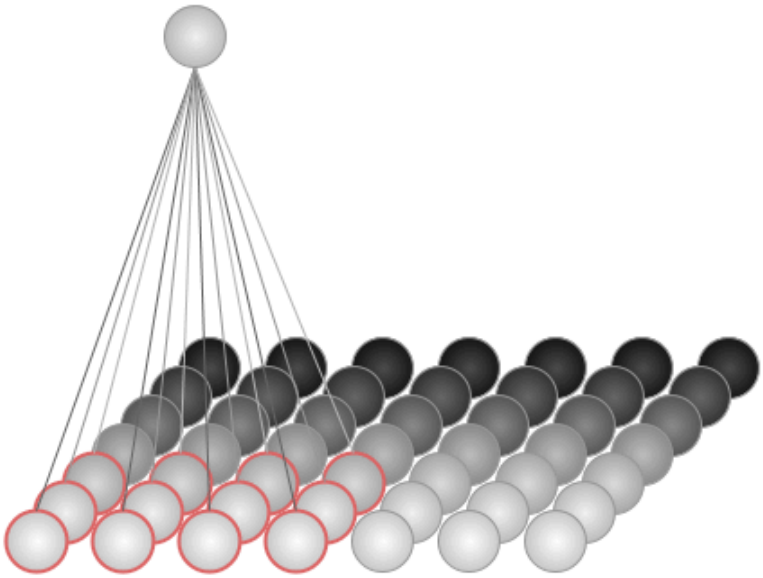
We will refer to the above operation in CNNs as a **convolution**.

As an example, consider a grayscale image $\mathbf{x} =: \mathbf{h}^{(0)} \in \mathbb{R}^{7 \times 7}$ with height and width $n_h = n_w = 7$, as illustrated in the following figure.



Pixels in a grayscale input image $\mathbf{x} \in \mathbb{R}^{7 \times 7}$

Suppose we define a kernel matrix $\mathbf{k} \in \mathbb{R}^{3\times 4}$. The operation (1) can be visualised in the following animation, as sweeping the convolutional kernel $\mathbf{k}$ across the input image. At each step, the weights of the kernel matrix $\mathbf{k} \in \mathbb{R}^{3\times 4}$ are multiplied pointwise by the values of the pixels outlined in red and summed together to produce the pre-activation of the neuron in the next hidden layer.



2-D convolution with kernel $\mathbf{k} \in \mathbb{R}^{3\times 4}$ operating on a grayscale input image $\mathbf{x} \in \mathbb{R}^{7\times 7}$

In convolutional neural networks, the convolutional layers usually consist of the operation (1) plus a bias term, followed by a pointwise activation function:

$$\mathbf{h}^{(k)} = \sigma\left((\mathbf{h}^{(k-1)} * \mathbf{k}^{(k-1)}) + b^{(k-1)}\right). \tag{2}$$

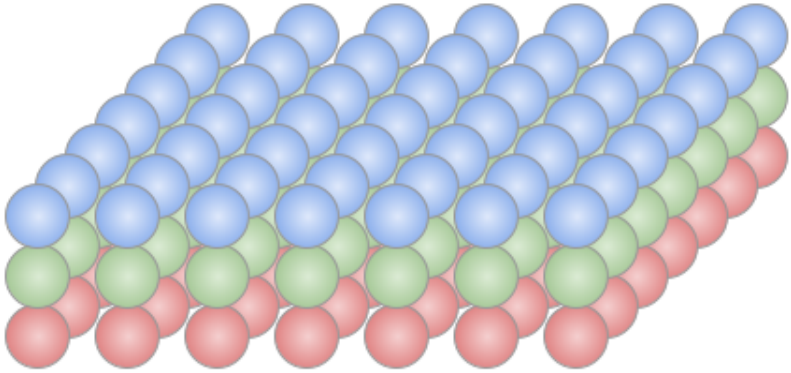
In the above, the superscript (k) indexes the layer as before, and the bias $b^{(k-1)} \in \mathbb{R}$ is added pointwise to the output of the convolution operation $(\mathbf{h}^{(k-1)} * \mathbf{k}^{(k-1)}) \in \mathbb{R}^{(n_h-k_h+1)\times(n_w-k_w+1)}$.

Different values of the kernel $\mathbf{k} \in \mathbb{R}^{3\times 4}$ will pick out different features in the input $\mathbf{h}^{(k-1)}$. The output of (2) is sometimes referred to as a **feature map**, and the kernel $\mathbf{k}$ is also referred to as a **filter**.

Note that the operation described above introduces a translational **equivariance** property in convolutional layers. That is, if the input image is translated, then the activations in the next hidden layer are also translated accordingly. Another way to view this is that the convolutional kernel searches for the same features across the input image.

Multi-channel inputs and outputs

The operations (1) and (2) can easily be extended to inputs with multiple channels. Consider, for example, an input RGB image, which has three channel values per pixel.

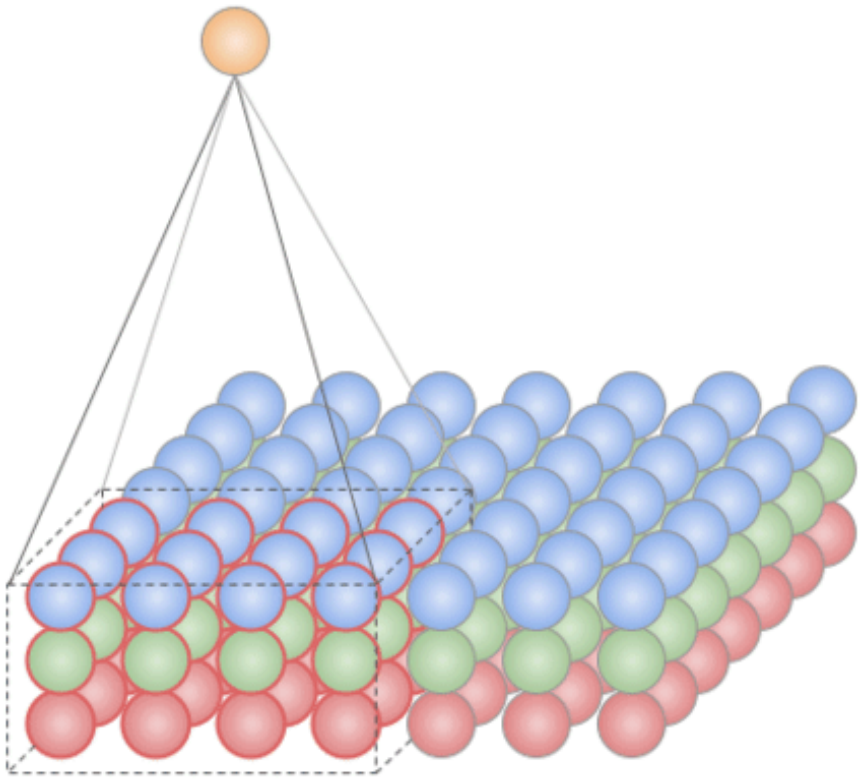


Pixels in an RGB input image $\mathbf{x} \in \mathbb{R}^{7\times7\times3}$

The input is now a rank-3 tensor $\mathbf{x} = \mathbf{h}^{(0)} \in \mathbb{R}^{7\times7\times3}$, and correspondingly we require a rank-3 kernel tensor $\mathbf{k} \in \mathbb{R}^{k_h\times k_w\times3}$. For illustration we will again choose $k_h = 3$, $k_w = 4$. The operation (1) now becomes

$$(\mathbf{h} * \mathbf{k})(i, j) = \sum_{m,n,p} h(i + m, j + n, p)k(m, n, p).$$

This is visualised in the following animation, where the kernel $\mathbf{k}$ again sweeps over the input image, this time multiplying a $3 \times 4 \times 3$ block of input pixels (outlined in red) elementwise by the values of the kernel tensor $\mathbf{k}$, and summing the results to produce the output neuron pre-activation.



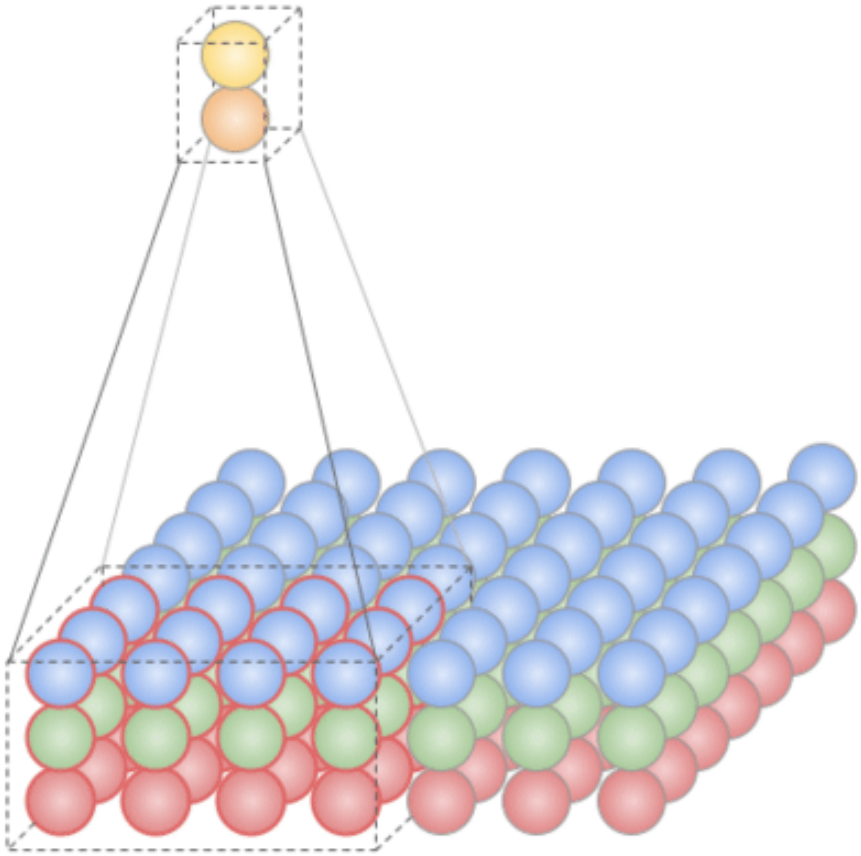
2-D convolution with kernel $\mathbf{k} \in \mathbb{R}^{3\times4\times3}$ operating on an RGB input image $\mathbf{x} \in \mathbb{R}^{7\times7\times3}$

In convolutional layers, many filters are stacked on top of each other, so as to produce a multichannel output. In practice, we implement this with a rank-4 kernel tensor $\mathbf{k} \in \mathbb{R}^{k_h,k_w,c_{in},c_{out}}$,

where c_{in} are the number of channels in the input, and c_{out} are the number of channels in the output:

$$(\mathbf{h} * \mathbf{k})(i, j, q) = \sum_{m,n,p} h(i+m, j+n, p) k(m, n, p, q).$$

This is visualised in the following animation, for an input $\mathbf{x} \in \mathbb{R}^{7\times7\times3}$ and kernel tensor $\mathbf{k} \in \mathbb{R}^{3\times4\times3\times2}$.



2-D convolution with kernel $\mathbf{k} \in \mathbb{R}^{3\times4\times3}$ operating on an RGB input image $\mathbf{x} \in \mathbb{R}^{7\times7\times3\times2}$ with 2 filters

The convolutional layer operation (2) now becomes the following

$$\mathbf{h}^{(k)} = \sigma\left((\mathbf{h}^{(k-1)} * \mathbf{k}^{(k-1)}) + \mathbf{b}^{(k-1)}\right), \tag{3}$$

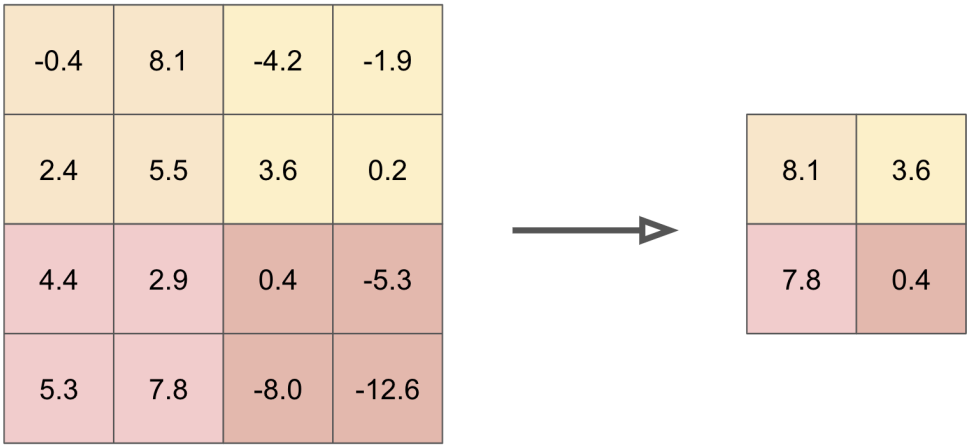
where again the superscript (k) indexes the layer, and the bias $\mathbf{b}^{(k-1)} \in \mathbb{R}^{c_{out}}$ is added pixel-wise to the output of the convolution operation $(\mathbf{h}^{(k-1)} * \mathbf{k}^{(k-1)}) \in \mathbb{R}^{(n_h-k_h+1)\times(n_w-k_w+1)\times c_{out}}$.

Pooling layers

In many CNN models, convolutional layers are alternated with pooling layers. Pooling layers downsample the spatial dimensions of a layer by computing a summary statistic of (often non-overlapping) regions of the input layer's post-activations.

A typical pooling layer type is the **max pooling** layer (Zhou & Chellappa 1988) which takes the maximum activation in a region as the single neuron activation for that region. For example, we

could divide the input layer into 2×2 squares and take the maximum value for each square. This results in halving the spatial dimensions of the following layer.



Max pooling using 2×2 pooling windows

Pooling operations are usually performed separately for each input channel, so that the spatial dimensions are downsampled, but the channel dimensions stay the same. Other common pooling operations include average pooling, or computing the l^2 norm of the pixel values within each input region.

CNNs and feature maps

In this section we will use the `Conv2D` and `MaxPool2D` layer to implement the convolution and pooling operations described above, and see how these easily fits into our existing model-building workflow.

We will also see the effect of different kernel tensor choices on the output feature maps, and look at more complex feature maps from a pre-trained model.

In [2]:
```
import keras
from keras import ops
```

The `Conv2D` and `MaxPool2D` classes are imported from the `keras.layers` module just as the `Input`, `Flatten` and `Dense` layers we have already worked with. Note that there are also 1-D and 3-D variants of these layers available, which both work in a similar way.

In [4]:
```
# Define a dummy model with Conv2D and MaxPool2D layers

from keras import Sequential
from keras.layers import Input, Conv2D, MaxPool2D

model = Sequential([
    Input(shape=(32, 32, 3)),
    Conv2D(8, (3, 5), activation='relu'),
    MaxPool2D((2, 2)),
    Conv2D(16, 3, activation='relu'),
    MaxPool2D(2)
])
```

In [5]:
```
# Print the model summary

model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 30, 28, 8)	368
max_pooling2d_2 (MaxPooling2D)	(None, 15, 14, 8)	0
conv2d_3 (Conv2D)	(None, 13, 12, 16)	1,168
max_pooling2d_3 (MaxPooling2D)	(None, 6, 6, 16)	0

Total params: 1,536 (6.00 KB)
Trainable params: 1,536 (6.00 KB)
Non-trainable params: 0 (0.00 B)

In [6]:
```
# Inspect the layer variables' shapes

print(model.layers[0].kernel.shape)
print(model.layers[0].bias.shape)

print(model.layers[2].kernel.shape)
print(model.layers[2].bias.shape)
```

```
(3, 5, 3, 8)
(8,)
(3, 3, 8, 16)
(16,)
```

Edge detection filters

The kernels (or filters) in CNNs are typically learned with backpropagation. However, simple low-level features such as edge detection kernels can also be designed by hand. In this section we will see the output of such low-level kernels.

In [7]:
```
# Define a simple model with a Conv2D layer

model = Sequential([
    Input(shape=(None, None, 1)),
    Conv2D(1, (3, 3), activation=None, use_bias=False)
])
```

A shape dimension of `None` indicates that the model can take flexible input sizes in this dimension.

In [8]:
```
# Inspect the model's weights

model.weights
```

Out[8]:
```
[<KerasVariable shape=(3, 3, 1, 1), dtype=float32, path=sequential_2/conv2d_4/kern
el>]
```

In [9]:
```
# Load an image as grayscale

import matplotlib.pyplot as plt
from pathlib import Path

image = keras.utils.load_img(Path("./figures/oscar.png"),
                             color_mode='grayscale')
# Loads into PIL format
image = keras.utils.img_to_array(image, dtype='int32')
plt.figure(figsize=(8, 6))
plt.imshow(image, cmap='gray')
plt.axis('off')
plt.show()
```

A simple and intuitive edge detection kernel is the Sobel operator:

In [10]:
```
# Define simple edge detection filters

sobel_x = ops.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]],
                    dtype='float32')
sobel_y = ops.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]],
                    dtype='float32')

print(sobel_x)
print(sobel_y)
```

```
tf.Tensor(
[[ 1.  0. -1.]
 [ 2.  0. -2.]
 [ 1.  0. -1.]], shape=(3, 3), dtype=float32)
tf.Tensor(
[[ 1.  2.  1.]
 [ 0.  0.  0.]
 [-1. -2. -1.]], shape=(3, 3), dtype=float32)
```

In [11]:
```
# Set the model kernel

def assign_filter(arr):
    model.weights[0].assign(arr[:, :, None, None])
```

In [13]:
```
# Compute the feature maps

assign_filter(sobel_x)
gx = model(image[None, ...])[0]

assign_filter(sobel_y)
gy = model(image[None, ...])[0]

g = ops.sqrt(ops.square(gx) + ops.square(gy))
```

In [14]:
```
# View the image and feature map

fig = plt.figure(figsize=(17, 6))
fig.add_subplot(121)
```

```
plt.imshow(image, cmap='gray')
plt.axis('off')
fig.add_subplot(122)
plt.imshow(ops.convert_to_numpy(g), cmap='gray')
plt.axis('off')
plt.show()
```

Extract learned features from a pre-trained model

In this section we will load a CNN model that has been pre-trained on the ImageNet dataset, which is a large scale image classification dataset which to date has over 20,000 categories and over 14 million images. Large deep learning models trained on this dataset tend to learn general, useful representations of image features that can be used for a range of image processing tasks.

Below we will load the VGG-19 model (Simonyan & Zisserman 2015), which is available to load as a pre-trained model in the `keras.applications` module. This might take a minute or two to download the first time you run the cell.

In [15]:
```
# Load the VGG-19 model

vgg = keras.applications.VGG19(weights='imagenet', include_top=False)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/v
gg19/vgg19_weights_tf_dim_ordering_tf_kernels_notop.h5
80134624/80134624 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

In [16]:
```
# Print the model summary

vgg.summary()
```

```
Model: "vgg19"
```

Layer (type)	Output Shape	Param #
input_layer_3 (InputLayer)	(None, None, None, 3)	0
block1_conv1 (Conv2D)	(None, None, None, 64)	1,792
block1_conv2 (Conv2D)	(None, None, None, 64)	36,928
block1_pool (MaxPooling2D)	(None, None, None, 64)	0
block2_conv1 (Conv2D)	(None, None, None, 128)	73,856
block2_conv2 (Conv2D)	(None, None, None, 128)	147,584
block2_pool (MaxPooling2D)	(None, None, None, 128)	0
block3_conv1 (Conv2D)	(None, None, None, 256)	295,168
block3_conv2 (Conv2D)	(None, None, None, 256)	590,080
block3_conv3 (Conv2D)	(None, None, None, 256)	590,080
block3_conv4 (Conv2D)	(None, None, None, 256)	590,080
block3_pool (MaxPooling2D)	(None, None, None, 256)	0
block4_conv1 (Conv2D)	(None, None, None, 512)	1,180,160
block4_conv2 (Conv2D)	(None, None, None, 512)	2,359,808
block4_conv3 (Conv2D)	(None, None, None, 512)	2,359,808
block4_conv4 (Conv2D)	(None, None, None, 512)	2,359,808
block4_pool (MaxPooling2D)	(None, None, None, 512)	0
block5_conv1 (Conv2D)	(None, None, None, 512)	2,359,808
block5_conv2 (Conv2D)	(None, None, None, 512)	2,359,808
block5_conv3 (Conv2D)	(None, None, None, 512)	2,359,808
block5_conv4 (Conv2D)	(None, None, None, 512)	2,359,808
block5_pool (MaxPooling2D)	(None, None, None, 512)	0

Total params: 20,024,384 (76.39 MB)

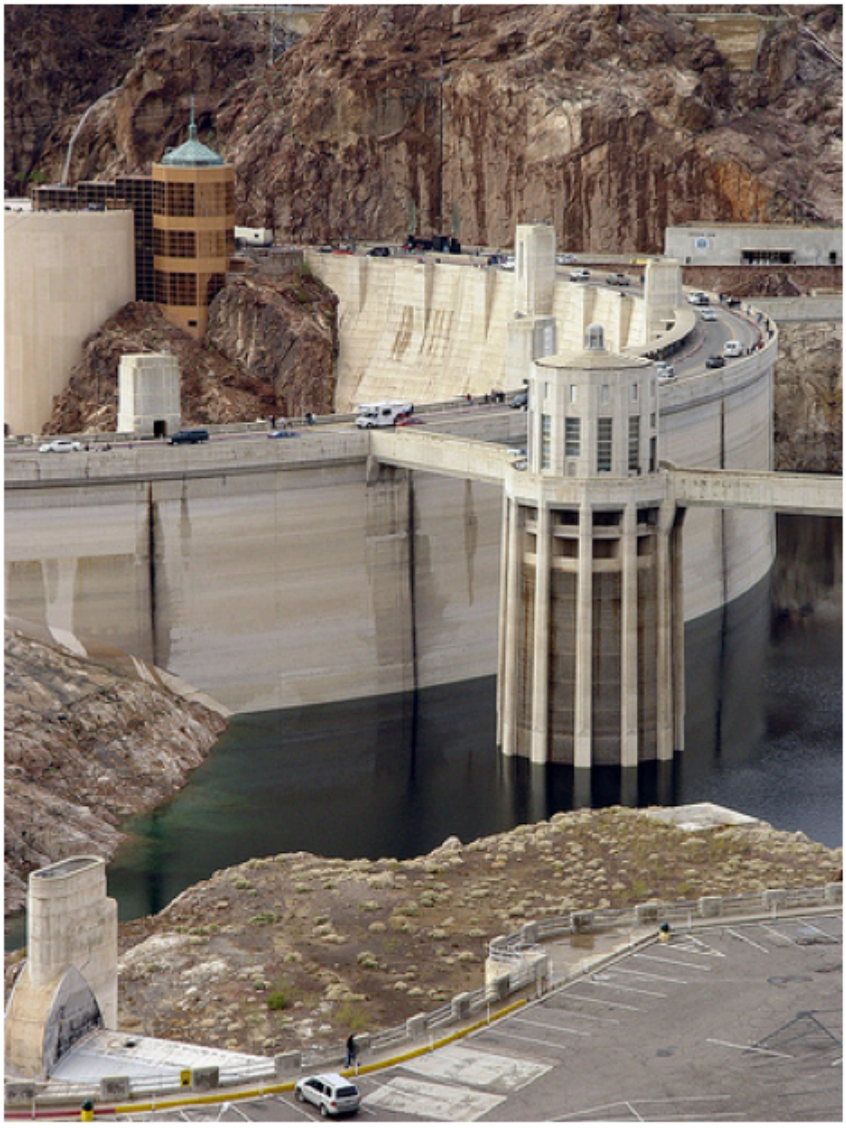
Trainable params: 20,024,384 (76.39 MB)

Non-trainable params: 0 (0.00 B)

We will visualise the features extracted by this model at different levels of hierarchy for the following image:

In [17]:
```
# Load a colour image

image = keras.utils.load_img(Path("./figures/hoover_dam.JPEG"),
                             color_mode='rgb')
image = keras.utils.img_to_array(image, dtype='int32')
plt.figure(figsize=(6, 10))
plt.imshow(image)
plt.axis('off')
plt.show()
```



We will use the functional API to create a multi-output model that outputs different hidden layer outputs within the model. The Keras functional API is a way to create models that are more flexible than the Sequential API. The functional API can handle models that are defined as more complicated DAGS than just a linear sequence of layers. It can also handle shared layers and multiple inputs or outputs.

Defining models using the functional API typically involves defining one or more input Tensors `inputs` (usually using the `Input` layer). The graph computations are carried by defining additional Tensors as the result of operations on previously defined Tensors (these operations are often themselves defined by layer objects). After all computations have been carried out, the graph outputs are defined as one or more output Tensors `outputs`. These `inputs` and `outputs`

Tensors are then used to define a model as `model = Model(inputs=inputs, outputs=outputs)`.

It is also worth knowing that any Keras `Model` (including models created using the `Sequential` API) have `inputs` and `outputs` attributes. These attributes store the input and output Tensors of the model. Below, we use the `inputs` attribute of the pre-trained VGG-19 model and the output Tensors of selected layers to define a new `Model` object.

In [18]:
```
# Define the multi-output model

from keras.models import Model

inputs = vgg.inputs
layer_names = ['block1_conv2', 'block2_conv2', 'block3_conv4',
               'block4_conv4', 'block5_conv4']
outputs = [
    vgg.get_layer(layer_name).output for layer_name in layer_names]
vgg_features = Model(inputs=inputs, outputs=outputs)
```

In [19]:
```
# View the model inputs and outputs Tensors

print(vgg_features.inputs)
print(vgg_features.outputs)
```

```
[<KerasTensor shape=(None, None, None, 3), dtype=float32, sparse=None, name=keras_te
nsor_12>]
[<KerasTensor shape=(None, None, None, 64), dtype=float32, sparse=False, name=keras_
tensor_14>, <KerasTensor shape=(None, None, None, 128), dtype=float32, sparse=False,
name=keras_tensor_17>, <KerasTensor shape=(None, None, None, 256), dtype=float32, sp
arse=False, name=keras_tensor_22>, <KerasTensor shape=(None, None, None, 512), dtype
=float32, sparse=False, name=keras_tensor_27>, <KerasTensor shape=(None, None, None,
512), dtype=float32, sparse=False, name=keras_tensor_32>]
```

In [21]:
```
# Extract the hierarchical features for this image

image_processed = keras.applications.vgg19.preprocess_input(image)
features = vgg_features(image_processed[None, ...])
features = [image] + [ops.convert_to_numpy(f) for f in features]
```

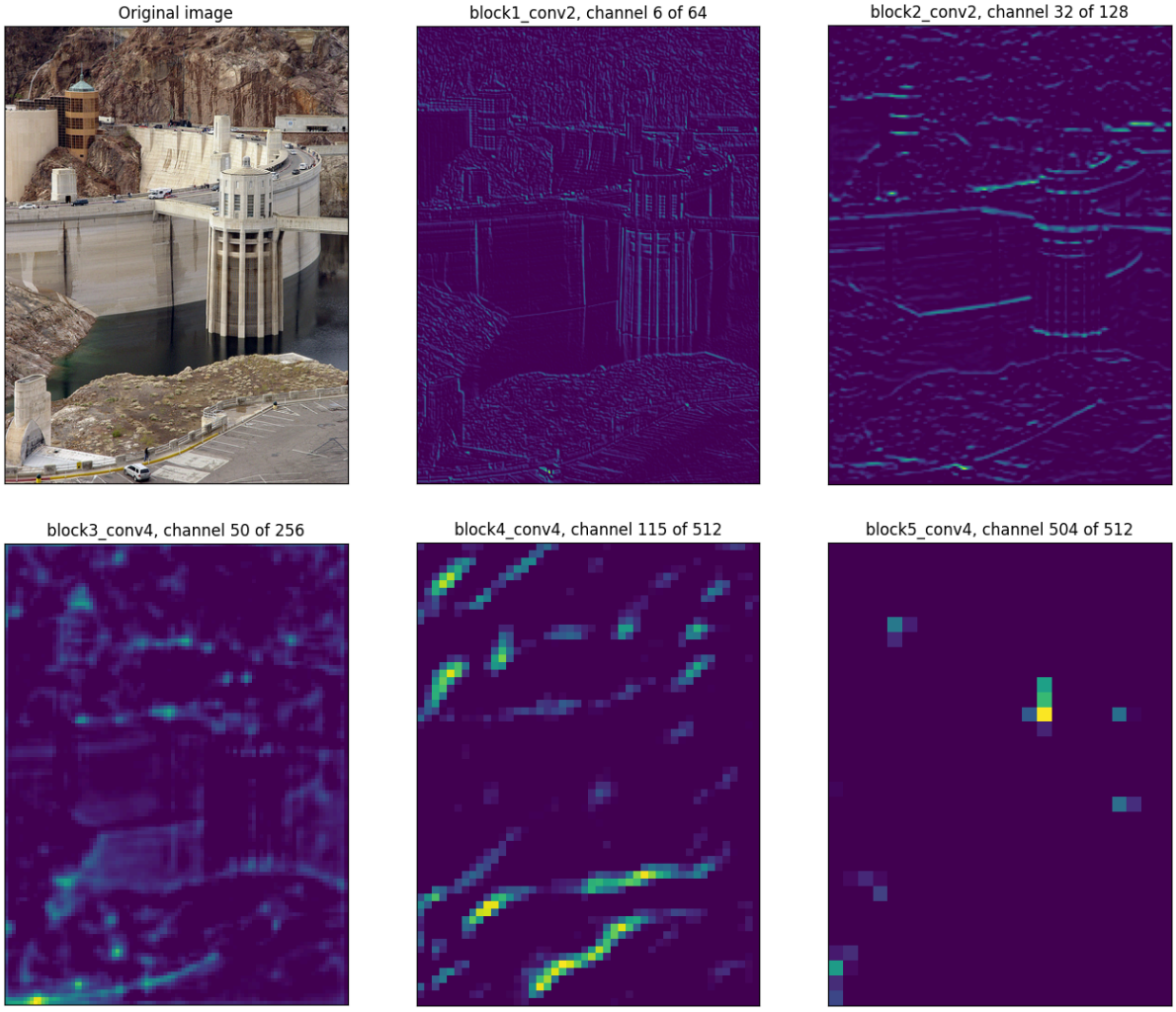
In [22]:
```
# Visualise the features

import numpy as np

n_rows, n_cols = 2, 3
fig, axes = plt.subplots(n_rows, n_cols, figsize=(16, 14))
fig.subplots_adjust(hspace=0.05, wspace=0.2)

for i in range(len(features)):
    feature_map = features[i]
    num_channels = feature_map.shape[-1]
    row = i // n_cols
    col = i % n_cols
    if i == 0:
        axes[row, col].imshow(image)
        axes[row, col].set_title('Original image')
    else:
        random_feature = np.random.choice(num_channels)
        axes[row, col].imshow(feature_map[0, ..., random_feature])
        axes[row, col].set_title('{}, channel {} of {}'.format(
            layer_names[i-1],random_feature + 1, num_channels))

    axes[row, col].get_xaxis().set_visible(False)
    axes[row, col].get_yaxis().set_visible(False)
plt.show()
```



Exercise: load one of your own images to view the features extracted by the VGG-19 network.

Padding and strides

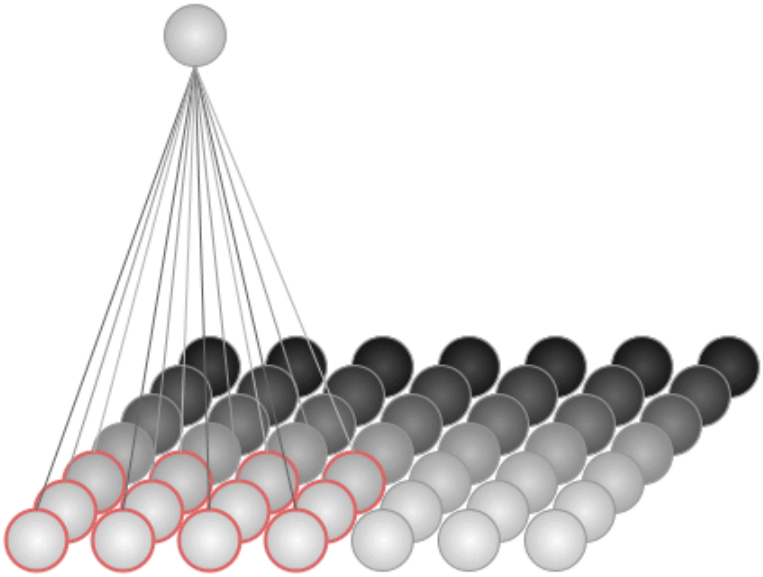
Padding and strides are additional properties of convolutional (and pooling) layers that can give some flexibility over the spatial dimensions of the output.

In this section, we will define these properties and the effects they have on the input and output dimensions of a convolutional layer. For a more complete guide to convolutional arithmetic, see Dumoulin & Visin 2016.

In [23]:
```
import keras
from keras import ops
```

Padding

Recall that in our earlier example, an input grayscale image $\mathbf{x} \in \mathbb{R}^{7\times7\times1}$ convolved with a kernel $\mathbf{k} \in \mathbb{R}^{3\times4\times1\times1}$ produced an output of size $\mathbf{h} \in \mathbb{R}^{5\times4\times1}$:



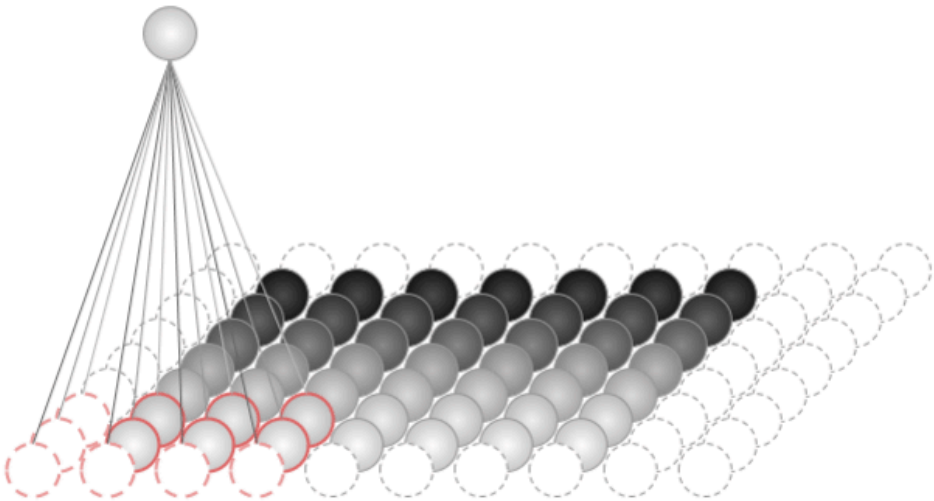
A kernel $\mathbf{k} \in \mathbb{R}^{3\times4\times1\times1}$ operating on a grayscale input image $\mathbf{x} \in \mathbb{R}^{7\times7\times1}$ produces an output $\mathbf{h} \in \mathbb{R}^{5\times4\times1}$

In general, for a spatial dimension of size i and a kernel of width k, the output size o is given by

$$o = i - k + 1 \tag{4}$$

In many model architectures, it is desirable to keep the spatial dimensions the same in the output of a convolutional layer. This can be achieved by padding the input layer with zeros.

In the case of our kernel tensor $\mathbf{k} \in \mathbb{R}^{3\times4\times1\times1}$, if we add 2 zeros in the first dimension and 3 zeros in the second dimension (distributed on either side of the image), this will result in an output $\mathbf{h} \in \mathbb{R}^{7\times7\times1}$ that has the same spatial dimensions as the input $\mathbf{x}$. This type of padding is known as "SAME" padding.



A kernel $\mathbf{k} \in \mathbb{R}^{3\times4\times1\times1}$ operating on a grayscale input image $\mathbf{x} \in \mathbb{R}^{7\times7\times1}$ with "SAME" padding

Now, if p zeros are added to our input size i with kernel width k, then the output size o is given by

$$o = i + p - k + 1 \tag{5}$$

If $p = k - 1$ ("SAME" padding) then the $o = i$. If $p = 0$ ("VALID" padding) then we recover (4) and $o = i - k + 1$. There is also "FULL" padding where $p = 2(k - 1)$, so that $o = i + k - 1$, although this is less common.

In Keras, we can easily apply zero padding to convolutional layers with the `padding` keyword argument, which can be set to `"VALID"` (the default) or `"SAME"`.

In [24]:
```
# Create a CNN with 'VALID' and 'SAME' padding

from keras.models import Model
from keras.layers import Input, Conv2D, MaxPool2D

inputs = Input(shape=(32, 32, 3))
h = Conv2D(16, (3, 5), padding="VALID", activation='relu')(inputs)
h = MaxPool2D(2)(h)
outputs = Conv2D(16, 3, padding="SAME", activation='relu')(h)

model = Model(inputs=inputs, outputs=outputs)
model.summary()
```

Model: "functional_4"

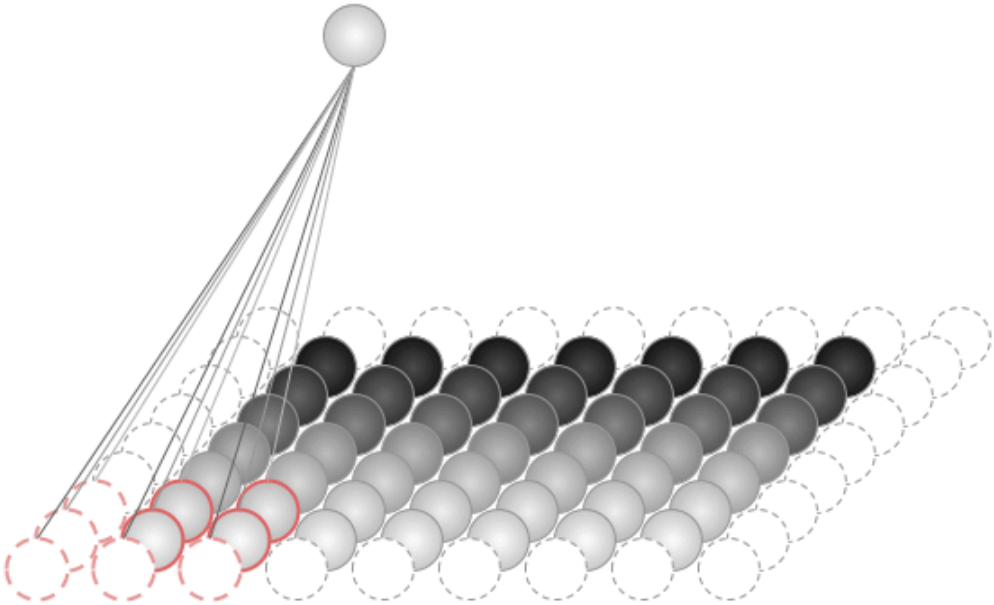
Layer (type)	Output Shape	Param #
input_layer_4 (InputLayer)	(None, 32, 32, 3)	0
conv2d_5 (Conv2D)	(None, 30, 28, 16)	736
max_pooling2d_4 (MaxPooling2D)	(None, 15, 14, 16)	0
conv2d_6 (Conv2D)	(None, 15, 14, 16)	2,320

Total params: 3,056 (11.94 KB)
Trainable params: 3,056 (11.94 KB)
Non-trainable params: 0 (0.00 B)

Note the input and output shapes of each of the convolutional layers.

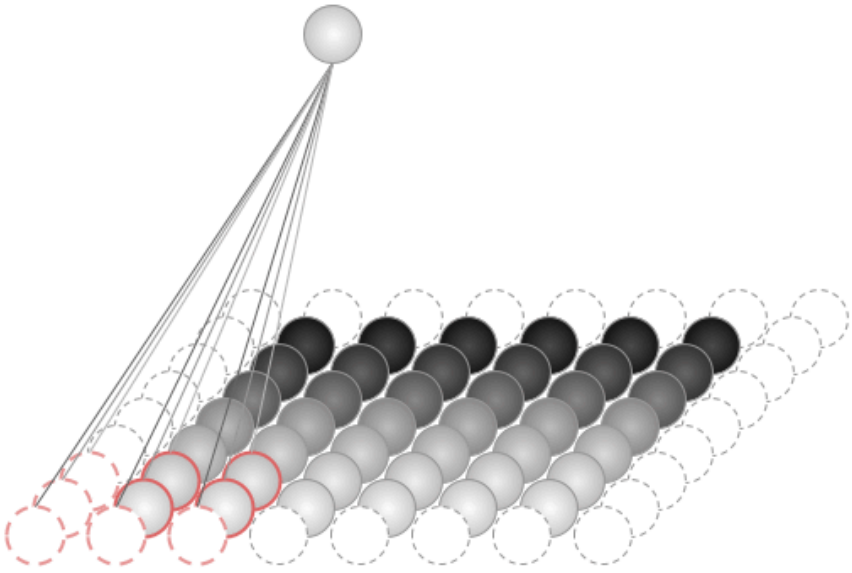
Strides

Convolutions may also use a stride s, which is the distance between consecutive positions of the kernel. So far, all of our examples have used $s = 1$, however it is easy to see that using $s > 1$ leads to a downsampling of the input. The following animation shows our input $\mathbf{x} \in \mathbb{R}^{7\times7\times1}$, this time with a kernel $\mathbf{k} \in \mathbb{R}^{3\times3\times1\times1}$, "SAME" padding, and a stride $s = 2$ in both spatial dimensions:



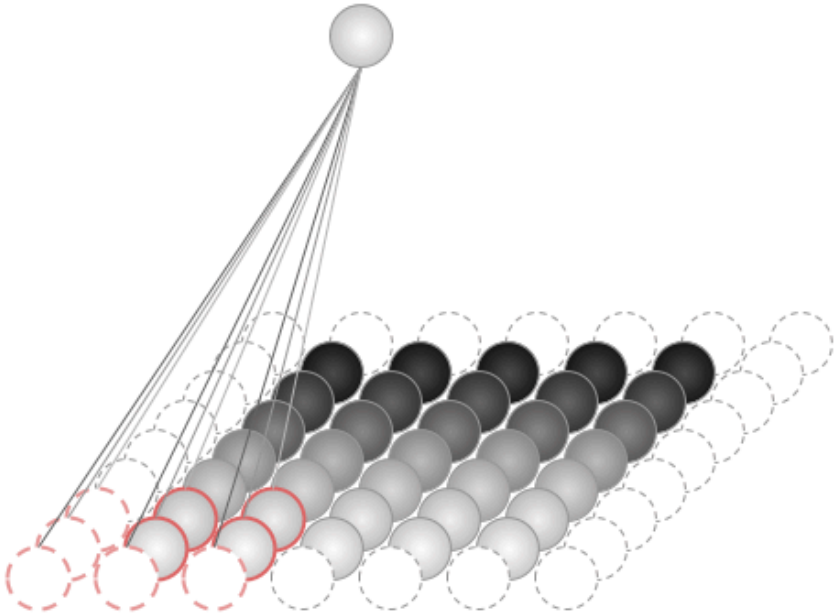
A kernel $\mathbf{k} \in \mathbb{R}^{3\times3\times1\times1}$ operating on a grayscale input image $\mathbf{x} \in \mathbb{R}^{7\times7\times1}$ with "SAME" padding and stride of $(2, 2)$

In the above example, the stride and kernel size exactly divides the input and padding size, however this does not need to be the case, such as in the following example where $\mathbf{x} \in \mathbb{R}^{7\times6\times1}$:



A kernel $\mathbf{k} \in \mathbb{R}^{3\times3\times1\times1}$ operating on a grayscale input image $\mathbf{x} \in \mathbb{R}^{7\times6\times1}$ with "SAME" padding and stride of $(2, 2)$

In this example the input $\mathbf{x} \in \mathbb{R}^{7\times6\times1}$ has been downsampled to an output $\mathbf{h} \in \mathbb{R}^{4\times3\times1}$. Note that if the input image size was instead $7 \times 5 \times 1$ then the output size would again be $4 \times 3 \times 1$.



A kernel $\mathbf{k} \in \mathbb{R}^{3\times3\times1\times1}$ operating on a grayscale input image $\mathbf{x} \in \mathbb{R}^{7\times5\times1}$ with "SAME" padding and stride of $(2, 2)$

In all cases, for a spatial dimension of size i with padding p and a kernel of width k with stride s, the output size o is given by

$$o = \left\lfloor \frac{i + p - k}{s} \right\rfloor + 1 \tag{6}$$

In Keras, strides can be set in both convolutional and pooling layers using the `strides` keyword argument.

Just as with the `kernel_size` argument for `Conv2D` layers (and the `pool_size` argument for `MaxPool2D` layers), either a tuple of integers can be passed in, or a single integer - which indicates the stride should be the same in all spatial dimensions.

In `MaxPool2D` layers, the default stride is set to be equal to the `pool_size`.

In [25]:
```
# Create a CNN using strides in the Conv2D and MaxPool2D layers

from keras.models import Model
from keras.layers import Input, Conv2D, MaxPool2D, Flatten, Dense

inputs = Input(shape=(128, 128, 3))
h = Conv2D(16, (5, 5), padding="VALID", strides=(2, 3),
           activation='relu')(inputs)
h = MaxPool2D(2)(h)
h = Conv2D(16, 7, padding="SAME", activation='relu')(h)
h = MaxPool2D((2, 4), strides=(3, 2))(h)
h = Conv2D(16, 3, padding="SAME", strides=2, activation='relu')(h)
h = Flatten()(h)
h = Dense(16, activation='relu')(h)
outputs = Dense(1, activation='sigmoid')(h)

model = Model(inputs=inputs, outputs=outputs)
model.summary()
```

```
Model: "functional_5"
```

Layer (type)	Output Shape	Param #
input_layer_5 (InputLayer)	(None, 128, 128, 3)	0
conv2d_7 (Conv2D)	(None, 62, 42, 16)	1,216
max_pooling2d_5 (MaxPooling2D)	(None, 31, 21, 16)	0
conv2d_8 (Conv2D)	(None, 31, 21, 16)	12,560
max_pooling2d_6 (MaxPooling2D)	(None, 10, 9, 16)	0
conv2d_9 (Conv2D)	(None, 5, 5, 16)	2,320
flatten (Flatten)	(None, 400)	0
dense (Dense)	(None, 16)	6,416
dense_1 (Dense)	(None, 1)	17

Total params: 22,529 (88.00 KB)
Trainable params: 22,529 (88.00 KB)
Non-trainable params: 0 (0.00 B)

Note the input and output shapes of each of each layers, and compare with the formula (6) for the output size given above.

Transposed convolutions

Transposed convolutions are a type of convolutional layer that are not as common as the convolutional and pooling layers introduced earlier, but are important for certain applications. They can be seen as a kind of inverse of regular convolutional layers. The main problem they are trying to solve is to give a consistent way of upsampling an input, rather than downsampling it.

We've seen that in convolutional layers and in pooling layers, depending on the layer settings, they can reduce the spatial dimensions of the input. In certain deep learning architectures, we would instead like to increase the spatial dimensions of an input. An example of this is when we are using encoder and decoder networks, which we will come to later on in the course.

Transposed convolutions give us a way of doing this, while still preserving the main structural properties of convolutional layers. They are the analogue to transposing the weight matrix in fully connected layers; they essentially swap the forward and backward passes of a convolution. Here we will briefly cover the core properties of the transposed convolution. For a more detailed treatment, see Dumoulin & Visin 2016.

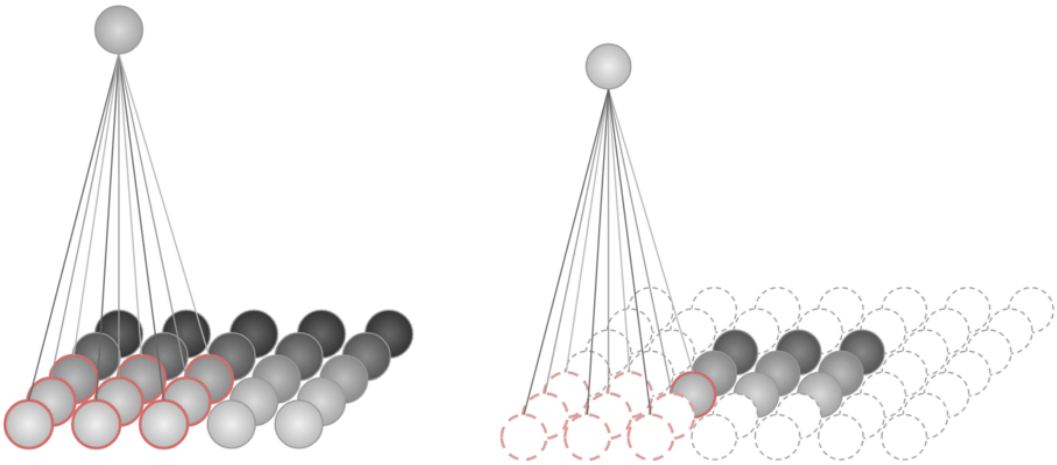
Every regular convolutional layer has an associated transposed convolution that reverses the dimensions of input and output, whilst preserving the connectivity pattern between the layers. This transposed convolution can always be interpreted as a regular convolution, with particular kernel size, padding and stride parameters.

In the case of a convolution with stride $s = 1$, kernel size k, padding p and input size i, recall the output size is $o = i + p - k + 1$. There is an associated transposed convolution with kernel size $k' = k$, stride $s' = s = 1$, and padding $p' = 2(k - 1) - p$. Its output size is given by

$$o' = i' + (k - 1) - p,$$

so with $i' = o$, we have $o' = i$, and the transposed convolution reverses the input and output dimensions.

The following animations show an example of a 3×3 convolution applied to an input $\mathbf{x} \in \mathbb{R}^{5 \times 5}$, with a stride of 1 and no padding. The associated transposed convolution has an input $\mathbf{y} \in \mathbb{R}^{3 \times 3}$, stride of one and padding of two zeros on either side of the input ("FULL" padding).



A regular convolution (left) and its associated transposed convolution (right). The input and output dimensions are reversed, but the connectivity pattern between neurons remains the same

In Keras, the transposed convolution is available as the `Conv2DTranspose` layer (also 1D and 3D variants).

In [26]:
```
# Create a transposed convolution layer

from keras.models import Sequential
from keras.layers import Input, Conv2DTranspose

cnn = Sequential([
    Input(shape=(3, 3, 1)),
    Conv2DTranspose(1, (3, 3), padding="VALID")
])
cnn.summary()
```

Model: "sequential_3"

Layer (type)	Output Shape	Param #
conv2d_transpose (Conv2DTranspose)	(None, 5, 5, 1)	10

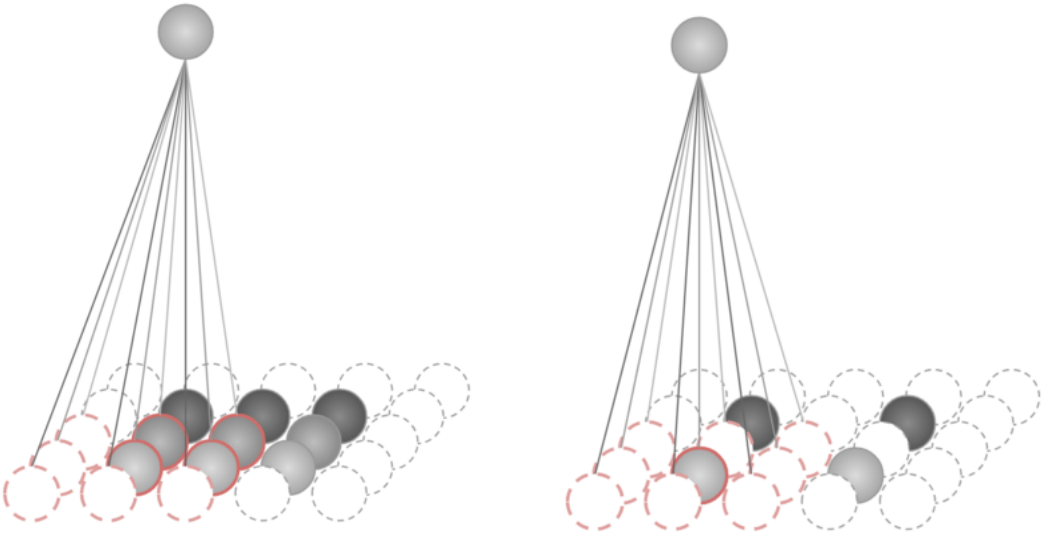
Total params: 10 (40.00 B)
Trainable params: 10 (40.00 B)
Non-trainable params: 0 (0.00 B)

For a regular convolution with stride $s > 1$, we can think of the associated transposed convolution as having a stride $s' < 1$ (another name for transposed convolutions is **fractionally strided convolutions**). In this case, the situation is more complicated. This is because for $s > 1$, different sized inputs can lead to the same size output.

Consider first the case where the regular convolution is such that s divides $(i + p - k)$. Recall that the output size is then $o = \left(\frac{i+p-k}{s}\right) + 1$. Then, the input to the associated transposed convolution adds $s - 1$ zeros between its input units. It has kernel size $k' = k$, $s' = 1$, and padding $p' = 2(k - 1) - p$. The output size is given by

$$o' = s(i' - 1) + k - p.$$

The following shows an example. An input $\mathbf{x} \in \mathbb{R}^{3\times3}$ is processed by a convolution with a 3×3 kernel, stride $s = 2$ and padding $p = 2$ ("SAME" padding). Its output size is 2×2. The associated transposed convolution takes an input $\mathbf{y} \in \mathbb{R}^{2\times2}$, inserts one zero in between the input units and padding $p' = 2$.



A regular convolution with stride $s = 2$ (left) and its associated transposed convolution (right). Note that $s \mid (i + p - k)$. The transposed convolution inserts one zero in between each input unit

In [31]:
```
# Create a transposed convolution layer with fractional stride

cnn = Sequential([
    Input(shape=(2, 2, 1)),
    Conv2DTranspose(1, (3, 3), strides=2, padding="SAME",
                    output_padding=1)
])
cnn.summary()
```

Model: "sequential_6"

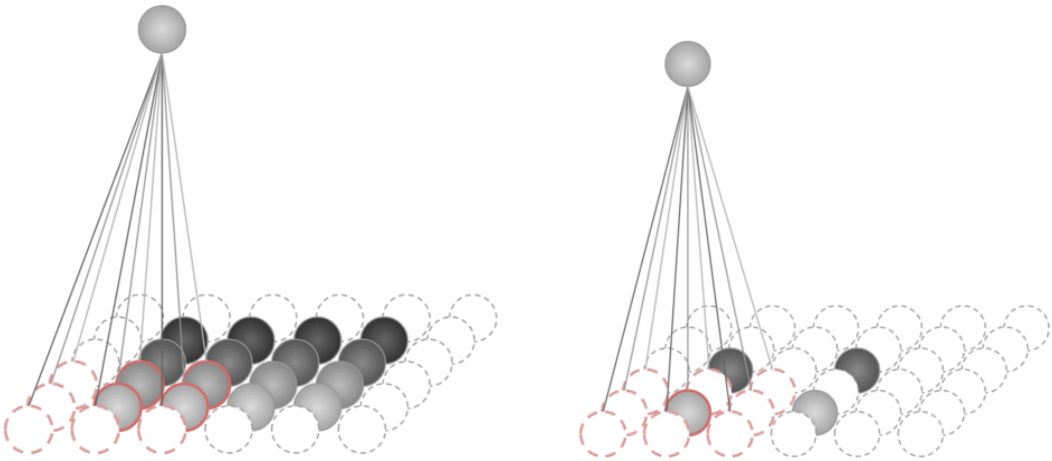
Layer (type)	Output Shape	Param #
conv2d_transpose_4 (Conv2DTranspose)	(None, 4, 4, 1)	10

Total params: 10 (40.00 B)
Trainable params: 10 (40.00 B)
Non-trainable params: 0 (0.00 B)

The case where s does not divide $(i + p - k)$ is accounted for by the parameter $a = (i + p - k) \mod s$. The transposed convolution again adds $s - 1$ zeros between input units, but also adds additional padding of a zeros, which then increases the dimension of its output. It has kernel size $k' = k$, $s' = 1$, padding $p' = 2(k - 1) - p + a$, and output size

$$o' = s(i' - 1) + a + k - p$$

The following shows hows the same output size as the previous example can be produced by the same convolution operation, for a different input size. The input is now $\mathbf{x} \in \mathbb{R}^{4\times4}$, with a 3×3 kernel, stride $s = 2$ and padding $p = 2$ as before. The associated transposed convolution again takes an input $\mathbf{y} \in \mathbb{R}^{2\times2}$, but this time produces an output with spatial dimensions 4×4.



A regular convolution with stride $s = 2$ where $s \nmid (i + p - k)$ (left) and its associated transposed convolution (right). An additional $a = (i + p + k) \bmod s = 1$ zero is added to the input of the transposed convolution

In [32]:
```
# Create a transposed convolution layer with fractional stride and additional outpu

cnn = Sequential([
    Input(shape=(2, 2, 1)),
    Conv2DTranspose(1, (3, 3), strides=2, padding="SAME",
                    output_padding=1)
])
cnn.summary()
```

Model: "sequential_7"

Layer (type)	Output Shape	Param #
conv2d_transpose_5 (Conv2DTranspose)	(None, 4, 4, 1)	10

Total params: 10 (40.00 B)
Trainable params: 10 (40.00 B)
Non-trainable params: 0 (0.00 B)

References

- Dumoulin, V. & Visin, F. (2016), "A guide to convolution arithmetic for deep learning", arXiv preprint, abs/1603.07285.
- Fukushima, K. (1980), "Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position", *Biological Cybernetics*, **3**6 (4), 193–202.
- Hubel, D. H. & Wiesel, T. N. (1959), "Receptive fields of single neurones in the cat's striate cortex", *Journal of Physiology* **148** (3), 574–91.

- LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989) "Backpropagation Applied to Handwritten Zip Code Recognition", AT&T Bell Laboratories.
- Rumelhart, D. E., Hinton, G., and Williams, R. (1986b), "Learning representations by back-propagating errors", Nature, **323**, 533-536.
- Simonyan, K. & Zisserman, A. (2015), "Very Deep Convolutional Networks for Large-Scale Image Recognition", in *3rd International Conference on Learning Representations, (ICLR) 2015*, San Diego, CA, USA.
- Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., Vezhnevets, A. S., Leblond, R., Pohlen, T., Dalibard, V., Budden, D., Sulsky, Y., Molloy, J., Paine, T. L., Gulcehre, C., Wang, Z., Pfaff, T., Wu, Y., Ring, R., Yogatama, D., Wünsch, D., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Kavukcuoglu, K., Hassabis, D., Apps, C., & Silver, D.(2019) "Grandmaster level in StarCraft II using multi-agent reinforcement learning", *Nature*, **575** (7782), 350-354.
- Zhou, Y. & Chellappa, R. (1988), "Computation of optical flow using a neural network", in *IEEE International Conference on Neural Networks*, IEEE, 71-78.

In [33]:
```
# Define a simple model with a Conv2D layer

model = Sequential([
    Input(shape=(None, None, 1)),
    Conv2D(1, (3, 3), activation=None, use_bias=False)
])
```

In [ ]: