Convolutional Neural Networks

- Convolutional Neural Networks (CNNs) are special type of NNs particularly suitable for images (e.g. 2D, 3D, 4D).
- Nowdays, CNNs are also the NN of choice for 1D sequential signals, such as audio, text, and time series.

Why CNNs

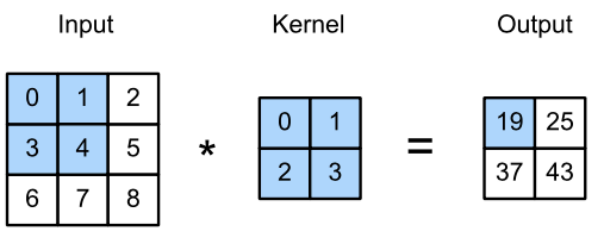
- So far we considered NNs comprised of Fully Connected layers.
- An FC layer is not suitable for images.
- E.g.: Assume a 2D grayscale image (1 channel) of resolution 256×256:
    - A single neuron connected to all pixels will require $256^2 = 65K$ params
        - If first layer: 128 of those neurons, in total $65 \times 128 \approx 9M$ params !

Convolutional Layer in a nutshell

- Divide the image into 3×3 windows.
- Apply the same FC layer to all windows.
- A single neuron will now have 3×3 params, and the whole layer $9 \times 128 \approx 1K$ params!

The Convolution Operation

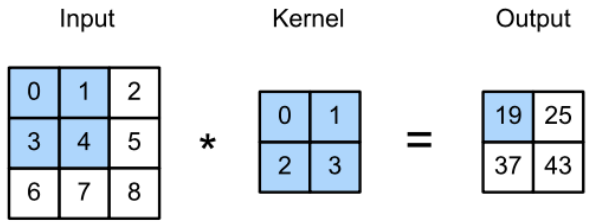
- Convolutional layers are a misnomer, since they are based on Cross-Correlation operations
- Abusing terminolodgy we will assume Convolution = Cross-Correlation.
- Example: Assume the input is a matrix (i.e. two-dimensional tensor) of size (shape) 3×3 or $(3, 3)$.
- Assume a neuron (also called *convolutional kernel*) of size 2×2.



- The output of the convolution operation between the input and the kernel is a matrix of 2×2.
- To calculate the first (top-left) output value, we position the kernel at at the top-left corner of the input tensor.
- Then we multiple element-wise the elements of the kernel with the correponding elements of the input, and then sum:

$$0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$$

The Convolution Operation



- To calculate the remaining output values: we slide the kernel across the input tensor, from left to right and top to bottom. Each time we repeat the same calculation. The remaining 3 outputs are given by:

$$1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25,$$
$$3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37,$$
$$4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43.$$

- Note that along each axis, the output size is slightly smaller than the input size. Because the kernel has width and height greater than one, we can only properly compute the convolution operation only for locations where the kernel fits wholly within the image.
- The output size is given by the input size $n_h \times n_w$ minus the size of the convolution kernel $k_h \times k_w$ via :

$$(n_h - k_h + 1) \times (n_w - k_w + 1).$$

Convolution as Feature Detection

Input

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

6 x 6

*

Feature Detector

1	0	-1
1	0	-1
1	0	-1

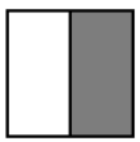
3 x 3

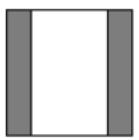
=

Feature Map

-0	30	30	0
0	30	30	0
0	30	30	0
0	30	30	0

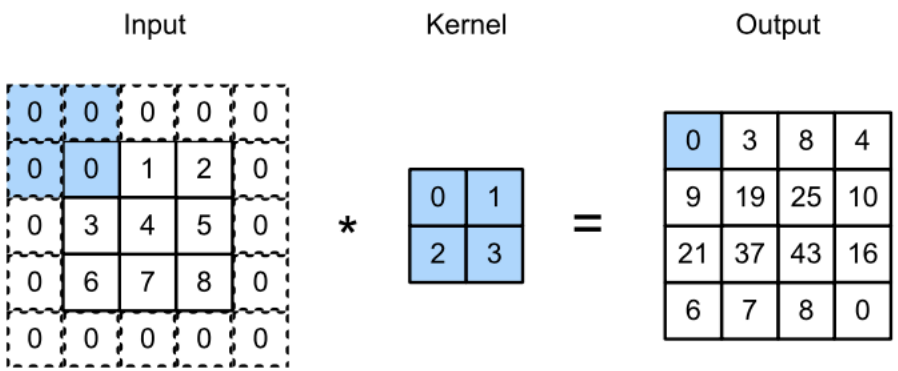
4 x 4





Padding

- As described above, one tricky issue when applying convolutional layers is that we tend to lose pixels on the perimeter of our image.
- Since we typically use small kernels, for any given convolution, we might only lose a few pixels, but this can add up as we apply many successive convolutional layers.
- One straightforward solution to this problem is to add extra pixels of filler around the boundary of our input image, thus increasing the effective size of the image.
- Typically, we set the values of the extra pixels to zero.
- In the example below we pad a 3×3 input, increasing its size to 5×5. The corresponding output then increases to a 4×4 matrix.



Padding

- In general, if we add a total of p_h rows of padding (roughly half on top and half on bottom) and a total of p_w columns of padding (roughly half on the left and half on the right), the output shape will be

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1).$$

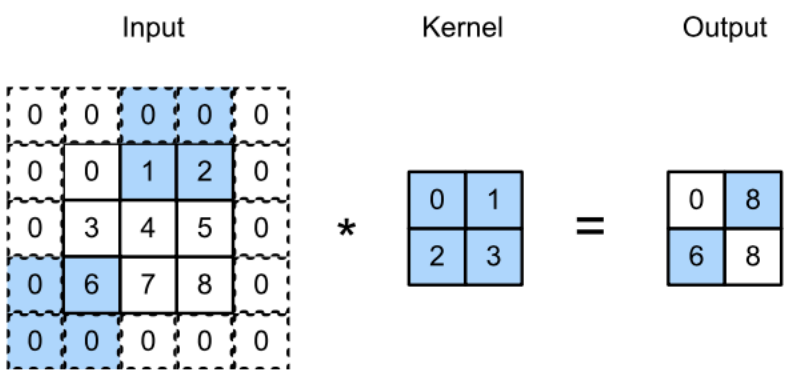
- This means that the height and width of the output will increase by p_h and p_w, respectively.

- In many cases, we will want to set $p_h = k_h - 1$ and $p_w = k_w - 1$ to give the input and output the same height and width.

- CNNs commonly use convolution kernels with odd height and width values, such as 1, 3, 5, or 7. Choosing odd kernel sizes has the benefit that we can preserve the spatial dimensionality while padding with the same number of rows on top and bottom, and the same number of columns on left and right.

Stride

- When computing the convolution, we start with the convolution window at the top-left corner of the input tensor, and then slide it over all locations both down and to the right.
- In previous examples, we default to sliding one element at a time.
- However, sometimes, because we wish to downsample, we move our window more than one element at a time, skipping the intermediate locations.
- We refer to the number of rows and columns traversed per slide as the *stride*.
- So far, we have used strides of 1, both for height and width.
- An example with stride 2:



Multiple Input and Multiple Output Channels

- Until now, we simplified all of our numerical examples by working with just a single input and a single output channel.

- This has allowed us to think of our inputs, convolution kernels, and outputs each as two-dimensional tensors.

- When we add channels into the mix, our inputs and outpus both become three-dimensional tensors.

- For example, each RGB input image has shape $3 \times h \times w$.
- We refer to this axis, with a size of 3, as the *channel* dimension.
- We will take a deeper look at convolution kernels with multiple input and multiple output channels.

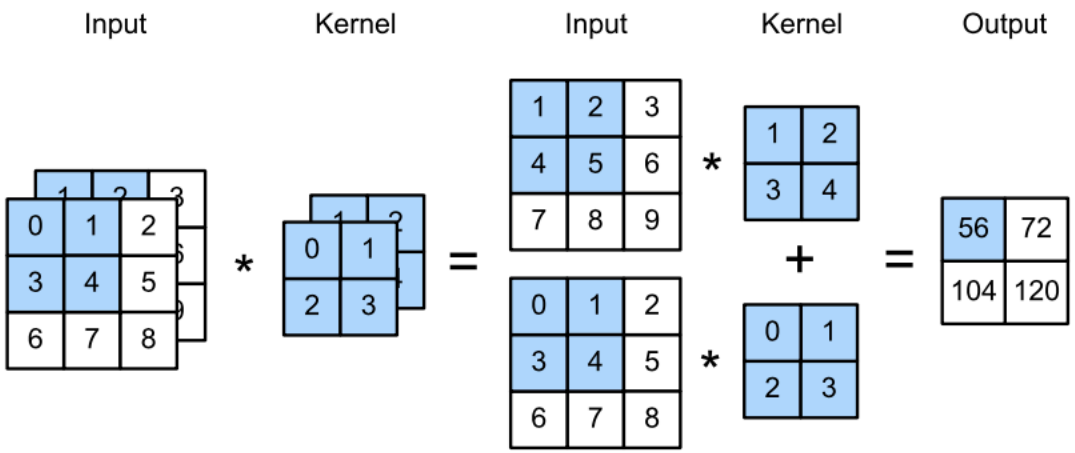
Multiple Input Channels

- When the input data contain multiple channels, we need to construct a convolution kernel with the same number of input channels as the input data
- Assuming that the number of channels for the input data is c_i, the number of input channels of the convolution kernel also needs to be c_i.
- If our convolution kernel's window shape is $k_h \times k_w$, then when $c_i = 1$, we can think of our convolution kernel as just a two-dimensional tensor of shape $k_h \times k_w$.

- However, when $c_i > 1$, we need a kernel that contains a tensor of shape $k_h \times k_w$ for *every* input channel. Concatenating these c_i tensors together yields a convolution kernel of shape $c_i \times k_h \times k_w$.
- Since the input and convolution kernel each have c_i channels, we can perform a cross-correlation operation on the two-dimensional tensor of the input and the two-dimensional tensor of the convolution kernel for each channel, adding the c_i results together (summing over the channels) to yield a two-dimensional tensor.
- This is the result of a two-dimensional convolution between a multi-channel input and a multi-input-channel convolution kernel.

Multiple Input Channels

- An example of a two-dimensional convolution with two input channels:



- The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation:
$(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56.$

Multiple Output Channels

- Regardless of the number of input channels, so far we always ended up with one output channel.
- However, it turns out to be essential to have multiple channels at each layer.
- In the most popular neural network architectures, we actually increase the channel dimension as we go deeper in the neural network, typically downsampling to trade off spatial resolution for greater *channel depth*.
- Intuitively, you could think of each channel as responding to some different set of features.
    - In reality features are not learned independent but are optimized to be jointly useful.
        - E.g.: it may not be that a single channel learns an edge detector

- Denote by c_i and c_o the number of input and output channels, respectively, and let k_h and k_w be the height and width of the kernel.
- To get an output with multiple channels, we repeat the single output case c_o times with c_o different kernels and then concatenate the c_o different outputs.
    - We create a kernel tensor of shape $c_i \times k_h \times k_w$ for *every* output channel.
- All c_o different kernels can be put together, in a 4D tensor of size $c_o \times c_i \times k_h \times k_w$.

An example

- Input layer has spatial resolution 26x26 and 128 channels.
- Output layer has spatial resolution 26x26 and 192 channels.
- $k_h = k_w = 3$.
- What are the dimensions of each of the convolutional kernels?
- How many of them?
- What are the dimensions of the weight tensor?

In [29]:
```
import torch
import torch.nn as nn
```

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, bias=True)
```

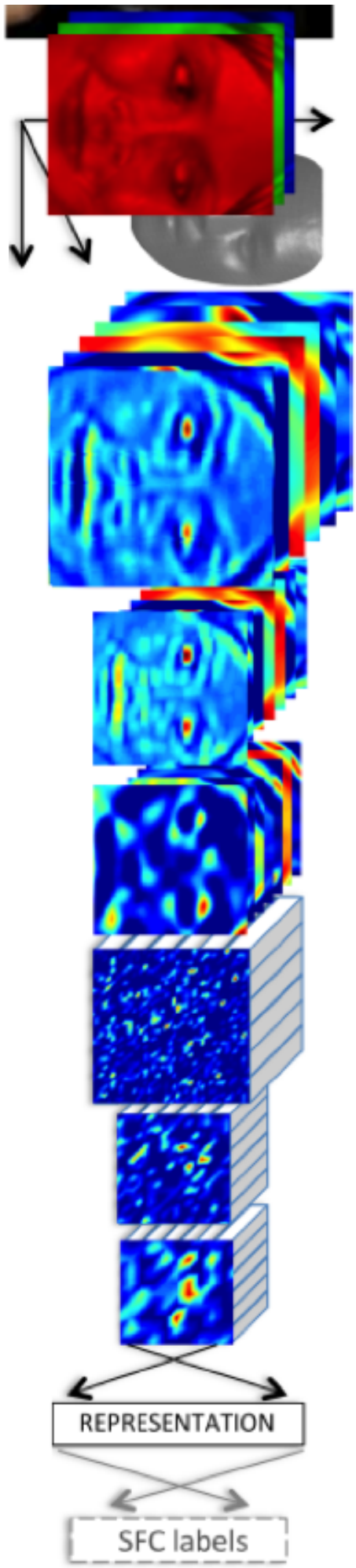
In [30]:
```
inp = torch.rand(16, 128, 26, 26)
print(inp.size())
```

```
torch.Size([16, 128, 26, 26])
```

In [31]:
```
my_conv = nn.Conv2d(128, 192, 1, 1)
```

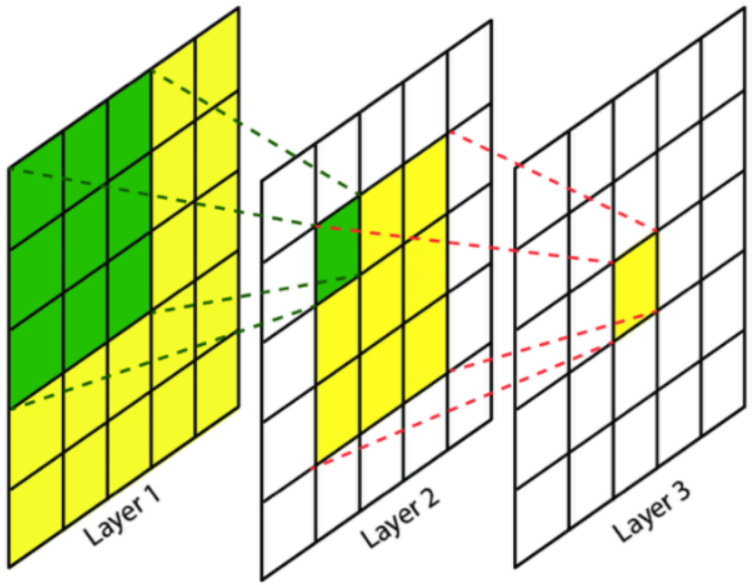
In [32]:
```
out = my_conv(inp)
print(out.size())
```

```
torch.Size([16, 192, 26, 26])
```


REPRESENTATION
SFC labels

Receptive field

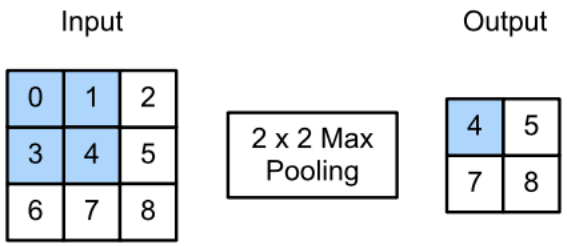
- The receptive field refers to the area that is used to calculate the output of a convolution.
- By stacking many conv. layers the effective receptive field increases.
    - This is why we need models with **many Conv. Layers**!



- The receptive field of each convolution layer with a 3x3 kernel. The green area marks the receptive field of one pixel in Layer 2, and the yellow area marks the receptive field of one pixel in Layer 3.

Pooling

- **Maximum pooling** and **average pooling**:
  - a fixed-shape window is slid over all regions of the input tensor
  - At each location that the pooling window hits, it computes the maximum or average value of the input subtensor in the window,



- The output tensor has a height of 2 and a width of 2. The four elements are derived from the maximum value in each pooling window:

$$\max(0, 1, 3, 4) = 4,$$
$$\max(1, 2, 4, 5) = 5,$$
$$\max(3, 4, 6, 7) = 7,$$
$$\max(4, 5, 7, 8) = 8.$$

Why pooling?

- With pooling we gradually reduce the spatial resolution of our features, aggregating information so that the deeper we go in the network, the larger the receptive field (in the input) to which each hidden node is sensitive.
- Often our ultimate task asks some global question about the image, e.g., *does it contain a cat?*
    - So typically the units of our final layer should be sensitive to the entire input.
    - By gradually aggregating information, yielding coarser and coarser maps, we accomplish this goal of ultimately learning a global representation

```
torch.nn.AvgPool2d(kernel_size, stride=None, padding=0)
```

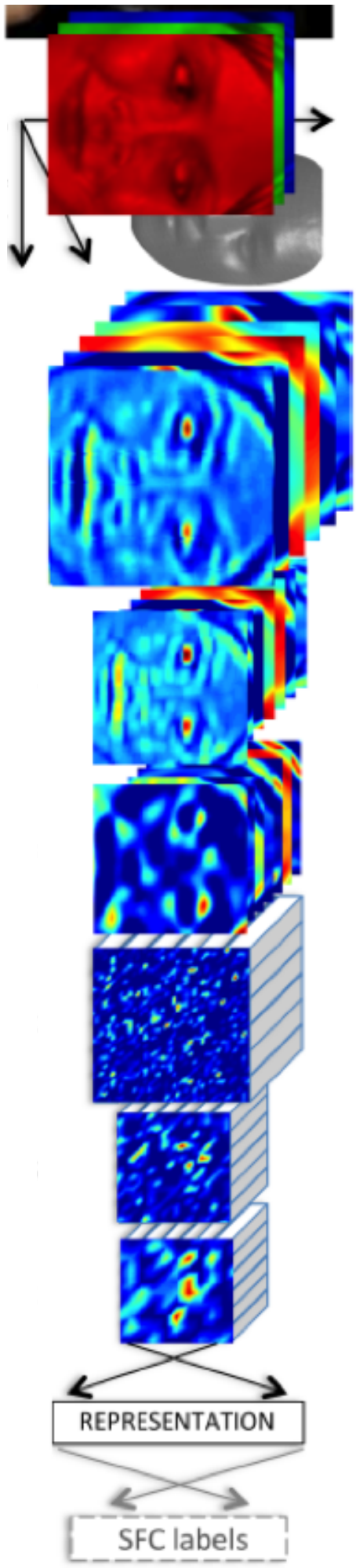
In [35]:
```
my_pool = torch.nn.AvgPool2d(3, 2, 1)
```

In [38]:
```
print(out.size())
```

```
torch.Size([16, 192, 26, 26])
```

In [39]:
```
out1 = my_pool(out)
print(out1.size())
```

```
torch.Size([16, 192, 13, 13])
```


REPRESENTATION
SFC labels

Summary

- Convolutional filters are like small neurons used to scan the image.
- A convolutional filter has as many channels as the number of channels of the input feature map.
- The number of channels of the output feature map is equal to the number of convolutional filters of the convolutional layer
- With pooling we gradually reduce the spatial resolution of our features