Machine Learning

Convolutional Neural Networks

Gabriele Russo Russo   Francesco Lo Presti

Laurea Magistrale in Ingegneria Informatica – A.Y. 2023/24

TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Dipartimento di Ingegneria Civile e Ingegneria Informatica

Motivation

- Recall the "Fashion MINST" example, where we classified images using a NN
- Image = 2D grid of pixels
- We ignored this structure and *flattened* images into vectors
  - We could even shuffle the vector elements, as the NN was invariant to feature order
- How to leverage prior knowledge that nearby pixels are typically related to each other?

Motivation (2)

- Example: we are asked to classify images of cats and dogs
- Consider a dataset of 1-megapixel labeled photographs
  - Each input instance has 1 million pixels
- Even an aggressive reduction to 1,000 hidden units would require a fully connected layer with $10^6 \times 10^3 = 10^9$ parameters
  - Training would be extremely difficult or even unfeasible

Convolutional Neural Networks (CNN)

LeCun et al., “Backpropagation applied to handwritten zip code recognition”. Neural Computation (1989)

- Convolutional Neural Networks (CNN) are specialized NNs for processing data that has a known grid-like topology, e.g.:
  - time-series data (1D topology)
  - images (2D grids of pixels)
- “Convolutional” because of the convolution operation used in the NN
- A CNN is a neural network that uses convolution in at least one layer

Convolution

Given two real-valued functions $x(t)$ and $w(t)$, their convolution is defined as

$$s(t) = (x * w)(t) = \int_{-\infty}^{\infty} x(u)w(t-u)du$$

Instead, when working with discrete data:

$$s(t) = (x * w)(t) = \sum_{i=-\infty}^{\infty} x(i)w(t-i)$$

Convolution: Example

- Tracking the location of a spaceship with a laser sensor
- Sensor provides the position $x(t)$ at time t
- Sensor is noisy, so we want to average over several measurements
- A function $w(a)$ weighs the importance of a measurement given its age a

$$s(t) = (x * w)(t) = \int_{-\infty}^{\infty} x(u)w(t-u)du$$

Convolution with Tensors

$$s(t) = (x * w)(t) = \sum_{i=-\infty}^{\infty} x(i)w(t-i)$$

- In CNNs, we work with multidimensional arrays (tensors)
- Convolution arguments are tensors indicated, respectively, as input and kernel (or, feature map)
- E.g., for a 2D image $\boldsymbol{X}$ as input, we use a 2D kernel $\boldsymbol{K}$

$$(X * K)(i,j) = \sum_a \sum_b X(a,b)K(i-a, j-b)$$

Basic Convolutional Layer

- Let's use convolution to construct a hidden layer
- A convolutional layer takes a tensor $\boldsymbol{X}$ in input and produces an output tensor $\boldsymbol{H}$
- $\boldsymbol{H}$ computed by calculating convolution $\forall(i, j)$
  - We are skipping many details at this point, including the use of a nonlinear activation function
- The kernel $\boldsymbol{K}$ comprises the parameters of the layer (to be trained)

Convolution vs. Cross-Correlation

- Convolution is commutative because we flip the kernel relative to the input

$$\sum_a \sum_b X(a,b)K(i-a,j-b) = \sum_a \sum_b K(a,b)X(i-a,j-b)$$

- Most the ML libraries use the term "convolution" but actually implement cross-correlation

$$S(i,j) = \sum_a \sum_b K(a,b)X(i+a,j+b)$$

- Besides losing commutative property, no impact in practice because we are going to *learn* the kernel (flipped or not)

Example (with cross-correlation)

Input I

Kernel K

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 5 & 0 & 0 \\ 7 & 8 & 9 & 1 \end{bmatrix}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

$$(K * I)(1,1) = a + 2b + 2c + 5d$$

$$(K * I)(1,2) = 2a + 3b + 5c$$

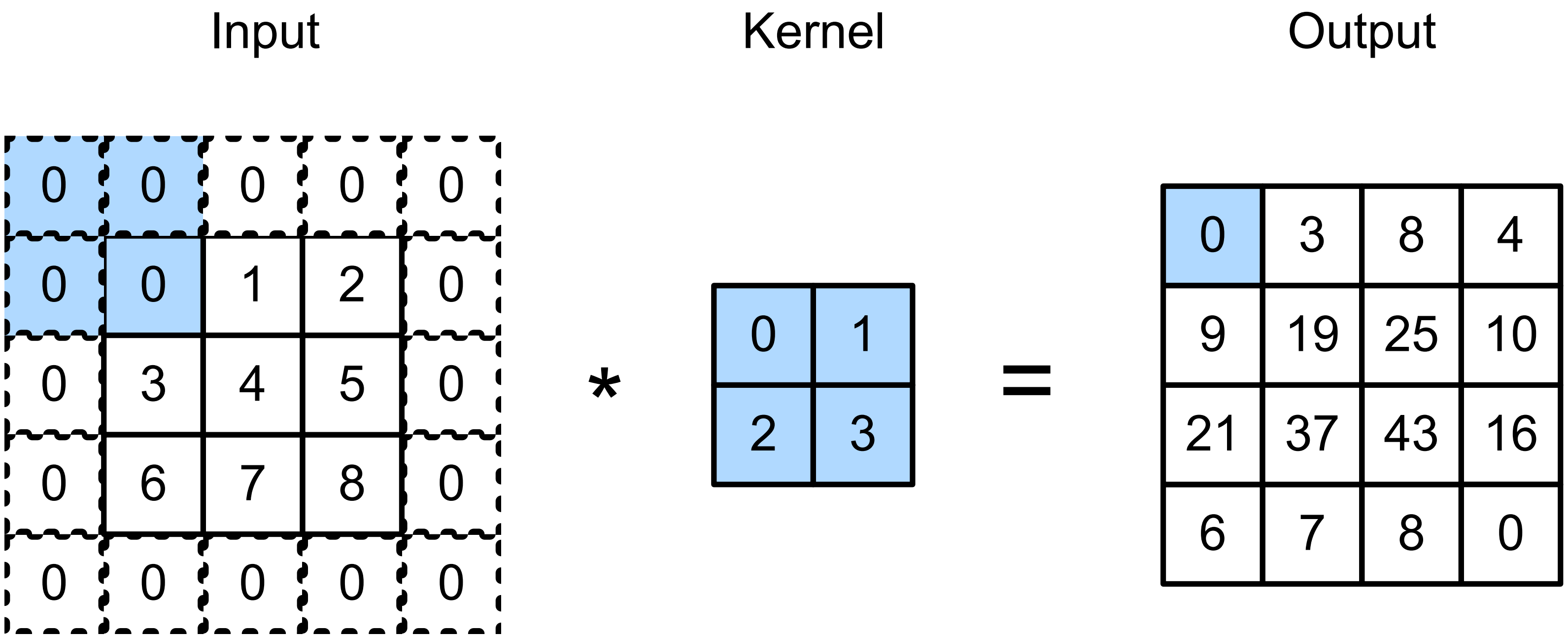
Example

Input X	**Kernel K**	$\boldsymbol{H}$
$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$	$\begin{bmatrix} 19 & 25 \\ 37 & 43 \end{bmatrix}$

- We started with a 3x3 input and got a 2x2 output
- What if we want to stack N convolutional layers??

Zero Padding

- To solve this problem, we can add extra pixels of filler around the boundary of our input (padding)
- Typically, we set the values of the extra pixels to zero



Zero Padding (2)

- With input of size $n_h \times n_w$ and kernel of size $k_h \times k_w$, output has shape $(n_h - k_h + 1) \times (n_w - k_w + 1)$
- Adding p_h rows and p_w columns of padding, we get: $(n_h + p_h - k_h + 1) \times (n_w + p_w - k_w + 1)$
- To give input and output the same size, usually we set $p_h = k_h - 1$ and $p_w = k_w - 1$
- CNN kernels usually have odd width and height
  - If k_h is odd, we exactly add $\frac{p_h}{2}$ rows both on top and bottom (similarly for columns)
  - As an additional benefit, the output element $\boldsymbol{H}_{ij}$ is calculated with the convolution window centered on $\boldsymbol{X}_{ij}$

Example (with padding)

Input (3x4)

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 5 & 0 & 0 \\ 7 & 8 & 9 & 1 \end{bmatrix}$$

With Padding

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 2 & 3 & 4 & 0 \\ 0 & 2 & 5 & 0 & 0 & 0 \\ 0 & 7 & 8 & 9 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Kernel K

$$\begin{bmatrix} a & b & c \\ c & a & b \\ e & c & f \end{bmatrix}$$

$$(K * I)(1,1) = \ldots$$

Why Convolution?

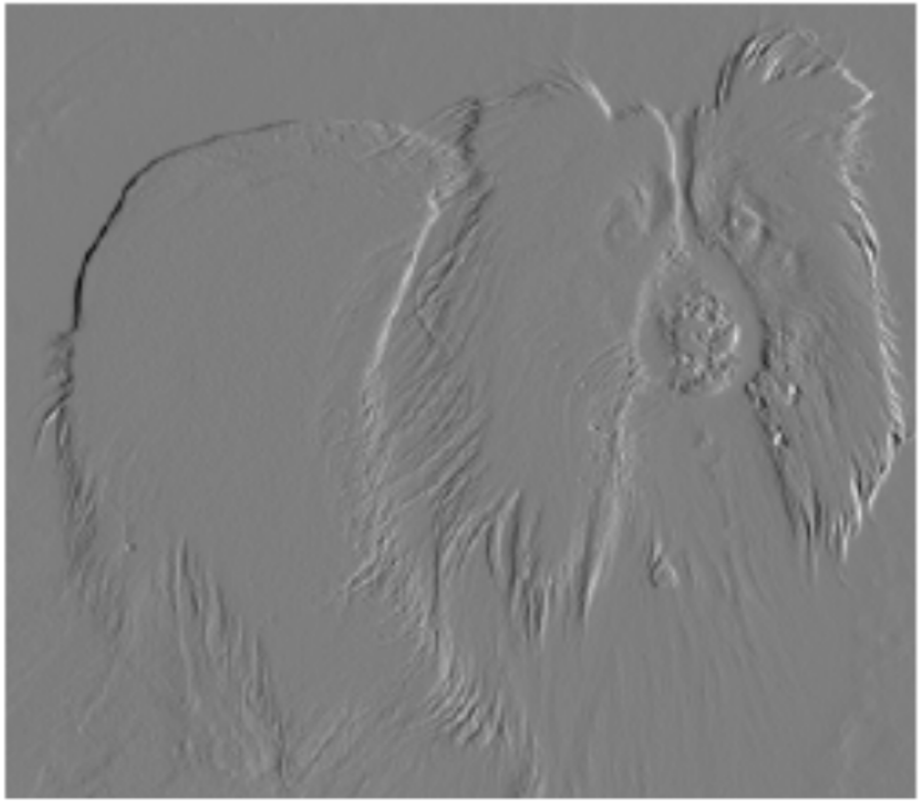
Convolution leverages 3 important ideas that can help ML:

- sparse interactions
- parameter sharing
- equivariant representations

Sparse Interactions

- In traditional NN layers, every output unit possibly interacts with every input unit
- In CNNs, by making the kernel smaller than the input we need to store fewer parameters and improve the overall efficiency of the model
  - we can detect small, meaningful features (e.g., edges) with kernels that occupy only tens or hundreds of pixels

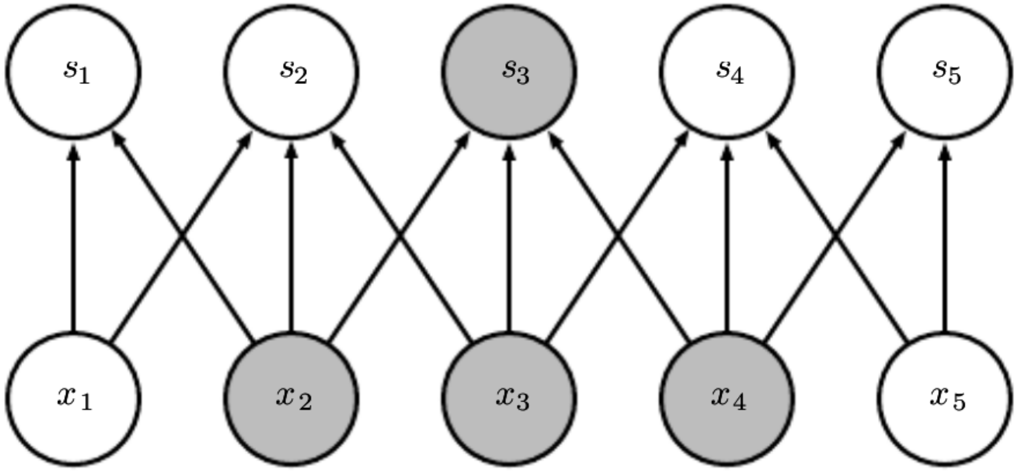
Example: Detecting Vertical Edges

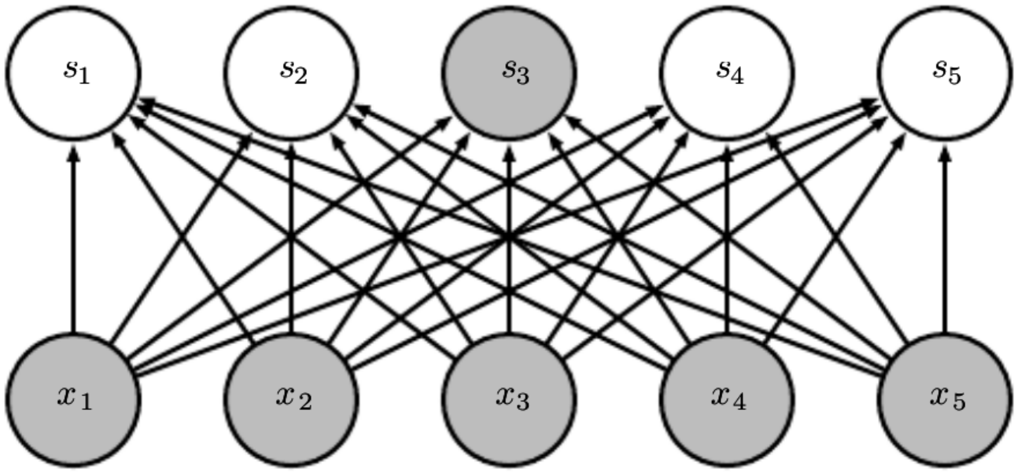
A 2-pixel kernel is enough: $K = [1, -1]$

Receptive Field

We highlight the receptive field of one unit, that is the input units affecting this output unit.

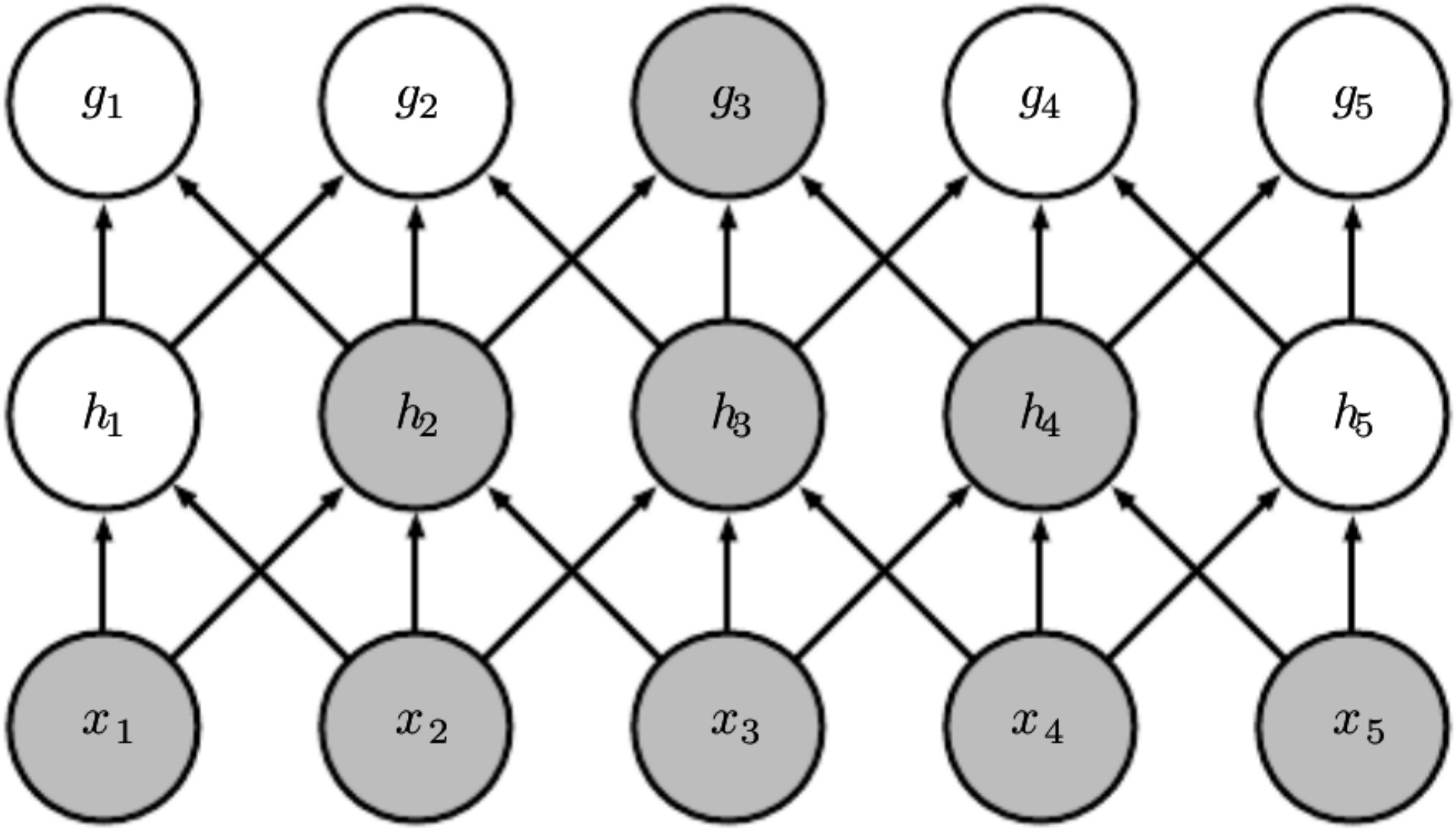


Convolutional
(3-pixel kernel)



Fully Connected

Receptive Field with More Layers



Even though *direct* connections are very sparse, units in the deeper layers can be *indirectly* connected to all or most of the input image.

Parameter Sharing

- In a traditional NN, each element of the weight matrix is used exactly once when computing the output of a layer
- In CNNs, the same elements of the kernel are used at every position of the input
  - the same parameters are shared across output units
- While the complexity of backpropagation does not change, we have important benefits
- Reduced memory requirement!
- Reduced number of parameters to learn!

Equivariant Representation

- Parameter sharing (as defined above) causes the layer to have a property called equivariance to translation
- e.g., if we move an object in the input, its representation will move the same amount in the output
- Clearly, convolution is not naturally equivariant to some other transformations, such as changes in the scale or rotation of an image

Convolution as Matrix Multiplication

- Convolution (and cross-correlation) can be regarded as traditional matrix multiplication
- The matrices to multiply must be constructed from the original input and kernel
- For instance, consider the following input A and kernel K (without padding):

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \qquad K = \begin{bmatrix} k_{1,1} & k_{1,2} \\ k_{2,1} & k_{2,2} \end{bmatrix}$$

We construct a matrix M from K and a vector v from A

$$M = \begin{bmatrix} k_{1,1} & k_{1,2} & 0 & k_{2,1} & k_{2,2} & 0 & 0 & 0 & 0 \\ 0 & k_{1,1} & k_{1,2} & 0 & k_{2,1} & k_{2,2} & 0 & 0 & 0 \\ 0 & 0 & 0 & k_{1,1} & k_{1,2} & 0 & k_{2,1} & k_{2,2} & 0 \\ 0 & 0 & 0 & 0 & k_{1,1} & k_{1,2} & 0 & k_{2,1} & k_{2,2} \end{bmatrix}$$

$$v = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{2,1} & a_{2,2} & a_{2,3} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$$

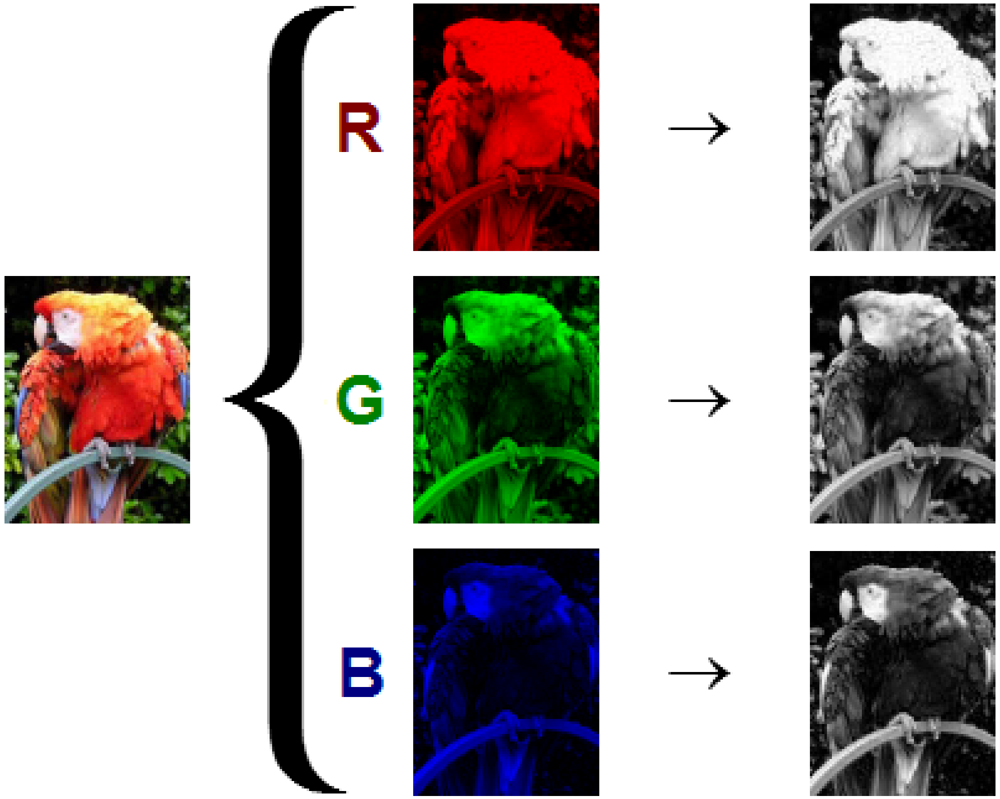
$$Mv^T = \begin{bmatrix} a_{1,1}k_{1,1} + a_{1,2}k_{1,2} + a_{1,3} \cdot 0 + a_{2,1}k_{2,1} + a_{2,2}k_{2,2} + a_{2,3} \cdot 0 + a_{3,1} \cdot 0 + a_{3,2} \cdot 0 + a_{3,3} \cdot 0 \\ a_{1,1} \cdot 0 + a_{1,2}k_{1,1} + a_{1,3}k_{1,2} + a_{2,1} \cdot 0 + a_{2,2}k_{2,1} + a_{2,3}k_{2,2} + a_{3,1} \cdot 0 + a_{3,2} \cdot 0 + a_{3,3} \cdot 0 \\ a_{1,1} \cdot 0 + a_{1,2} \cdot 0 + a_{1,3} \cdot 0 + a_{2,1}k_{1,1} + a_{2,2}k_{1,2} + a_{2,3} \cdot 0 + a_{3,1}k_{2,1} + a_{3,2}k_{2,2} + a_{2,3} \cdot 0 \\ a_{1,1} \cdot 0 + a_{1,2} \cdot 0 + a_{1,3} \cdot 0 + a_{2,1} \cdot 0 + a_{2,2}k_{1,1} + a_{2,3}k_{1,2} + a_{3,1} \cdot 0 + a_{3,2}k_{2,1} + a_{3,3}k_{2,2} \end{bmatrix}$$

Reshaping into a 2x2 matrix we get the result:

$$\begin{bmatrix} a_{1,1}k_{1,1} + a_{1,2}k_{1,2} + a_{2,1}k_{2,1} + a_{2,2}k_{2,2} & a_{1,2}k_{1,1} + a_{1,3}k_{1,2} + a_{2,2}k_{2,1} + a_{2,3}k_{2,2} \\ a_{2,1}k_{1,1} + a_{2,2}k_{1,2} + a_{3,1}k_{2,1} + a_{3,2}k_{2,2} & a_{2,2}k_{1,1} + a_{2,3}k_{1,2} + a_{3,2}k_{2,1} + a_{3,3}k_{2,2} \end{bmatrix}$$

Channels

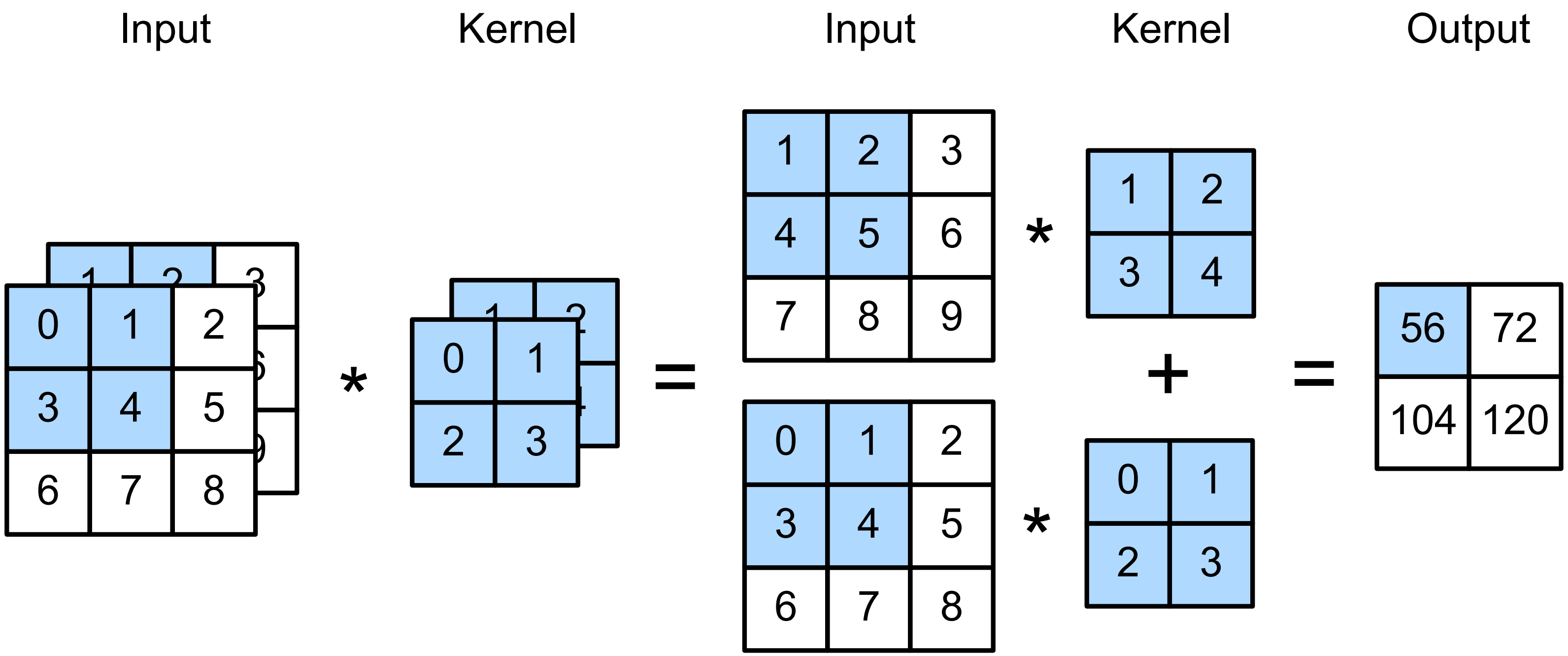
- Images usually consist of multiple channels, usually three: red, green and blue (RGB)
- Represented as 3D tensors with shape: $3 \times h \times w$



Multiple Input Channels

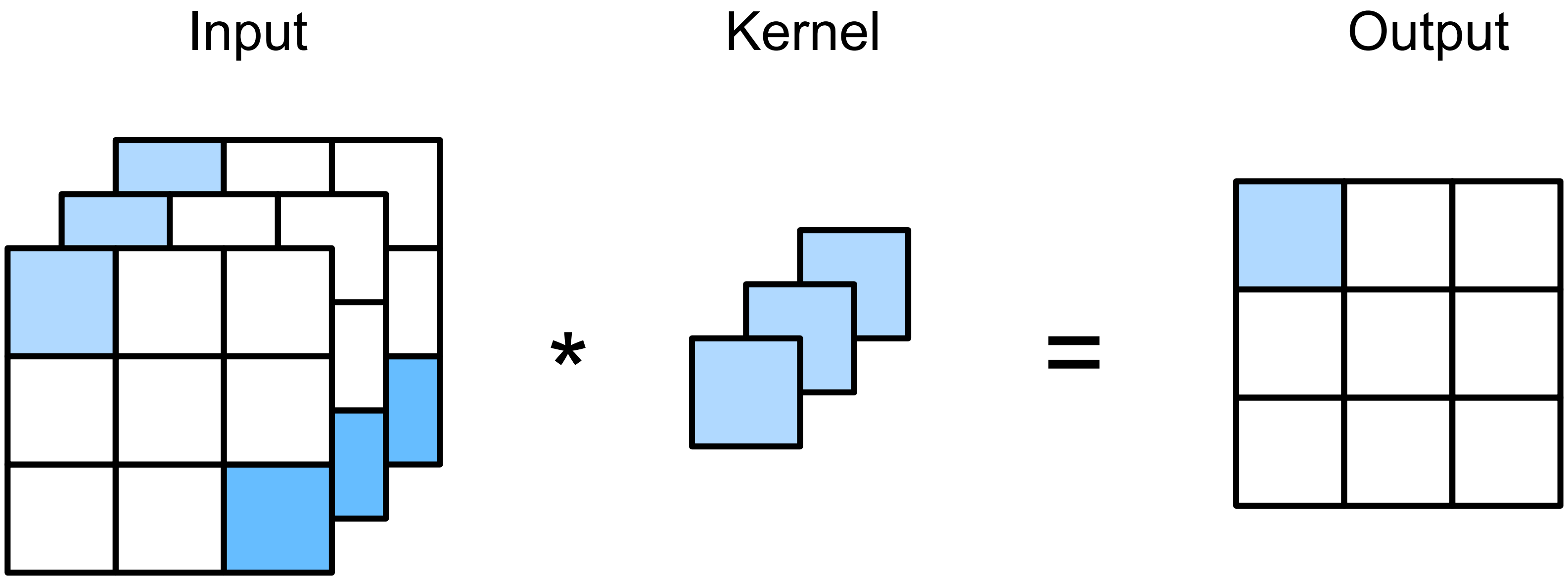
With $c_i > 1$ input channels, we need a 3D kernel as well (but we still get a 2D output tensor)

$$H_{i,j} = \sum_a \sum_b \sum_c K_{a,b,c} X_{i+a,j+b,c}$$



Example: 1x1 kernel

- Apparently, a 1×1 kernel does not make much sense, as it cannot correlate any pixels!
- With multi-channel input, a $1 \times 1 \times c_i$ kernel can be actually useful
  - e.g., to average the value of a pixel across the channels



Multiple Output Channels

- Regardless of the number of input channels, so far we always ended up with 1 output channel
- In practice, a convolutional layer applies multiple kernels at the same time, each leading to an output channel
  - Different kernels can discover different things in the input
- Having c_o kernels with shape $k_h \times k_w \times c_i$, equivalent to a single kernel with shape $k_h \times k_w \times c_i \times c_o$

$$H_{i,j,k} = \sum_a \sum_b \sum_c K_{a,b,c,k} X_{i+a,j+b,c}$$

Convolutional Layer: Revisited

- Tensor $\boldsymbol{X}$ in input, with shape $n_h \times n_w \times c_i$
- Kernel $\boldsymbol{K}$ with shape $k_h \times k_w \times c_i \times c_o$ comprises the parameters of the layer
  - Plus a bias term for every kernel
- Output tensor $\boldsymbol{H}$ with shape $n_h \times n_w \times c_o$
  - Assuming zero padding is used as described above
- $\boldsymbol{H}$ computed by calculating convolution with $\boldsymbol{K}$ at every input position, and possibly applying a nonlinear activation function (e.g., ReLU)

$$H_{i,j,k} = \phi\left(\sum_{u,v,c} K_{u,v,c,k} X_{i+u,j+v,c} + b_k\right)$$

where b_k is the bias term for the k-th output channel.

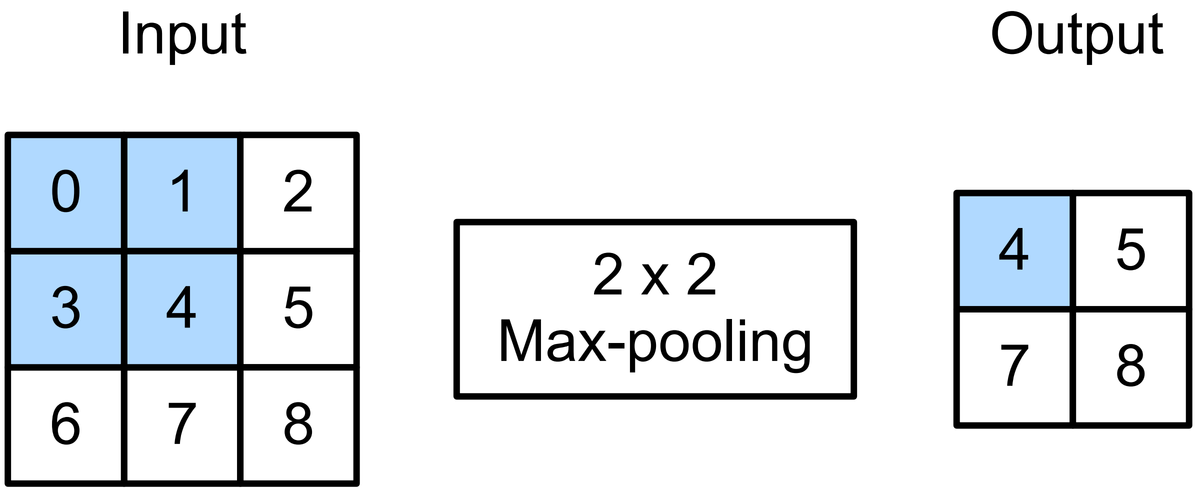
Convolution with Stride

- When computing cross-correlation, the convolution window starts from the upper-left corner of the input, and then slides over all locations down and to the right
  - So far, we defaulted to sliding one element at a time
- Sometimes we prefer to move the window more than one element at a time, skipping intermediate locations
  - Computational efficiency
  - Downsampling the input image
- Stride = number of rows and columns traversed per slide
  - e.g., with stride 2, we halve both width and height of the image

Pooling

- Convolutional layers are usually followed by pooling layers
  - Some books just consider convolution and pooling distinct steps of the same "complex" layer
- Pooling operator: fixed-shape window that is slid over all regions in the input according to its stride, computing a single output for each location traversed
- Similar to convolution, but with no parameters!
- Pooling operators are deterministic, typically calculating either the maximum or the average value of the elements in the pooling window
  - "max pooling" and "average pooling" (for short)

Example



Pooling (2)

- In presence of multiple channels, pooling is applied to each channel separately
  - Output tensor has the same number of channels as the input
- Since pooling aggregates information from an area, it is frequent to match pooling window size and stride
  - with a 3×3 window, stride is set to 3