Convolutional Neural Networks (CNN)

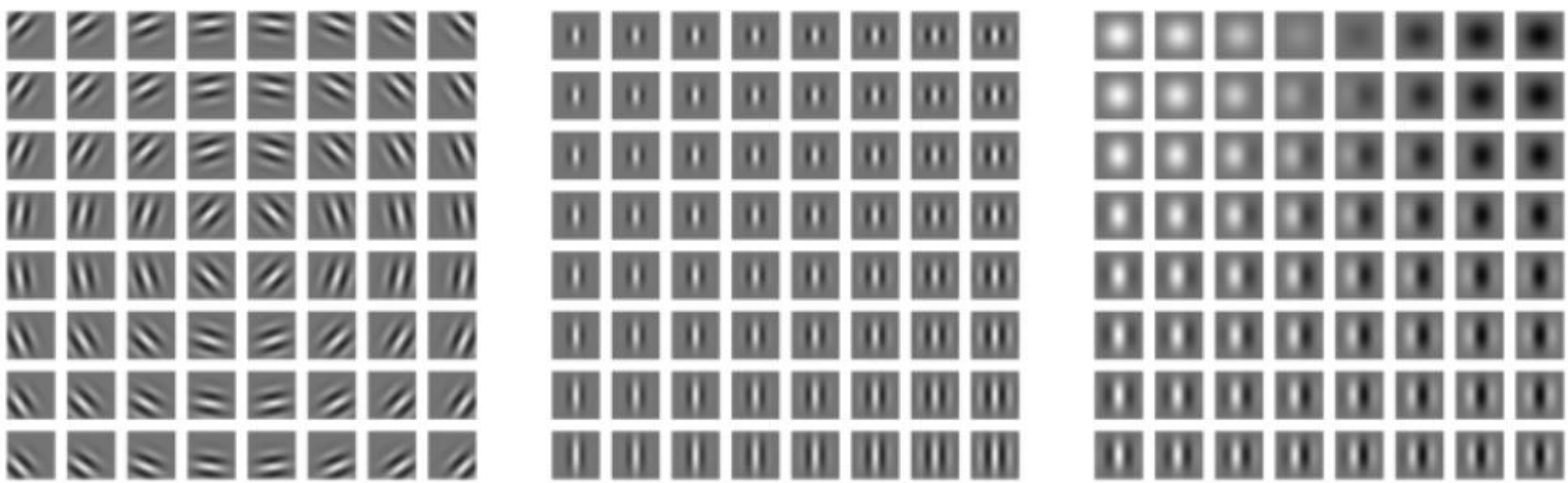
David I. Inouye

Why convolutional networks?

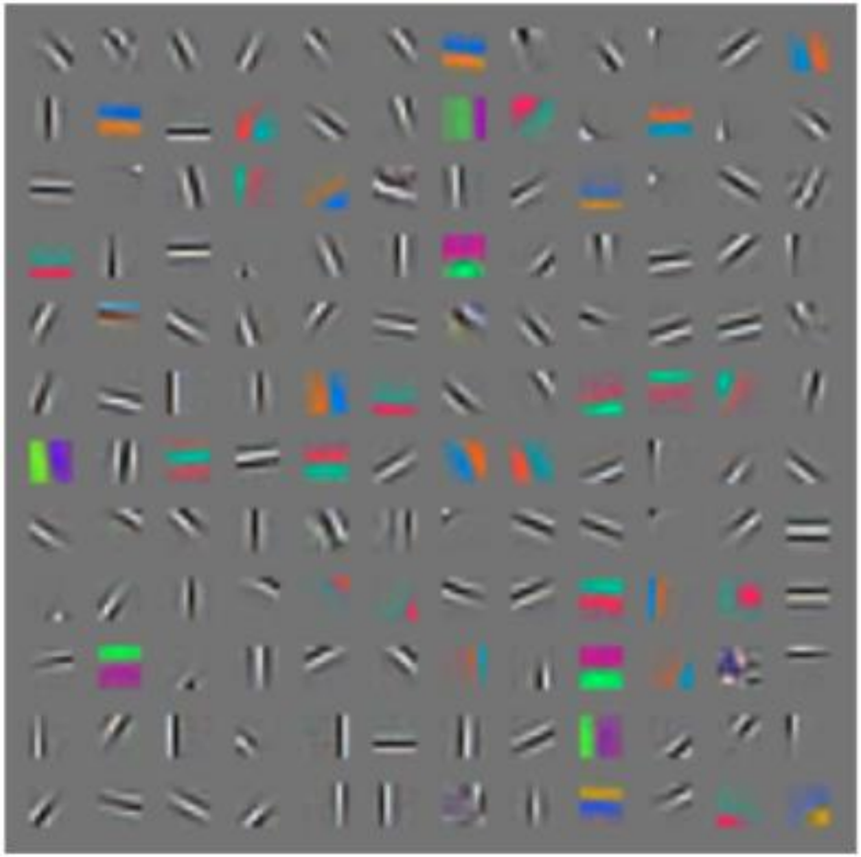
- Neuroscientific inspiration

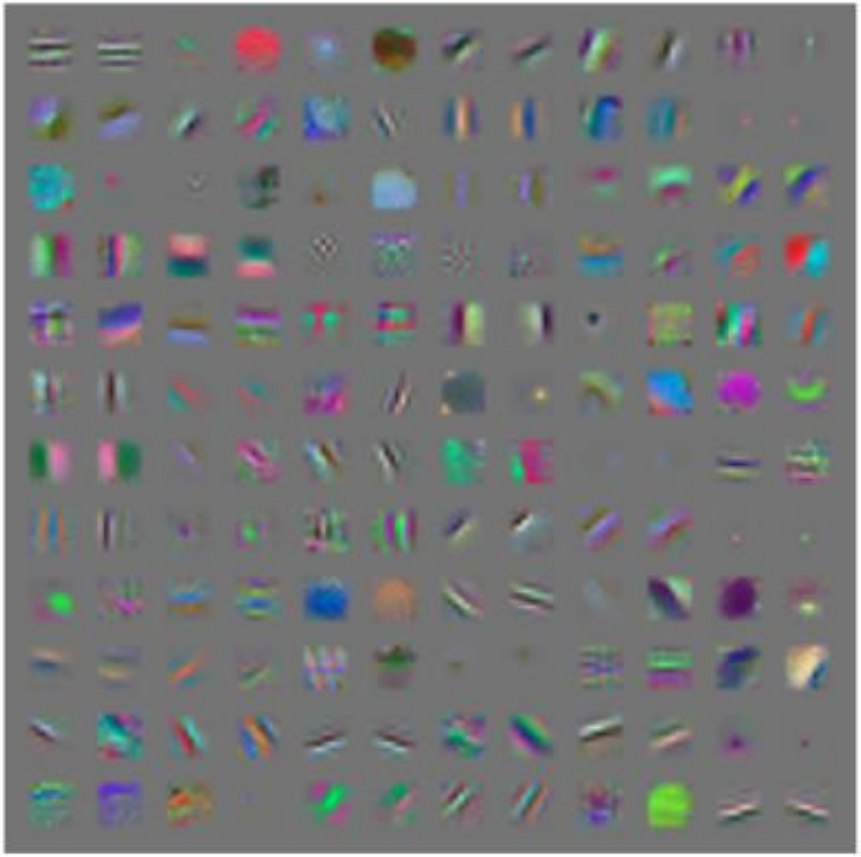
- Computational reasons
  - Sparse computation (compared to full deep networks)
  - Shared parameters (only a small number of shared parameters)
  - Translation invariance

Motivation for convolution networks: Gabor functions derived from neuroscience experiments are simple convolutional filters [DL, ch. 9]



Convolutional networks automatically learn filters similar to Gabor functions [DL, ch. 9]





1D convolutions are similar but slightly different than signal processing / math convolutions

x	1	2	3	2	5	1

f	1	2

y	5	8	7	12	7

Padding or stride parameters alter the computation and output shape

x

1	2	3	2	5	1

f

1	2

Stride of 2

y

5	7	7

1D convolutions are similar but slightly different than signal processing / math convolutions

x	1	2	3	2	5	1

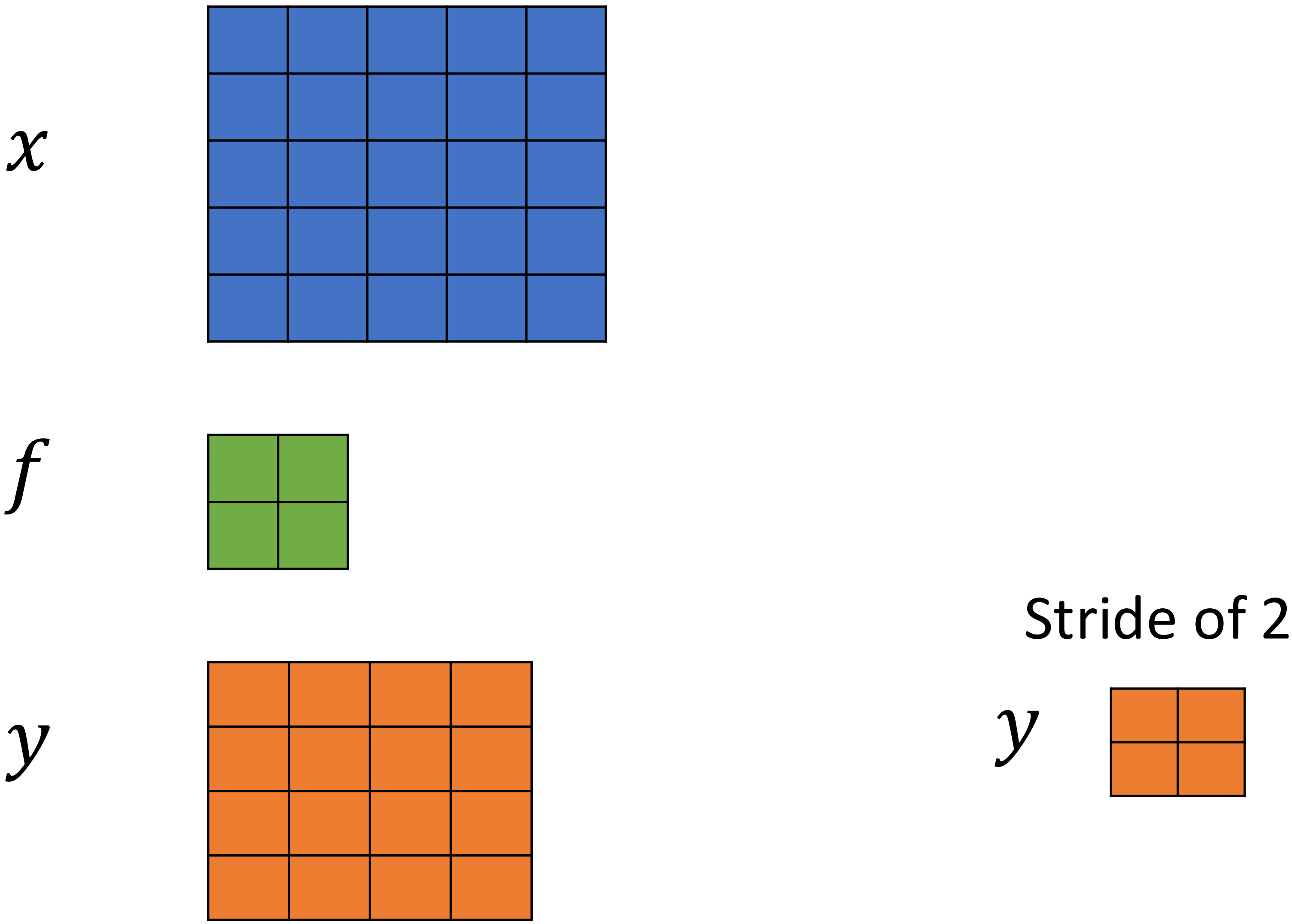
f	1	2

Zero padding of 1

y	2	5	8	7	12	7	1

Switch to demo of 1D

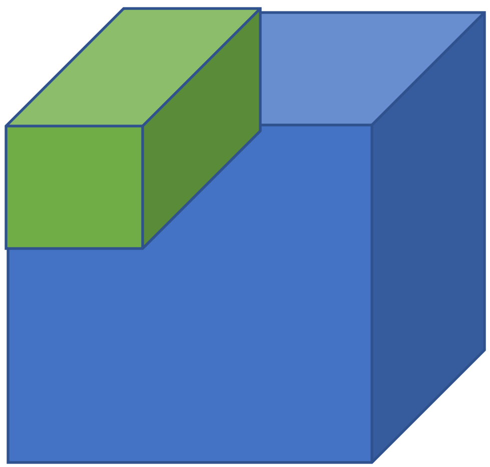
2D convolutions are simple generalizations to matrices



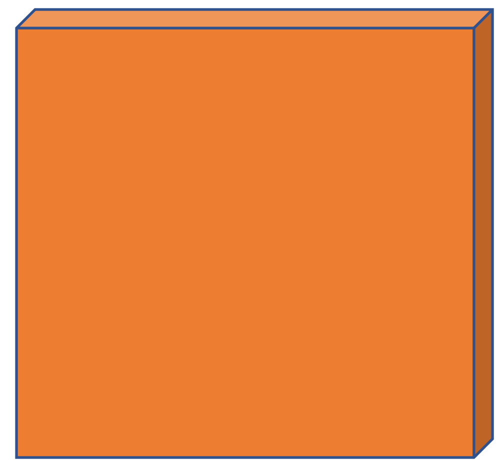
Switch to demo of 2D

2D convolutions ***with channels*** are like simple 2D convolutions but all arrays have a channel dimension

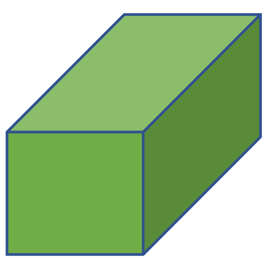
$x \in \mathcal{R}^{c \times h \times w}$



$y \in \mathcal{R}^{1 \times h' \times w'}$

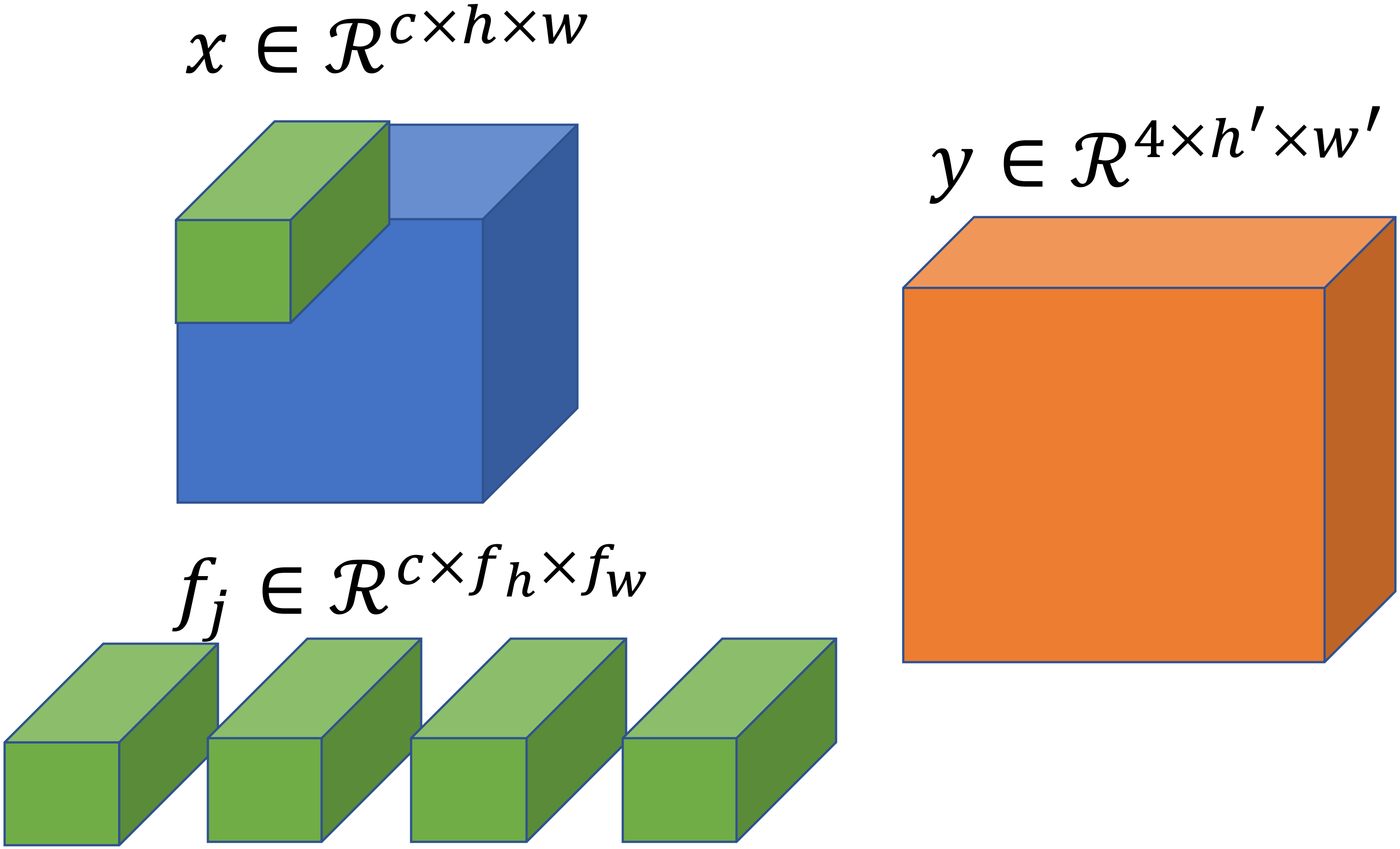


$f \in \mathcal{R}^{c \times f_h \times f_w}$



"$f_h \times f_w$ convolution" (channel dimension is assumed)

Multiple convolutions increase the output channel dimension



Reasoning about input and output shapes is important for debugging and designing CNNs

- Convolution input parameters
  - $ChannelIn = C_{in}$
  - $ChannelOut = C_{out}$ $(equivalent\ to\ \#\ filters)$
  - $KernelSize = [K_0, K_1]$
  - $Stride = [S_0, S_1]$
  - $Padding = [P_0, P_1]$
- $C_{out} = \#\ filters$
- Output spatial dimensions
  - $H_{out} = \left\lfloor \frac{(H_{in} + 2\ P_0 - K_0)}{S_0} + 1 \right\rfloor$
  - $W_{out} = \left\lfloor \frac{(W_{in} + 2\ P_1 - K_1)}{S_1} + 1 \right\rfloor$
- Output batch dimension should match input

Common convolution configurations

$$H_{out} = \left\lfloor \frac{(H_{in} + 2\,P_0 - K_0)}{S_0} + 1 \right\rfloor$$

- Output has same height and width as input
  - 1 x 1 convolution with padding=0, stride=1
  - 3 x 3 convolution with padding=1, stride=1
  - 5 x 5 convolution with padding=2, stride=1

- Output has half the height and width of input
  - 2 x 2 convolution with padding=0, stride=2
  - 4 x 4 convolution with padding=1, stride=2

Switch to demo of 2D with channels, activation functions, and pooling

Transposed convolution can be used to **upsample** an tensor/image to have higher dimensions

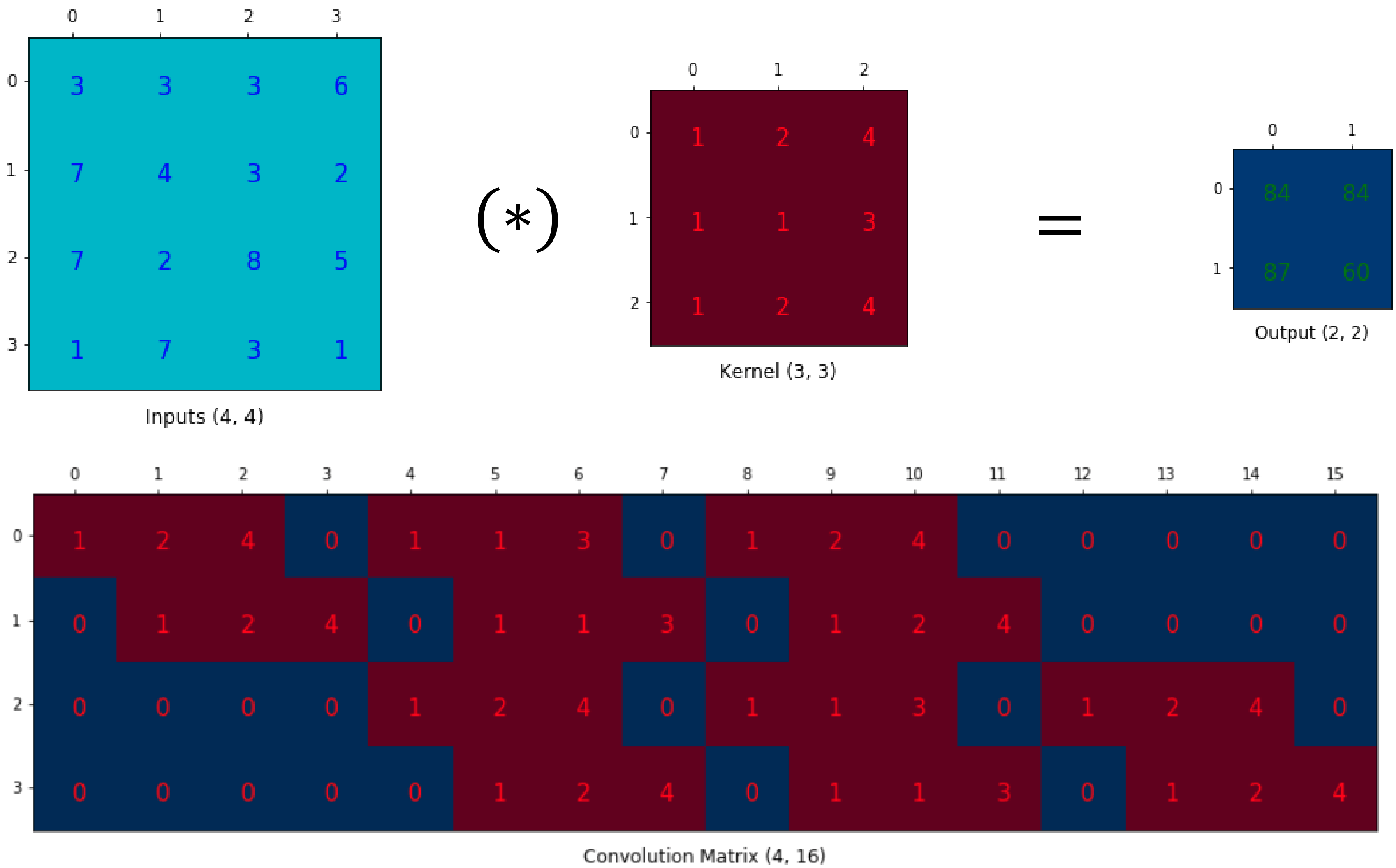
- Also known as:
  - Fractionally-strided convolution
  - Improperly, deconvolution
- Remember: Convolution is like matrix multiplication

$$y = x\,(*)\,f \Leftrightarrow \mathrm{vec}(y) = A_f \mathrm{vec}(x)$$

- Transpose convolution is the transpose of A_f:

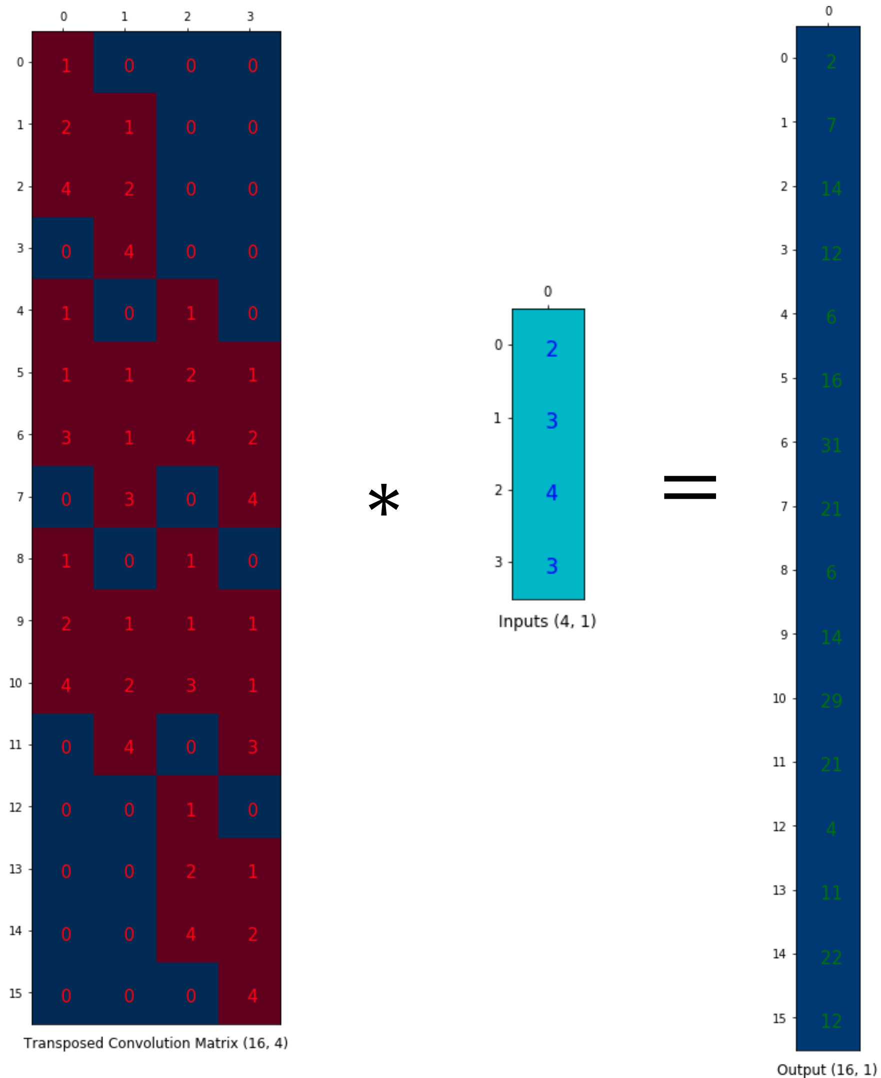
$$\mathrm{vec}(y) = A_f^T \mathrm{vec}(x)$$

Convolution operator with corresponding matrix

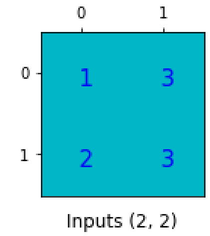


https://github.com/naokishibuya/deep-learning/blob/master/python/transposed_convolution.ipynb

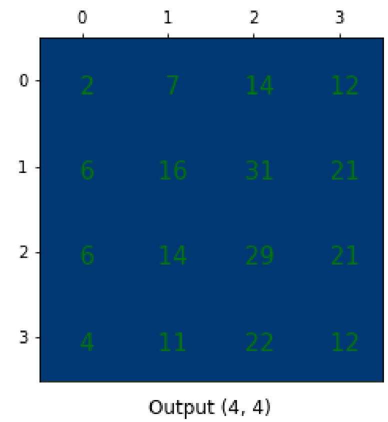
Transposed convolution operator with corresponding matrix



Reshaped input

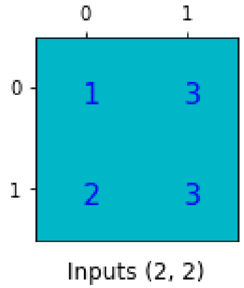


Reshaped output



https://github.com/naokishibuya/deep-learning/blob/master/python/transposed_convolution.ipynb

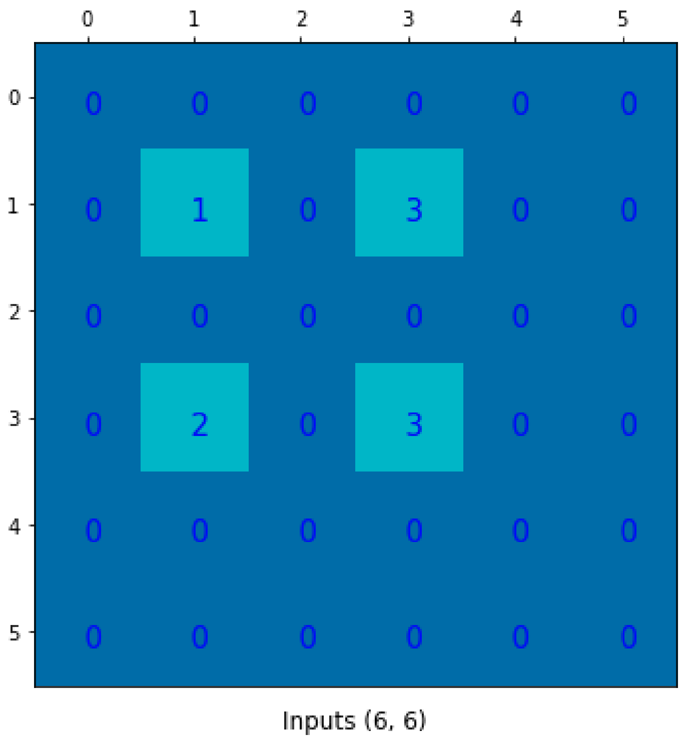
Transposed convolution can be **equivalent** to a simple convolution with zero rows/columns added (added zeros simulate fractional strides)



Original input

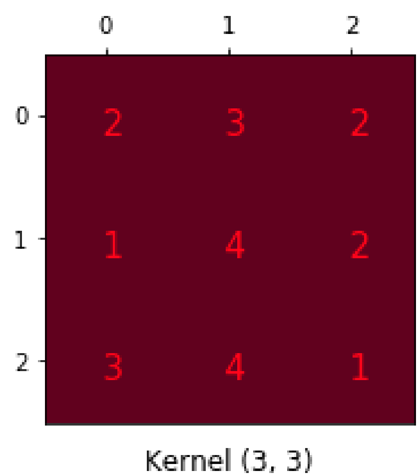
(Note: More modern upsampling layers upsample by imputing/interpolating non-zeros and then apply convolution.)

Zero-padded input



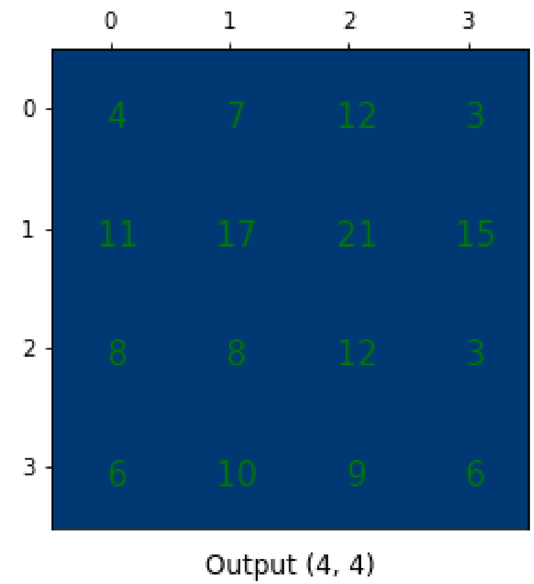
(*)

Kernel



=

Output



Computing tensor shapes with transpose convolutions

- Channels is computed the same as convolution
- For spatial dimensions, you switch the input and output dimensions
  - Reason about the standard convolution dimensions
  - And then flip input and output dimensions
- Like convolutions, output has same height and width as input
  - 1 x 1 convolution with padding=0, stride=1
  - 3 x 3 convolution with padding=1, stride=1
  - (Stride of 1 is equivalent to stride of 1 convolution)
- Output has **double (upsample)** the height and width of input
  - 2 x 2 convolution with padding=0, stride=2
  - 4 x 4 convolution with padding=1, stride=2
  - 6 x 6 convolution with padding=2, stride=2

Demo of CIFAR-10 CNN in Pytorch

Two important modern CNN architecture concepts: **batch normalization** and **residual networks**

<u>Batch normalization</u> dynamically normalizes each feature to have zero mean and unit variance

- Basic idea: Normalize input batch of each layer **<u>during the forward pass</u>**

1. Input is **minibatch** of data $X^t \in \mathbb{R}^{m \times d}$ at iteration t
2. Compute mean and standard deviation for every feature

$$\mu_j^t = \mathbb{E}\left[x_j^t\right], \sigma_j^t = \sqrt{\mathbb{E}\left[\left(x_j^t - \mu_j^t\right)^2\right]}, \qquad \forall j \in \{1, \cdots, d\}$$

3. Normalize each feature (note **<u>different for every batch</u>**)

$$\tilde{x}_{i,j}^t = \frac{\left(x_{i,j}^t - \mu_j^t\right)}{\sigma_j^t}$$

4. Output $\tilde{X}^t$

Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018). How does batch normalization help optimization?. In *Advances in Neural Information Processing Systems* (pp. 2483-2493).

Because BatchNorm removes linear effects, extra linear parameters are also learned

- The form of this final update is:

$$\tilde{x}_{i,j}^{t} = \frac{\left(x_{i,j}^{t} - \mu_j^t\right)}{\sigma_j^t} \cdot \gamma_j + \beta_j$$

  - Where γ_j and β_j are learnable parameters
  - While μ_j^t and σ_j^t are computed from the **minibatch**
- But how do we compute μ_j^t and σ_j^t about during test time (i.e., no minibatch)?
- Use running average of mean and variance

$$\mu_{run}^{t} = \lambda\mu_{run}^{t-1} + (1-\lambda)\mu_{batch}^{t}$$
$${\sigma^2}_{run}^{t} = \lambda{\sigma^2}_{run}^{t-1} + (1-\lambda){\sigma^2}_{batch}^{t}$$

For CNNs, the channel dimension is treated as a "feature"

- If the input minibatch tensor is $X^t \in \mathbb{R}^{m \times c \times h \times w}$, then the channel dimension c is treated as a feature:

$$\mu_j^t = \mathbb{E}\left[x_j^t\right], \sigma_j^t = \sqrt{\mathbb{E}\left[\left(x_j^t - \mu_j^t\right)^2\right]},$$
$$\forall j \in \{1, \cdots, c\}$$

  - Where the mean is taken over **<u>both</u>** the batch dimension m **<u>and</u>** the spatial dimensions h and w
  - Called "Spatial Batch Normalization"
- Variants: Instance, Group or Layer Normalization

https://pytorch.org/docs/stable/nn.html#normalization-layers

<u>BatchNorm</u> can stabilize and accelerate training of deep models

- To use in practice:
  - Only normalize batches during training (`model.train()`)
  - **<u>Turn off</u>** after training (`model.eval()`)
    - Uses running average of mean and variance
- Surprisingly effective at stabilizing training, reducing training time, and producing better models
- Not fully understood why it works

Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018). How does batch normalization help optimization?. In *Advances in Neural Information Processing Systems* (pp. 2483-2493).

Demo of batch normalization in PyTorch

<u>Residual networks</u> add the input to the output of the CNN

- Most deep model layers have the form:

$$y = f(x)$$

  - Where f could be any function including a convolutional layer like $f(x) = \sigma\left(\text{Conv}\left(\sigma(\text{Conv}(x))\right)\right)$

- Residual layers add back in the input

$$y = f(x) + x$$

  - Notice that $f(x)$ models the difference between x and y (hence the name **<u>residual</u>**)

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).

A residual network enables deeper networks because gradient information can flow between layers

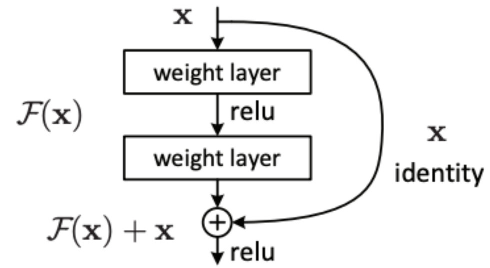
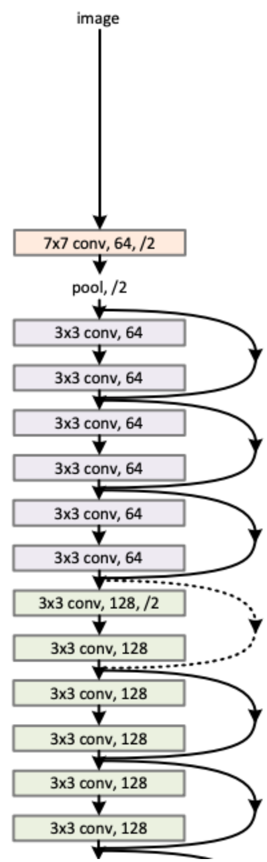


Figure 2. Residual learning: a building block.

- A data flow diagram shows the "shortcut" connections
- Consider composing 2 residual layers:
  - $z^{(1)} = f_1(x) + x$
  - $z^{(2)} = f_2\left(z^{(1)}\right) + z^{(1)}$
  - Or, equivalently
    $z^{(2)} = f_2(f_1(x) + x) + f_1(x) + x$
- If the residuals = 0, then this is merely the identity function



Images from: He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).

Detail: If the dimensionality is not the same, then use either fully connected layer or convolution layer to match

- In the 1D case, suppose $f(x)\colon \mathbb{R}^d \to \mathbb{R}^m$, then we need to multiply x by linear operator to match the dimension

$$y = f(x) + Wx, \qquad \text{where } W \in \mathbb{R}^{m \times d}$$

- Similarly, for images, if $f(x)\colon \mathbb{R}^{c \times h \times w} \to \mathbb{R}^{c' \times h' \times w'}$, we can apply a convolution layer to match the dimensions

$$y = f(x) + \text{conv}(x),$$
$$\text{where conv}(\cdot)\colon \mathbb{R}^{c \times h \times w} \to \mathbb{R}^{c' \times h' \times w'}$$

Demo of CNN with very simple residual network

U-Nets have an autoencoder structure with skip connections for **semantic segmentation** task

- Concatenation + convolution rather than residual skip connections
- ***Any*** (pretrained) classification backbone can be used for encoder
- State-of-the-art semantic segmentation are based on this idea

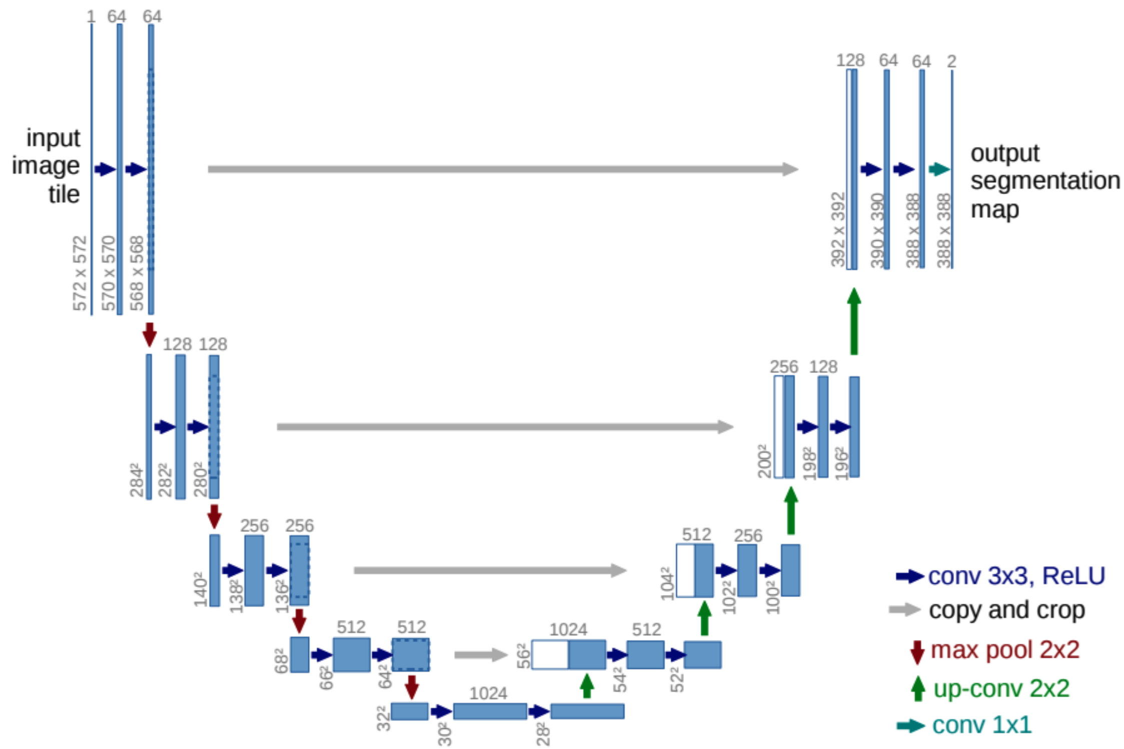


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

Figure from: Ronneberger, O., Fischer, P., & Brox, T. (2015, October). U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention* (pp. 234-241). Springer, Cham.