Lecture 4: Architectures for Grids

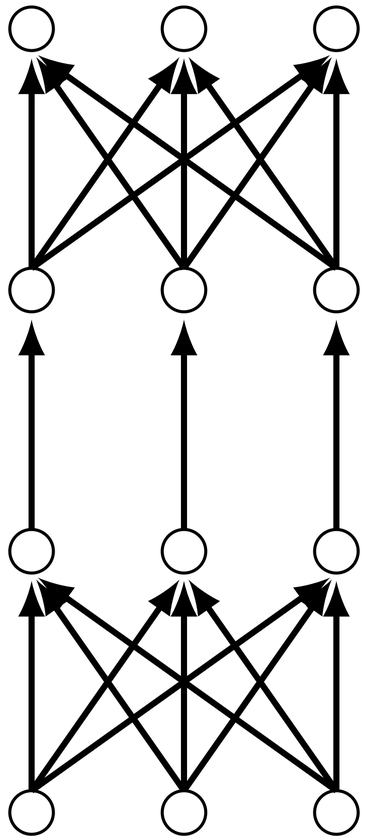
Speaker: Sara Beery

Multilayer Perceptron

linear comb. of neurons ▷

neuron-wise nonlinearity ▷

linear comb. of neurons ▷

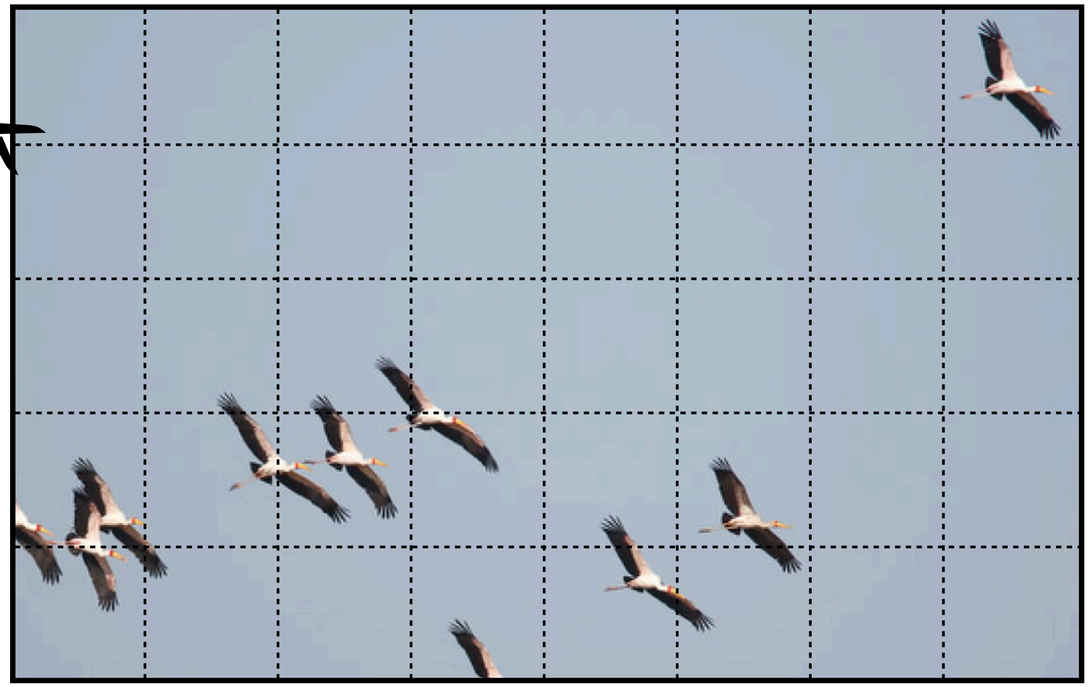


+ Universal
+ Simple (elegant theory)
+ Embarassingly parallel

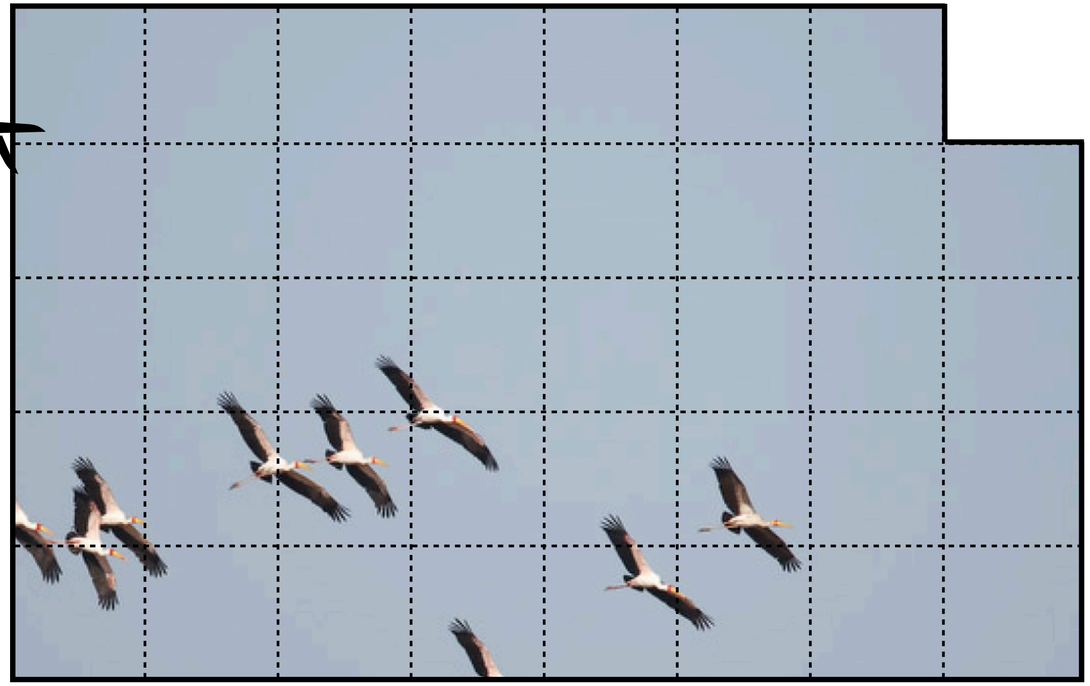
- Weak inductive biases
- Sample inefficient / data hungry
- Dense (fully-connected) linear layers take a lot of compute

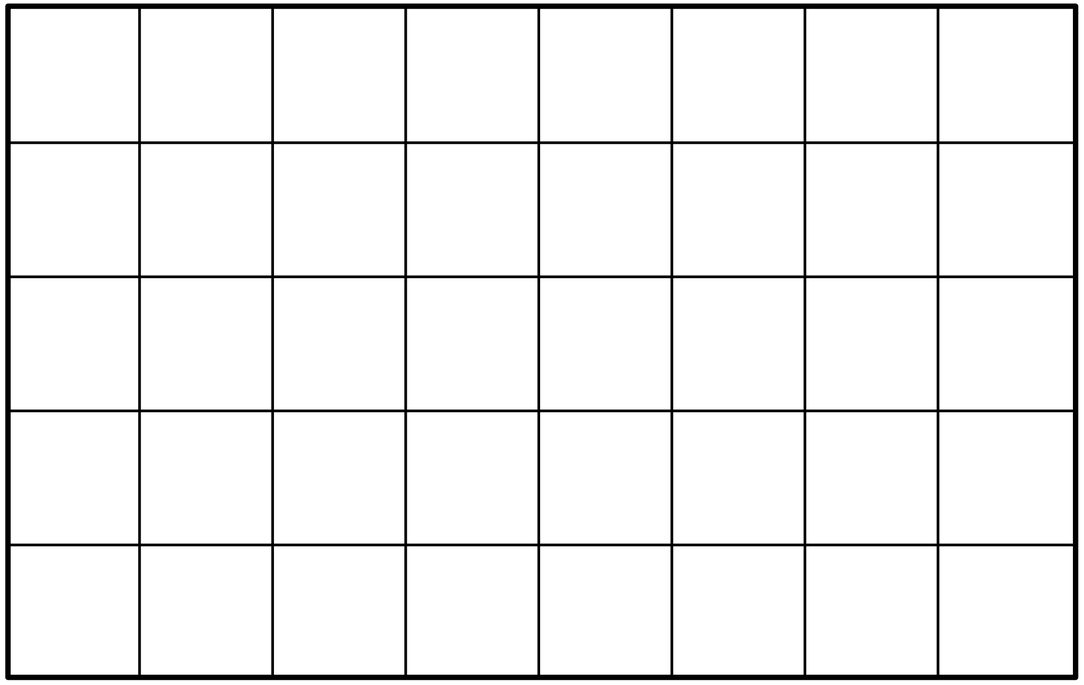
Convolutional Neural Networks

Photo credit: Fredo Durand



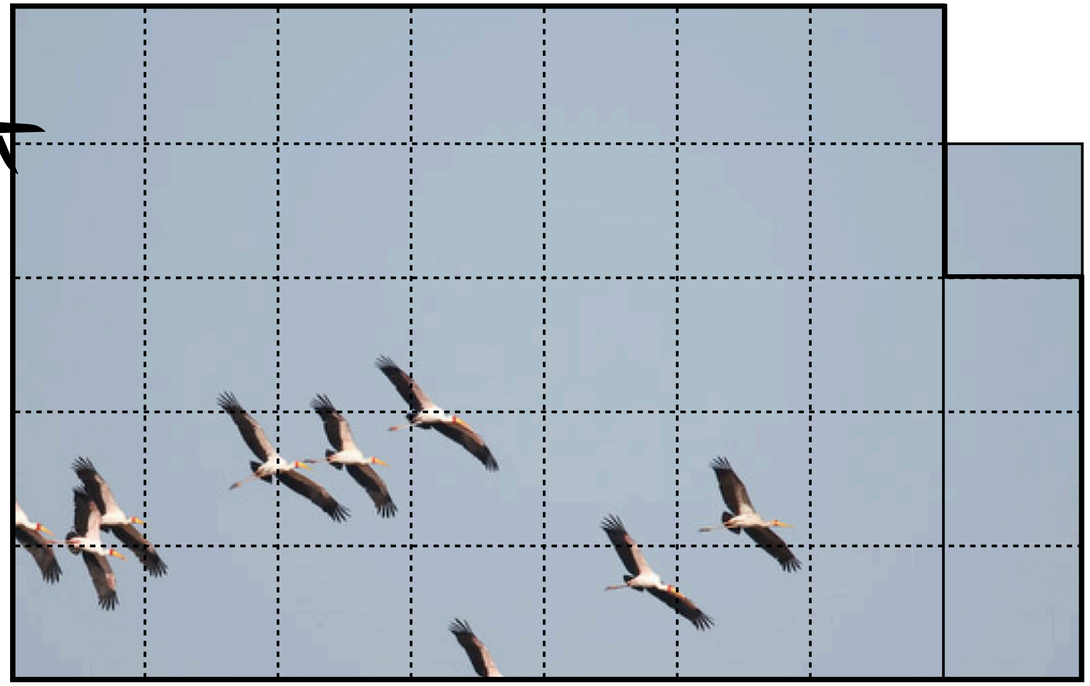

Classifier
Bird



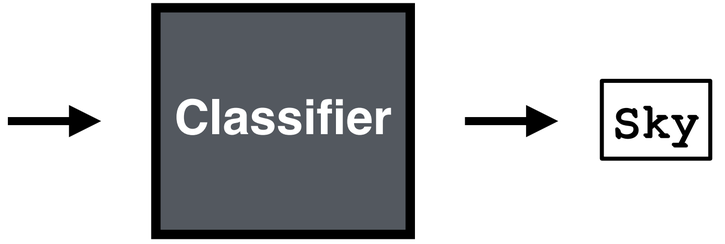


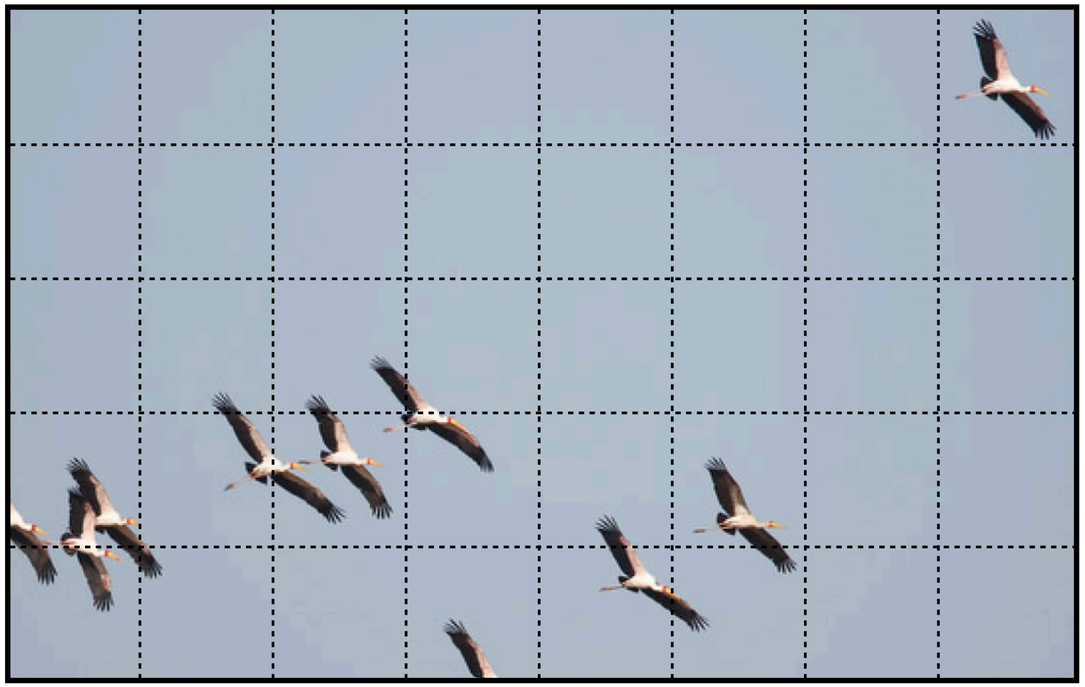
Classifier

Bird

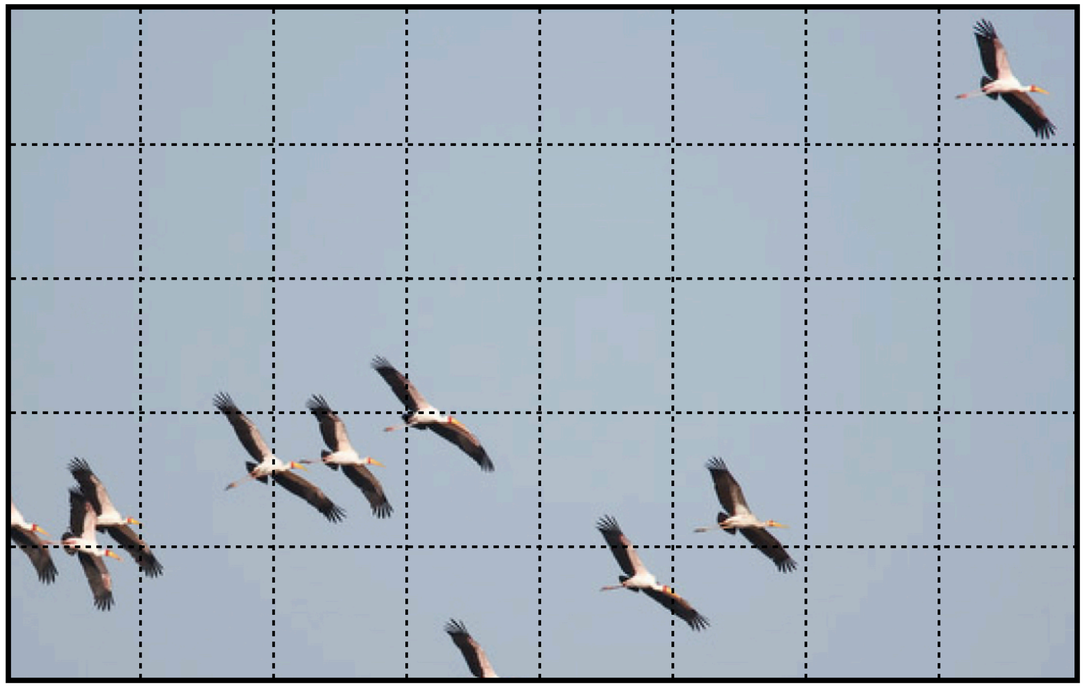


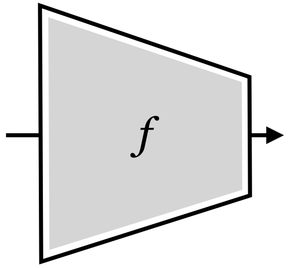
							Bird





Sky	Sky	Sky	Sky	Sky	Sky	Sky	Bird
Sky	Sky	Sky	Sky	Sky	Sky	Sky	Sky
Sky	Sky	Sky	Sky	Sky	Sky	Sky	Sky
Bird	Bird	Bird	Sky	Bird	Sky	Sky	Sky
Sky	Sky	Sky	Bird	Sky	Sky	Sky	Sky





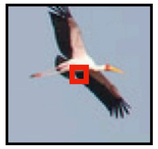
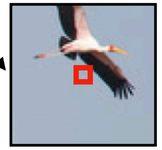
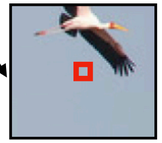
sky	sky	sky	sky	sky	sky	sky	bird
sky	sky	sky	sky	sky	sky	sky	sky
sky	sky	sky	sky	sky	sky	sky	sky
bird	bird	bird	bird	sky	bird	sky	sky
sky	sky	sky	bird	bird	sky	sky	sky

Problem:

What if objects don’t fit neatly into these patches?

How to increase the resolution of the output map?

What's the object class of the center pixel?

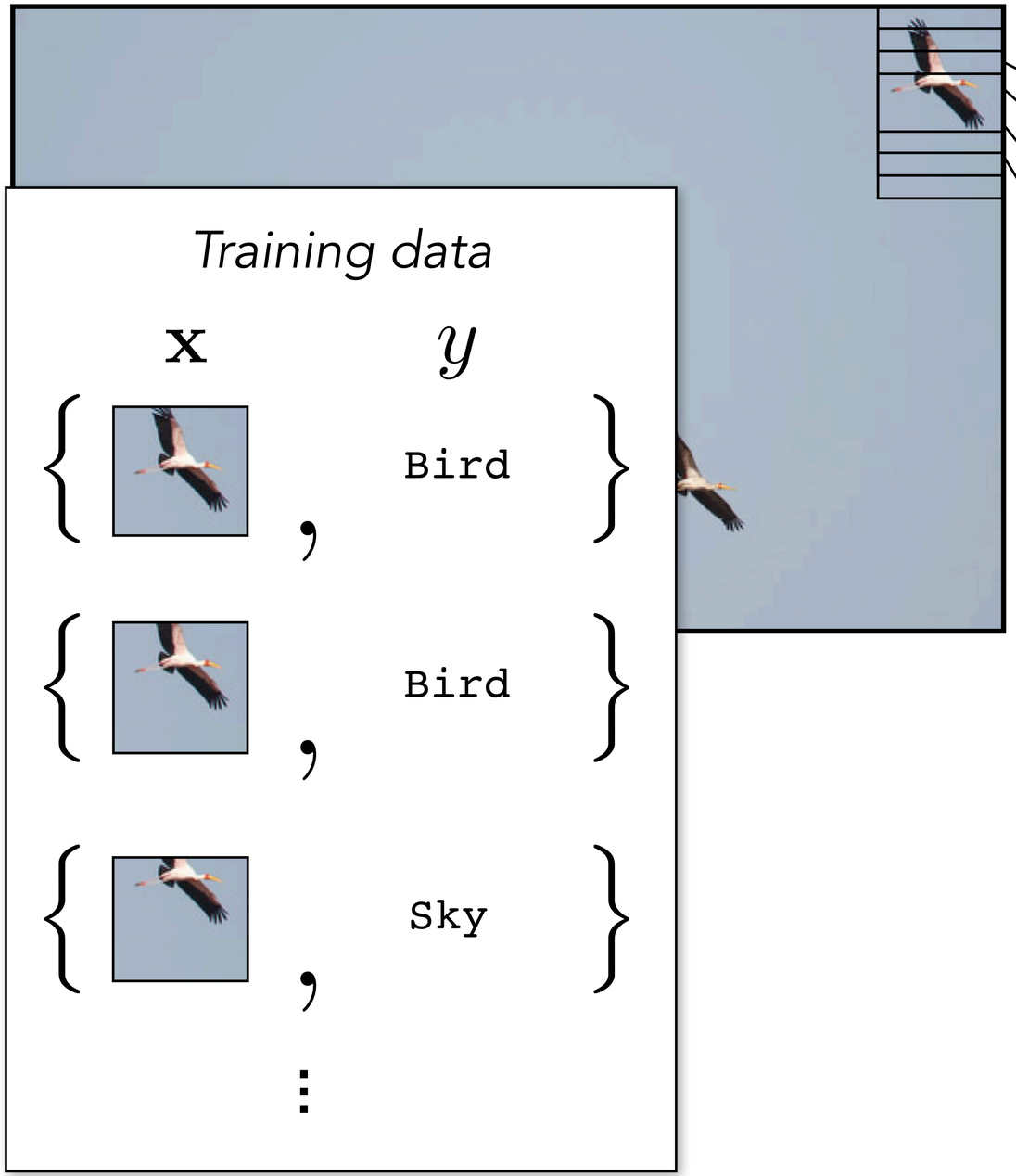





f → Bird

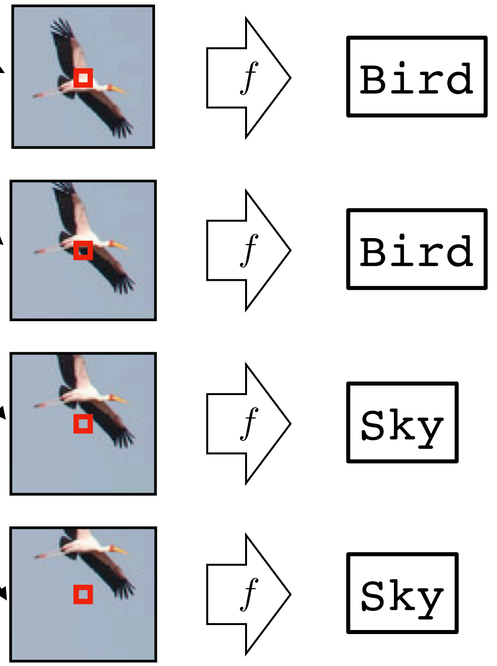
f → Bird

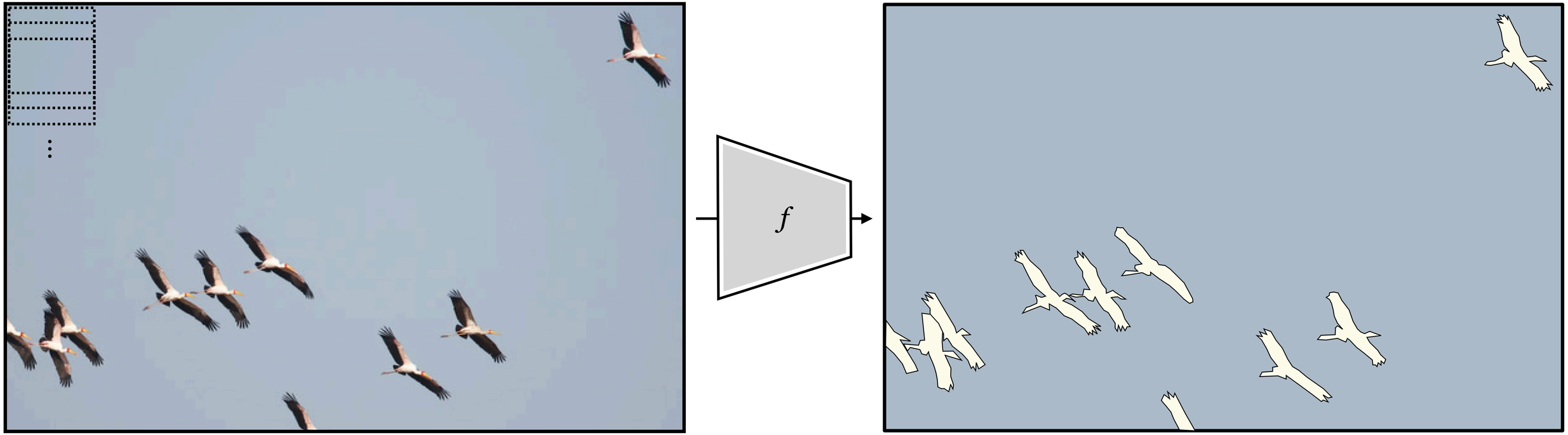
f → Sky

f → Sky



What's the object class of the center pixel?

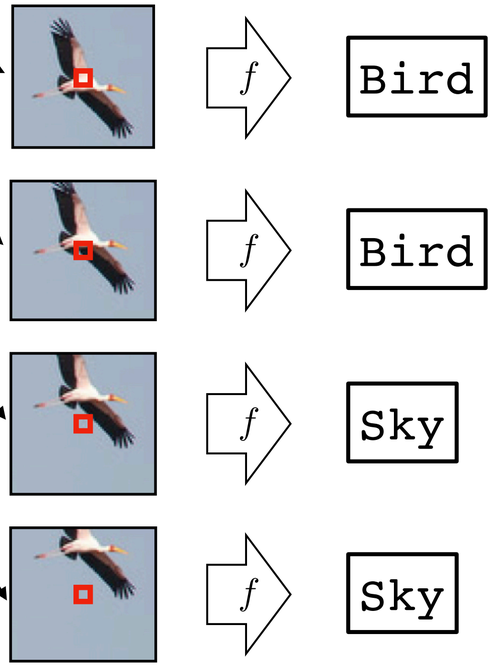


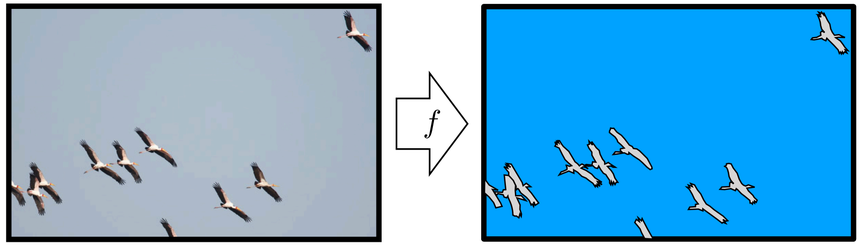


(Colors represent one-hot codes)

This problem is called **semantic segmentation**

What's the object class of the center pixel?



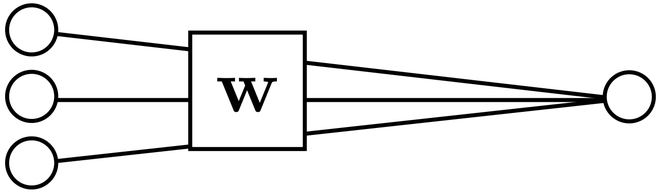


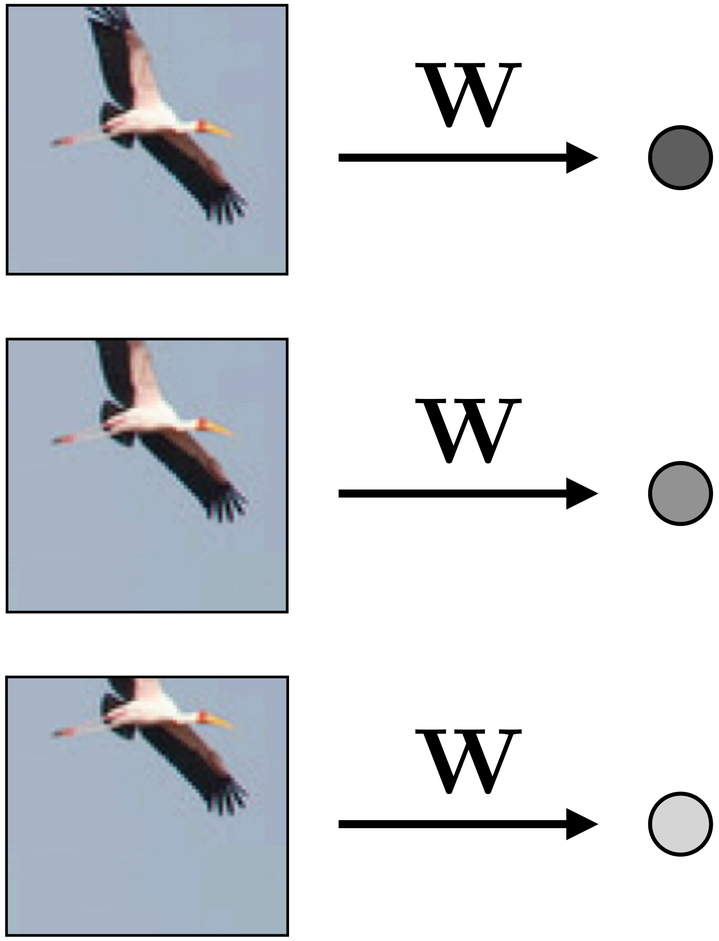
Translation invariance: process each patch in the same way.

An *equivariant* mapping:

$$f(\texttt{translate}(x)) = \texttt{translate}(f(x))$$

W computes a weighted sum of all pixels in the patch

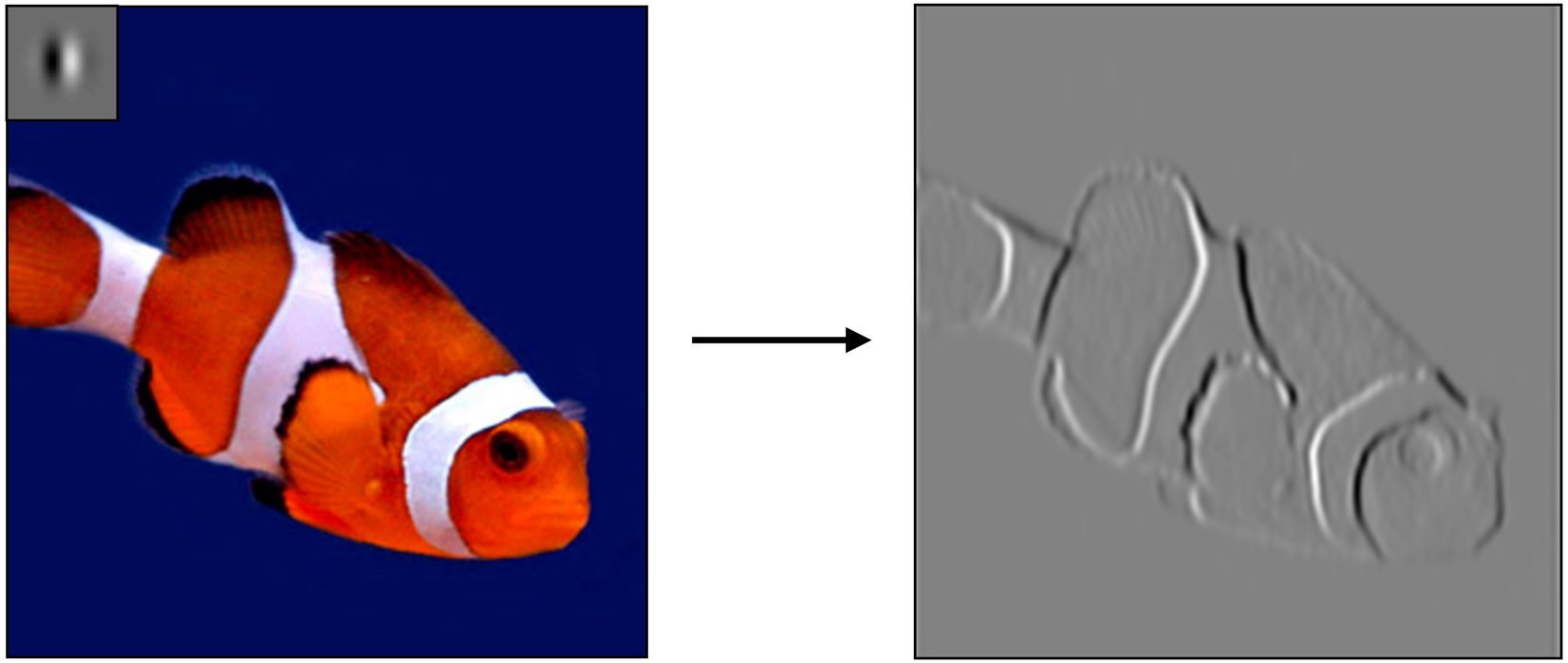


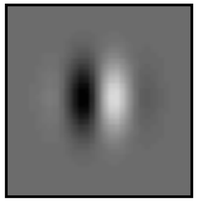


W is a **convolutional kernel** applied to the full image!

Convolution

Linear, shift-invariant transformation

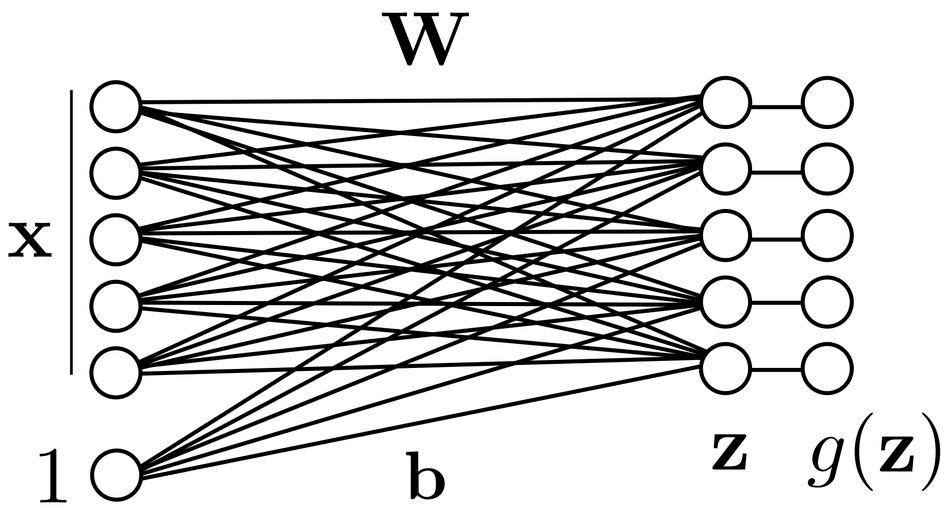


filter 

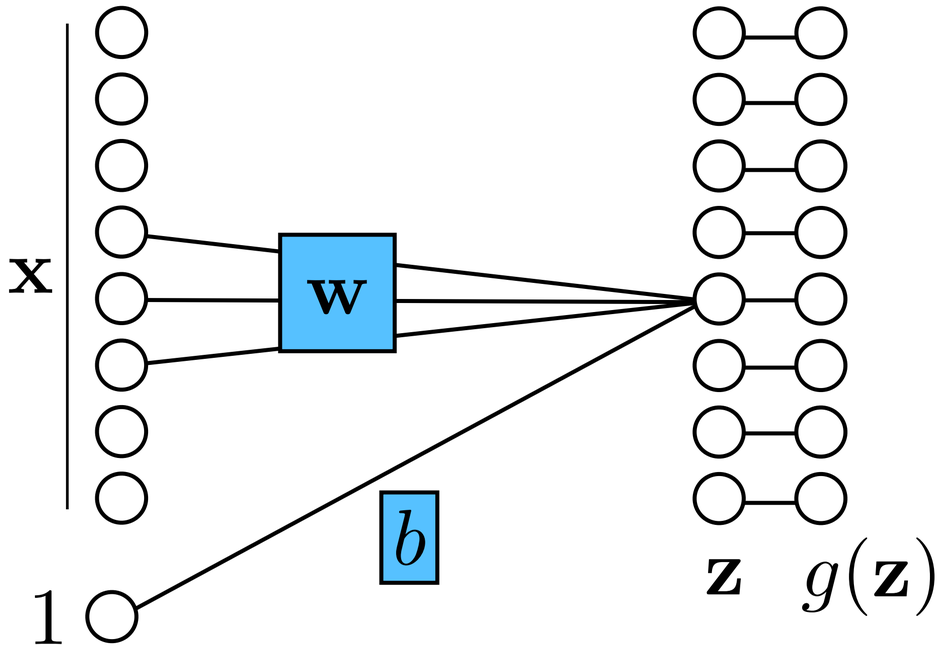
$$x_{\mathtt{out}}[n,m] = b + \sum_{k_1,k_2=-K}^{K} w[k_1,k_2]\, x_{\mathtt{in}}[n+k_1, m+k_2]$$

Fully-connected network

Fully-connected (fc) layer



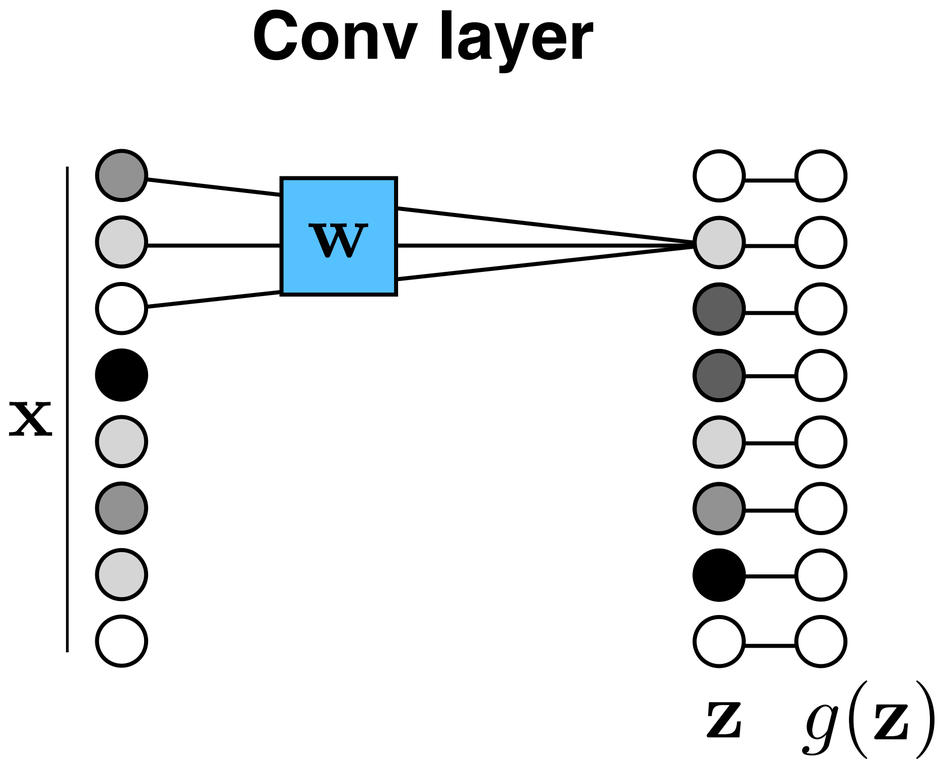
Locally connected network



Often, we can assume output is a **local** function of input.

If we use the same weights (**weight sharing**) to compute each local function, we get a convolutional neural network.

Convolutional neural network



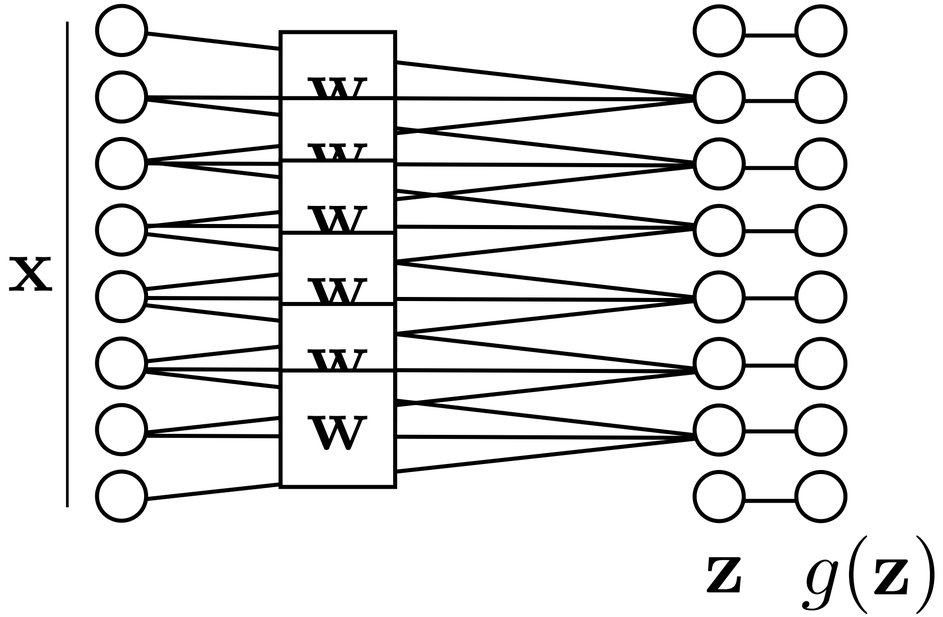
$$\mathbf{z} = \mathbf{w} \star \mathbf{x} + b$$

Often, we assume output is a **local** function of input.

If we use the same weights (**weight sharing**) to compute each local function, we get a convolutional neural network.

Weight sharing

Conv layer



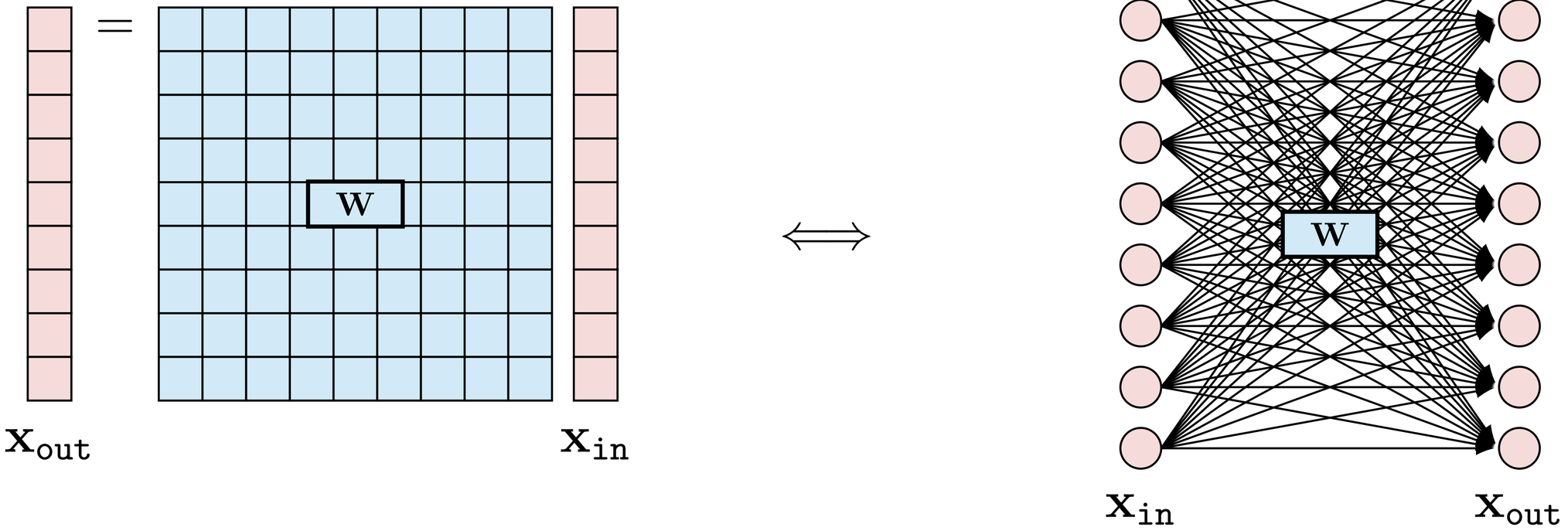
$$\mathbf{z} = \mathbf{w} \star \mathbf{x} + b$$

Often, we assume output is a **local** function of input.

If we use the same weights (**weight sharing**) to compute each local function, we get a convolutional neural network.

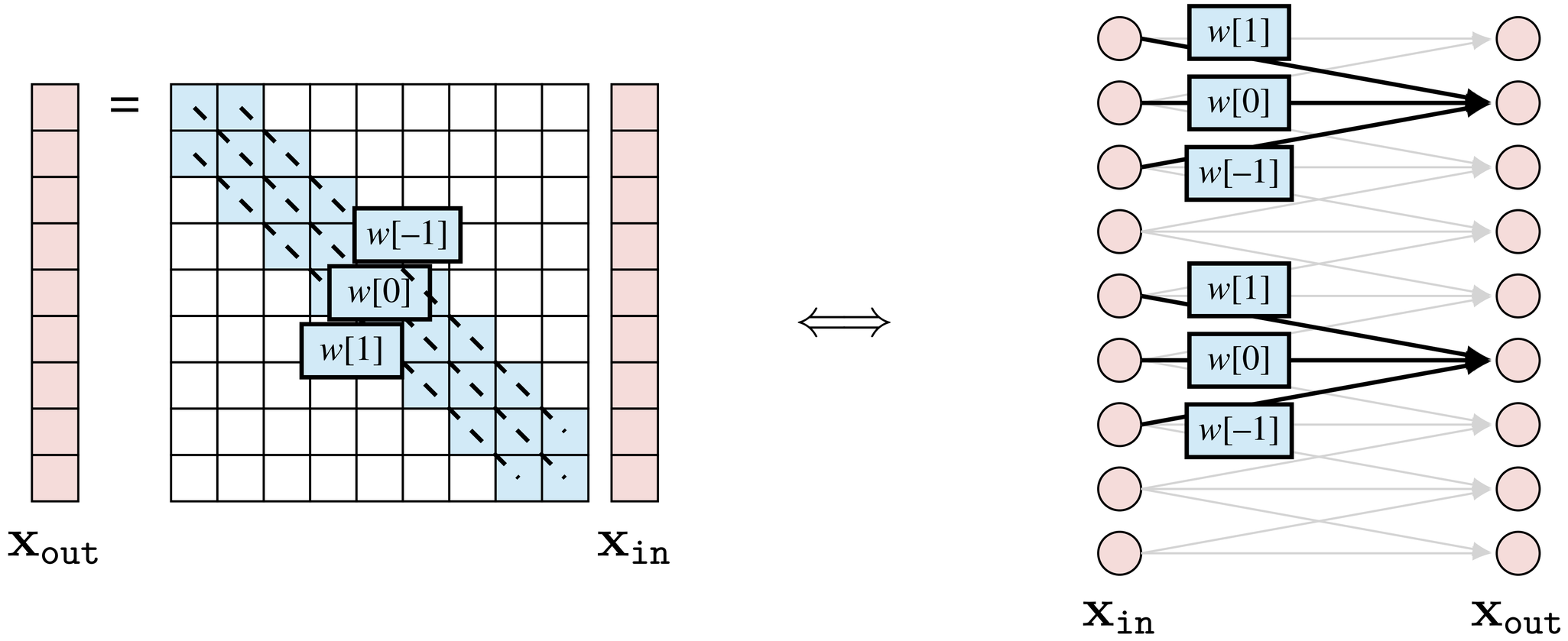
(Fully-connected) linear layer

$$\mathbf{x}_{\mathtt{out}} = \mathbf{W}\mathbf{x}_{\mathtt{in}} + \mathbf{b}$$



Convolutional layer

$$\mathbf{x}_{\mathtt{out}} = \mathbf{w} \star \mathbf{x}_{\mathtt{in}} + b$$



Toeplitz matrix

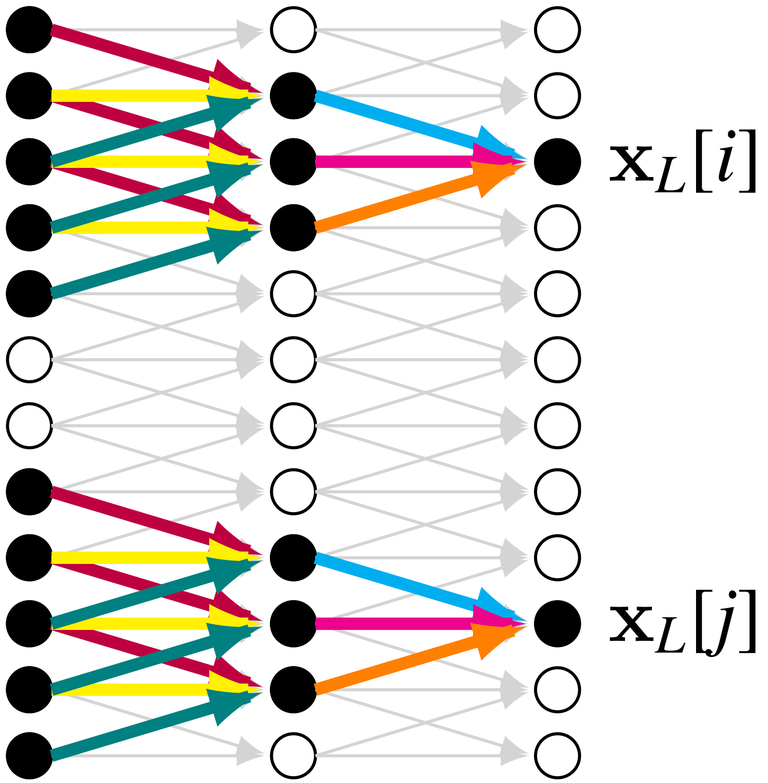
$$\begin{pmatrix} a & b & c & d & e \\ f & a & b & c & d \\ g & f & a & b & c \\ h & g & f & a & b \\ i & h & g & f & a \end{pmatrix}$$

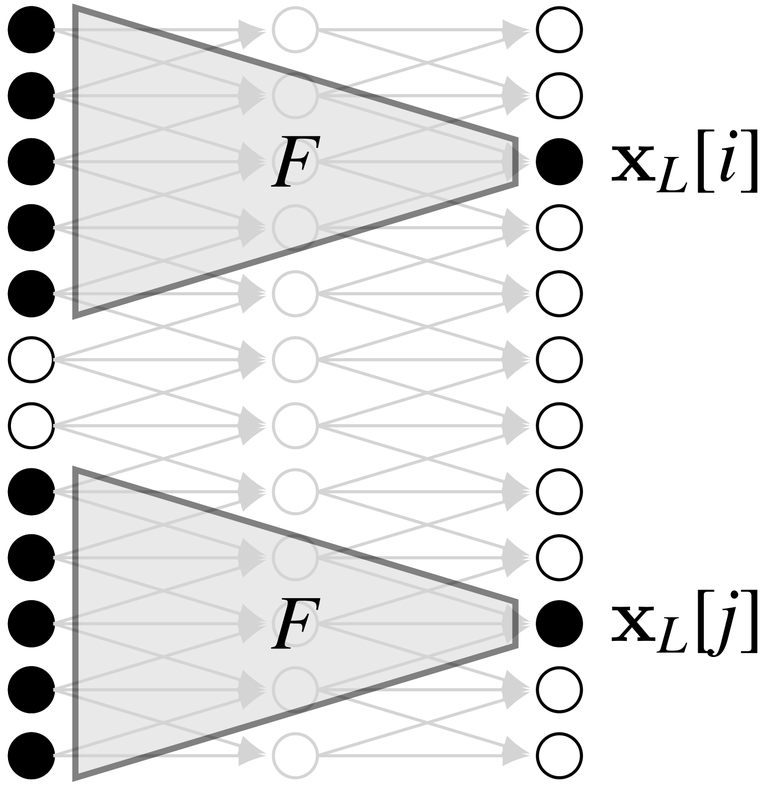
$$\mathbf{y} = \quad \mathbf{x}$$

e.g., pixel image

- Constrained linear layer
- Fewer parameters —> easier to learn, less overfitting

What happens when you stack convolutional layers?



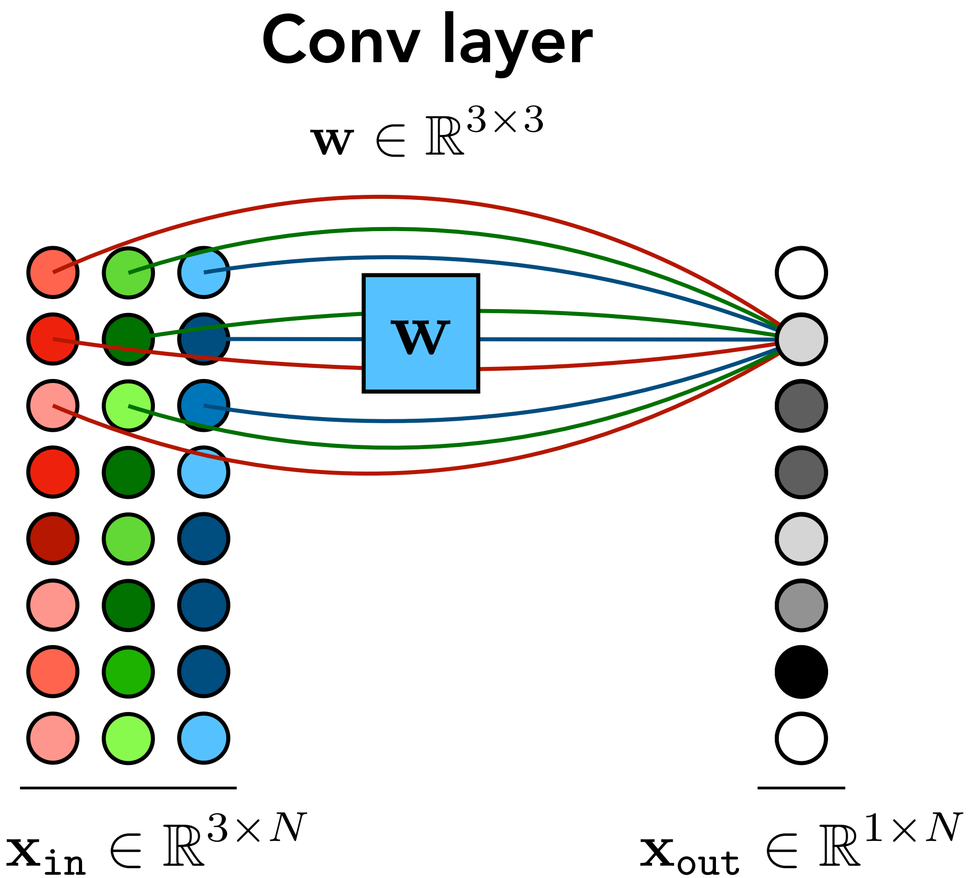


The whole CNN acts like a (nonlinear) convolutional filter!

What if we have color?

(aka multiple input channels?)

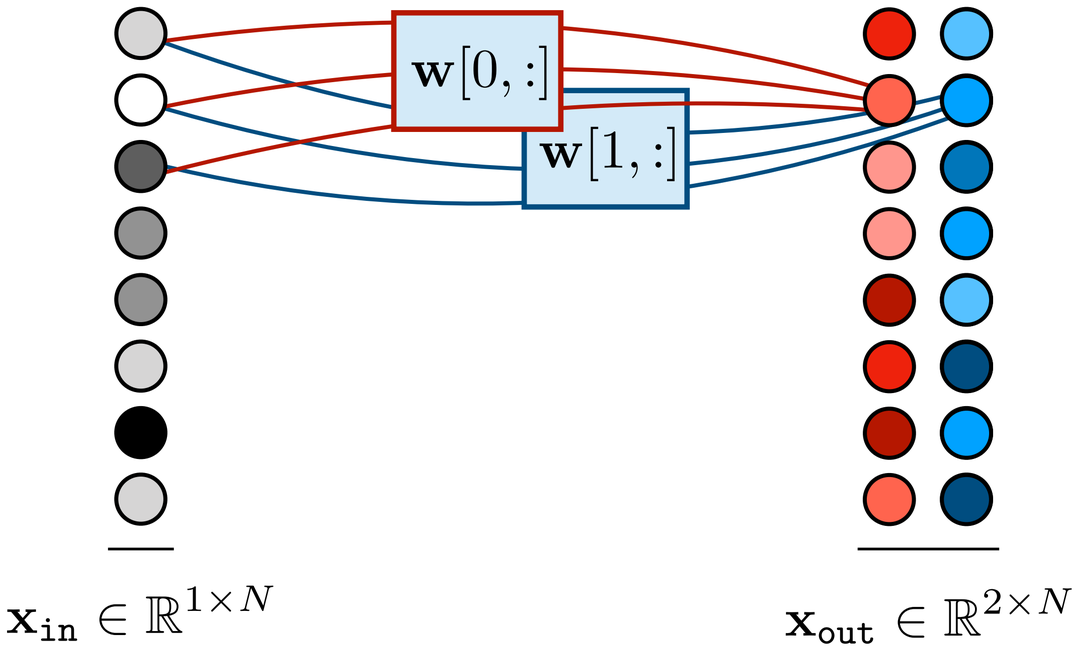
Multichannel inputs



$$\mathbf{x_{out}} = \sum_c \mathbf{w}[c,:] \star \mathbf{x_{in}}[c,:] + b[c]$$

Multichannel *outputs*

Conv layer



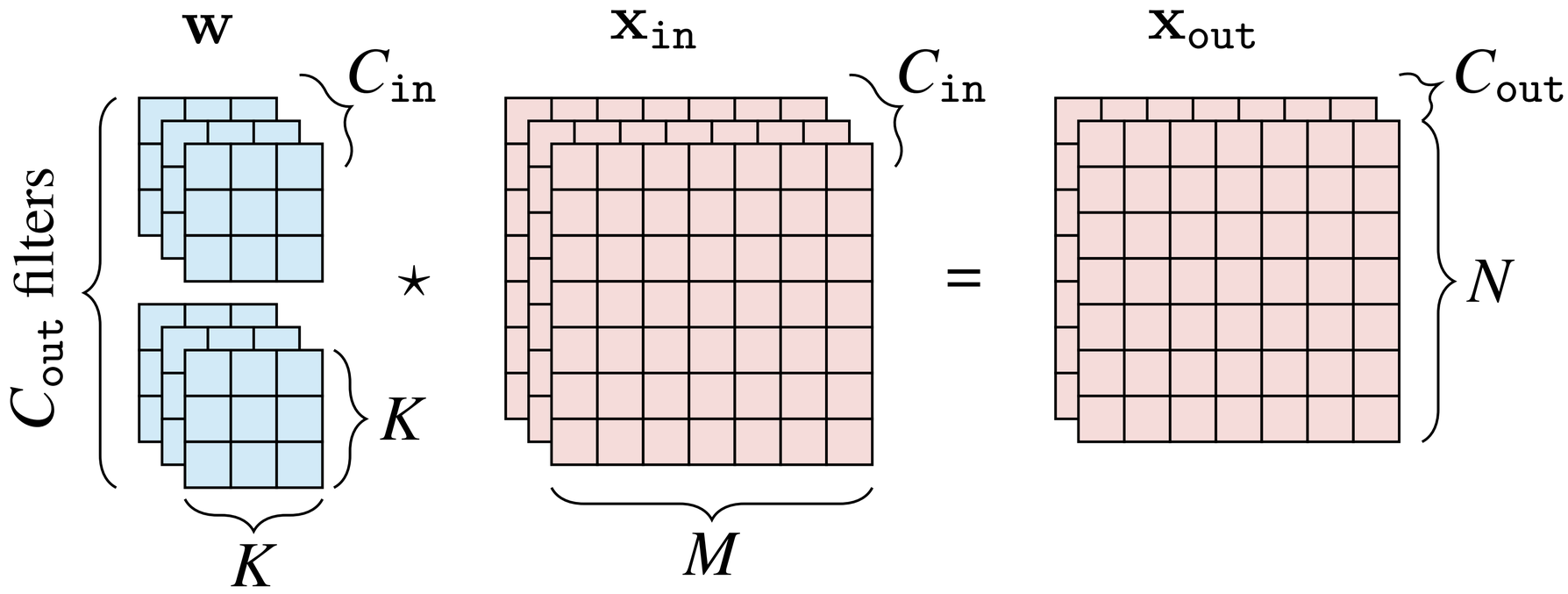
Filter bank of C filters

$$\mathbf{x}_{\mathbf{out}}[0,:] = \mathbf{w}[0,:] \star \mathbf{x}_{\mathbf{in}} + b[0]$$

$$\vdots$$

$$\mathbf{x}_{\mathbf{out}}[C,:] = \mathbf{w}[C-1,:] \star \mathbf{x}_{\mathbf{in}} + b[C-1]$$

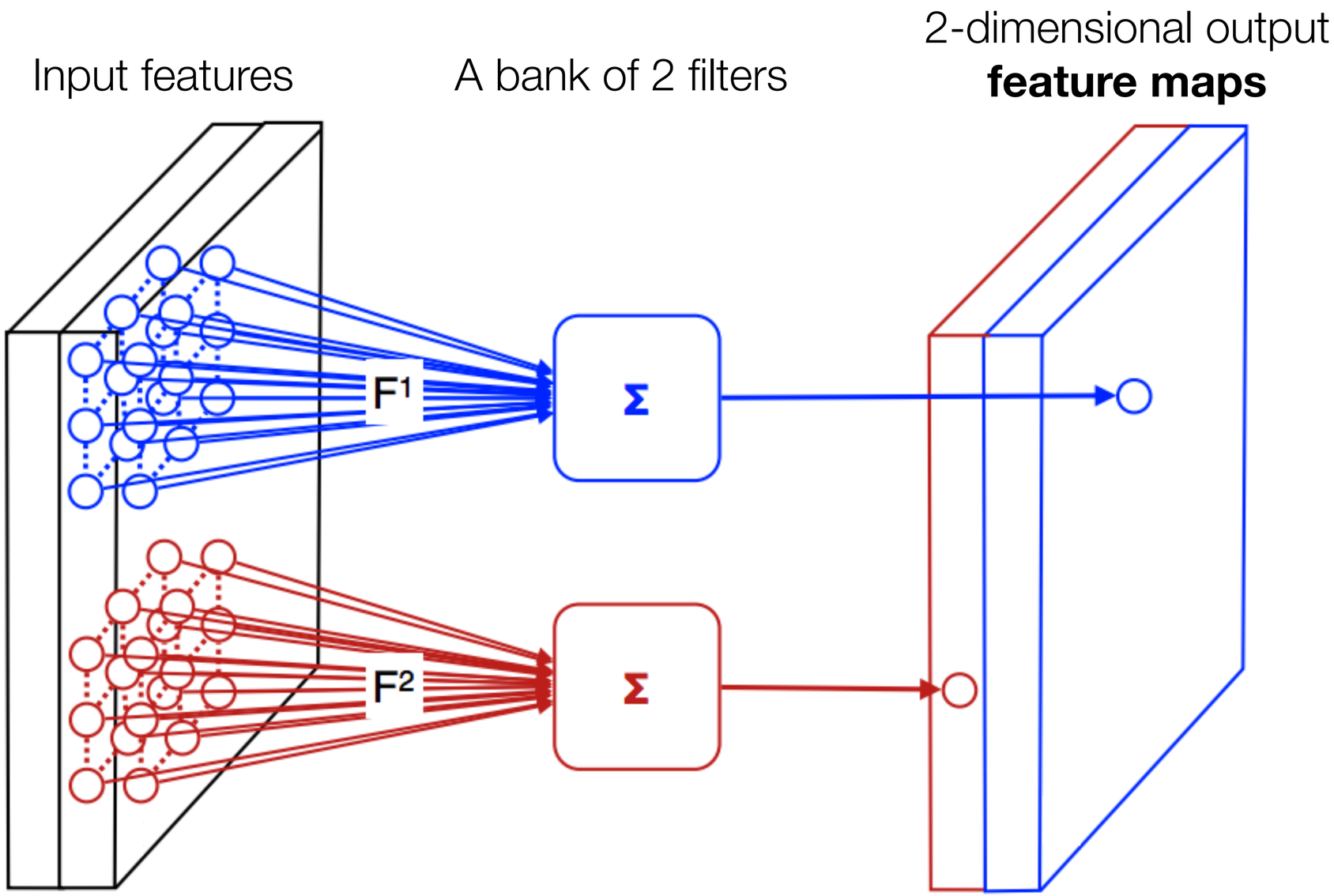
General Convolutional Layer Form: Multi-Input, Multi-Output



$$\mathbf{x}_{\text{out}}[c_2, :, :] = \sum_{c_1=1}^{C_{\text{in}}} \mathbf{w}[c_1, c_2, :, :] \star \mathbf{x}_{\text{in}}[c_1, :, :] + b[c_2]$$

Input channel

Output channel



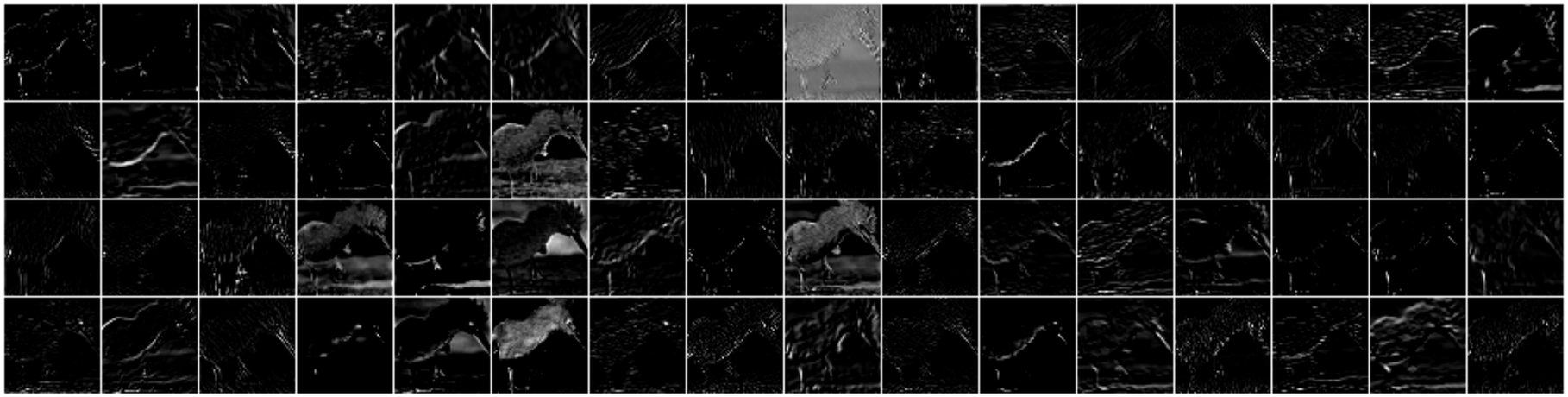
$$\mathbf{x_{in}} \in \mathbb{R}^{C_{\mathtt{in}} \times H \times W} \rightarrow \mathbf{x_{out}} \in \mathbb{R}^{C_{\mathtt{out}} \times H \times W}$$

[Figure modified from Andrea Vedaldi]

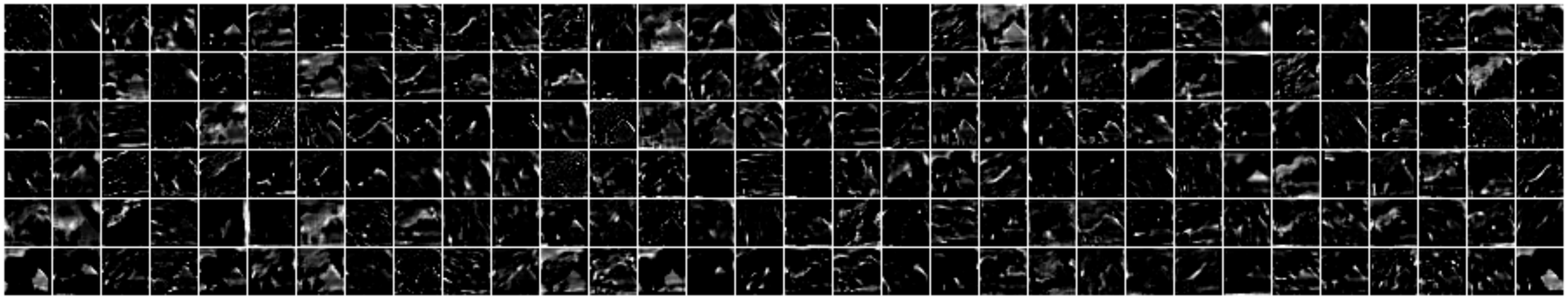
Feature maps

Input

conv1 (after first conv layer)



conv2 (after second conv layer)



- Each layer can be thought of as a set of C **feature maps** aka **channels**
- Each feature map is an NxM image

Filter sizes

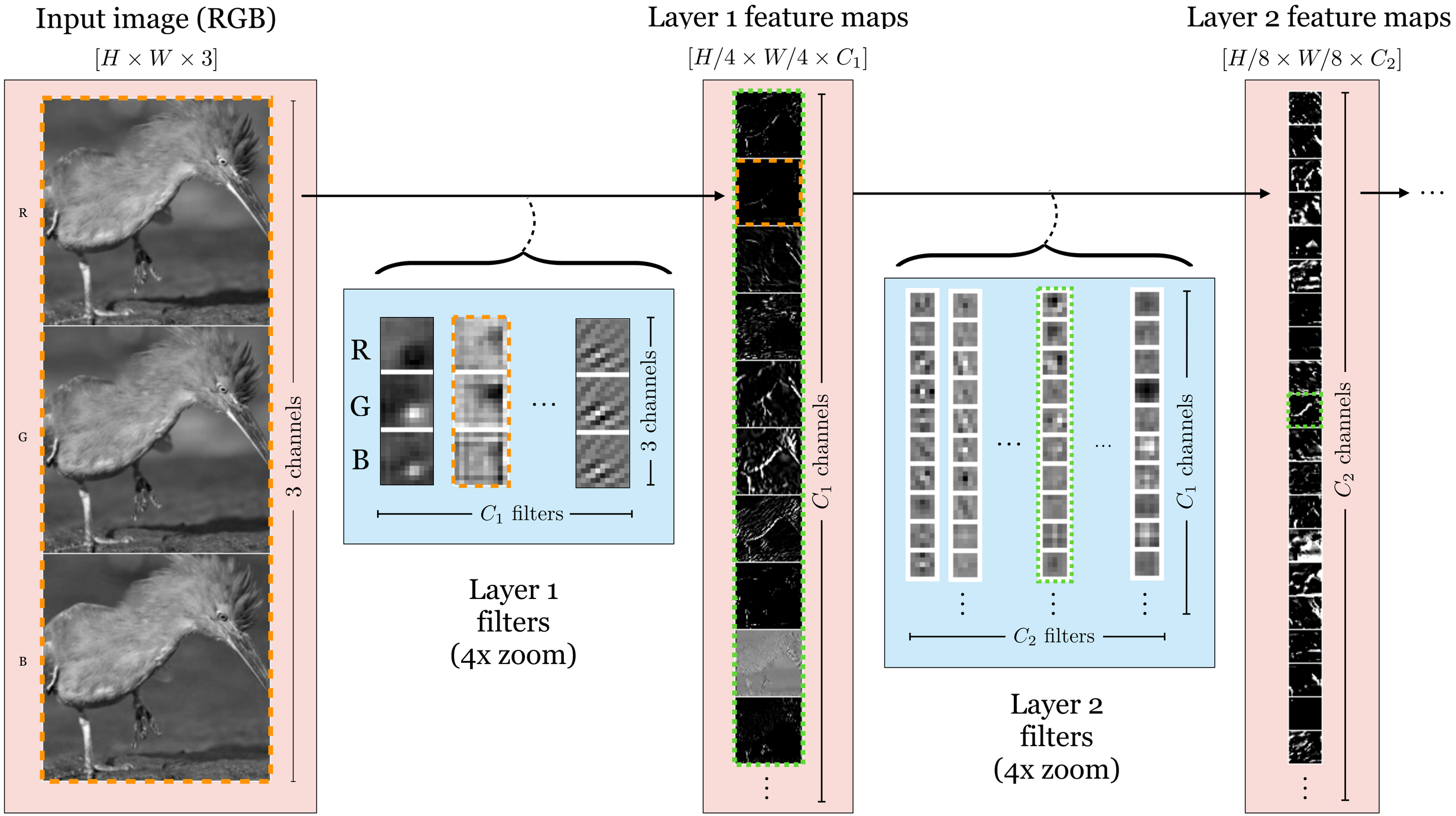
When mapping from

$$\mathbf{x}_l \in \mathbb{R}^{C_l \times N \times M} \rightarrow \mathbf{x}_{(l+1)} \in \mathbb{R}^{C_{(l+1)} \times N \times M}$$

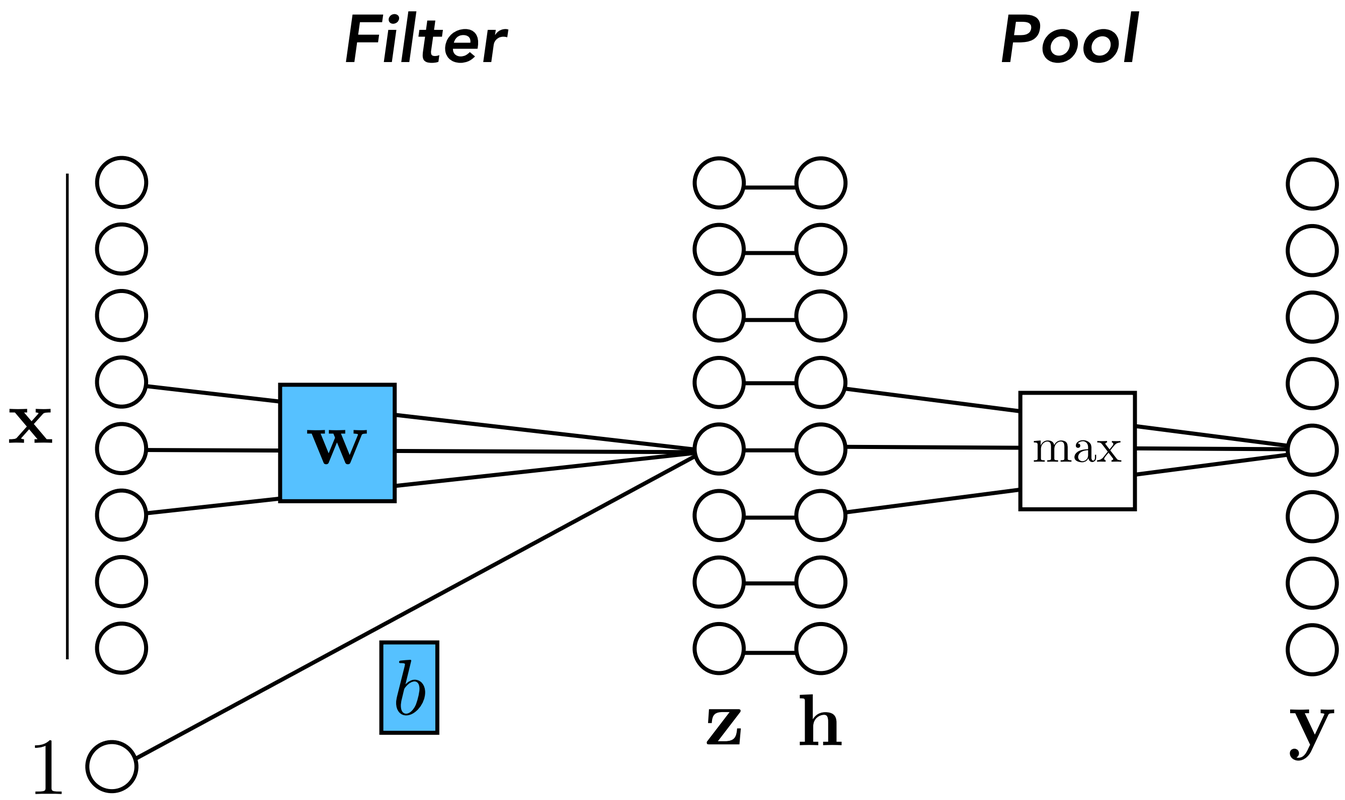
using an filter of spatial extent $K_1 \times K_2$

Number of parameters per filter: $K_1 \times K_2 \times C_l$

Number of filters: $C_{(l+1)}$


Input image (RGB)
$[H \times W \times 3]$
R
G
B
3 channels
R
G
B
C_1 filters
3 channels
Layer 1 filters (4x zoom)
Layer 1 feature maps
$[H/4 \times W/4 \times C_1]$
C_1 channels
C_1 channels
C_2 filters
Layer 2 filters (4x zoom)
Layer 2 feature maps
$[H/8 \times W/8 \times C_2]$
C_2 channels

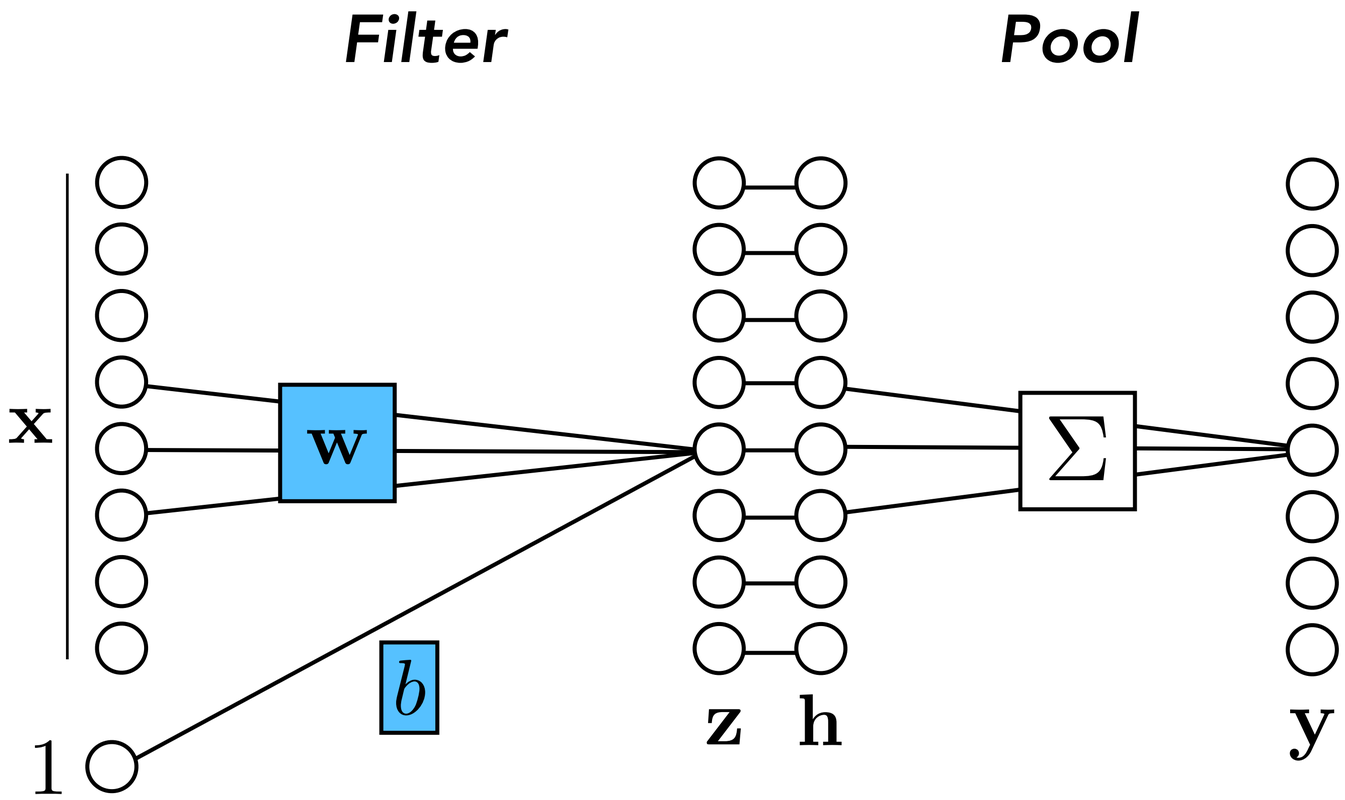
Pooling



Max pooling

$$y_j = \max_{j \in \mathcal{N}(j)} h_j$$

Pooling



Max pooling

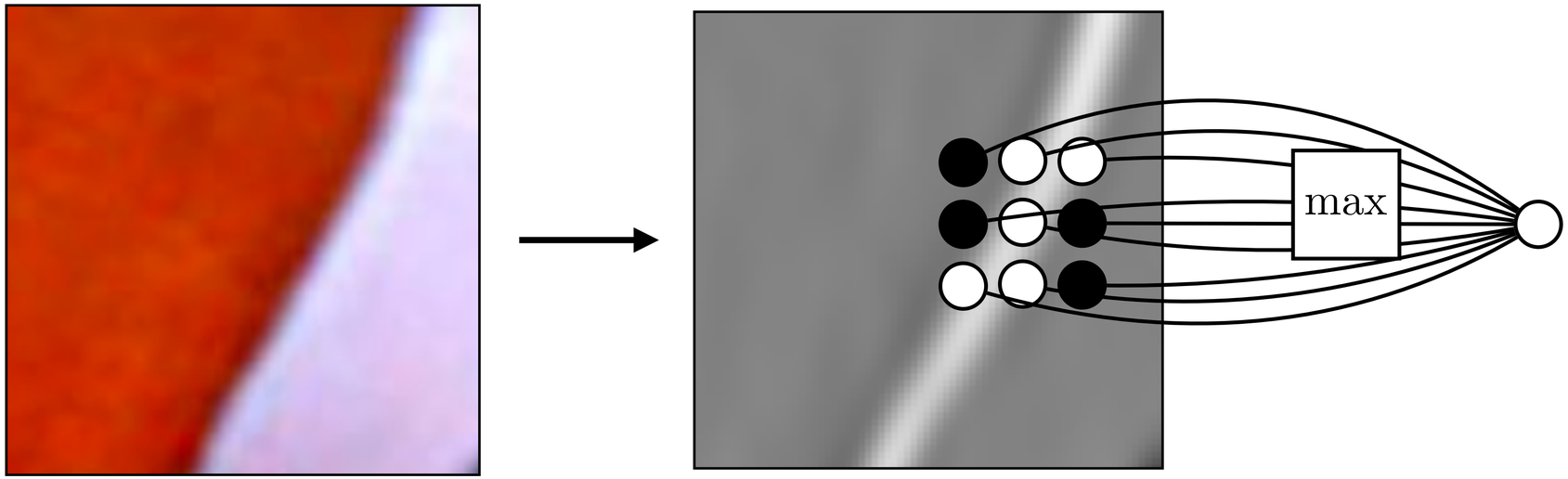
$$y_j = \max_{j \in \mathcal{N}(j)} h_j$$

Mean pooling

$$y_j = \frac{1}{|\mathcal{N}|} \sum_{j \in \mathcal{N}(j)} h_j$$

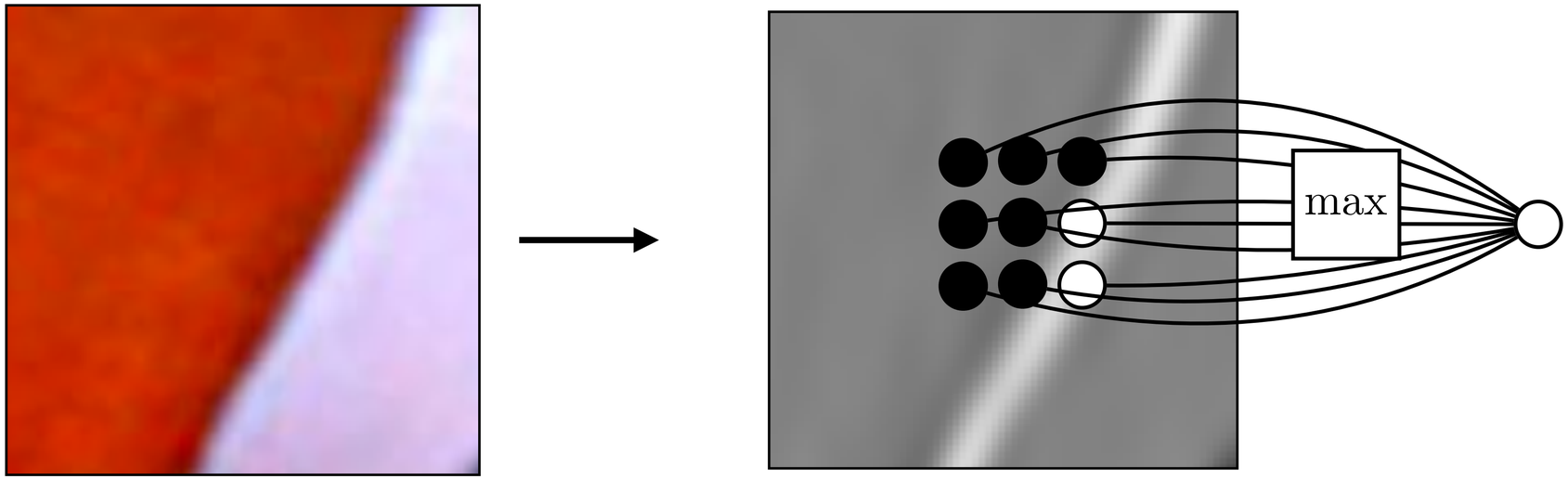
Pooling — Why?

Pooling across spatial locations achieves stability w.r.t. small translations:



Pooling — Why?

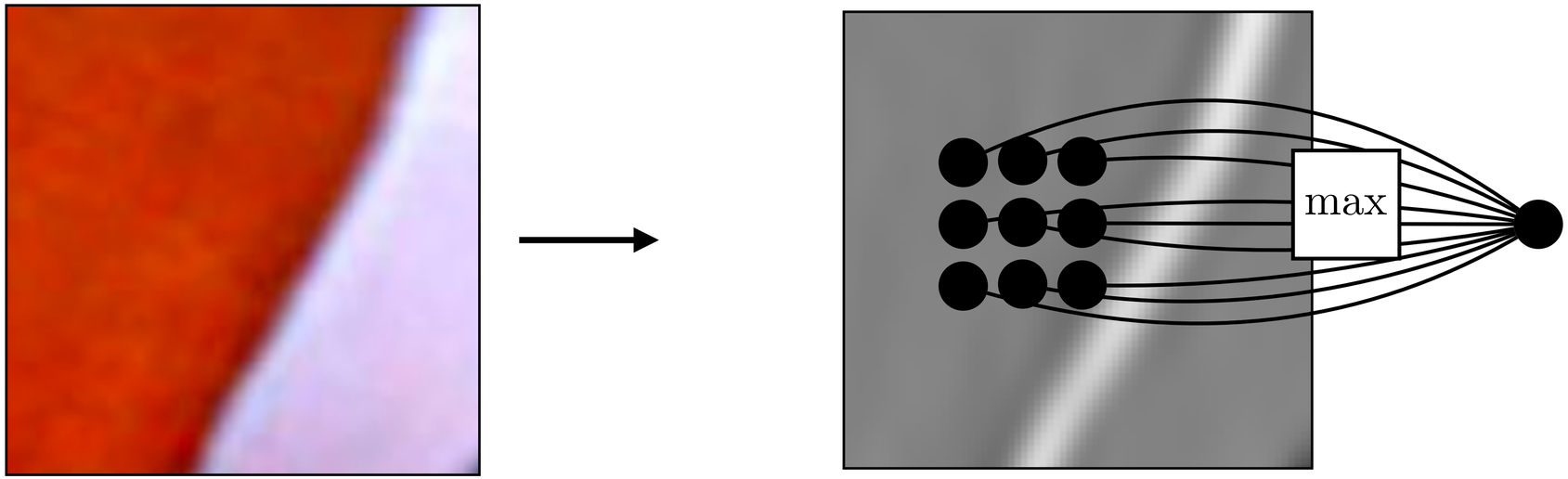
Pooling across spatial locations achieves stability w.r.t. small translations:



large response regardless of exact position of edge

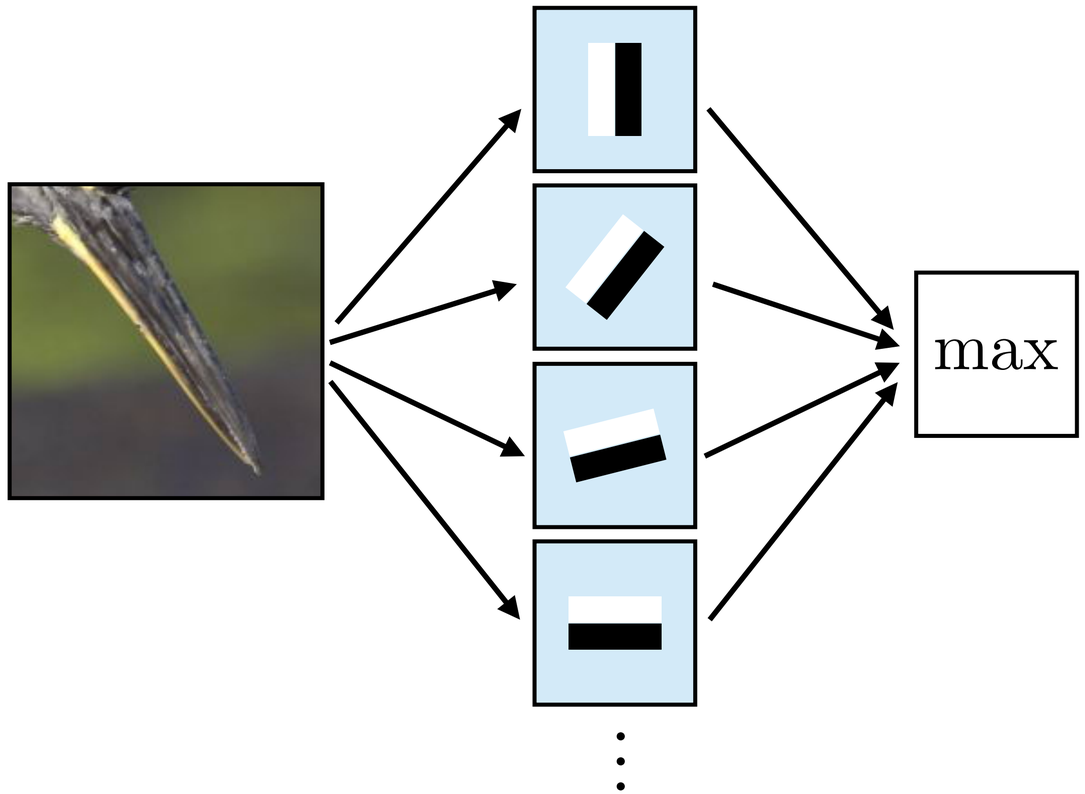
Pooling — Why?

Pooling across spatial locations achieves stability w.r.t. small translations:



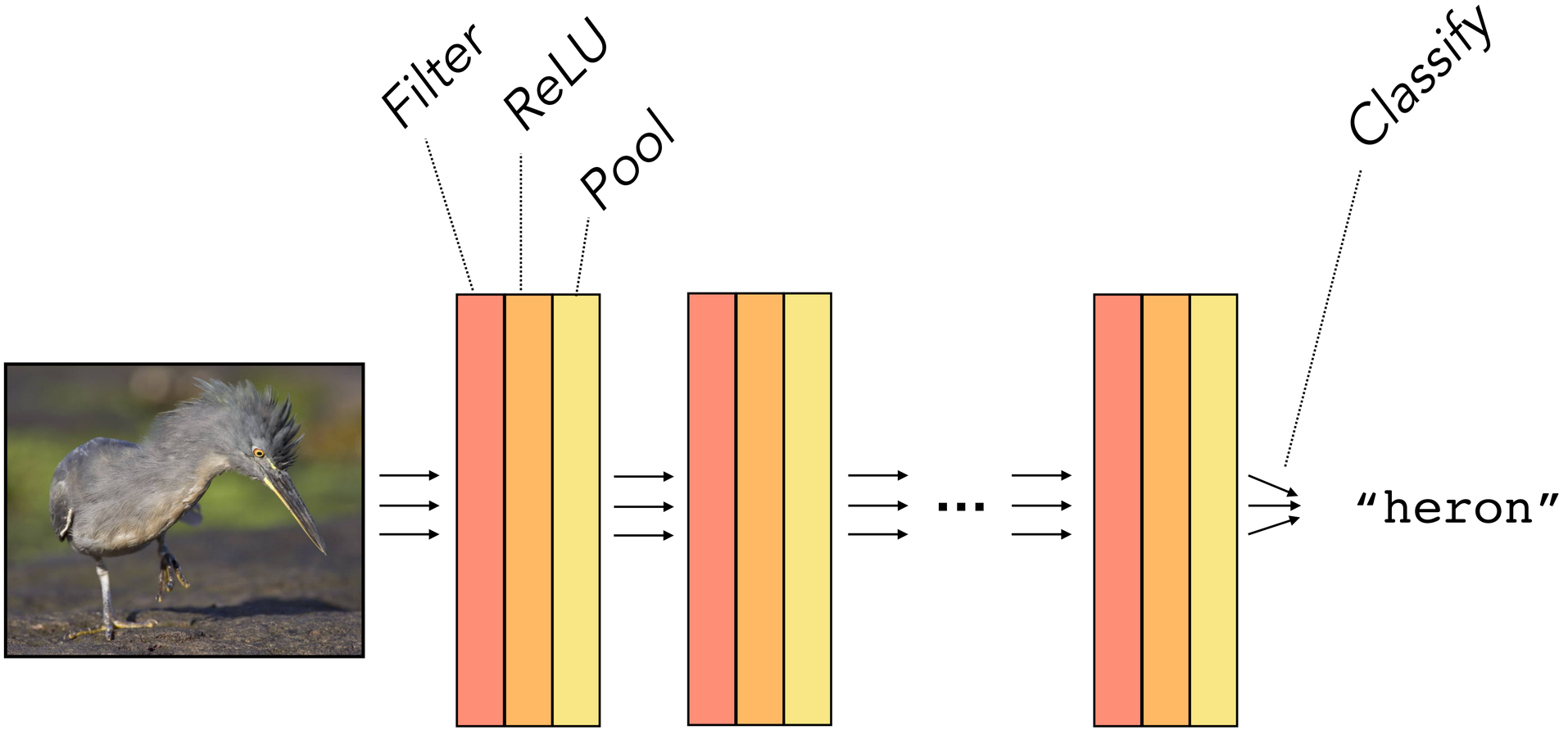
Pooling *across channels* — Why?

Pooling across feature channels (filter outputs)
can achieve other kinds of invariances:



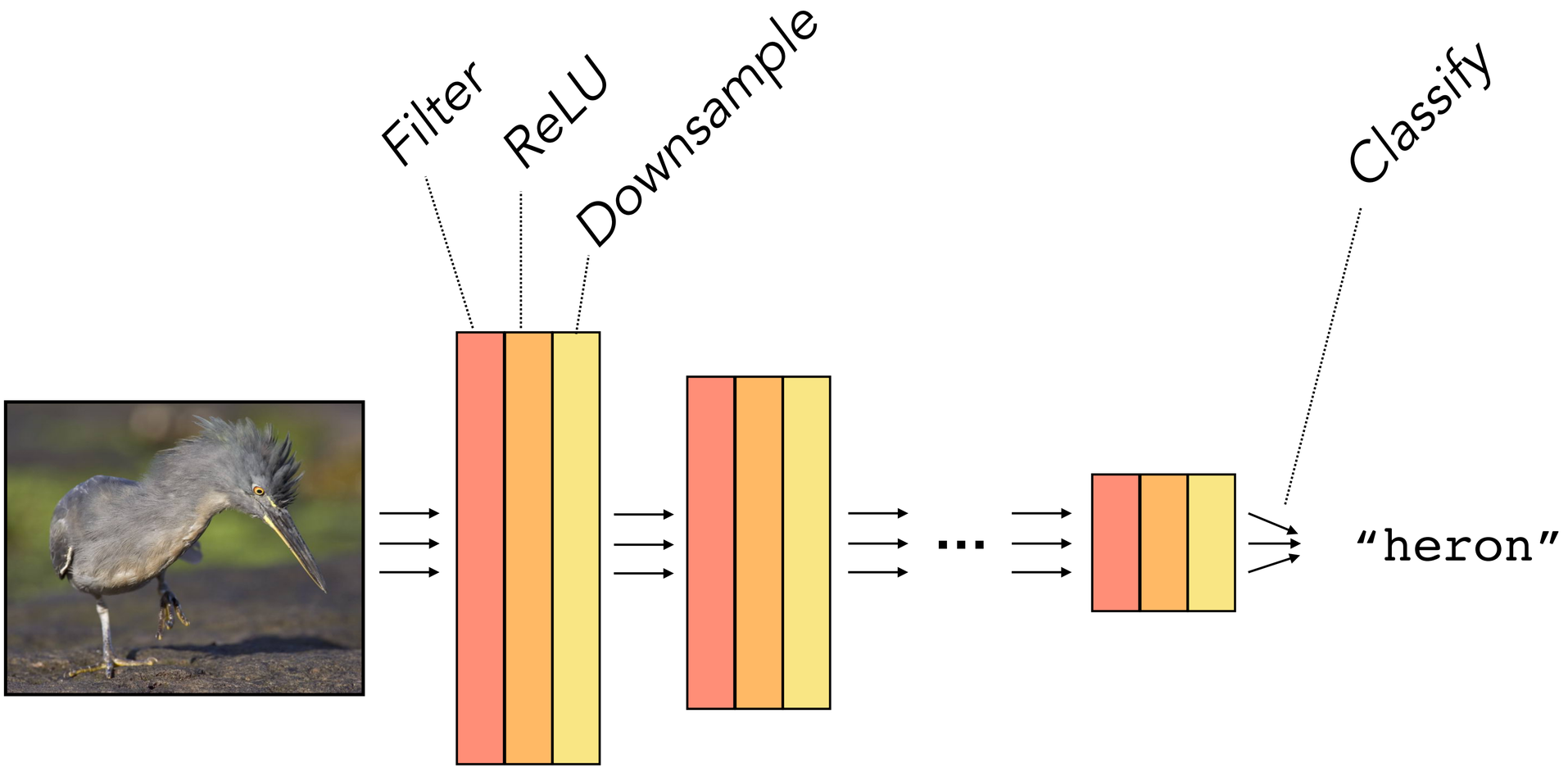
large response
for any edge,
regardless of its
orientation

Computation in a neural net



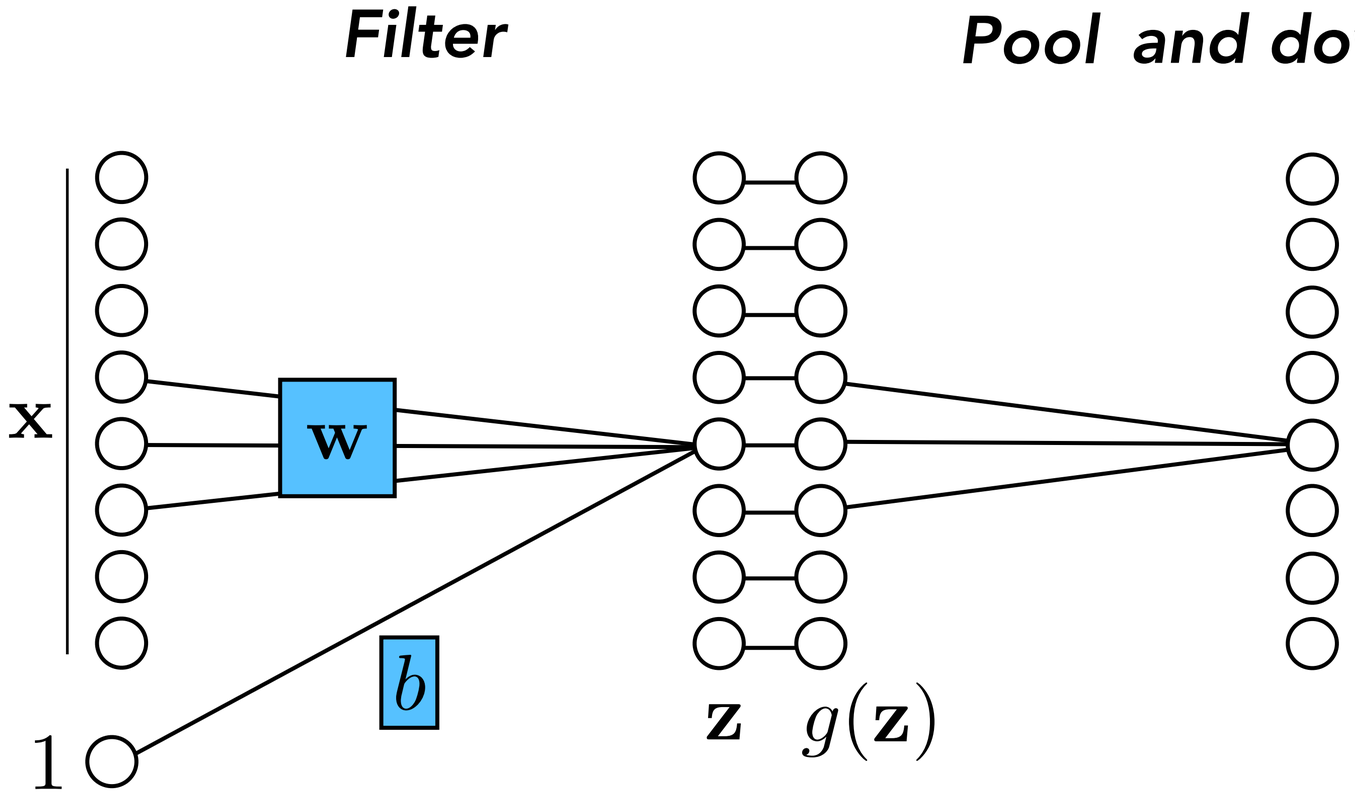
$$f(\mathbf{x}) = f_L(\dots f_2(f_1(\mathbf{x})))$$

Computation in a neural net

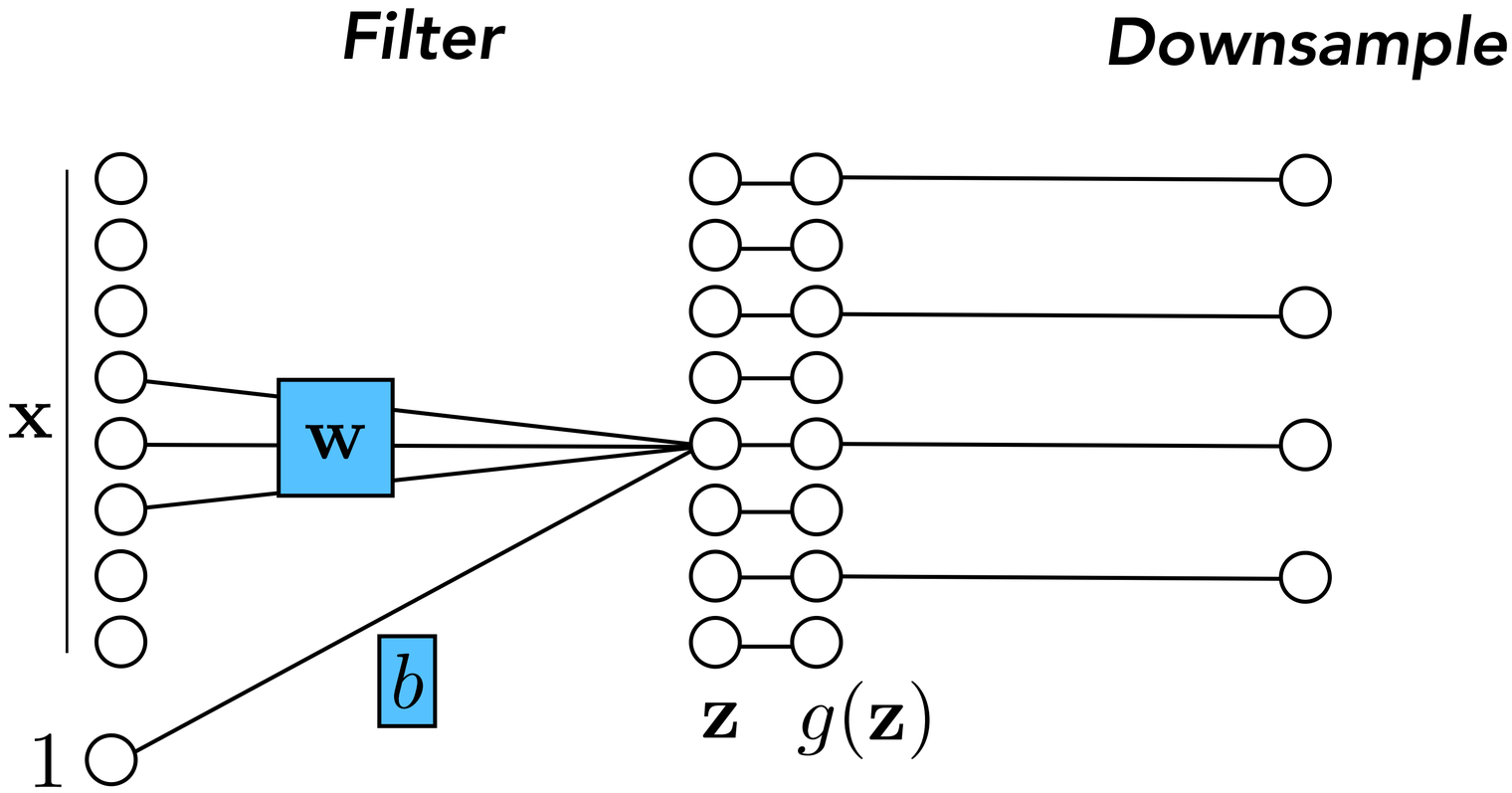


$$f(\mathbf{x}) = f_L(\dots f_2(f_1(\mathbf{x})))$$

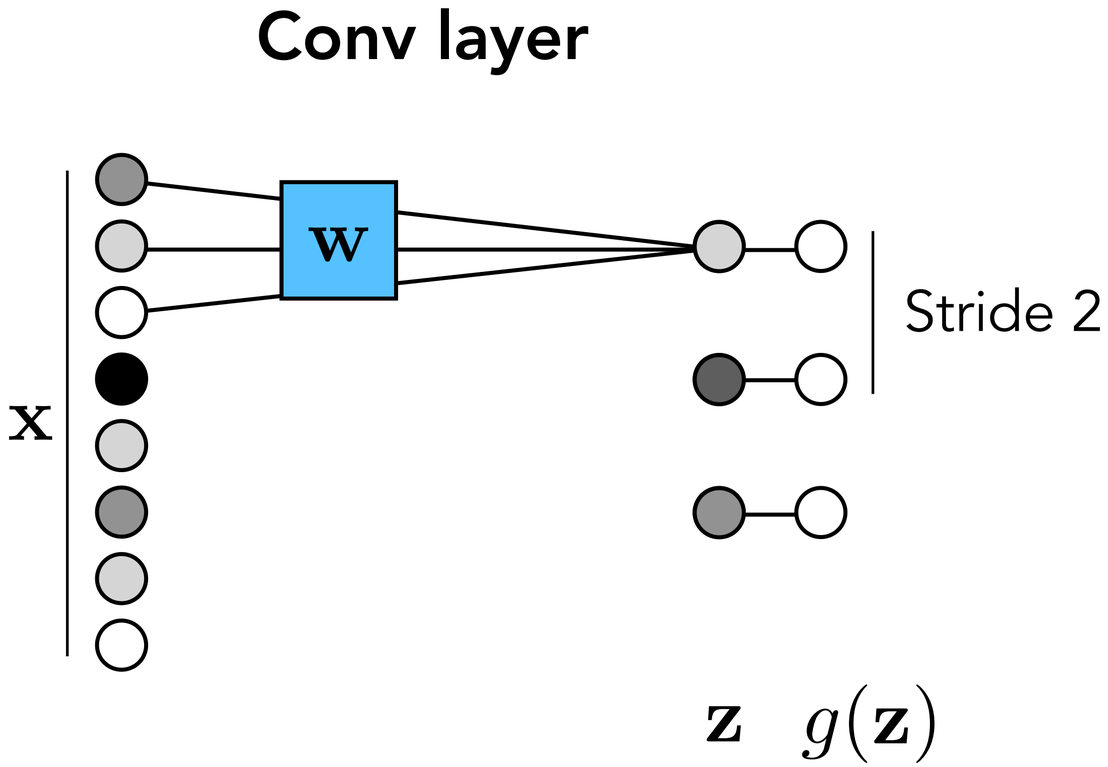
Downsampling



Downsampling

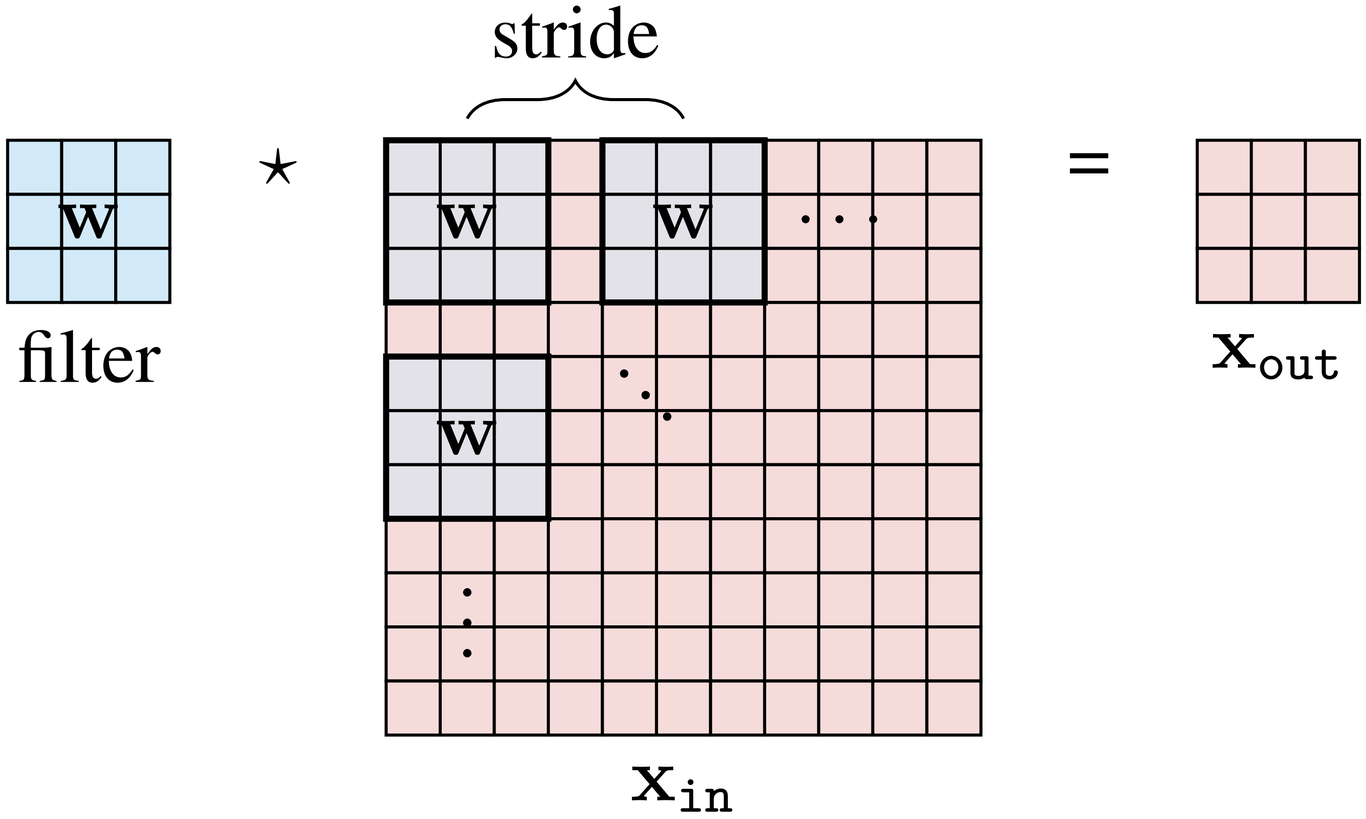


Strided operations

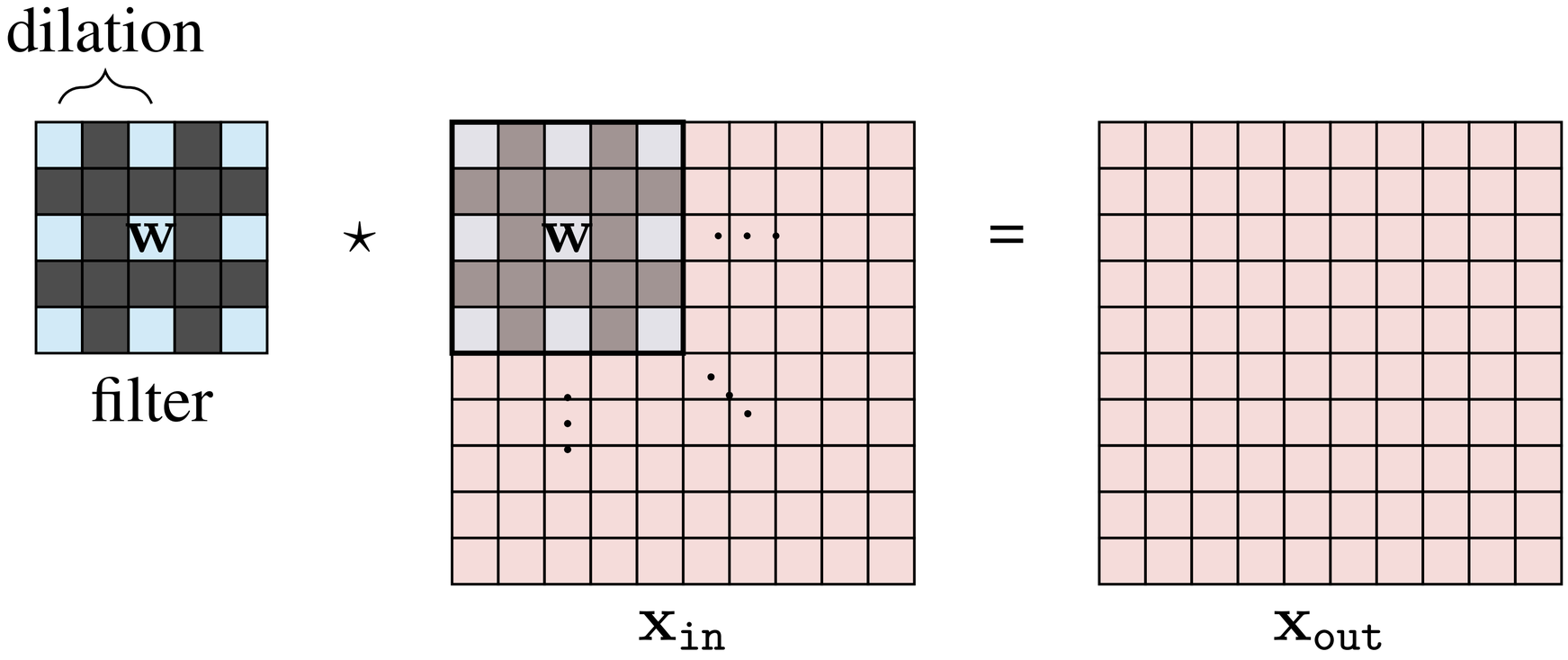


Strided operations combine a given operation (convolution or pooling) and downsampling into a single operation.

Strided operations (2D)

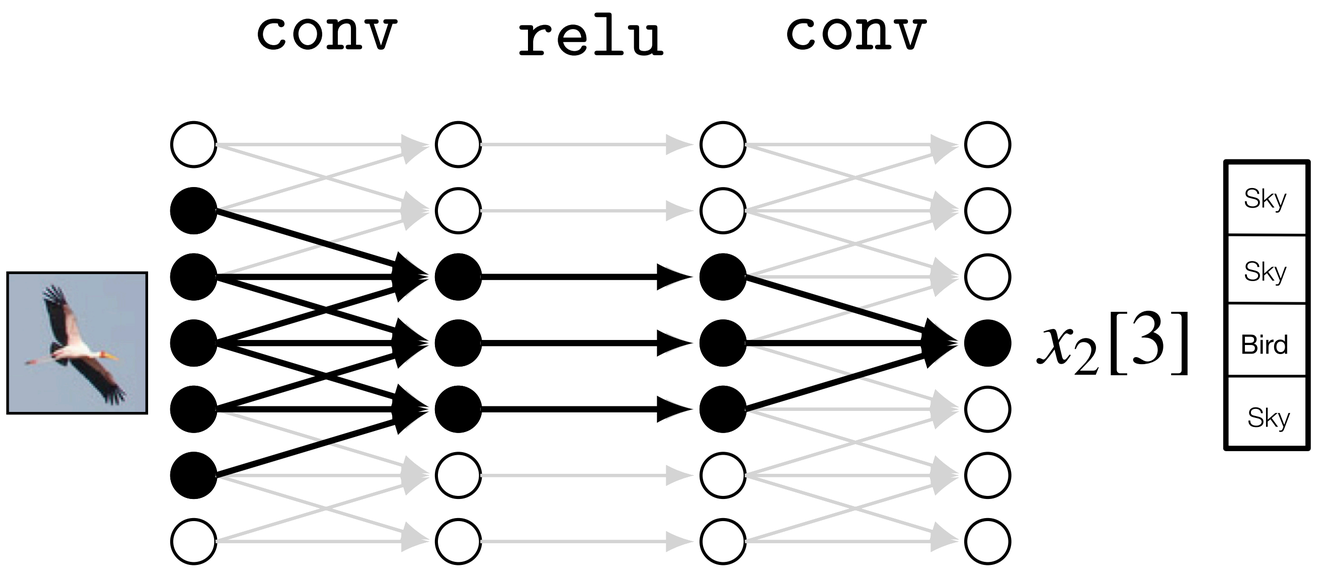


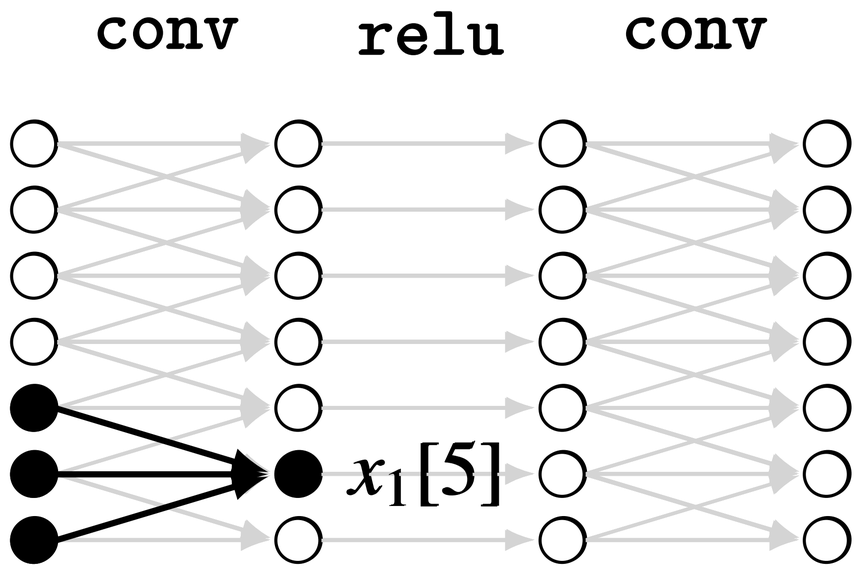
Dilated filter

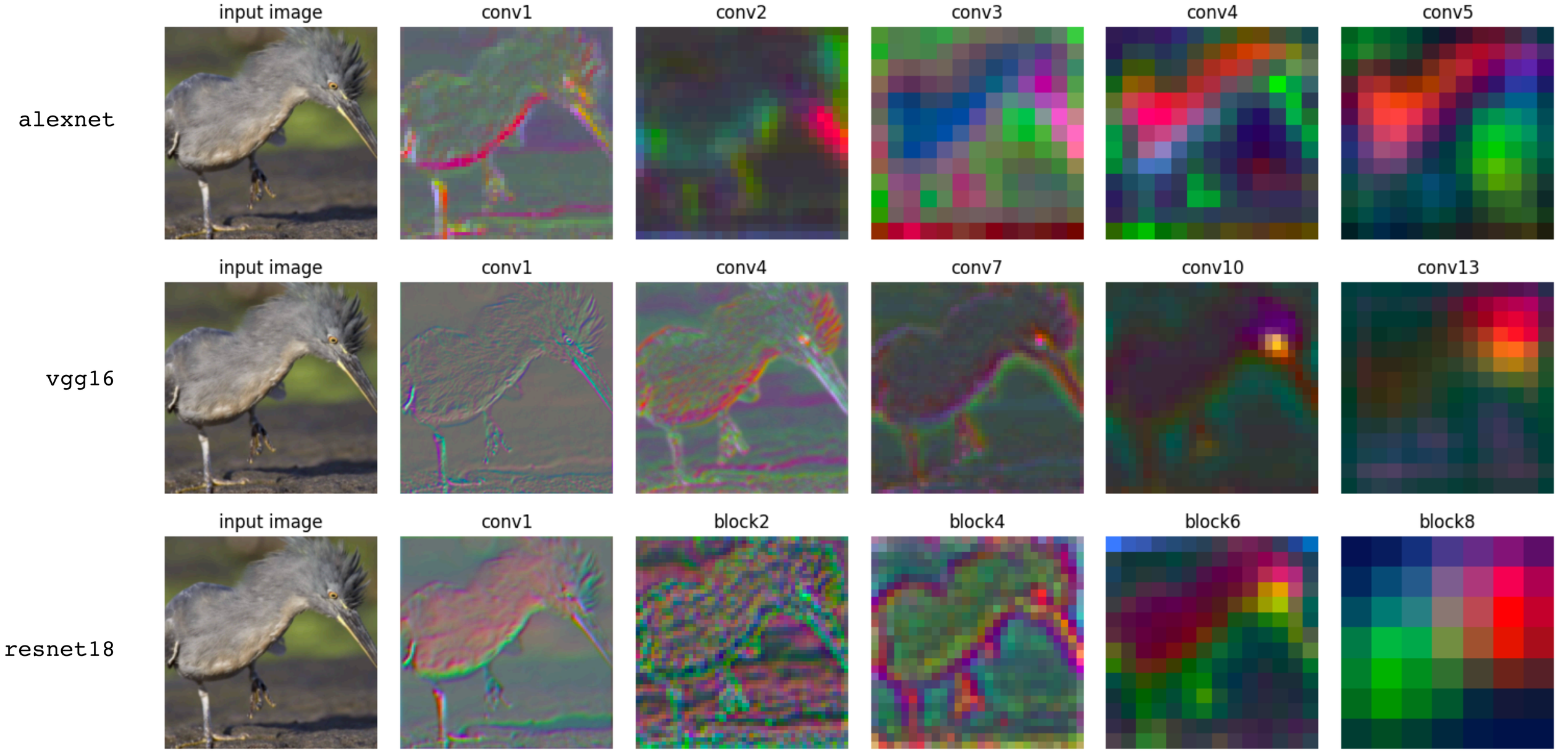


Covers a large receptive field with fewer parameters.

Receptive fields

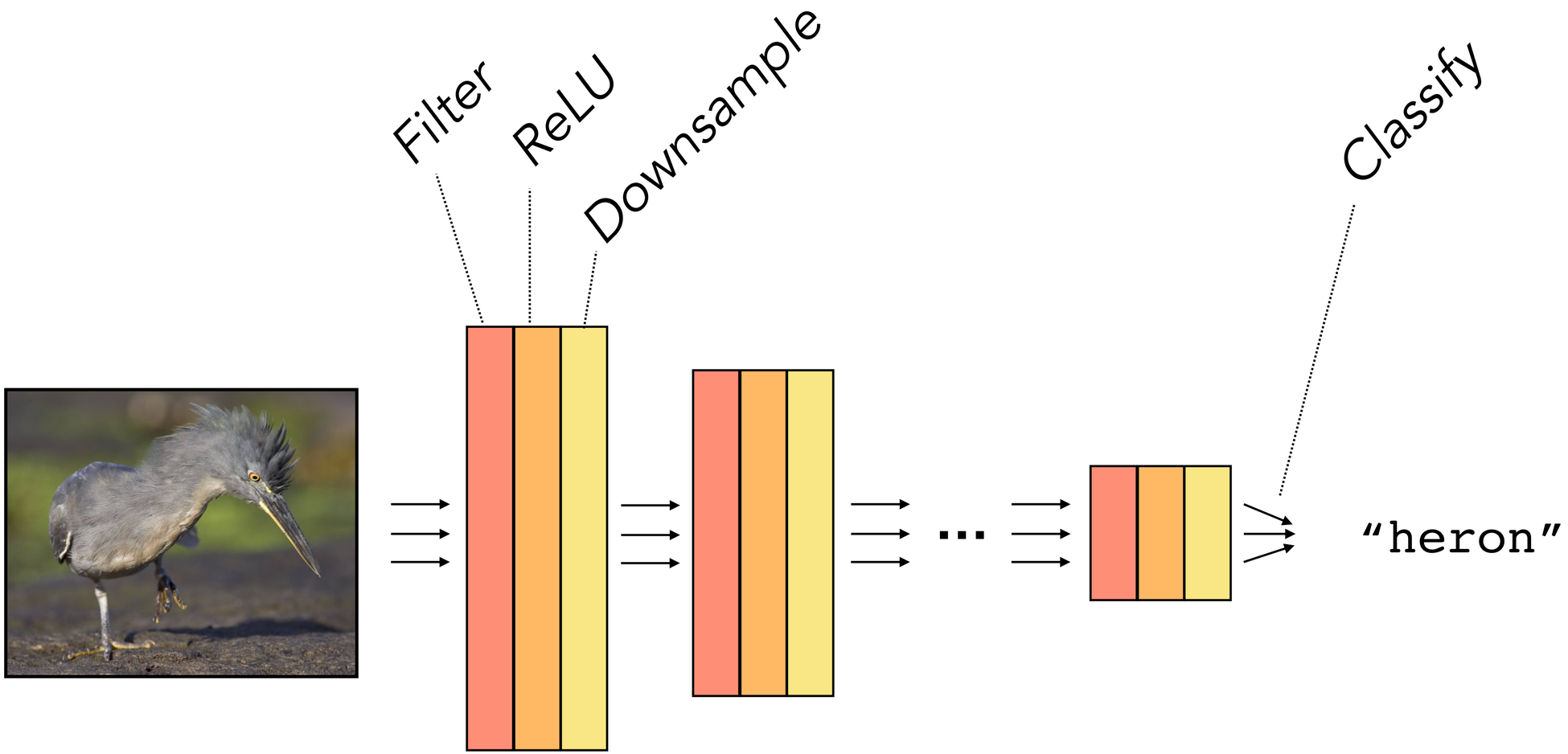




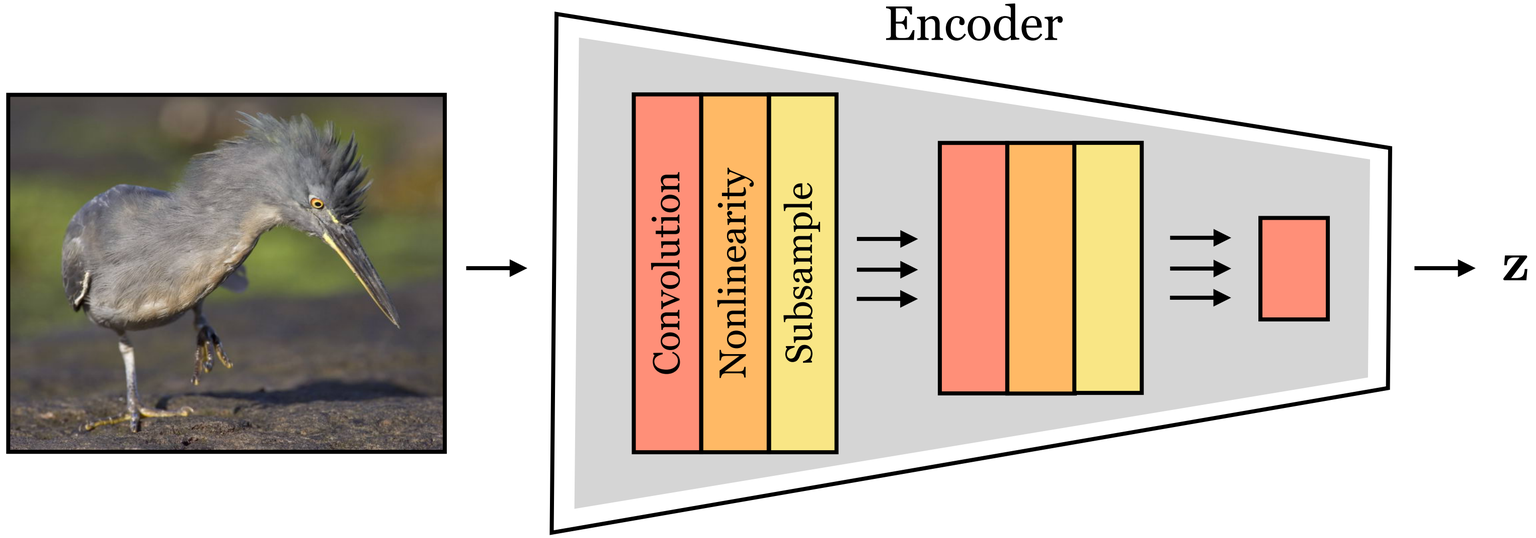

alexnet
input image
conv1
conv2
conv3
conv4
conv5
vgg16
input image
conv1
conv4
conv7
conv10
conv13
resnet18
input image
conv1
block2
block4
block6
block8

Popular CNN Architectures

Computation in a neural net



$$f(\mathbf{x}) = f_L(\dots f_2(f_1(\mathbf{x})))$$


Encoder
Convolution
Nonlinearity
Subsample
z

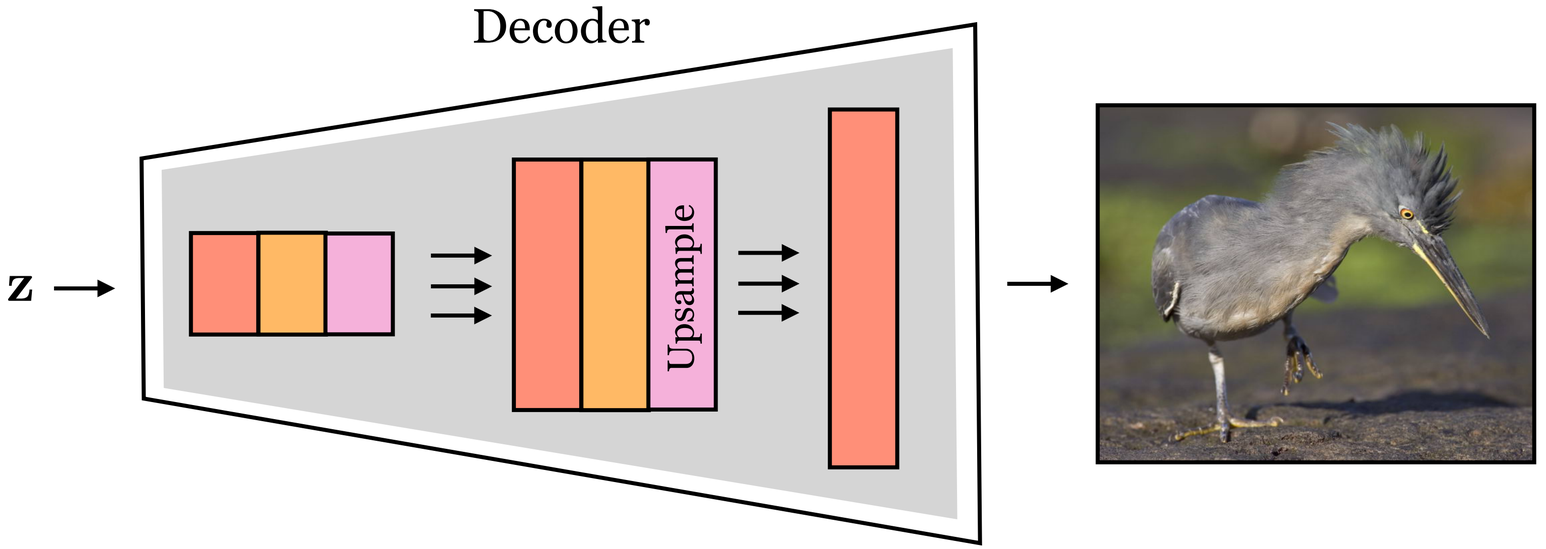

Decoder
z
Upsample

Image-to-image

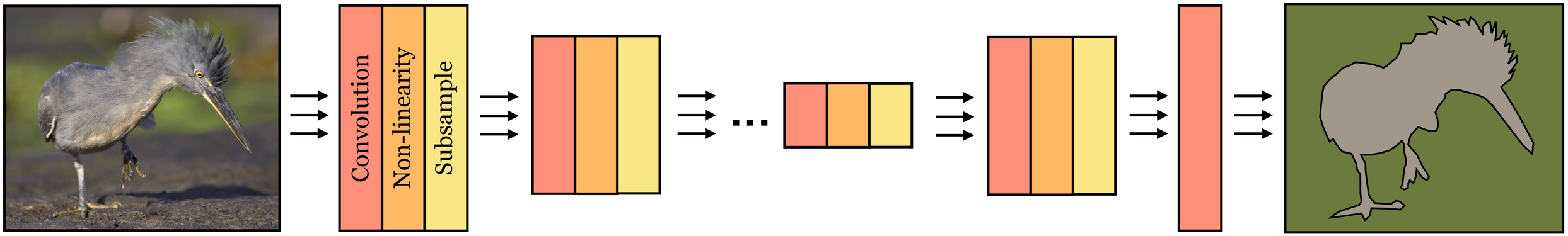
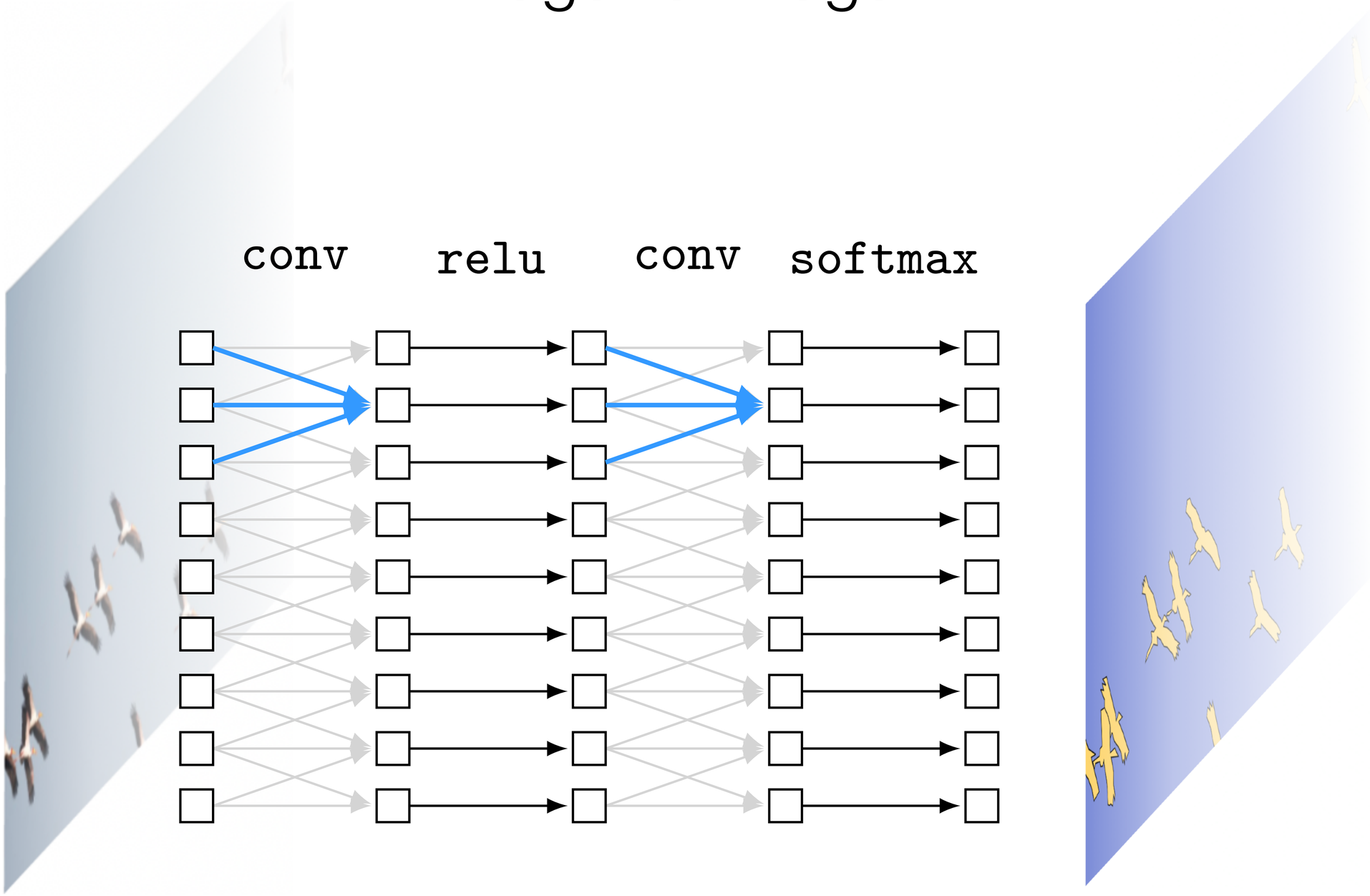
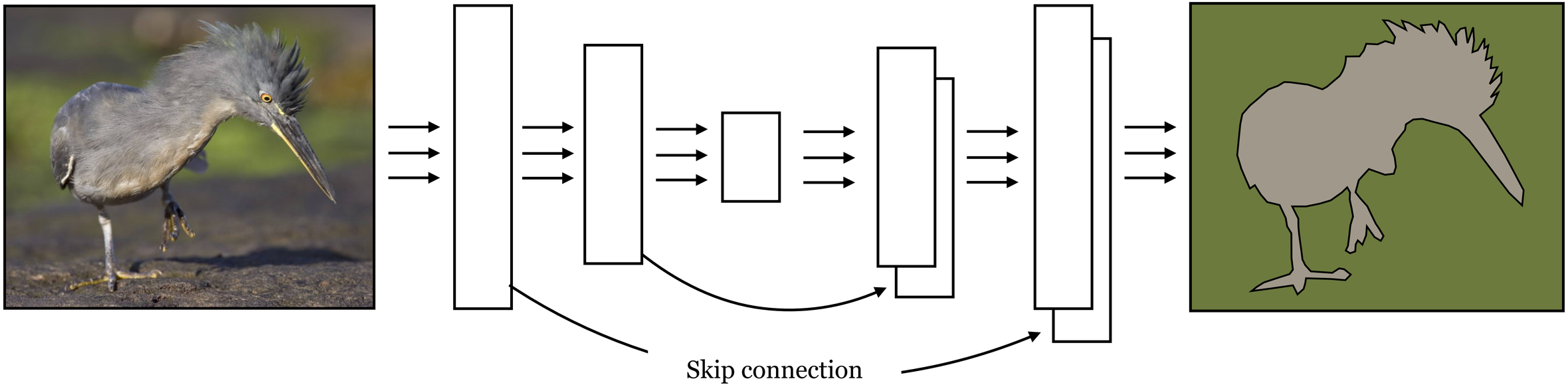


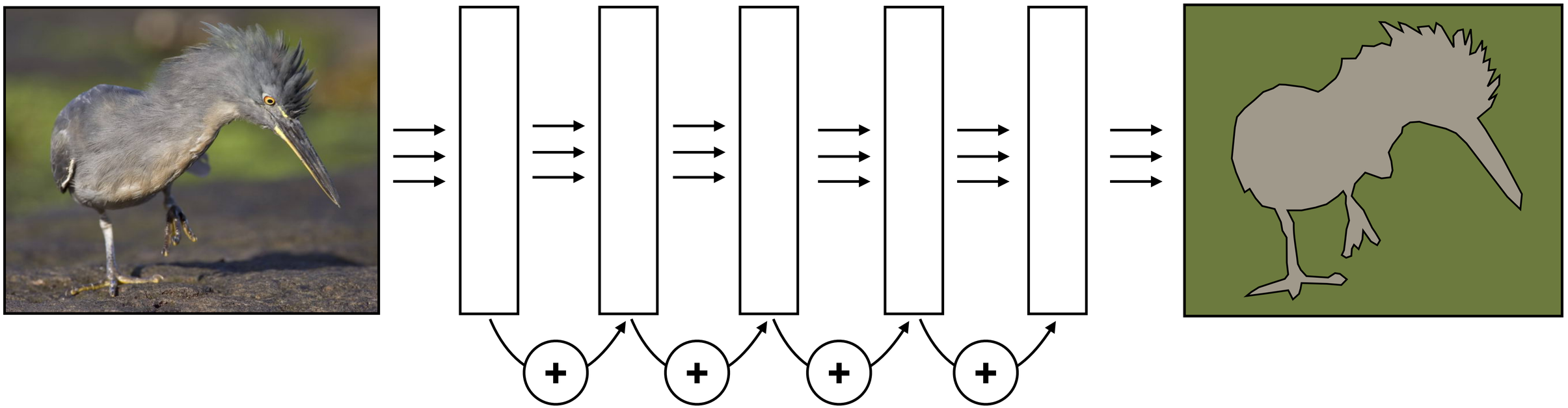
Image-to-image



U-net



ResNet



Residual connection: $\mathbf{x}_{\text{out}} = F(\mathbf{x}_{\text{in}}) + \mathbf{x}_{\text{in}}$

Or, if you want to change dimensionality: $\mathbf{x}_{\text{out}} = F(\mathbf{x}_{\text{in}}) + \mathbf{W}\mathbf{x}_{\text{in}}$