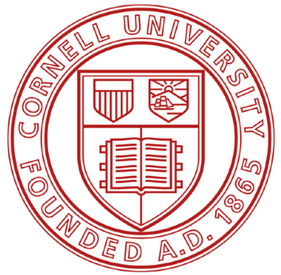
Lecture 4: Neural network review

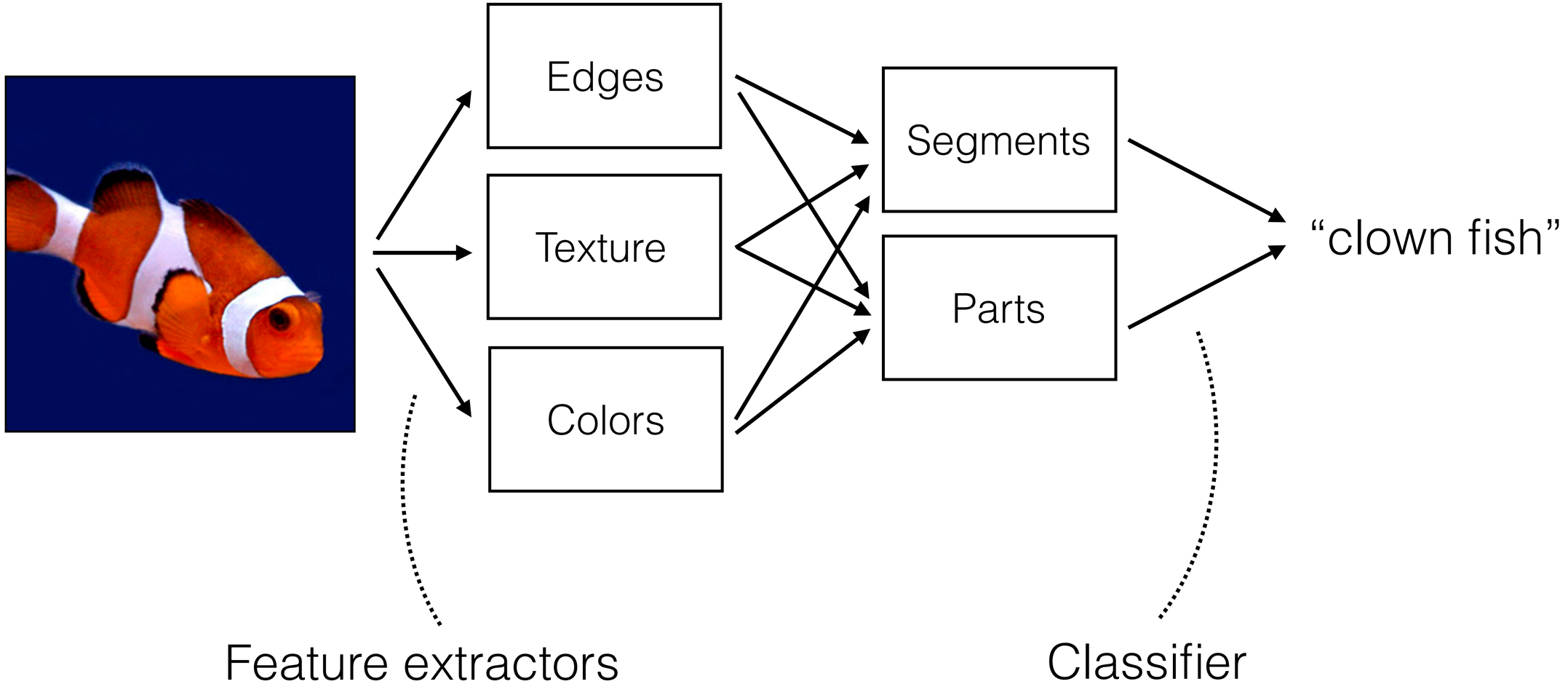
CS 5788: Introduction to Generative Models



Today

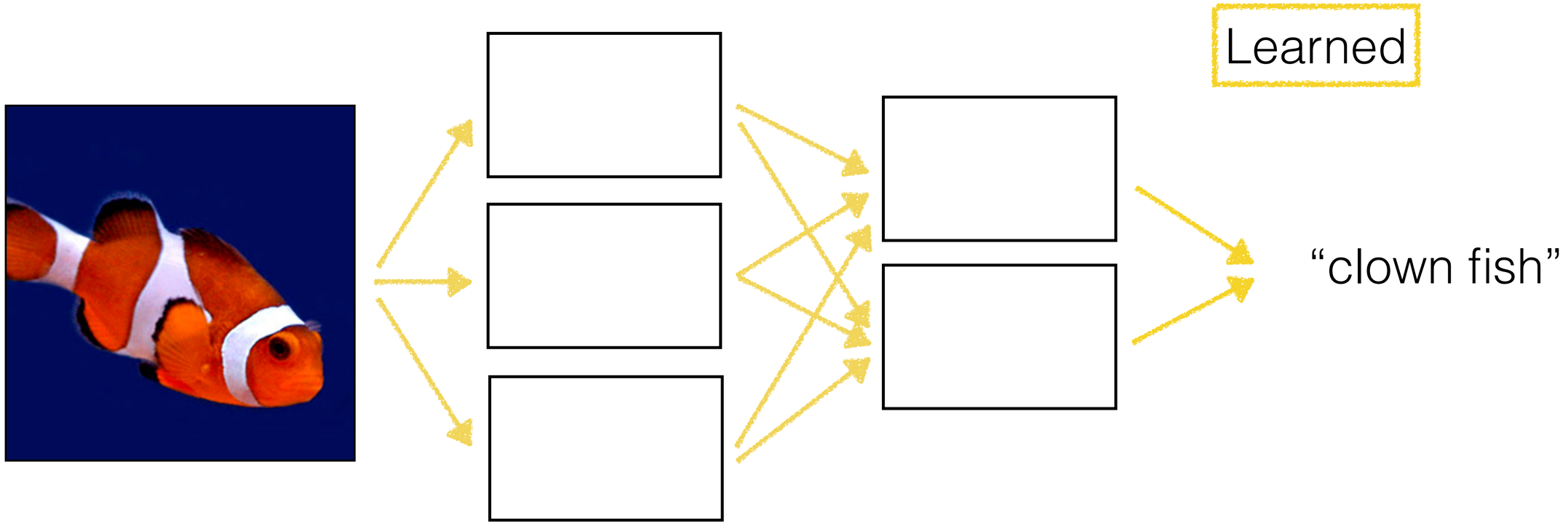
- Neural net review
  - MLPs, CNNs, and (very briefly) backprop
  - Won’t discuss transformers (we’ll save that for later).
- You won’t be tested on this, but you need it for understanding and implementing deep generative models

“Classic” recognition **without** neural nets



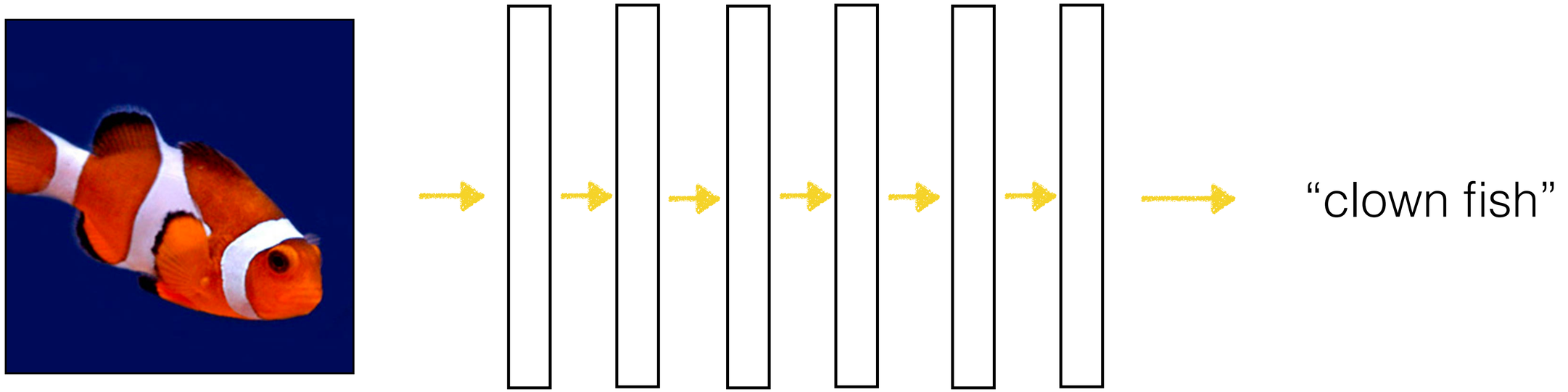
Source: Isola, Torralba, Freeman

Object recognition



Source: Isola, Torralba, Freeman

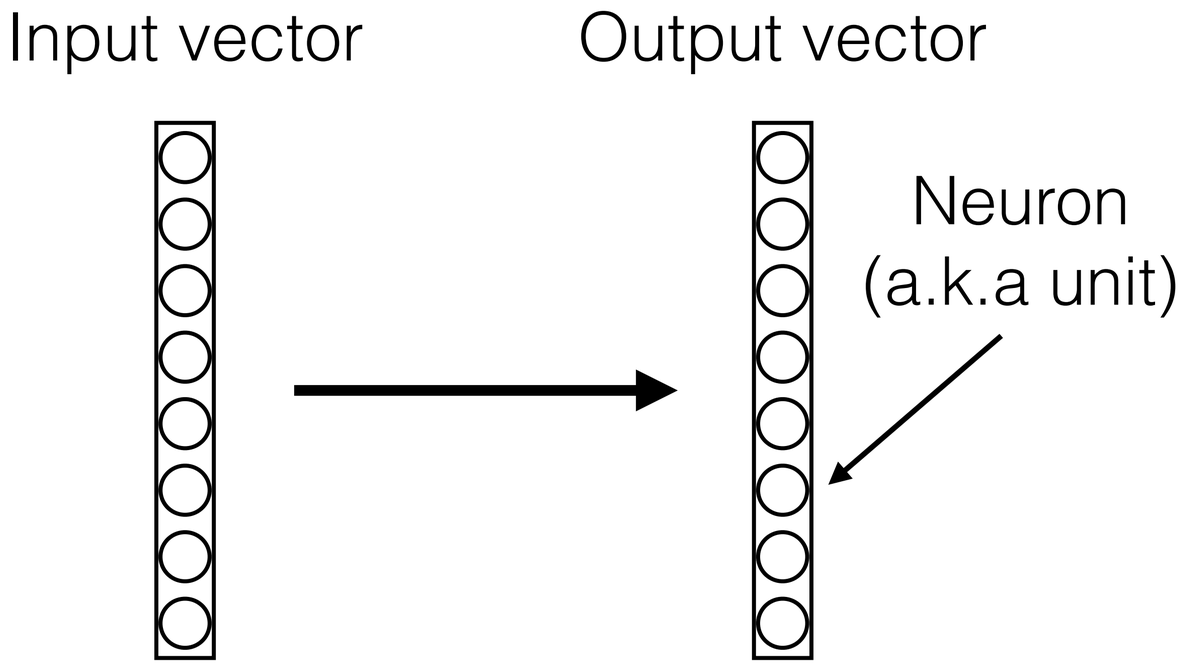
Neural net (for recognition)



Neural network

Source: Isola, Torralba, Freeman

Computation in a neural net



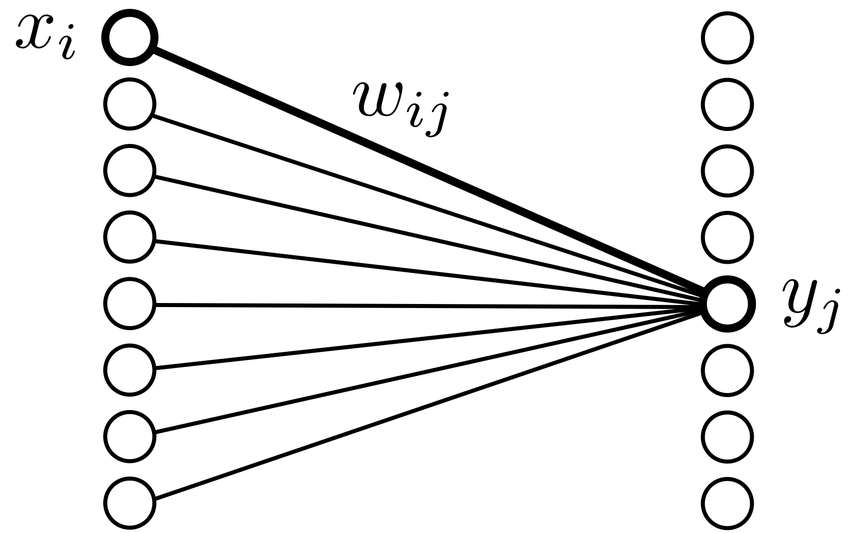
Adapted from: Isola, Torralba, Freeman

Computation in a neural net

Linear layer

Input representation

Output representation

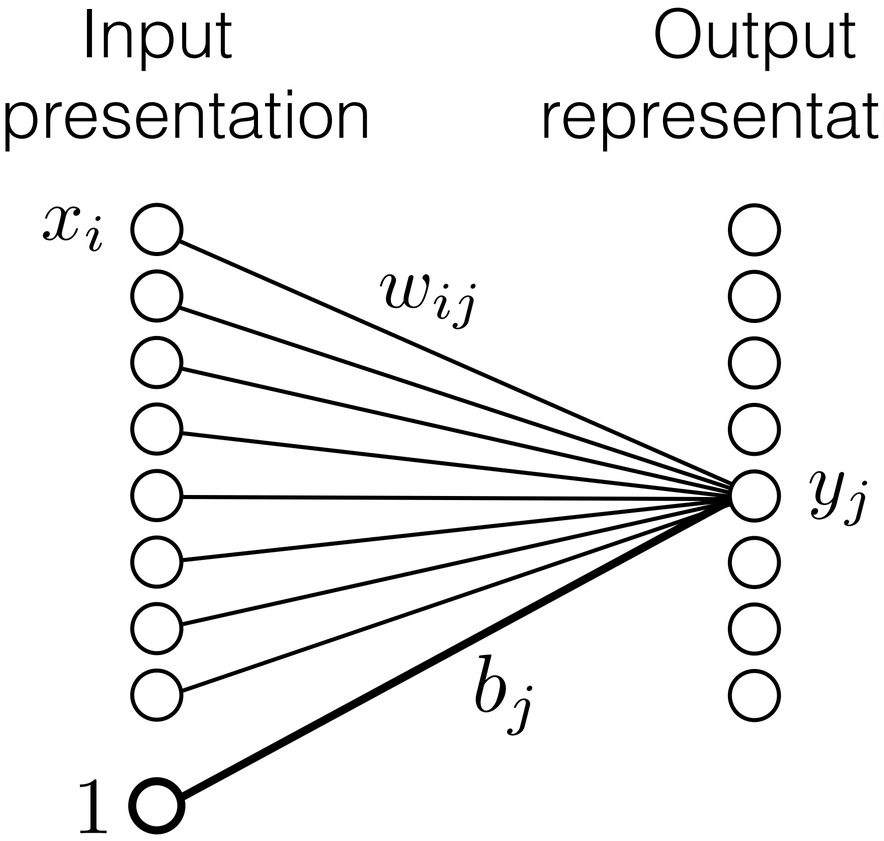


$$y_j = \sum_i w_{ij} x_i$$

Source: Isola, Torralba, Freeman

Computation in a neural net

Linear layer



$$y_j = \sum_i w_{ij} x_i + b_j$$

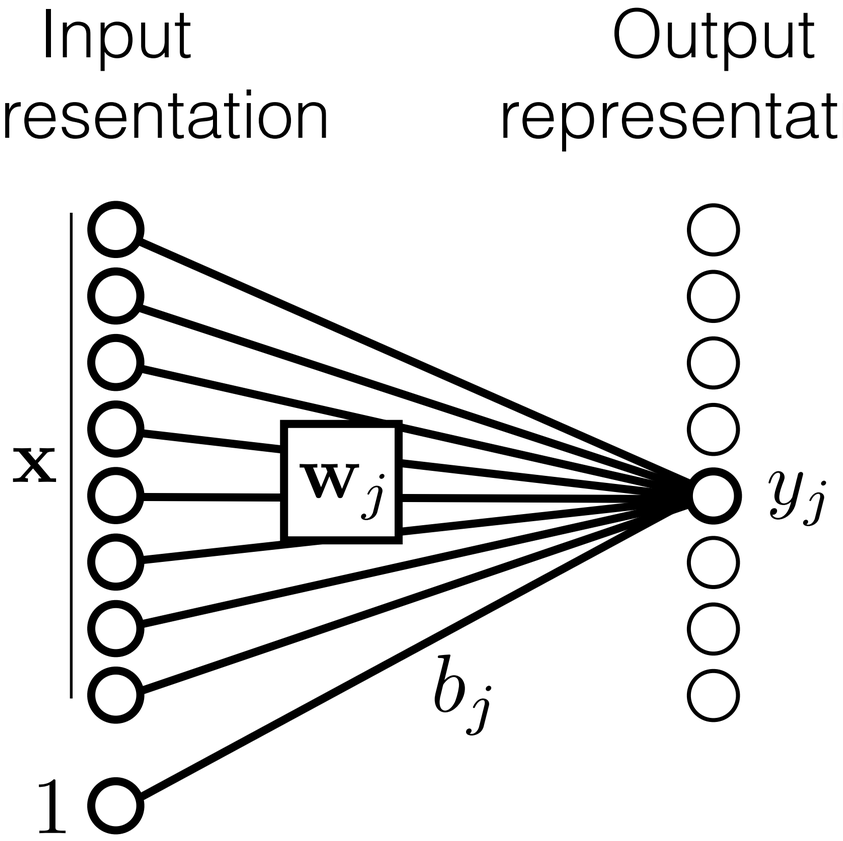
weights

bias

Source: Isola, Torralba, Freeman

Computation in a neural net

Linear layer



$$y_j = \mathbf{x}^T \mathbf{w}_j + b_j$$

weights

bias

$$\theta = \{\mathbf{W}, \mathbf{b}\}$$

parameters of the model

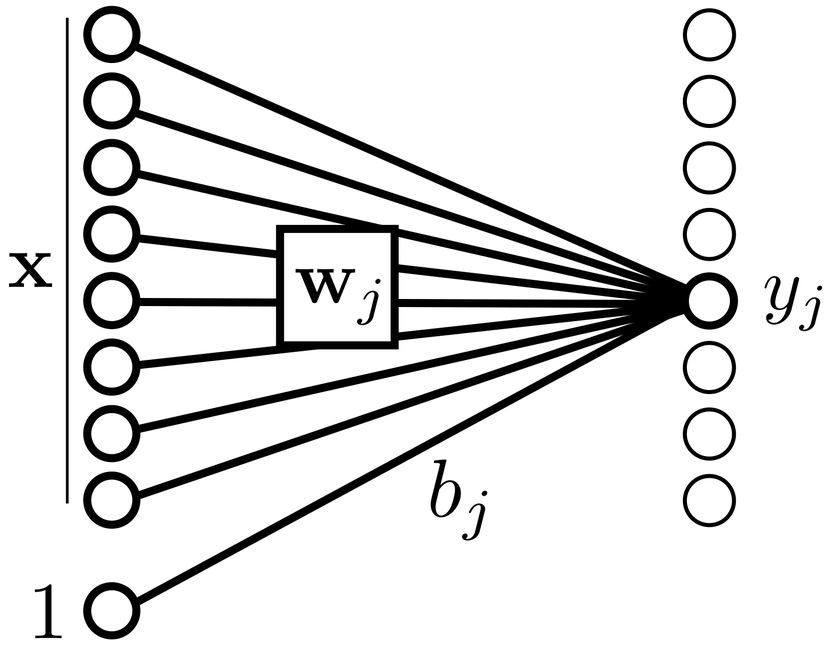
Source: Isola, Torralba, Freeman

Computation in a neural net

Linear layer

Input representation

Output representation



Full layer

weights

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

bias

$$\theta = \{\mathbf{W}, \mathbf{b}\}$$

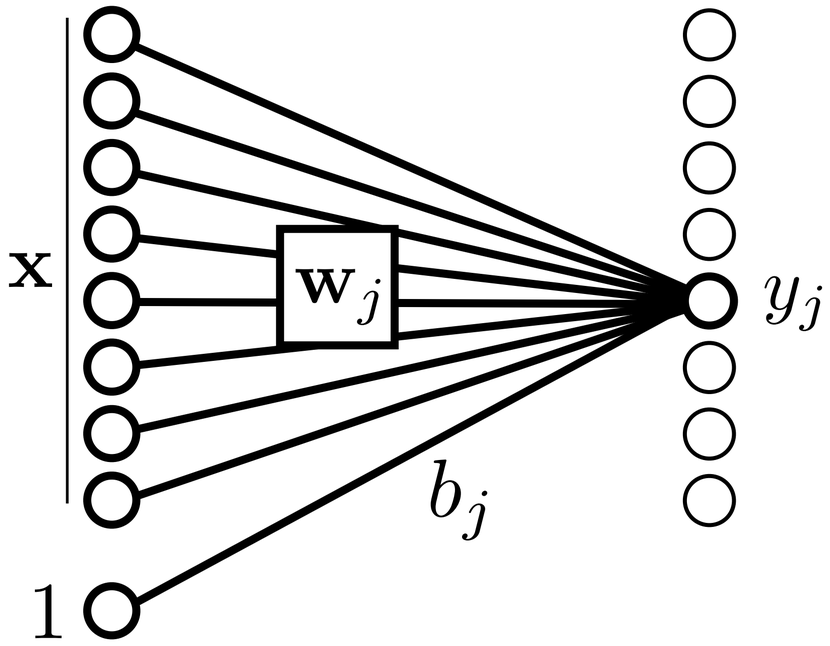
parameters of the model

Computation in a neural net

Linear layer

Input representation

Output representation



Full layer

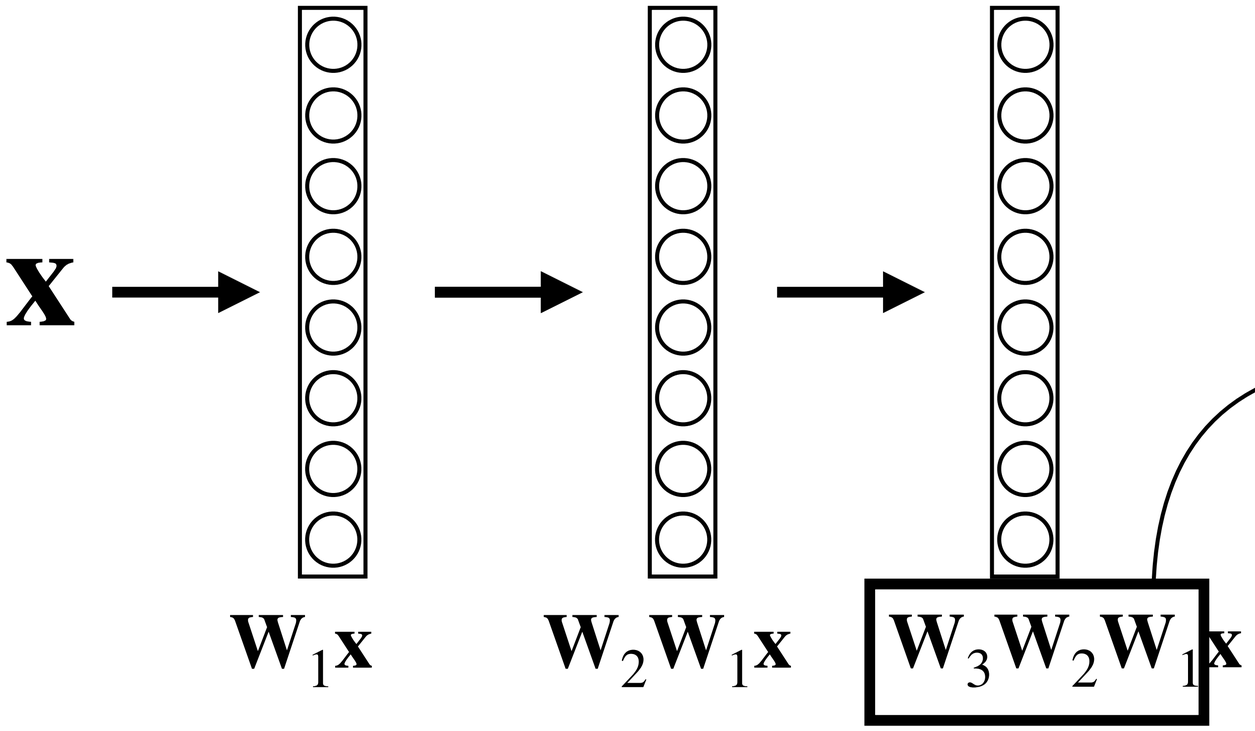
weights + bias

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

Can again simplify notation by appending a 1 to $\mathbf{x}$

What's the problem with this idea?

Consider stacking **multiple** layers:



Can be expressed as single linear layer!

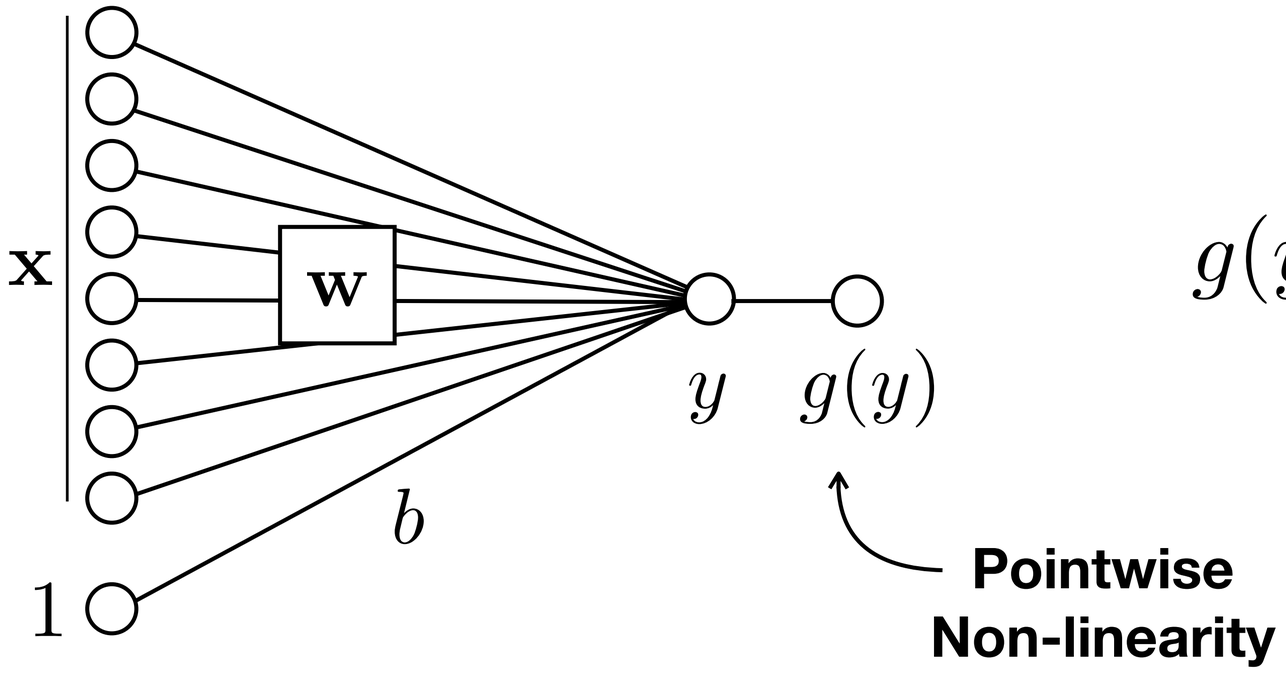
$$\left(\prod_i \mathbf{W}_i\right)\mathbf{x} = \hat{\mathbf{W}}\mathbf{x}$$

Limited power.

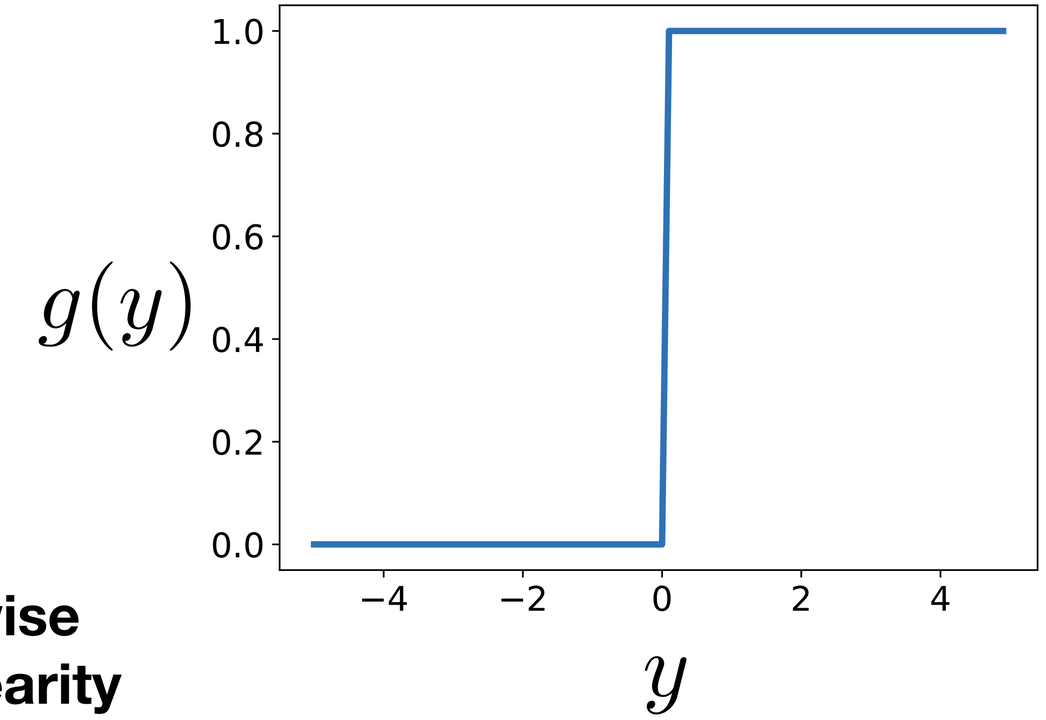
Solution: simple nonlinearity

Input representation

Output representation



$$g(y) = \begin{cases} 1, & \text{if} \quad y > 0 \\ 0, & \text{otherwise} \end{cases}$$

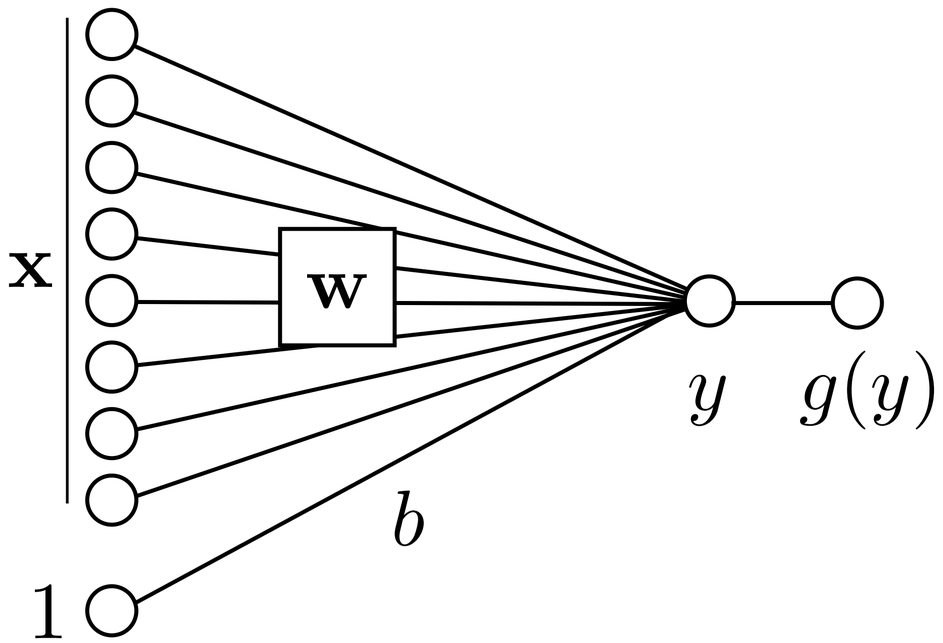


Source: Isola, Torralba, Freeman

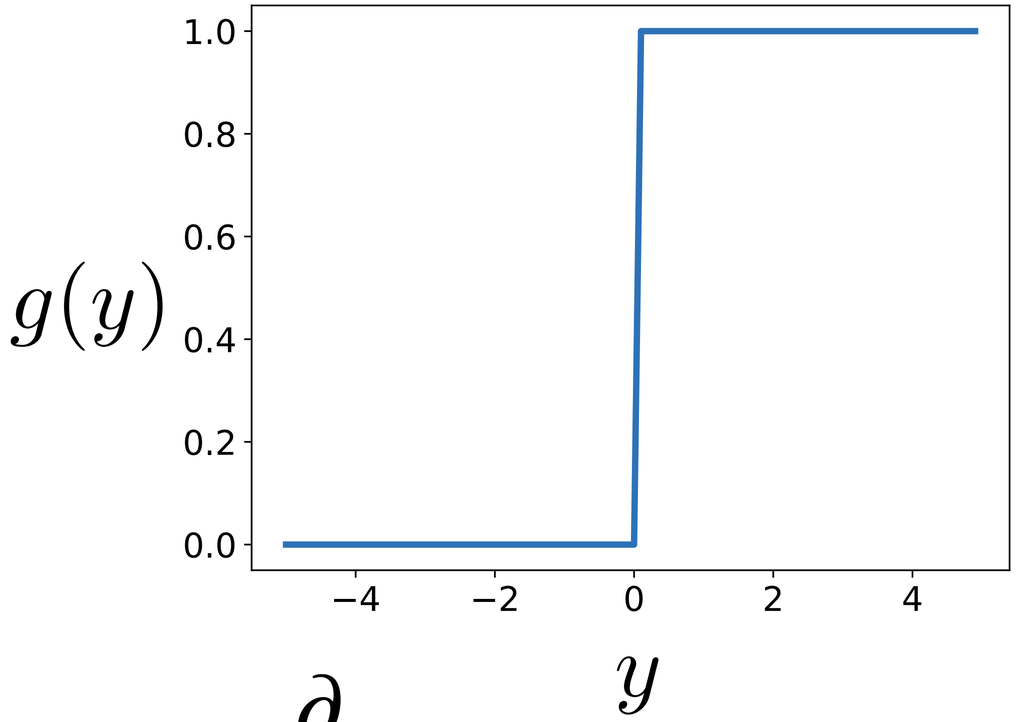
Computation in a neural net — nonlinearity

Input representation

Output representation



$$g(y) = \begin{cases} 1, & \text{if} \quad y > 0 \\ 0, & \text{otherwise} \end{cases}$$



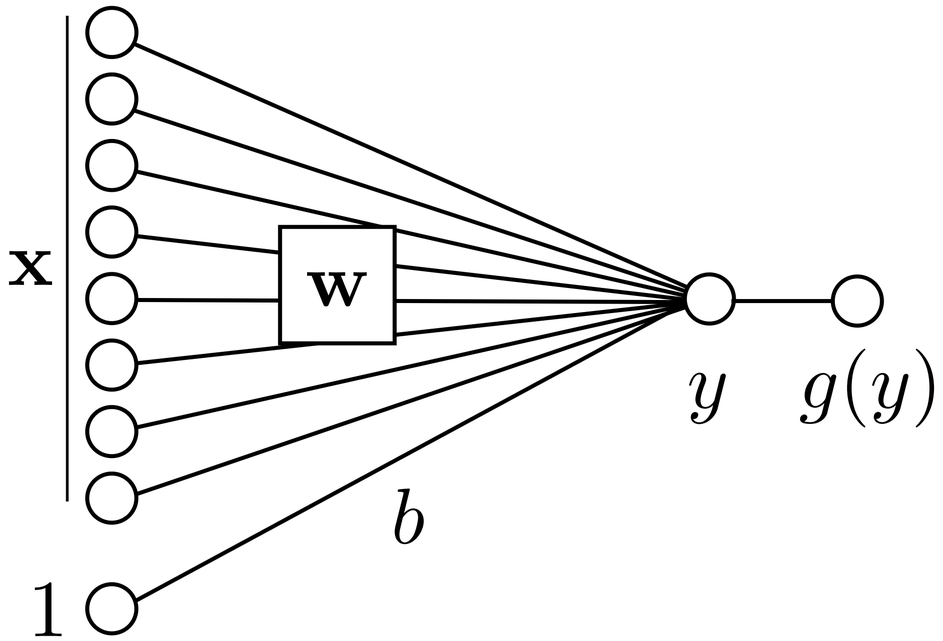
Can't use with gradient descent, $\frac{\partial}{\partial y} g = 0$

Source: Isola, Torralba, Freeman

Computation in a neural net — nonlinearity

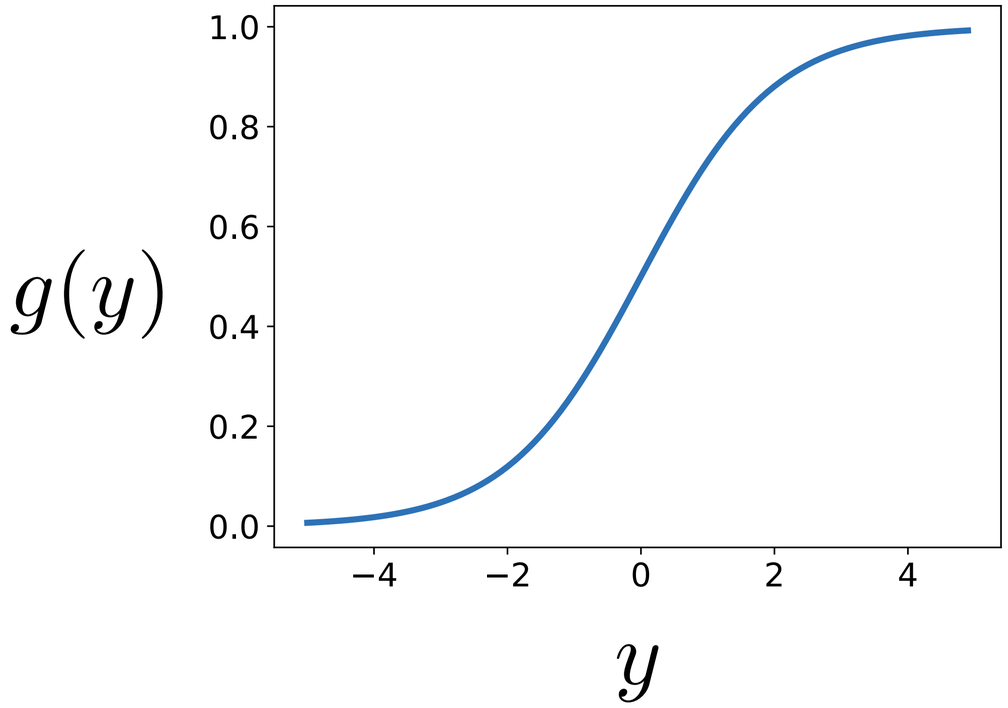
Input representation

Output representation



Sigmoid

$$g(y) = \sigma(y) = \frac{1}{1 + e^{-y}}$$



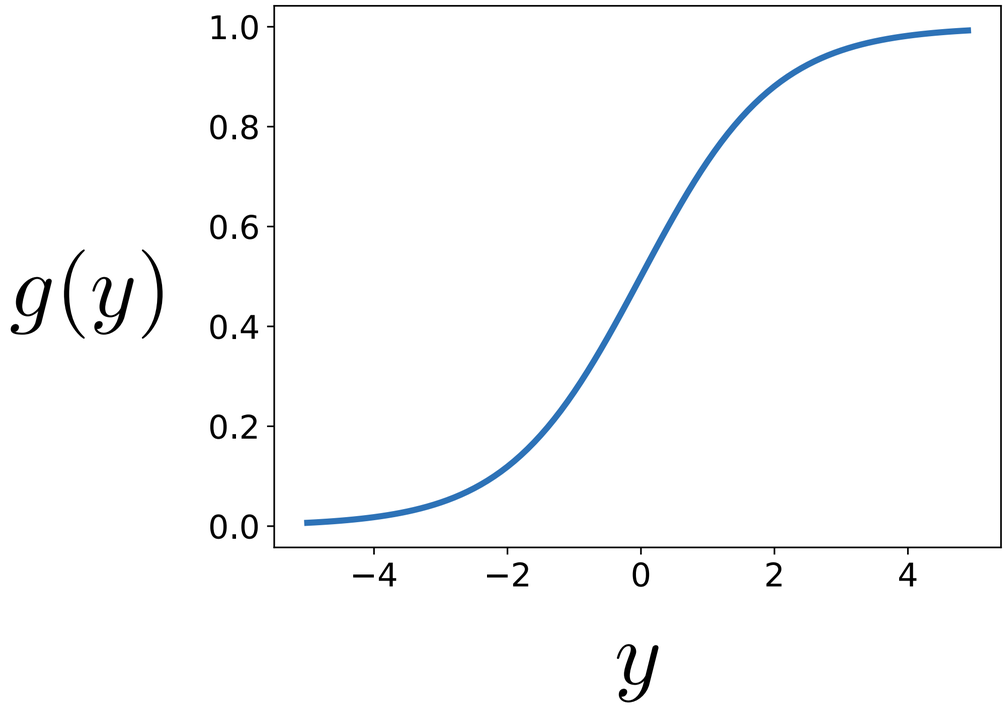
Source: Isola, Torralba, Freeman

Computation in a neural net — nonlinearity

- Interpretation as firing rate of neuron
- Bounded between [0,1]
- Saturation for large +/- inputs
- Gradients go to zero
- Centered at 0.5. Better in practice to use: $\tanh(y) = 2g(y) - 1$

Sigmoid

$$g(y) = \sigma(y) = \frac{1}{1 + e^{-y}}$$



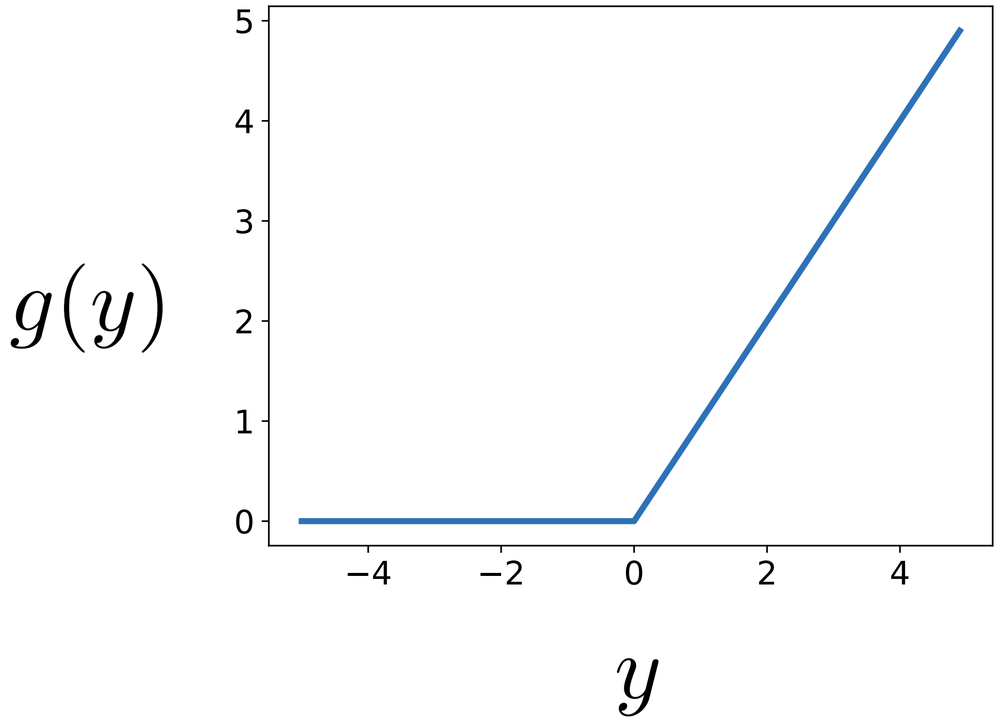
Source: Isola, Torralba, Freeman

Computation in a neural net — nonlinearity

- Unbounded output (on positive side)
- Efficient to implement: $\frac{\partial g}{\partial y} = \begin{cases} 0, & \text{if} \quad y < 0 \\ 1, & \text{if} \quad y \geq 0 \end{cases}$
- Also seems to help convergence (6x speedup vs. tanh in [Krizhevsky et al. 2012])
- Drawback: if strongly in negative region, unit is dead forever (no gradient).
- Default choice: widely used in current models!

Rectified linear unit (ReLU)

$$g(y) = \max(0, y)$$



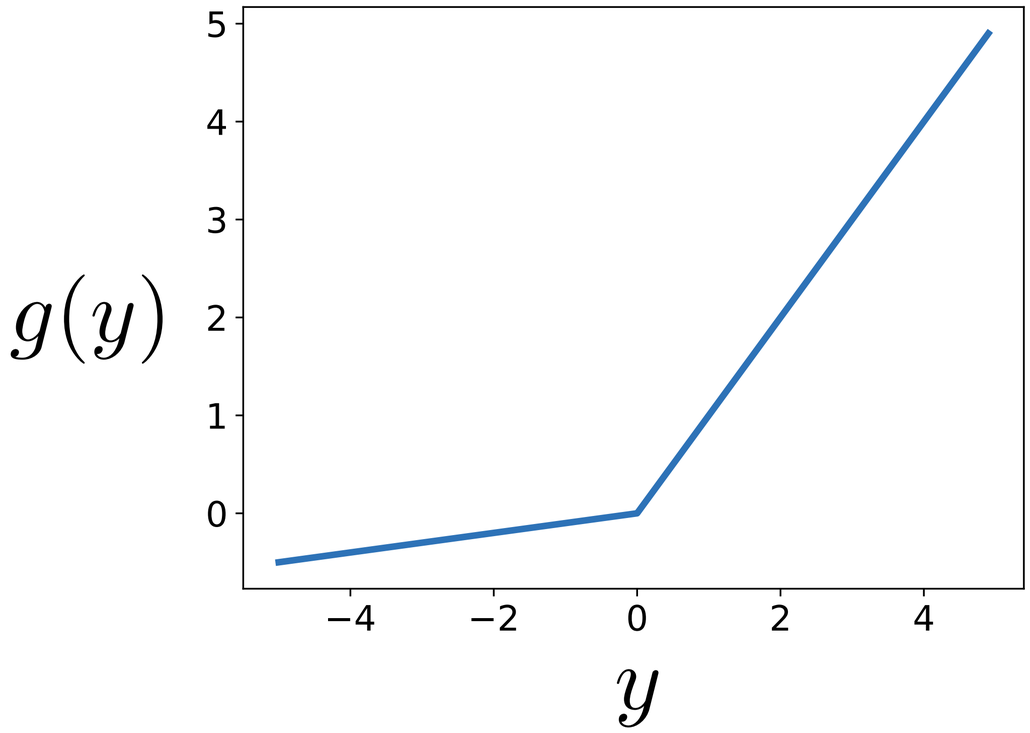
Source: Isola, Torralba, Freeman

Computation in a neural net — nonlinearity

- where α is small (e.g., 0.02)
- Efficient to implement: $\frac{\partial g}{\partial y} = \begin{cases} -a, & \text{if} \quad y < 0 \\ 1, & \text{if} \quad y \geq 0 \end{cases}$
- Has non-zero gradients everywhere (unlike ReLU)

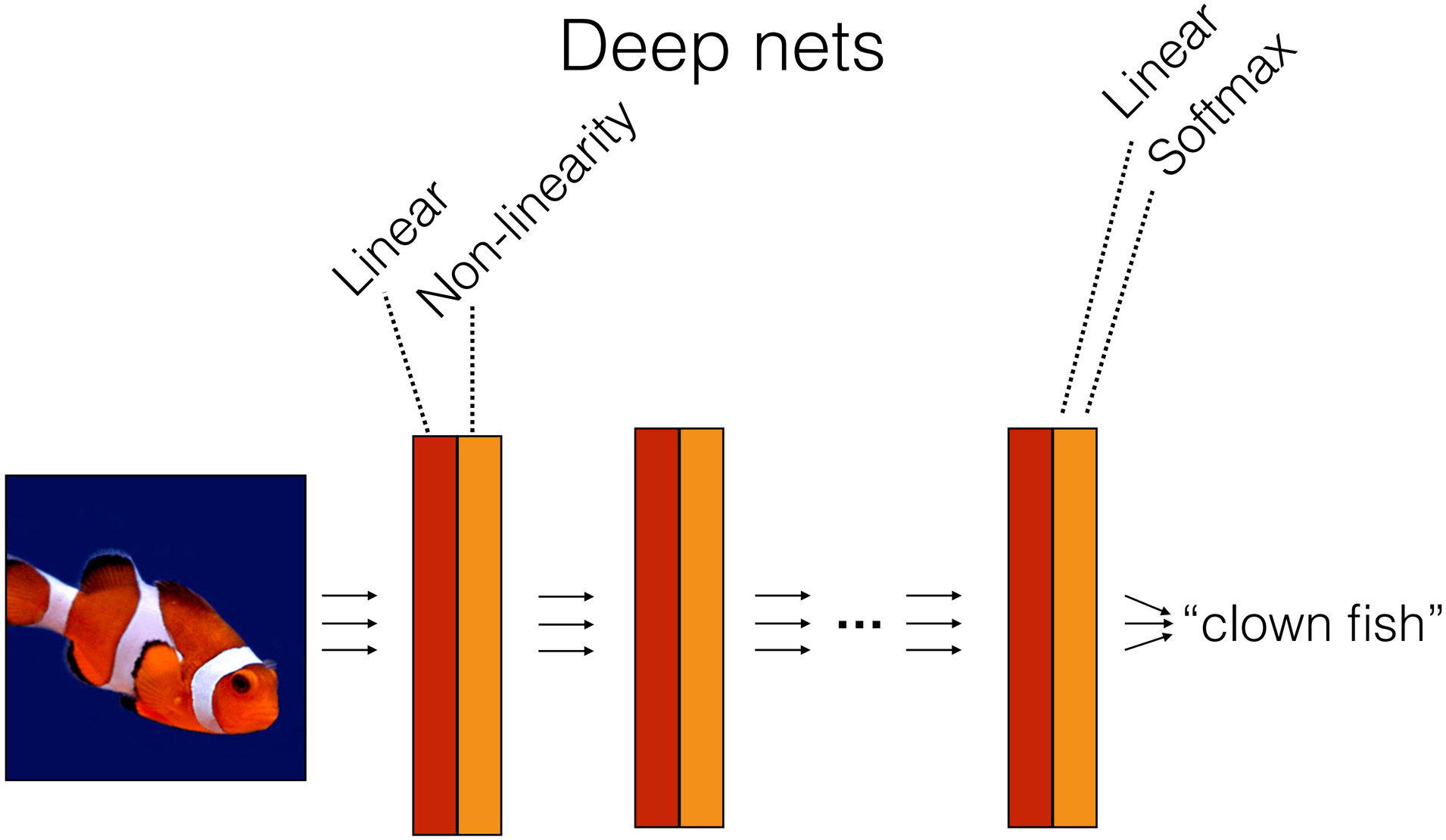
Leaky ReLU

$$g(y) = \begin{cases} \max(0, y), & \text{if} \quad y \geq 0 \\ a \min(0, y), & \text{if} \quad y < 0 \end{cases}$$



Source: Isola, Torralba, Freeman

Deep nets



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

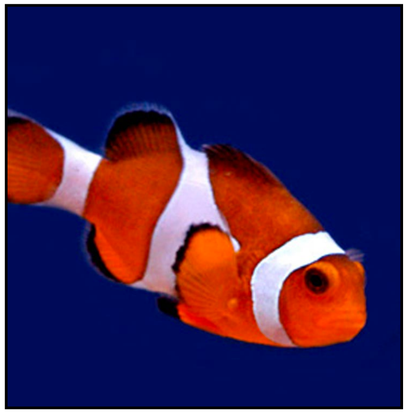
Source: Isola, Torralba, Freeman

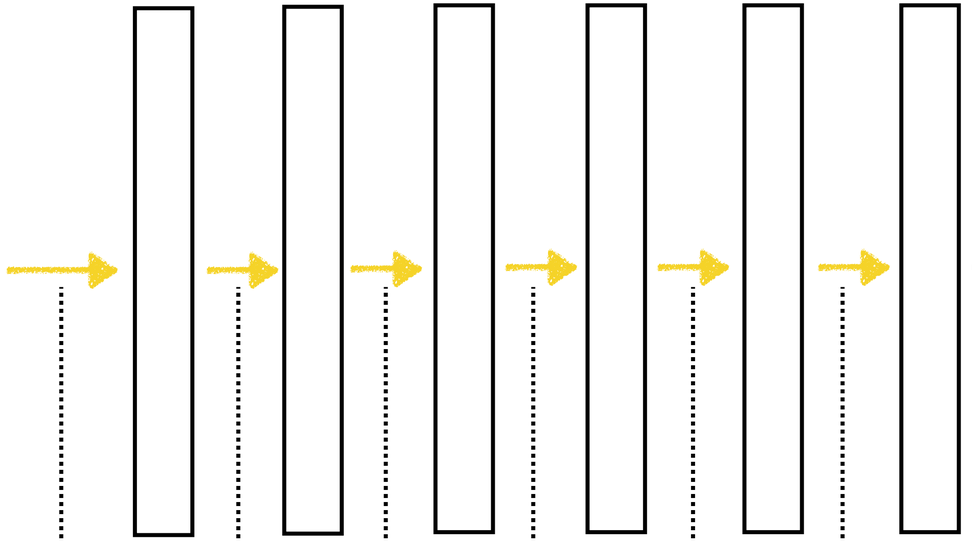
How do we learn the parameters?

$\mathbf{y}_1$
"clown fish"

Learned

$\mathbf{x}_1$





$\mathcal{L}(f_\theta(\mathbf{x}_1), \mathbf{y}_1)$

$$\theta^* = \arg\min_\theta \sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)$$

Source: Isola, Torralba, Freeman

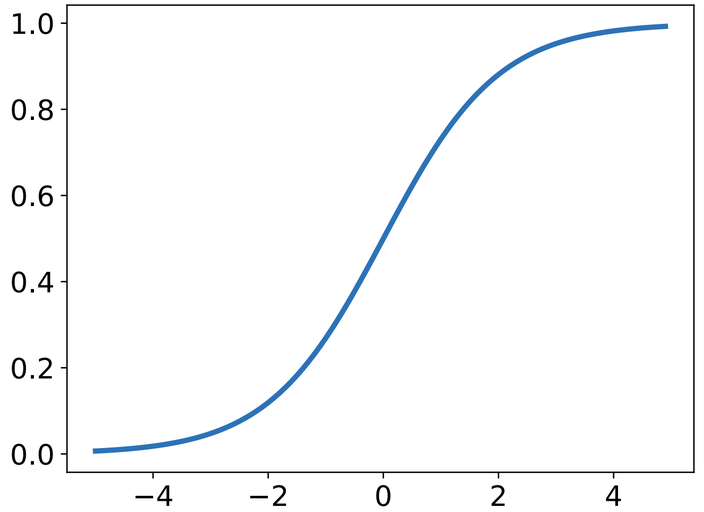
Learning parameters

Squared loss with single-variable network:

$$L = \frac{1}{2}(y - f(x))^2$$

$$L = \frac{1}{2}(y - \sigma(wx + b))^2$$

$$\sigma(y) = \frac{1}{1 + e^{-y}}$$



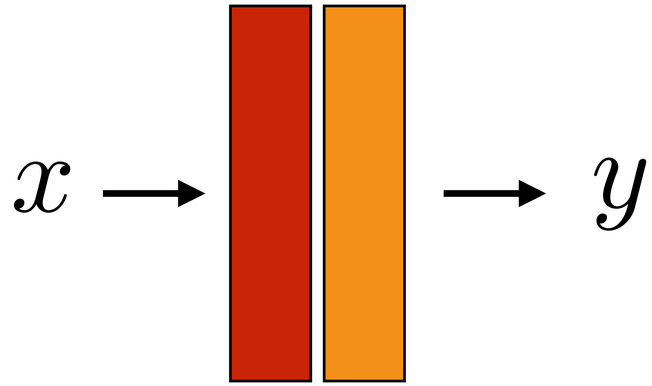
Example source: Roger Grosse

Learning parameters

Squared loss with single-variable network:

$$L = \frac{1}{2}(y - f(x))^2$$

$$L = \frac{1}{2}(y - \sigma(wx + b))^2$$



Want: derivatives $\frac{\partial L}{\partial w}, \frac{\partial L}{\partial b}$

Example source: Roger Grosse

Learning parameters

Writing out the layers explicitly:

$$z = wx + b$$

$$t = \sigma(z)$$

$$L = \frac{1}{2}(y - t)^2$$

How do we do this?
Use the chain rule!

$$\frac{\partial}{\partial x} f(g(x)) = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$$

Example source: Roger Grosse

Computing derivatives with the chain rule

$$\frac{\partial L}{\partial w} = \frac{\partial}{\partial w}\left[\frac{1}{2}(y - \sigma(wx + b))^2\right]$$

$$= (y - \sigma(wx + b))\frac{\partial}{\partial w}(y - \sigma(wx + b))$$

$$= (y - \sigma(wx + b))\sigma'(wx + b)\frac{\partial}{\partial w}(wx + b)$$

$$= (y - \sigma(wx + b))\sigma'(wx + b)x$$

$$\frac{\partial L}{\partial b} = \frac{\partial}{\partial b}\left[\frac{1}{2}(y - \sigma(wx + b))^2\right]$$

$$= (y - \sigma(wx + b))\frac{\partial}{\partial b}(y - \sigma(wx + b))$$

$$= (y - \sigma(wx + b))\sigma'(wx + b)\frac{\partial}{\partial b}(wx + b)$$

$$= (y - \sigma(wx + b))\sigma'(wx + b)$$

Example source: Roger Grosse

Learning parameters

Writing out the layers:

$z = wx + b$ linear (affine) layer

$t = \sigma(z)$ nonlinearity

$L = \frac{1}{2}(y - t)^2$ loss

Example source: Roger Grosse

Learning parameters

Writing out the layers:

$$z = wx + b$$

$$t = \sigma(z)$$

$$L = \frac{1}{2}(y - t)^2$$

Another way to write derivatives:

$$\frac{\partial L}{\partial t} = t - y$$

Each step is easy to compute, and we can reuse computation.

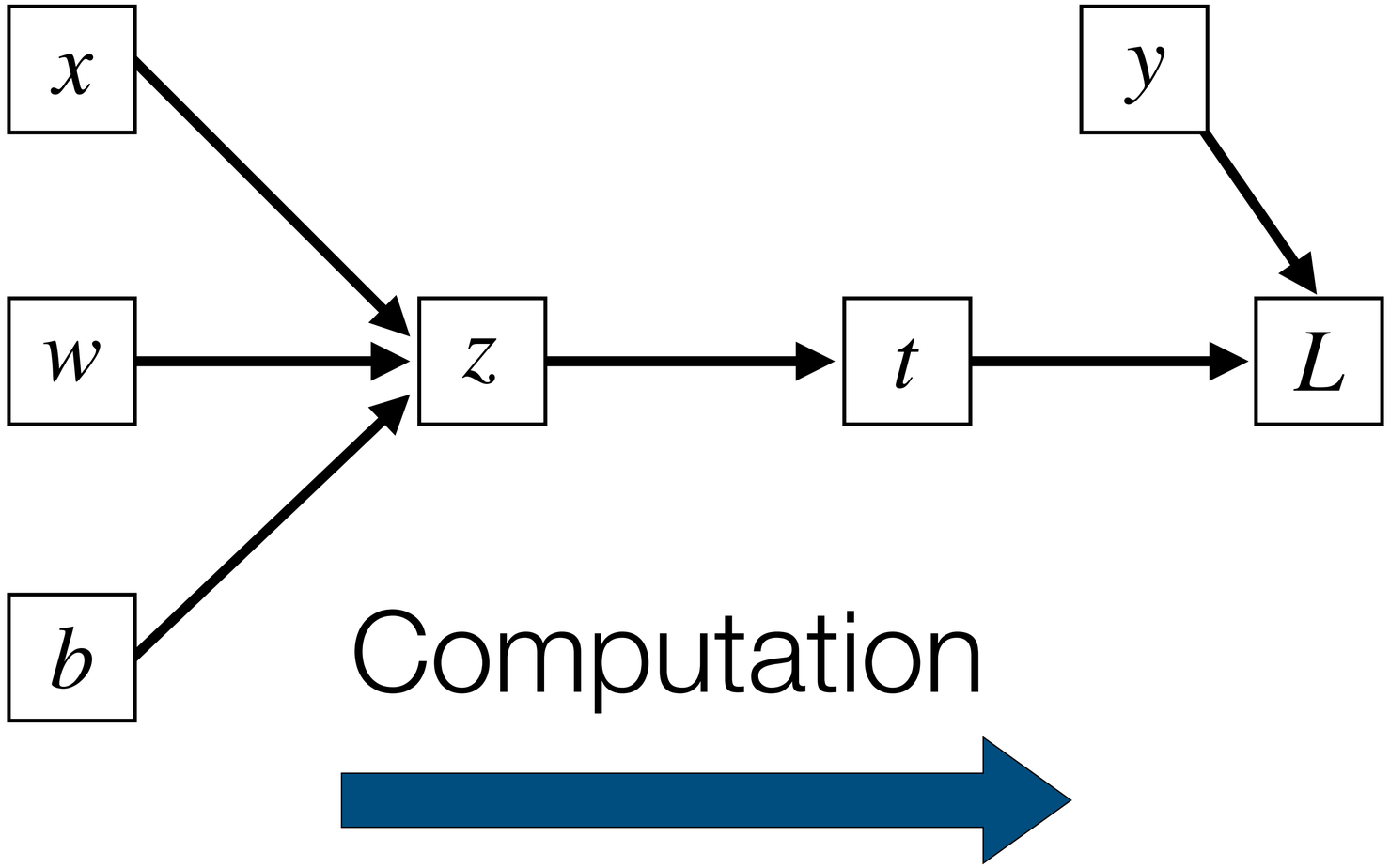
$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial t}\frac{\partial t}{\partial z} = \frac{\partial L}{\partial t}\sigma'(z)$$

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial z}x$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial z}$$

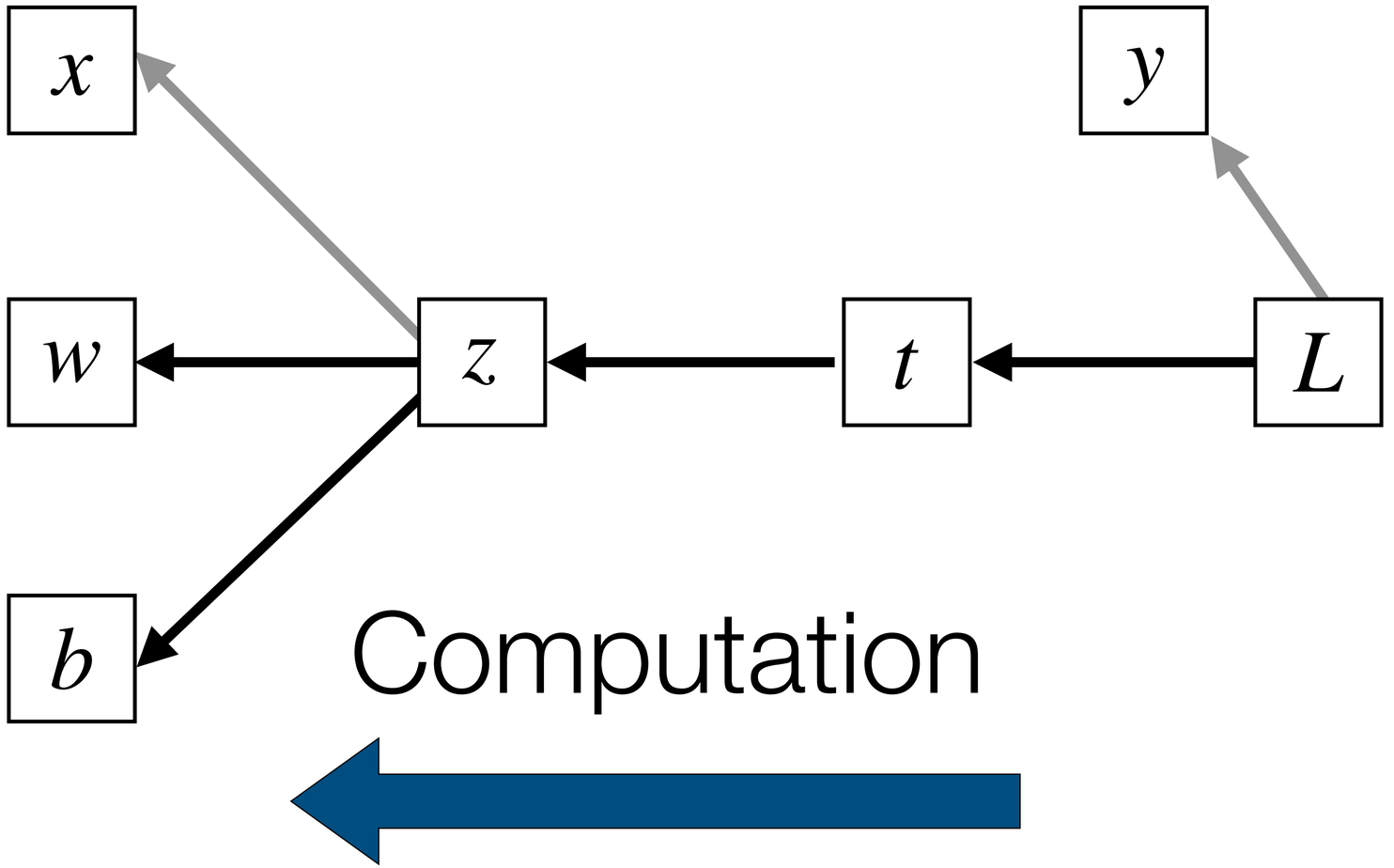
Example source: Roger Grosse

Computation graph: loss



Adapted from: Roger Grosse

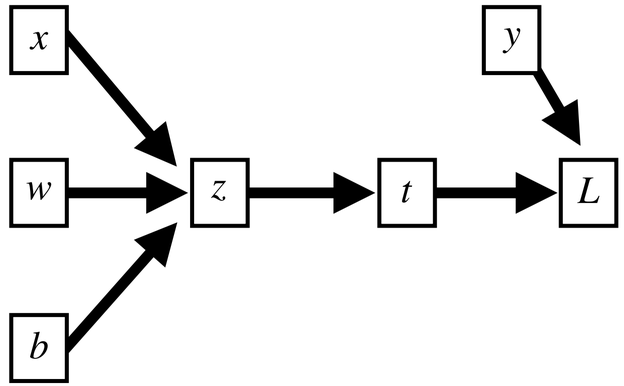
Computation graph: derivatives



Adapted from: Roger Grosse

Backpropagation

Step #1: Compute loss and every value in computation graph.



Forward pass:

for i = 0 … N:

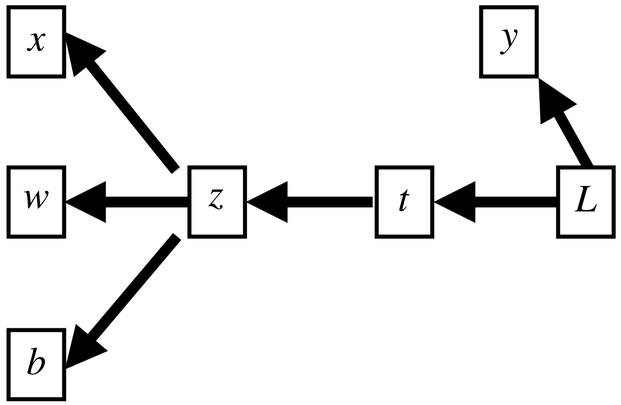
compute each node value v_i using values from previous nodes

where N nodes $v_0, v_1, \ldots v_n$ are ordered topographically (i.e. inputs always come before outputs).

Adapted from: Roger Grosse

Backpropagation

Step #2: Compute derivatives.



Backward pass:

$v_N = 1$

for i = N-1 .. 0:

$$\frac{\partial L}{\partial v_i} = \sum_{j \in \text{Outputs}(v_i)} \frac{\partial L}{\partial v_j} \frac{\partial v_j}{\partial v_i}$$

Starting from the loss, work your way to the inputs. Compute the derivative of the loss w.r.t. each value.

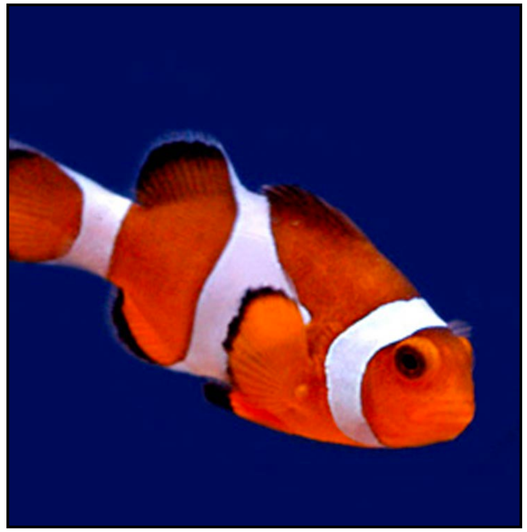
Summary: compute derivatives efficiently using the chain rule.

Adapted from: Roger Grosse

Automatic differentiation

- **Backpropagation**: algorithm for computing derivatives
- An instance of **reverse-mode automatic differentiation**
- Usually implement backprop using autodifferentiation (“autodiff”) software package.
  - Build your program out of primitive operations, similar to `numpy` operations
  - Behind the scenes, the autodiff package builds the computation graph for you!
  - It’s *not* finite difference. Computes exact gradients using backprop!

Processing images with neural nets



Classifier

"Fish"

image **x**

label y

Source: Isola, Torralba, Freeman

Processing images with neural nets

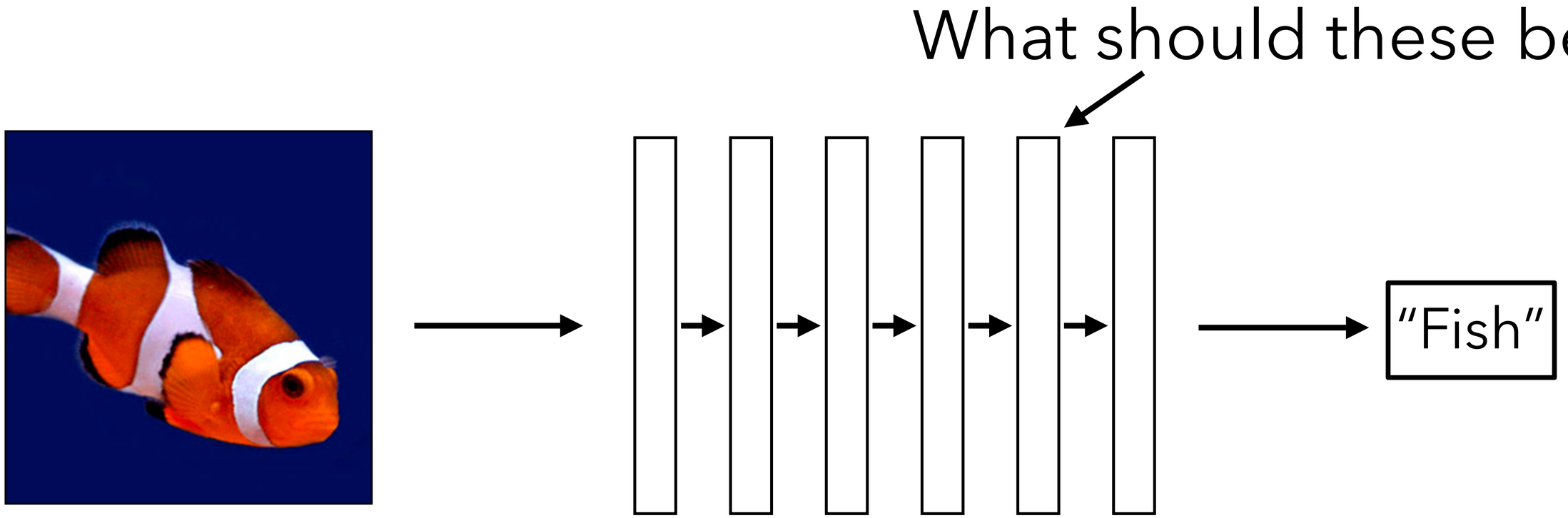


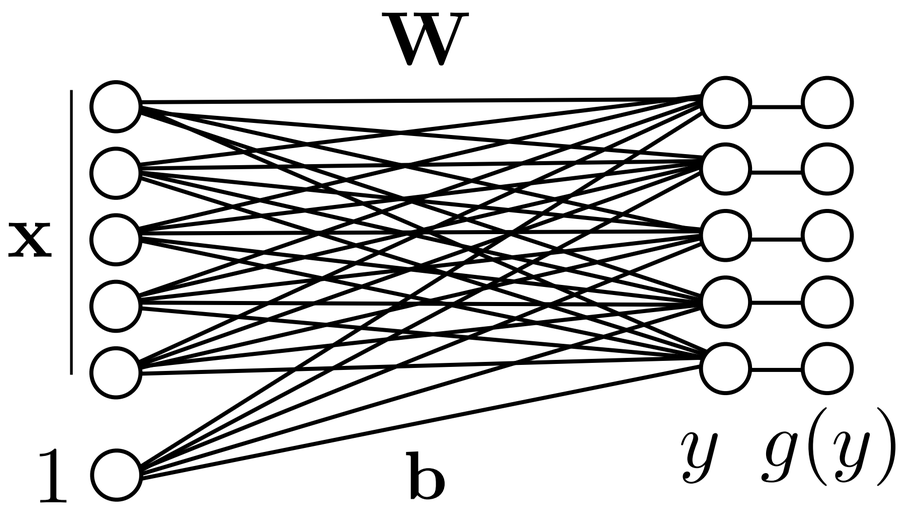
image **x**

label y

Source: Isola, Torralba, Freeman

Idea #1: Fully-connected network

Fully-connected (a.k.a. linear) layer

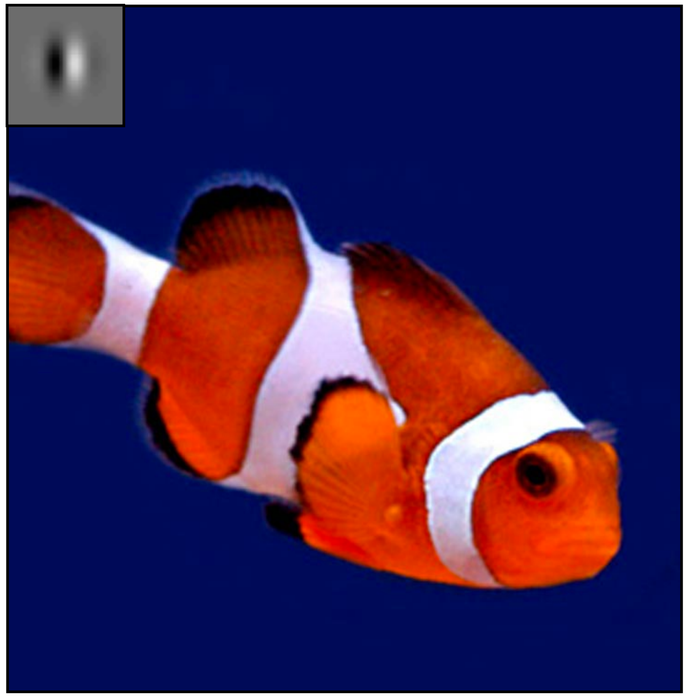


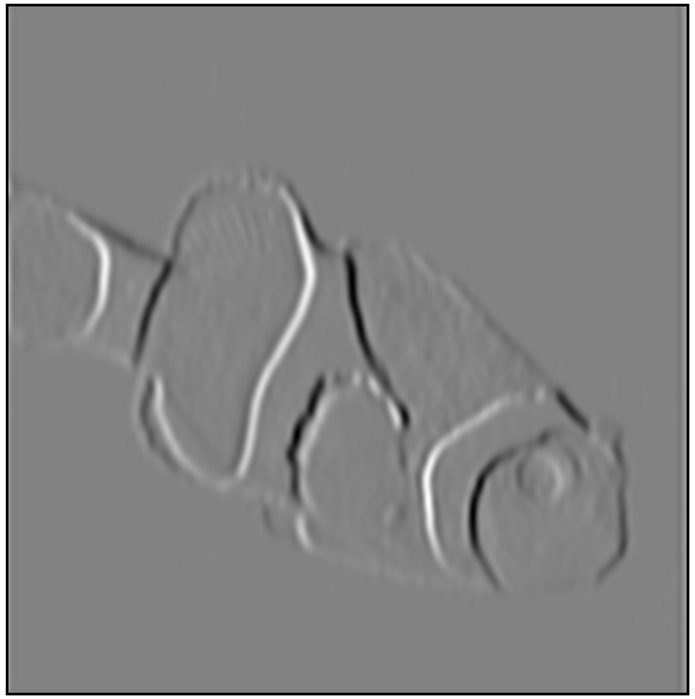
$\mathbf{x} =$ 

But $\mathbf{x}$ is really big!

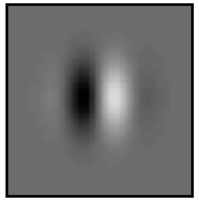
Say, 256 x 256 x 3 = 197k

Can we use convolution in a neural network?



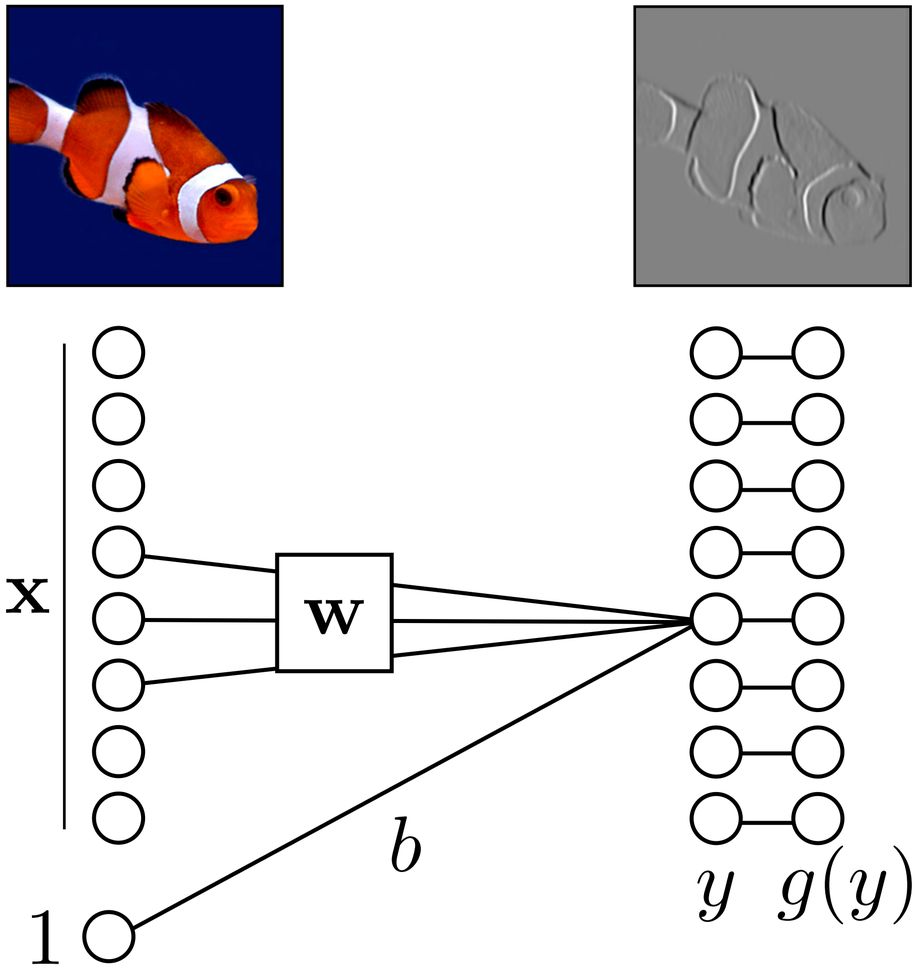


filter



Source: Isola, Torralba, Freeman

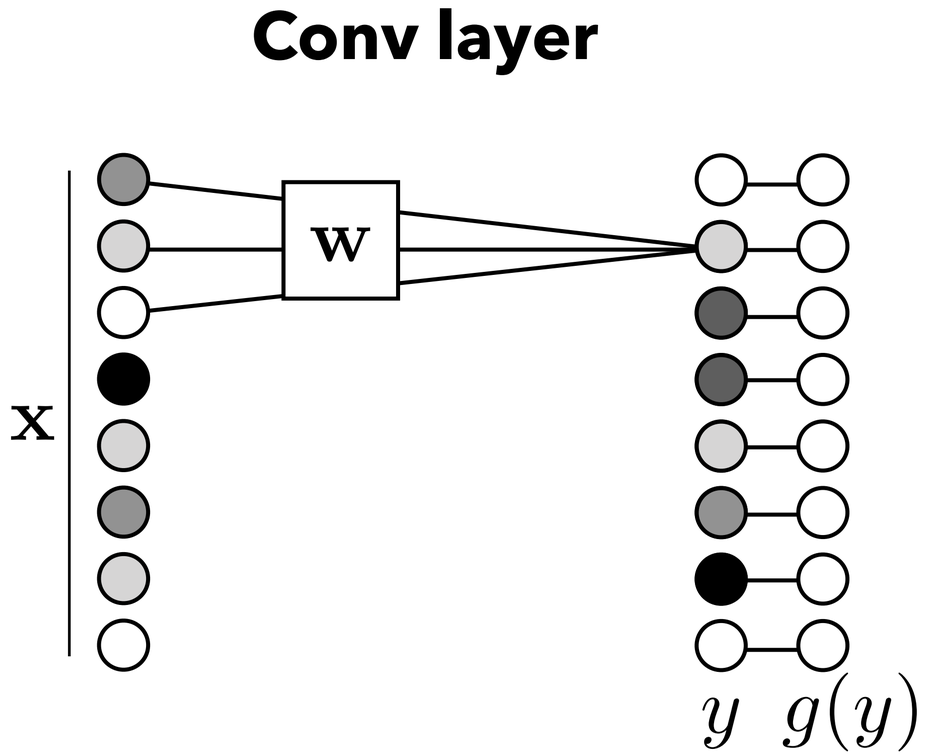
Sparsely connected network



Each unit is connected to a subset of the units in the previous layer.

Source: Isola, Torralba, Freeman

Convolutional neural network

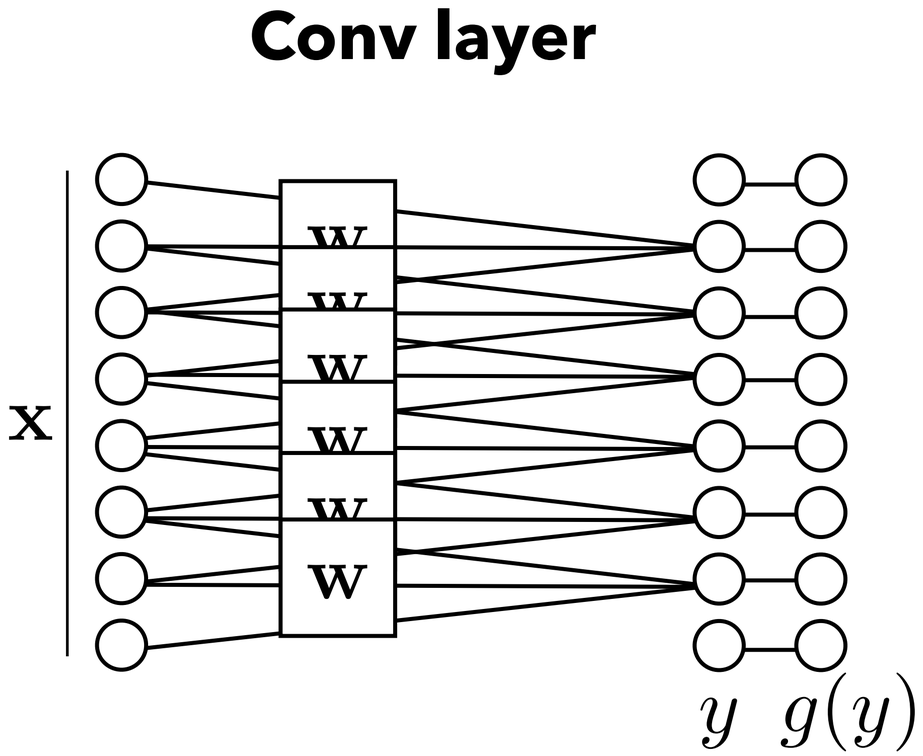


Each output unit is computed from an image patch.

a.k.a. CNN or ConvNet

Source: Isola, Torralba, Freeman

Weight sharing

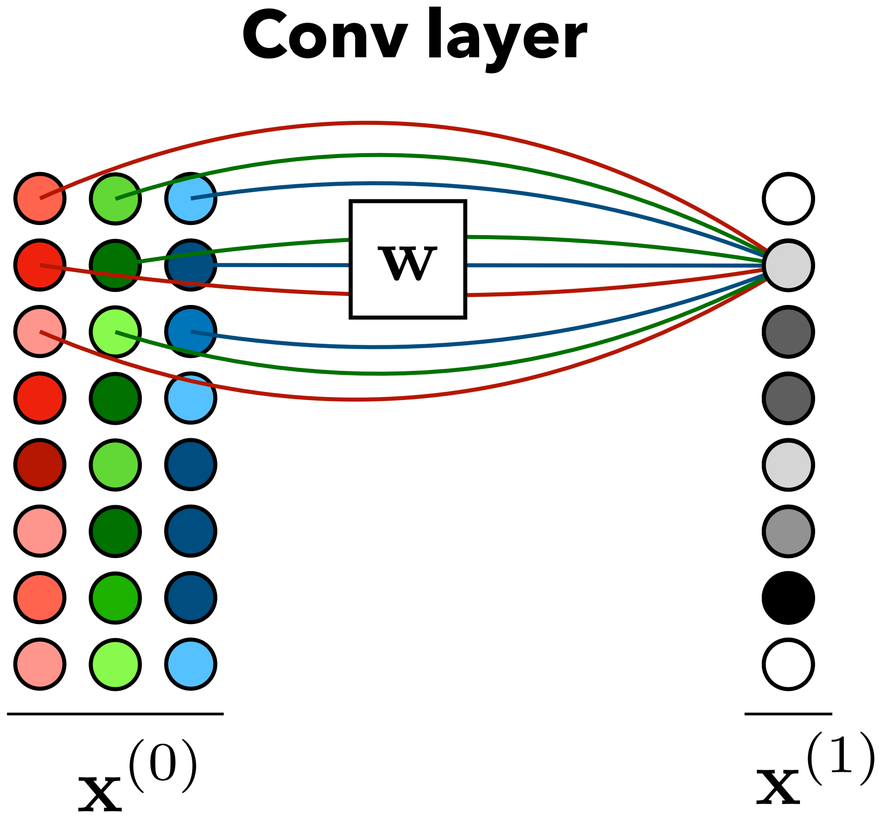


We “share” weights for each patch.

If a feature is useful in one position, it should be useful in others, too.

Source: Isola, Torralba, Freeman

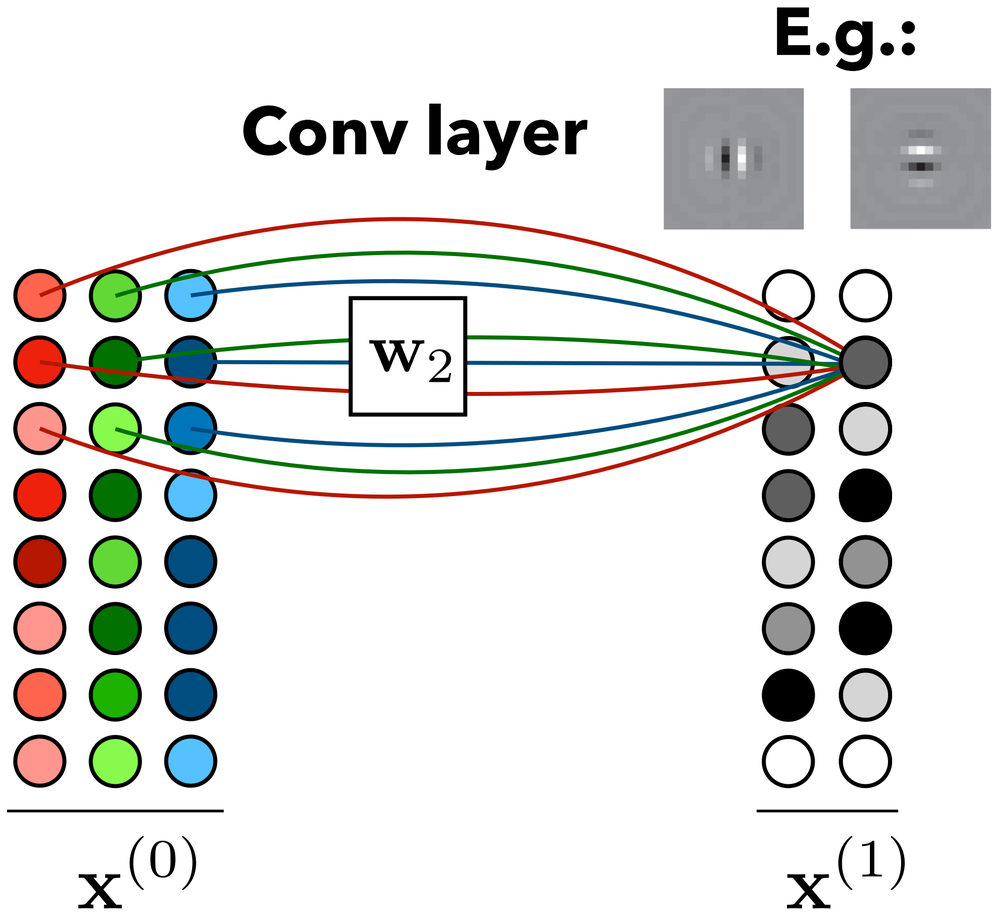
Multiple input channels



$$\mathbb{R}^{N \times C} \rightarrow \mathbb{R}^{N \times 1}$$

Source: Isola, Torralba, Freeman

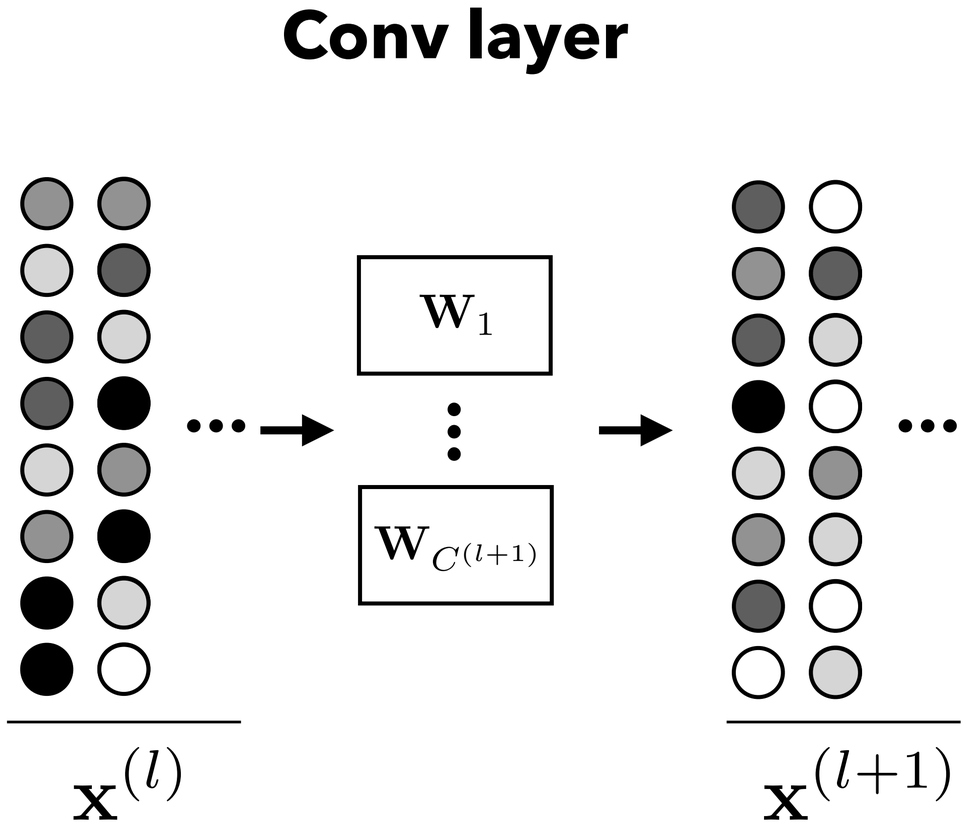
Multiple output channels



$$\mathbb{R}^{N \times C^{(0)}} \rightarrow \mathbb{R}^{N \times C^{(1)}}$$

Source: Isola, Torralba, Freeman

Multiple input and output channels



$$\mathbb{R}^{N \times C^{(l)}} \rightarrow \mathbb{R}^{N \times C^{(l+1)}}$$

Source: Isola, Torralba, Freeman

Image classification

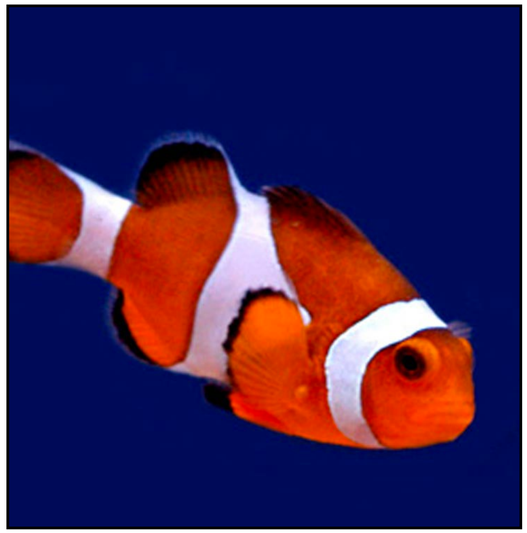
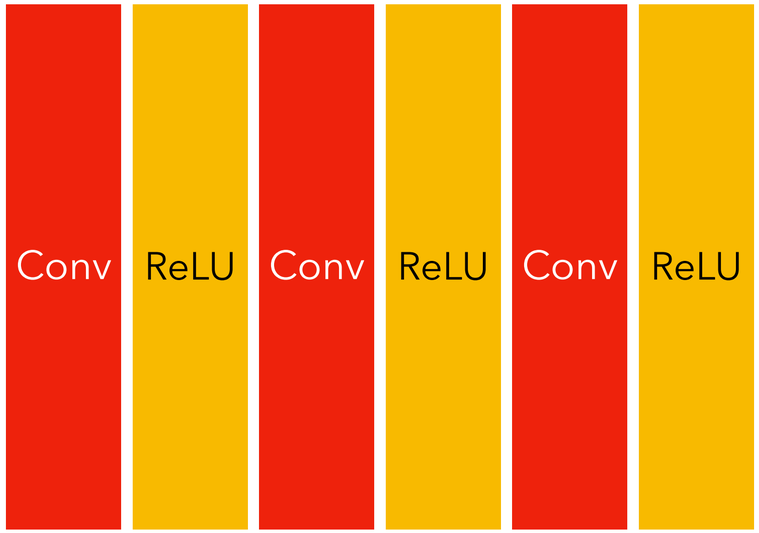


image **x**



"Fish"

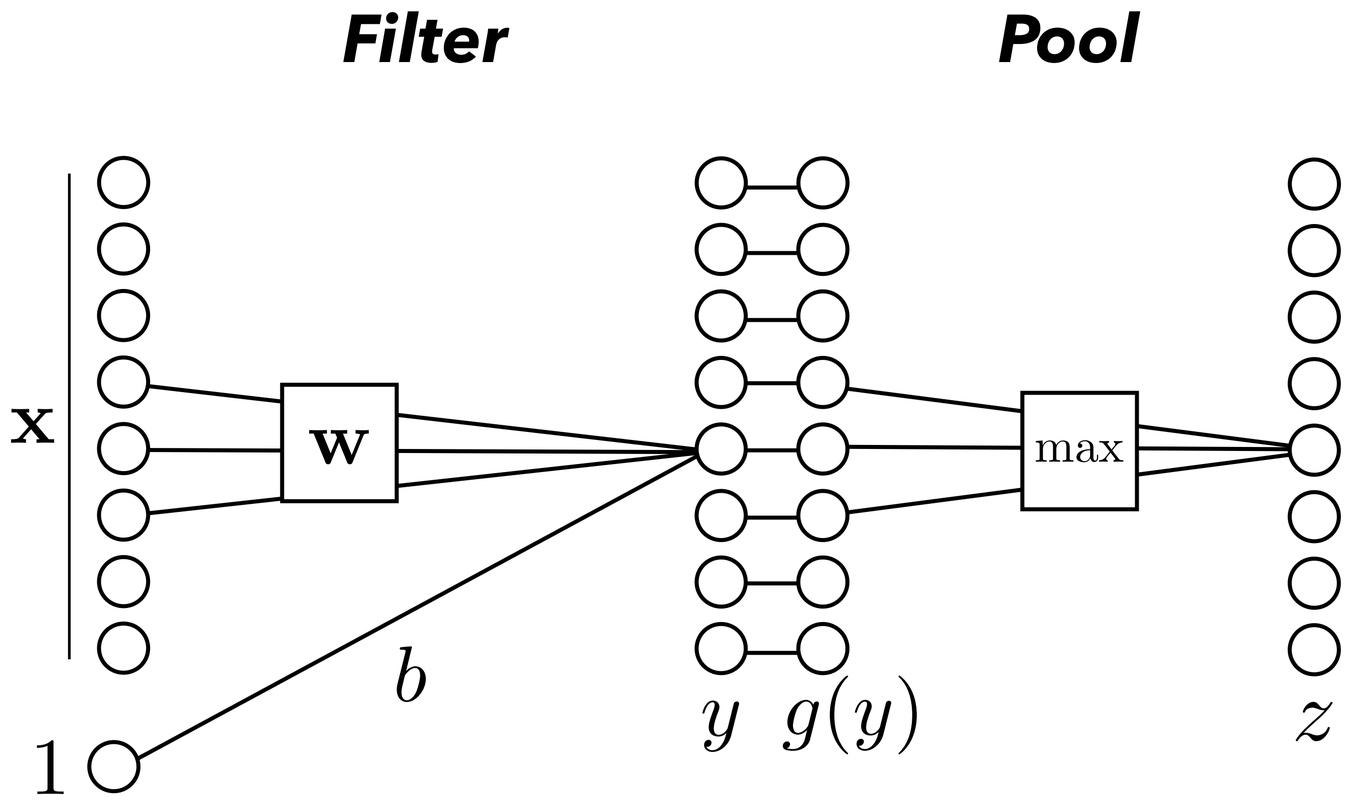
label y

Problems with this idea:

1. No "global" processing.
2. How do you get the final label?

Source: Isola, Torralba, Freeman

Pooling

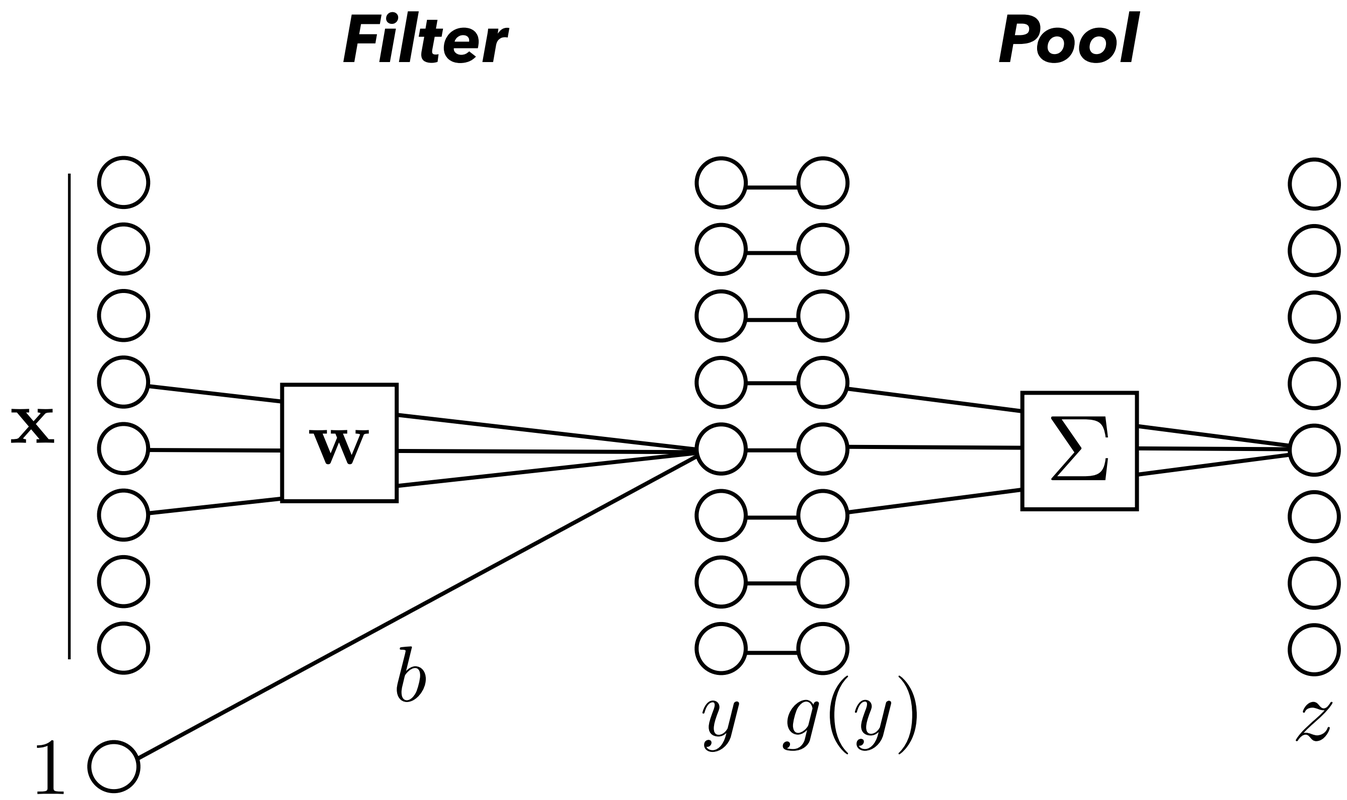


Max pooling

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

Source: Isola, Torralba, Freeman

Pooling



Max pooling

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

Mean pooling

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

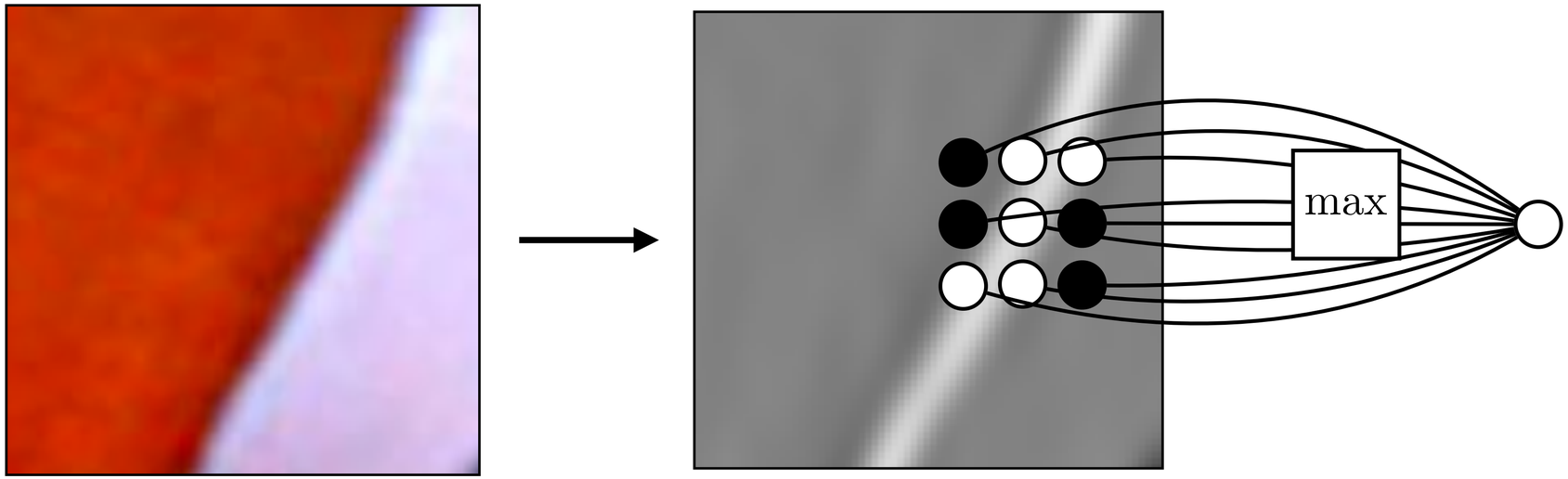
Blurring [Zhang 2019]

$$z = \text{conv}(y, \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix})$$

Adapted from: Isola, Torralba, Freeman

Pooling – Why?

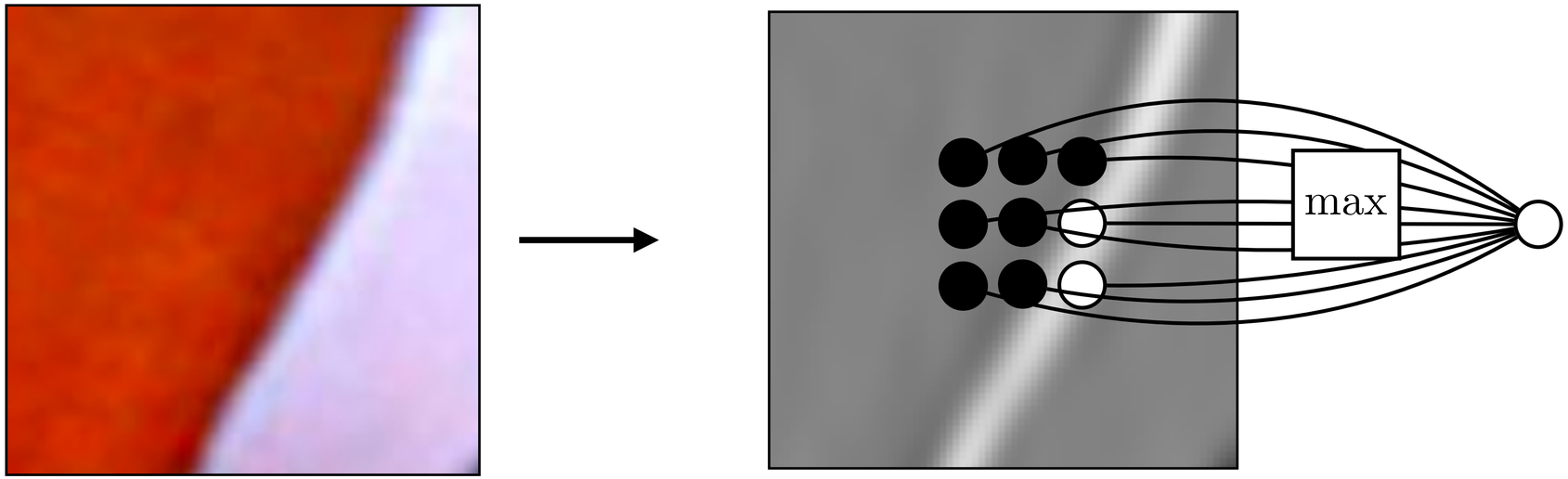
Pooling across spatial locations achieves stability w.r.t. small translations:



Source: Isola, Torralba, Freeman

Pooling – Why?

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

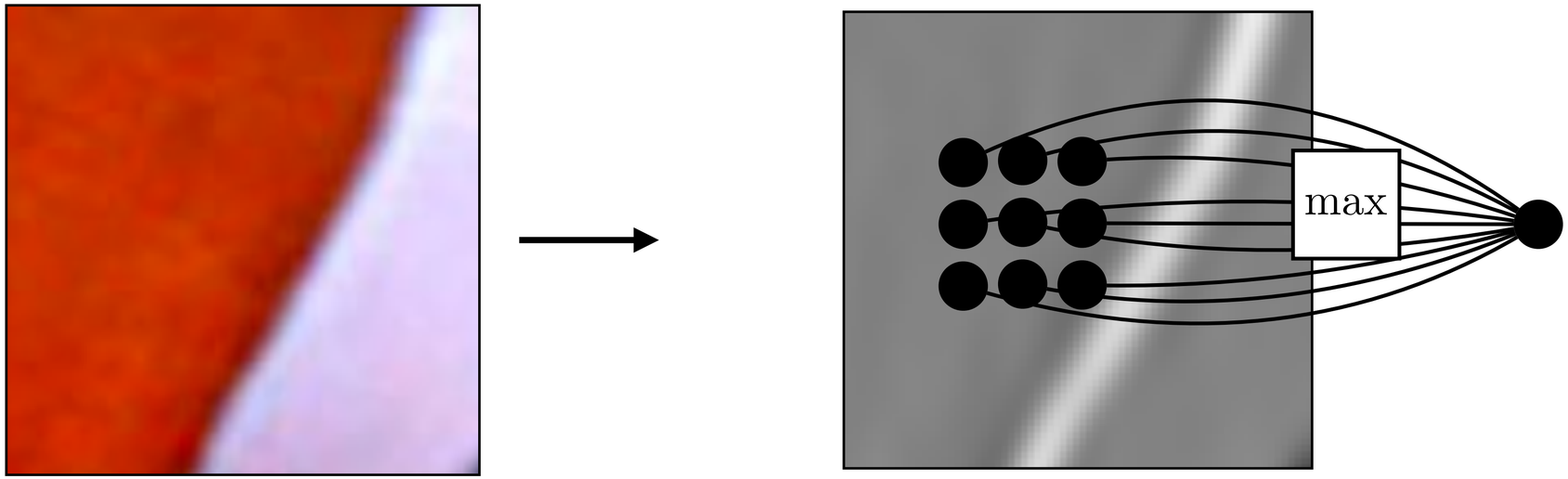


same large response regardless of exact position of edge

Source: Isola, Torralba, Freeman

Pooling – Why?

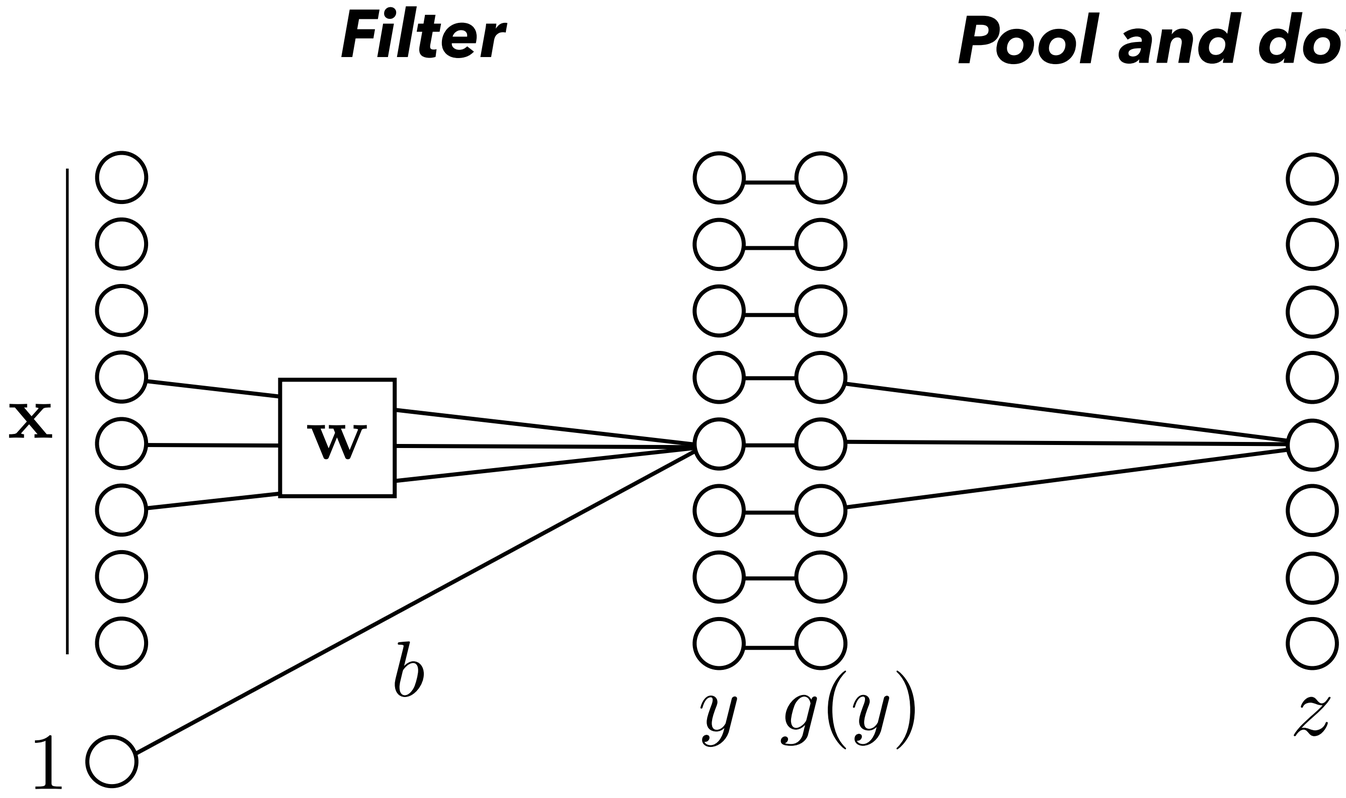
Pooling across spatial locations achieves stability w.r.t. small translations:



however, if the image is translated a lot…

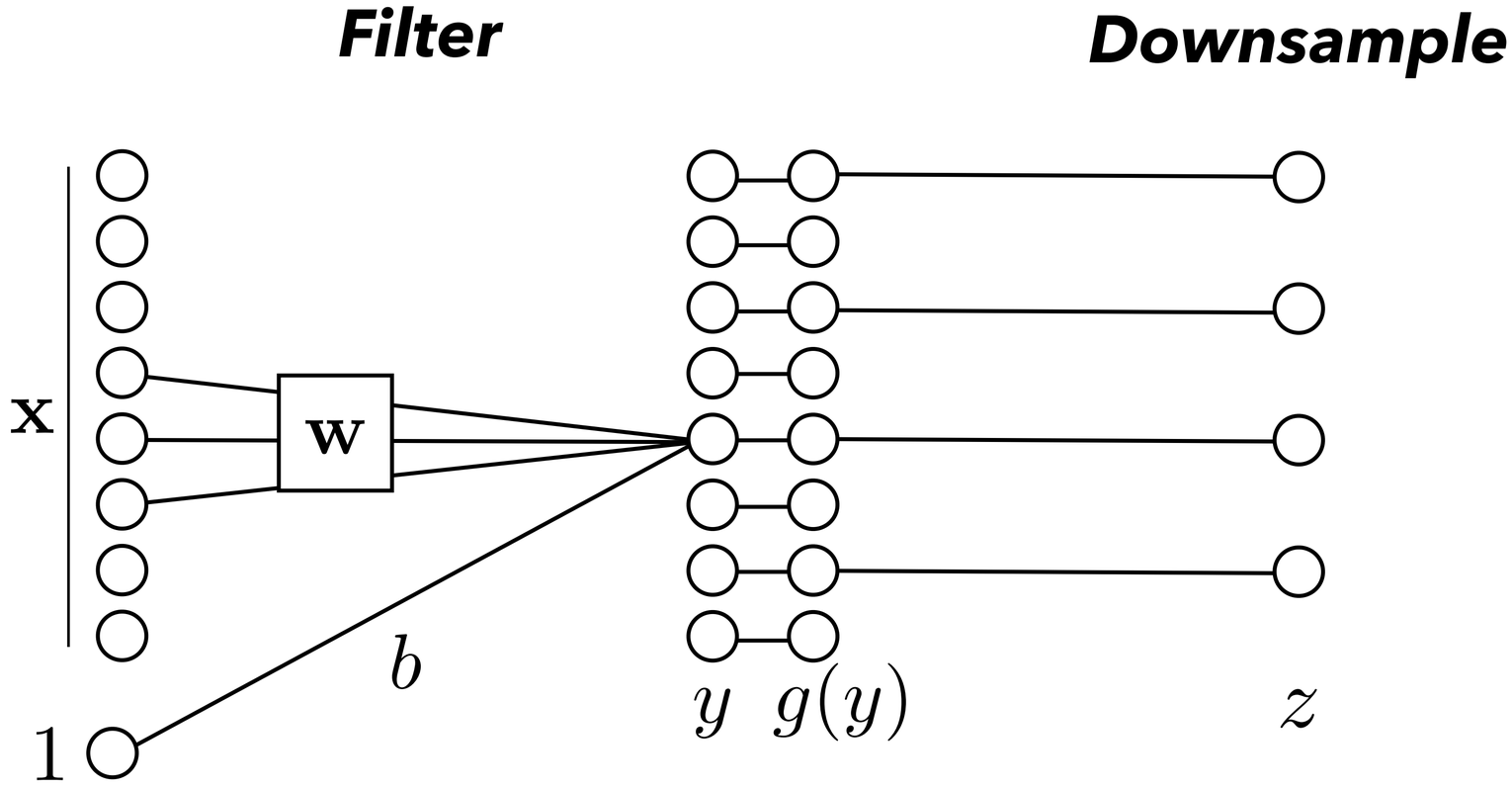
Source: Isola, Torralba, Freeman

Downsampling



Source: Isola, Torralba, Freeman

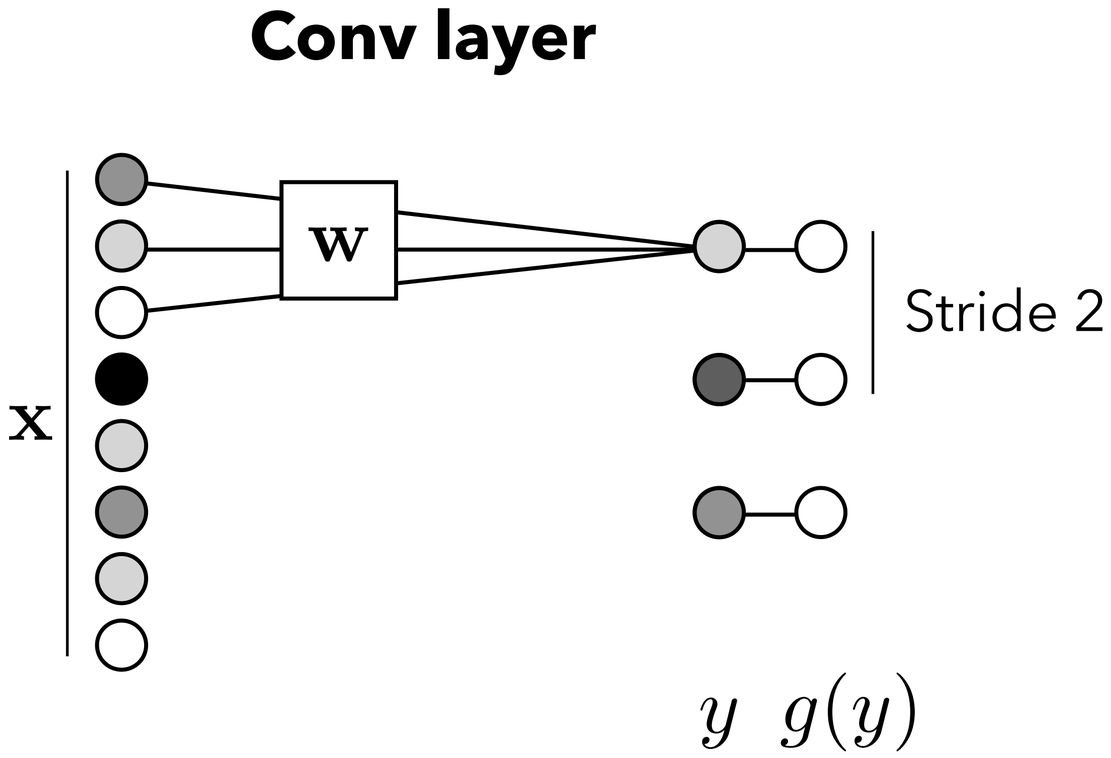
Downsampling



$$\mathbb{R}^{H^{(l)} \times W^{(l)} \times C^{(l)}} \rightarrow \mathbb{R}^{H^{(l+1)} \times W^{(l+1)} \times C^{(l+1)}}$$

Source: Isola, Torralba, Freeman

Strided operations

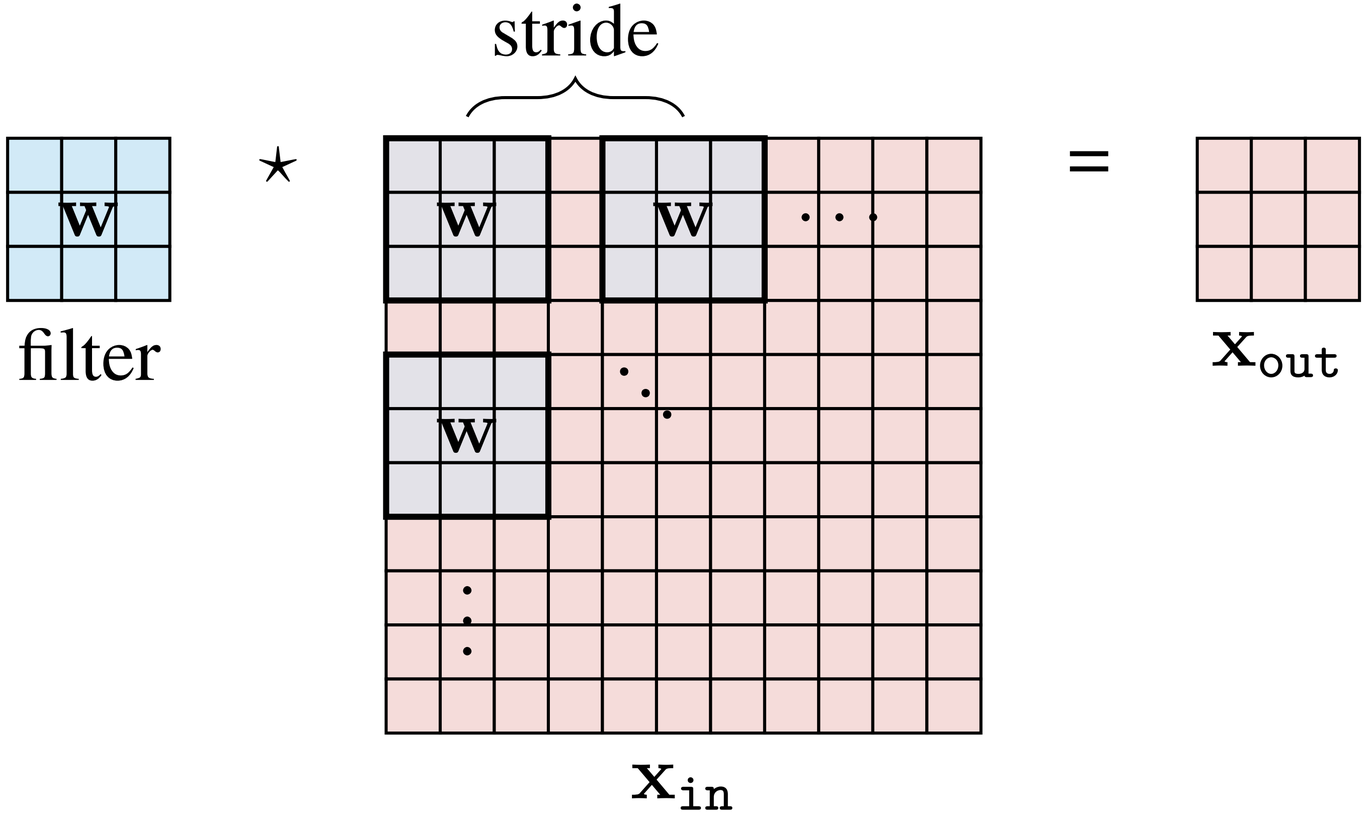


Strided operations combine a given operation (convolution or pooling) and downsampling into a single operation.

Strided convolution is an alternative to pooling layers: just do a strided convolution!

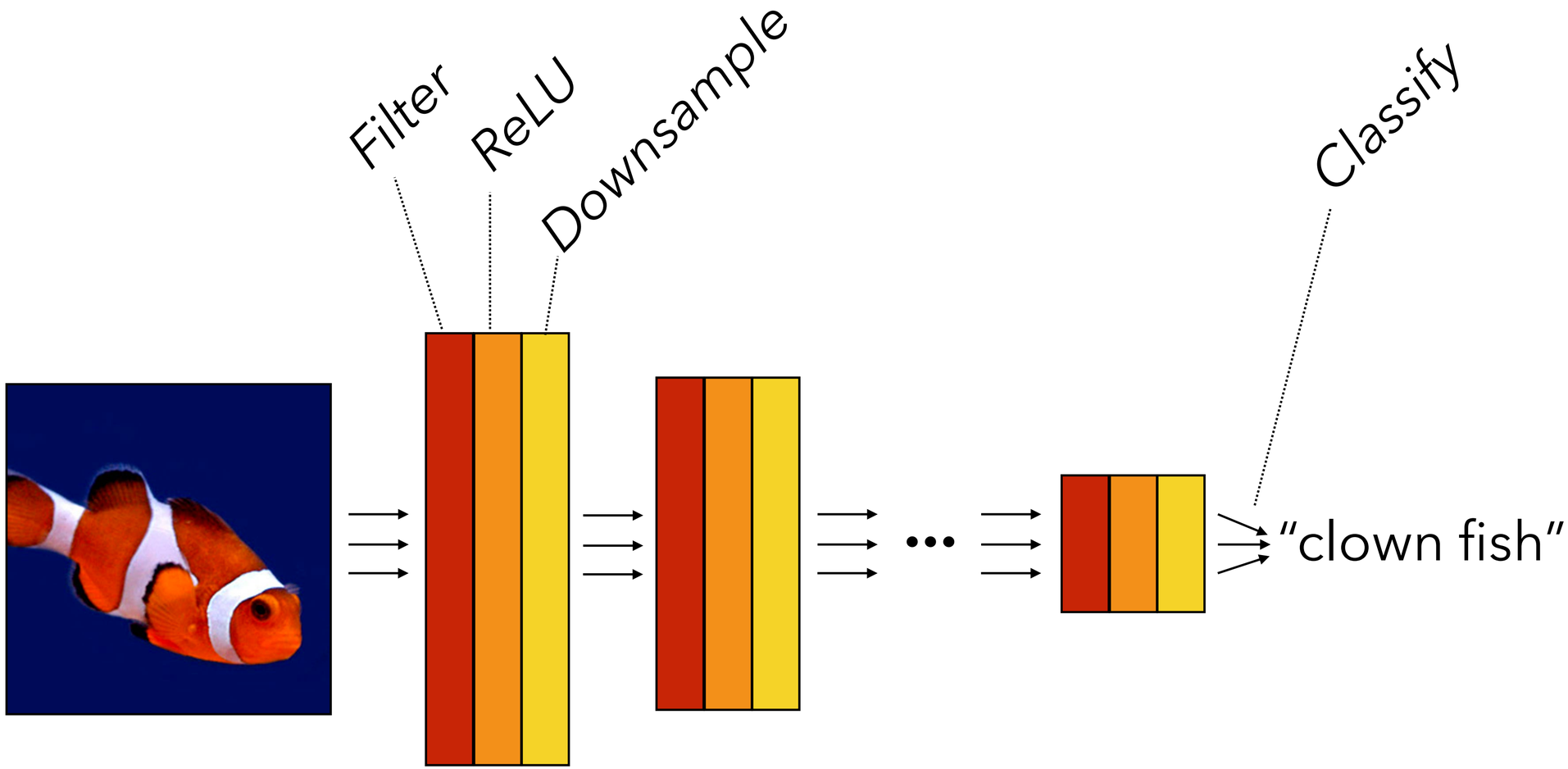
Source: Isola, Torralba, Freeman

Strided operations (2D)



Source: Foundations of Computer Vision. Torralba, Isola, Freeman

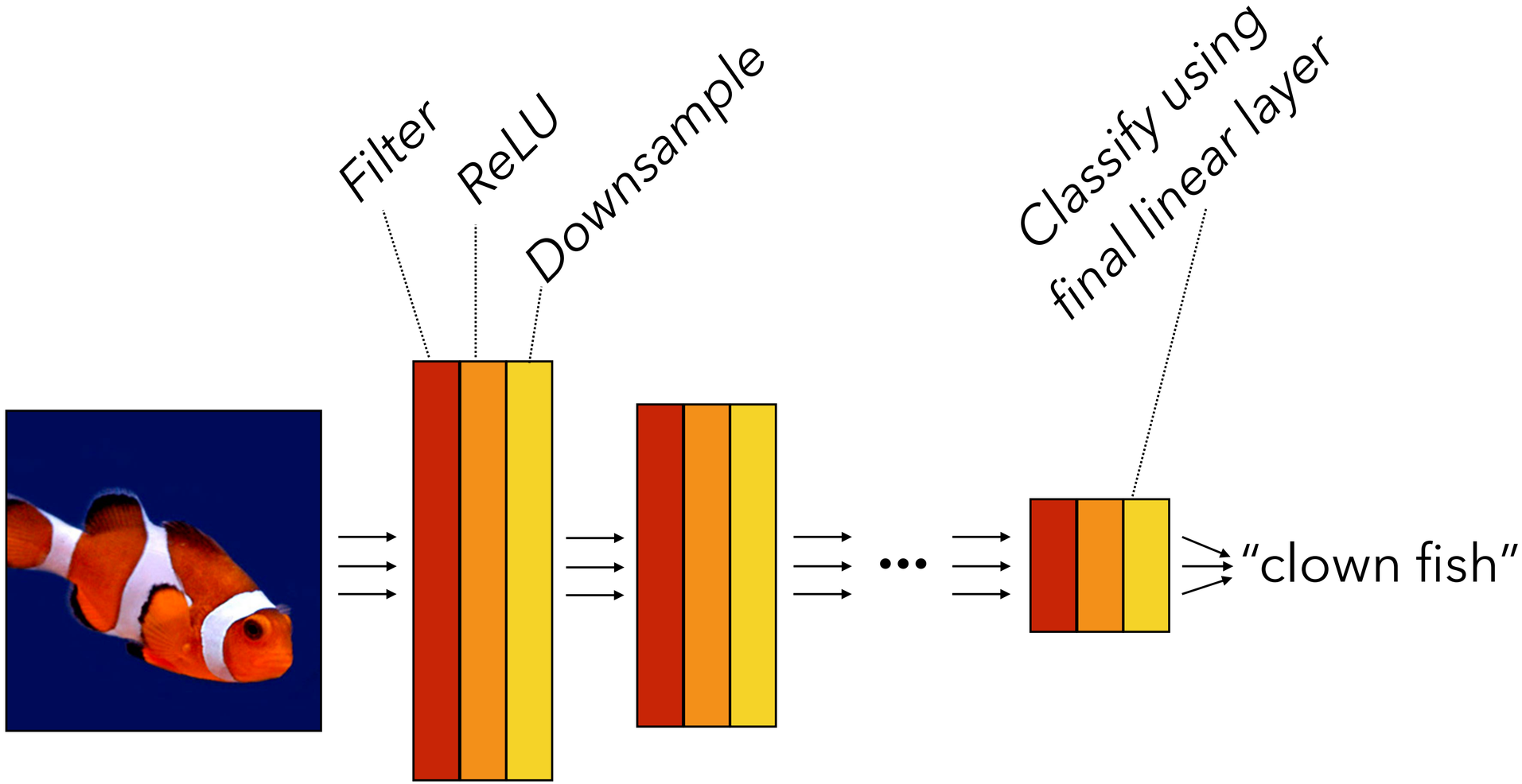
Computation in a neural net



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

Source: Isola, Torralba, Freeman

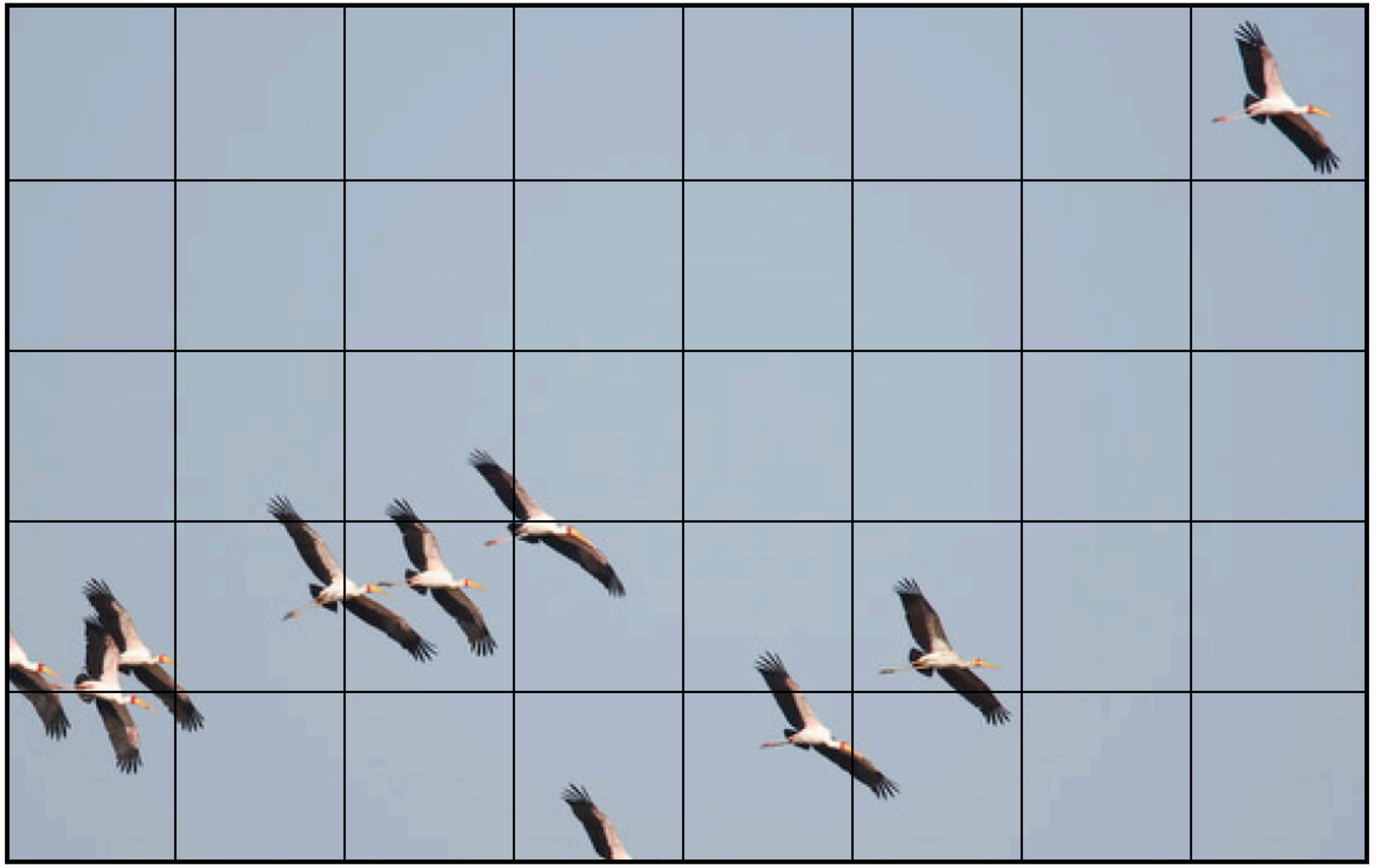
Computation in a neural net



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

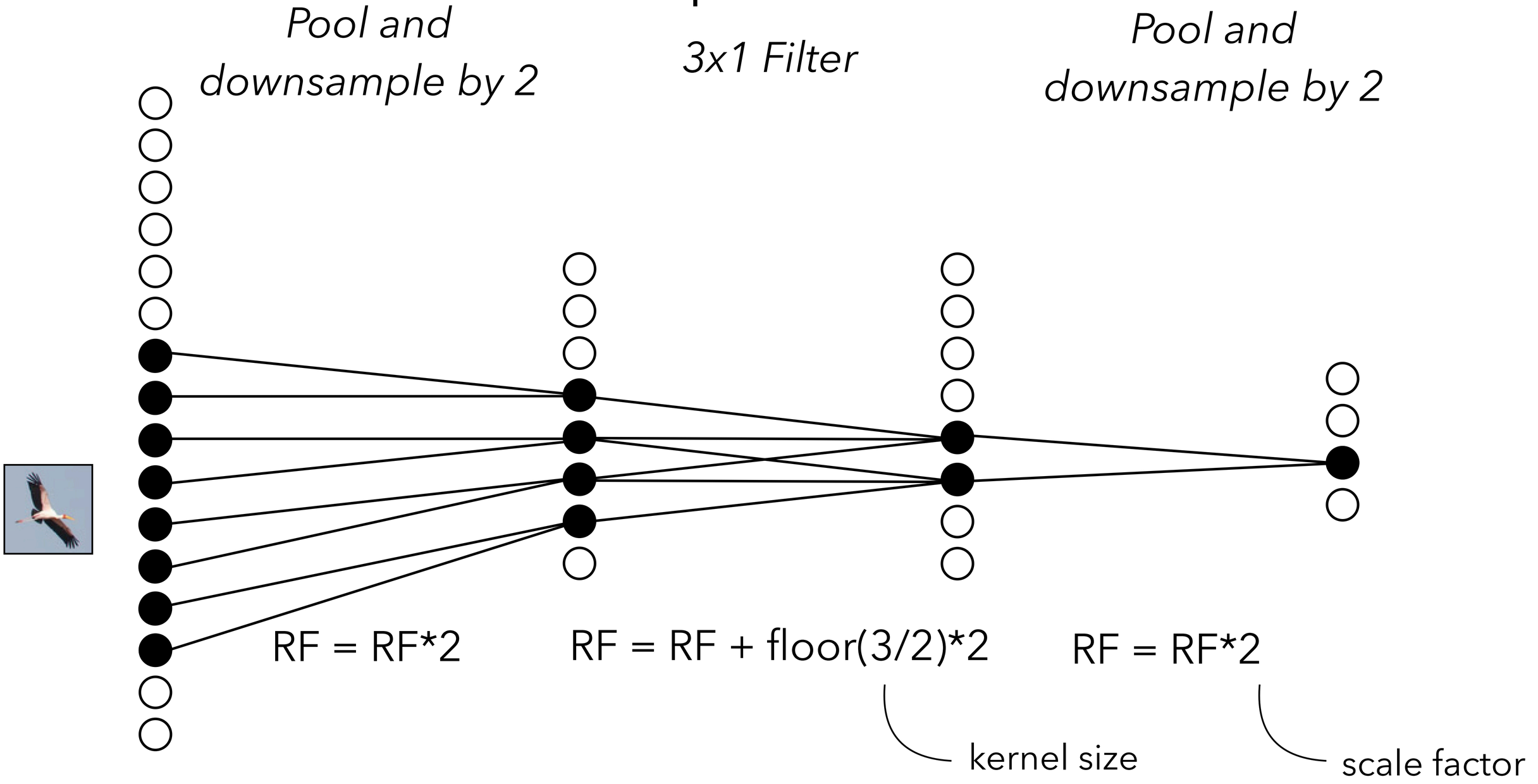
Source: Isola, Torralba, Freeman

Receptive fields



Source: Isola, Torralba, Freeman

Receptive fields

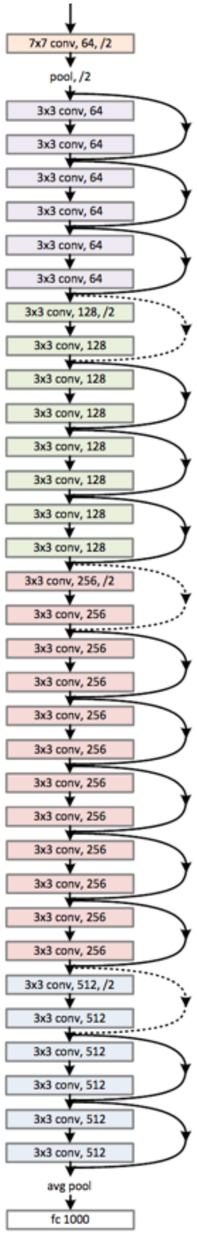


[See also: http://fomoro.com/tools/receptive-fields/index.html]

Source: Isola, Torralba, Freeman

2016: ResNet
>100 conv. layers

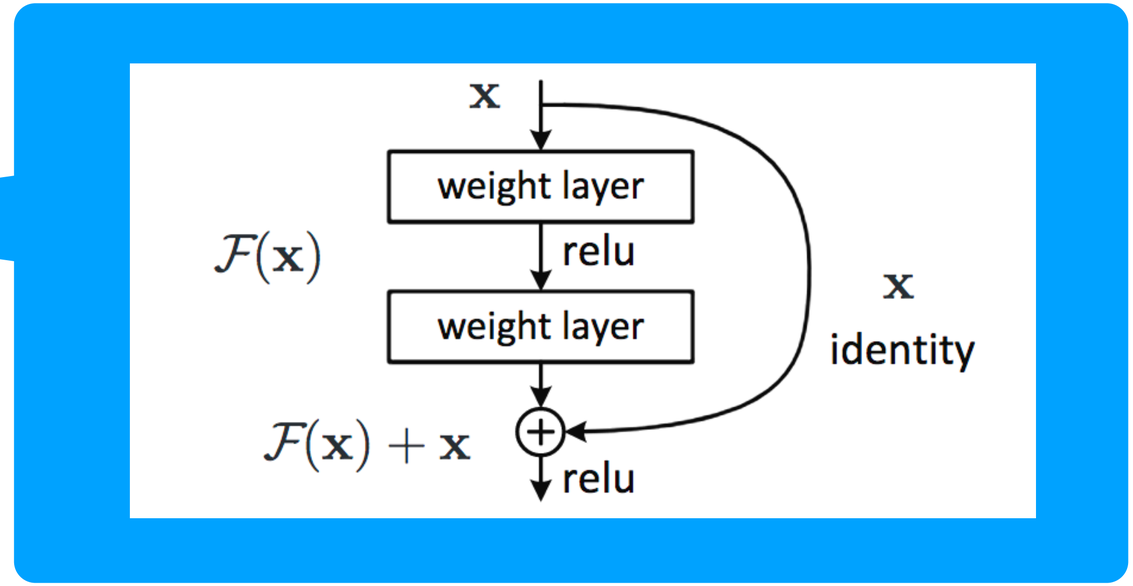
ResNet [He et al, 2016]



Error: 3.6%

Main developments

- Increased depth possible through residual blocks



Source: Isola, Torralba, Freeman

Residual Blocks

Problem: Hard to train very deep nets (50+ layers). This is an optimization issue, not overfitting: shallow models often get higher *training* accuracy than deep ones!

Idea: Make it easy to represent for the network to implement the identity.

Normal convolution + relu:

$$x_{i+1} = \text{relu}(x_i \circ f)$$

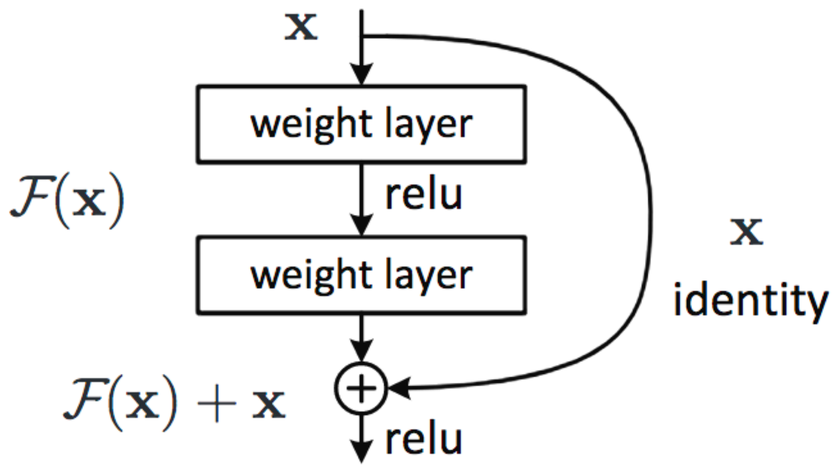
Residual connection

$$x_{i+1} = \text{relu}((x_i \circ f) + x_i)$$

More generally: do multiple convolutions (with nonlinearities) before summing:

$$x_{i+1} = \text{relu}(\mathcal{F}(x_i) + x_i)$$

Residual Blocks



Why do they work?

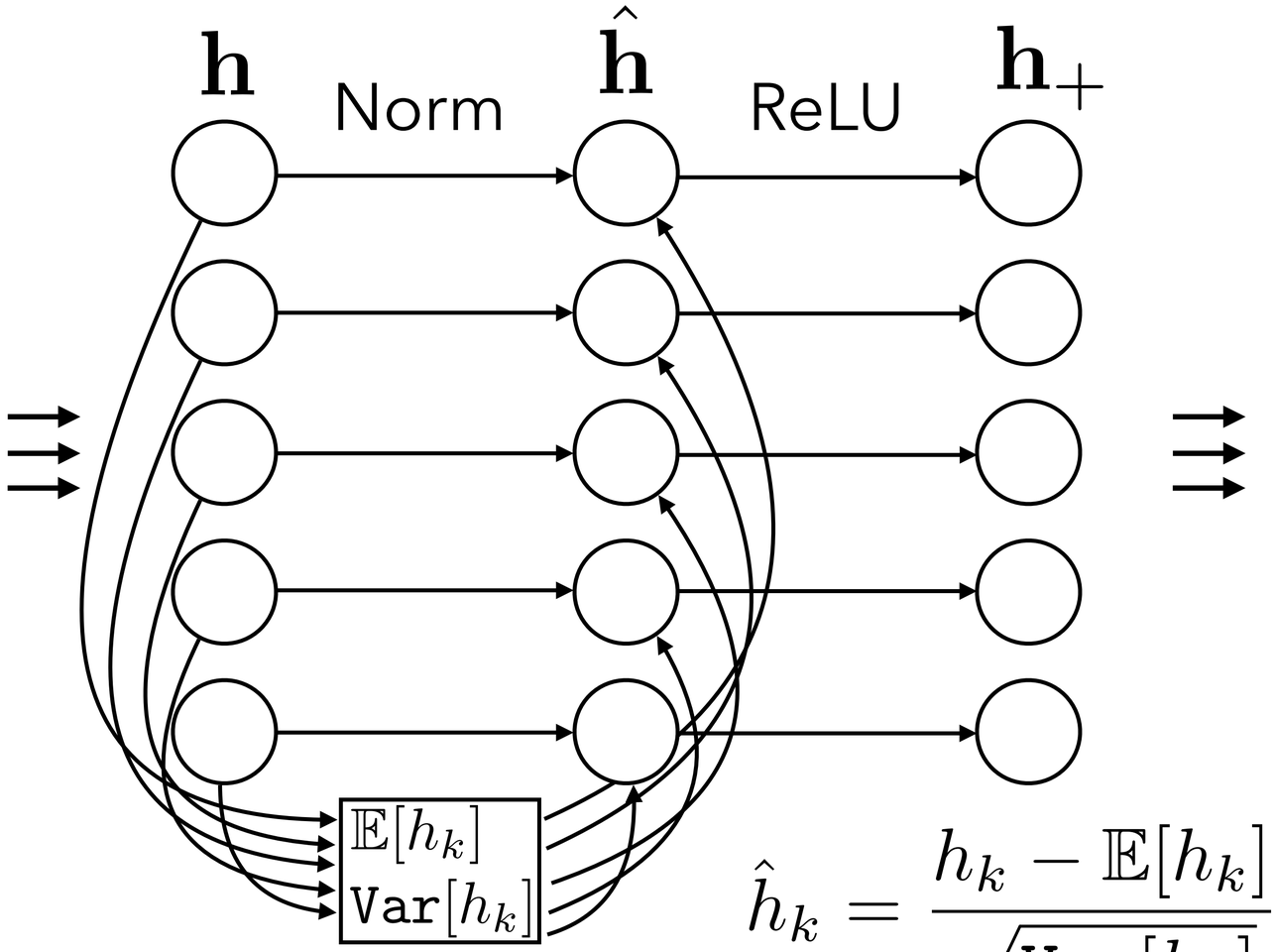
- Gradients can propagate faster (via the identity mapping)
- Within each block, only small residuals have to be learned

Source: Isola, Torralba, Freeman

Normalization layers

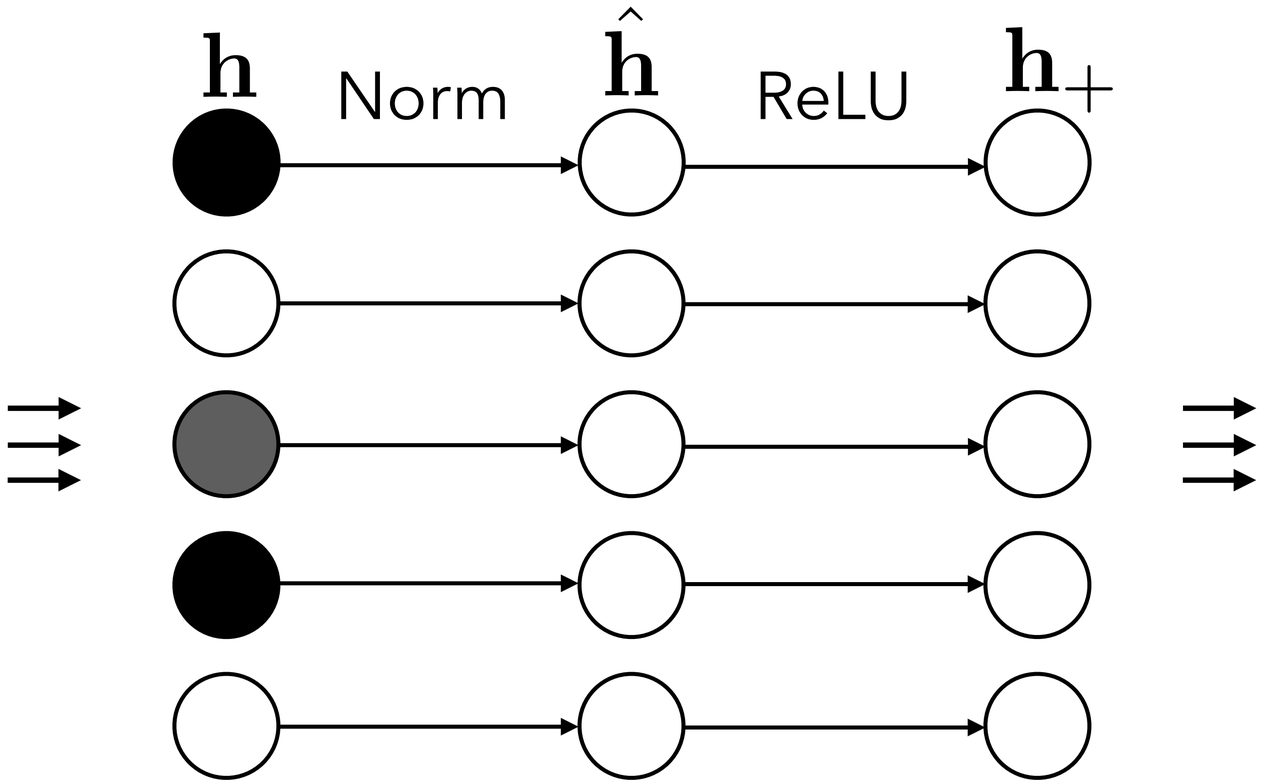
- Standardize activations by subtracting mean and dividing by standard deviation (averaged over all spatial locations).
- This provides a constant “interface” for later layers of the networks. Ensures that the previous layer will have unit variance and zero mean.
- Obtains invariance to mean and variance.
- Can allow you to train with larger learning rate and significantly speed up training!

Normalization layers



Source: Isola, Torralba, Freeman

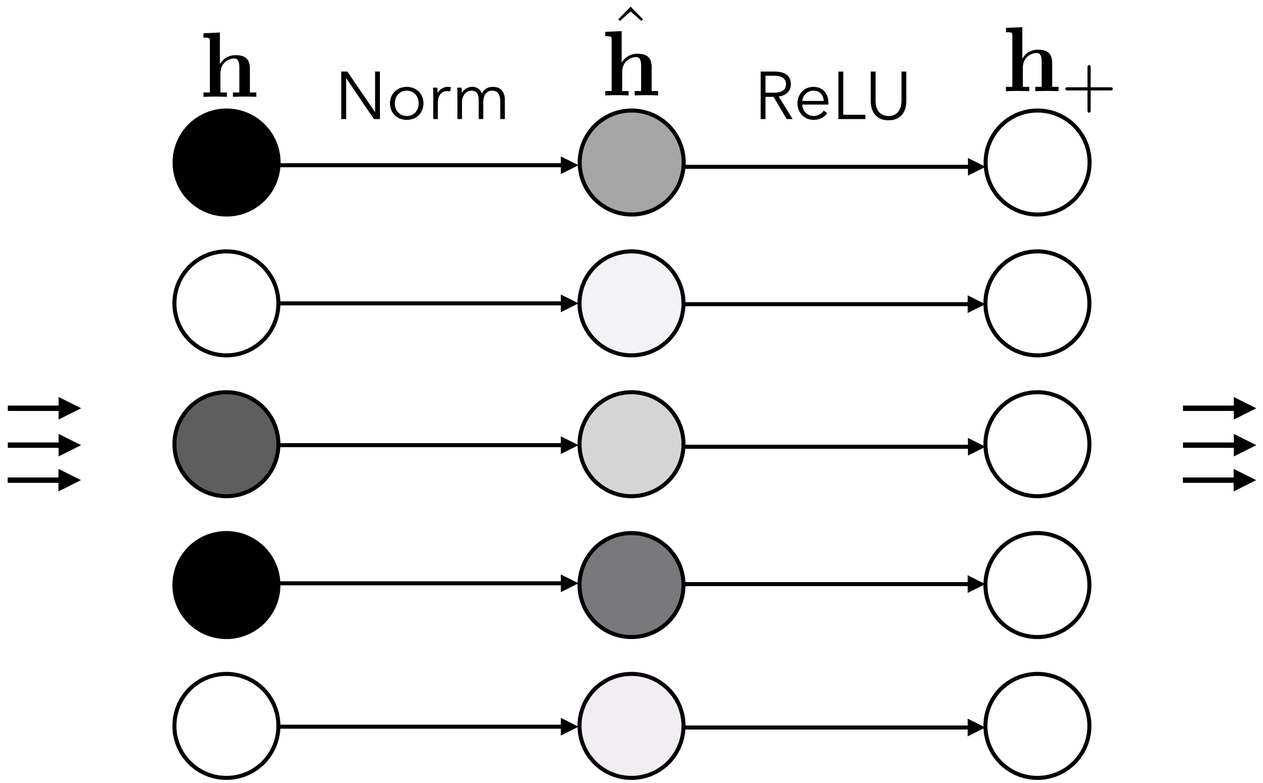
Normalization layers



$$\hat{h}_k = \frac{h_k - \mathbb{E}[h_k]}{\sqrt{\texttt{Var}[h_k]}}$$

Source: Isola, Torralba, Freeman

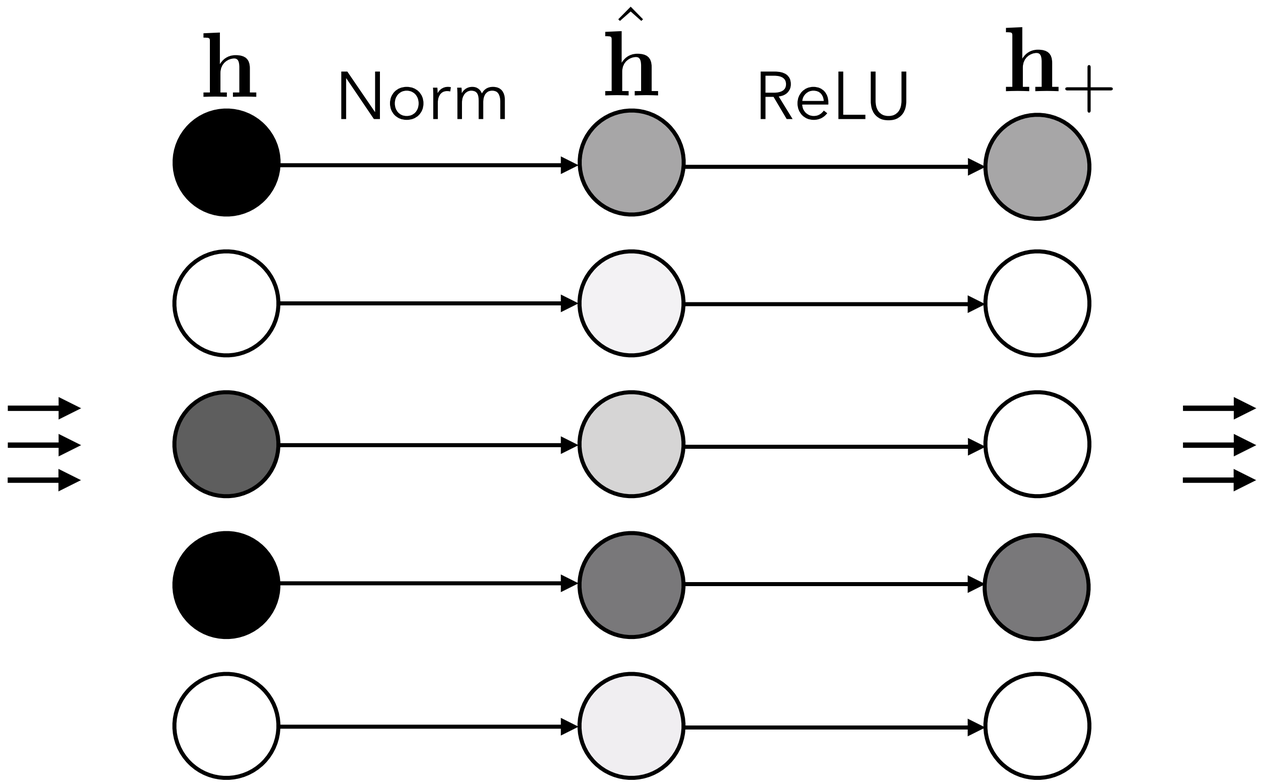
Normalization layers



$$\hat{h}_k = \frac{h_k - \mathbb{E}[h_k]}{\sqrt{\texttt{Var}[h_k]}}$$

Source: Isola, Torralba, Freeman

Normalization layers

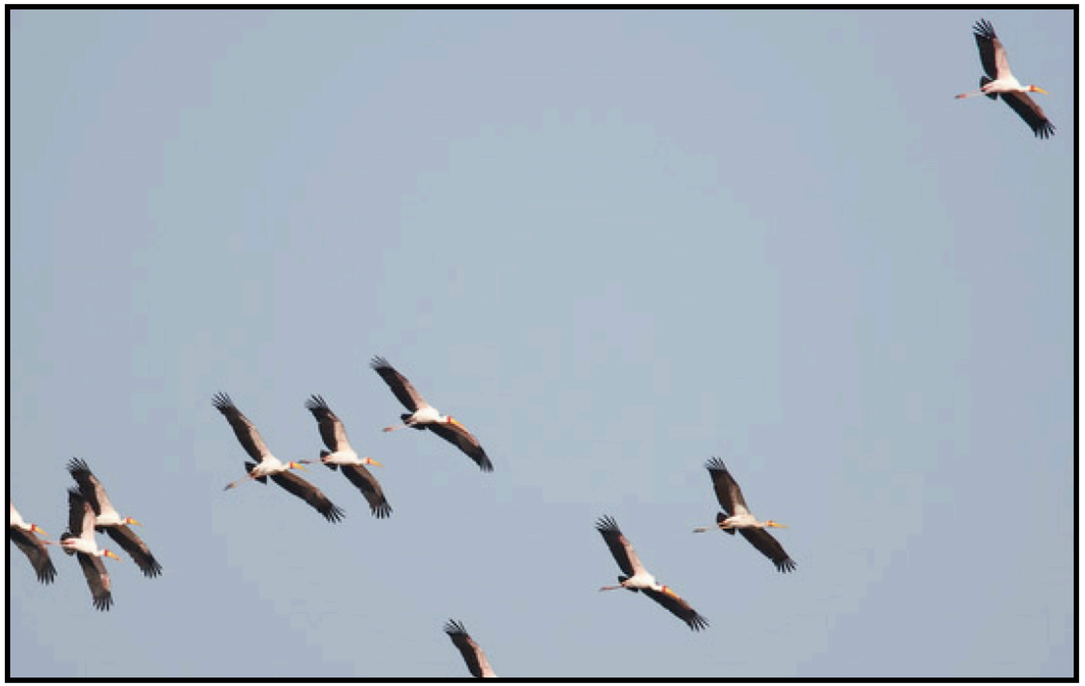


$$\hat{h}_k = \frac{h_k - \mathbb{E}[h_k]}{\sqrt{\texttt{Var}[h_k]}}$$

Source: Isola, Torralba, Freeman

Network designs for generative models

Object recognition: what objects are in the image?

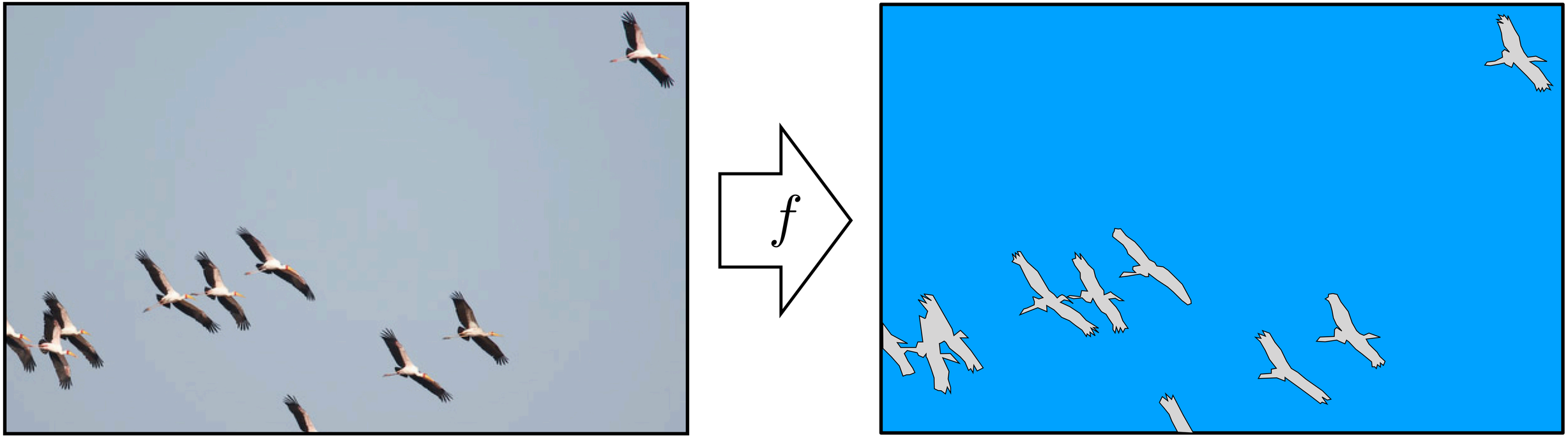


f

"Birds"

Source: Torralba, Freeman, Isola

Predicting something at every pixel

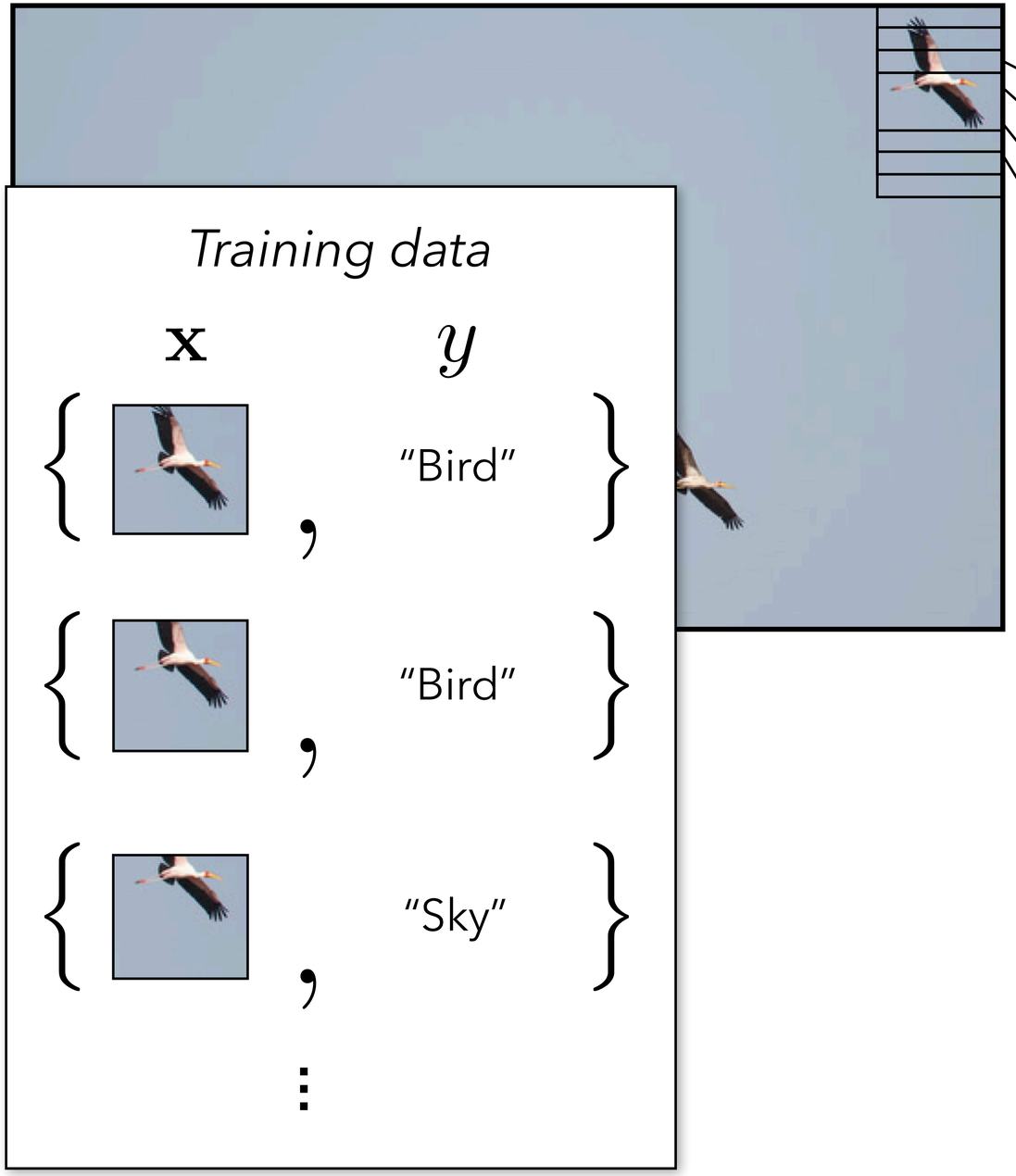


(Colors represent categories)

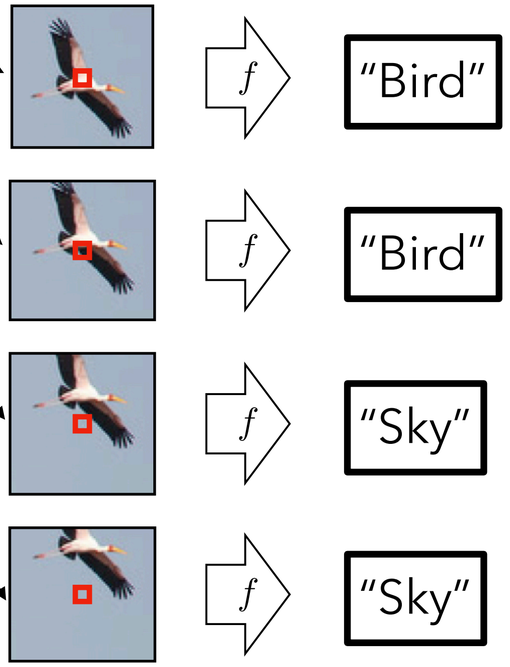
General technique: predict something at every pixel!

Source: Torralba, Freeman, Isola

Idea #1: Independently classify windows

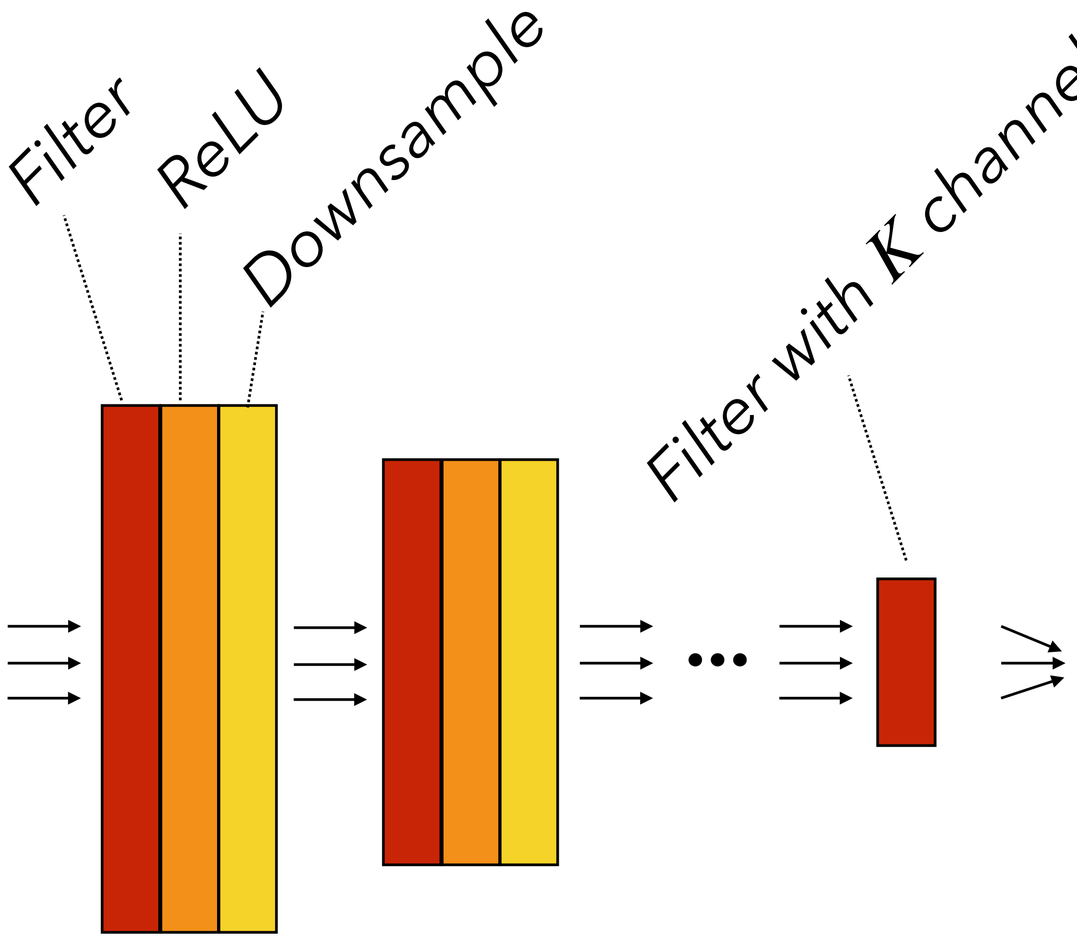


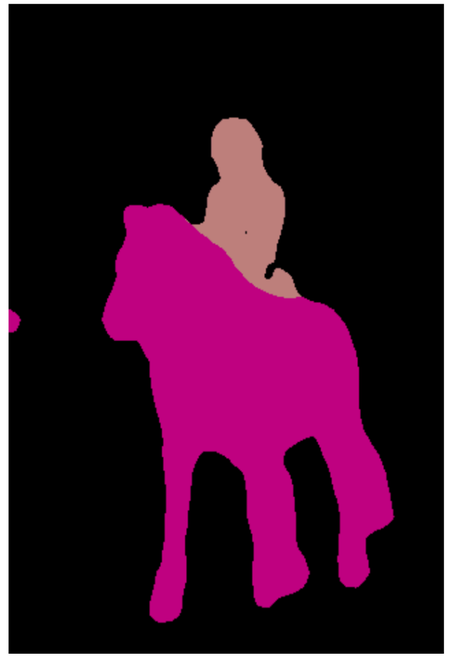
What's the object class of the center pixel?



Source: Torralba, Freeman, Isola

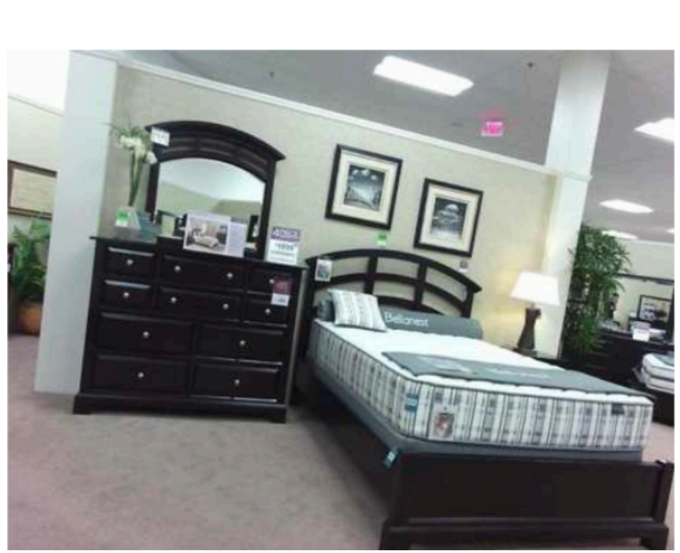
Idea #2: Fully convolutional networks

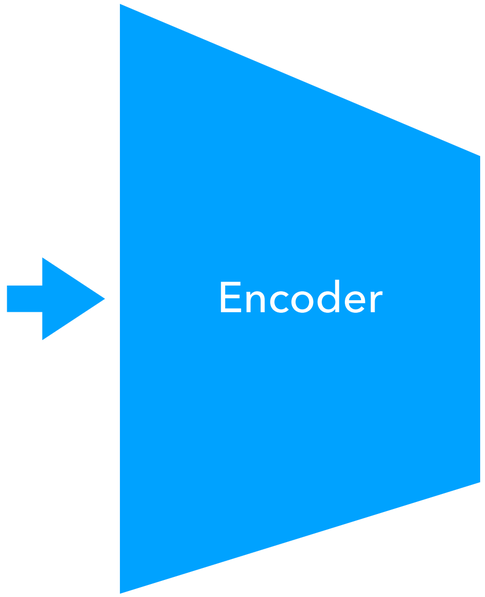




Classification problem with K classes

Idea #3: Encoder-decoder models

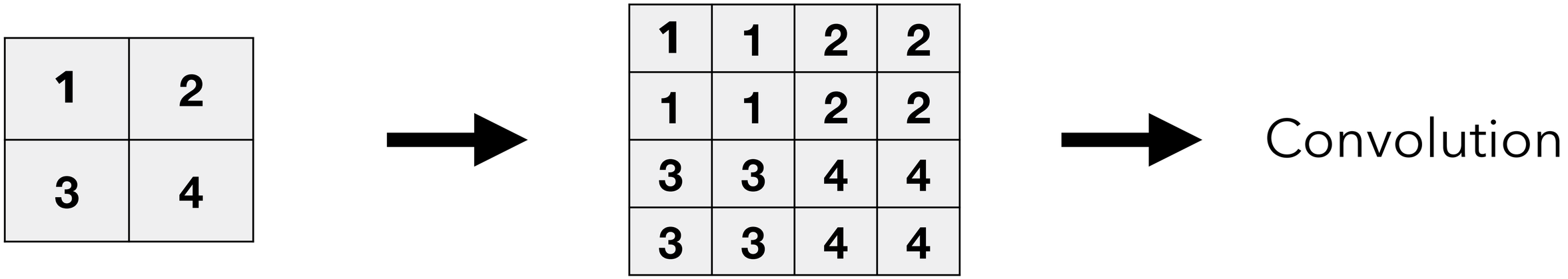




Convolutions **Deconvolutions**

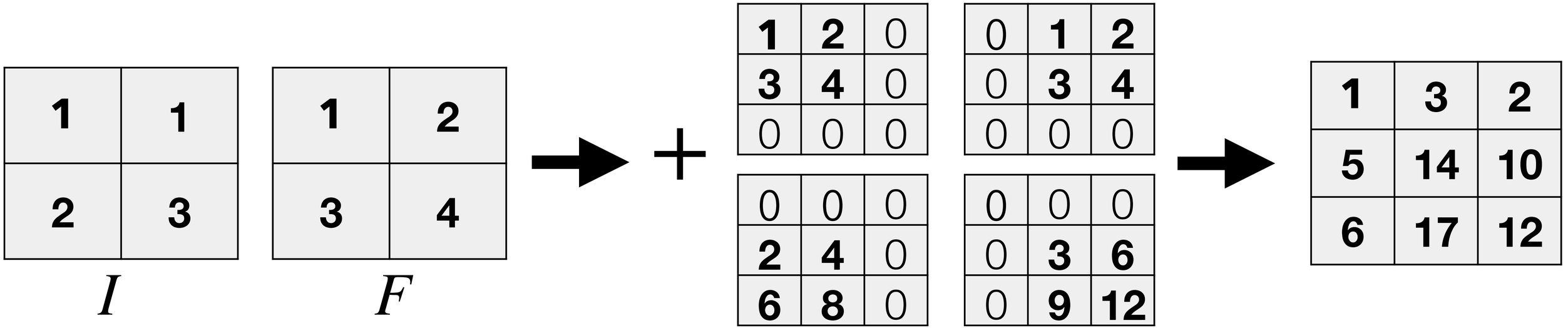
Source: Torralba, Freeman, Isola

Upsampling



- Often using nearest-neighbor upsampling
- Can also use interpolation.
- Produces fewer “checkerboard” artifacts

Transposed convolution



- Weight the filter by the image coefficient and sum.
- Also sometimes called “upconvolution” or “deconvolution”.

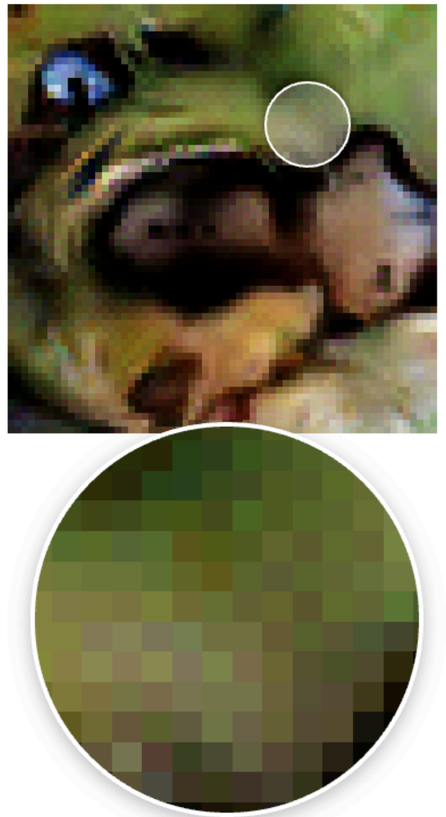
Transposed convolution

1	1
2	3

I

1	2
3	4

F

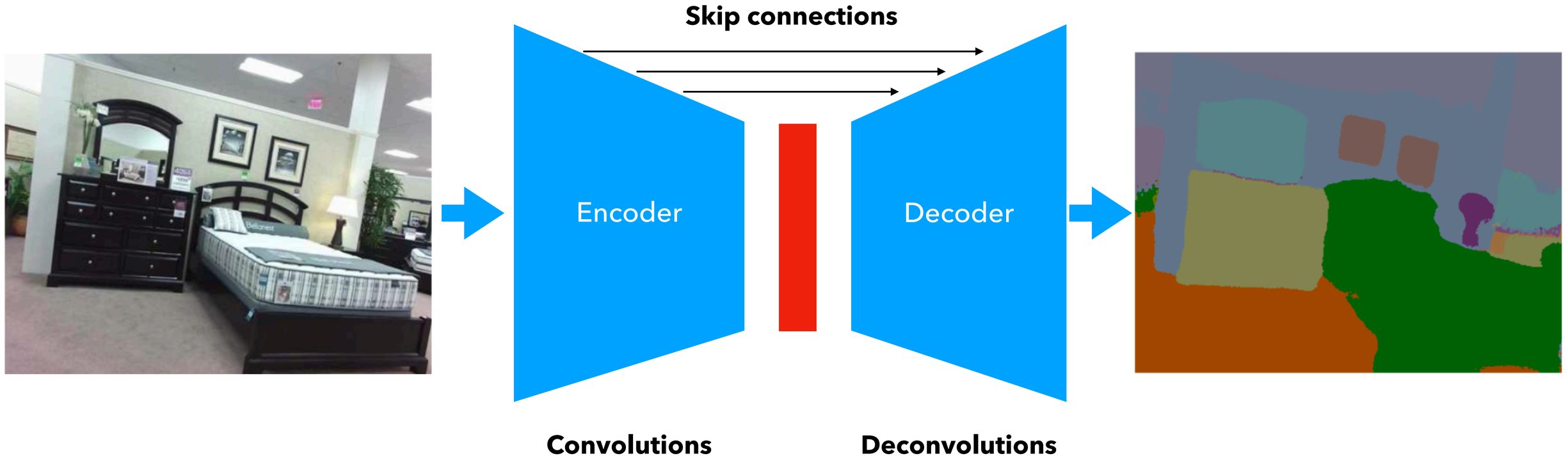


Donahue, et al., 2016 [3]

Can lead to “checkerboard” artifacts.

[Odena et al. Distill article]

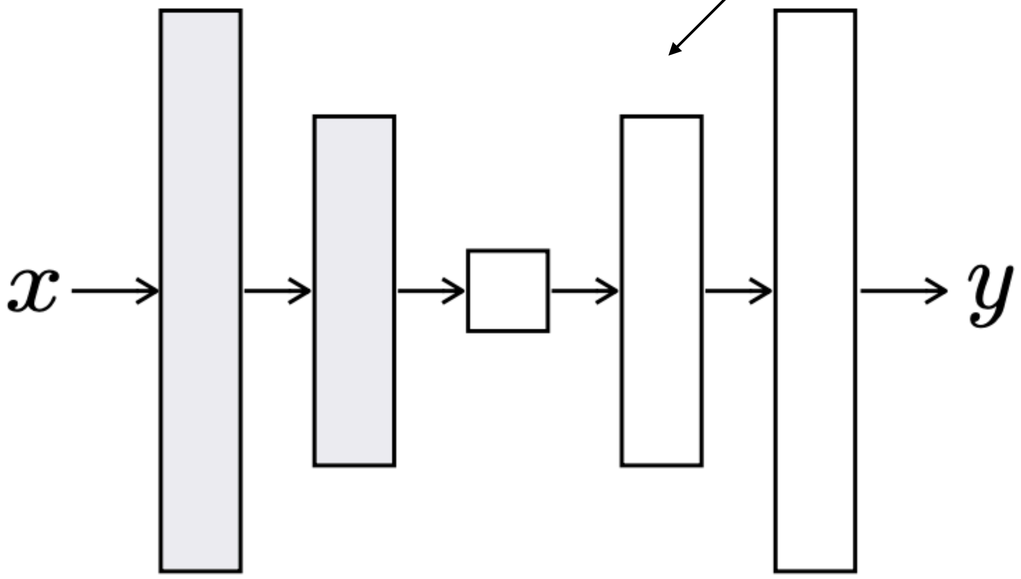
Encoder-decoder architectures



Source: Torralba, Freeman, Isola

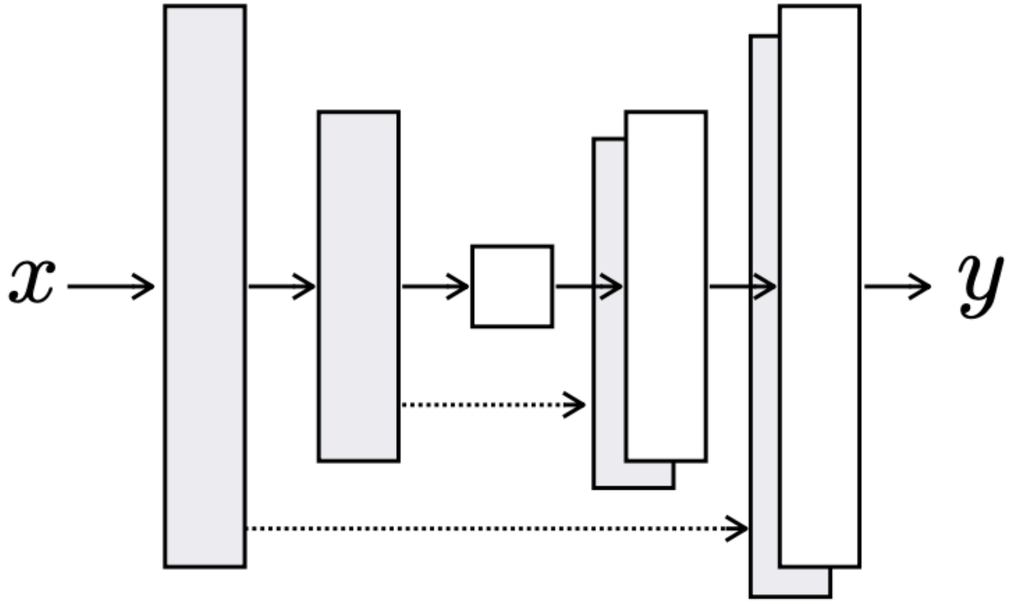
Encoder-decoder architectures

Transposed convolution



"Vanilla" encoder-decoder architecture

Early layers and late layers have same shape. Concatenate channel-wise!

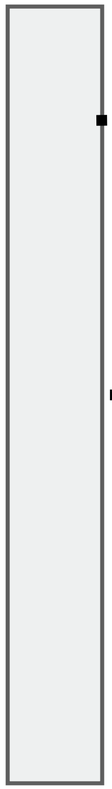


U-Net

Figures from [Isola et al., "Image-to-Image Translation with Conditional Adversarial Networks",

U-net

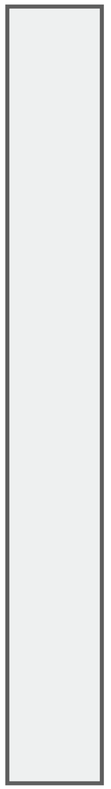
$W \times H$



$\frac{W}{2} \times \frac{H}{2}$

concat

$\cdots$

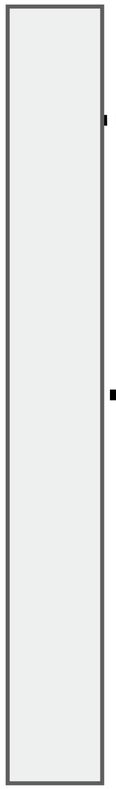


$\frac{W}{2} \times \frac{H}{2}$

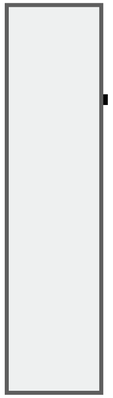
$W \times H$

U-net

$W \times H$



$\frac{W}{2} \times \frac{H}{2}$

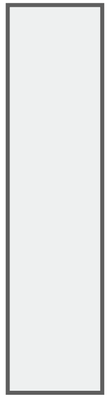


$\frac{W}{4} \times \frac{H}{4}$

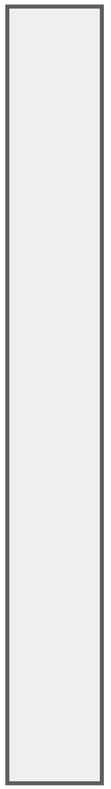
concat

concat

• • •



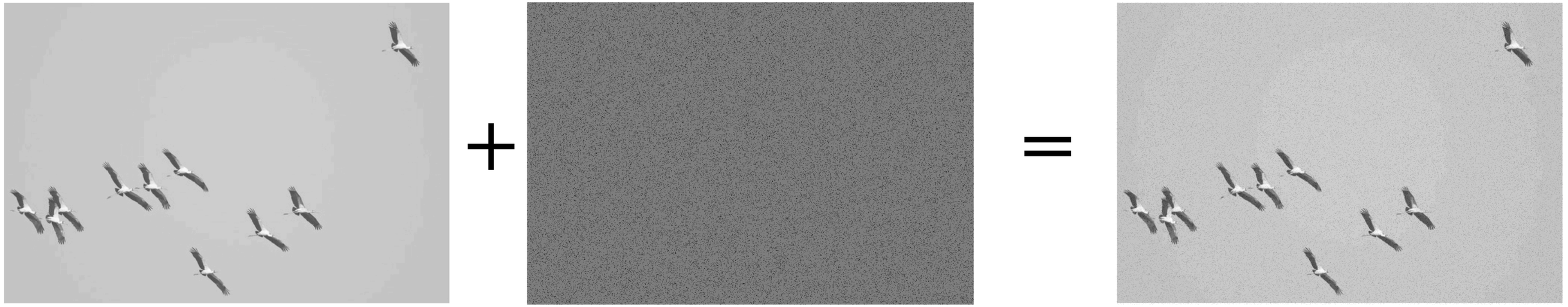
$\frac{W}{4} \times \frac{H}{4}$



$\frac{W}{2} \times \frac{H}{2}$

$W \times H$

Other uses for U-nets



Goal: recover the original image

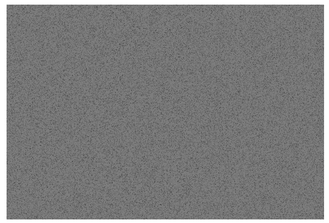
Recall: denoising problem

Denoising

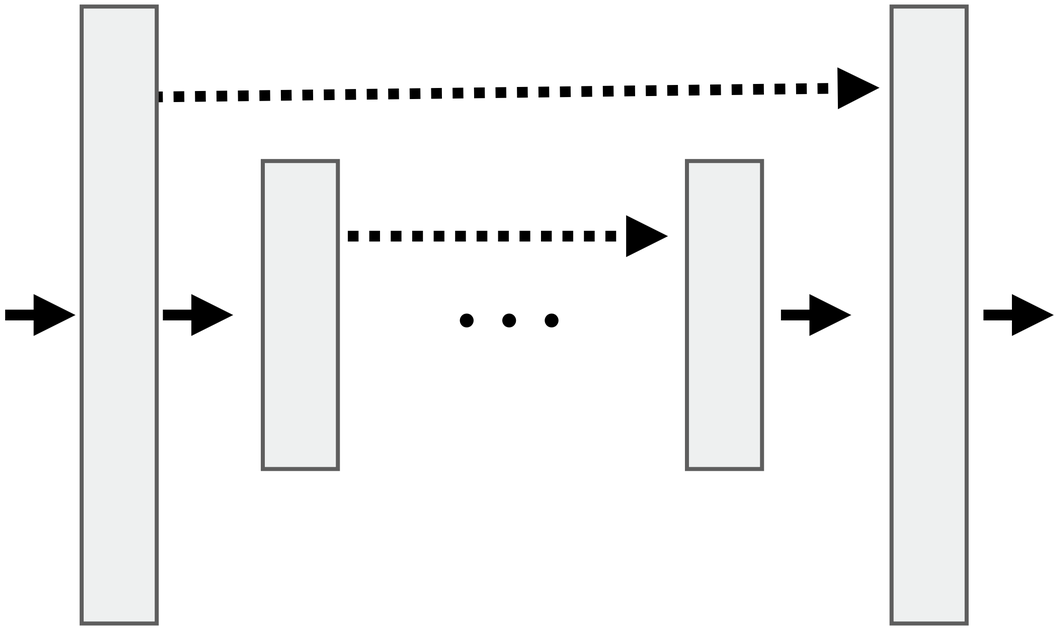
=

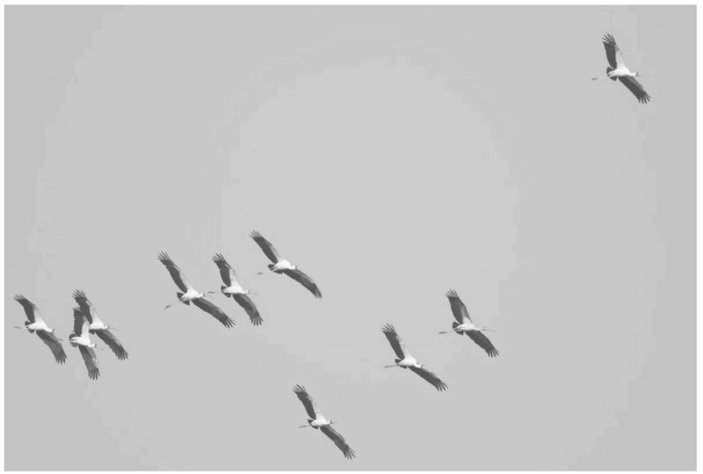
$\mathbf{x}_{\text{clean}}$

+



random noise





$\hat{\mathbf{x}}$

Loss: $\|\mathbf{x}_{\text{clean}} - \hat{\mathbf{x}}\|^2$

Next class: variational autoencoders