

COMP547

DEEP UNSUPERVISED LEARNING

Lecture #2 – Neural Networks Basics
and Spatial Processing with CNNs

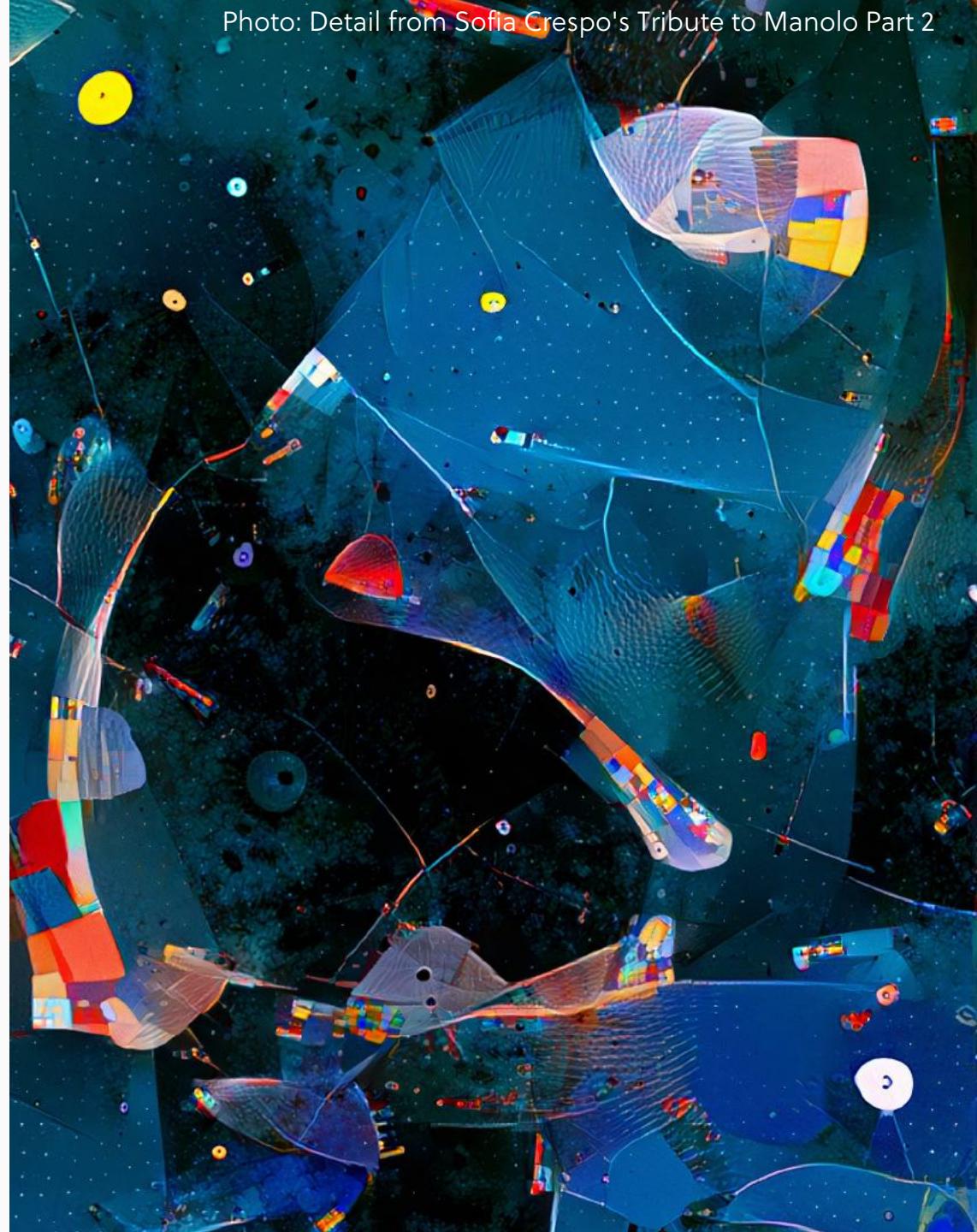


KOÇ
UNIVERSITY

Aykut Erdem // Koç University // Spring 2025

Previously on COMP547

- course logistics
- course topics
- what is deep unsupervised learning



Good news, everyone!

- Only 6 students have completed the survey so far!
 - It will be up until everyone completes the survey
- My office hour will be on Tuesdays btw 10:00-11:00
- PS hour will start next week!
 - In person!



Lecture overview

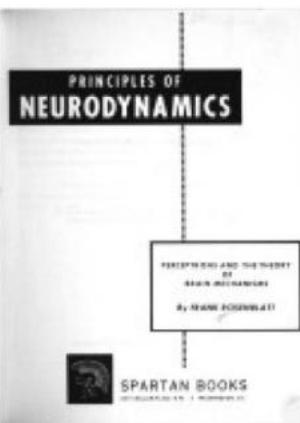
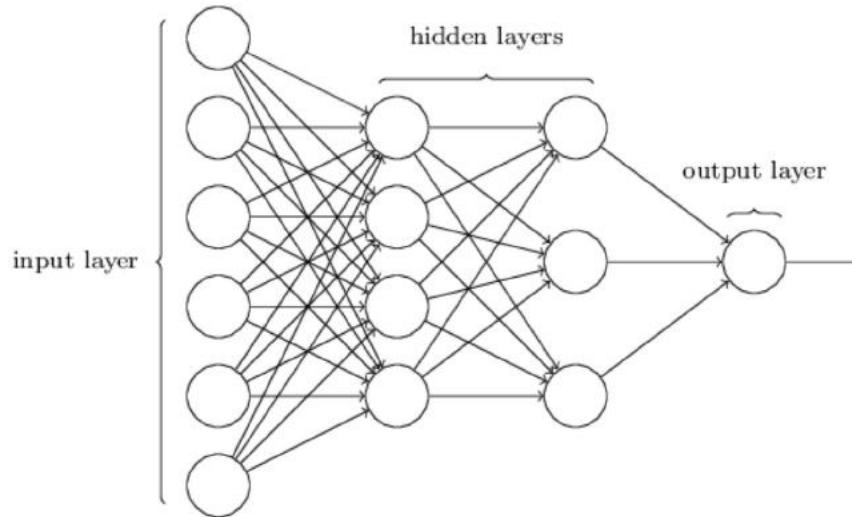
- deep learning
- computation in a neural net
- optimization
- backpropagation
- training tricks
- convolutional neural networks

Disclaimer: Much of the material and slides for this lecture were borrowed from

- Costis Daskalakis and Aleksander Madry's MIT 6.883 class
- Bill Freeman, Antonio Torralba and Phillip Isola's MIT 6.869 class

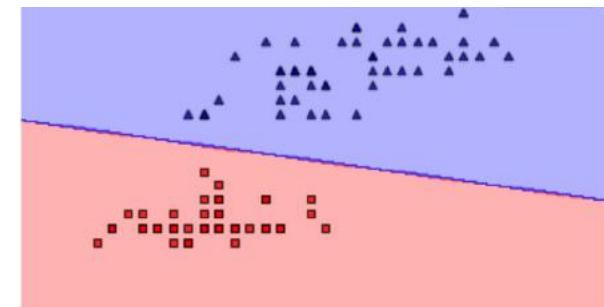
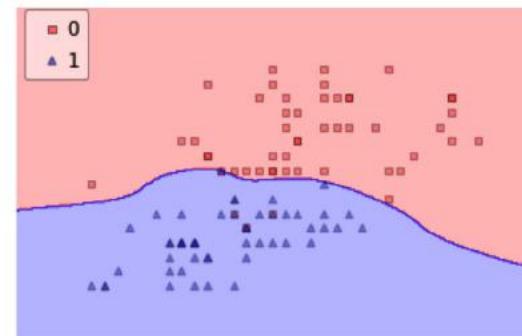
Humble beginnings

- Perceptron [Rosenblatt '58]

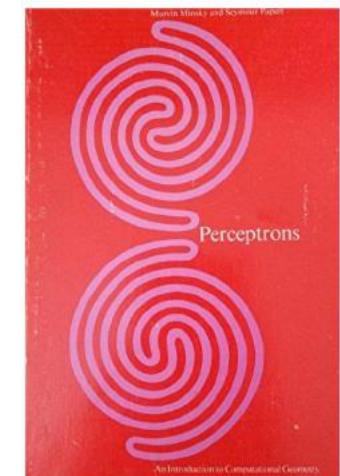


$$a_1 \xrightarrow{w_1} \text{node} \quad a_2 \xrightarrow{w_2} \text{node} \quad a_3 \xrightarrow{w_3} \text{node} \quad \vdots \quad a_n \xrightarrow{w_n} \text{node}$$
$$\sigma\left(b + \sum_{i=1}^n a_i w_i\right)$$

A diagram of a single neuron. Inputs $a_1, a_2, a_3, \dots, a_n$ enter the neuron through weights $w_1, w_2, w_3, \dots, w_n$. The neuron's output is given by the equation $\sigma\left(b + \sum_{i=1}^n a_i w_i\right)$. A red arrow points from this diagram to the next figure.

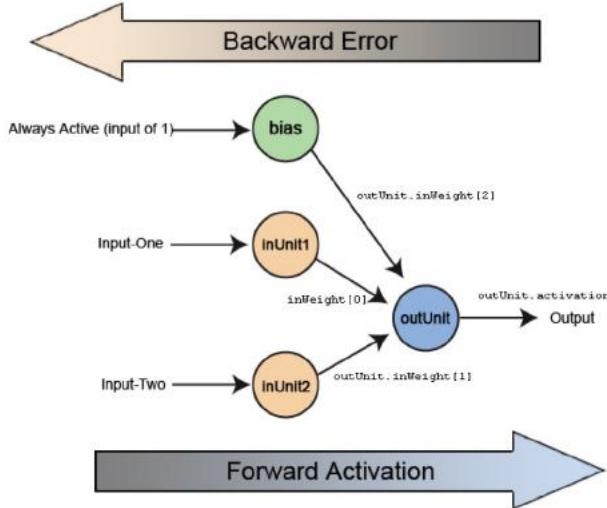


- Criticism of Perceptrons (XOR affair) [Minsky Papert '69]
 - Effectively causes a "deep learning winter"

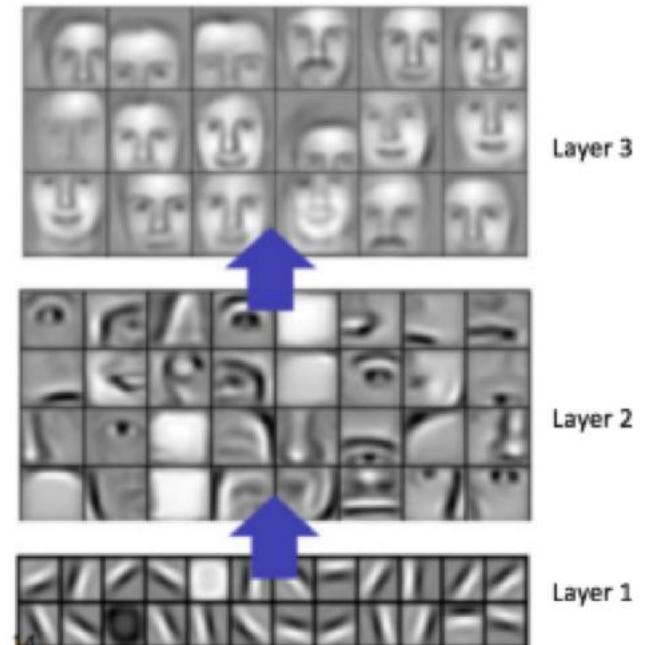


(Early) Spring

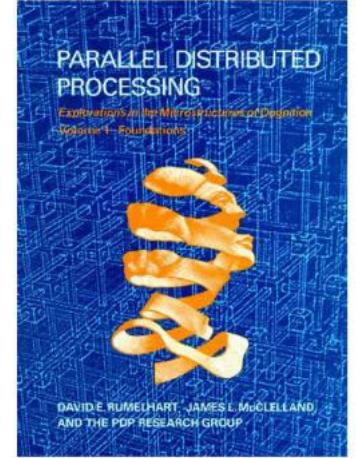
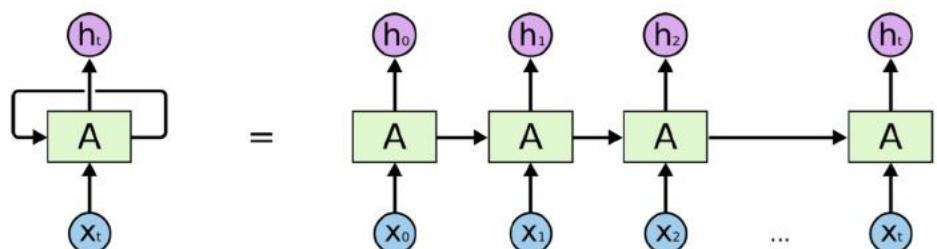
- Back-propagation [Rumelhart et al. '86, LeCun '85, Parker '85]



- Convolutional layers [LeCun et al. '90]

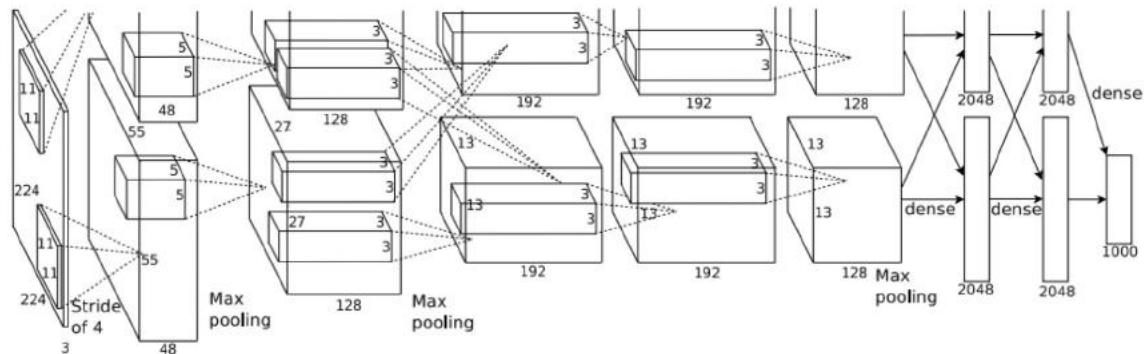
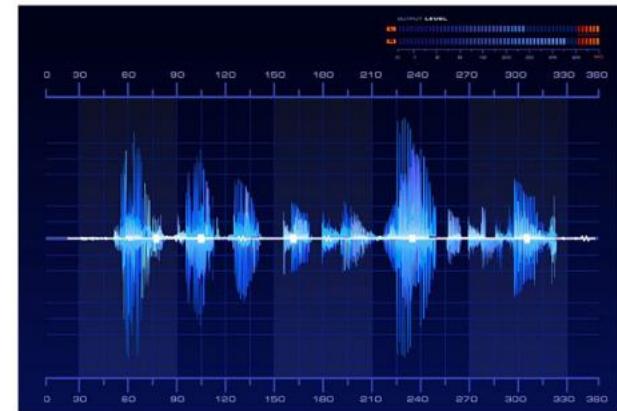


- Recurrent Neural Networks/Long Short-Term Memory (LSTM) [Hochreiter Schmidhuber '97]

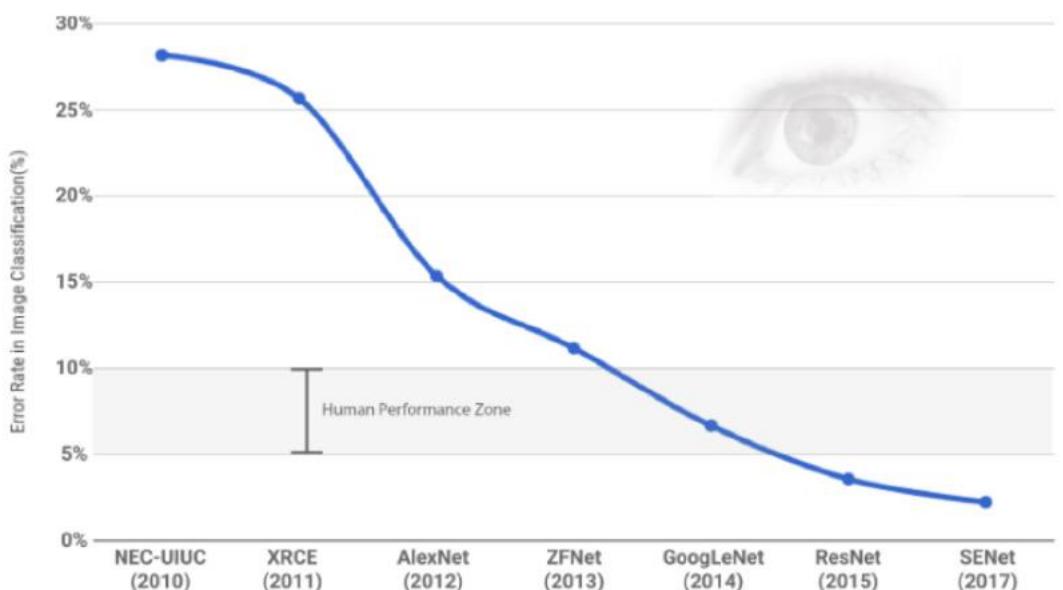


Summer

- 2006: First big success: speech recognition
- 2012: Breakthrough in computer vision: AlexNet [Krizhevsky et al. '12]



- 2015: Deep learning-based vision models outperform humans



What enabled this success?

- Better architectures (e.g., ReLUs) and regularization techniques (e.g. Dropout)

IMAGENET

- Sufficiently large datasets



- Enough computational power



Deep learning

- Modeling the world is incredibly complicated. We need high capacity models.
- In the past, we didn't have enough data to fit these models. But now we do!
- We want a class of **high capacity models** that are **easy to optimize**.

Deep neural networks!

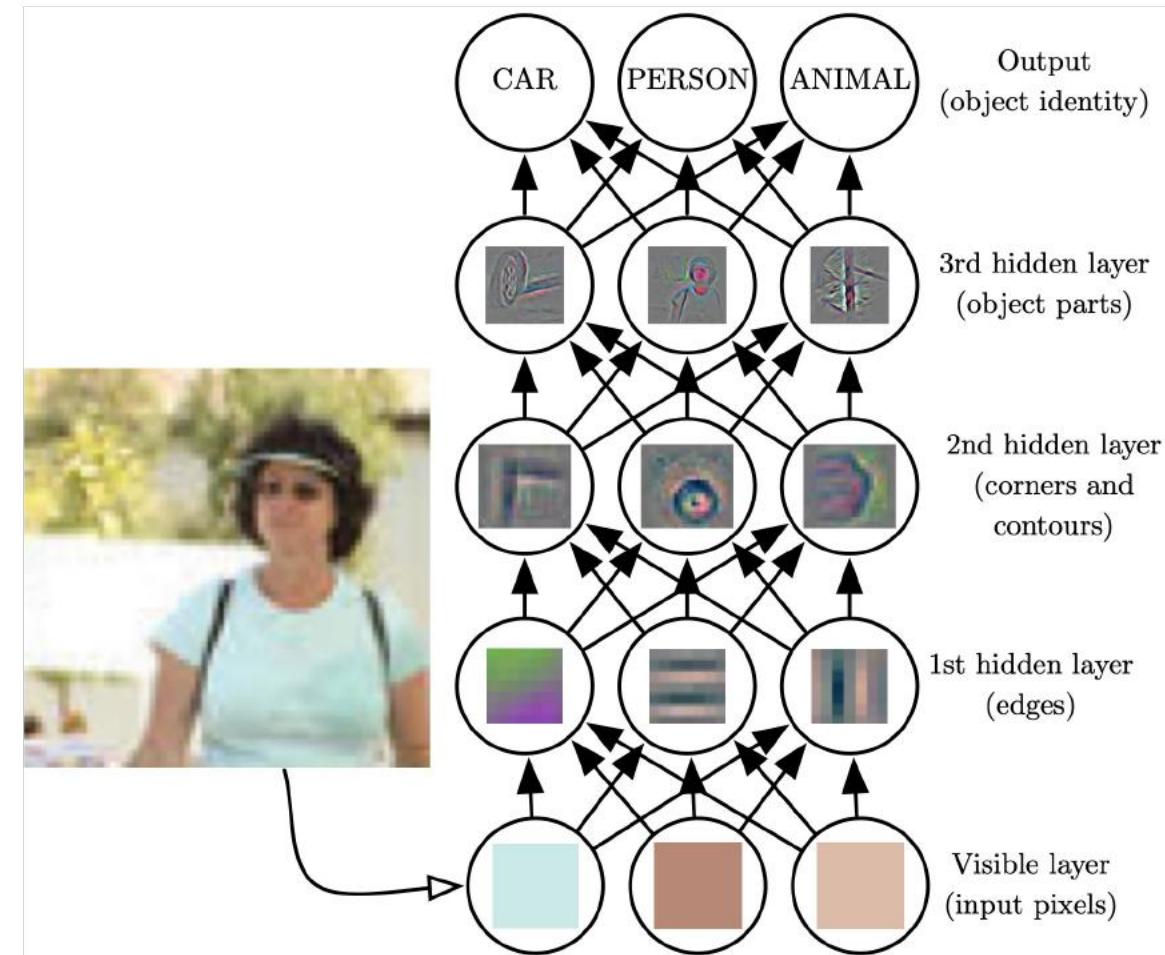
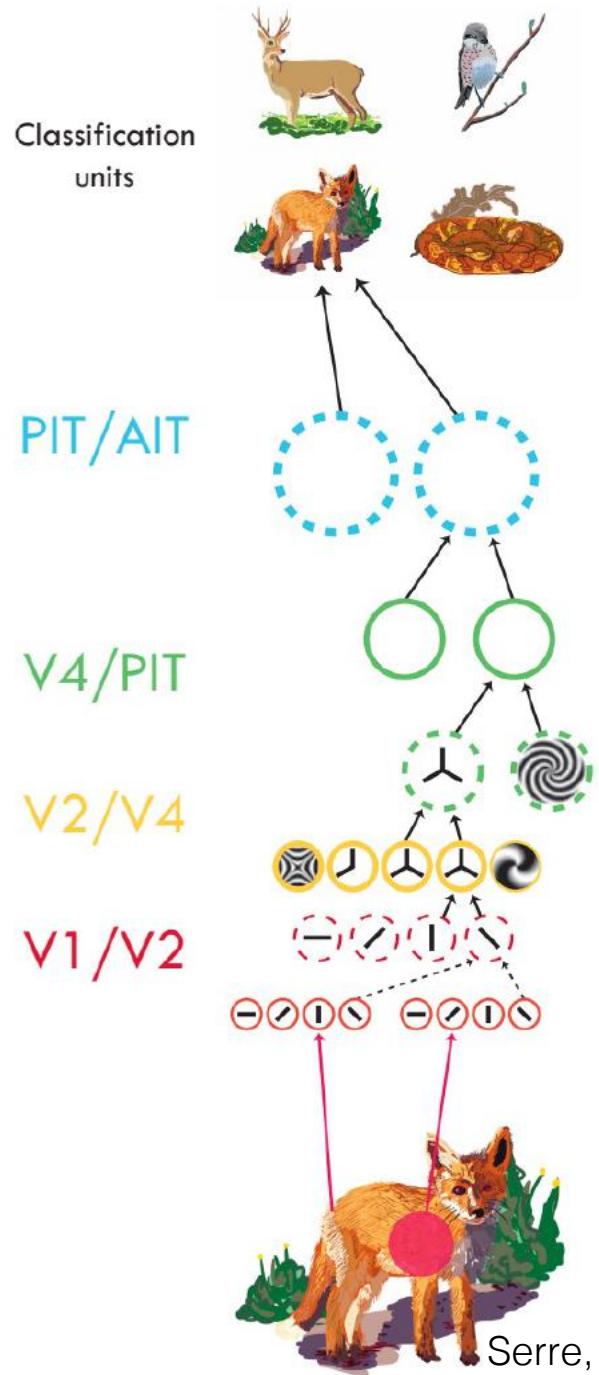
What is deep learning?



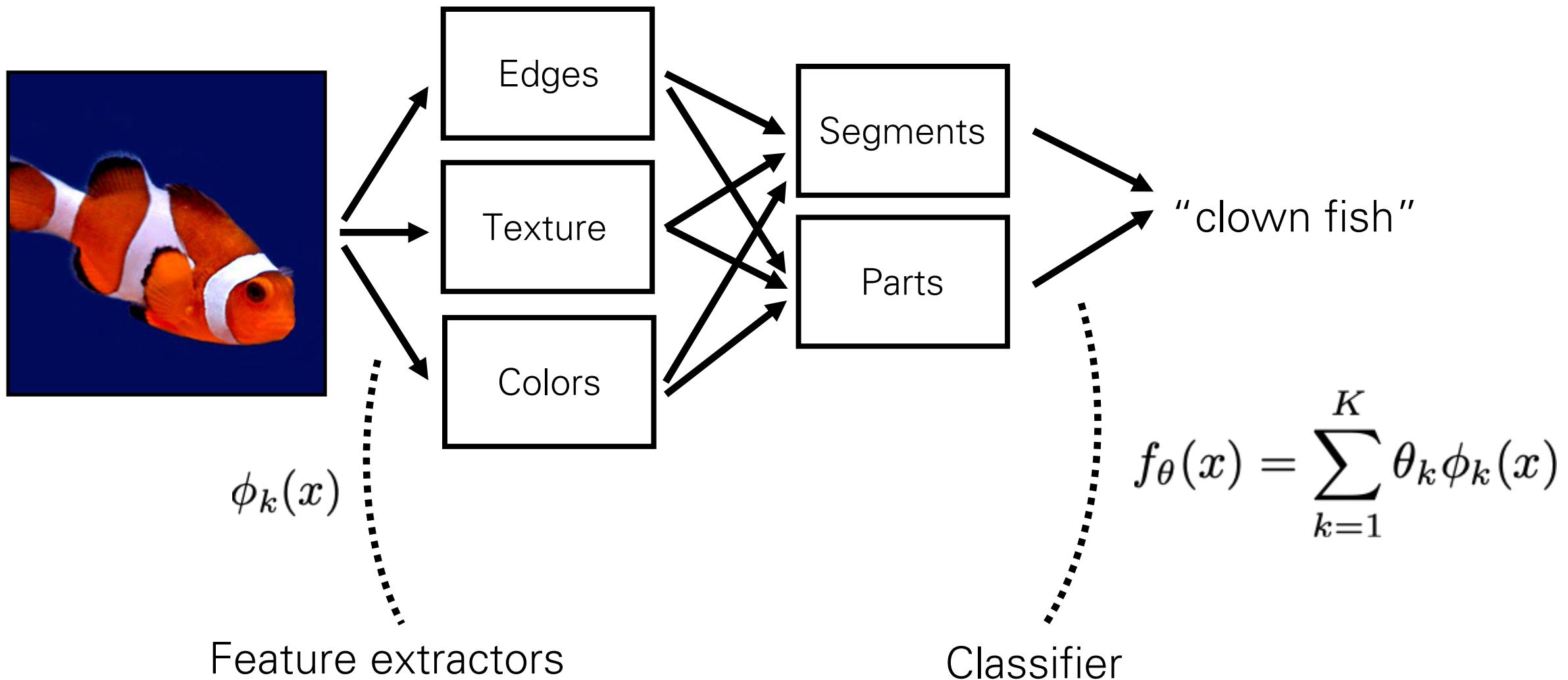
“Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction.”

– Yann LeCun, Yoshua Bengio and Geoff Hinton

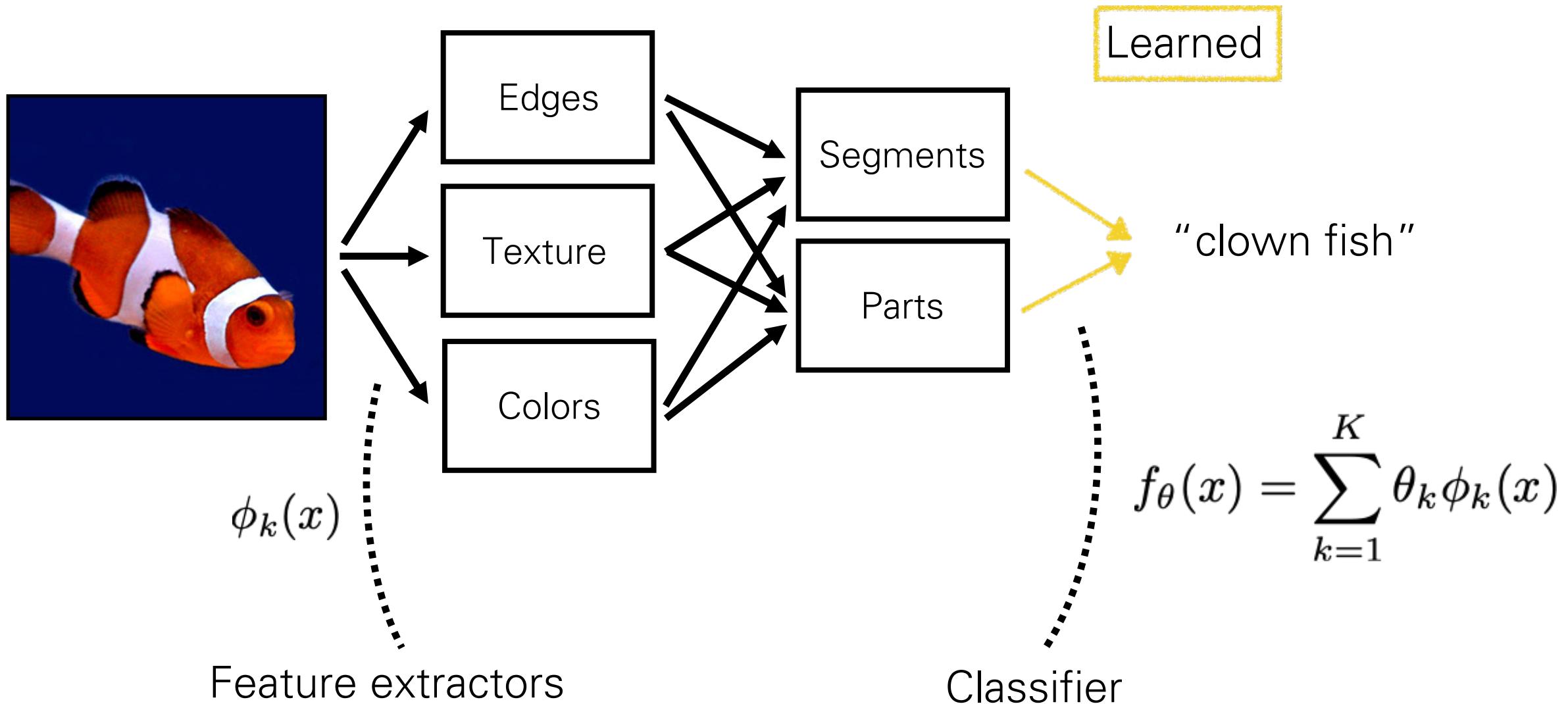
Y. LeCun, Y. Bengio, G. Hinton, "Deep Learning", Nature, Vol. 521, 28 May 2015



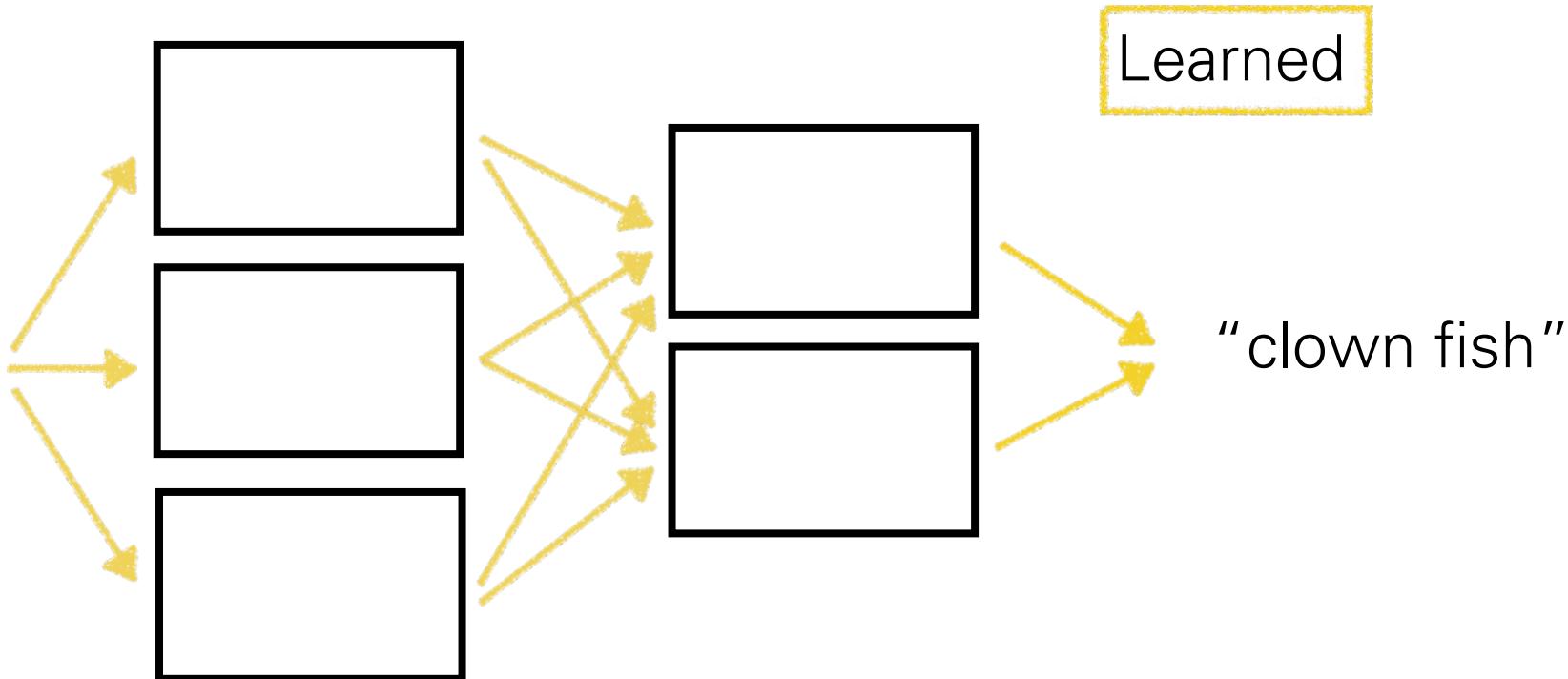
Object recognition



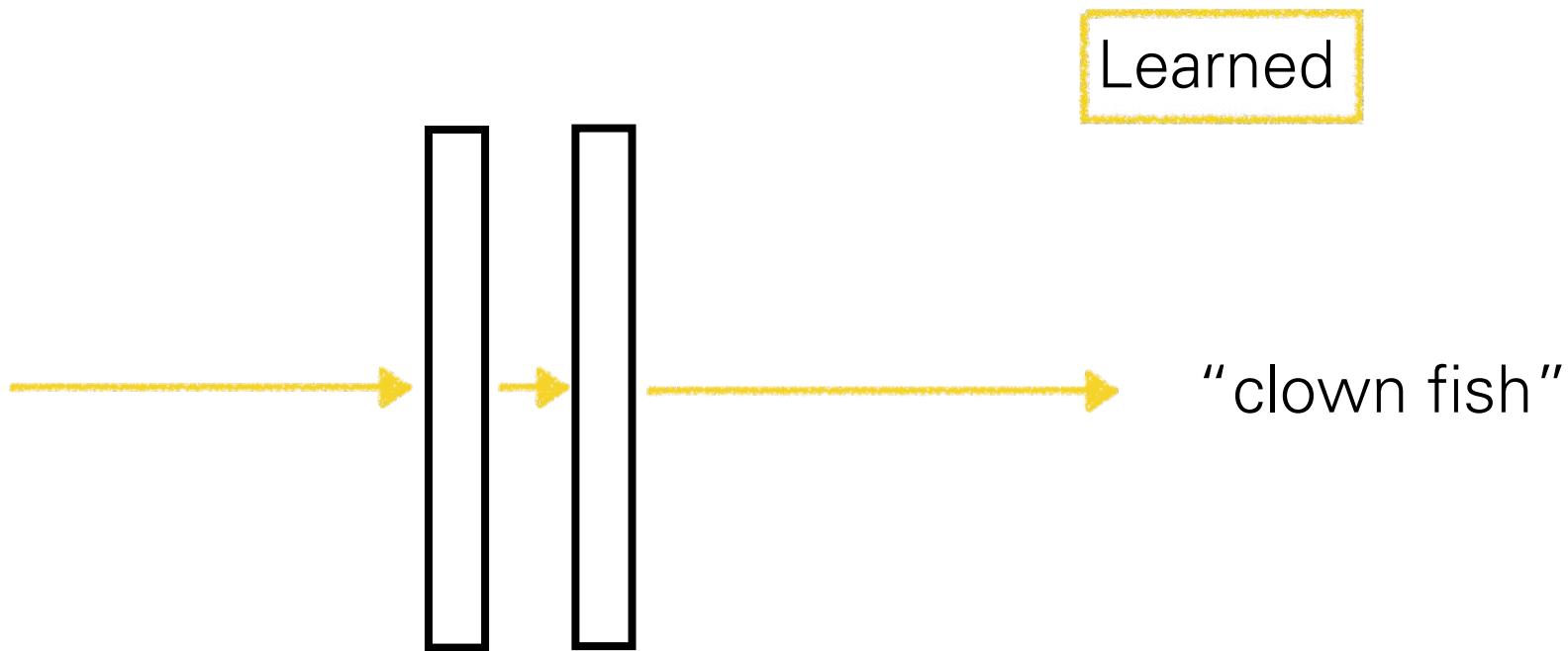
Object recognition



Object recognition

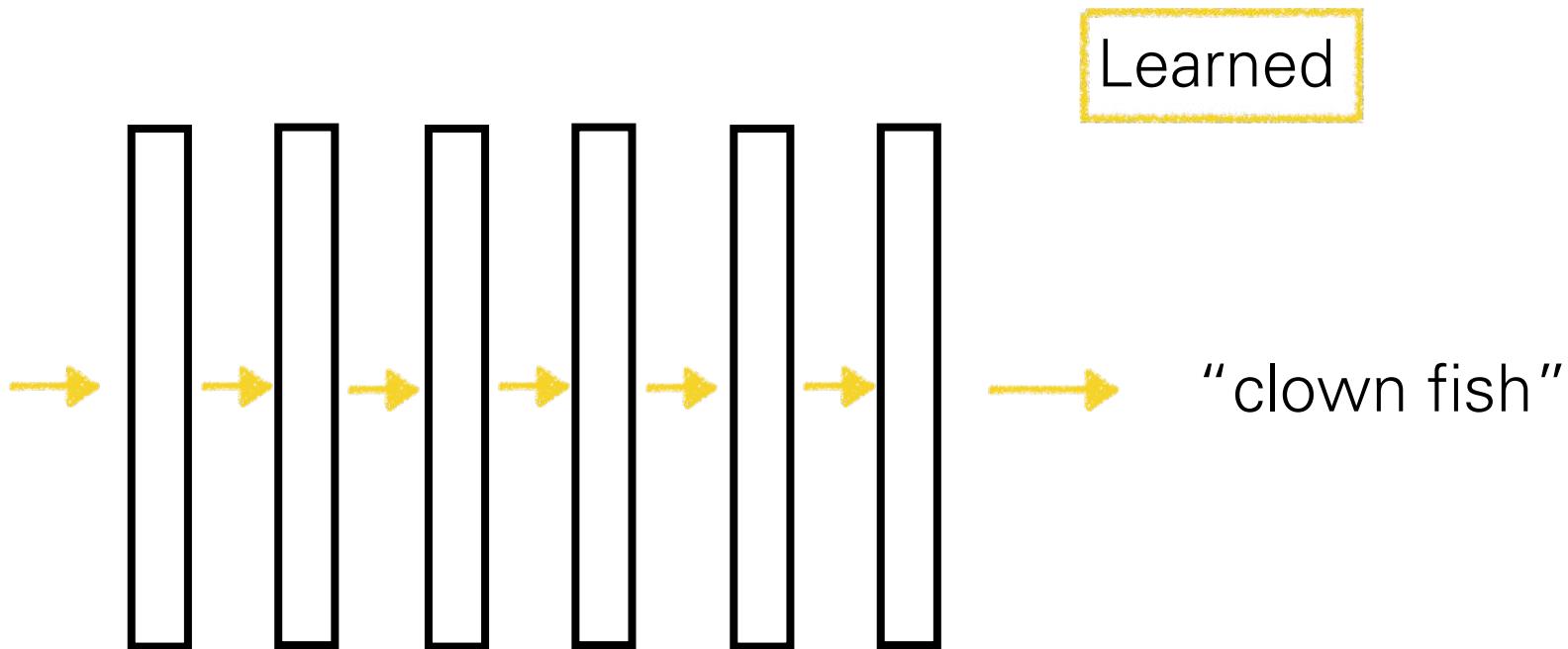


Object recognition



Neural net

Object recognition

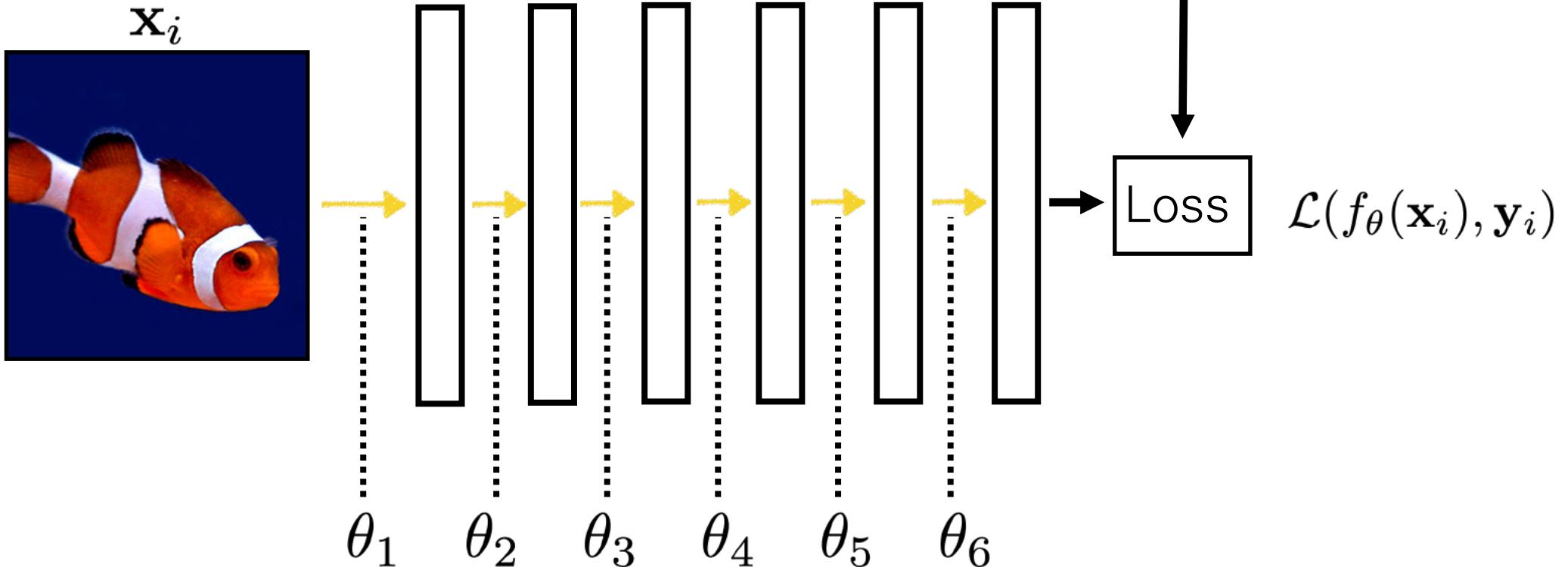


Deep neural net

Deep learning

\mathbf{y}_i
“clown fish”

Learned



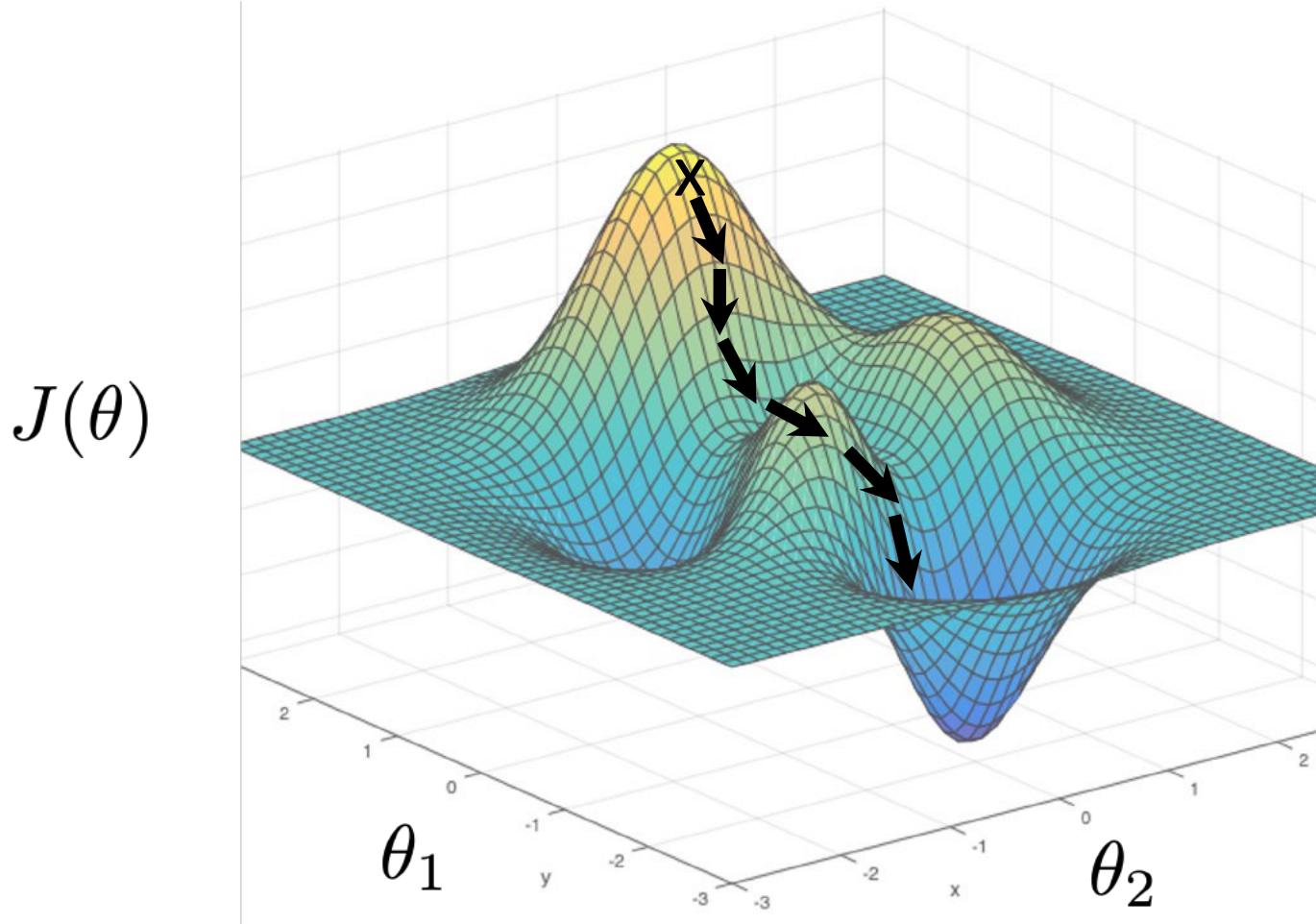
$$\theta^* = \arg \min_{\theta} \sum_{i=1}^N \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)$$

Gradient descent

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

$\underbrace{\hspace{10em}}_{J(\theta)}$

Gradient descent



$$\theta^* = \arg \min_{\theta} J(\theta)$$

Gradient descent

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

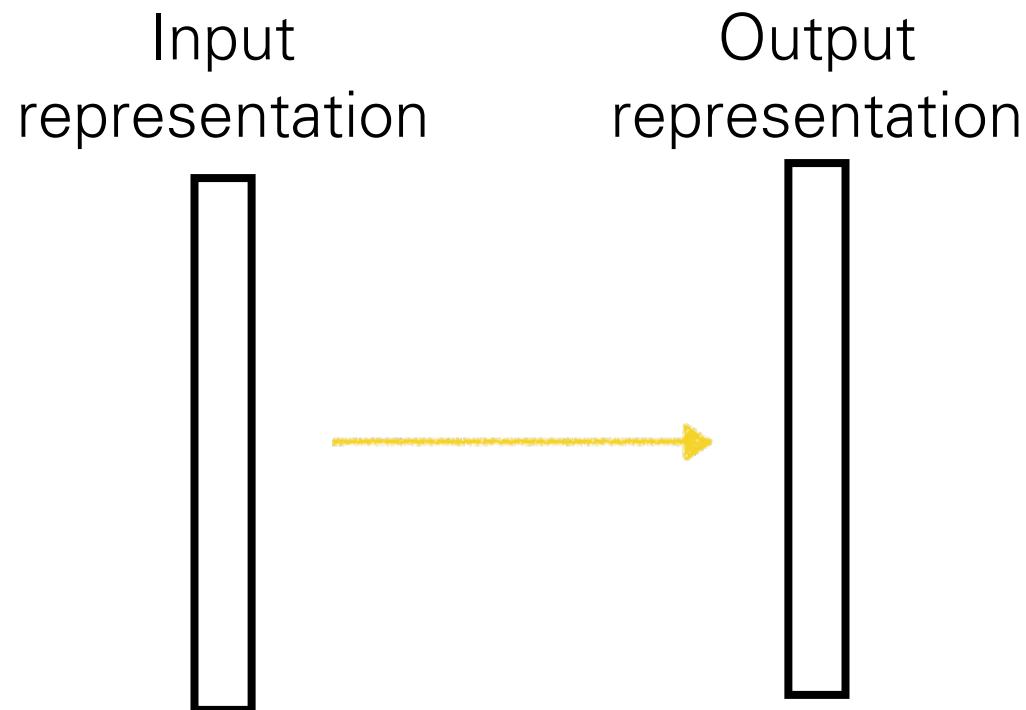
$\underbrace{\hspace{10em}}_{J(\theta)}$

One iteration of gradient descent:

$$\theta^{t+1} = \theta^t - \eta_t \frac{\partial J(\theta)}{\partial \theta} \Big|_{\theta=\theta^t}$$

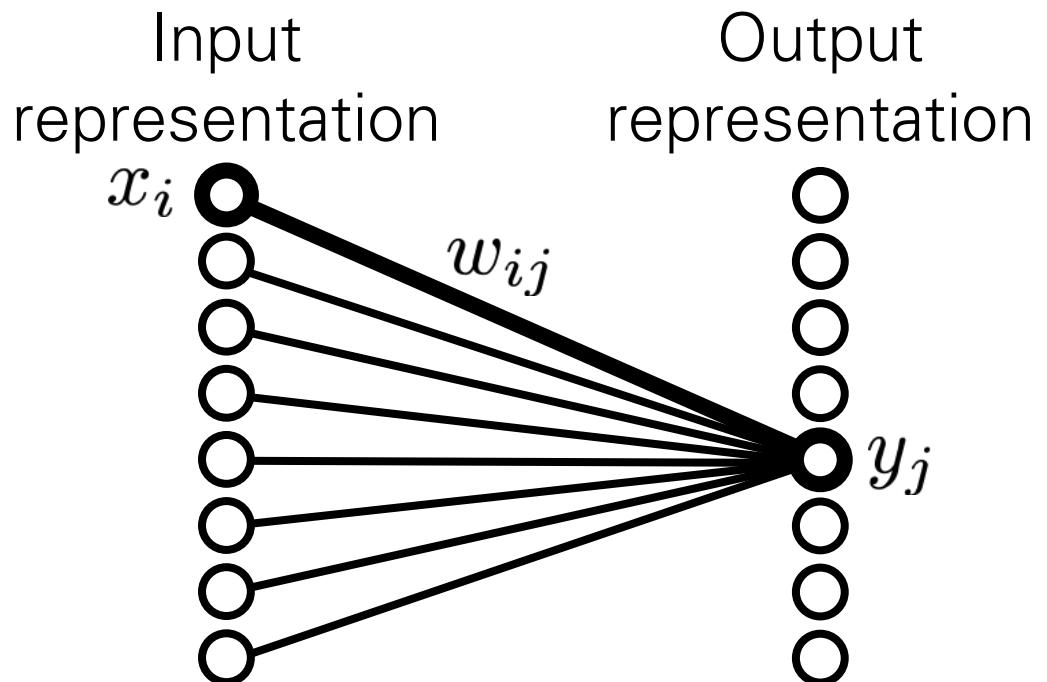
↓
learning rate

Computation in a neural net



Computation in a neural net

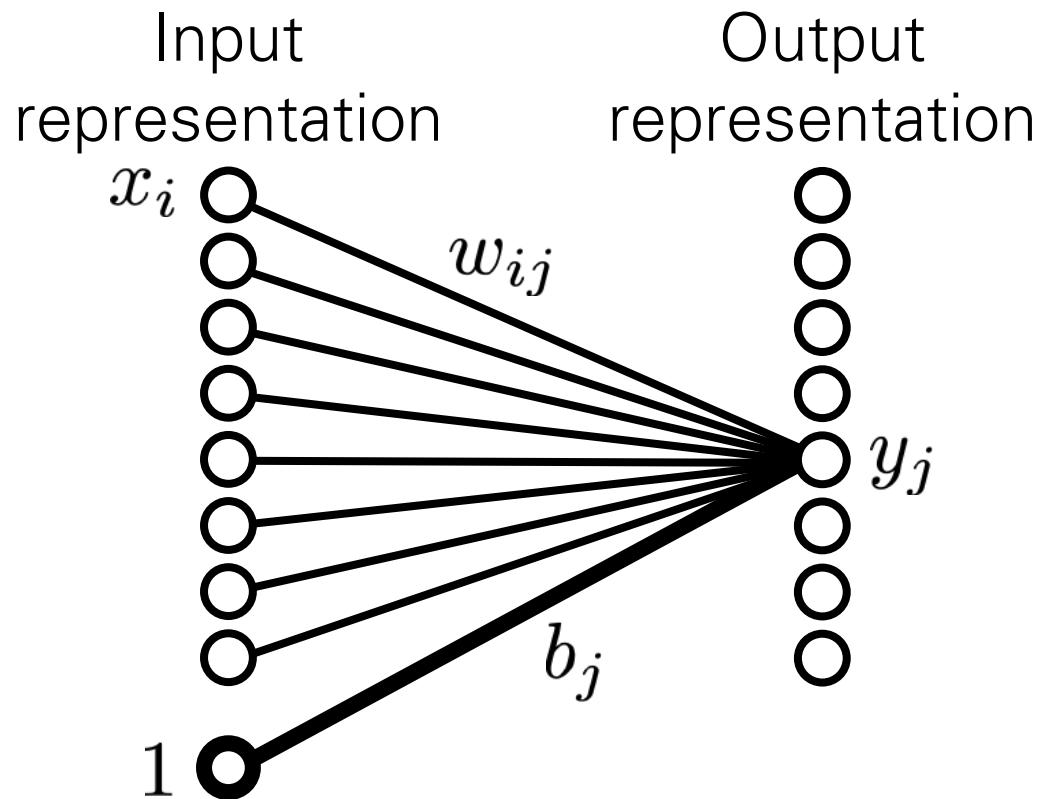
Linear layer



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

Computation in a neural net

Linear layer

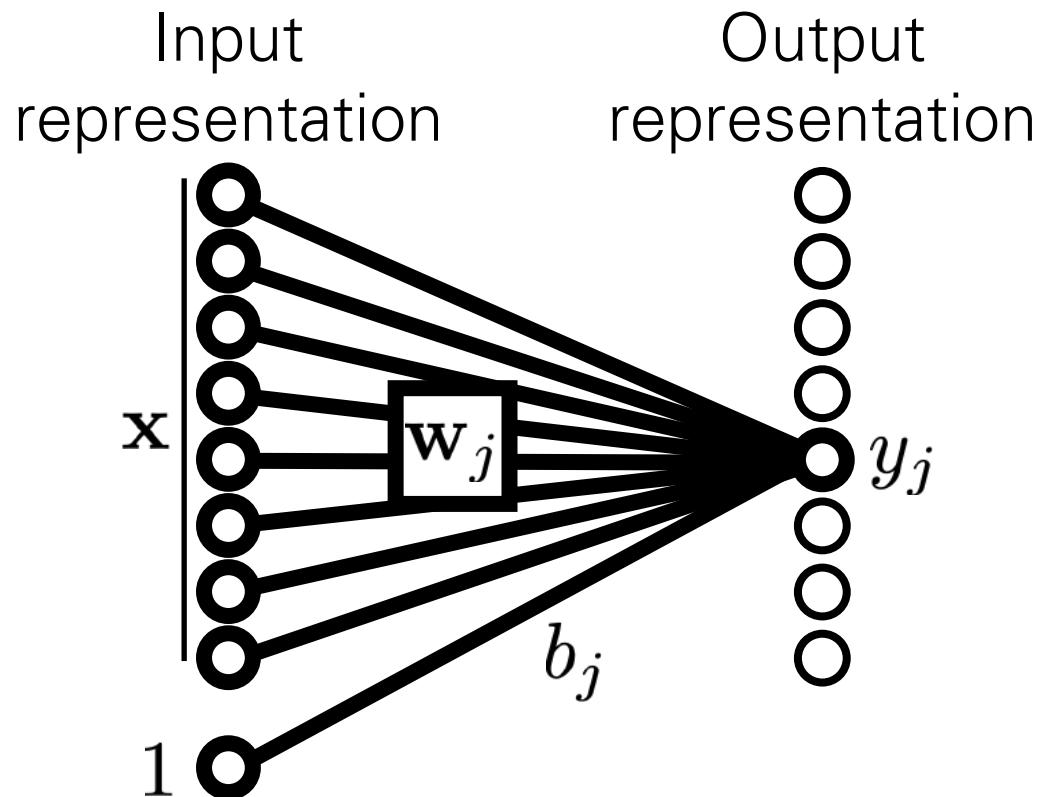


$$y_j = \sum_i w_{ij}x_i + b_j$$

weights
bias

Computation in a neural net

Linear layer



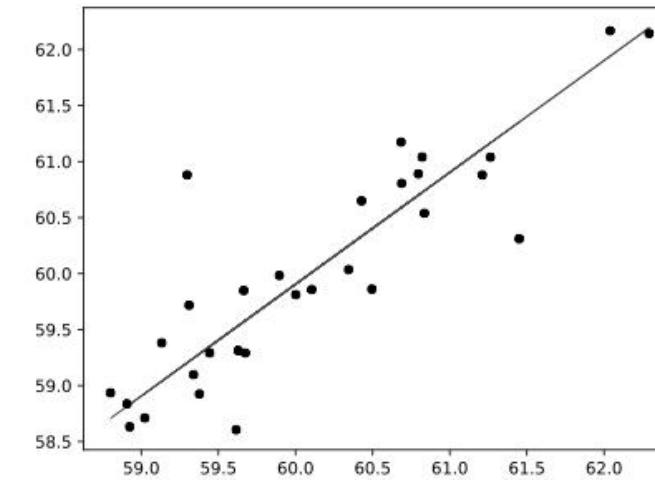
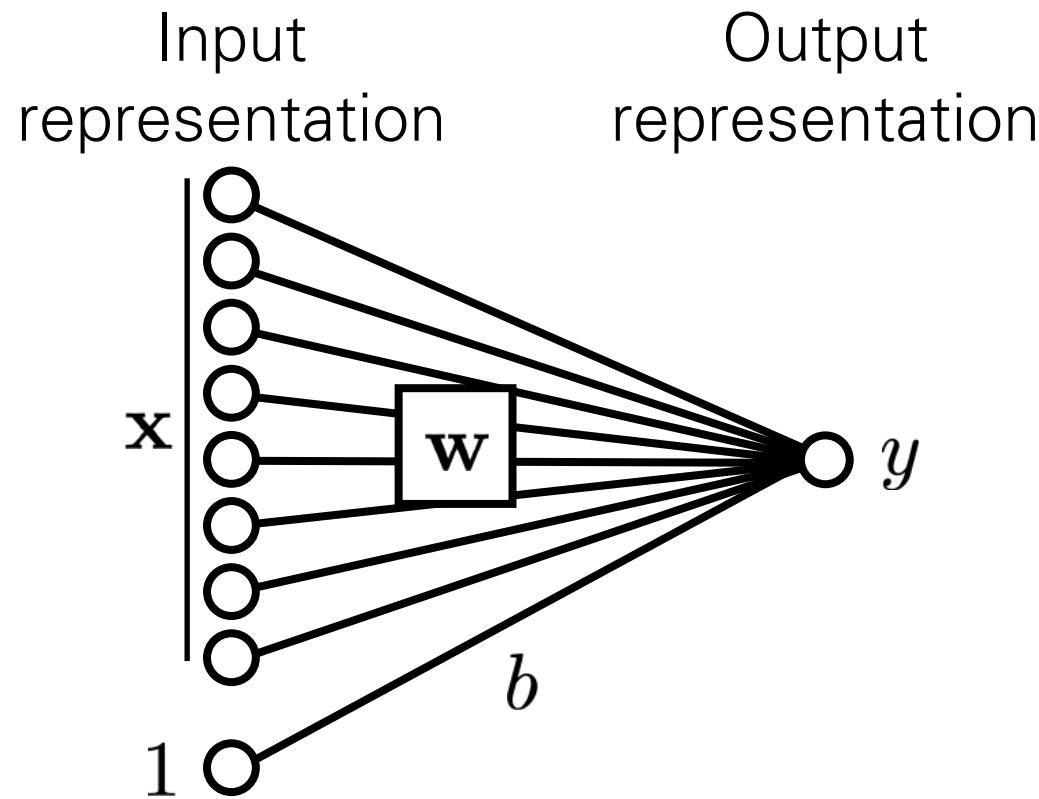
$$y_j = \mathbf{x}^T \mathbf{w}_j + b_j$$

weights
bias
parameters of the model

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

Example: linear regression with a neural net

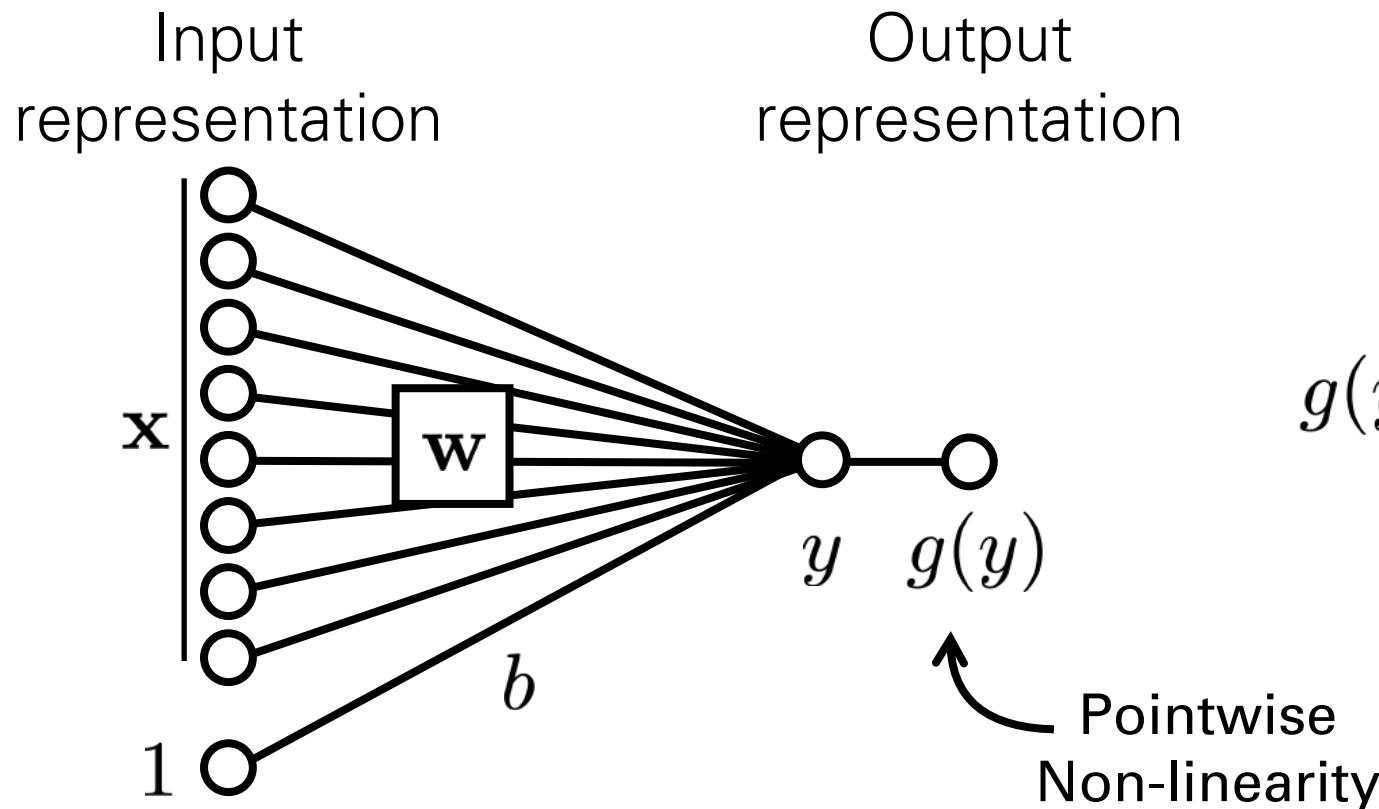
Linear layer



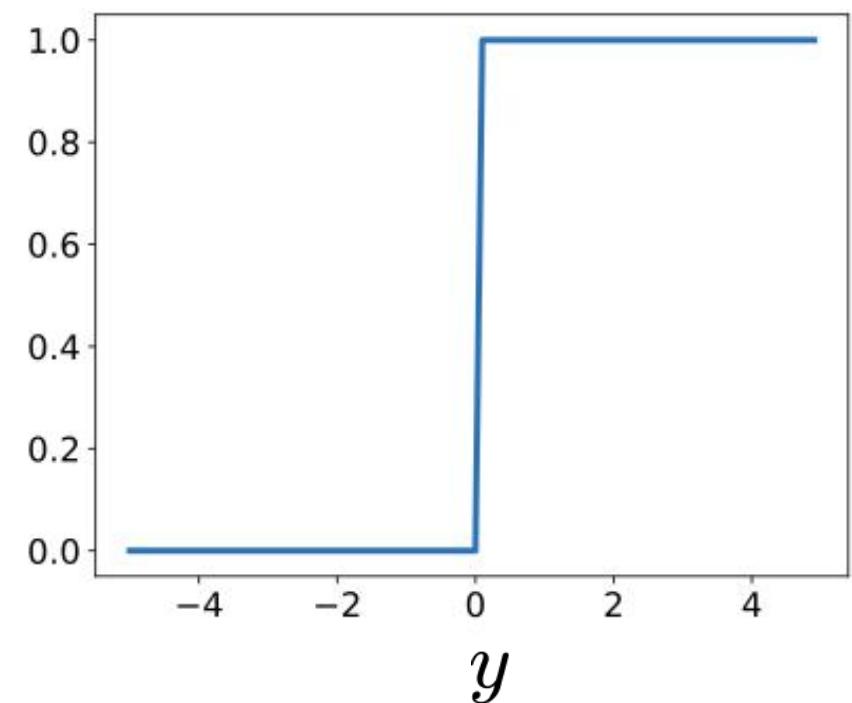
$$f_{\mathbf{w}, b}(\mathbf{x}) = \mathbf{x}^T \mathbf{w} + b$$

Computation in a neural net

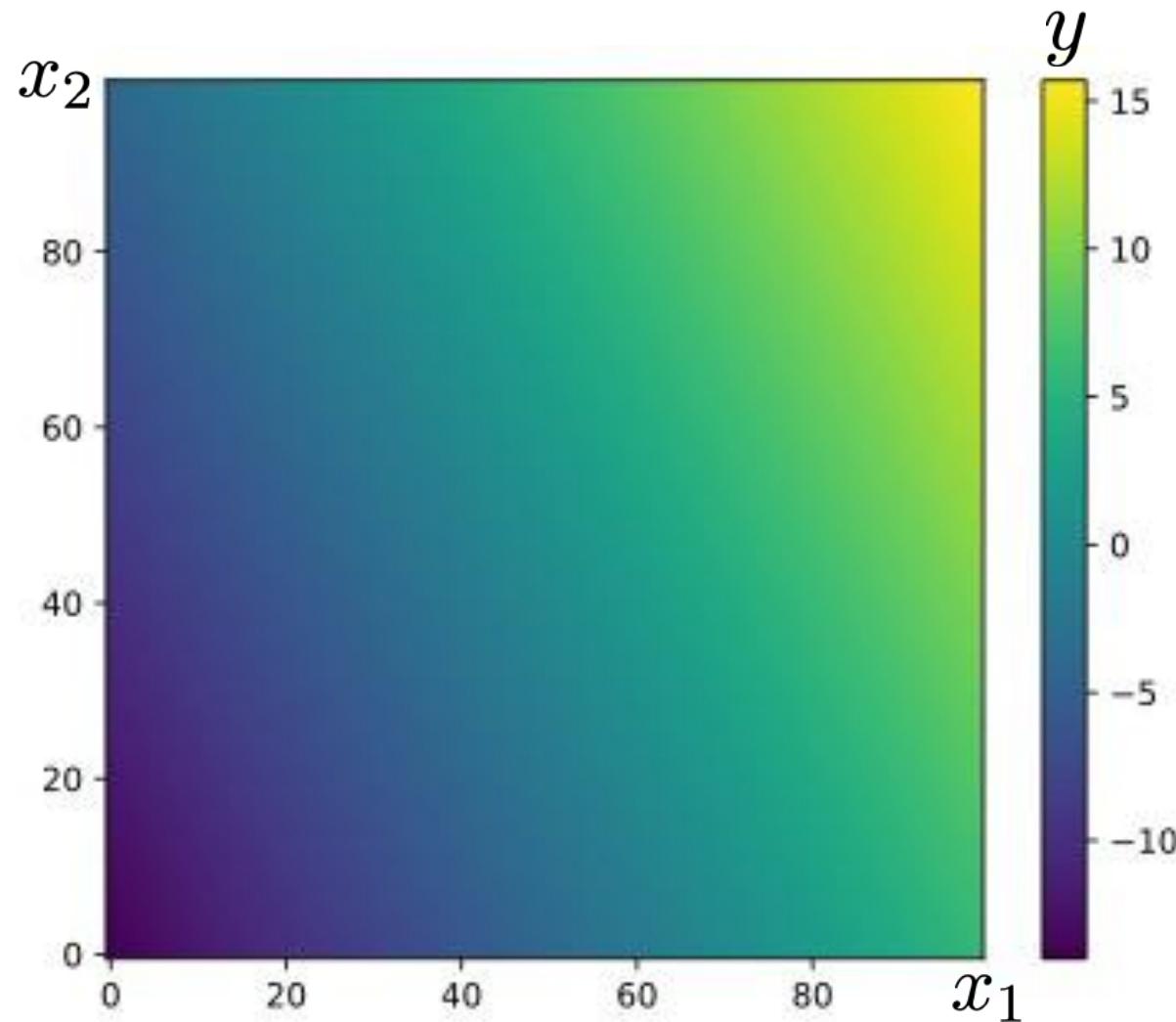
“Perceptron”



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

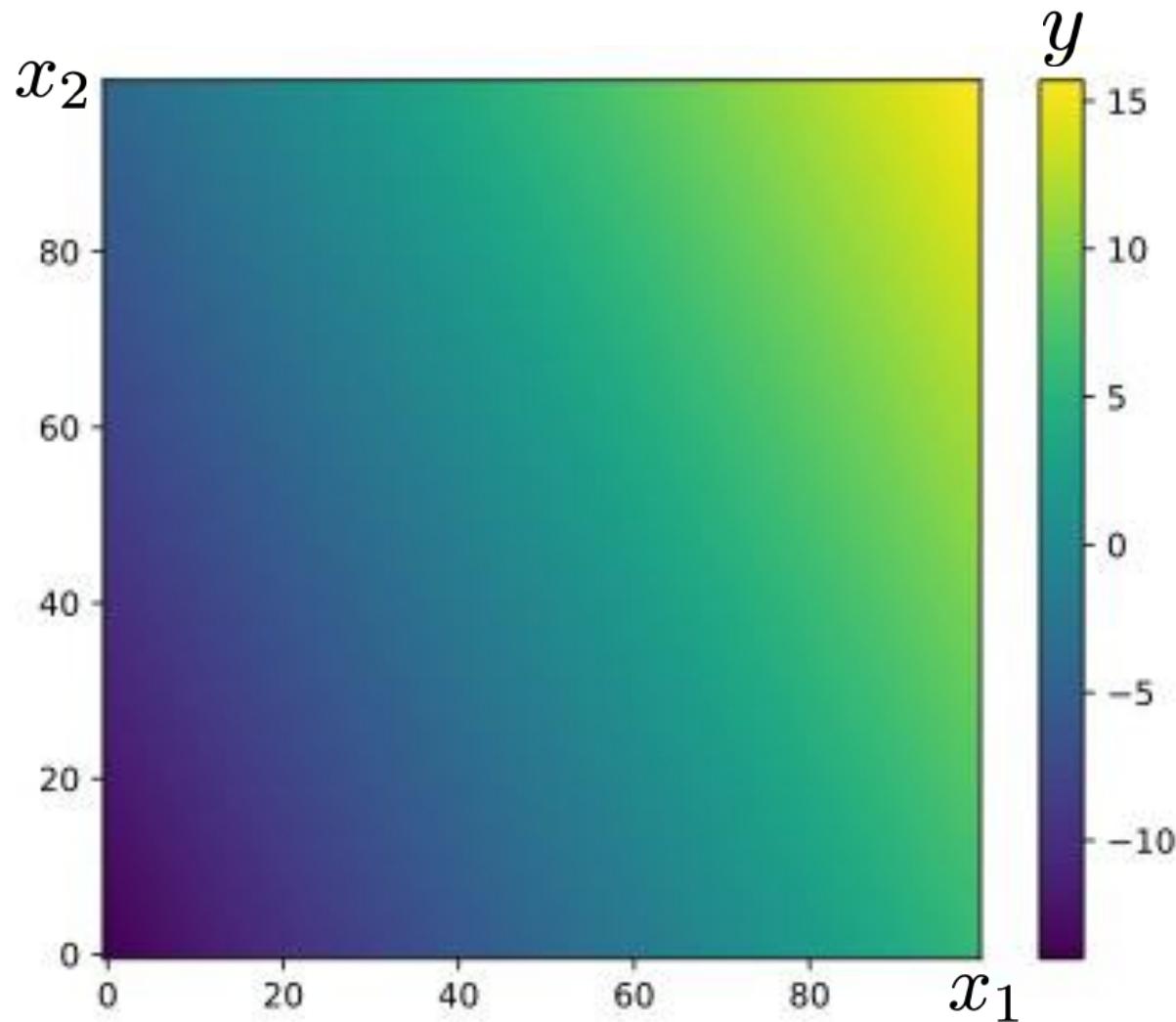


Example: linear classification with a perceptron



$$y = \mathbf{x}^T \mathbf{w} + b$$

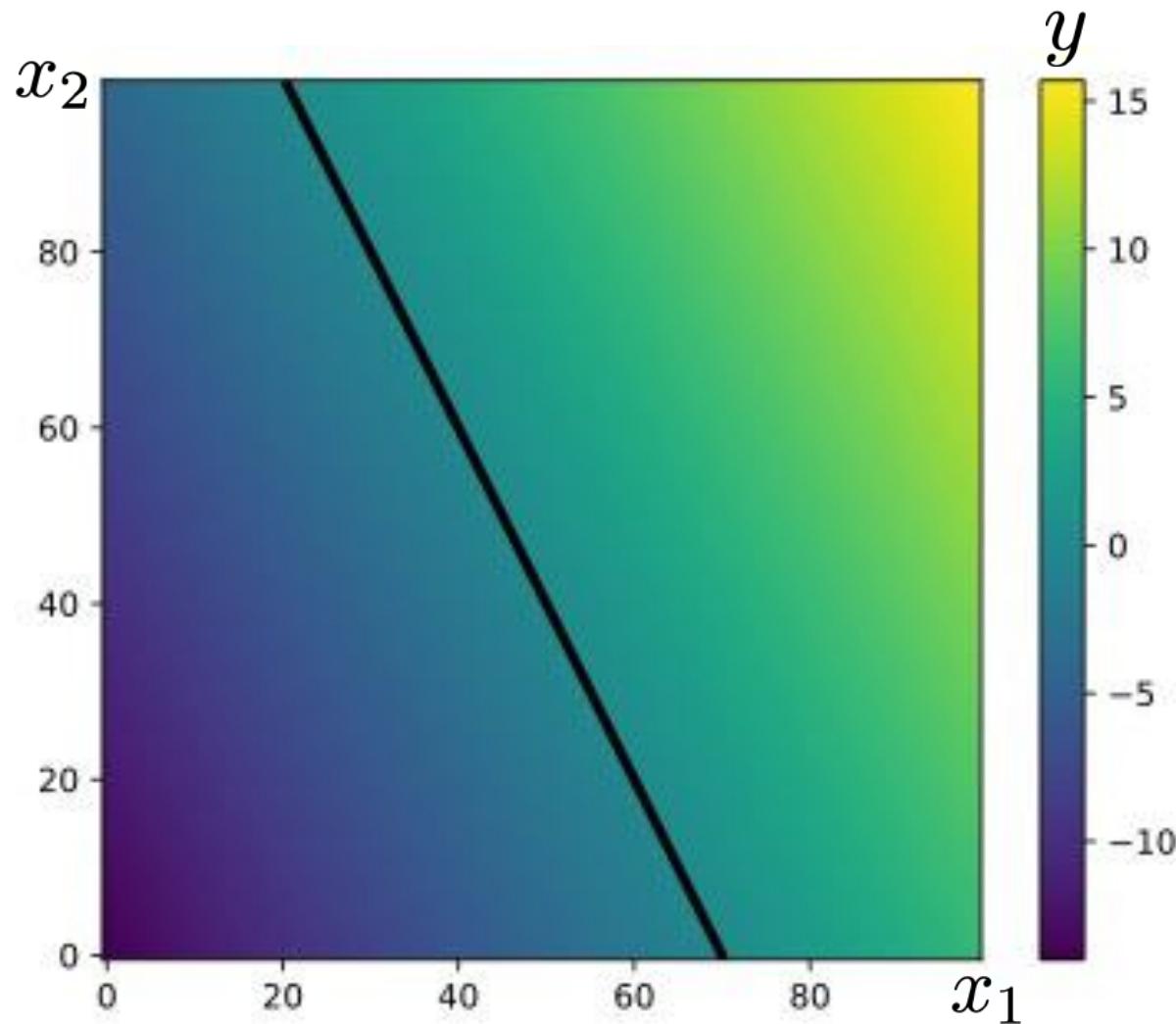
Example: linear classification with a perceptron



$$y = \mathbf{x}^T \mathbf{w} + b$$

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

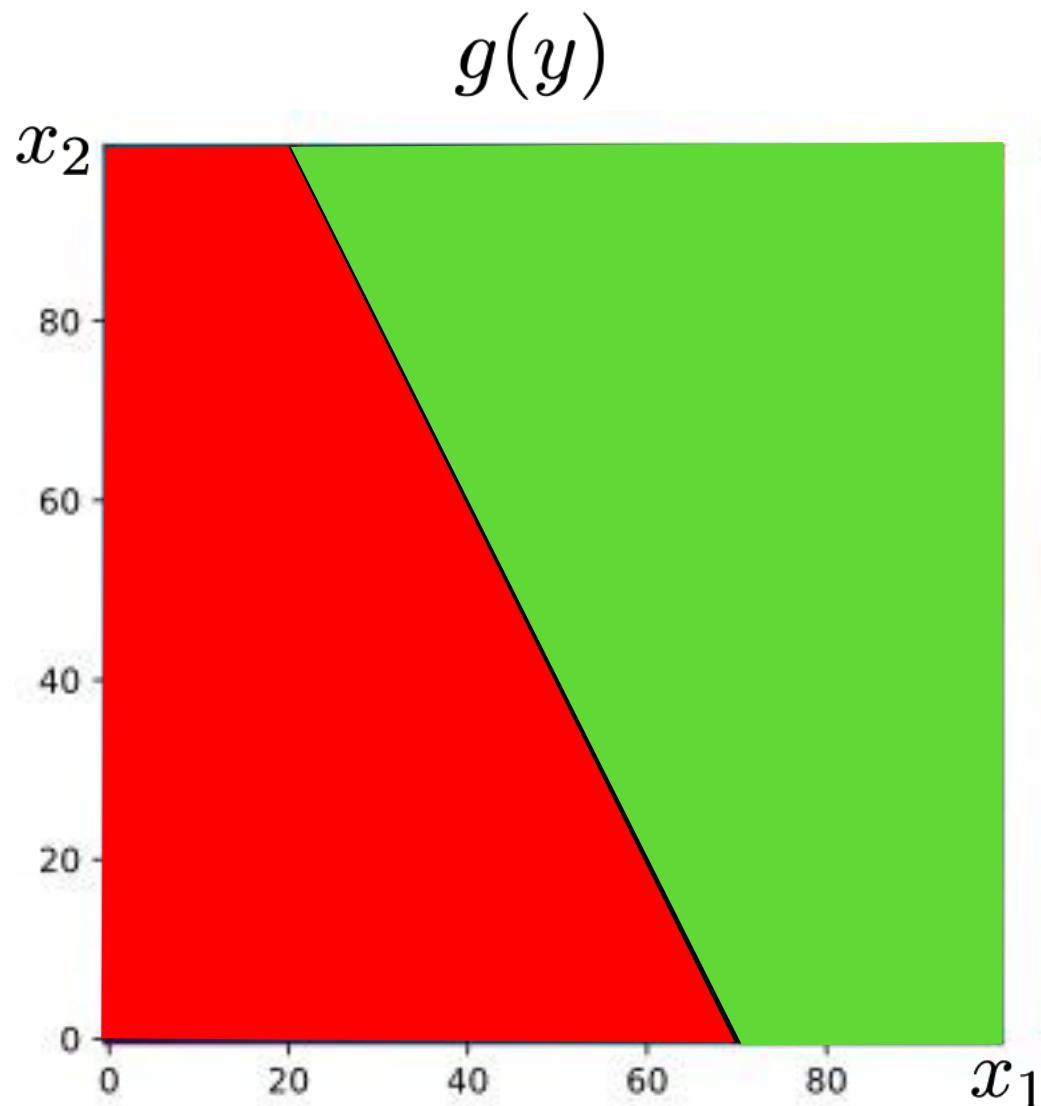
Example: linear classification with a perceptron



$$y = \mathbf{x}^T \mathbf{w} + b$$

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

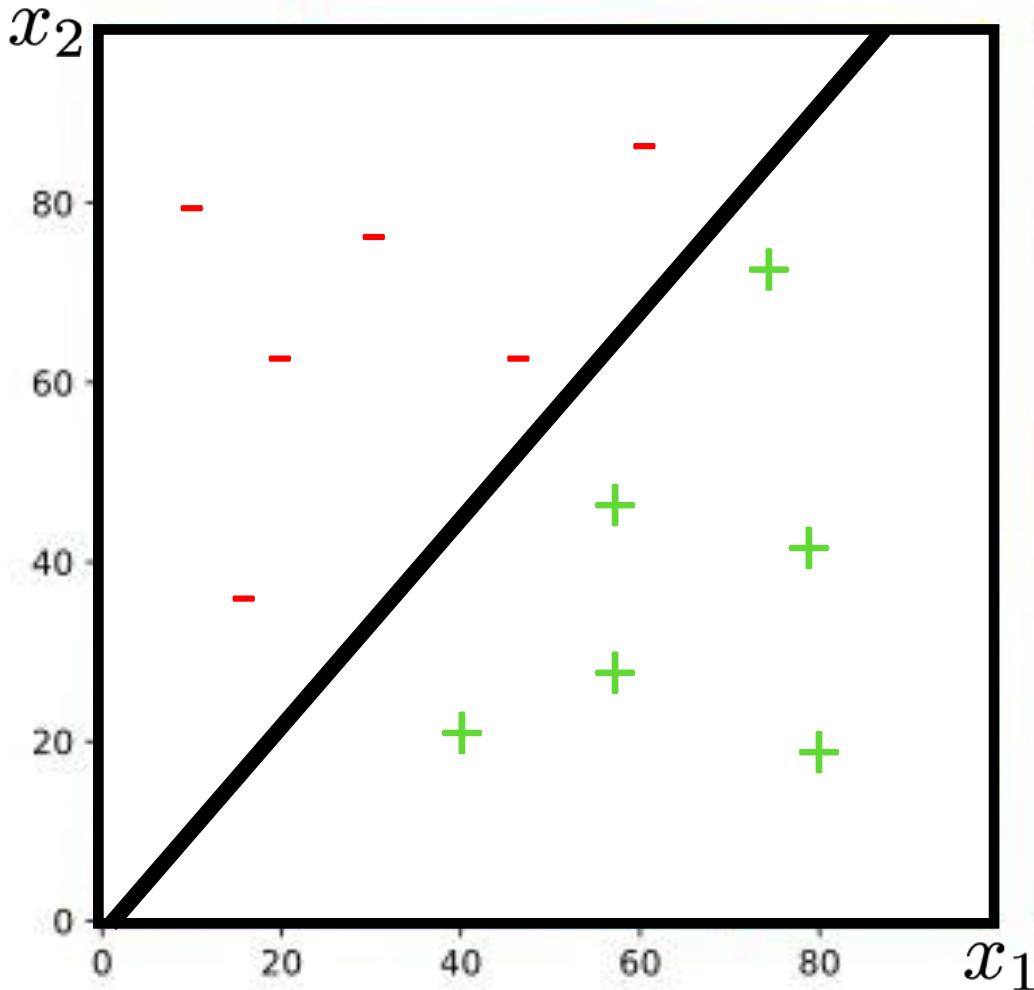
Example: linear classification with a perceptron



$$y = \mathbf{x}^T \mathbf{w} + b$$

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

Example: linear classification with a perceptron

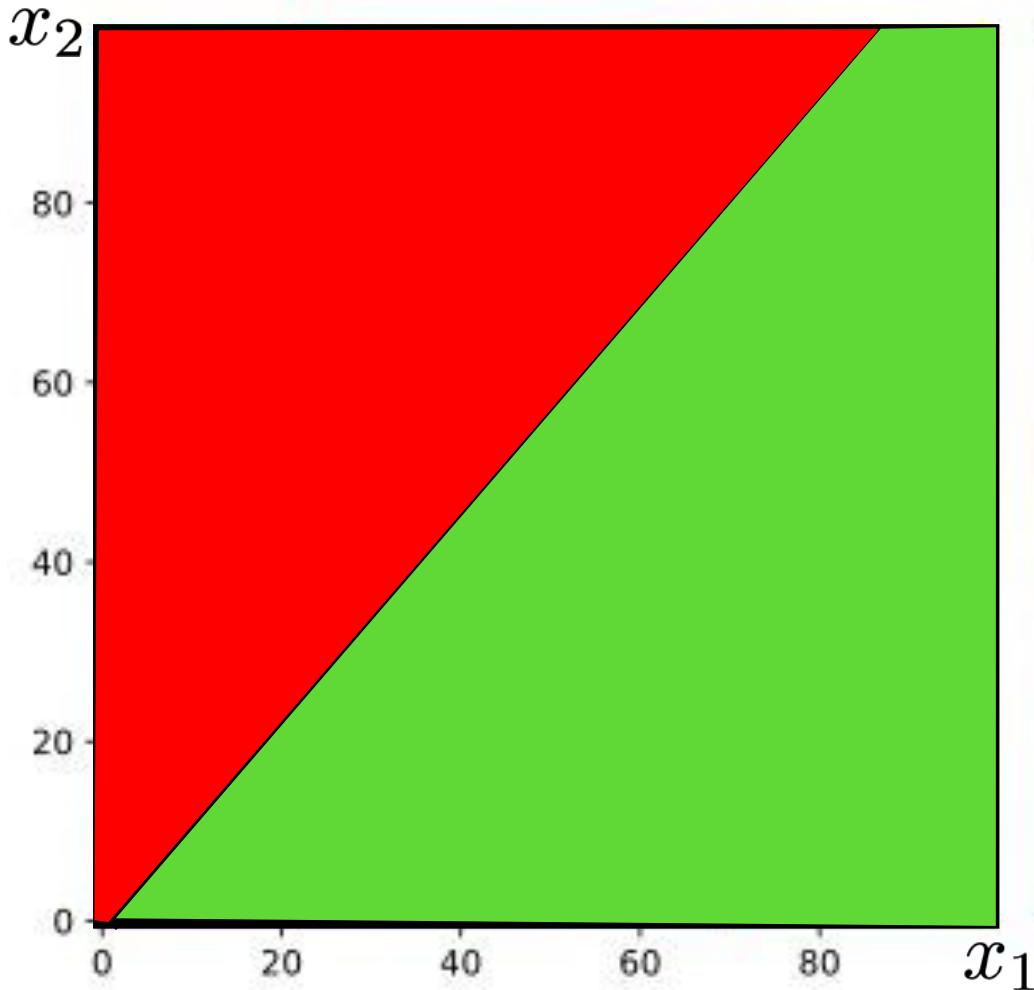


$$\hat{y} = \mathbf{x}^T \mathbf{w} + b$$

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

$$\mathbf{w}^*, b^* = \arg \min_{\mathbf{w}, b} \mathcal{L}(g(\hat{y}), y_i)$$

Example: linear classification with a perceptron

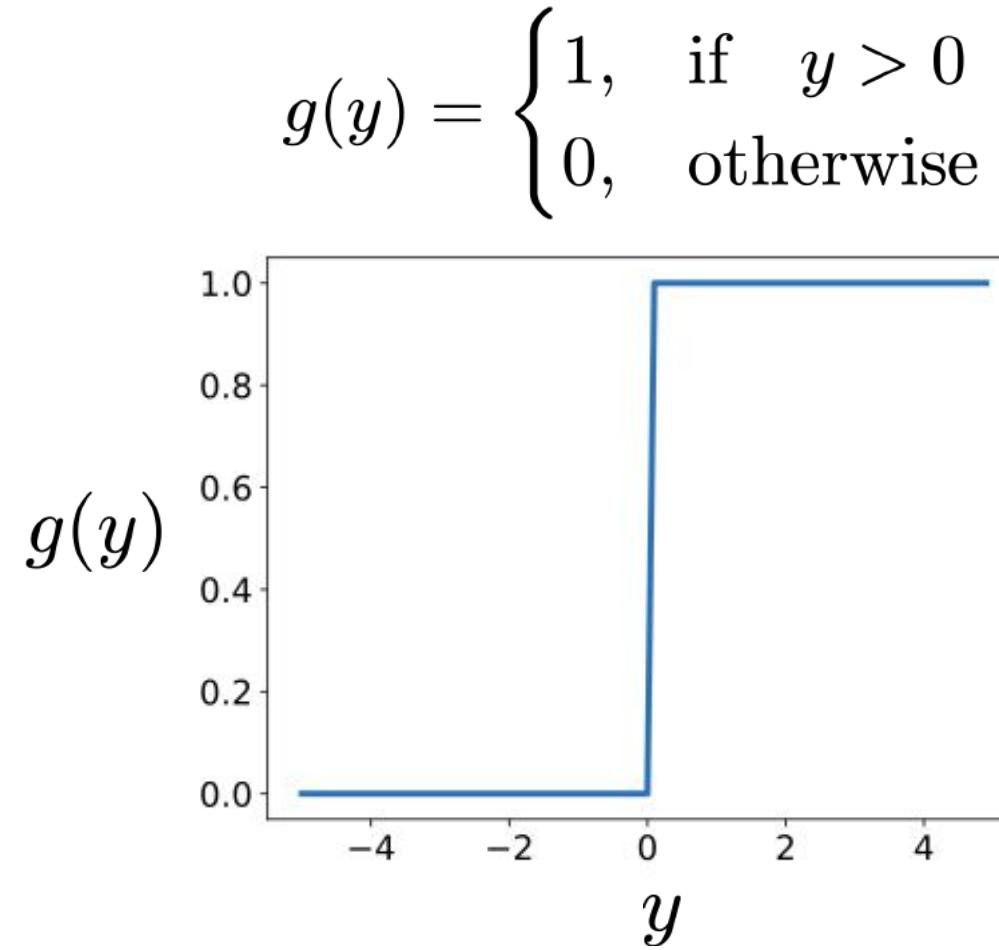
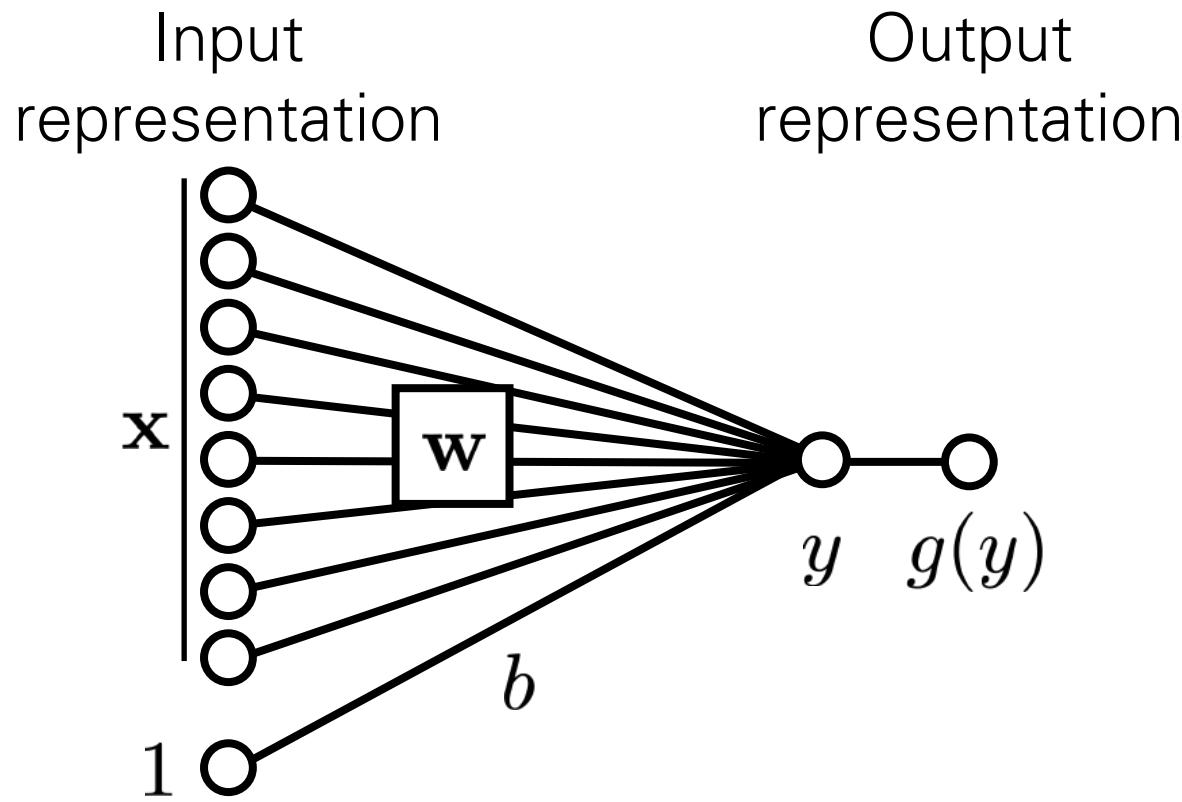


$$\hat{y} = \mathbf{x}^T \mathbf{w} + b$$

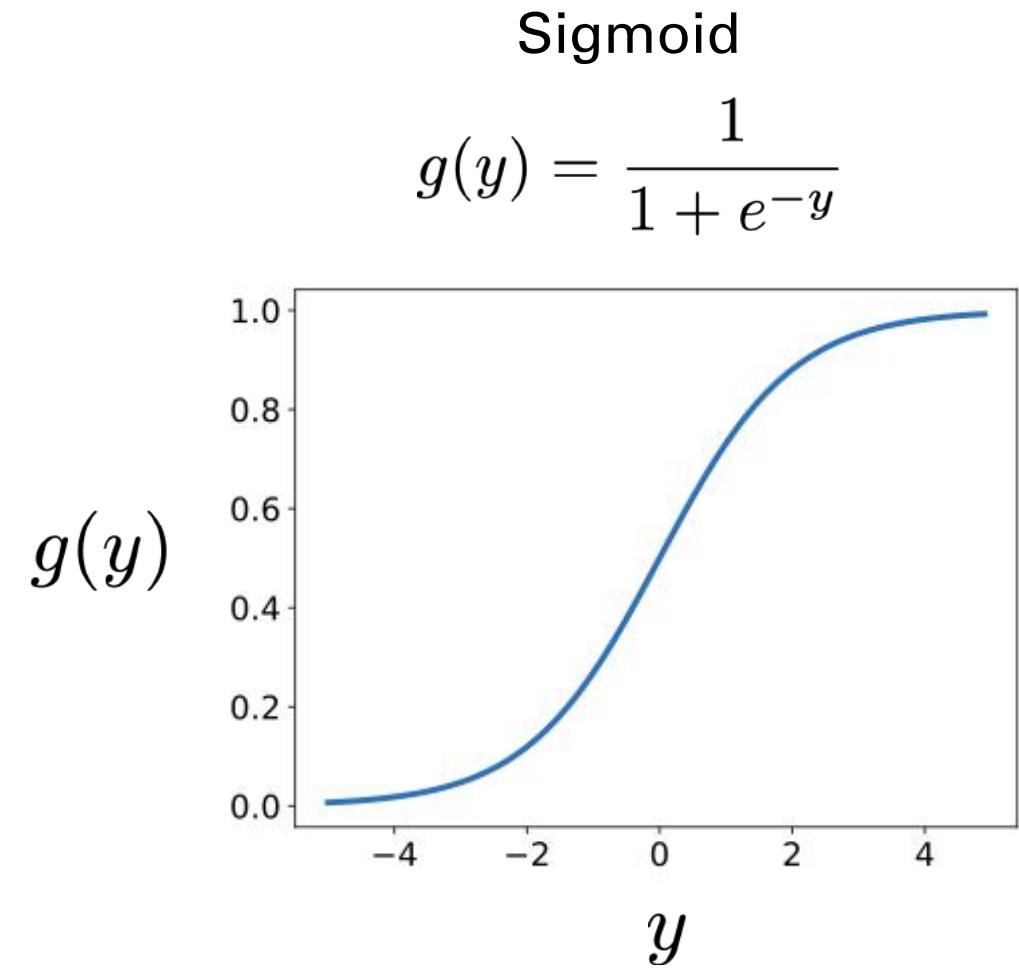
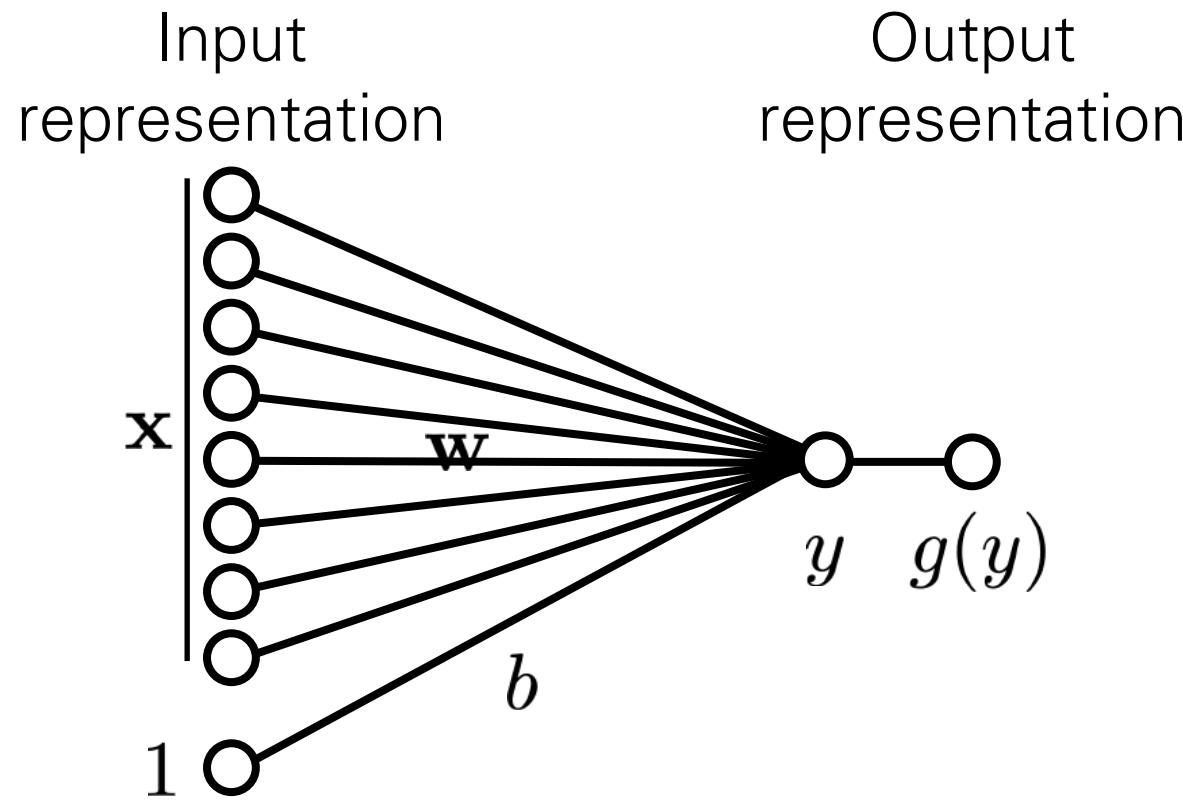
$$g(\hat{y}) = \begin{cases} 1, & \text{if } \hat{y} > 0 \\ 0, & \text{otherwise} \end{cases}$$

$$\mathbf{w}^*, b^* = \arg \min_{\mathbf{w}, b} \mathcal{L}(g(\hat{y}), y_i)$$

Computation in a neural net



Computation in a neural net – nonlinearity

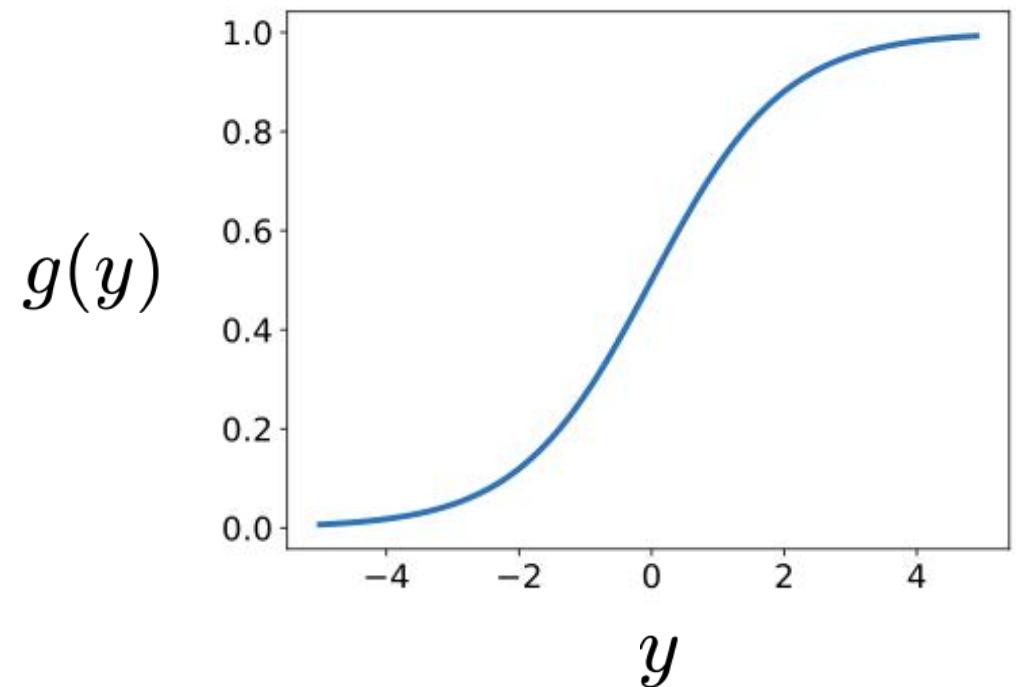


Computation in a neural net – nonlinearity

- Interpretation as firing rate of neuron
- Bounded between [0,1]
- Saturation for large +/- inputs
- Gradients go to zero
- Outputs centered at 0.5
(poor conditioning)
- Not used in practice

Sigmoid

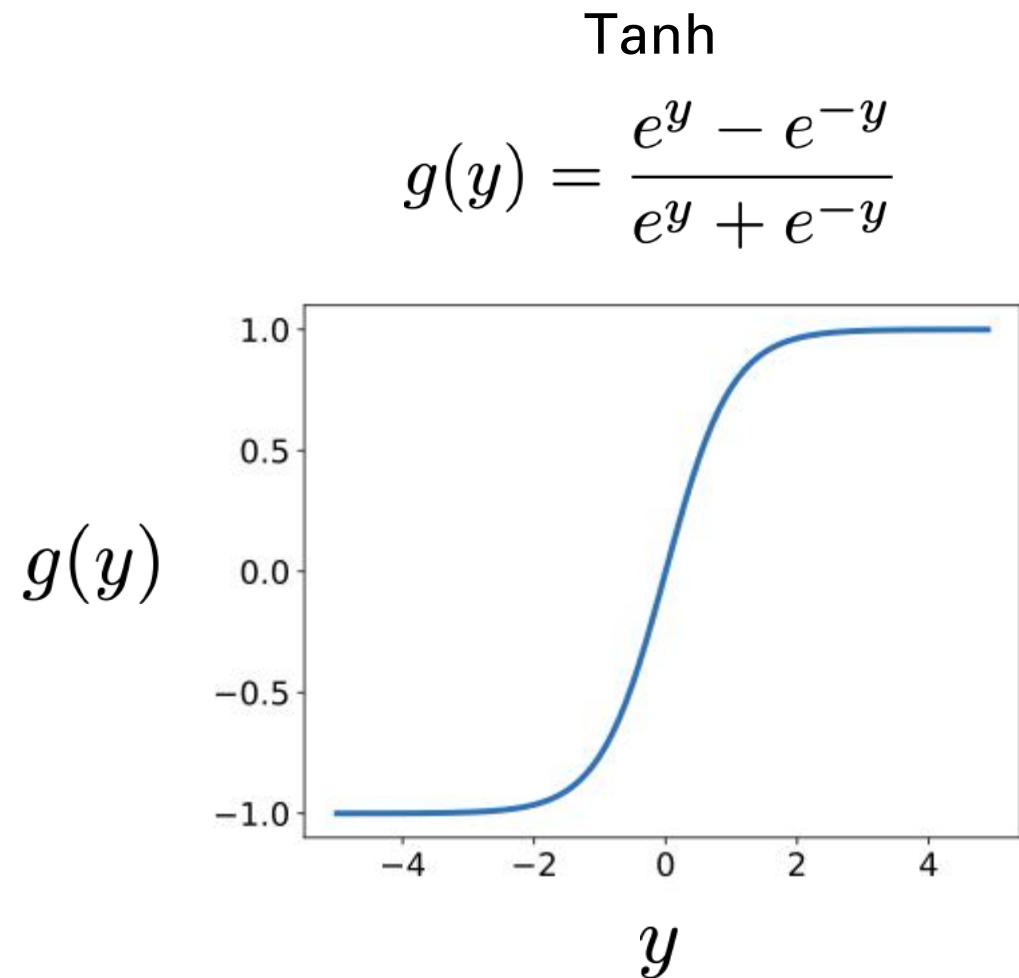
$$g(y) = \frac{1}{1 + e^{-y}}$$



Computation in a neural net – nonlinearity

- Bounded between $[-1, +1]$
- Saturation for large +/- inputs
- Gradients go to zero
- Outputs centered at 0
- Preferable to sigmoid

$$\tanh(x) = 2 \text{ sigmoid}(2x) - 1$$



Computation in a neural net – nonlinearity

- Unbounded output (on positive side)

Rectified linear unit (ReLU)

- Efficient to implement: $\frac{\partial g}{\partial y} = \begin{cases} 0, & \text{if } y < 0 \\ 1, & \text{if } y \geq 0 \end{cases}$

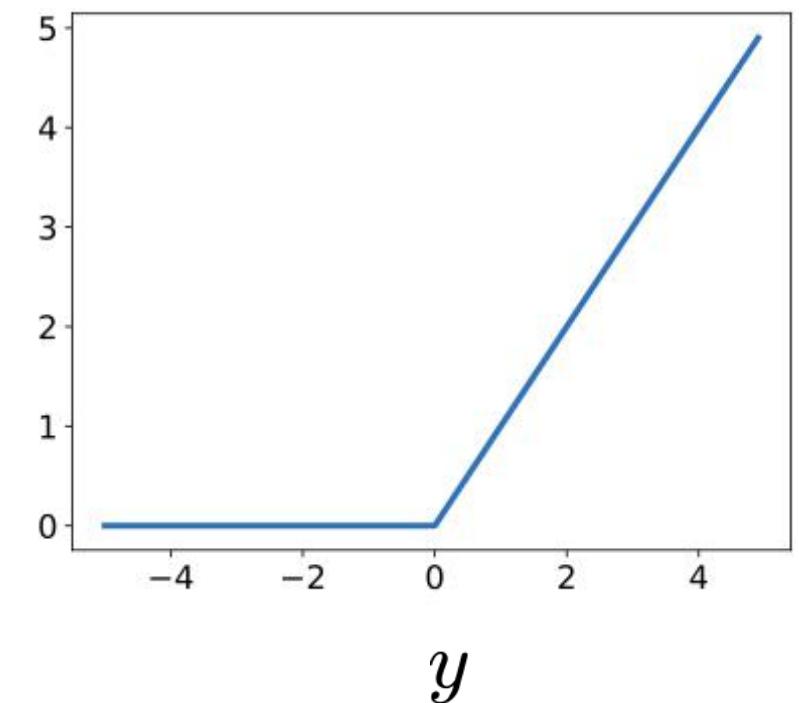
$$g(y) = \max(0, y)$$

- Also seems to help convergence
(see 6x speedup vs tanh in [Krizhevsky et al.])

- Drawback: if strongly in negative region,
unit is dead forever (no gradient).

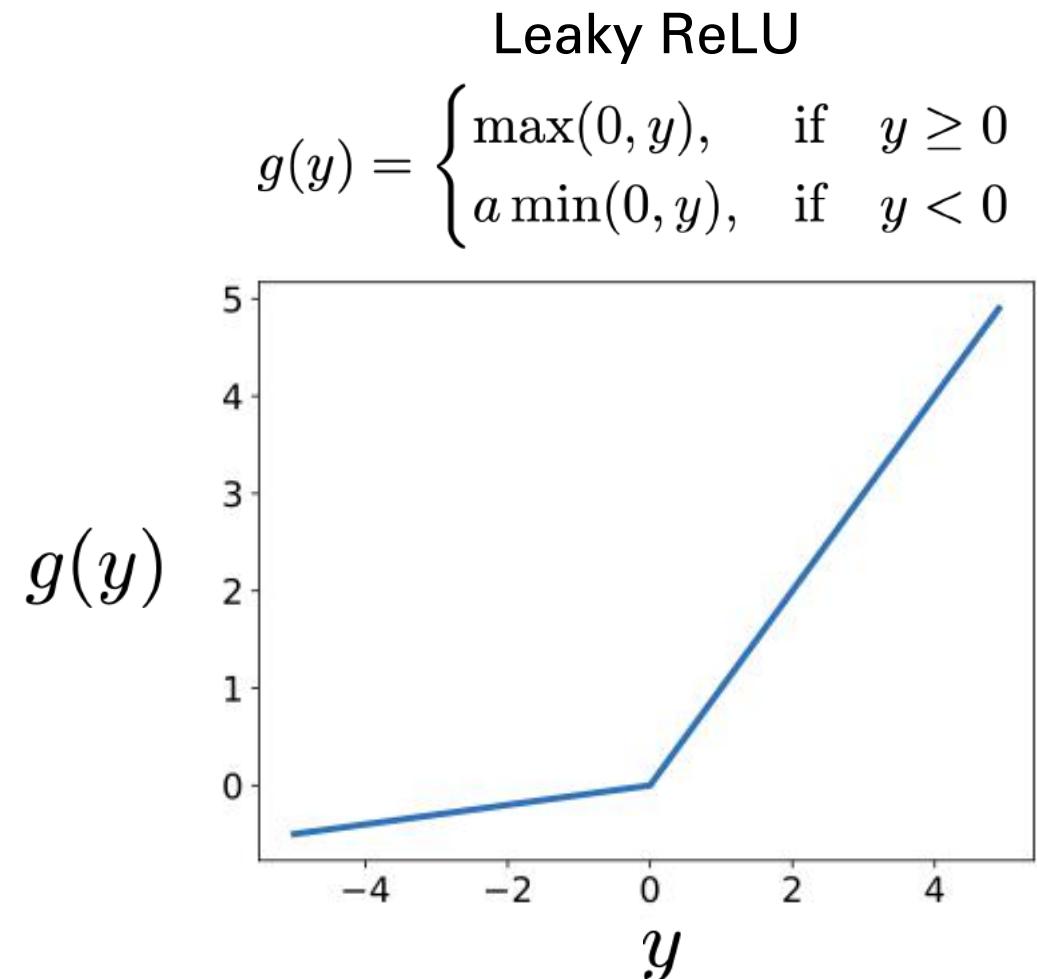
- Default choice: widely used in current models.

$$g(y)$$

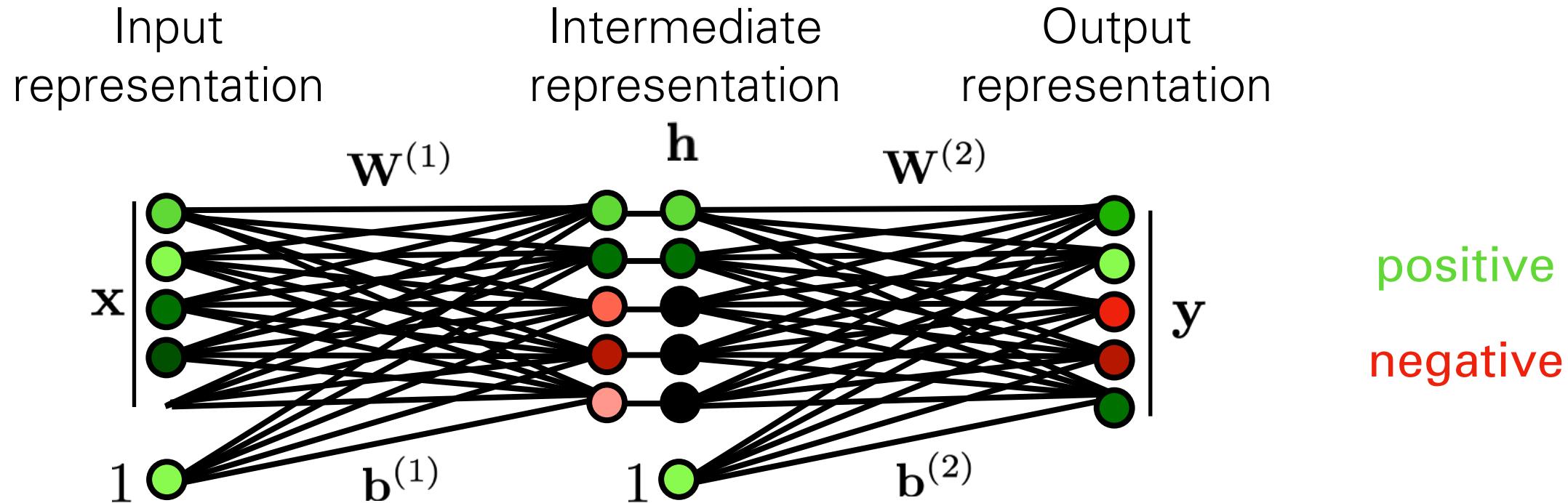


Computation in a neural net – nonlinearity

- where α is small (e.g. 0.02)
- Efficient to implement: $\frac{\partial g}{\partial y} = \begin{cases} -a, & \text{if } y < 0 \\ 1, & \text{if } y \geq 0 \end{cases}$
- Also known as probabilistic ReLU (PReLU)
- Has non-zero gradients everywhere (unlike ReLU)
- α can also be learned (see Kaiming He et al. 2015).



Stacking layers

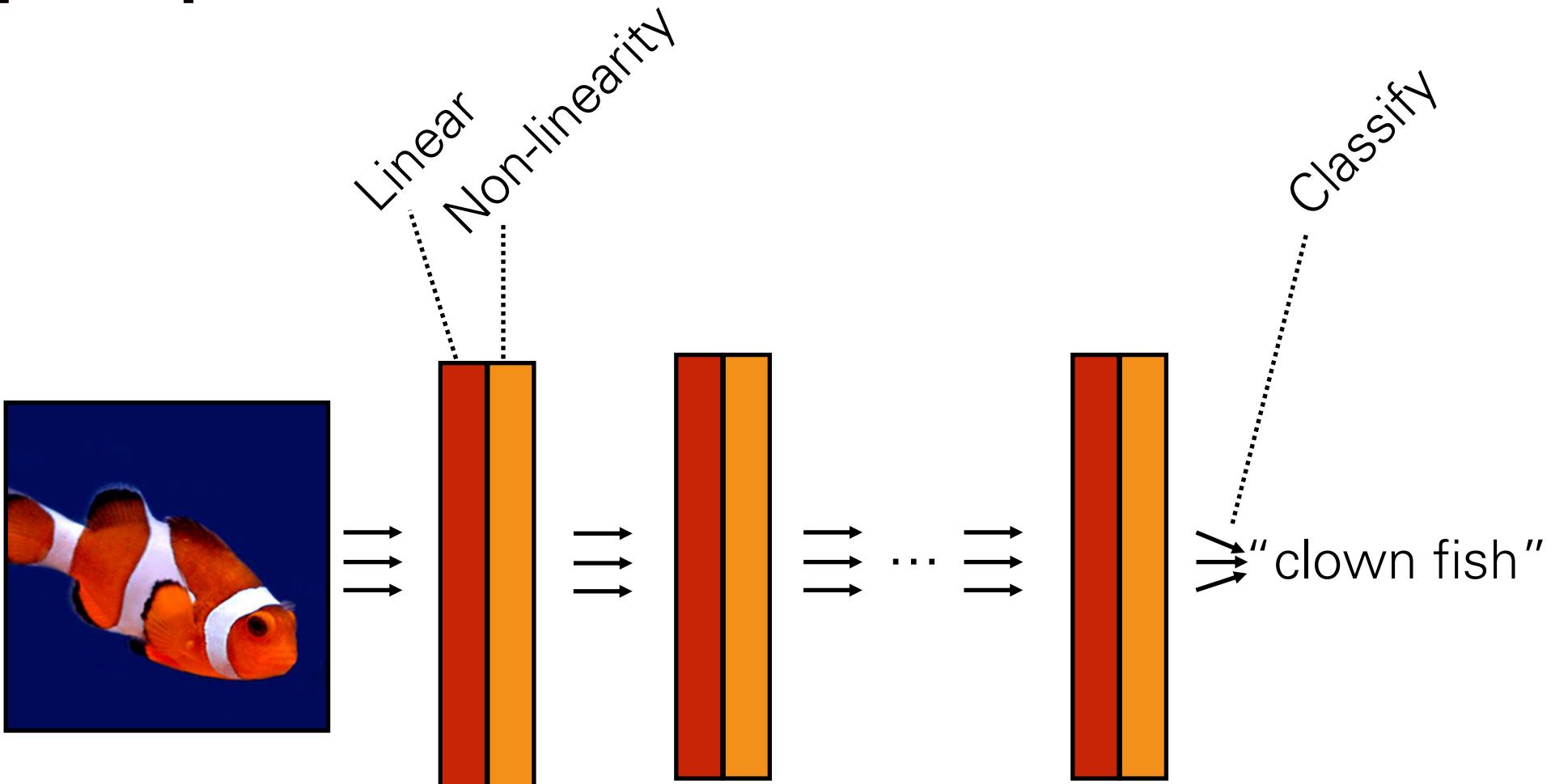


$$\theta = \{\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(L)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(L)}\}$$

Representational power

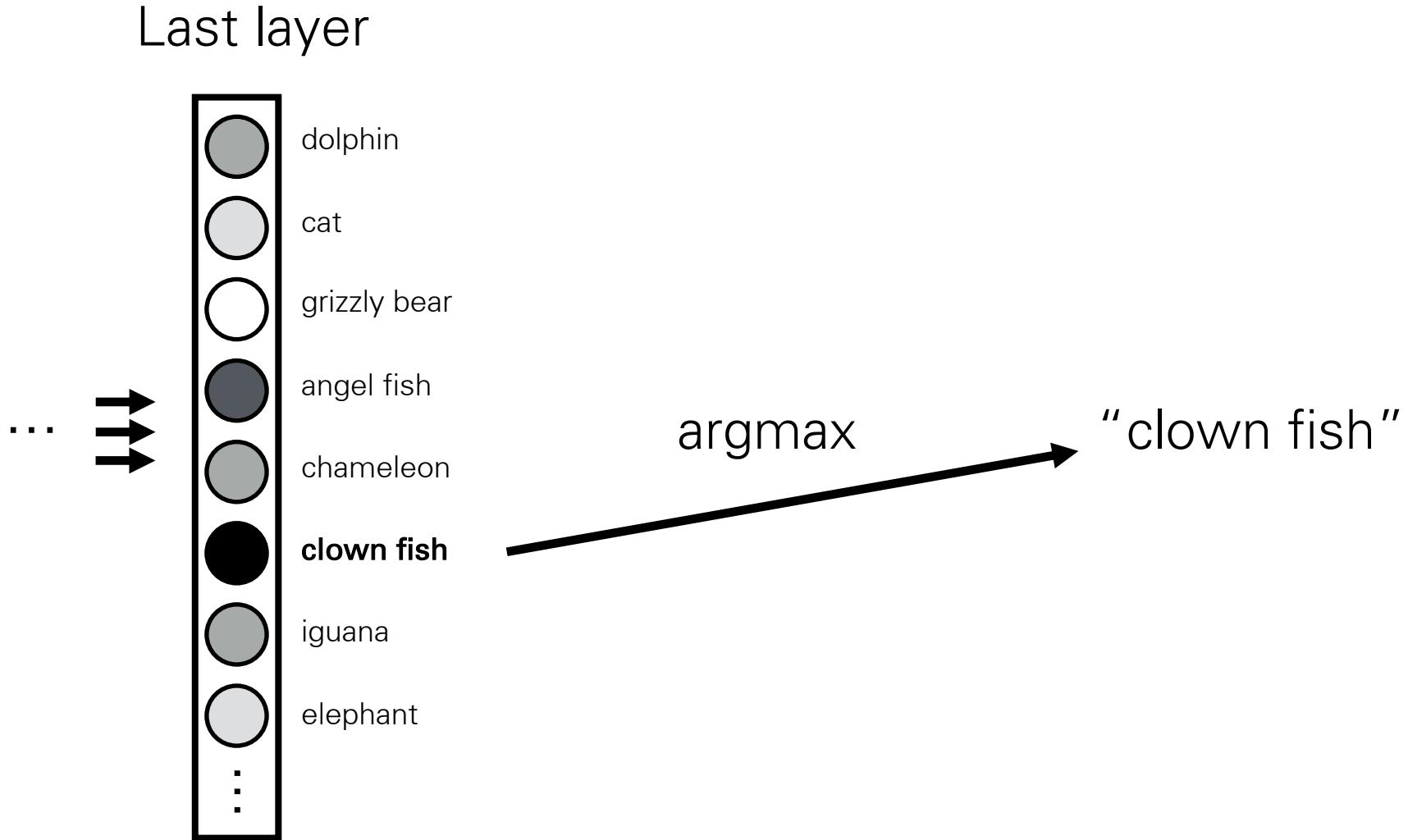
- 1 layer? Linear decision surface.
- 2+ layers? In theory, can represent any function.
Assuming non-trivial non-linearity.
 - Bengio 2009,
<http://www.iro.umontreal.ca/~bengioy/papers/ftml.pdf>
 - Bengio, Courville, Goodfellow book
<http://www.deeplearningbook.org/contents/mlp.html>
 - Simple proof by M. Nielsen
<http://neuralnetworksanddeeplearning.com/chap4.html>
 - D. Mackay book
<http://www.inference.phy.cam.ac.uk/mackay/itprnn/ps/482.491.pdf>
- But issue is efficiency: very wide two layers vs narrow deep model? In practice, more layers helps.

Deep supervised nets

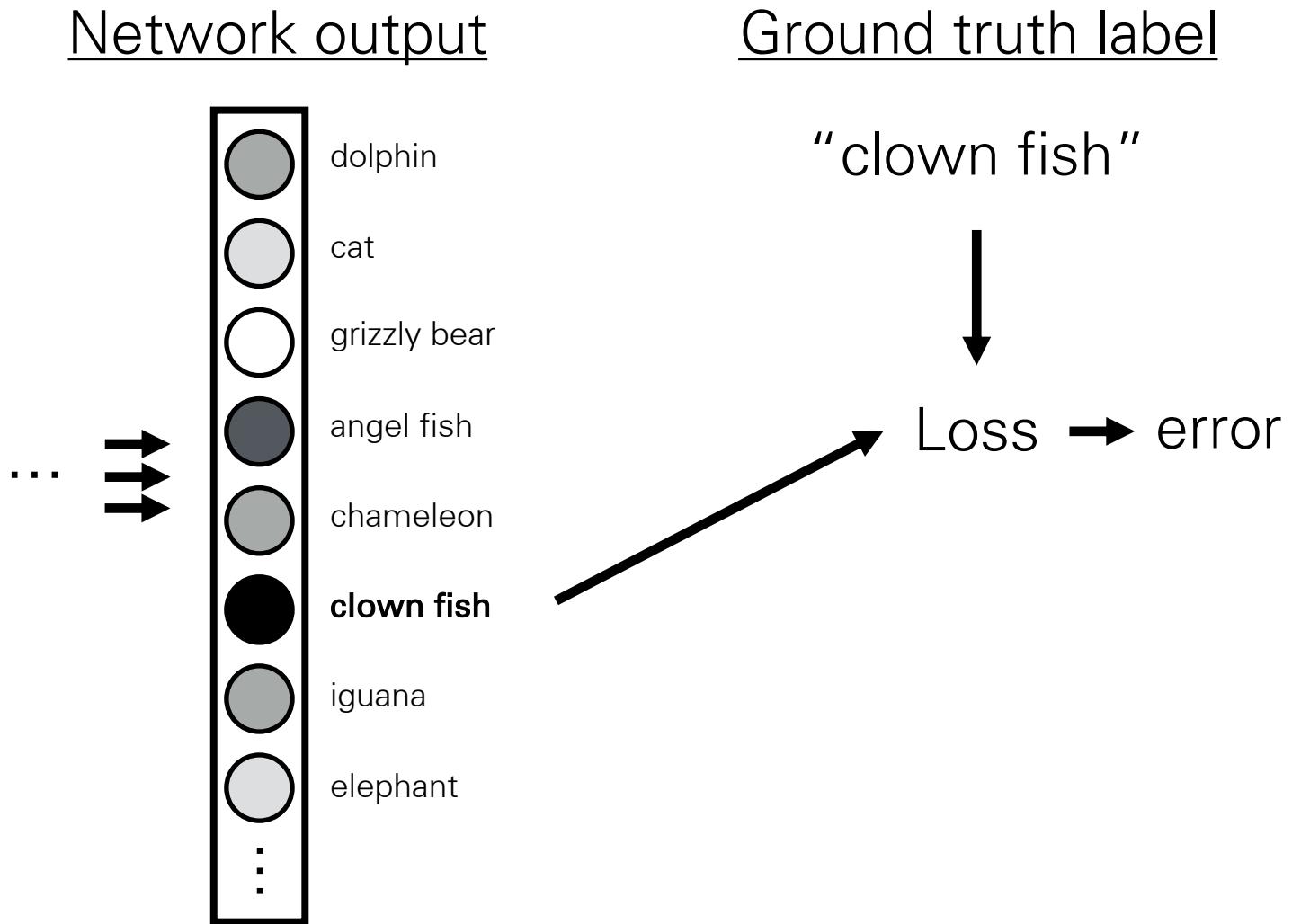


$$f(\mathbf{x}) = f_L(\dots(f_2(f_1(\mathbf{x}))))$$

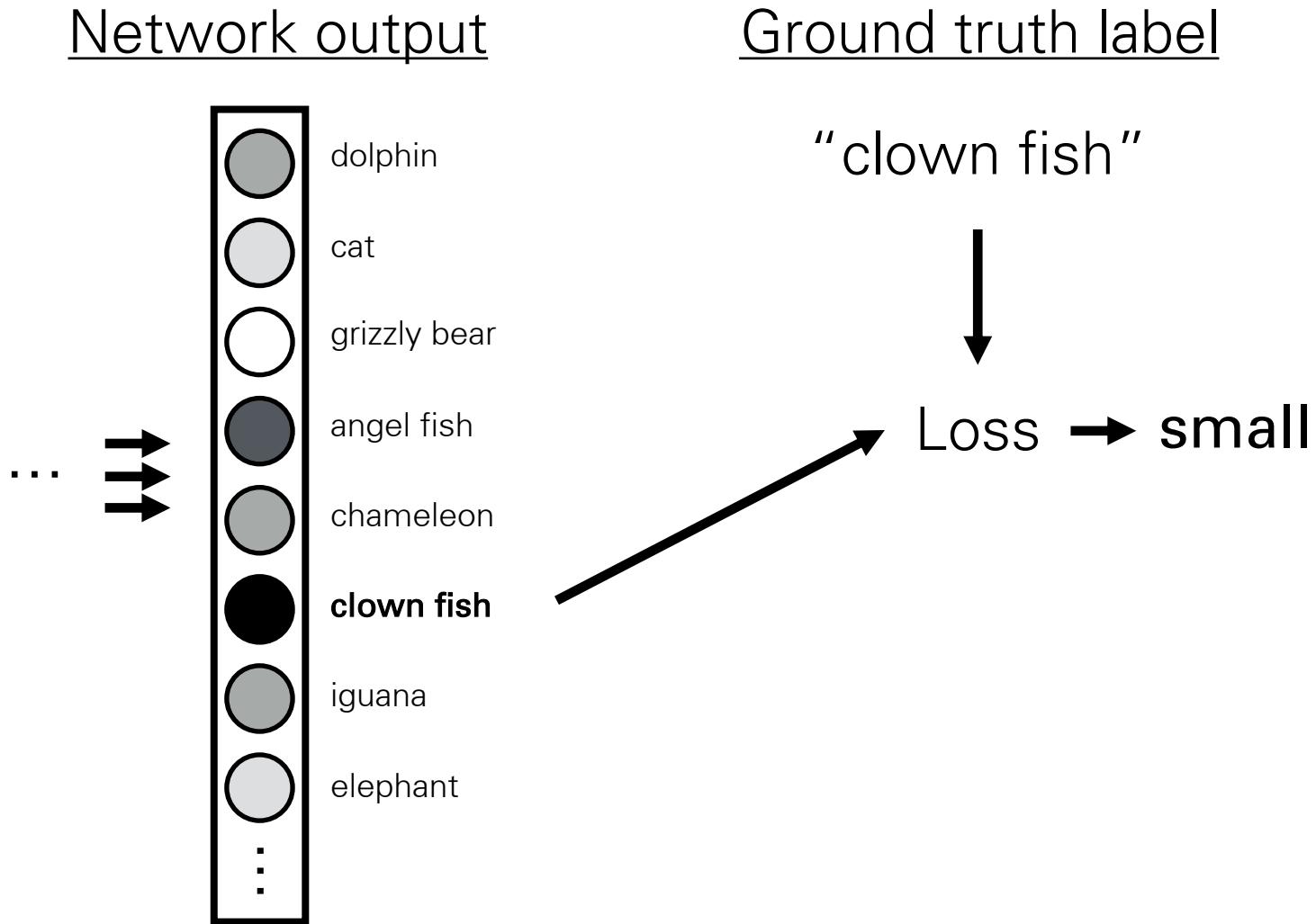
Classifier layer



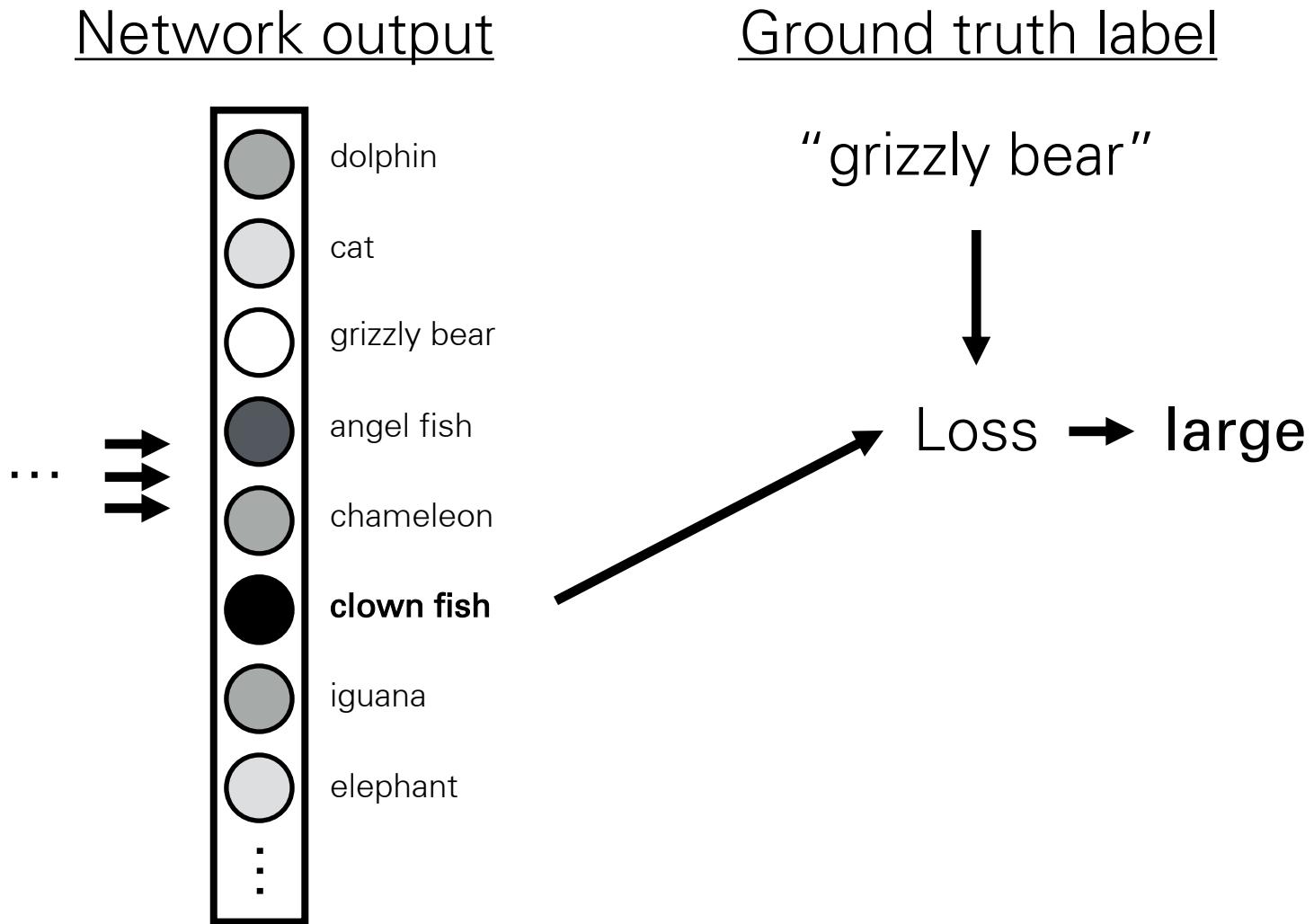
Loss function

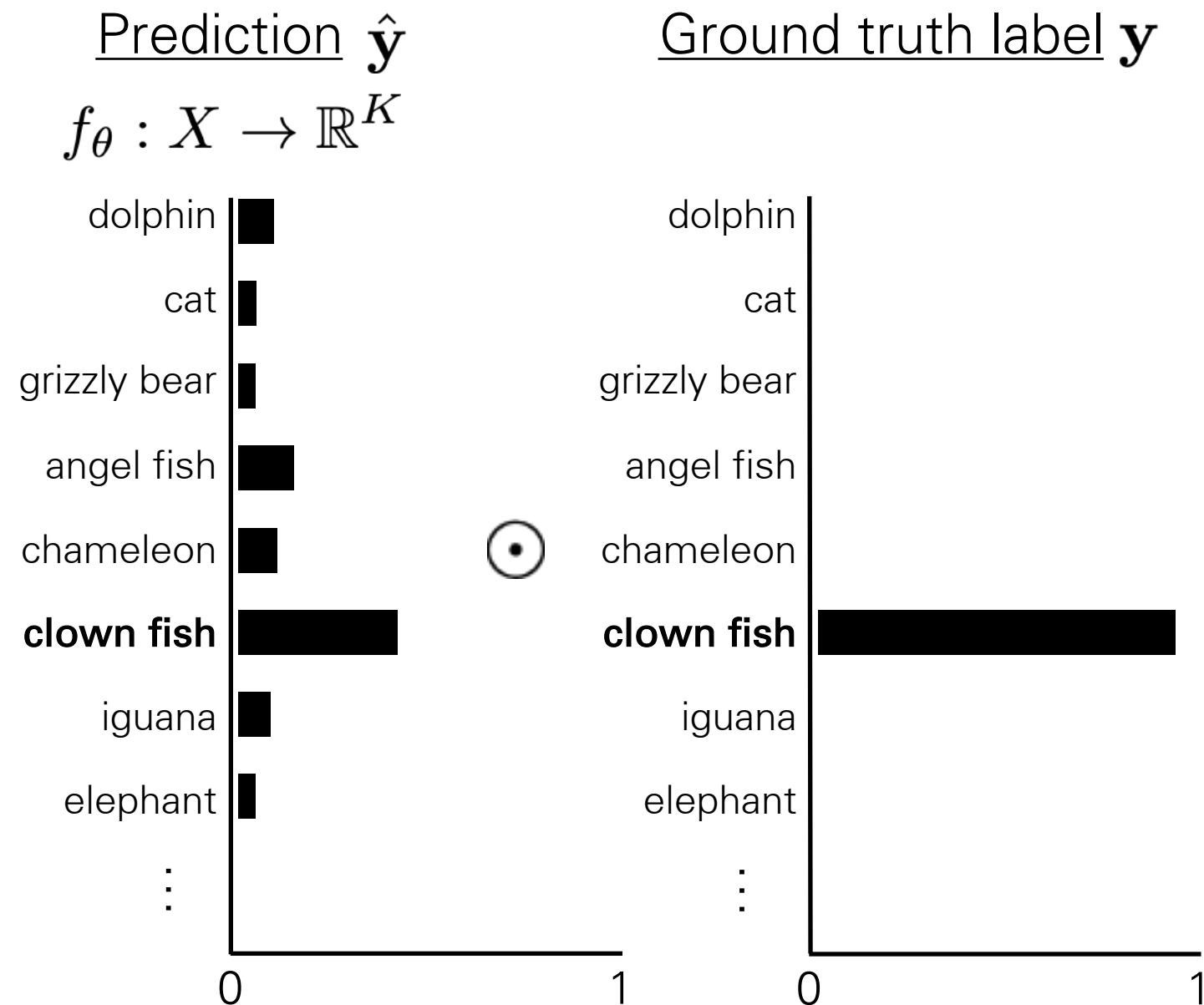
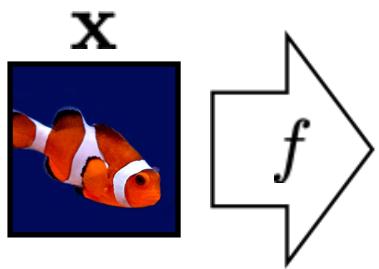


Loss function



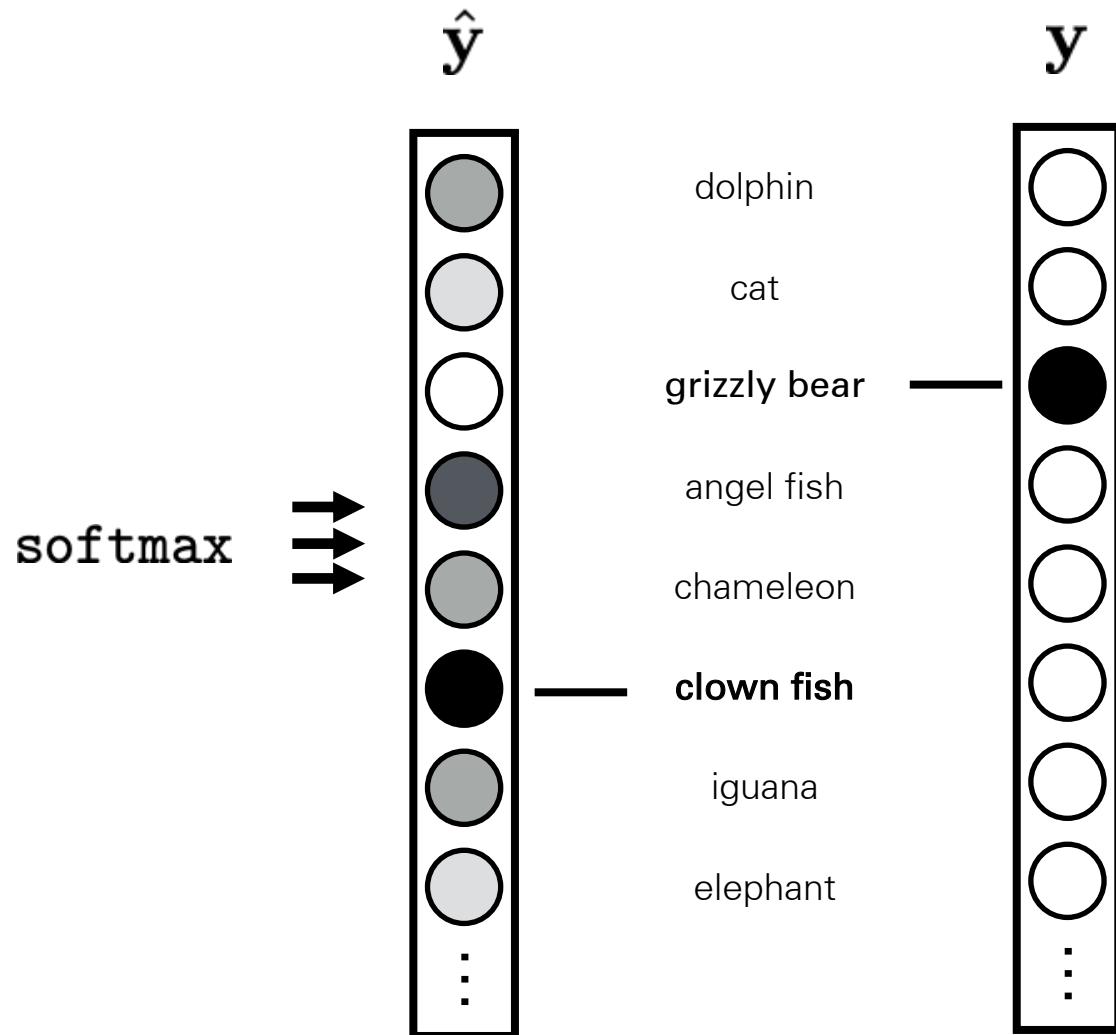
Loss function





Network output

Ground truth label



Probability of the observed
data under the model

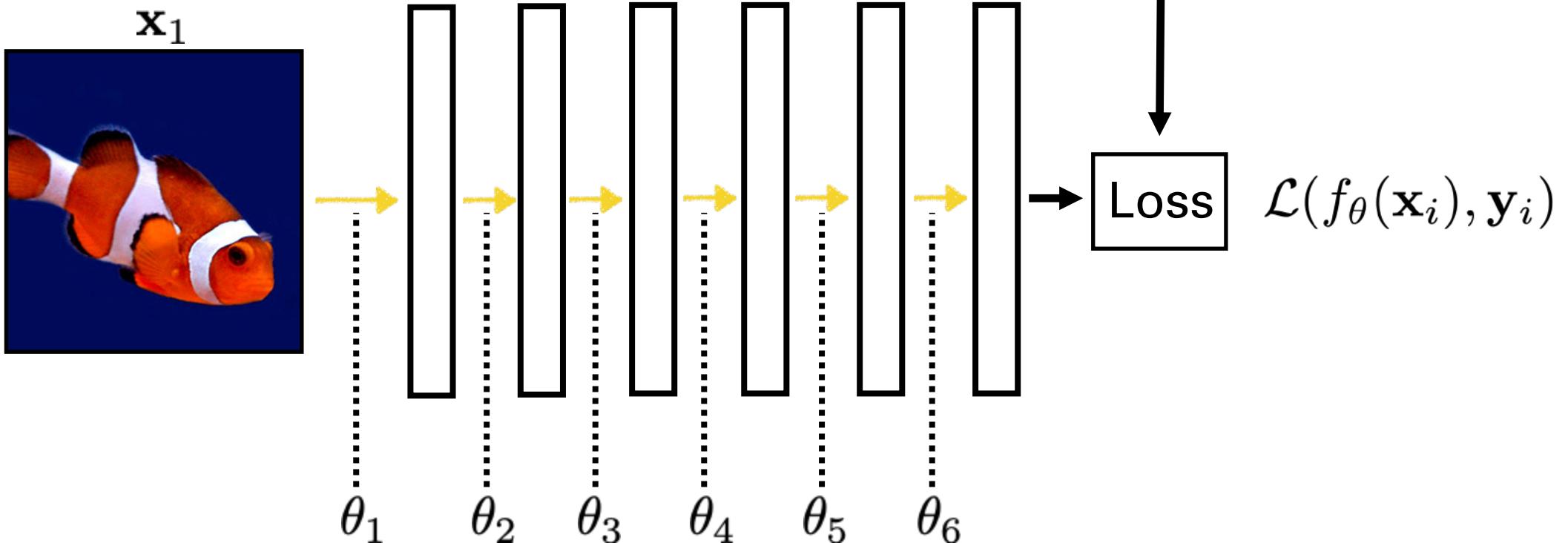
$$H(\mathbf{y}, \hat{\mathbf{y}}) = - \sum_{k=1}^K y_k \log \hat{y}_k$$

Deep learning

y_1

"clown fish"

Learned

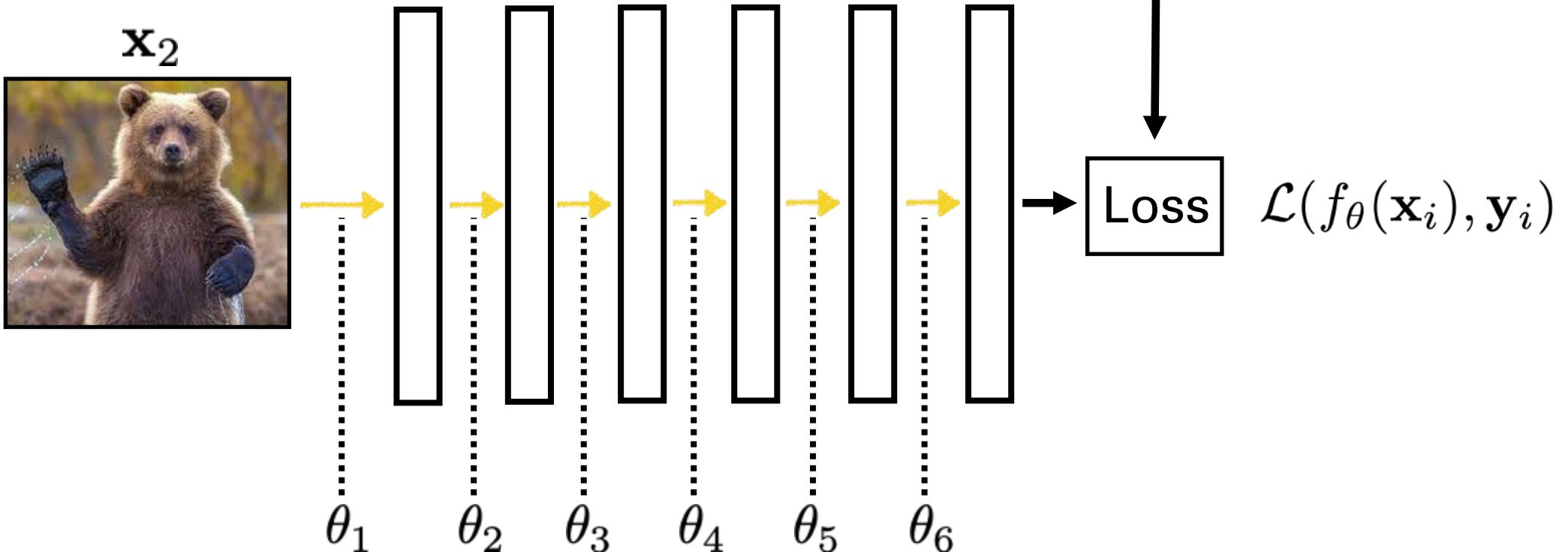


$$\theta^* = \arg \min_{\theta} \sum_{i=1}^N \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)$$

Deep learning

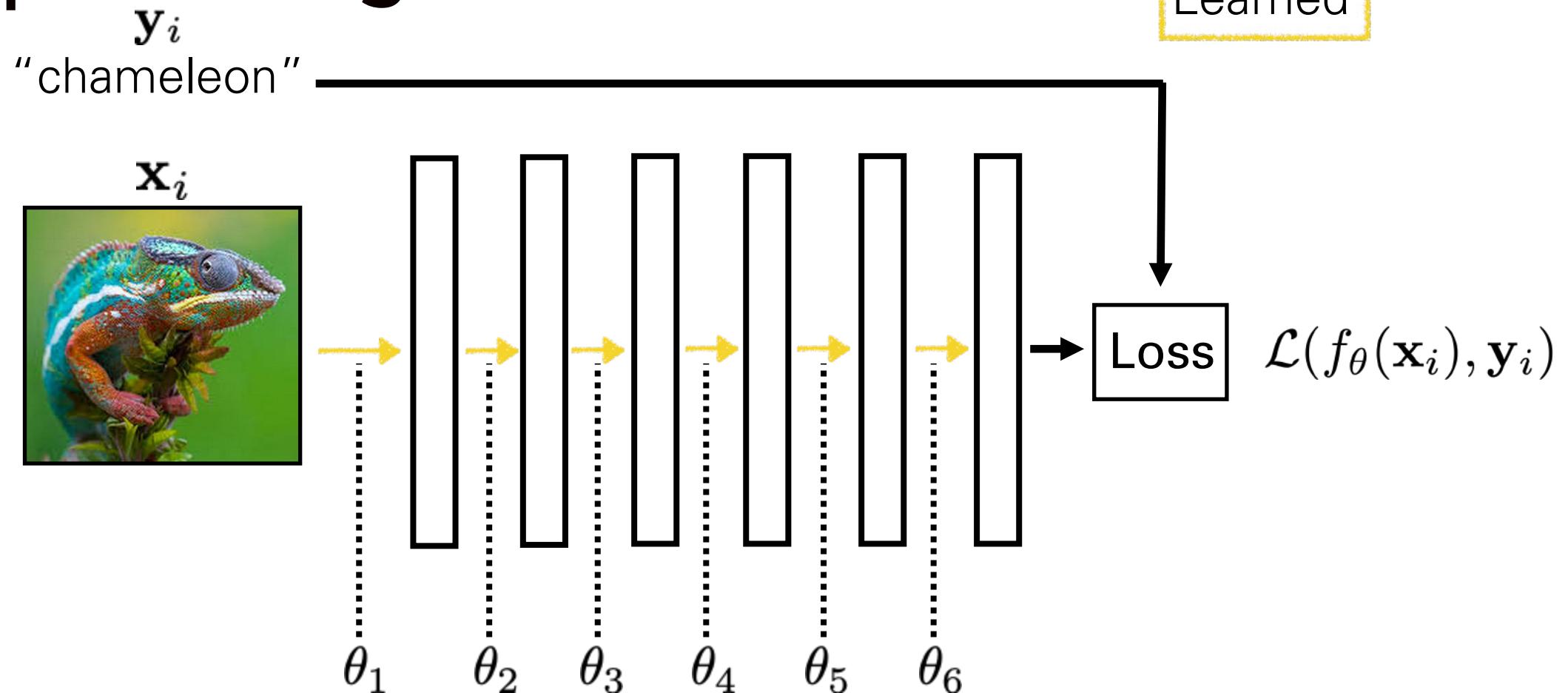
y_2
"grizzly bear"

Learned



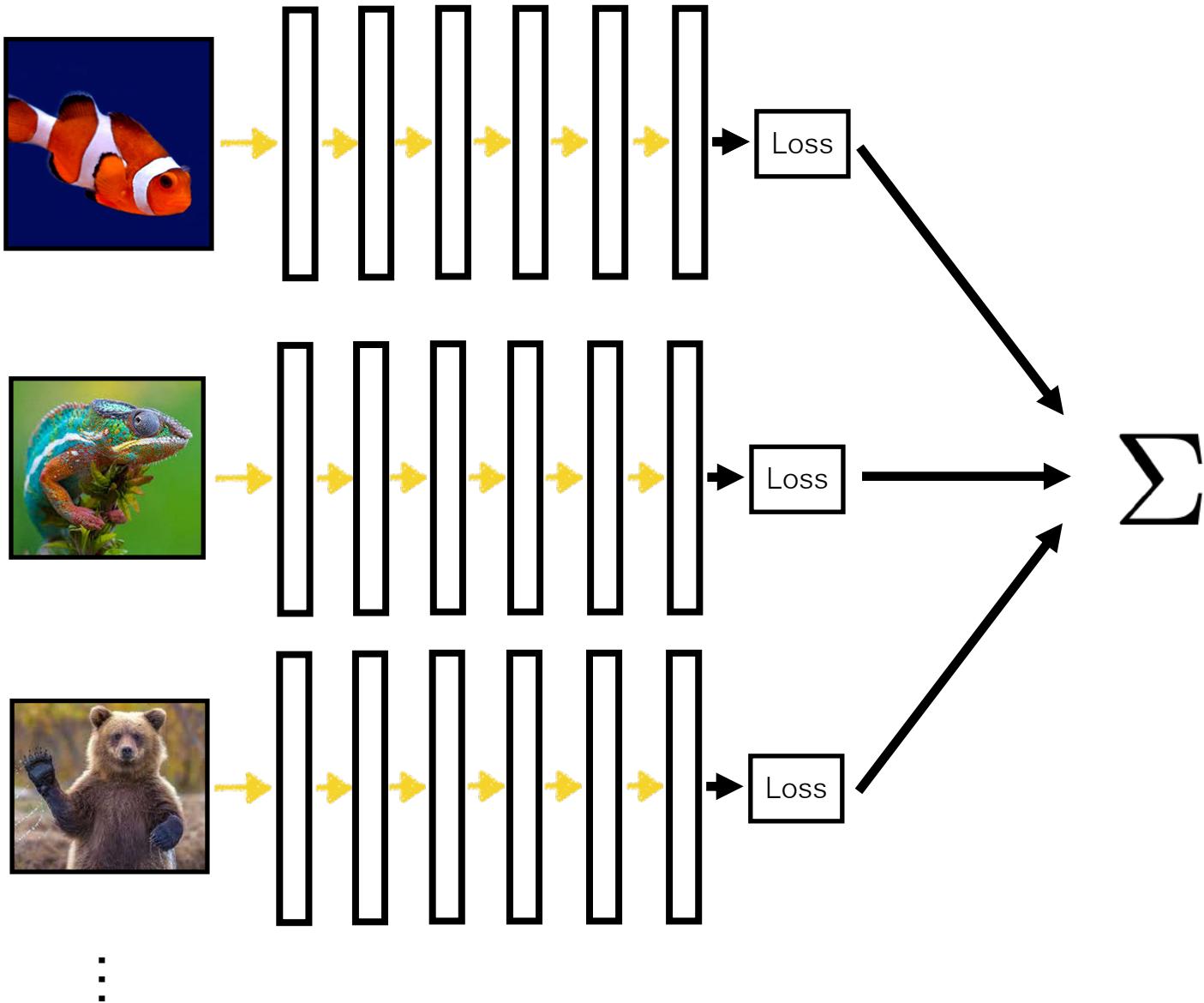
$$\theta^* = \arg \min_{\theta} \sum_{i=1}^N \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)$$

Deep learning



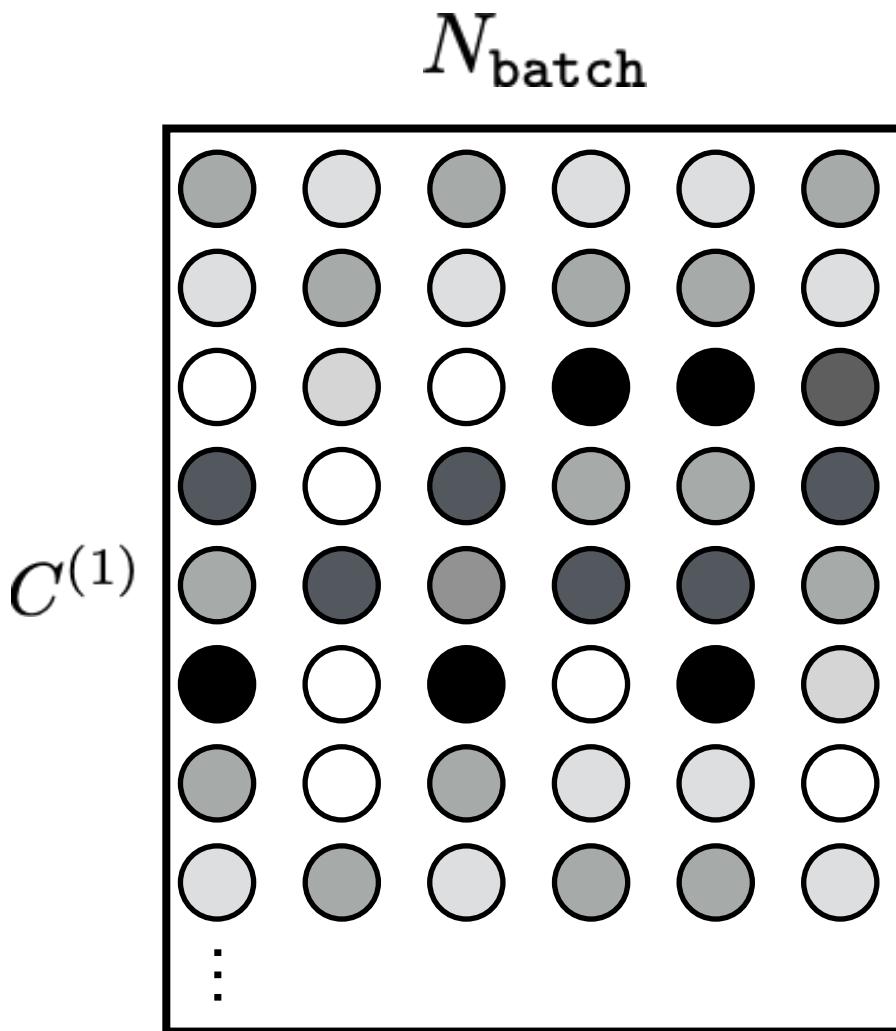
$$\theta^* = \arg \min_{\theta} \sum_{i=1}^N \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)$$

Batch (parallel) processing



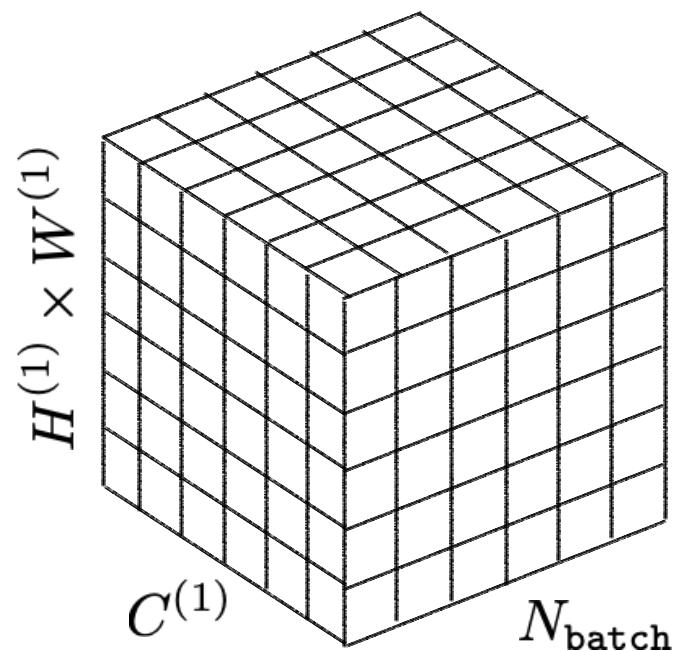
Tensors

$$\mathbf{h}^{(1)} \in \mathbb{R}^{N_{\text{batch}} \times C^{(1)}}$$

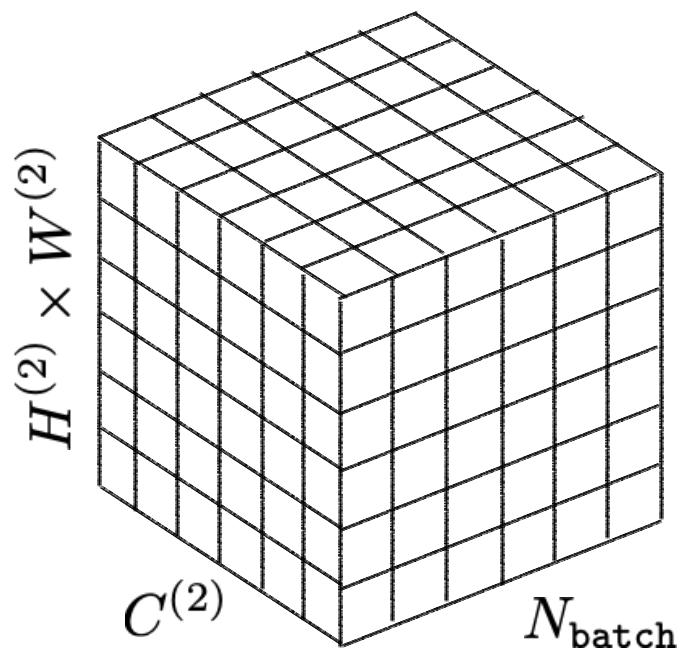


"Tensor flow"

$$\mathbf{h}^{(1)} \in \mathbb{R}^{N_{\text{batch}} \times H^{(1)} \times W^{(1)} \times C^{(1)}}$$



$$\mathbf{h}^{(2)} \in \mathbb{R}^{N_{\text{batch}} \times H^{(2)} \times W^{(2)} \times C^{(2)}}$$



Regularizing deep nets

- Deep nets have millions of parameters!
- On many datasets, it is easy to overfit — we may have more free parameters than data points to constrain them.
- How can we regularize to prevent the network from overfitting?
 1. Fewer neurons, fewer layers
 2. Weight decay
 3. Dropout
 4. Normalization layers
 5. ...

Recall: regularized least squares

$$f_{\theta}(x) = \sum_{k=0}^K \theta_k x^k$$

$$R(\theta) = \lambda \|\theta\|_2^2 \leftarrow$$

Only use polynomial terms if you really need them! Most terms should be zero

ridge regression, a.k.a., Tikhonov regularization

Probabilistic interpretation: R is a Gaussian **prior** over values of the parameters.

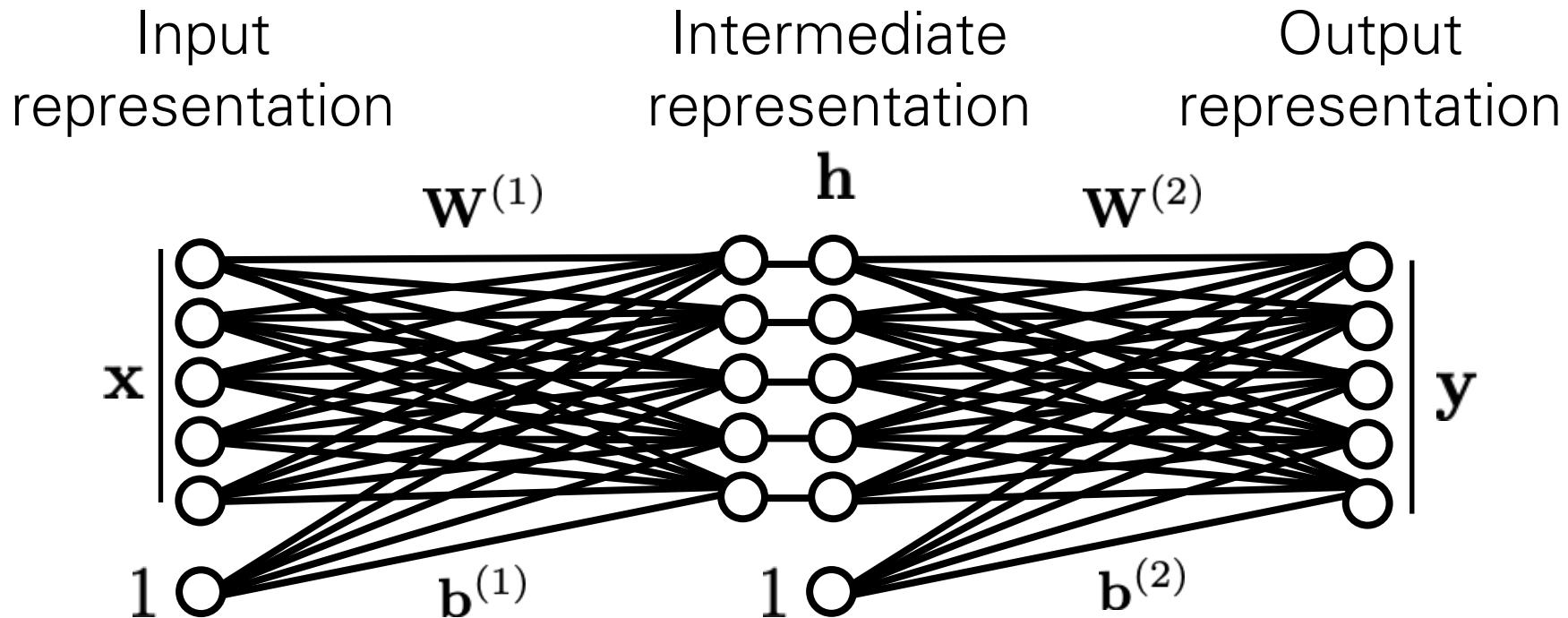
Recall: regularized least squares

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

$$R(\mathbf{W}) = \lambda \|\mathbf{W}\|_2^2 \quad \leftarrow \quad \text{weight decay}$$

“We prefer to keep weights small.”

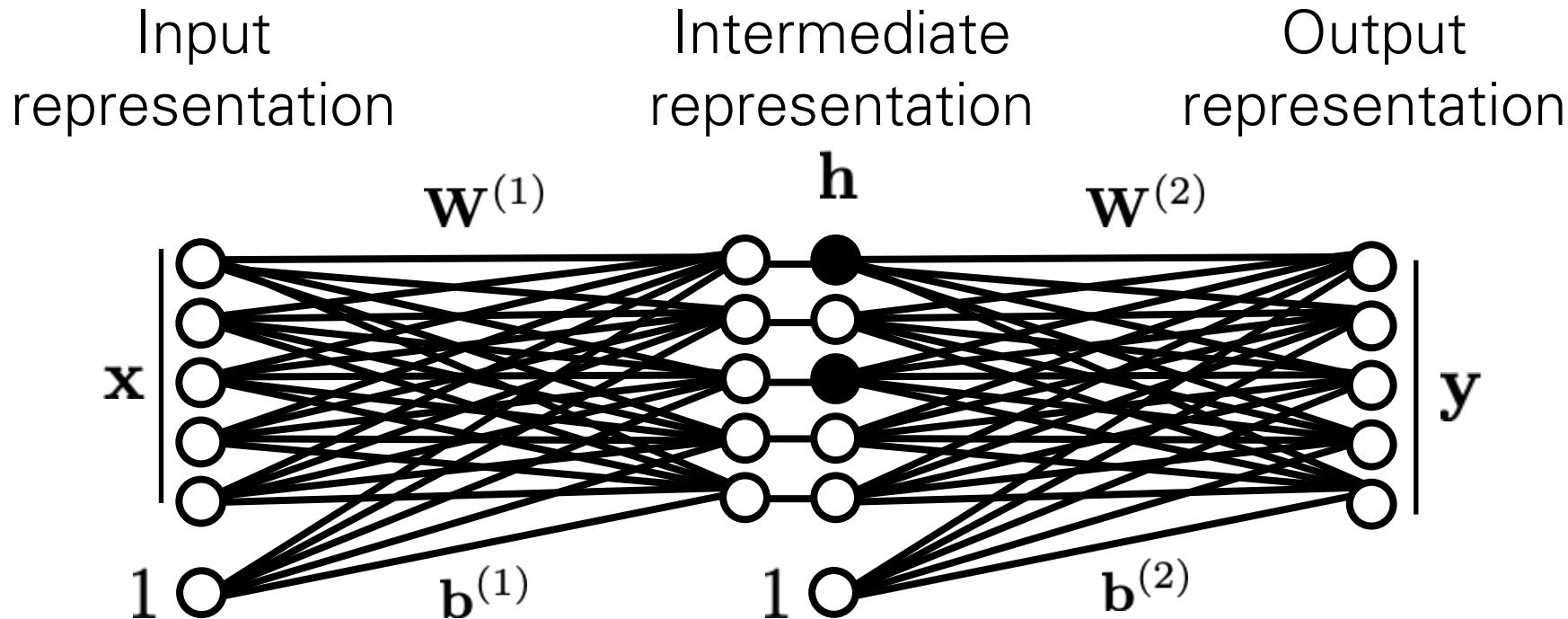
Dropout



$$\theta = \{\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(L)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(L)}\}$$

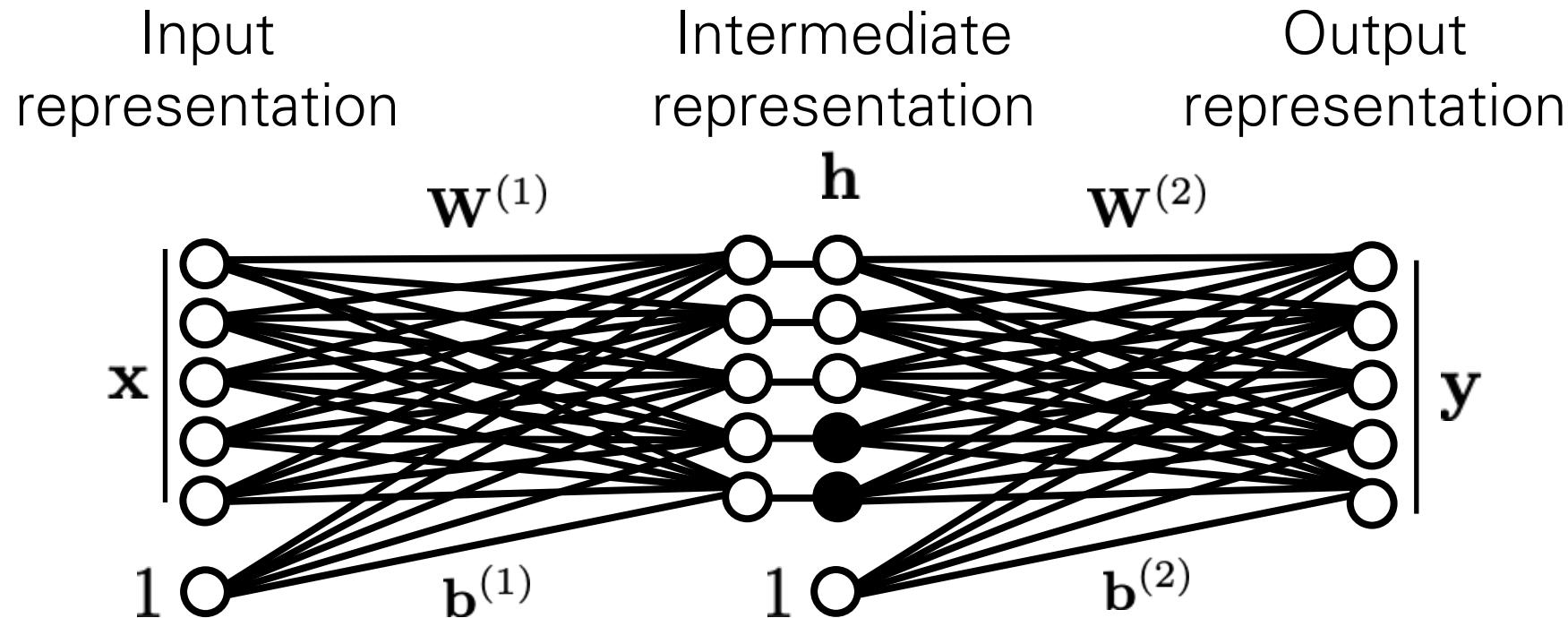
Randomly zero out
hidden units.

Dropout



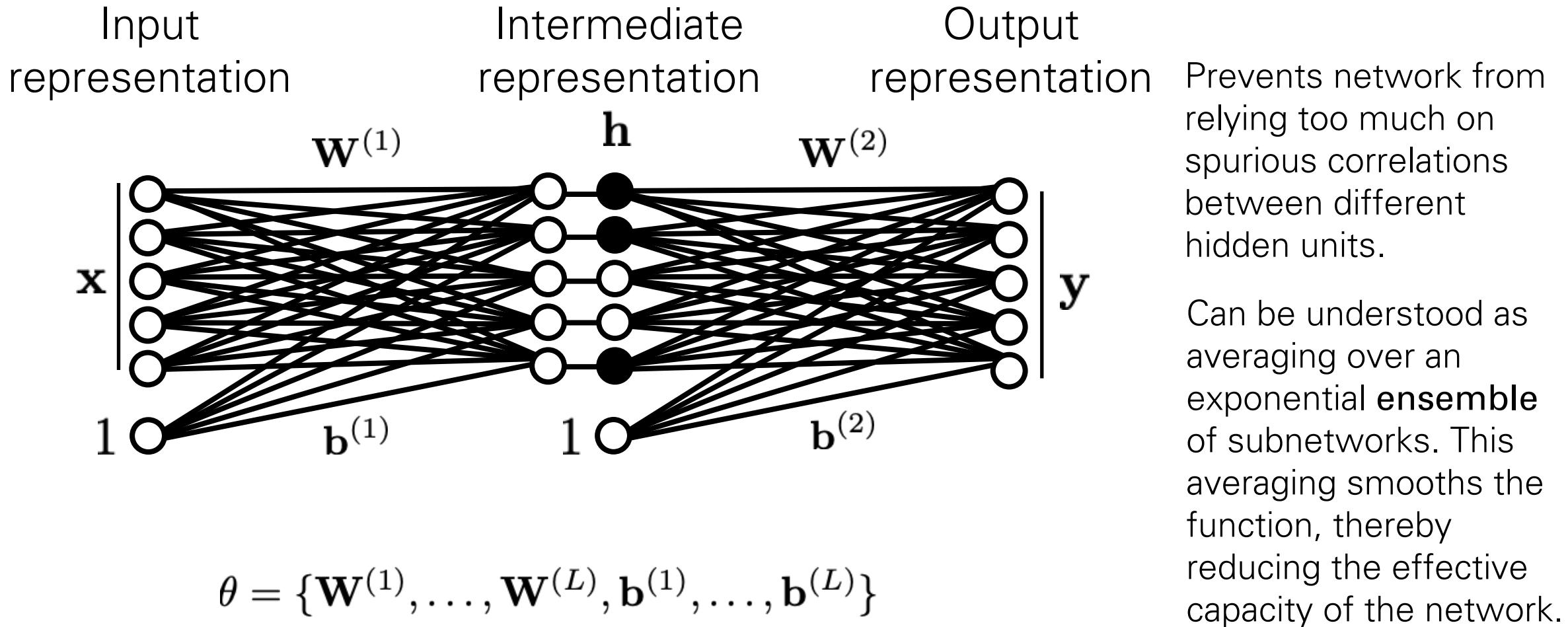
$$\theta = \{\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(L)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(L)}\}$$

Dropout

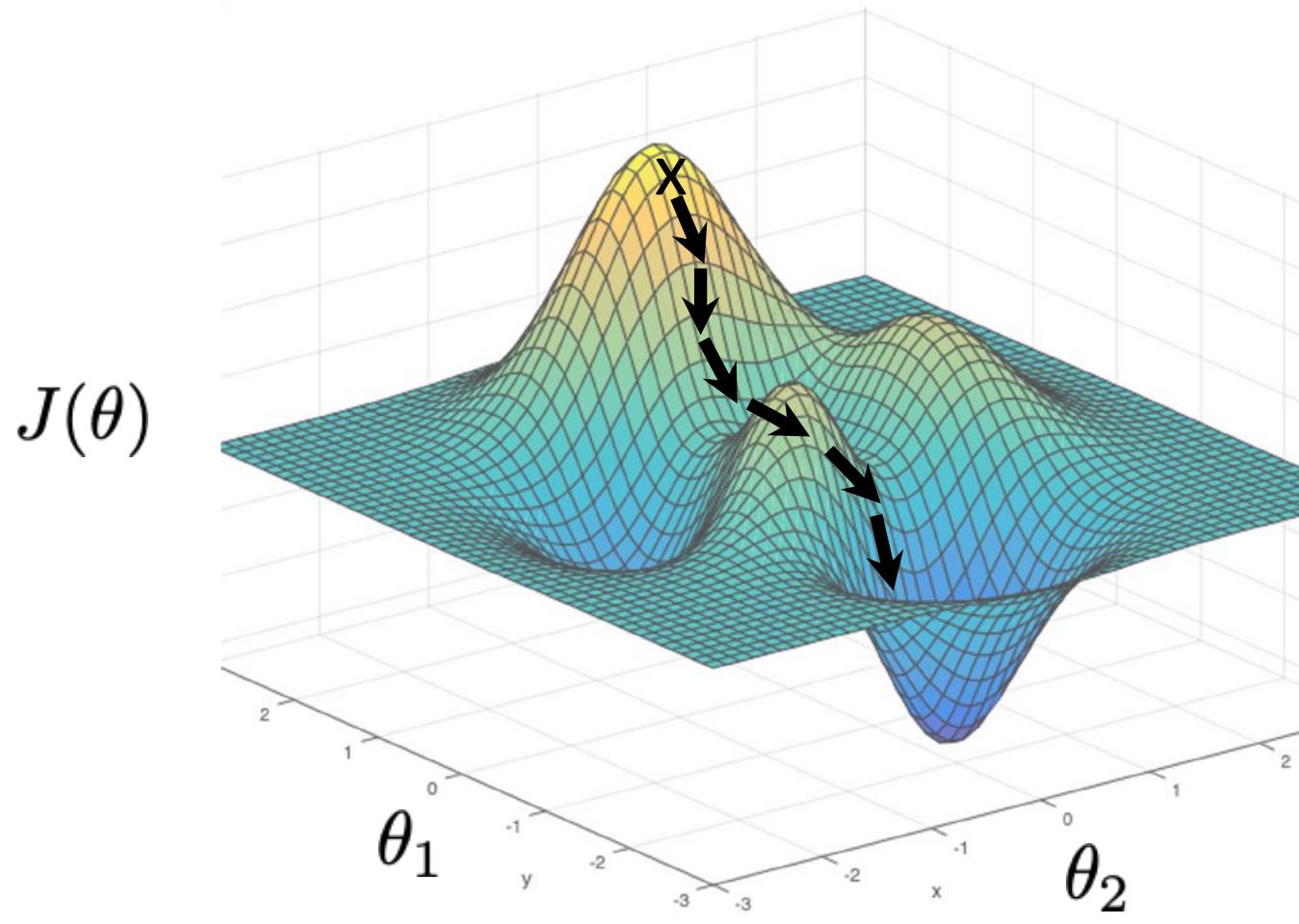


$$\theta = \{\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(L)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(L)}\}$$

Dropout



Gradient descent



$$\theta^* = \arg \min_{\theta} J(\theta)$$

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

$\underbrace{\hspace{10em}}_{J(\theta)}$

One iteration of gradient descent:

$$\theta^{t+1} = \theta^t - \eta_t \frac{\partial J(\theta)}{\partial \theta} \Big|_{\theta=\theta^t}$$

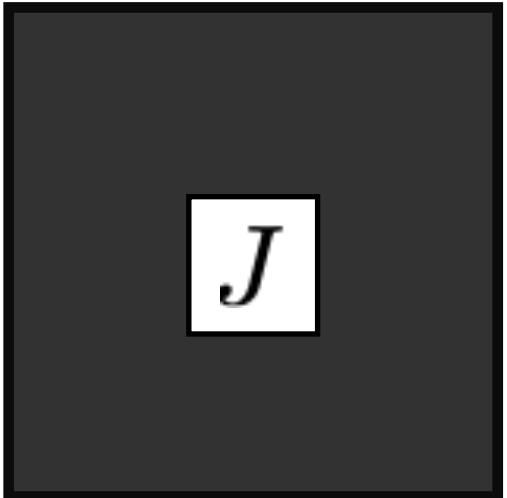


learning rate

Optimization

Params

$$\theta \rightarrow$$



$$J(\theta) \\ \nabla_{\theta} J(\theta) \\ H_{\theta}(J(\theta))$$

$$\theta^* = \arg \min_{\theta} J(\theta)$$

- What's the knowledge we have about J ?

- We can evaluate $J(\theta)$

Gradient

← Black box optimization

- We can evaluate $J(\theta)$ and $\nabla_{\theta} J(\theta)$

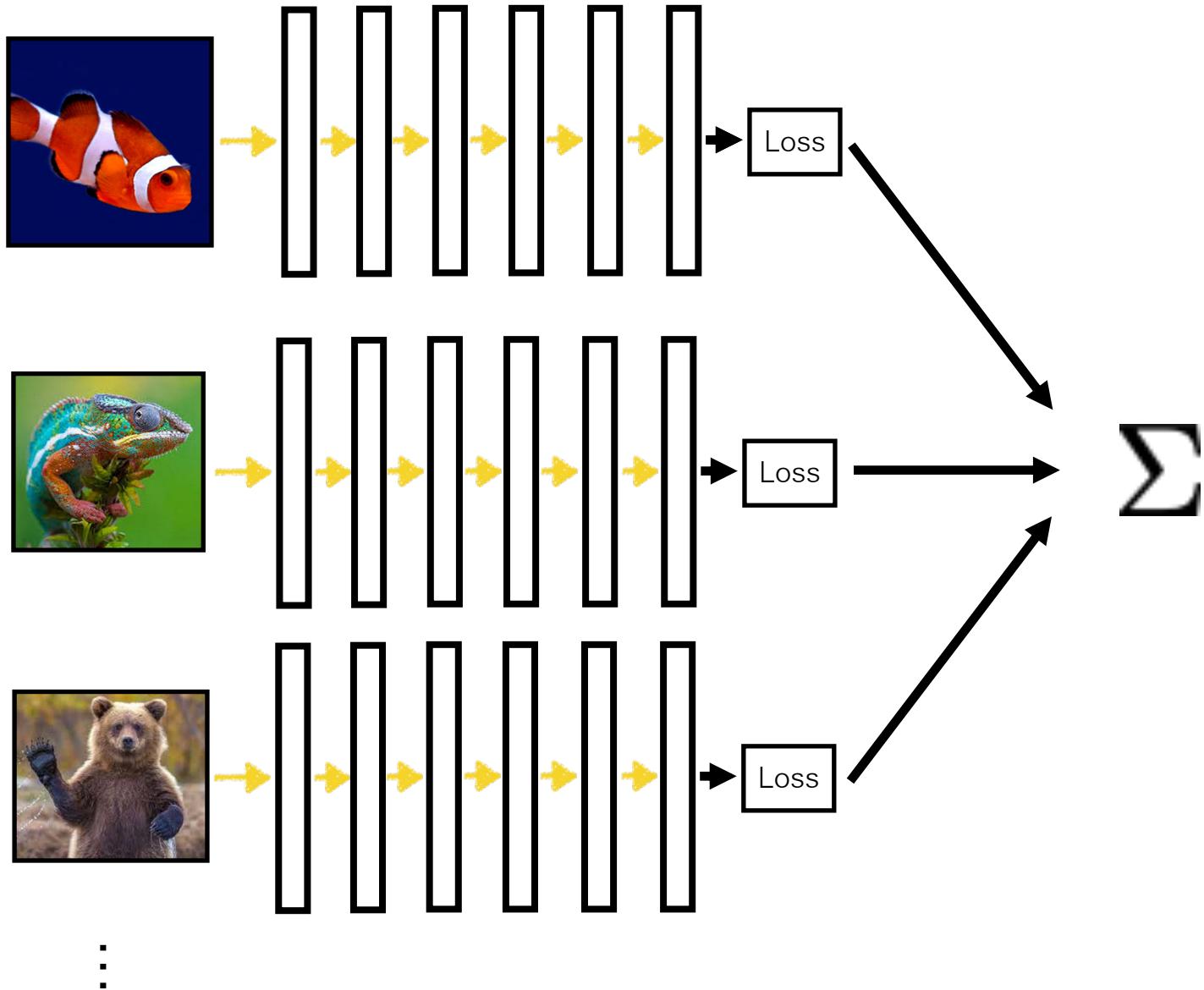
← First order optimization

- We can evaluate $J(\theta)$, $\nabla_{\theta} J(\theta)$, and $H_{\theta}(J(\theta))$

← Second order optimization

Hessian

Batch (parallel) processing



Stochastic gradient descent (SGD)

- Want to minimize overall loss function J , which is sum of individual losses over each example.
- In Stochastic gradient descent, compute gradient on sub-set (batch) of data.
 - If batchsize=1 then θ is updated after each example.
 - If batchsize=N (full set) then this is standard gradient descent.
- Gradient direction is noisy, relative to average over all examples (standard gradient descent).
- **Advantages**
 - Faster: approximate total gradient with small sample
 - Implicit regularizer
- **Disadvantages**
 - High variance, unstable updates

Momentum

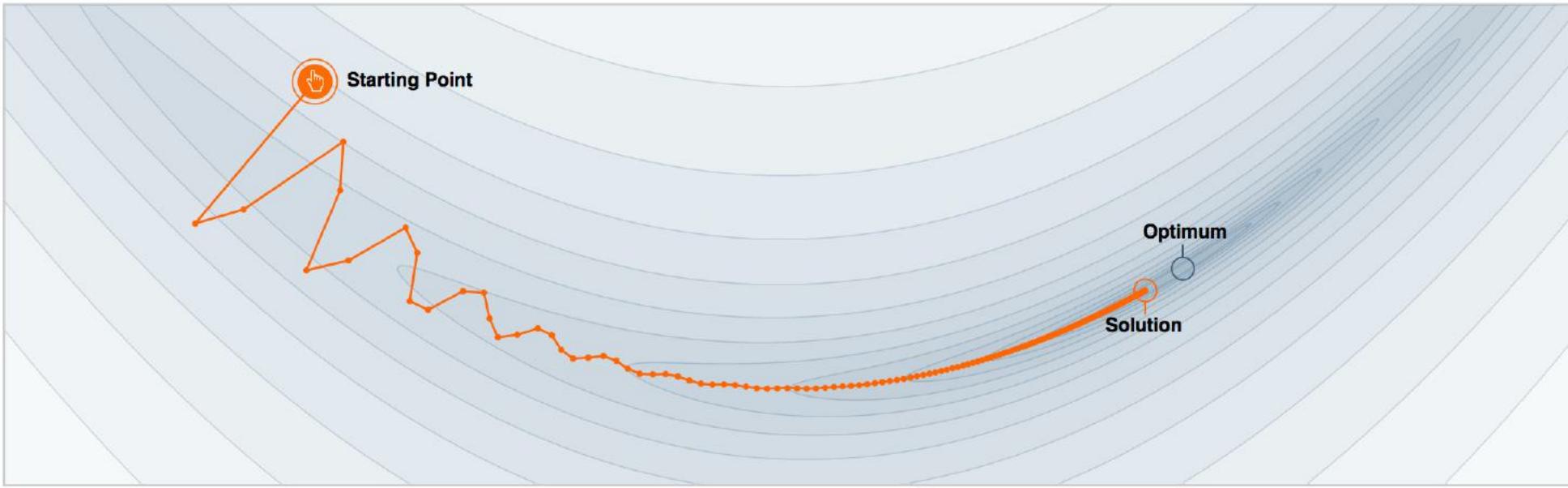
- **Basic idea:** like a ball rolling down a hill, we should build up speed so as to make faster progress when “on a roll”
- Can dampen oscillations in SGD updates
- Common in popular variants of SGD
 - Nesterov’s method
 - RMSProp
 - Adam



Raiders of the Lost Ark (excerpt)

(Source: Lucasfilm/Paramount Pictures, 1981)

Why Momentum Really Works



Step-size $\alpha = 0.02$

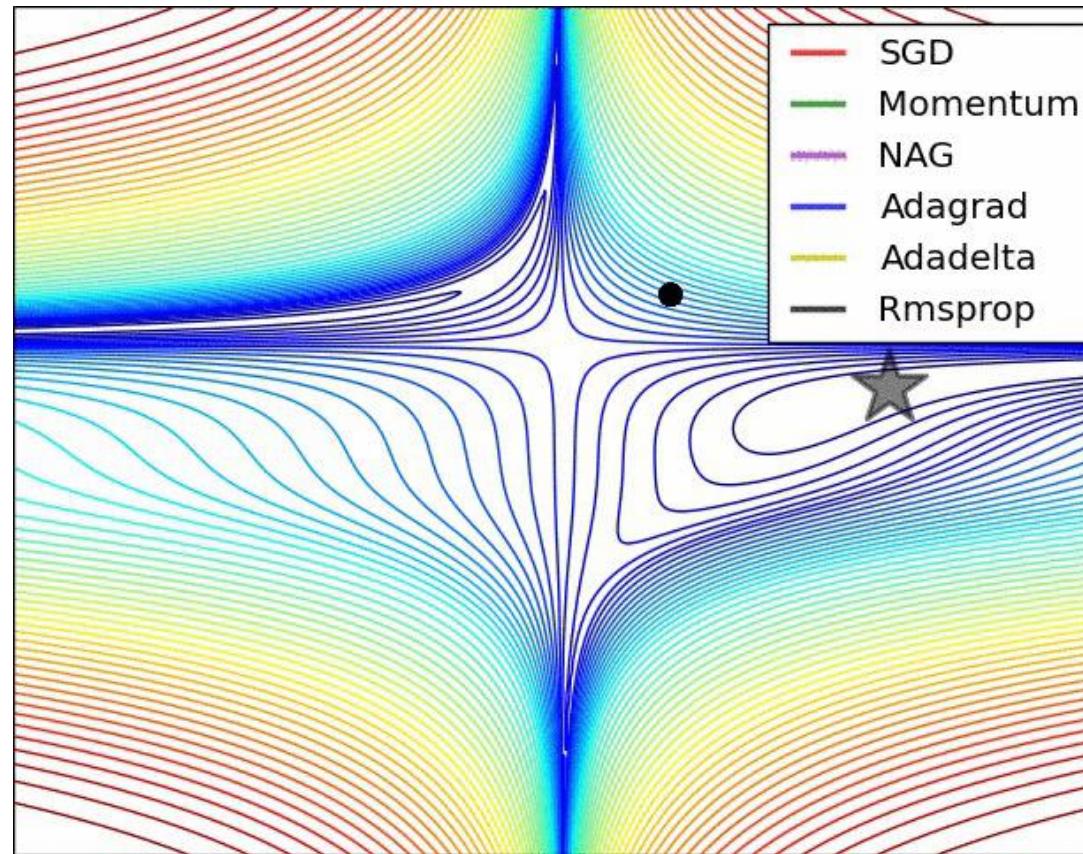


Momentum $\beta = 0.99$



We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

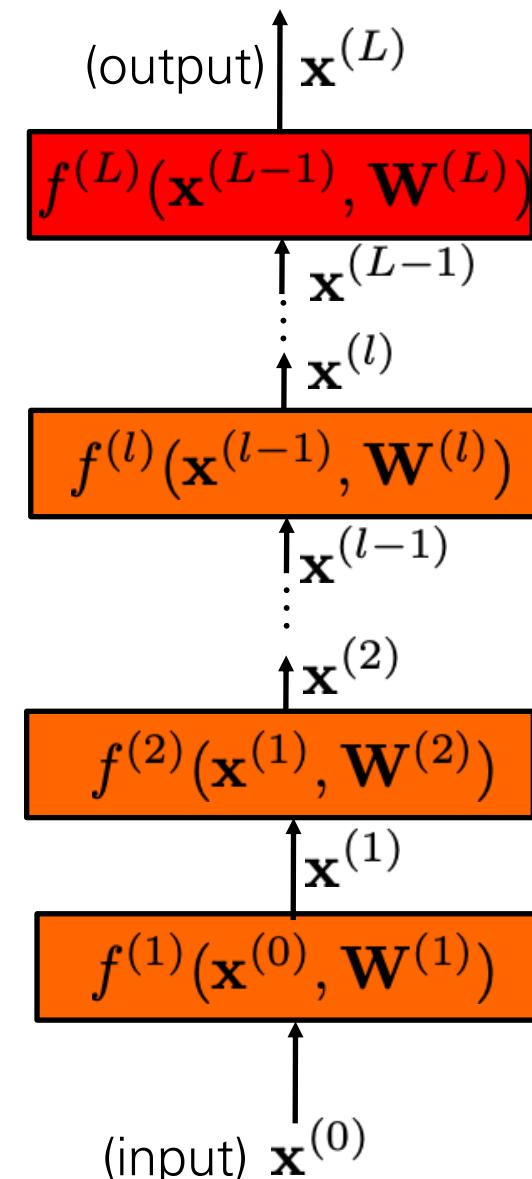
Comparison of gradient descent variants



[<http://ruder.io/optimizing-gradient-descent/>]

Forward pass

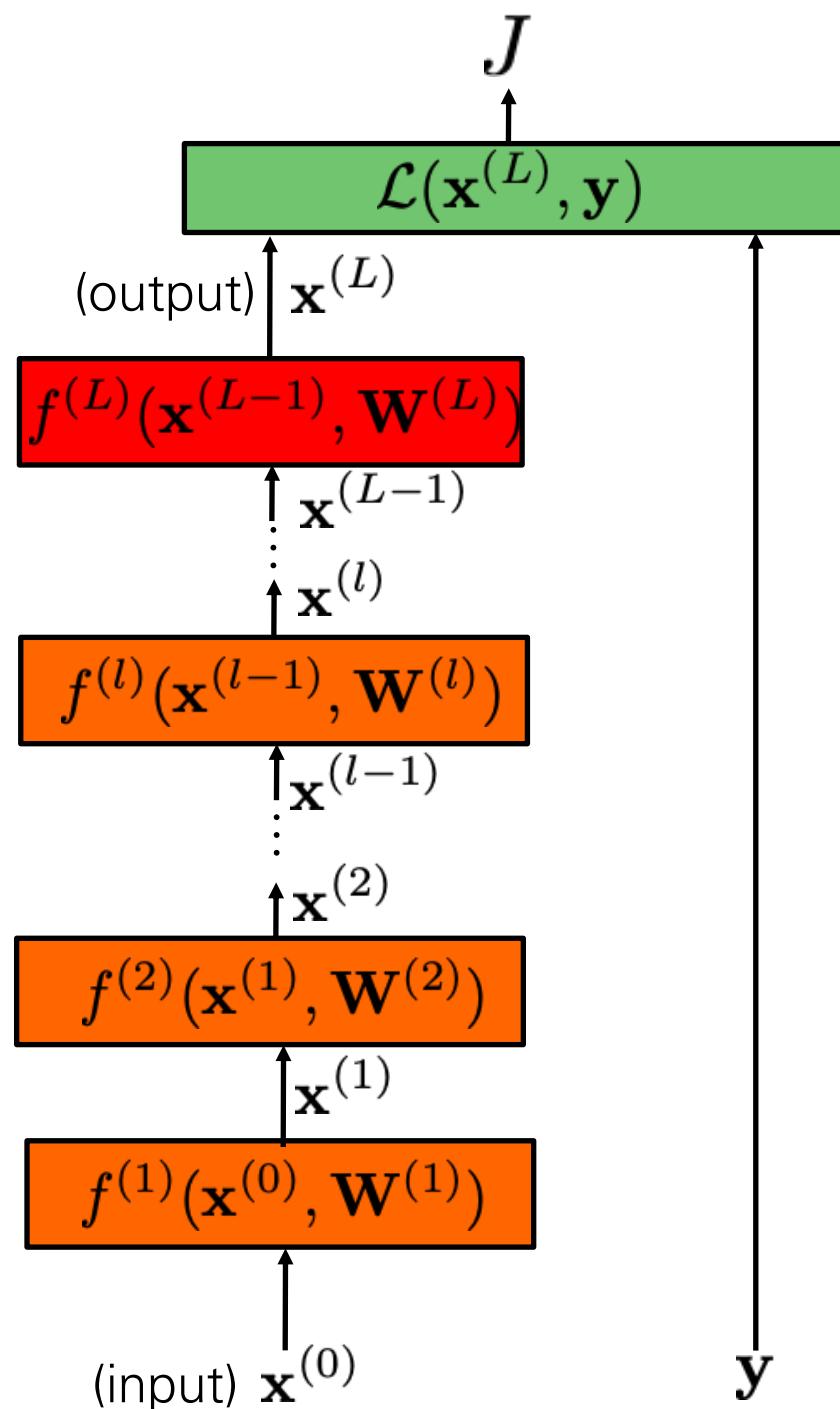
- Consider model with L layers. Layer l has vector of weights $\mathbf{W}^{(l)}$
- **Forward pass:** takes input $\mathbf{x}^{(l-1)}$ and passes it through each layer $f^{(l)}$:
$$\mathbf{x}^{(l)} = f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})$$
- Output of layer l is $\mathbf{x}^{(l)}$.
- Network output (top layer) is $\mathbf{x}^{(L)}$.



Forward pass

- Consider model with L layers. Layer l has vector of weights $\mathbf{W}^{(l)}$
- **Forward pass:** takes input $\mathbf{x}^{(l-1)}$ and passes it through each layer $f^{(l)}$:
$$\mathbf{x}^{(l)} = f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})$$
- Output of layer l is $\mathbf{x}^{(l)}$.
- Network output (top layer) is $\mathbf{x}^{(L)}$.
- **Loss function** \mathcal{L} compares $\mathbf{x}^{(L)}$ to \mathbf{y} .
- Overall energy is the sum of the cost over all training examples:

$$J = \sum_{i=1}^N \mathcal{L}(\mathbf{x}_i^{(L)}, \mathbf{y}_i)$$



Gradient descent

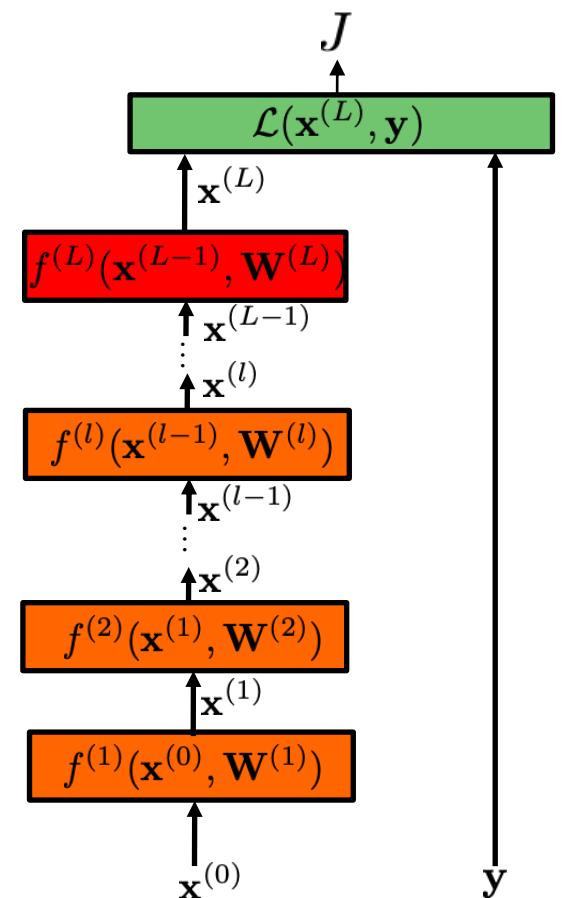
- We need to compute gradients of the cost with respect to model parameters $\mathbf{W}^{(l)}$.
- By design, each layer is differentiable with respect to its parameters and input.

Computing gradients

- To compute the gradients, we could start by writing the full energy J as a function of the network parameters.

$$J(\mathbf{W}) = \sum_{i=1} \mathcal{L}(f^{(L)}(\dots f^{(2)}(f^{(1)}(\mathbf{x}_i^{(0)}, \mathbf{W}^{(1)}), \mathbf{W}^{(2)}), \dots \mathbf{W}^{(L)}), \mathbf{y}_i)$$

And then compute the partial derivatives... instead, we can use the chain rule to derive a compact algorithm:
backpropagation



Backpropagation

- Forward pass: for each training example, compute the outputs for all layers:

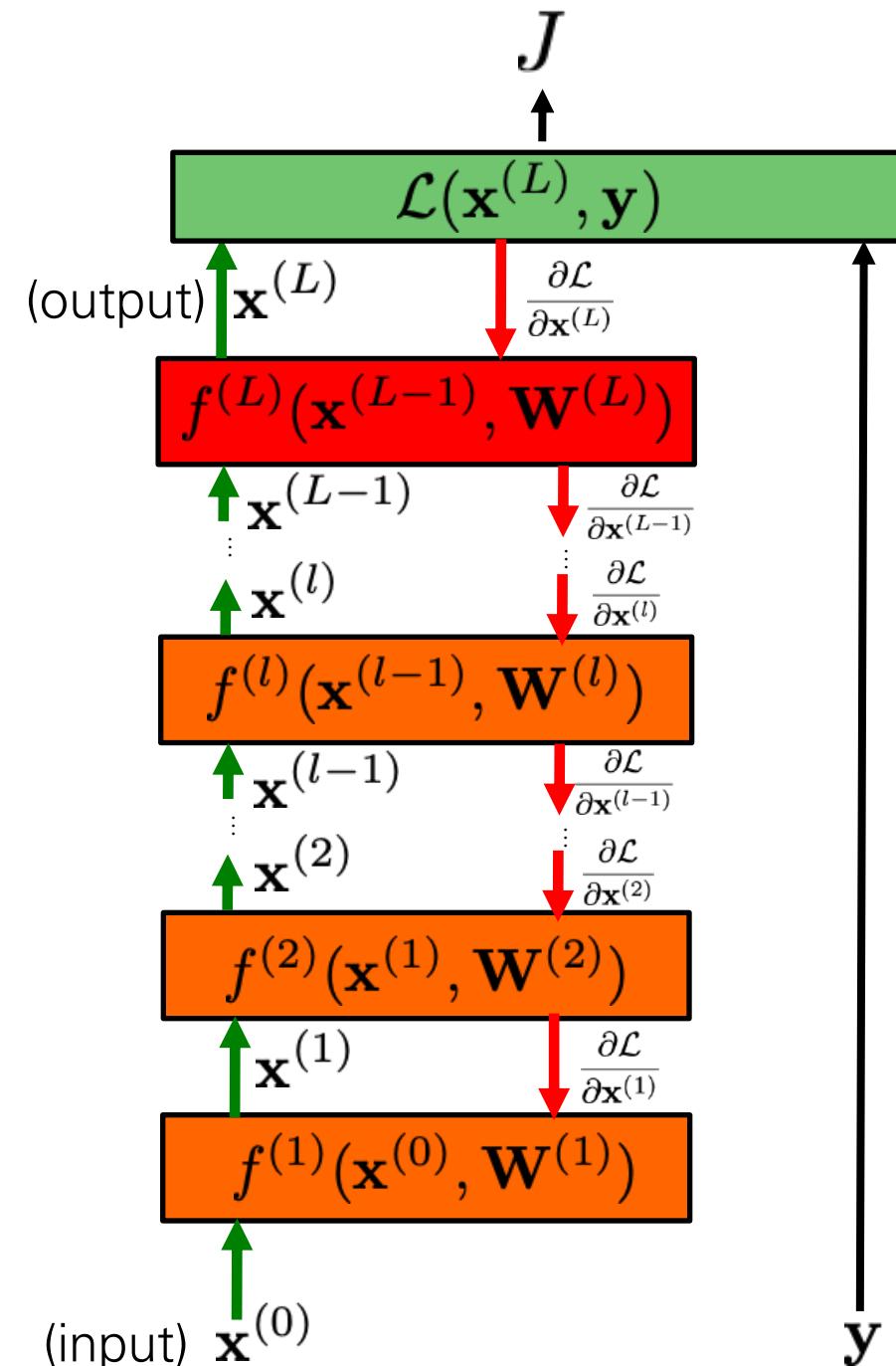
$$\mathbf{x}^{(l)} = f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})$$

- Backwards pass: compute loss derivatives iteratively from top to bottom:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l-1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l)}} \cdot \frac{\partial f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})}{\partial \mathbf{x}^{(l-1)}}$$

- Compute gradients w.r.t. weights, and update weights:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l)}} \cdot \frac{\partial f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})}{\partial \mathbf{W}^{(l)}}$$



Convolutional Neural Networks

Convolutional Neural Networks

LeCun et al. 1989



Neural network with specialized connectivity

Tailored to processing natural signals with a grid topology (e.g., images).

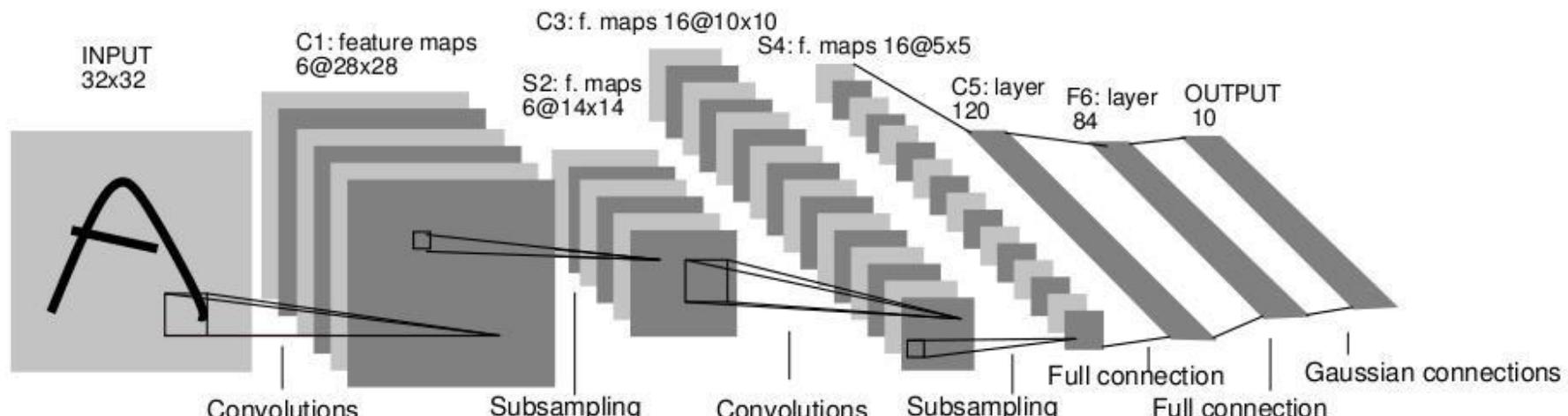


Image classification

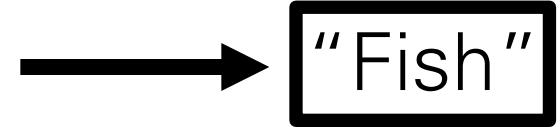
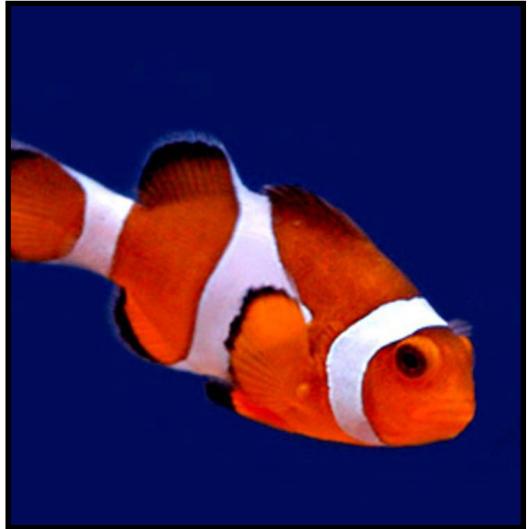
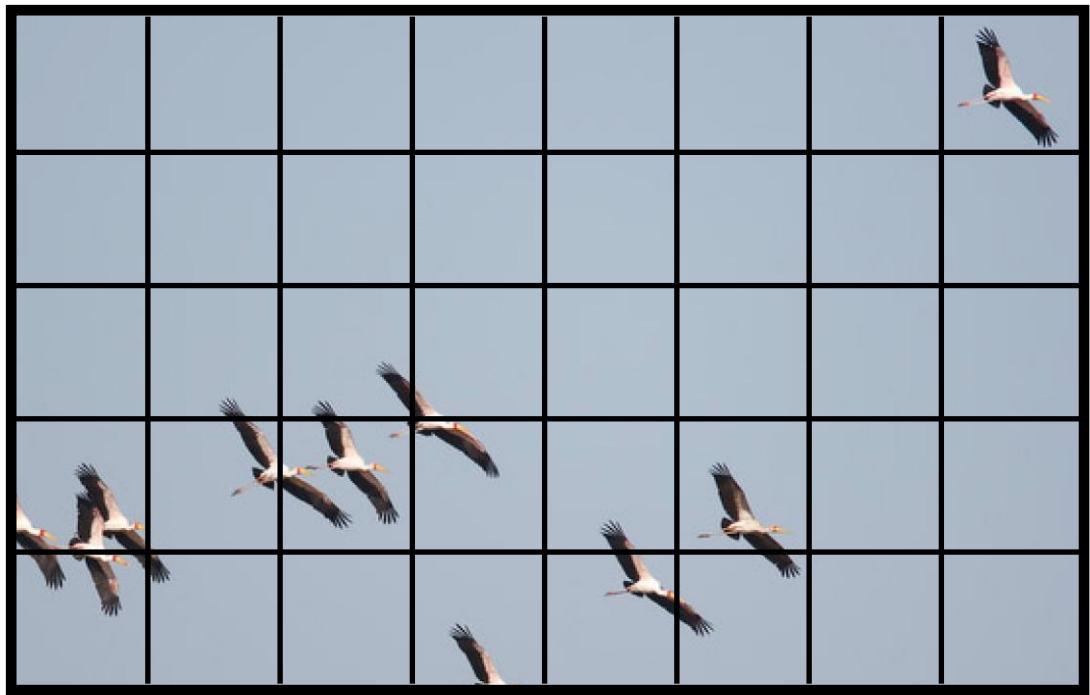


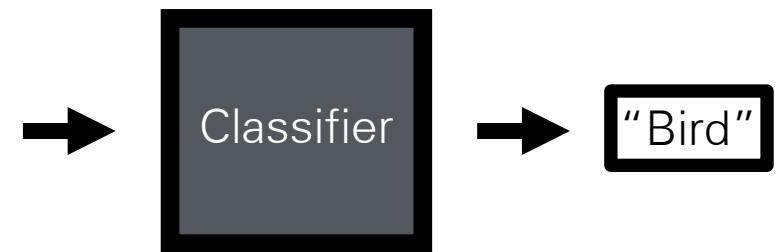
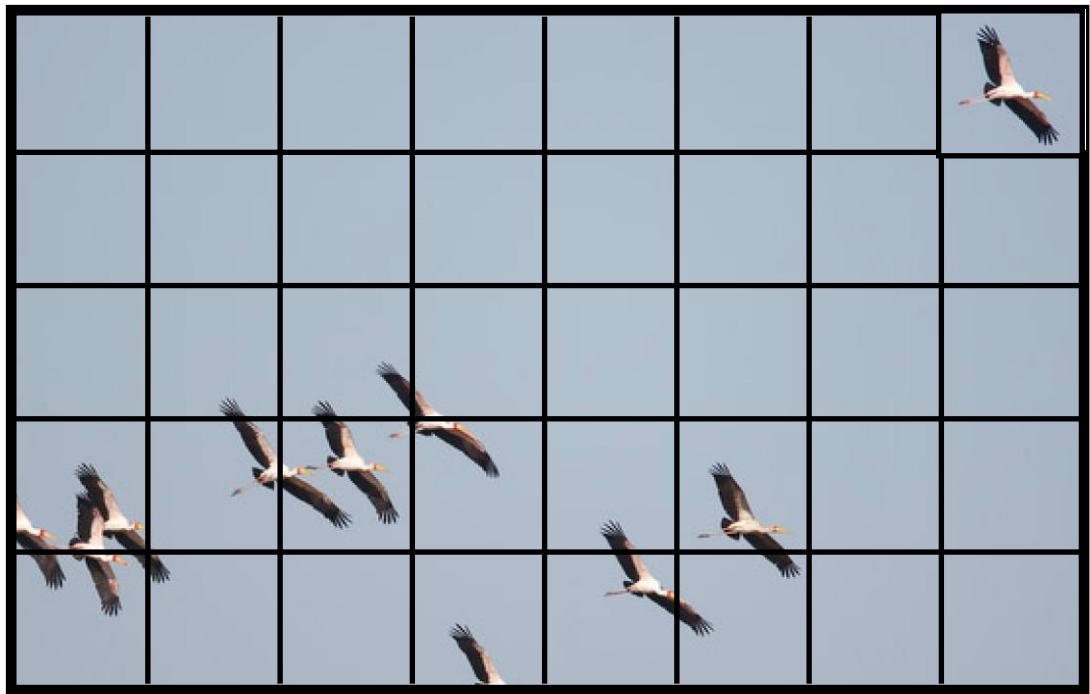
image x

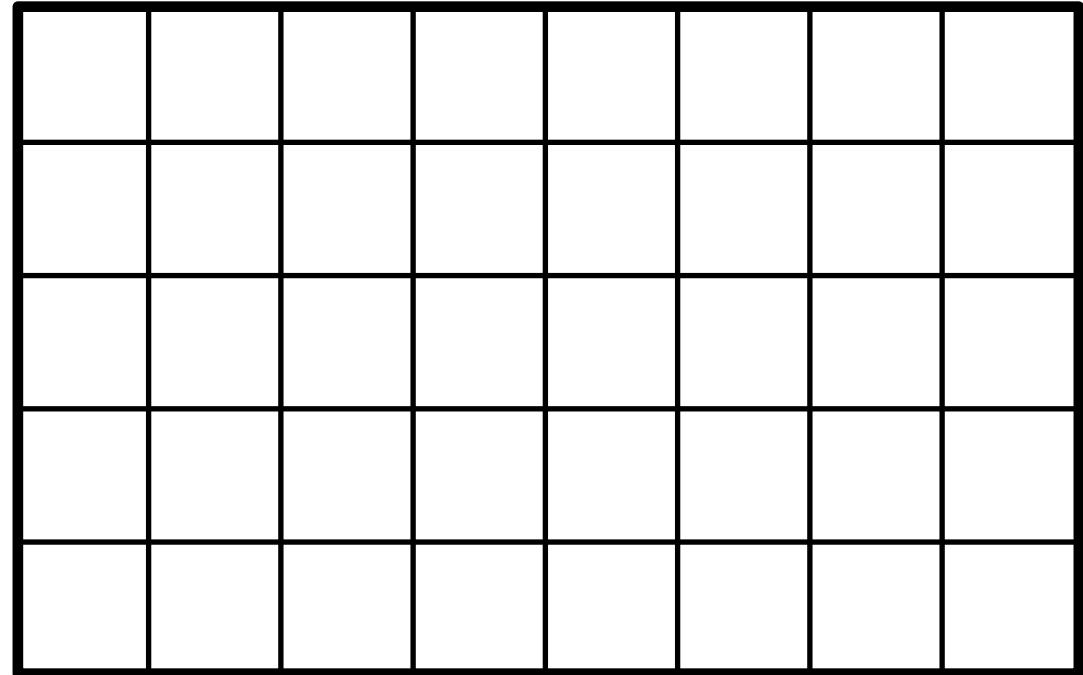
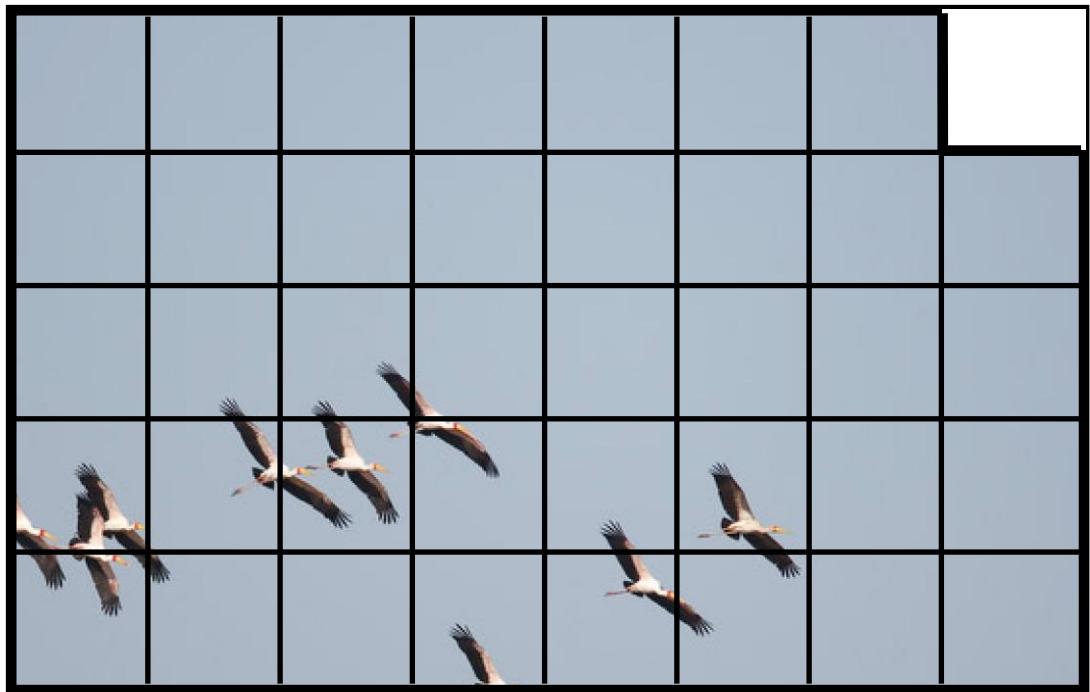
label y

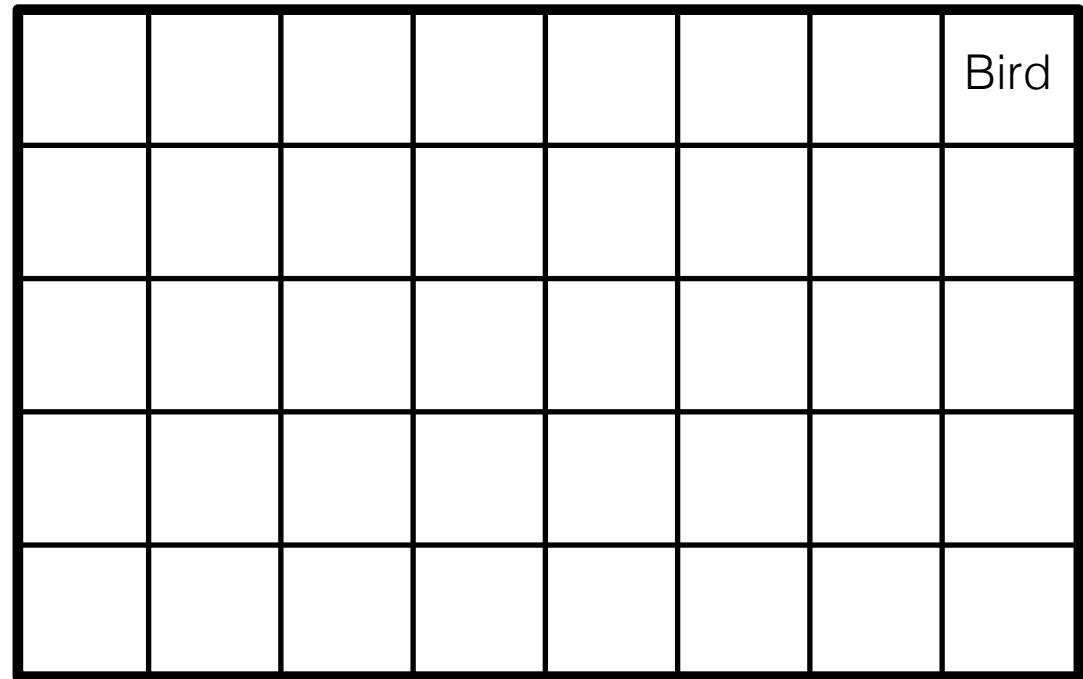
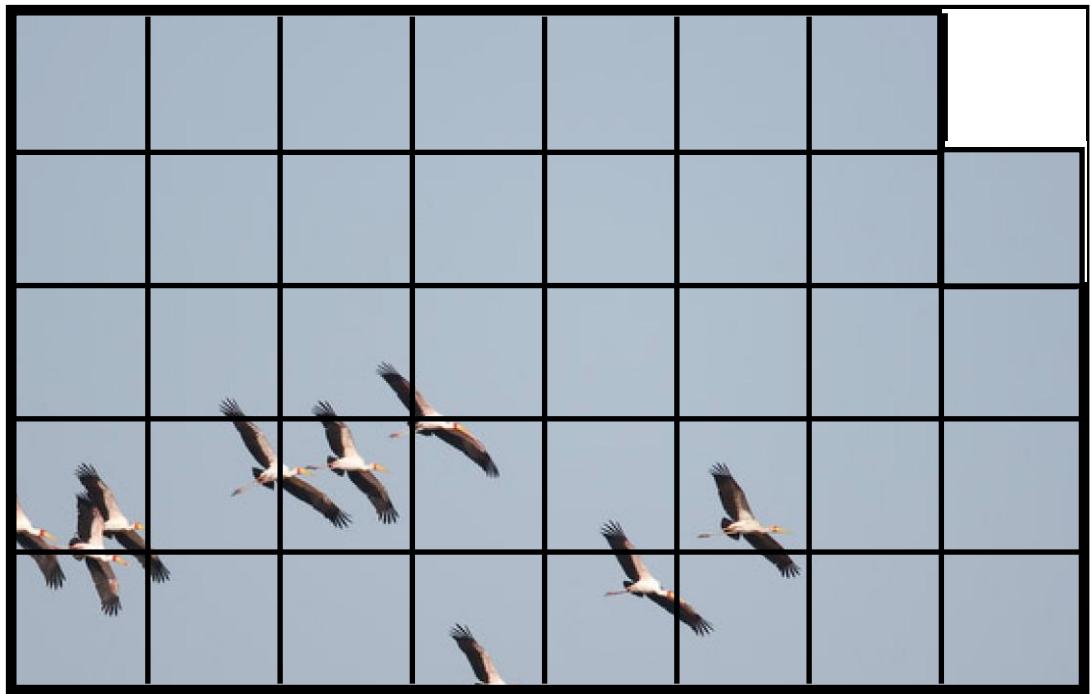


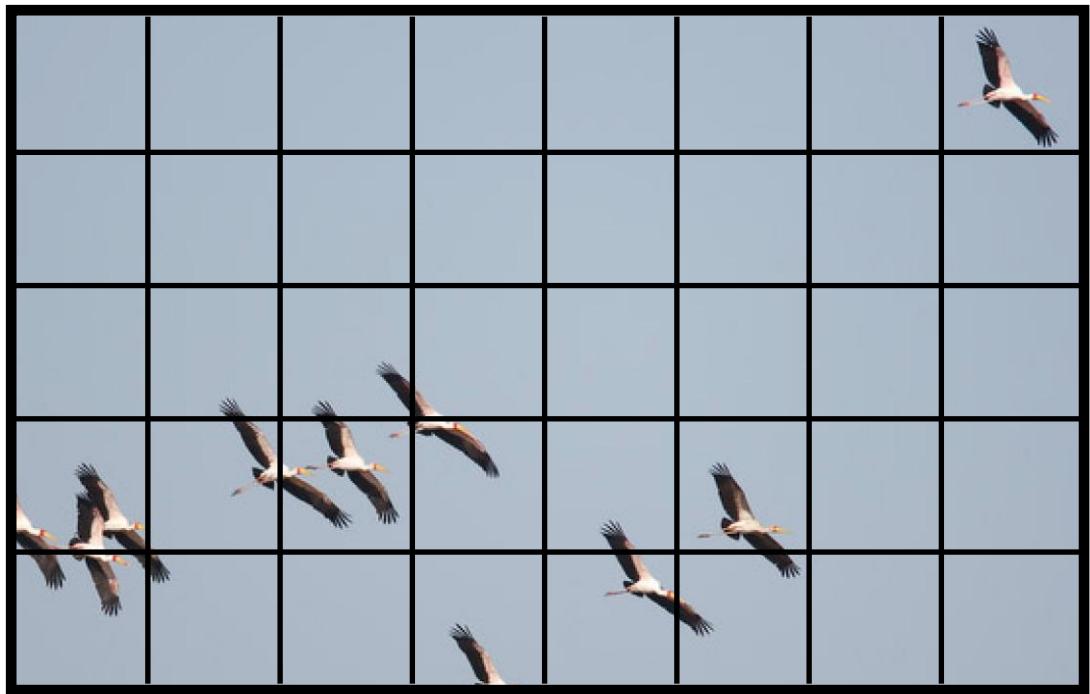
Photo credit: Fredo Durand



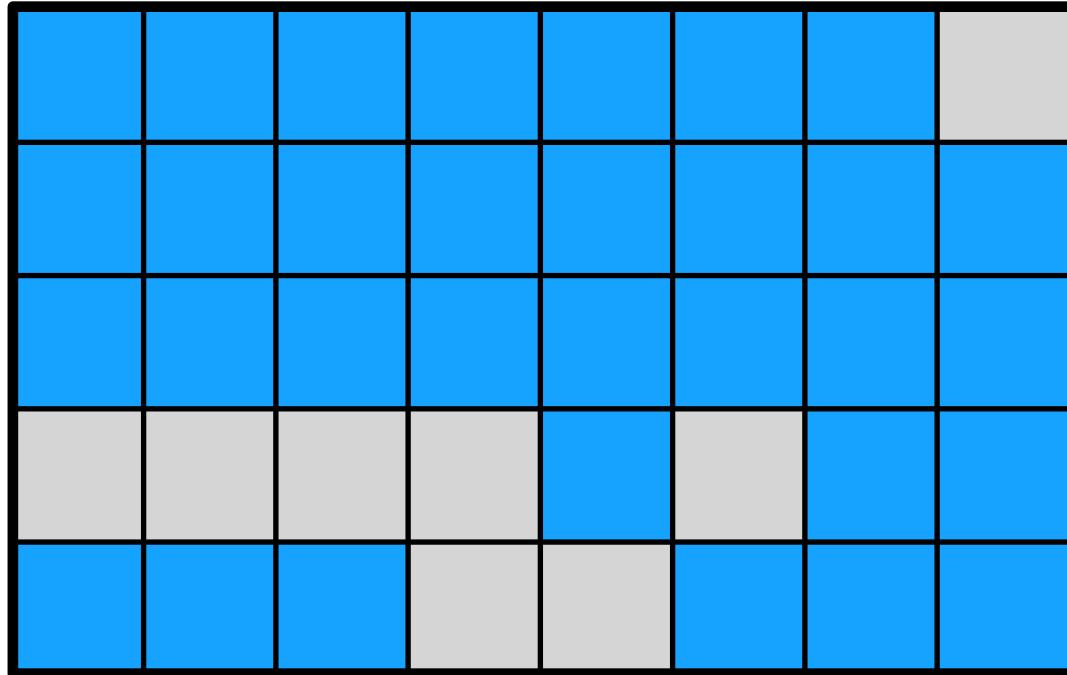
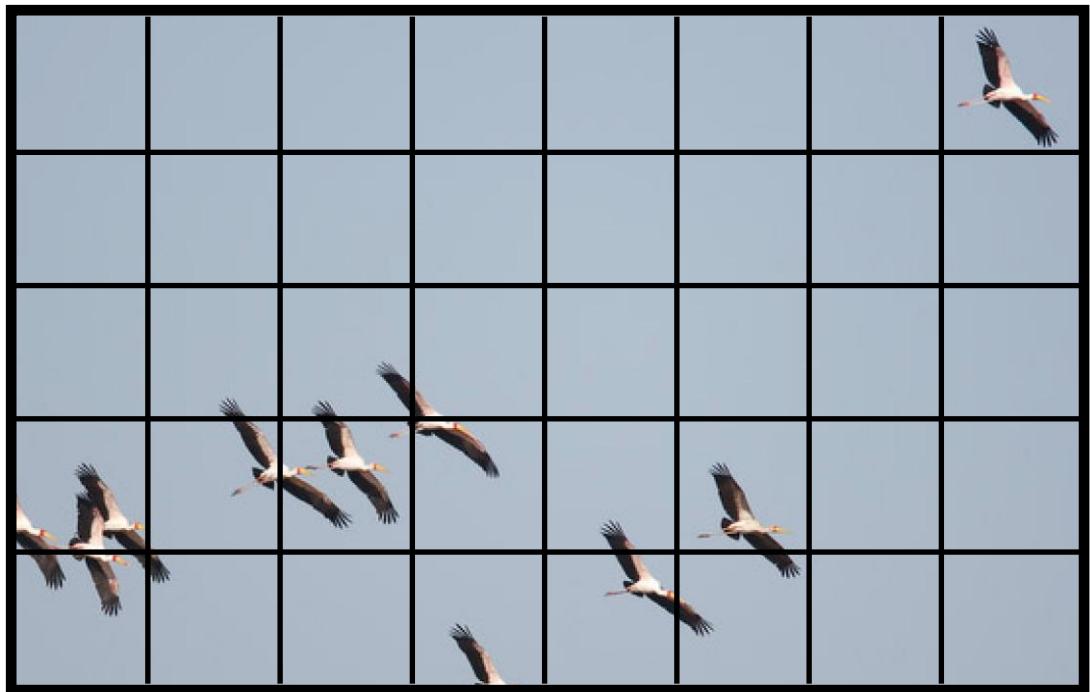


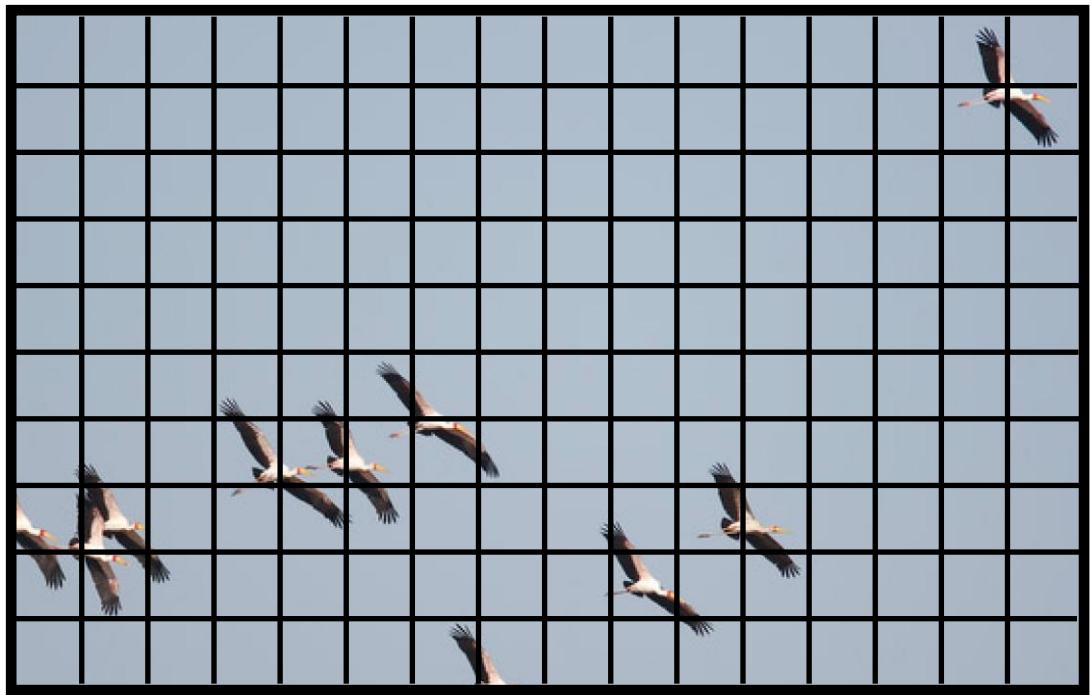


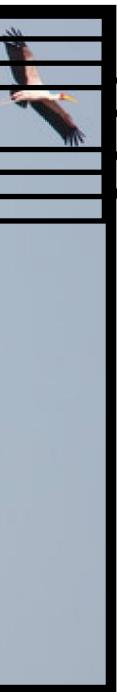




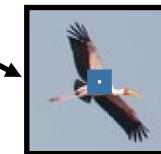
Sky	Sky	Sky	Sky	Sky	Sky	Sky	Bird
Sky	Sky	Sky	Sky	Sky	Sky	Sky	Sky
Sky	Sky	Sky	Sky	Sky	Sky	Sky	Sky
Bird	Bird	Bird	Sky	Bird	Sky	Sky	Sky
Sky	Sky	Sky	Bird	Sky	Sky	Sky	Sky



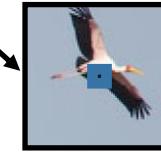




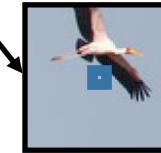
What's the object class of the center pixel?



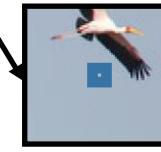
"Bird"



"Bird"



"Sky"

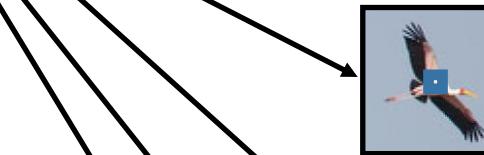


"Sky"

What's the object class of the center pixel?

Training data

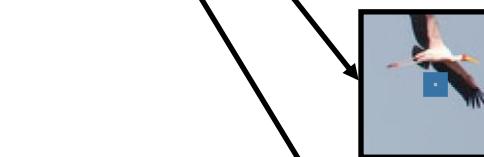
x	y
$\{$  $\}$	y "Bird"
$,$	
$\{$  $\}$	"Bird"
$,$	
$\{$  $\}$	"Sky"
$,$	
\vdots	



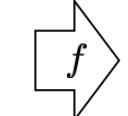
"Bird"



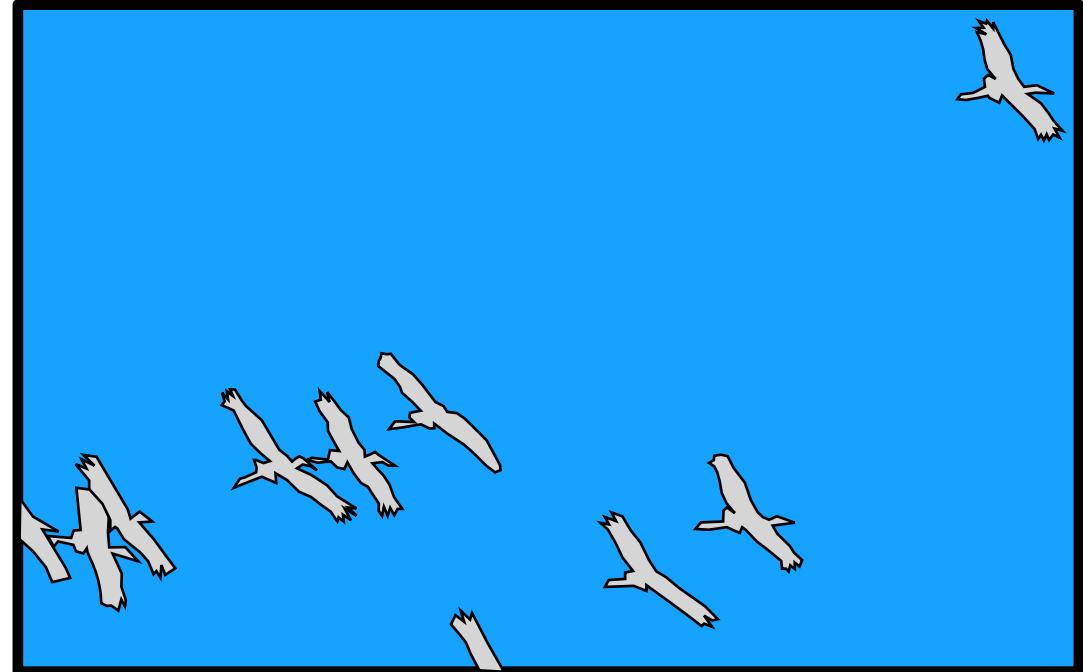
"Bird"



"Sky"



"Sky"

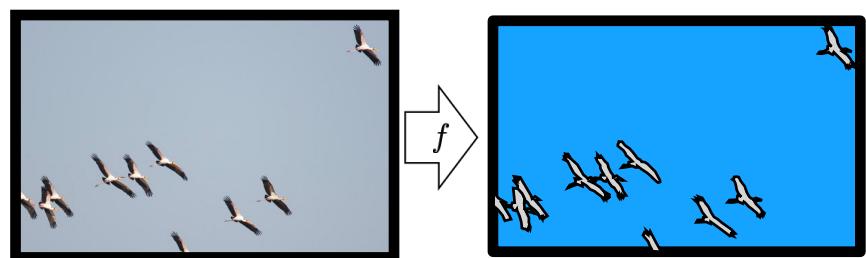
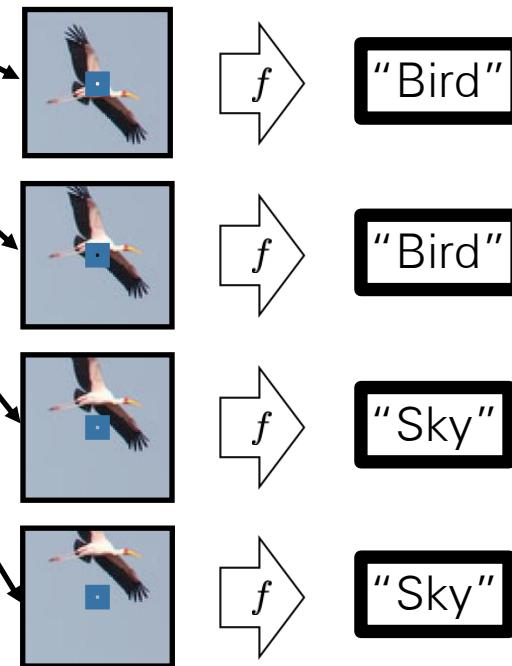


(Colors represent one-hot codes)

This problem is called **semantic segmentation**
in computer vision



What's the object class of the center pixel?

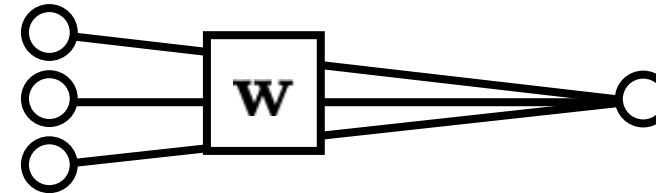


Translation invariance: process each patch in the same way.

An equivariant mapping:

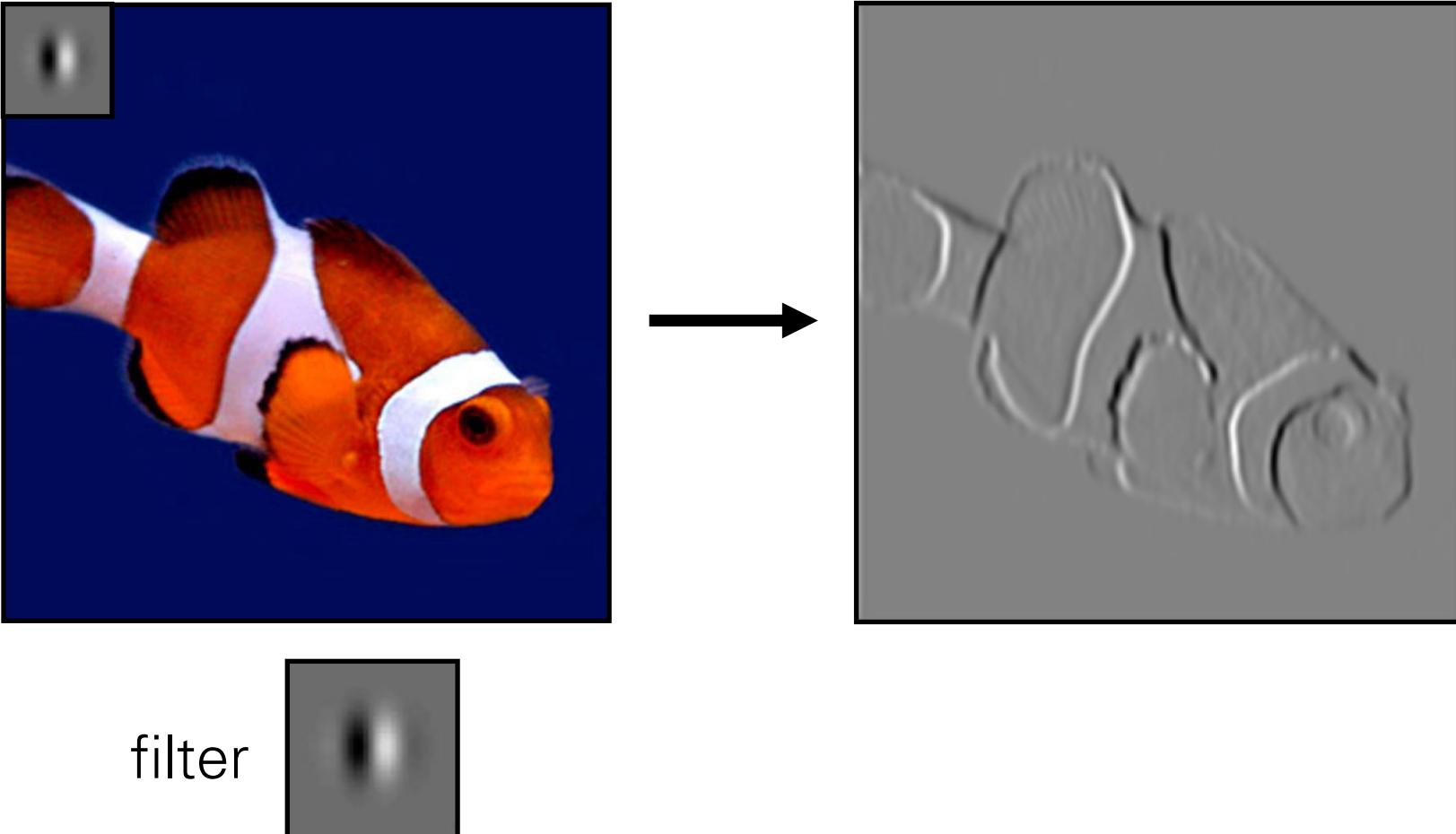
$$f(\text{translate}(x)) = \text{translate}(f(x))$$

W computes a weighted sum of all pixels in the patch



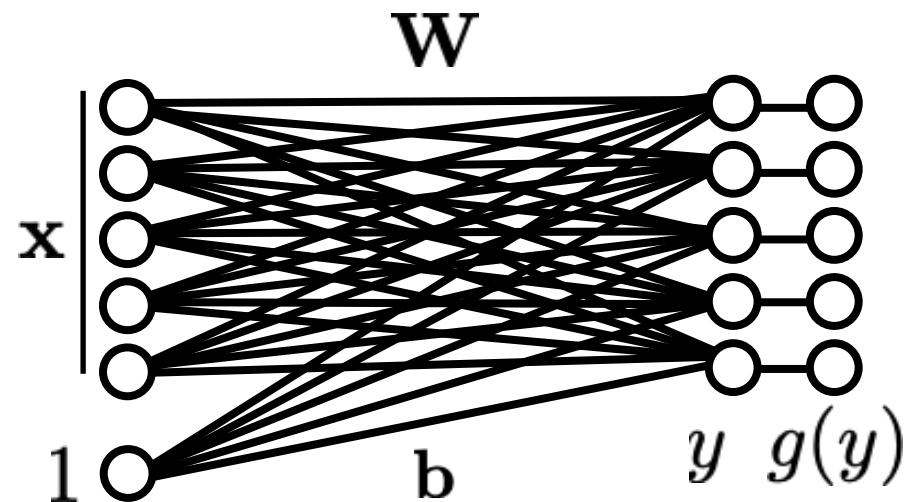
W is a convolutional kernel applied to the full image!

Convolution

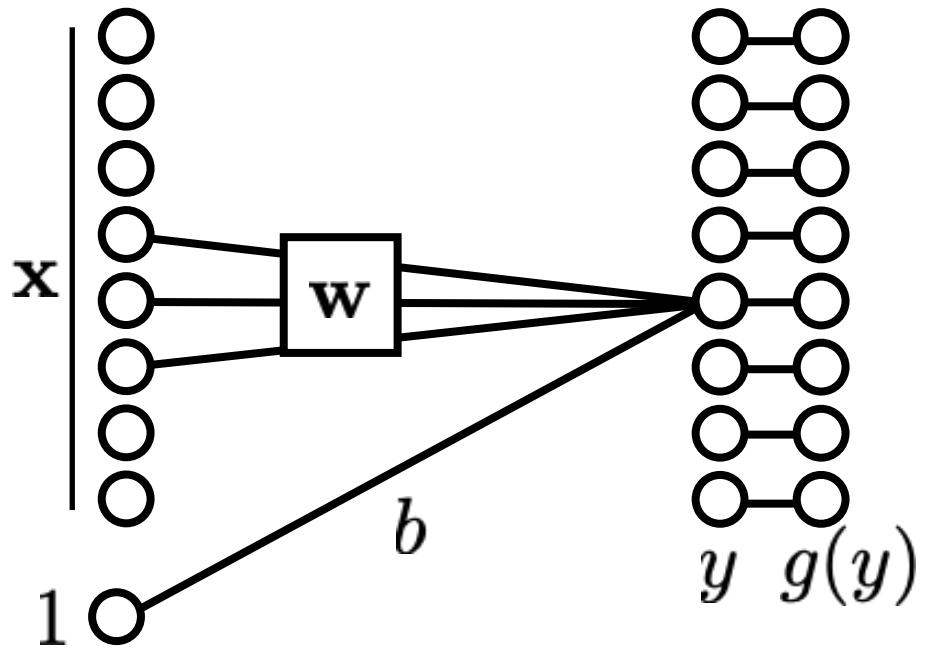


Fully-connected network

Fully-connected (fc) layer



Locally connected network

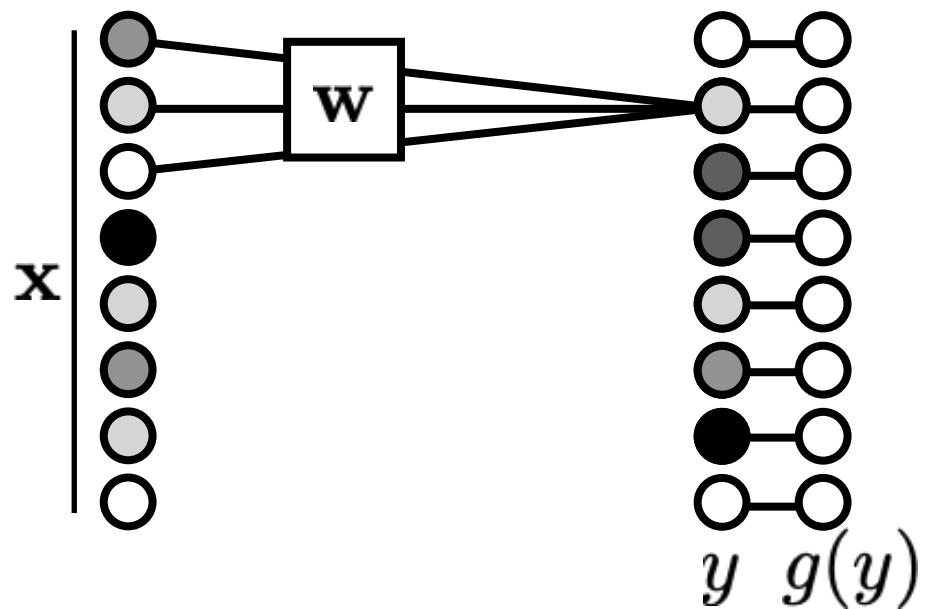


Often, we assume output is a **local** function of input.

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

Convolutional neural network

Conv layer

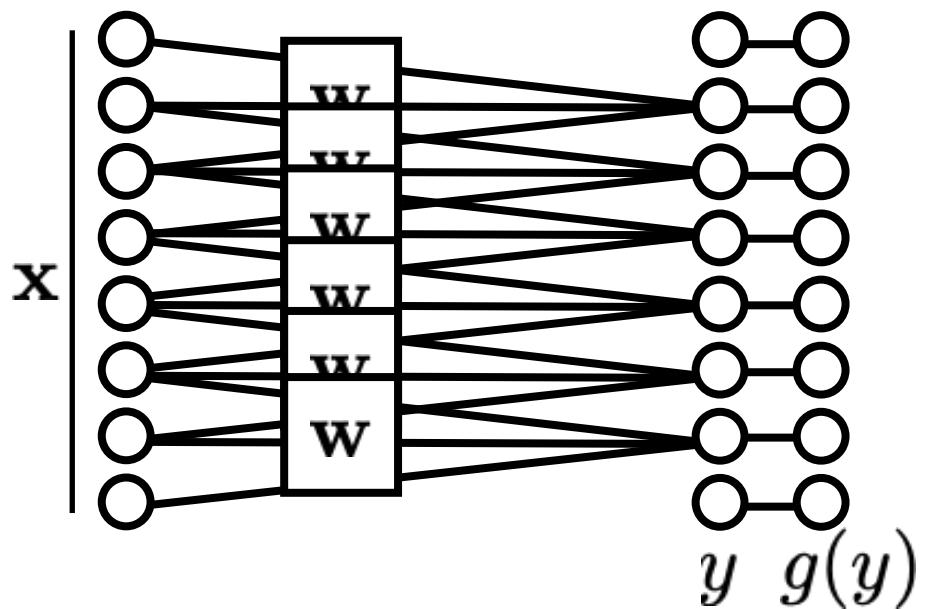


Often, we assume output is a **local** function of input.

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

Weight sharing

Conv layer



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.

Toeplitz matrix

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



1



*



$$\mathbf{x}^{(l)}$$

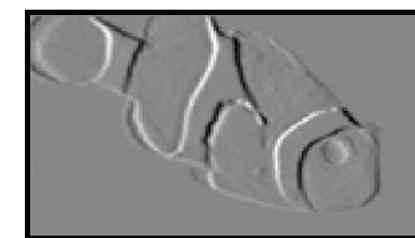
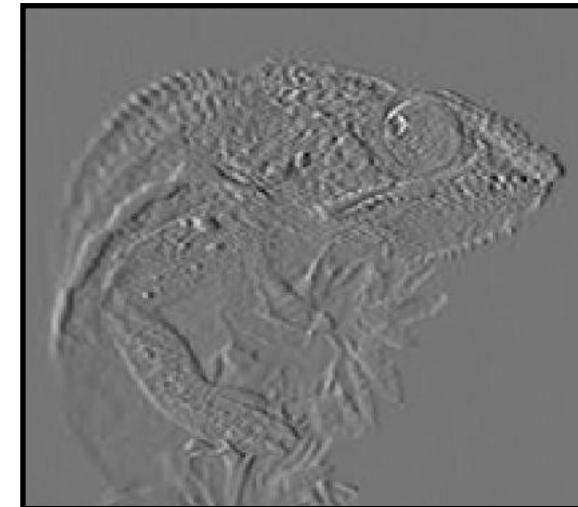
e.g., pixel image

- Constrained linear layer (infinitely strong regularization)
 - Fewer parameters —> easier to learn, less overfitting

$$\begin{array}{c} \boxed{} \\ = \\ \boxed{} * \end{array} \quad \begin{matrix} \text{Image} \\ \text{Matrix} \end{matrix} \quad \begin{array}{c} \boxed{} \\ \mathbf{x}^{(l+1)} \\ \mathbf{x}^{(l)} \end{array}$$

$$\mathbf{x}^{(l+1)} = \mathbf{W} * \mathbf{x}^{(l)}$$

The diagram illustrates a convolutional operation. On the left, a vertical vector $\mathbf{x}^{(l+1)}$ is shown. To its left is an equals sign ($=$). To the right of the equals sign is a central convolutional step. This step shows a 5x5 kernel (the weight matrix \mathbf{W}) being applied to a 5x5 input patch (the feature map $\mathbf{x}^{(l)}$). The result is a 1x5 output vector. On the far right, another vertical vector $\mathbf{x}^{(l)}$ is shown, preceded by a asterisk (*) symbol.



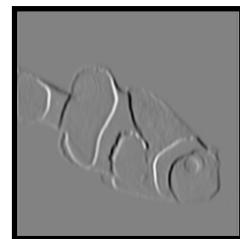
Conv layers can be applied to arbitrarily-sized inputs

Five views on convolutional layers

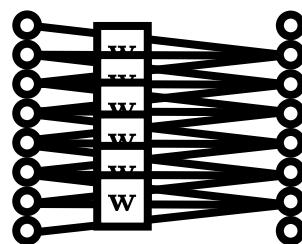
1. Equivariant with translation (stationarity) $f(\text{translate}(x)) = \text{translate}(f(x))$

2. Patch processing (Markov assumption)

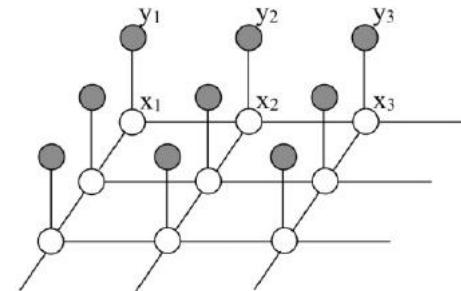
3. Image filter



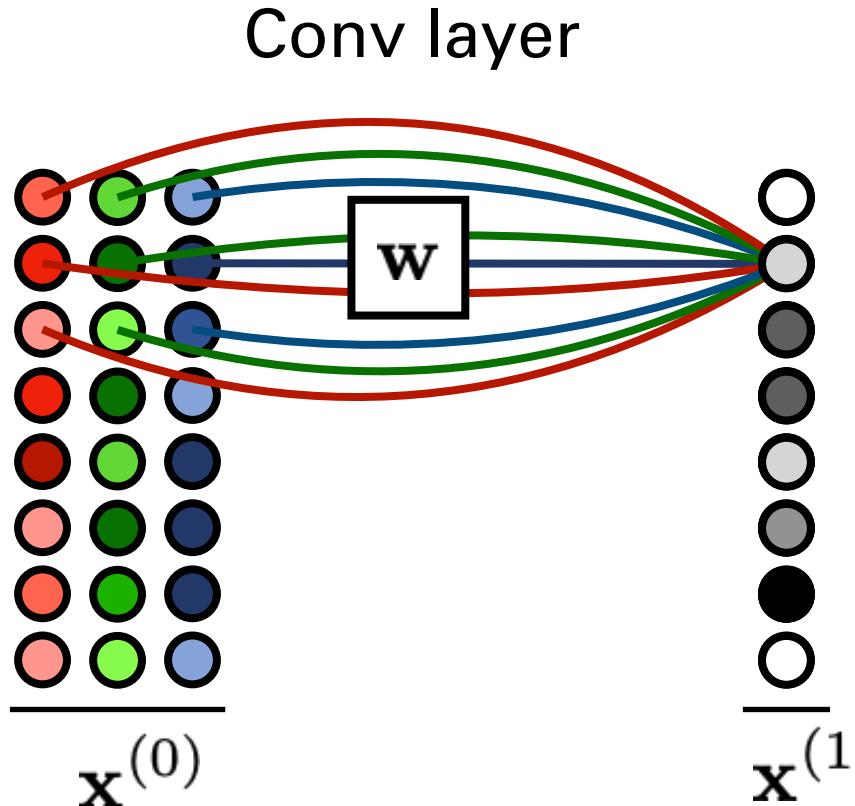
4. Parameter sharing



5. A way to process variable-sized tensors

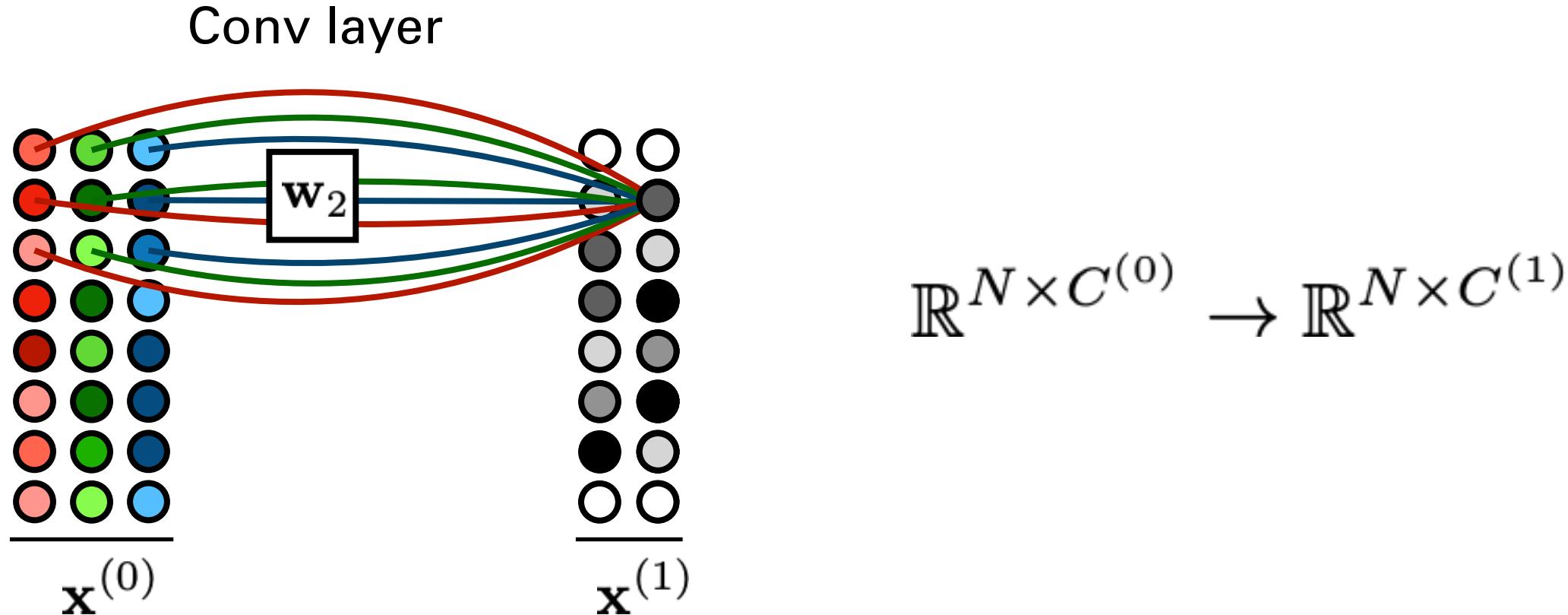


Multiple channels



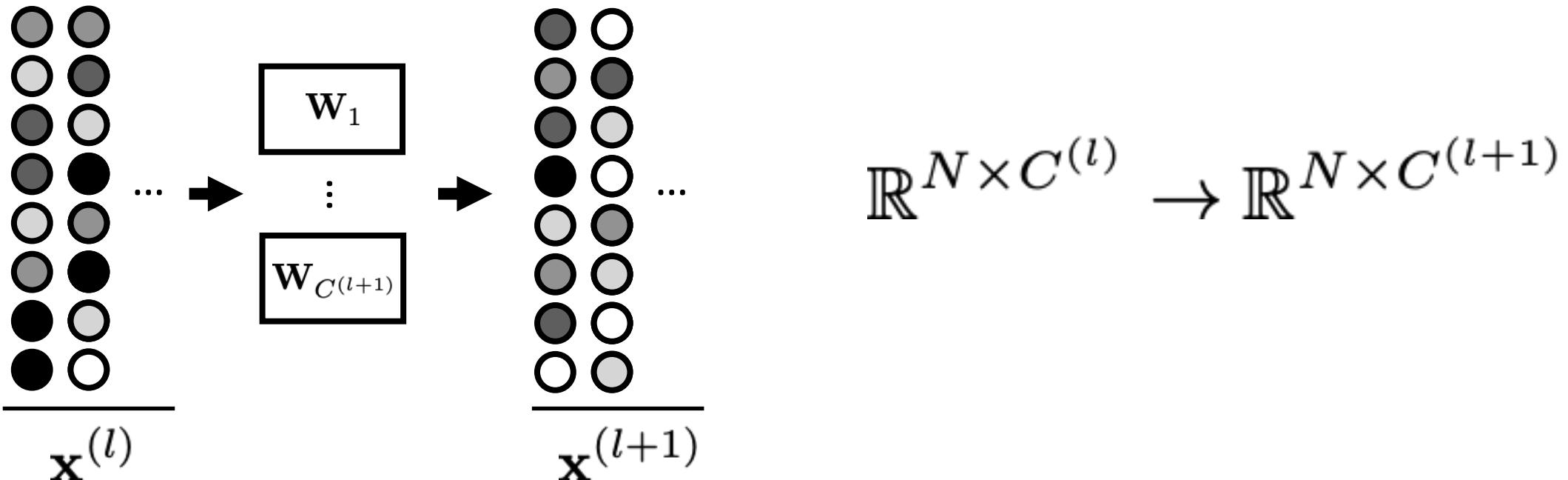
$$\mathbb{R}^{N \times C} \rightarrow \mathbb{R}^{N \times 1}$$

Multiple channels



Multiple channels

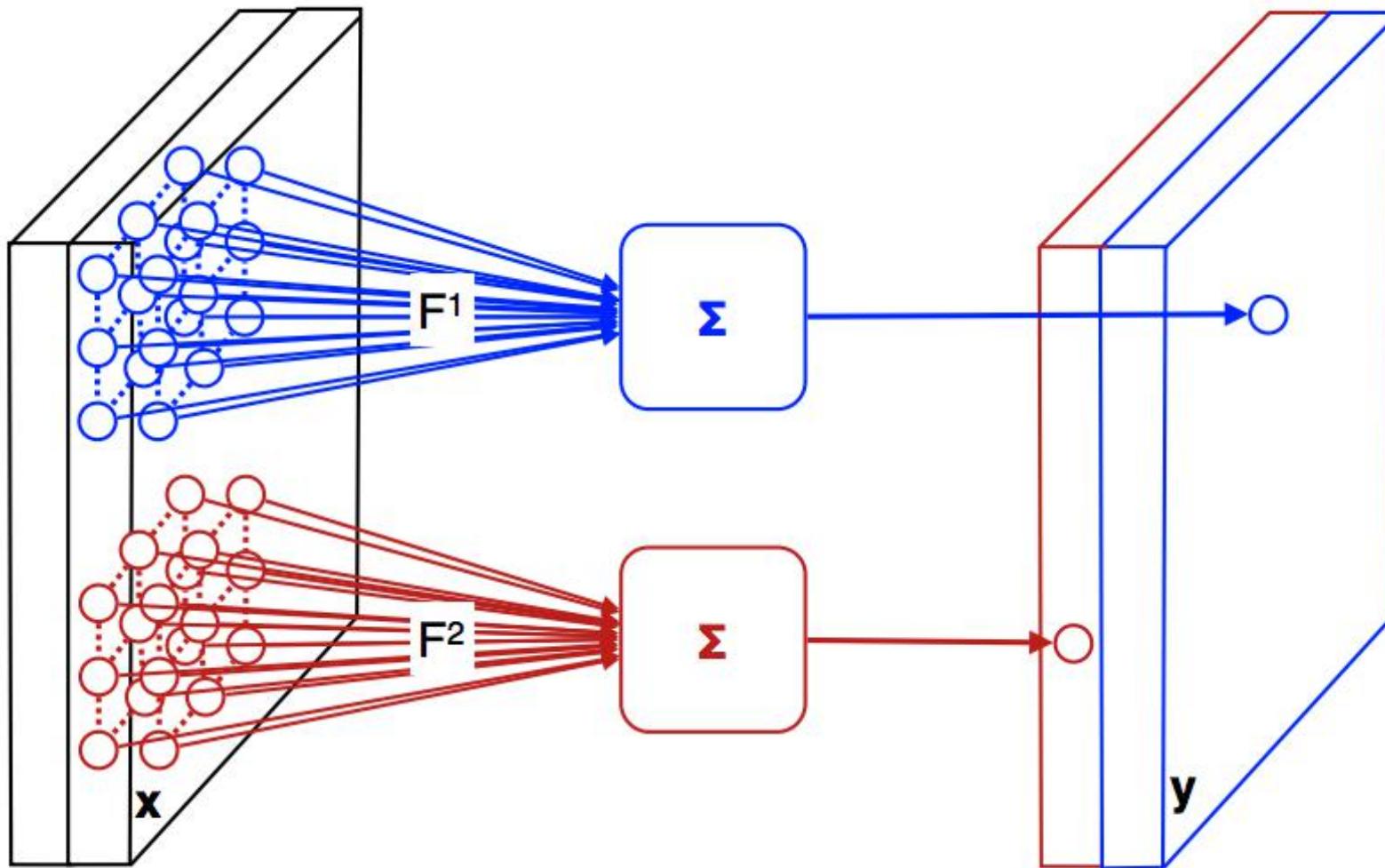
Conv layer



Input features

A bank of 2 filters

2-dimensional
output features

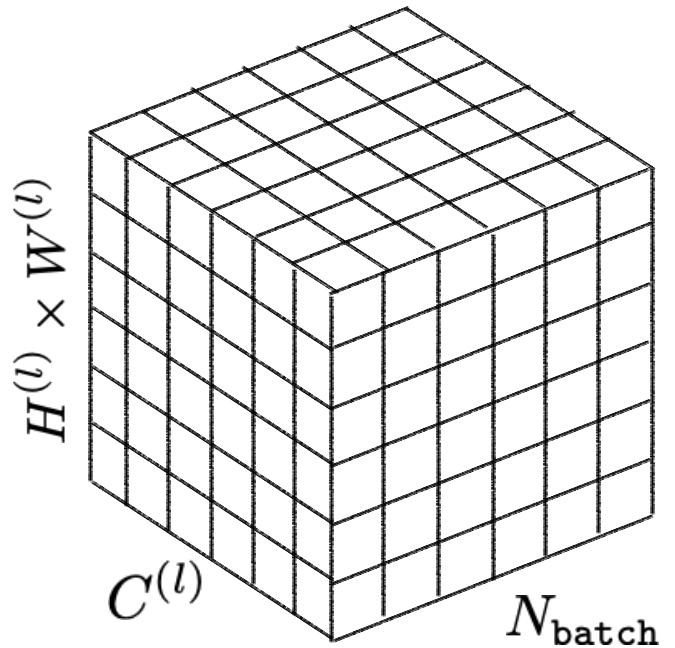


$$\mathbb{R}^{H \times W \times C^{(l)}} \rightarrow \mathbb{R}^{H \times W \times C^{(l+1)}}$$

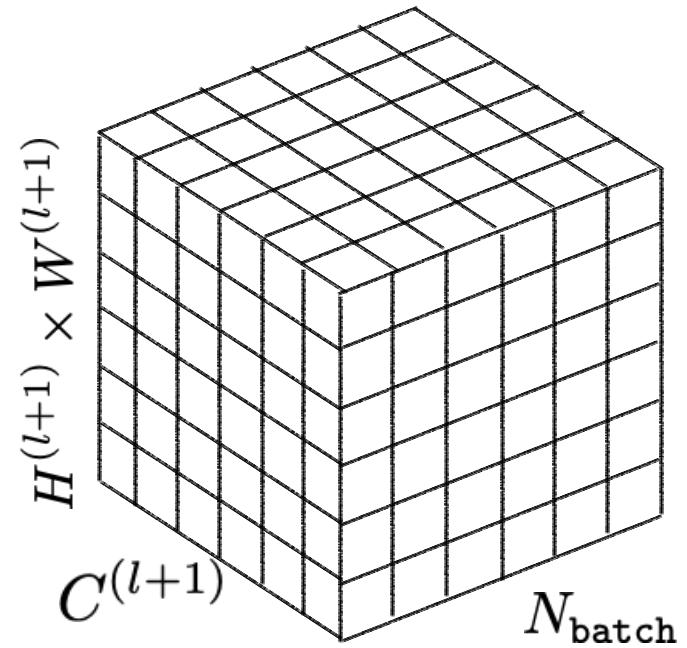
[Figure from Andrea Vedaldi]

"Tensor flow"

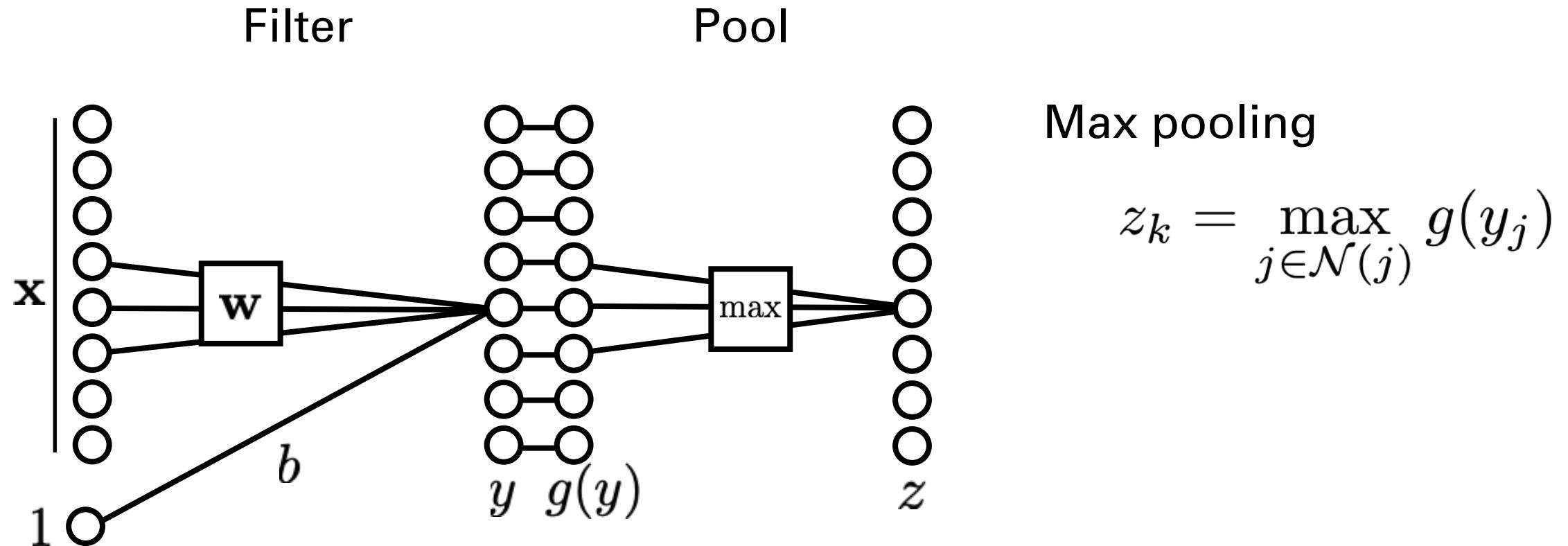
$$\mathbf{x}^{(l)} \in \mathbb{R}^{N_{\text{batch}} \times H^{(l)} \times W^{(l)} \times C^{(l)}}$$



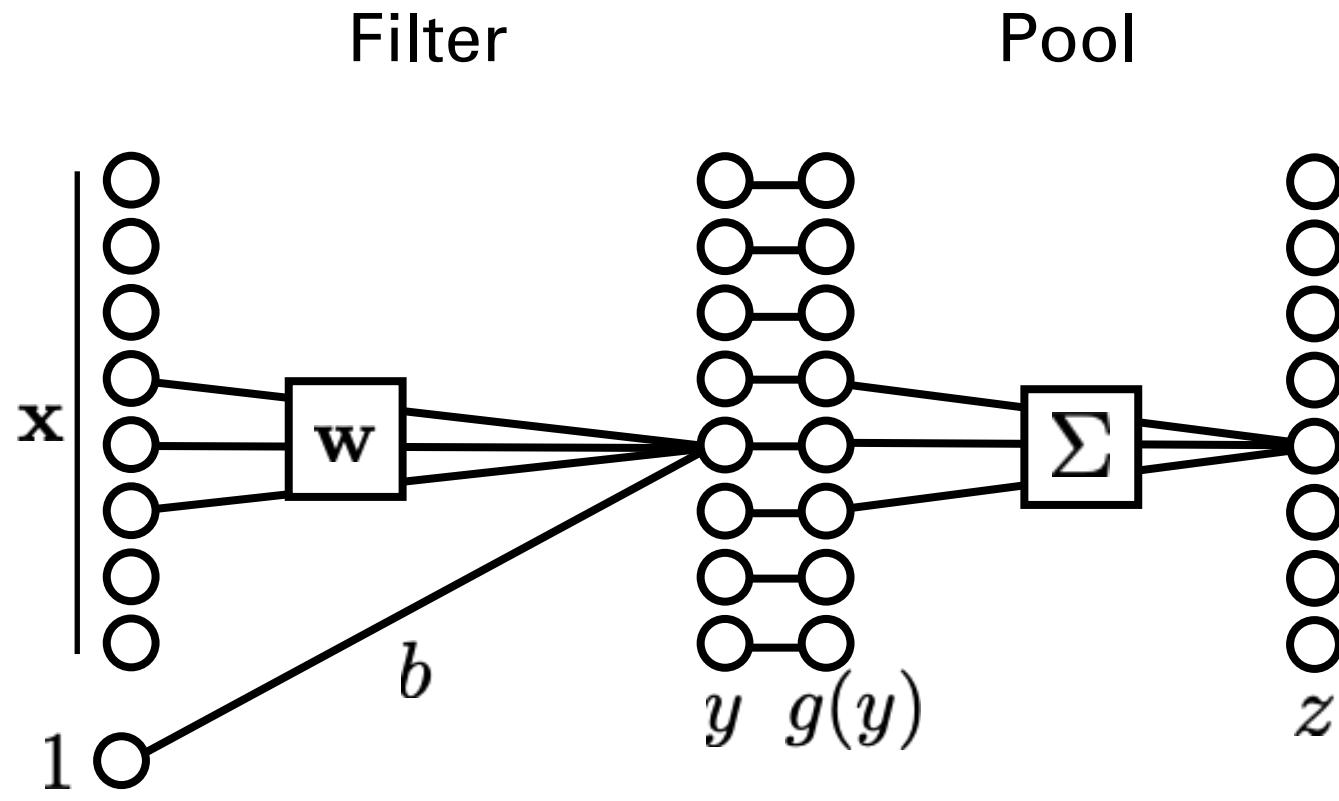
$$\mathbf{x}^{(l+1)} \in \mathbb{R}^{N_{\text{batch}} \times H^{(l+1)} \times W^{(l+1)} \times C^{(l+1)}}$$



Pooling



Pooling



Max pooling

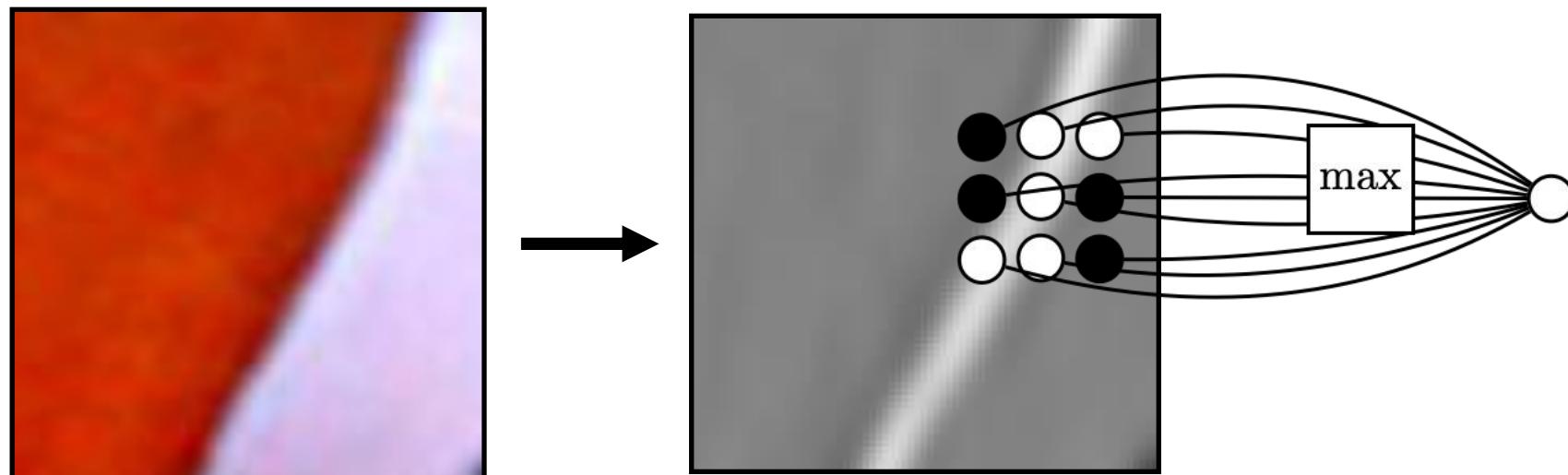
$$z_k = \max_{j \in \mathcal{N}(j)} g(y_j)$$

Mean pooling

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

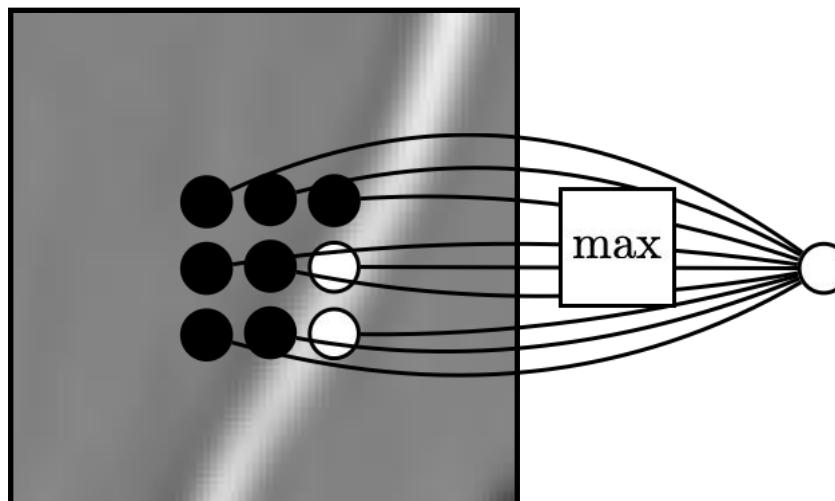
Pooling – Why?

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



Pooling – Why?

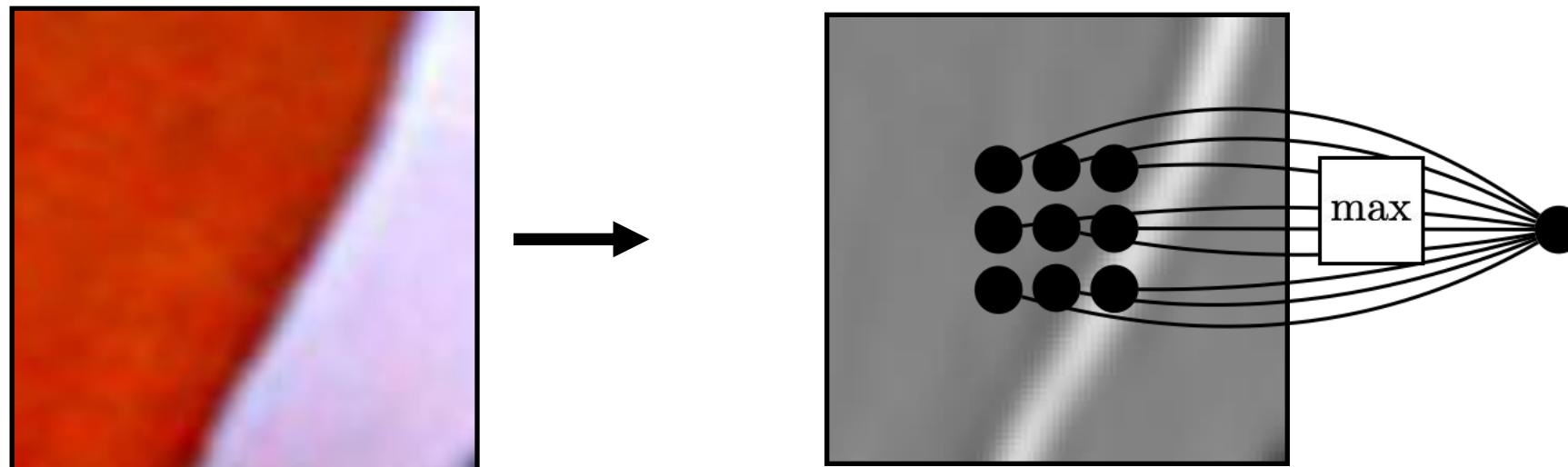
Pooling across spatial locations achieves stability w.r.t. small translations:



large response
regardless of exact
position of edge

Pooling – Why?

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



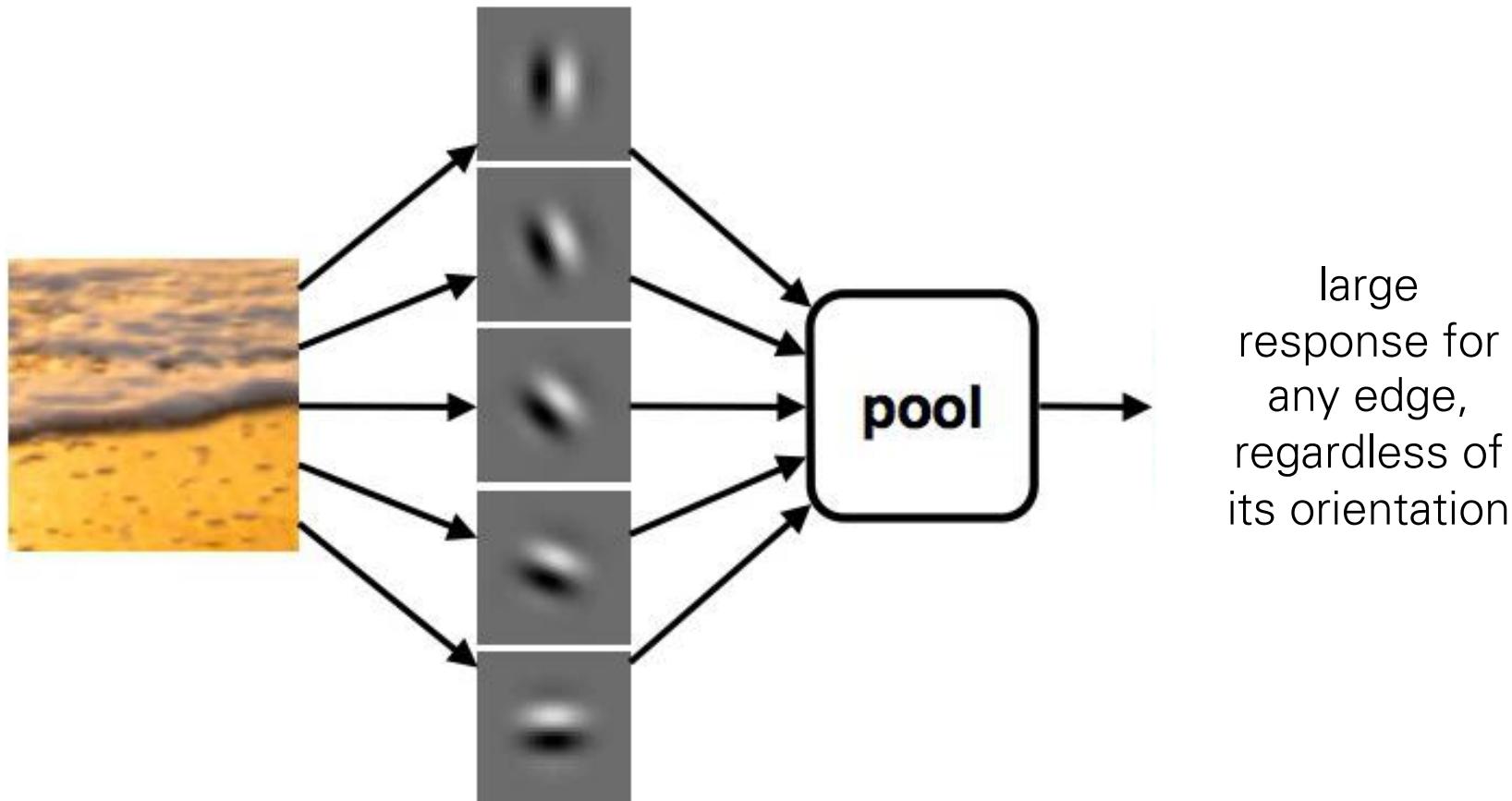
CNNs are stable w.r.t. diffeomorphisms



[“Unreasonable effectiveness of Deep Features as a Perceptual Metric”, Zhang et al. 2018]

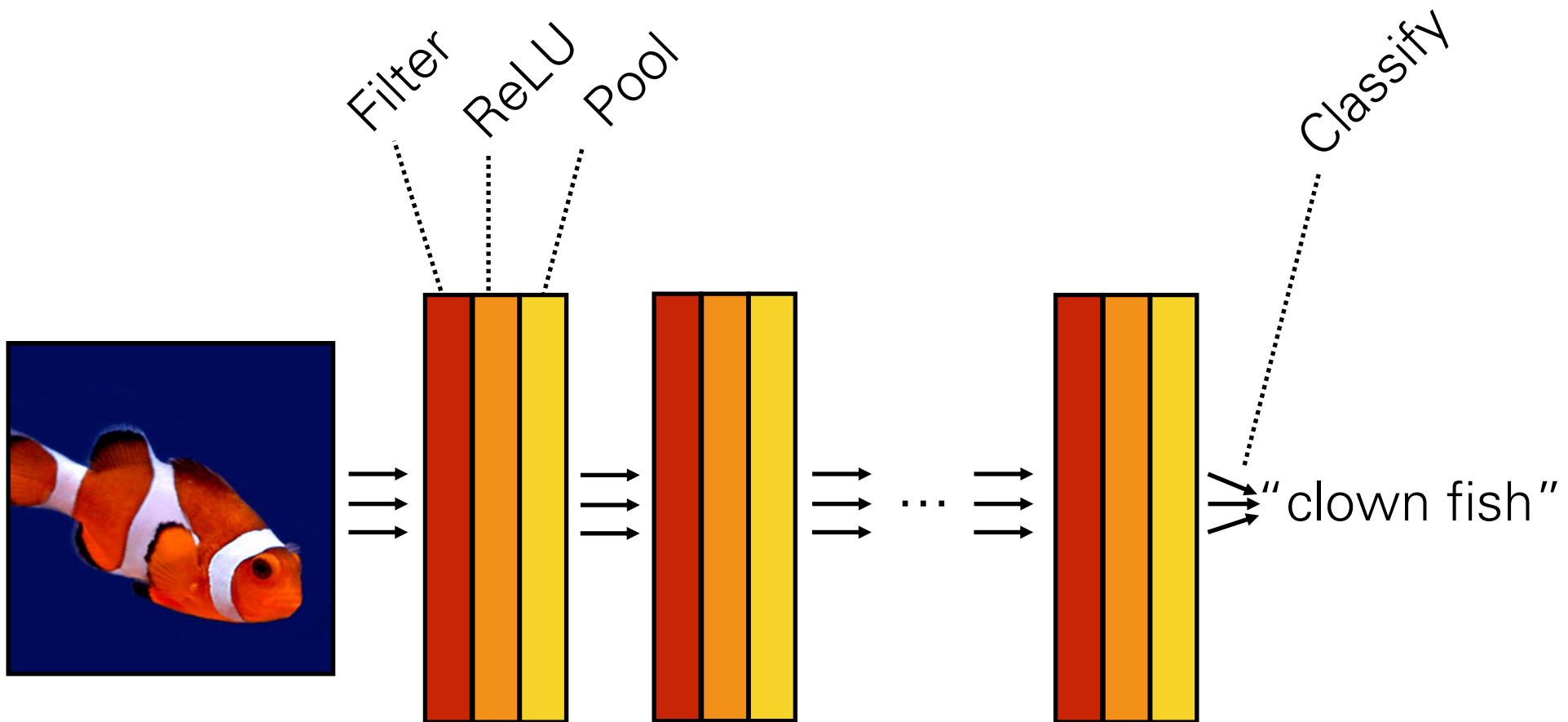
Pooling – Why?

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

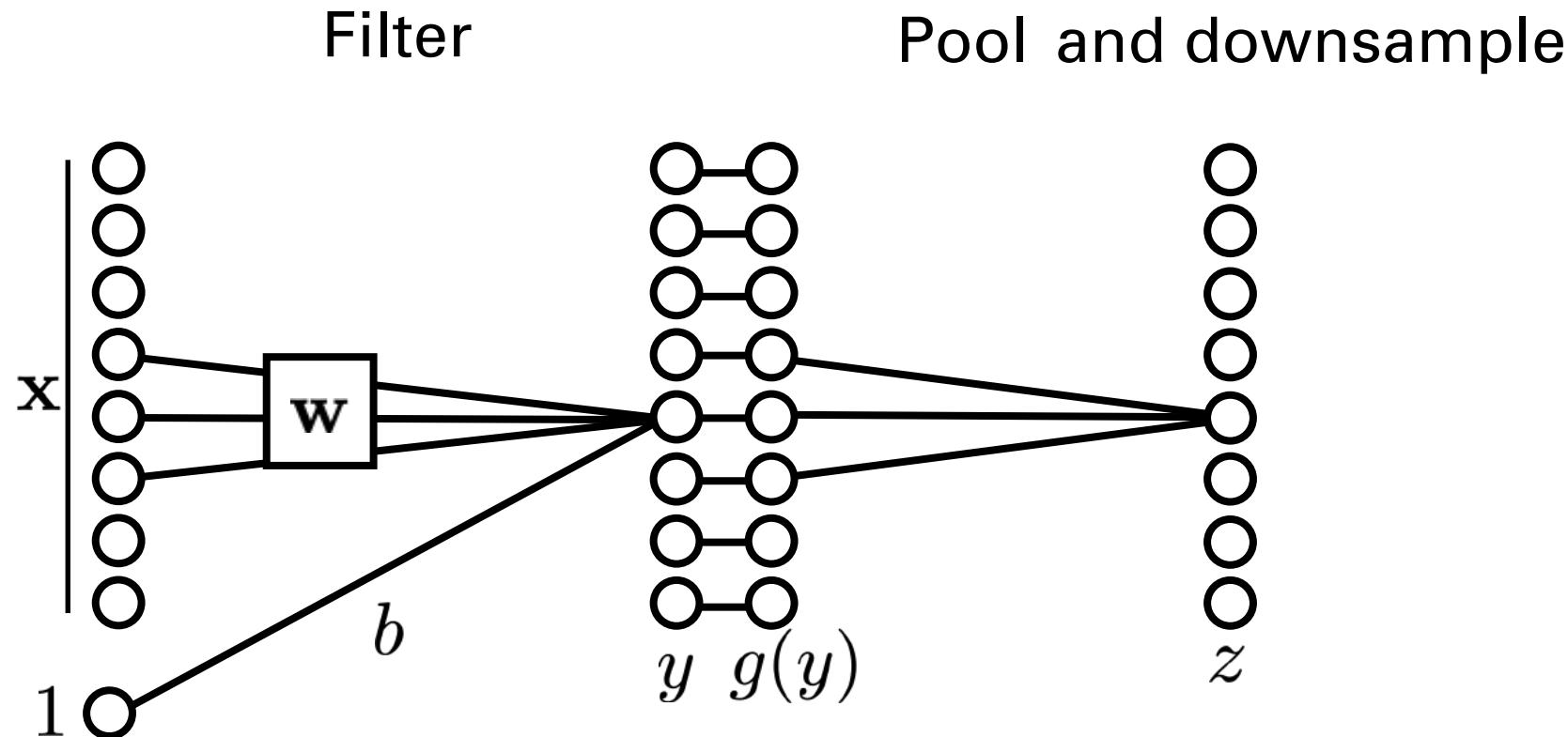


[Derived from slide by Andrea Vedaldi]

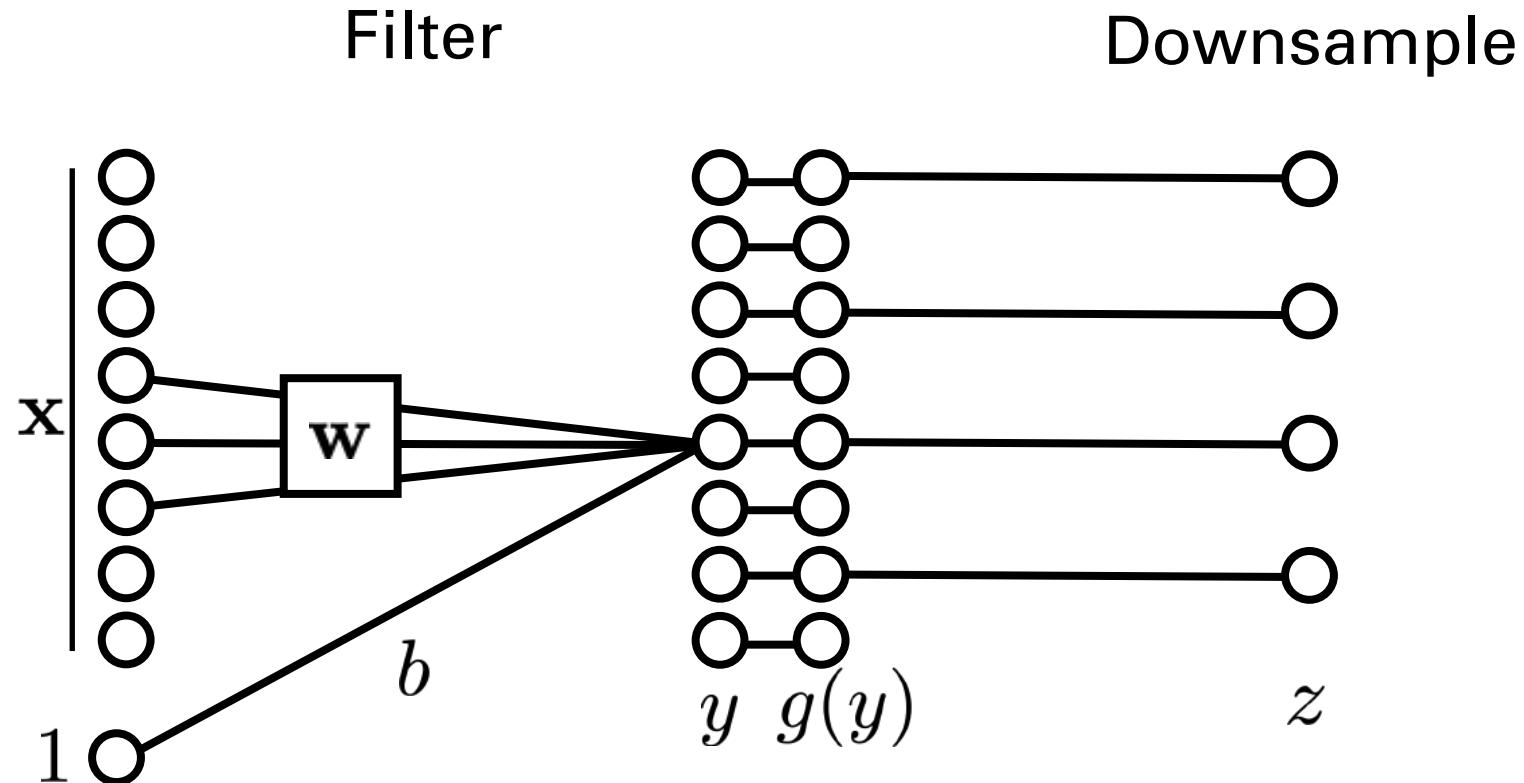
Computation in a neural net



Downsampling

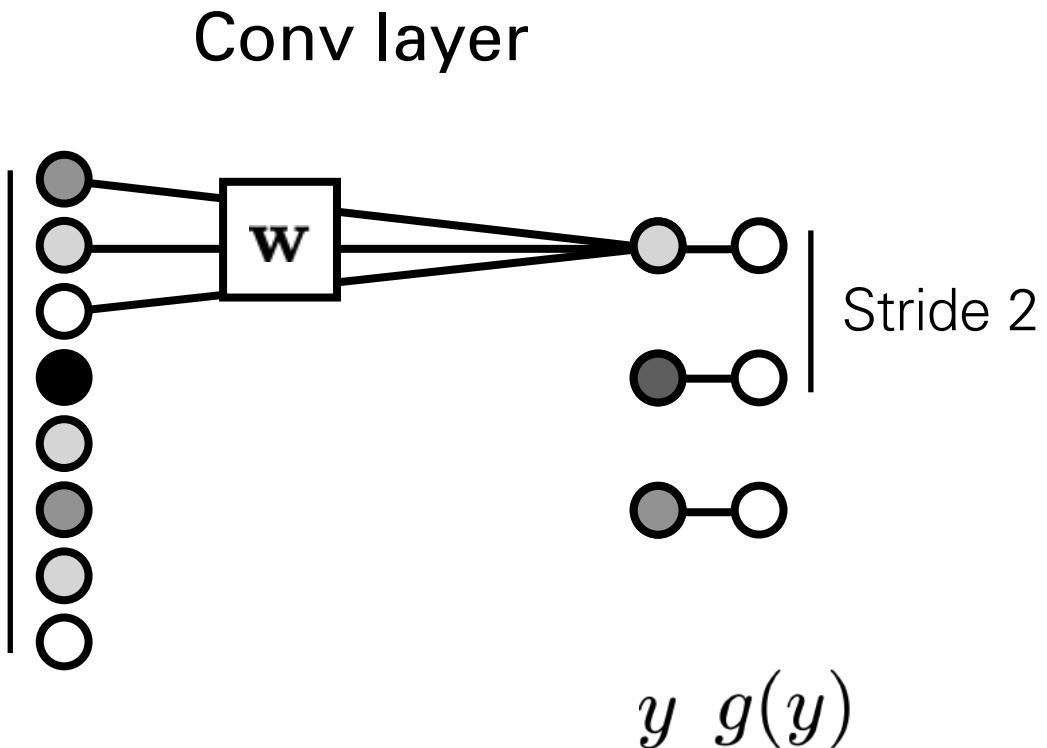


Downsampling



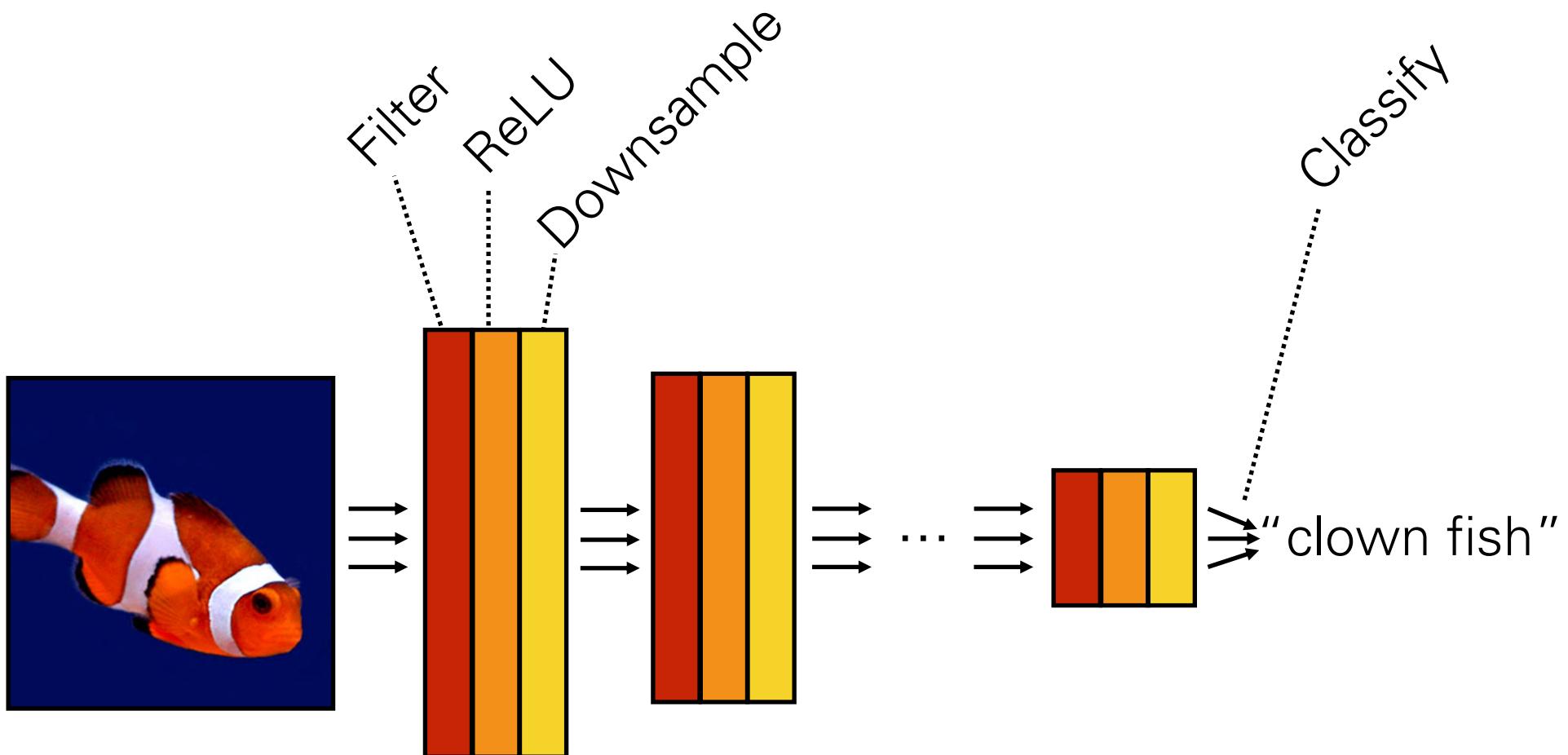
$$\mathbb{R}^{H^{(l)} \times W^{(l)} \times C^{(l)}} \rightarrow \mathbb{R}^{H^{(l+1)} \times W^{(l+1)} \times C^{(l+1)}}$$

Strided convolution



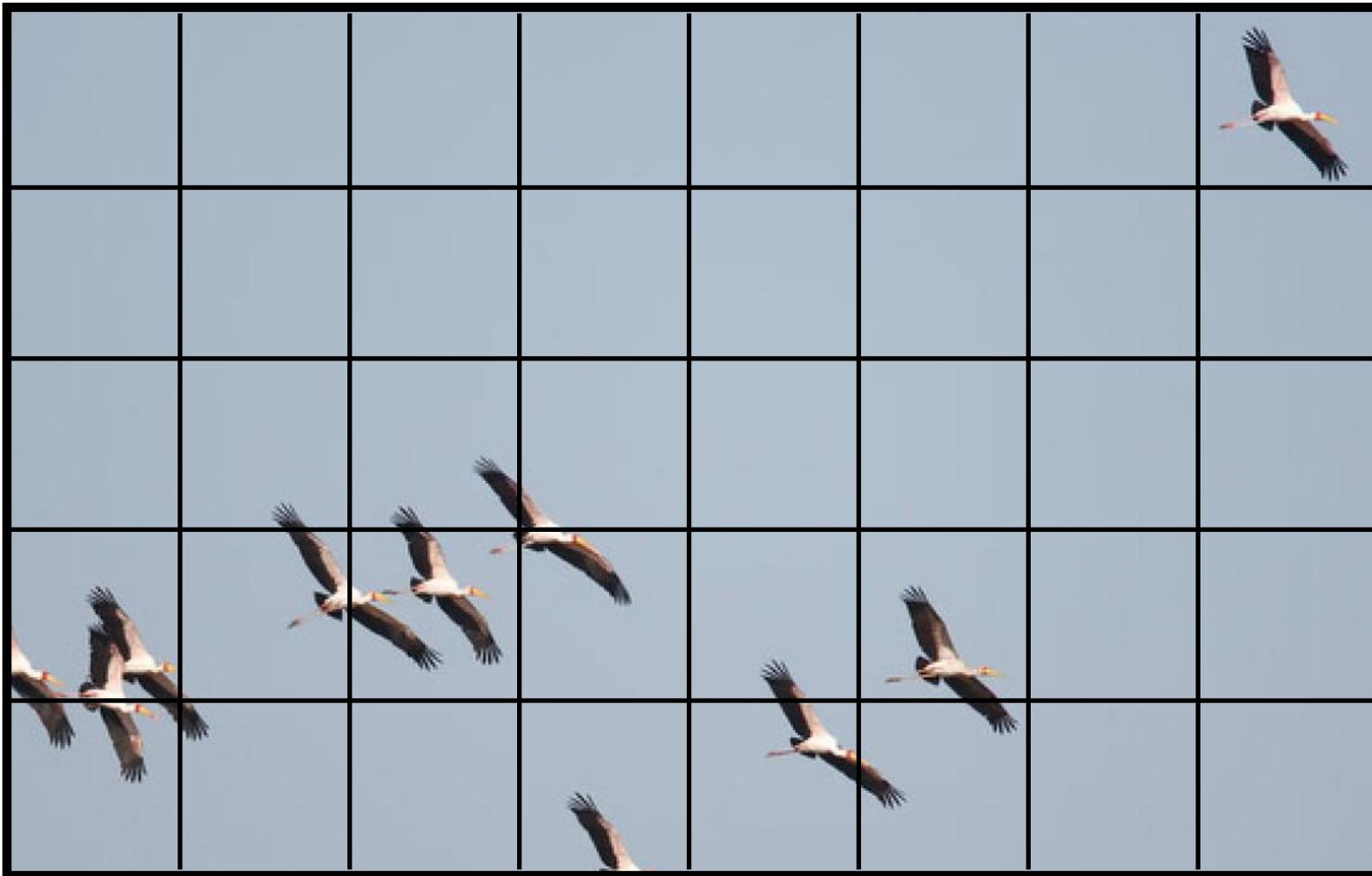
Strided convolutions combine convolution and downsampling into a single operation.

Computation in a neural net

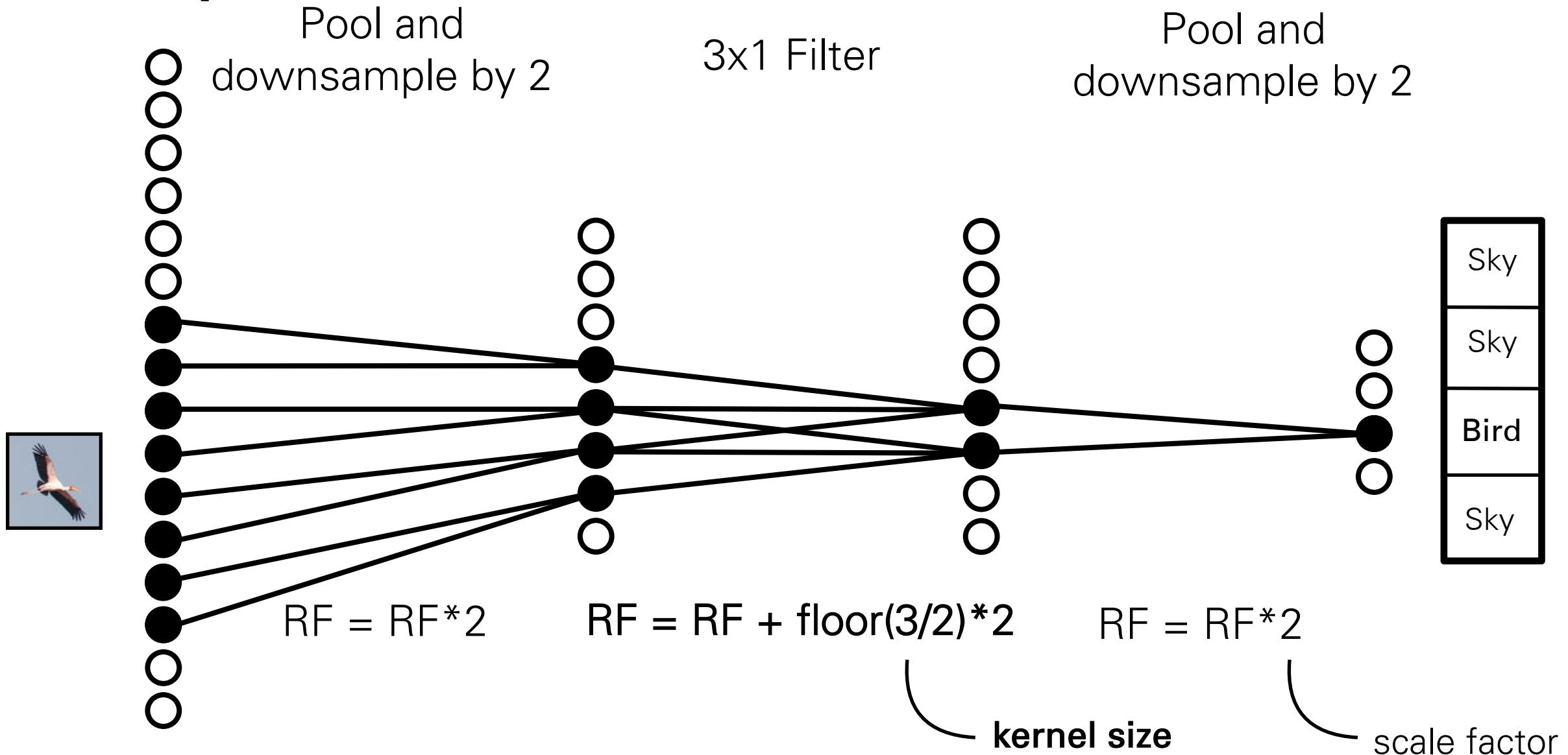


$$f(\mathbf{x}) = f_L(\dots(f_2(f_1(\mathbf{x}))))$$

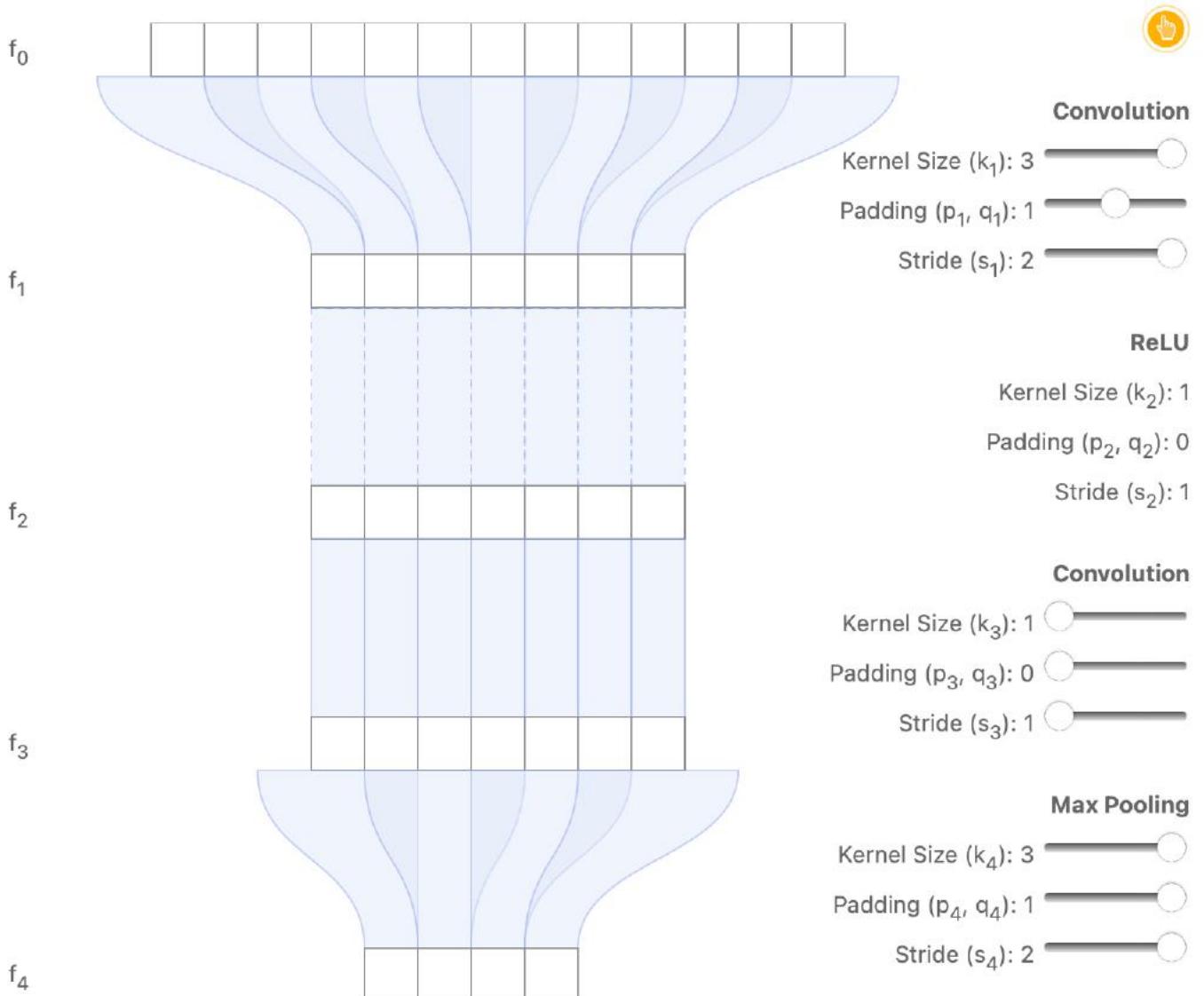
Receptive fields



Receptive fields



Our notation is further illustrated with the simple network below. In this case, $L = 4$ and the model consists of a convolution, followed by ReLU, a second convolution and max-pooling.²



Why CNNs?

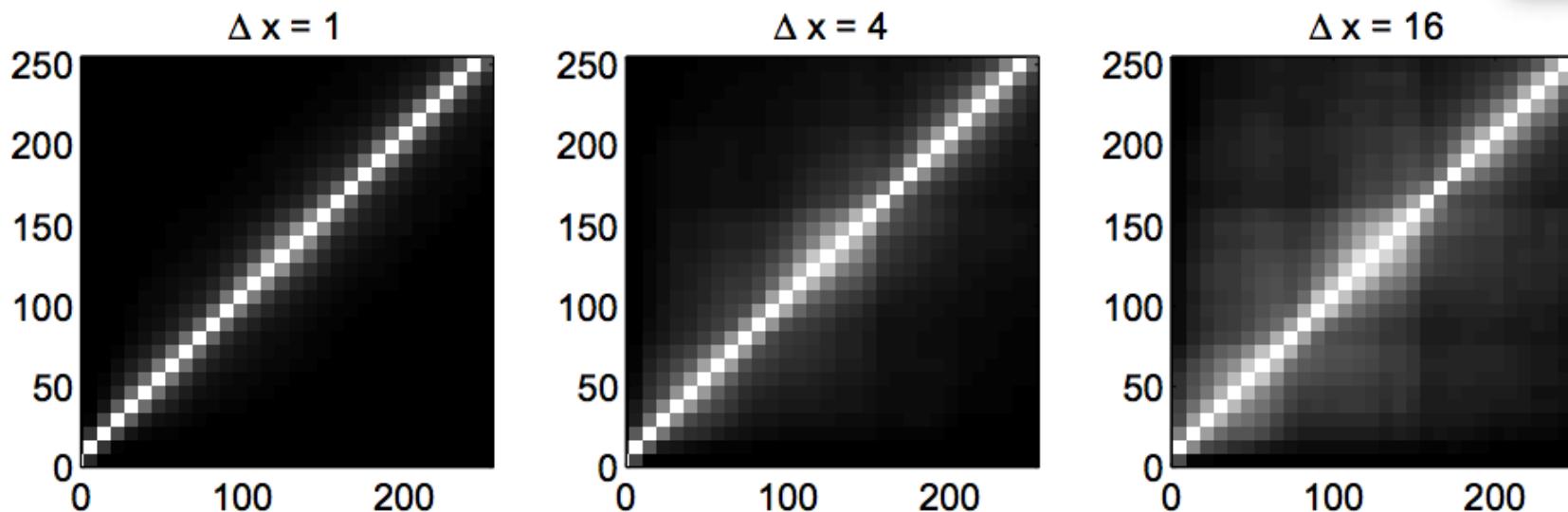
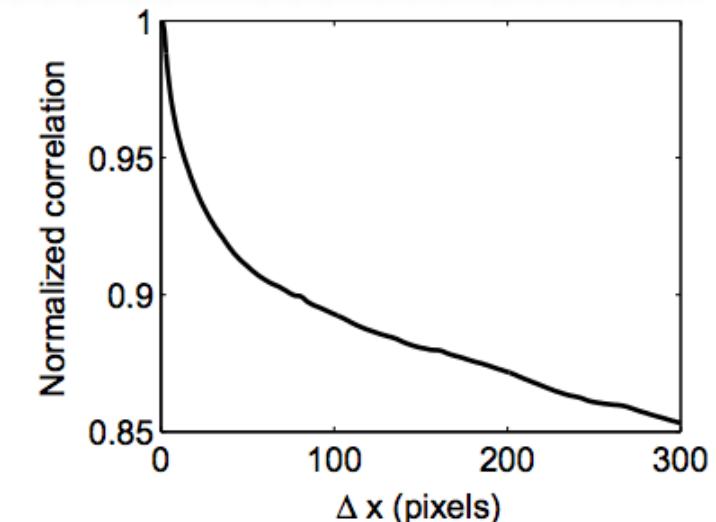


Fig. 1. (a) Scatterplots of pairs of pixels at three different spatial displacements, averaged over five examples images. (b) Autocorrelation function. Photographs are of New York City street scenes, taken with a Canon 10D digital camera, and processed in RAW linear sensor mode (producing pixel intensities are in roughly proportional to light intensity). Correlations were computed on the logs of these sensor intensity values [41].

The screenshot shows a PDF document window titled "simoncelli05a-preprint.pdf" with "Page 1 of 13". The document header includes "To appear in: Handbook of Video and Image Processing, 2nd edition" and "ed. Alan Bovik, @Academic Press, 2005". The main title is "4.7 Statistical Modeling of Photographic Images" and the author is "Eero P. Simoncelli" from "New York University". The date is "January 18, 2005". The text discusses the nonuniform probability distribution of visual images and its exploitation in image processing and machine vision. It also mentions how to build and test a probability model for images, referencing an empirically-driven methodology based on the study of discretized (pixelated) images.



Why CNNs?

Statistical dependences between pixels decay as a power law of distance between the pixels.

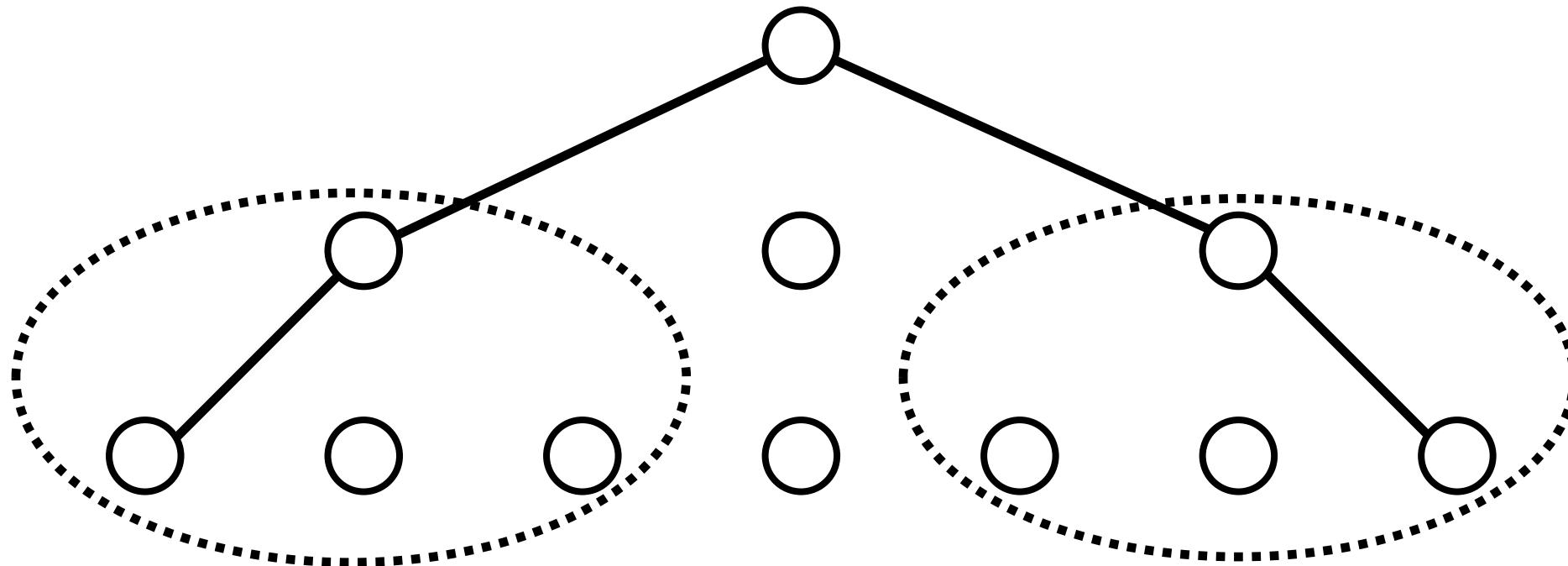
It is therefore often sufficient to model local dependences only. → **Convolution**

More generally, we should allocate parameters that model dependences in proportion to the strength of those dependences. → **Multiscale, hierarchical representations**

[For more discussion, see “Why does Deep and Cheap Learning Work So Well?”, Lin et al. 2017]

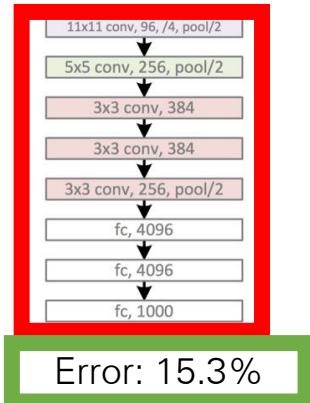
Why CNNs?

Capturing long-range dependences:



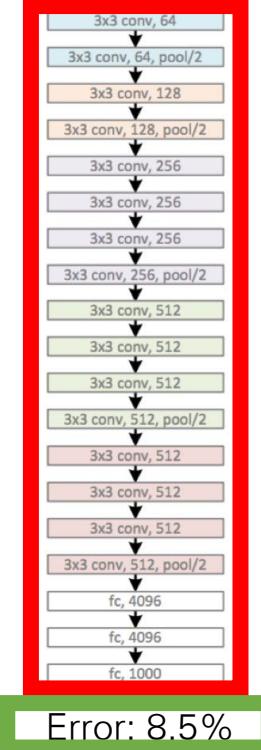
Deep Neural Networks for Visual Recognition

2012: AlexNet
5 conv. layers



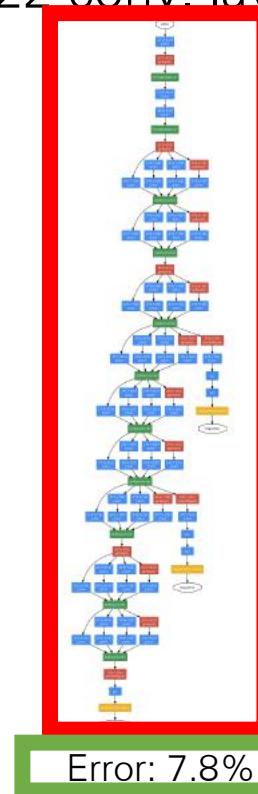
Error: 15.3%

2014: VGG
16 conv. layers



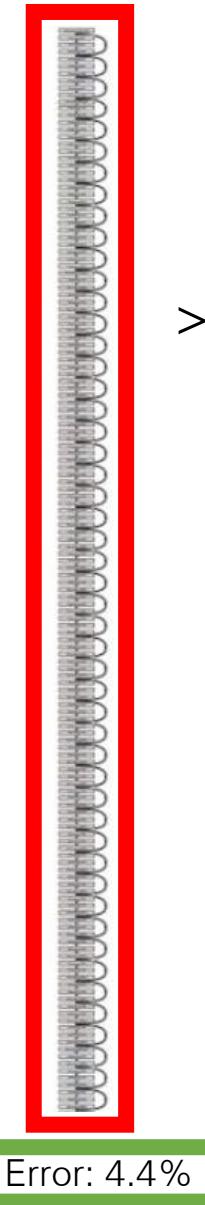
Error: 8.5%

2015: GoogLeNet
22 conv. layers



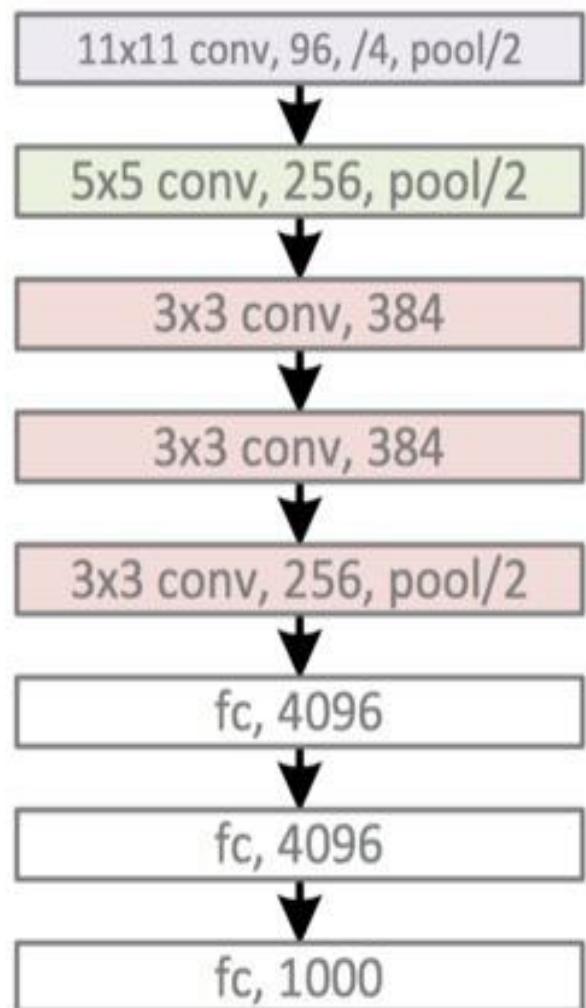
Error: 7.8%

2016: ResNet
>100 conv. layers



Error: 4.4%

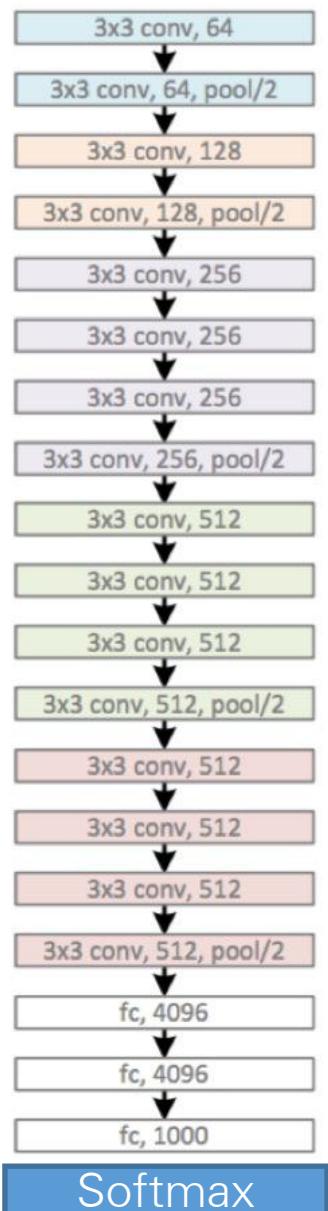
2012: AlexNet
5 conv. layers



Error: 15.3%

VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION

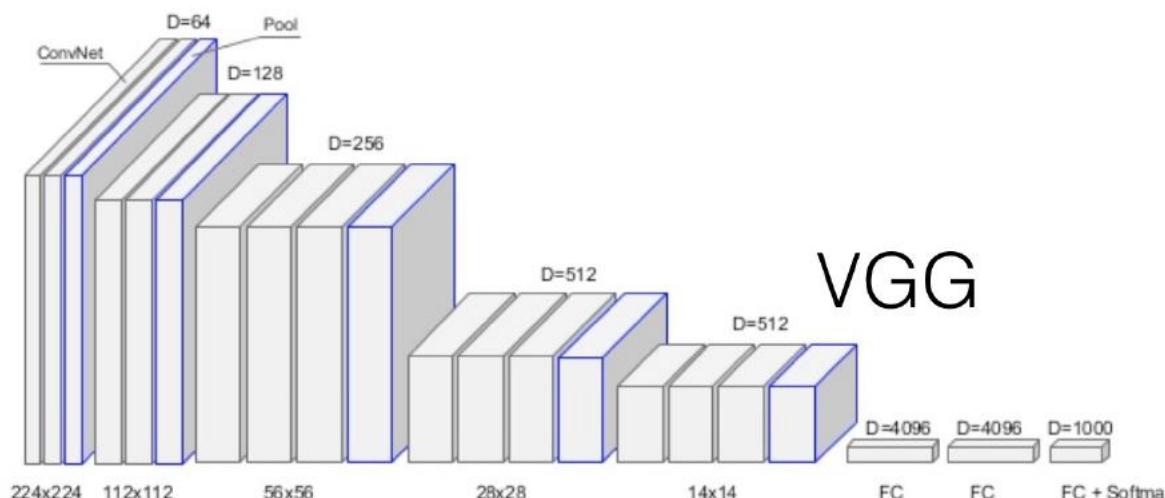
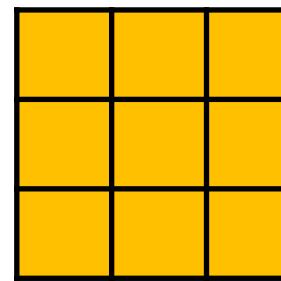
2014: VGG
16 conv.
layers



Error: 8.5%

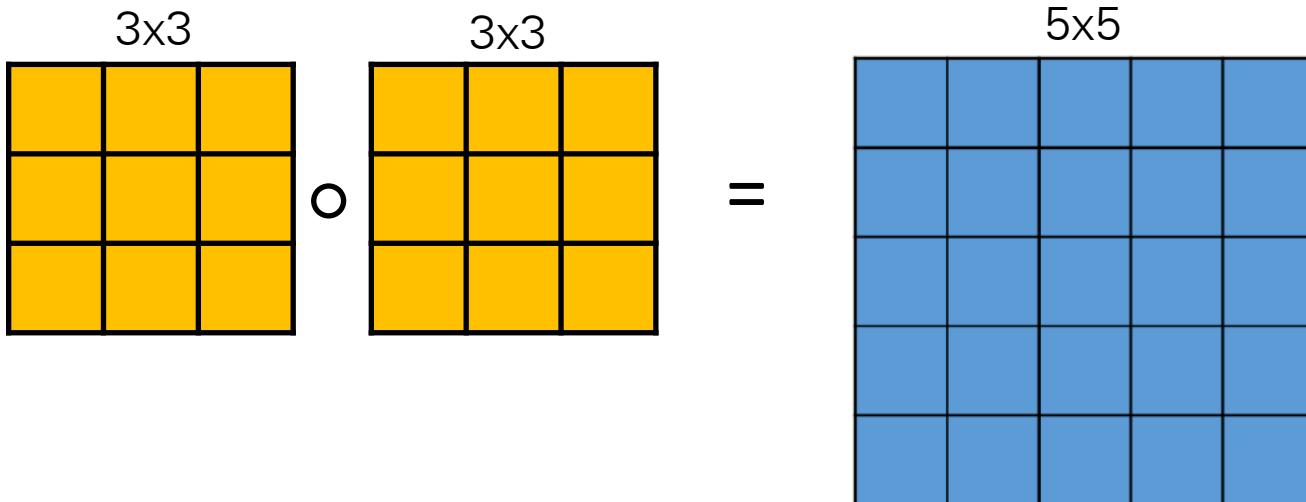
<https://arxiv.org/pdf/1409.1556.pdf>

Small convolutional kernels: 3x3
ReLU non-linearities
>100 million parameters.

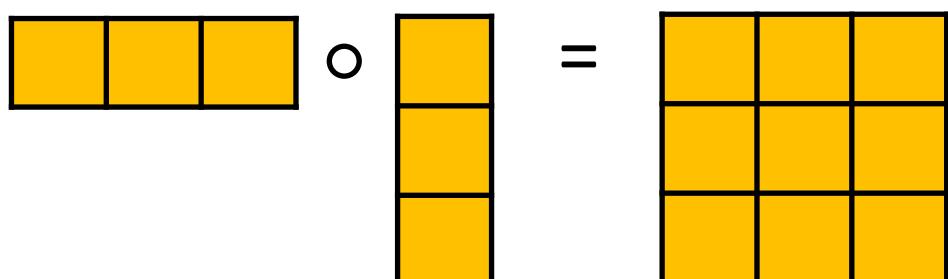


VGG

Chaining convolutions

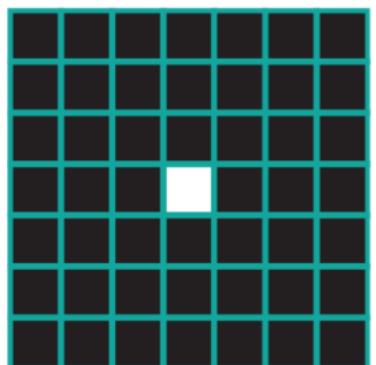


25 coefficients, but only
18 degrees of freedom

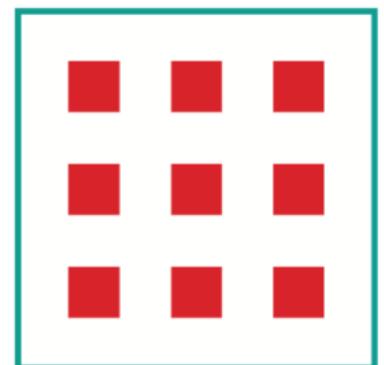


9 coefficients, but only
6 degrees of freedom.
Only separable filters... would this be enough?

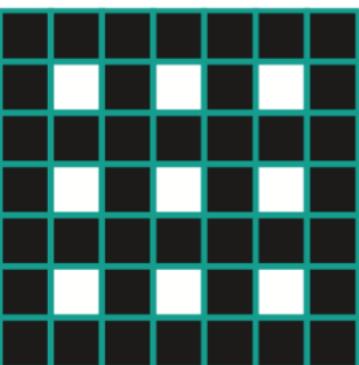
Dilated convolutions



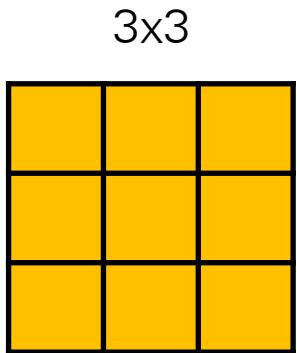
(a) Input



(b) Dilation 2



(c) Output



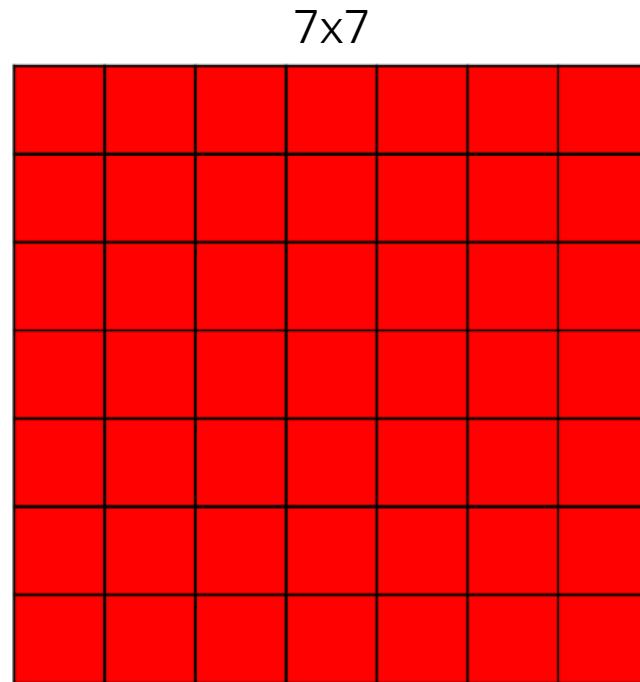
3x3

O

5x5				
a	0	b	0	c
0	0	0	0	0
d	0	e	0	f
0	0	0	0	0
g	0	h	0	i

25 coefficients
9 degrees of freedom

=



49 coefficients
18 degrees of freedom

[<https://arxiv.org/pdf/1511.07122.pdf>]

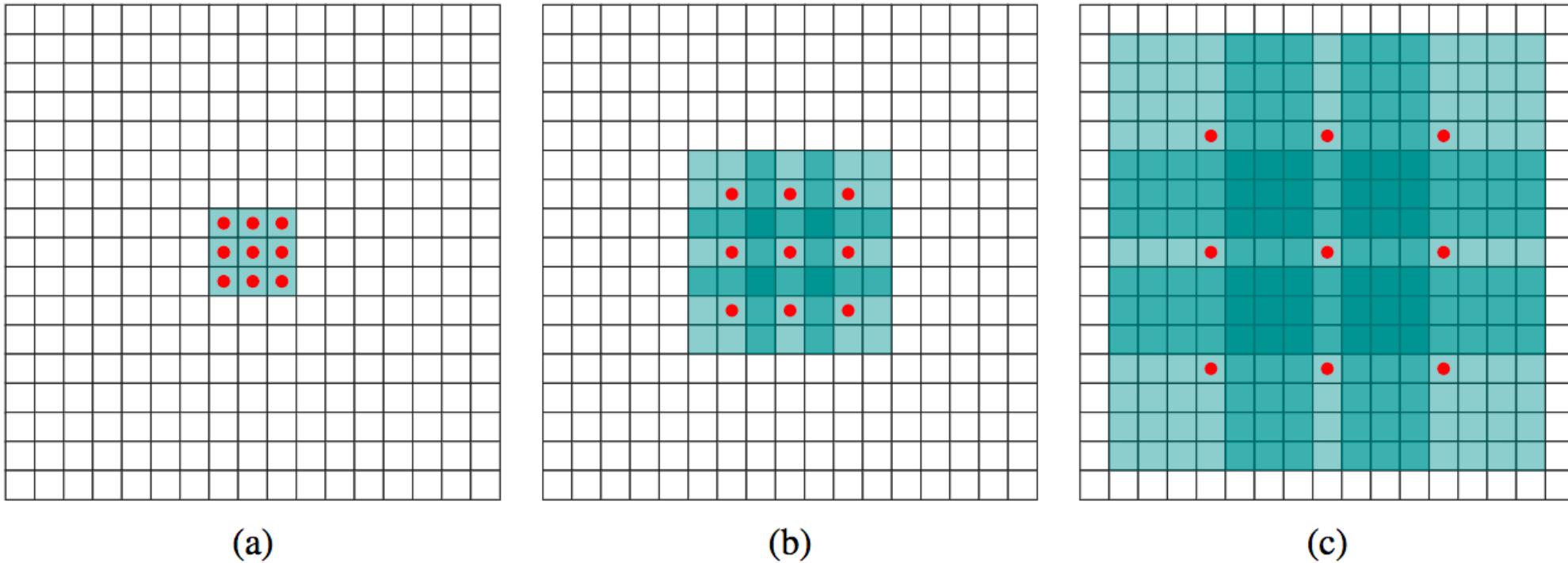


Figure 1: Systematic dilation supports exponential expansion of the receptive field without loss of resolution or coverage. (a) F_1 is produced from F_0 by a 1-dilated convolution; each element in F_1 has a receptive field of 3×3 . (b) F_2 is produced from F_1 by a 2-dilated convolution; each element in F_2 has a receptive field of 7×7 . (c) F_3 is produced from F_2 by a 4-dilated convolution; each element in F_3 has a receptive field of 15×15 . The number of parameters associated with each layer is identical. The receptive field grows exponentially while the number of parameters grows linearly.

[<https://arxiv.org/pdf/1511.07122.pdf>]

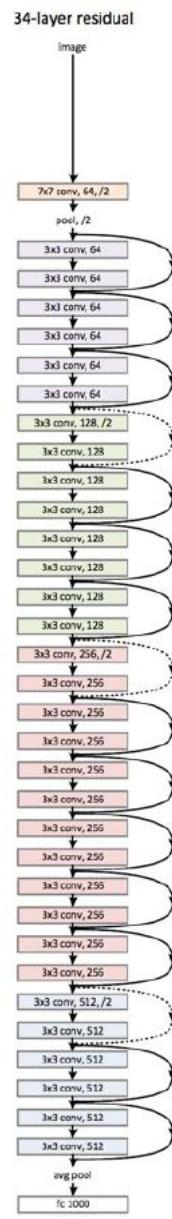
2016: ResNet
>100 conv. layers



Error: 4.4%

Deep Residual Learning for Image Recognition

<https://arxiv.org/pdf/1512.03385.pdf>



$$\mathcal{F}(x)$$

$$\mathcal{F}(x) + x$$

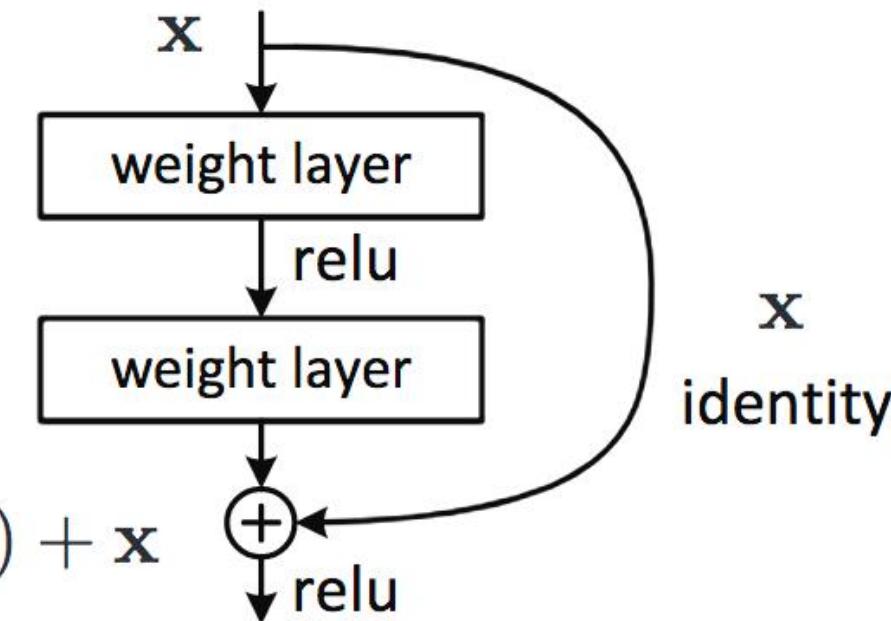
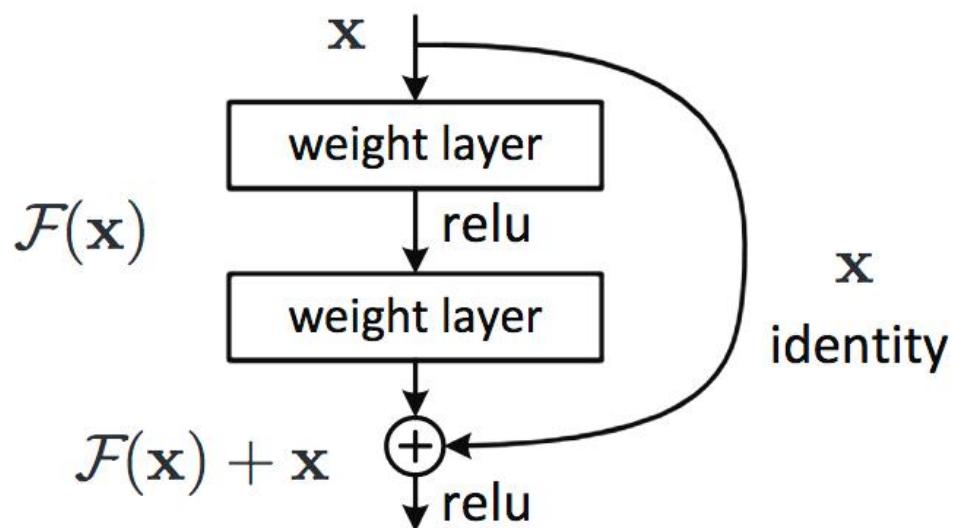
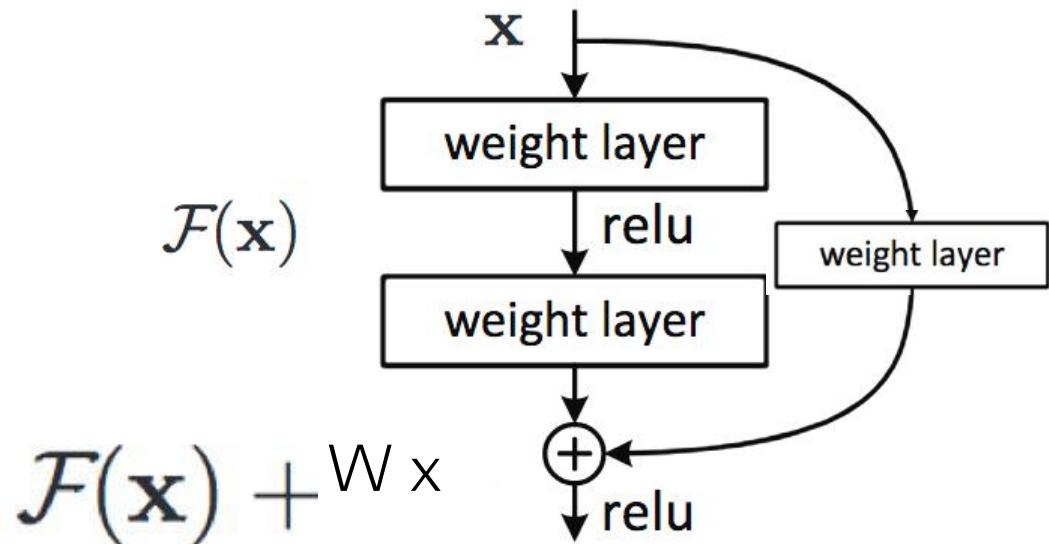


Figure 2. Residual learning: a building block.

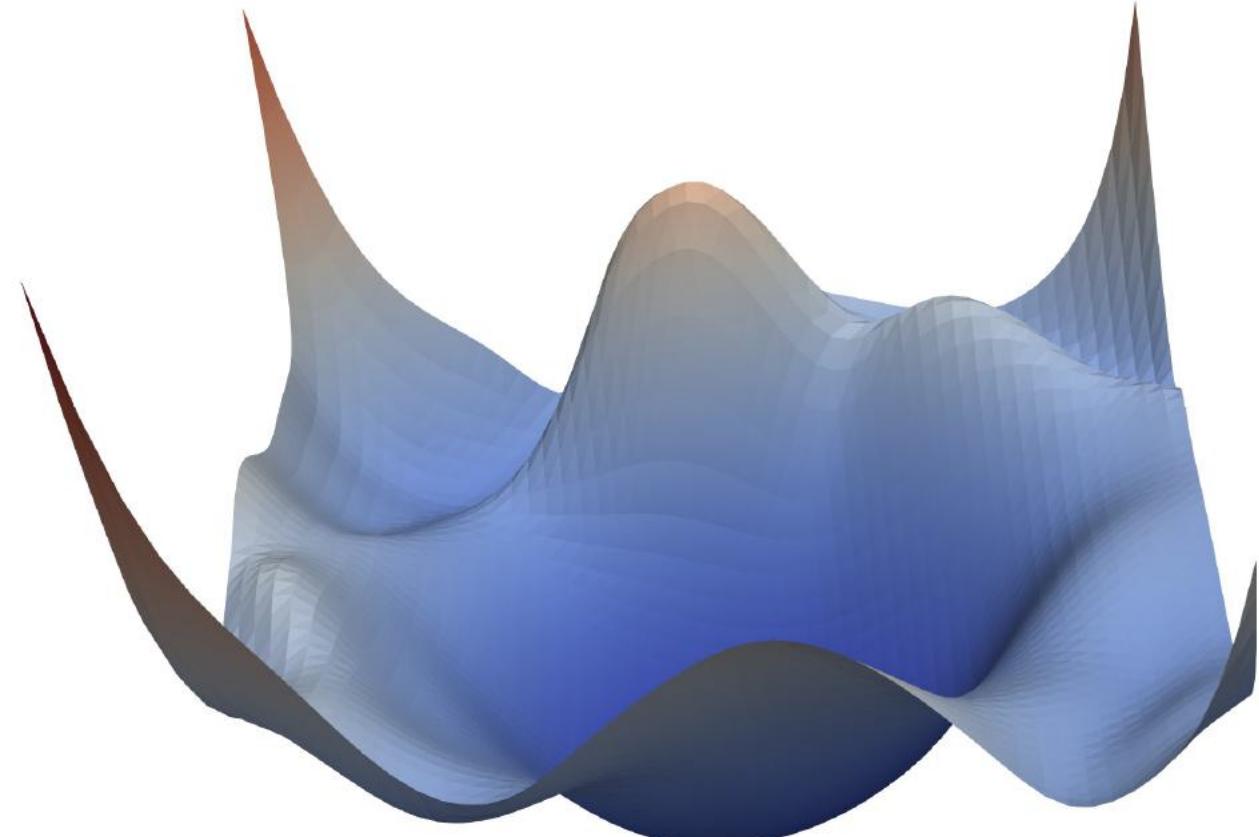
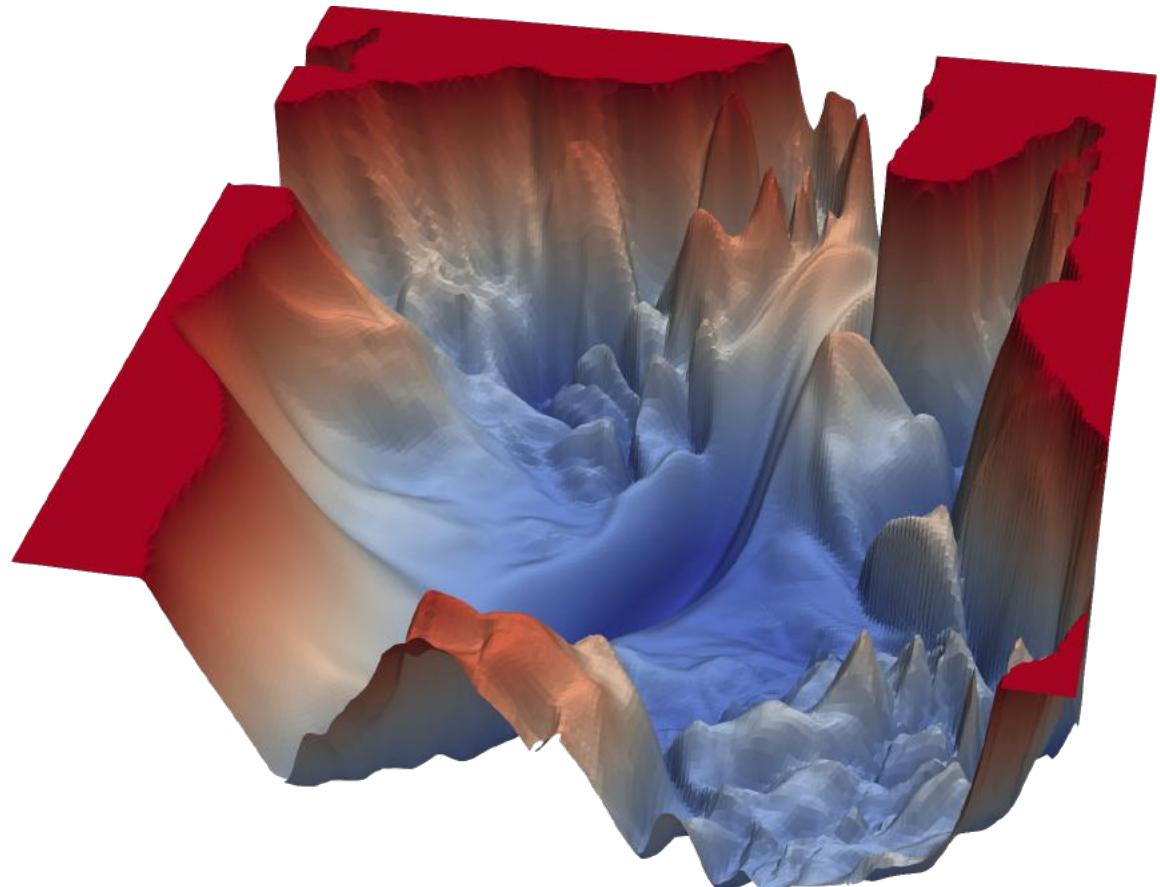
If output has same size as input:



If output has a different size:

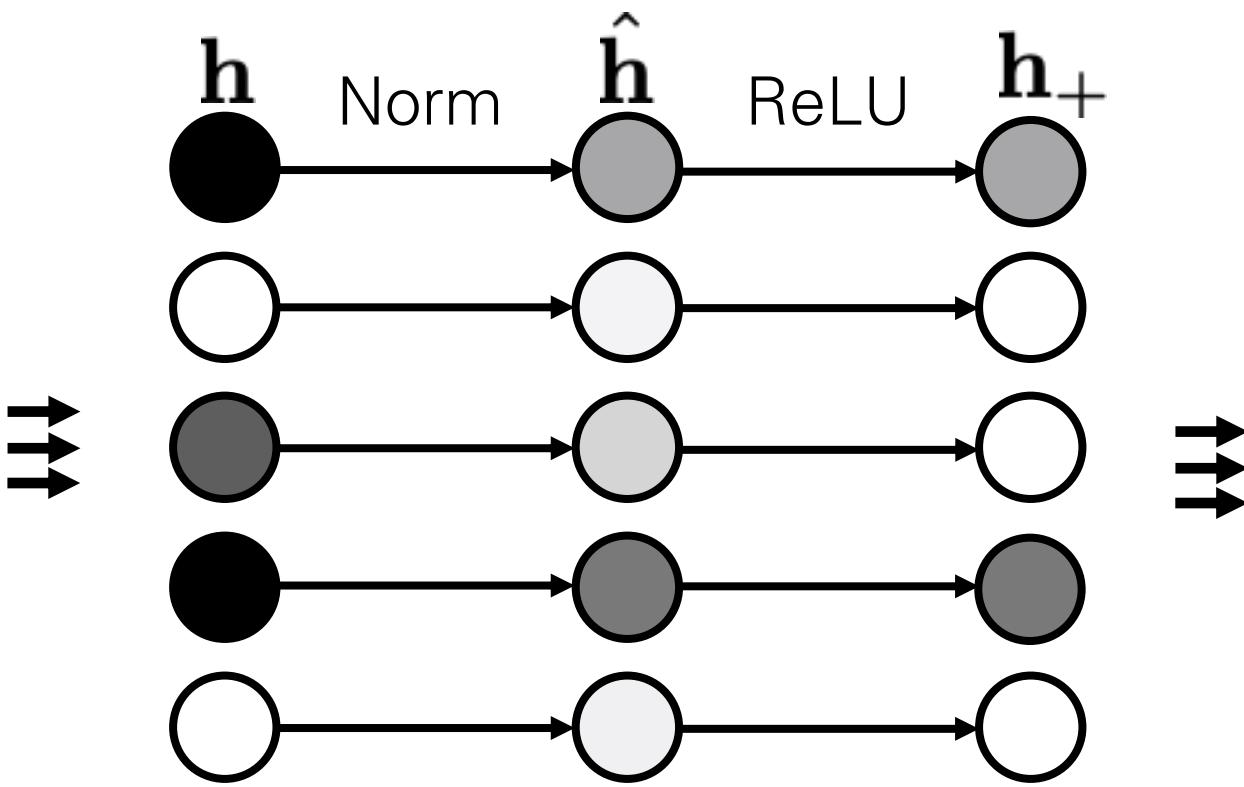


Residual Learning



- The loss surface of a 56-layer net using the CIFAR-10 dataset, both without (left) and with (right) residual connections.

Normalization layers



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

Keep track of mean and variance of a unit (or a population of units) over time.

Standardize unit activations by subtracting mean and dividing by variance.

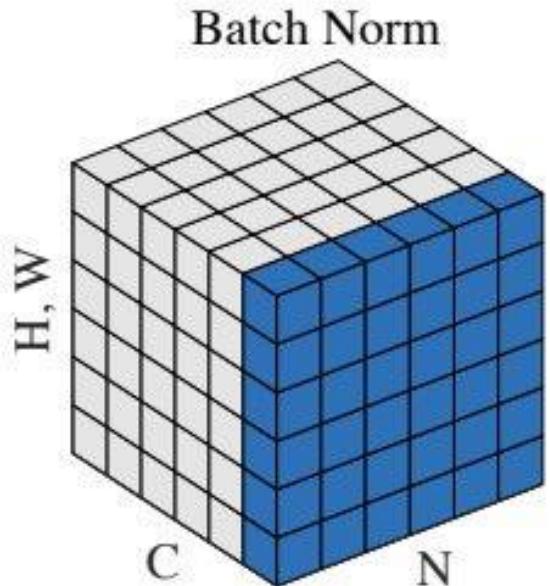
Squashes units into a **standard range**, avoiding overflow.

Also achieves **invariance** to mean and variance of the training signal.

Both these properties reduce the effective capacity of the model, i.e. regularize the model.

Normalization layers

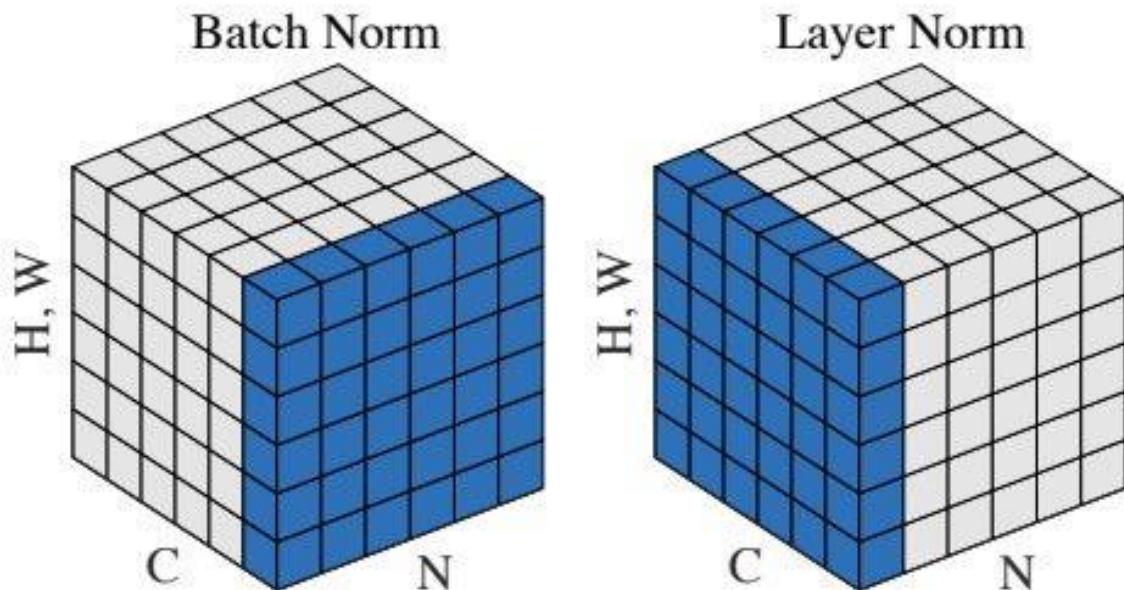
Normalization layers



Normalize w.r.t. a single hidden unit's pattern of activation over training examples (a batch of examples).

[Figure from Wu & He, arXiv 2018]

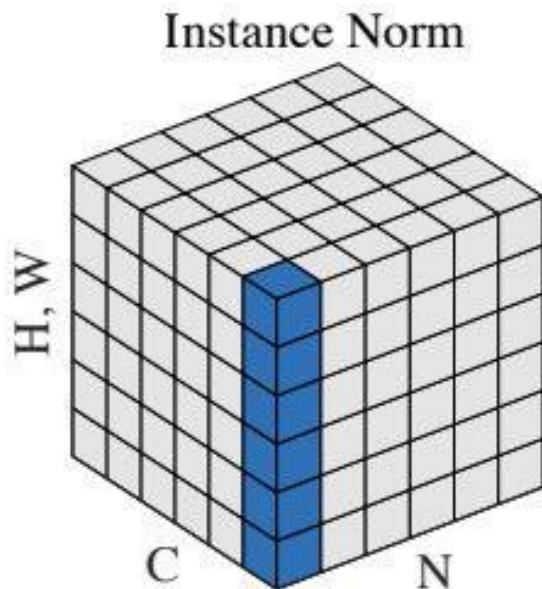
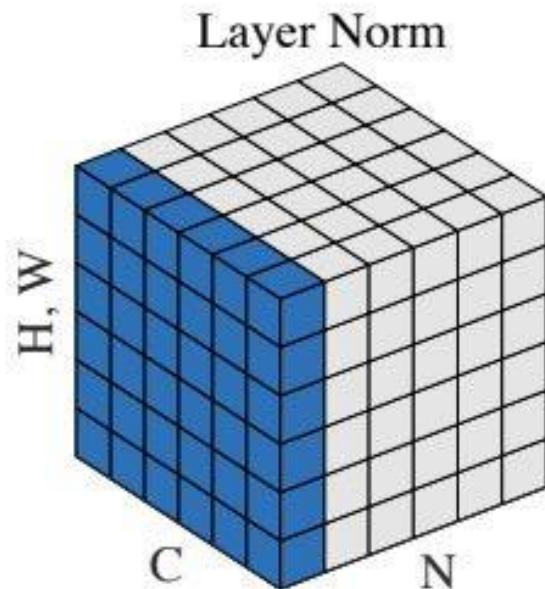
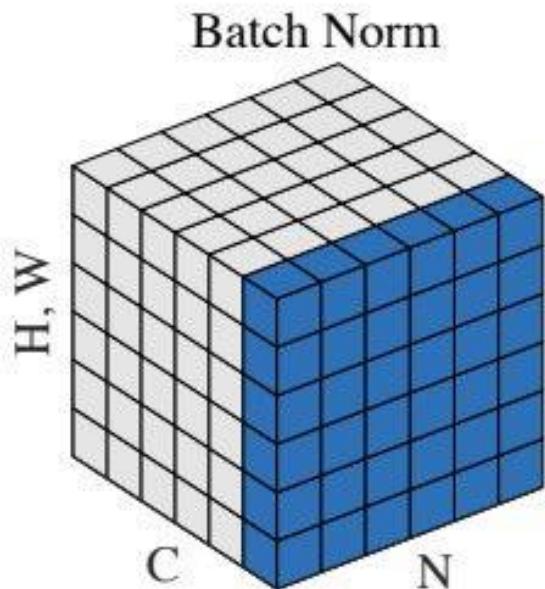
Normalization layers



Normalize w.r.t. the mean and variance of the activations of all the hidden units (neurons) on this layer (c).

[Figure from Wu & He, arXiv 2018]

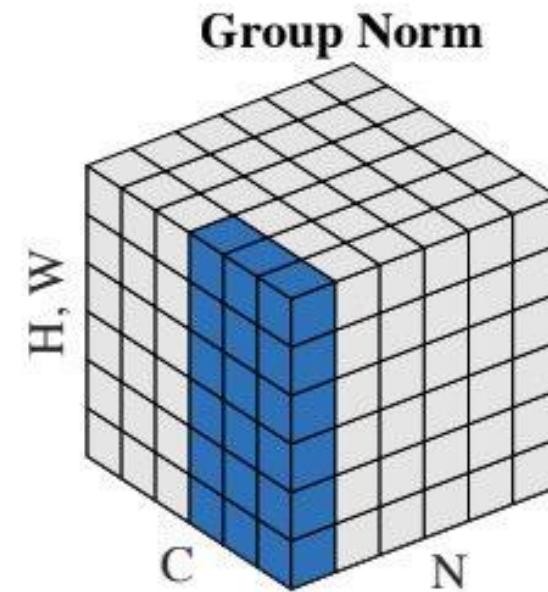
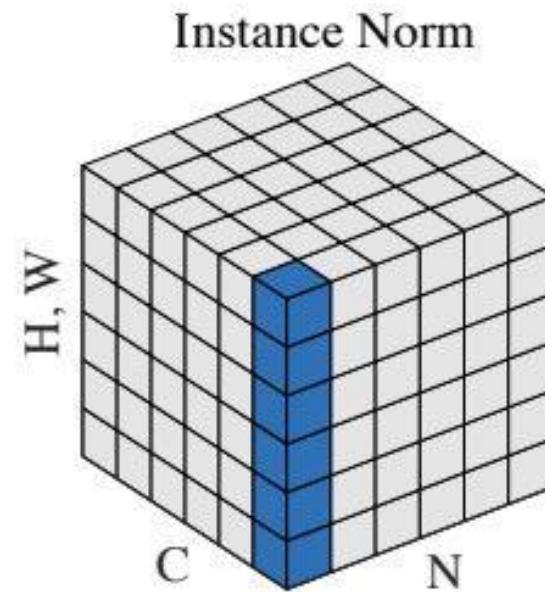
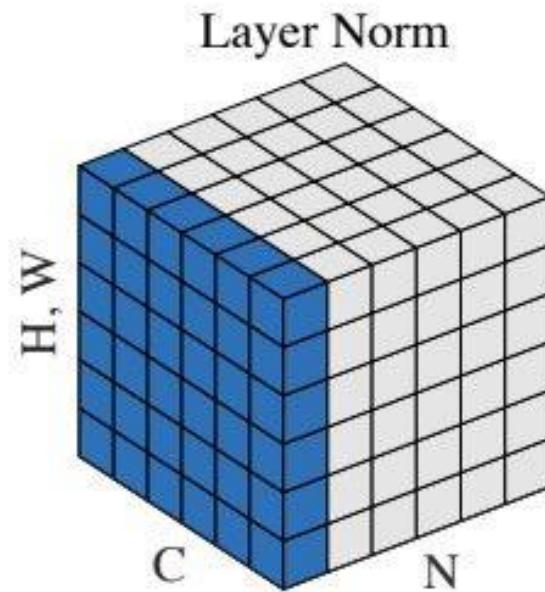
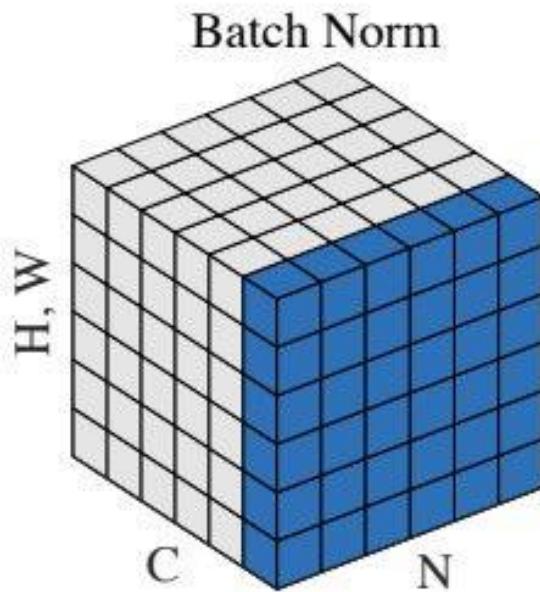
Normalization layers



Normalize w.r.t. the mean and variance of the activations of all the hidden units (neurons) on this layer (c) that process this particular location (h,w) in the image.

[Figure from Wu & He, arXiv 2018]

Normalization layers

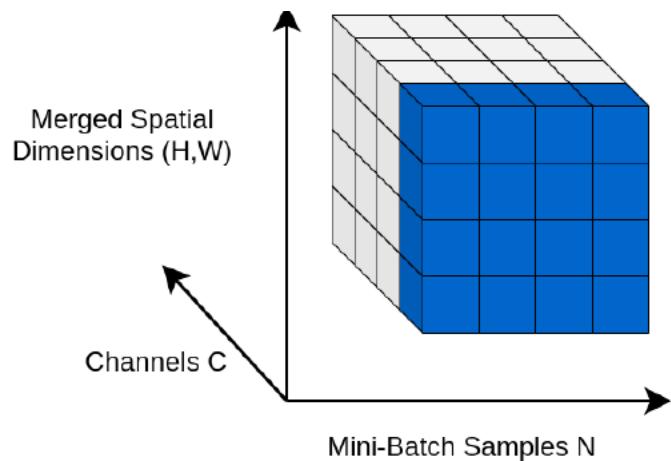


Might as well...

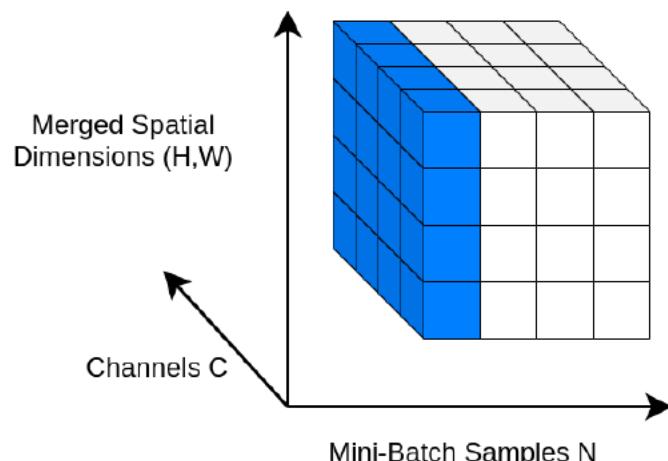
[Figure from Wu & He, arXiv 2018]

Normalization layers

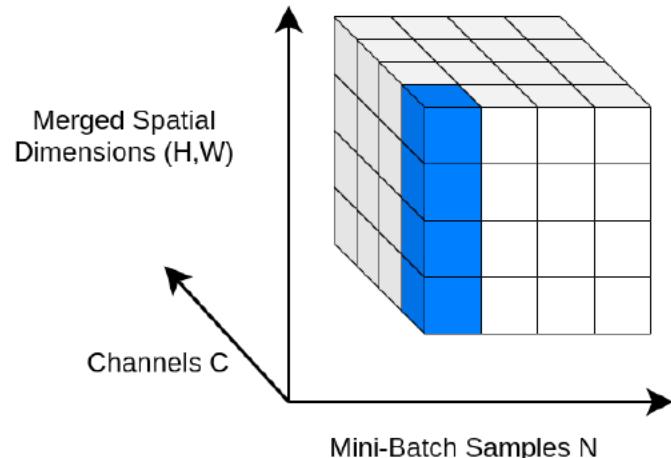
Batch Normalization (2015)



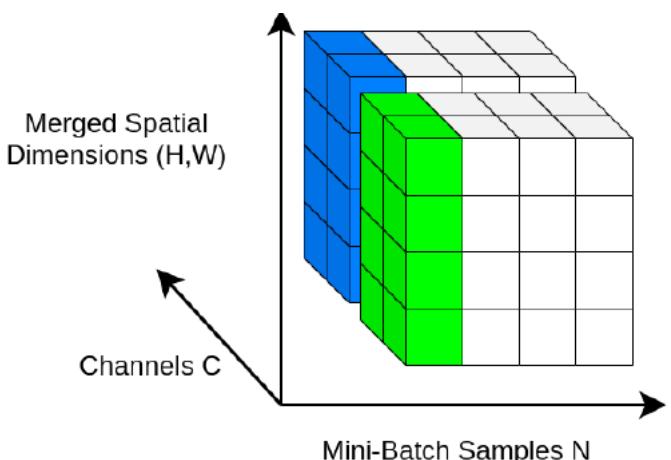
Layer Normalization (2016)



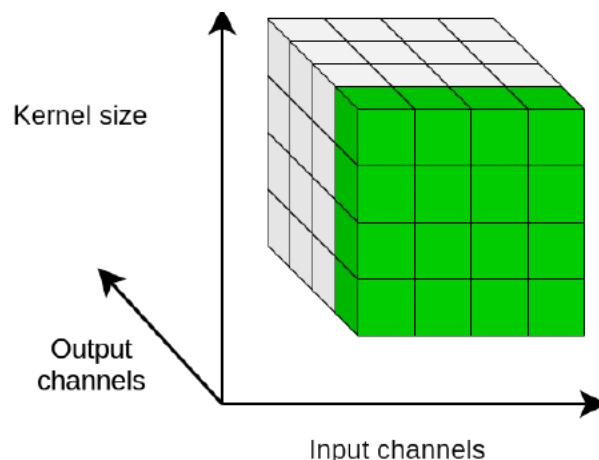
Instance Normalization (2016)



Group Normalization (2018)



Weight Standardization (2019)



No normalization layers

Published as a conference paper at ICLR 2021

CHARACTERIZING SIGNAL PROPAGATION TO CLOSE THE PERFORMANCE GAP IN UNNORMALIZED RESNETS

Andrew Brock, Soham De & Samuel L. Smith
Deepmind
[ajbrock, sohamde, slsmith@google.com](mailto:{ajbrock, sohamde, slsmith}@google.com)

ABSTRACT

Batch Normalization is a key component in almost all state-of-the-art image classifiers, but it also introduces practical challenges: it breaks the independence between training examples within a batch, can incur compute and memory overhead, and often results in unexpected bugs. Building on recent theoretical analyses of deep ResNets at initialization, we propose a simple set of analysis tools to characterize signal propagation on the forward pass, and leverage these tools to design highly performant ResNets without activation normalization layers. Crucial to our success is an adapted version of the recently proposed Weight Standardization. Our analysis tools show how this technique preserves the signal in networks with ReLU or Swish activation functions by ensuring that the per-channel activation means do not grow with depth. Across a range of FLOP budgets, our networks attain performance competitive with the state-of-the-art EfficientNets on ImageNet.

1 INTRODUCTION

BatchNorm has become a core computational primitive in deep learning (Ioffe & Szegedy, 2015), and it is used in almost all state-of-the-art image classifiers (Tan & Le, 2019; Wei et al., 2020). A number of different benefits of BatchNorm have been identified. It smoothens the loss landscape (Santurkar et al., 2018), which allows training with larger learning rates (Bjorck et al., 2018), and the noise arising from the minibatch estimates of the batch statistics introduces implicit regularization (Luo et al., 2019). Crucially, recent theoretical work (Balduzzi et al., 2017; De & Smith, 2020) has demonstrated that BatchNorm ensures good signal propagation at initialization in deep residual networks with identity skip connections (He et al., 2016b;a), and this benefit has enabled practitioners to train deep ResNets with hundreds or even thousands of layers (Zhang et al., 2019). However, BatchNorm also has many disadvantages. Its behavior is strongly dependent on the batch

arXiv:2102.06171v1 [cs.CV] 11 Feb 2021

High-Performance Large-Scale Image Recognition Without Normalization

Andrew Brock¹ Soham De¹ Samuel L. Smith¹ Karen Simonyan¹

Abstract

Batch normalization is a key component of most image classification models, but it has many undesirable properties stemming from its dependence on the batch size and interactions between examples. Although recent work has succeeded in training deep ResNets without normalization layers, these models do not match the test accuracies of the best batch-normalized networks, and are often unstable for large learning rates or strong data augmentations. In this work, we develop an adaptive gradient clipping technique which overcomes these instabilities, and design a significantly improved class of Normalizer-Free ResNets. Our smaller models match the test accuracy of an EfficientNet-B7 on ImageNet while being up to $8.7\times$ faster to train, and our largest models attain a new state-of-the-art top-1 accuracy of 86.5%. In addition, Normalizer-Free models attain significantly better performance than their batch-normalized counterparts when fine-tuning on ImageNet after large-scale pre-training on a dataset of 300 million labeled images, with our best models obtaining an accuracy of 89.2%.²

1. Introduction

The vast majority of recent models in computer vision are variants of deep residual networks (He et al., 2016b;a), trained with batch normalization (Ioffe & Szegedy, 2015).

Figure 1. ImageNet Validation Accuracy vs Training Latency. All numbers are single-model, single crop. Our NFNet-F1 model achieves comparable accuracy to an EffNet-B7 while being $8.7\times$ faster to train. Our NFNet-F5 model has similar training latency to EffNet-B7, but achieves a state-of-the-art 86.0% top-1 accuracy on ImageNet. We further improve on this using Sharpness Aware Minimization (Foret et al., 2021) to achieve 86.5% top-1 accuracy.

However, batch normalization has three significant practical disadvantages. First, it is a surprisingly expensive computational primitive, which incurs memory overhead (Rota Bulò et al., 2018), and significantly increases the time required to evaluate the gradient in some networks (Gitman & Ginsburg, 2017). Second, it introduces a discrepancy between the behaviour of the model during training and at inference time (Summers & Dinneen, 2019; Singh & Shrivastava, 2019).

Next Lecture: Sequential Processing with RNNs