

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

Photo: Detail from Sofia Crespo's Tribute to Manolo Part 2



Lecture overview

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

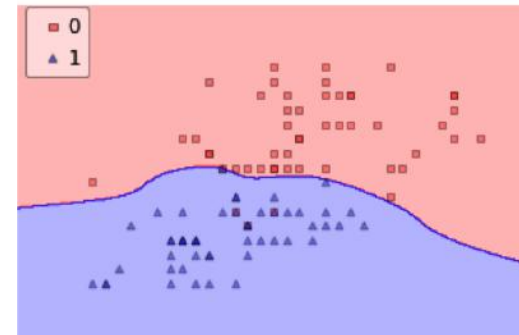
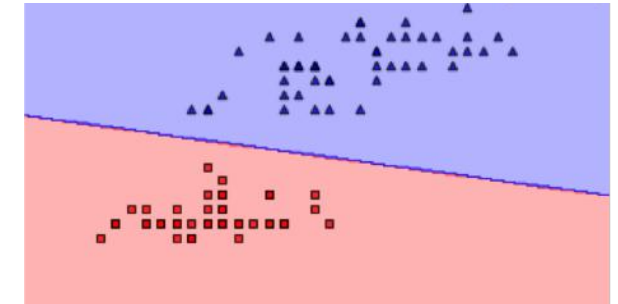
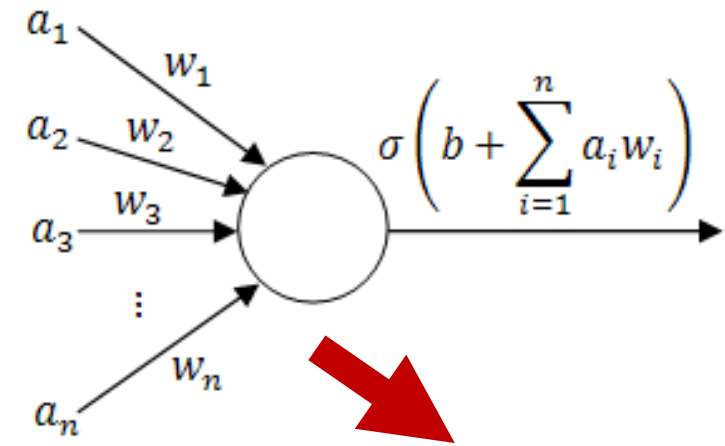
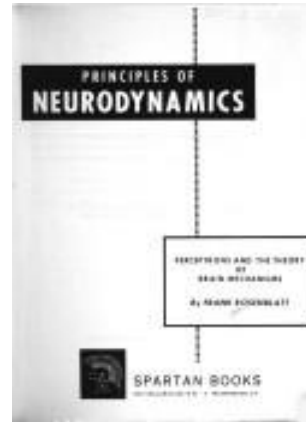
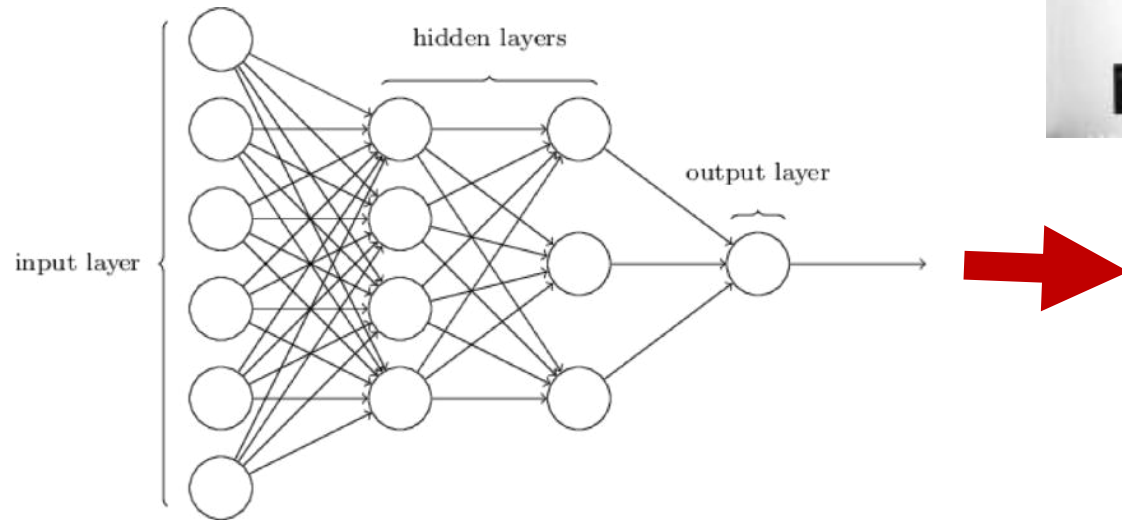
Disclaimer: Much of the material and slides for this lecture were borrowed from

—Costis Daskalakis and Aleksander Madry's MIT 6.883 class

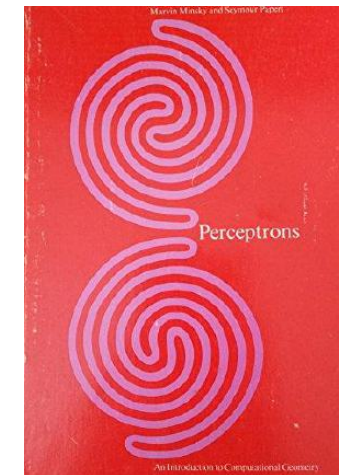
—Bill Freeman, Antonio Torralba and Phillip Isola's MIT 6.869 class

Humble beginnings

- Perceptron [Rosenblatt '58]

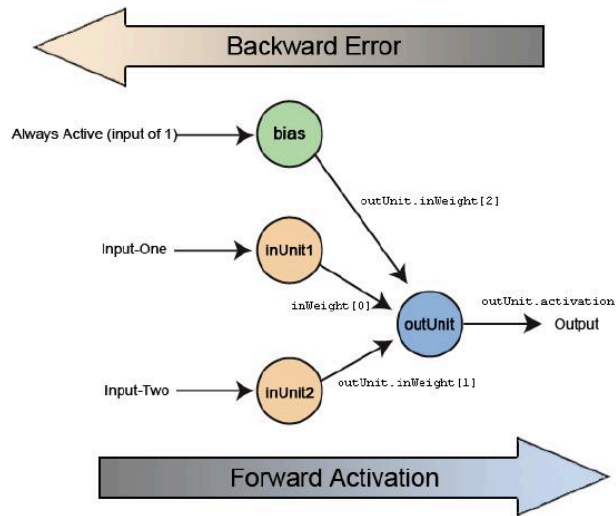


- Criticism of Perceptrons (XOR affair) [Minsky Papert '69]
 - Effectively causes a "deep learning winter"



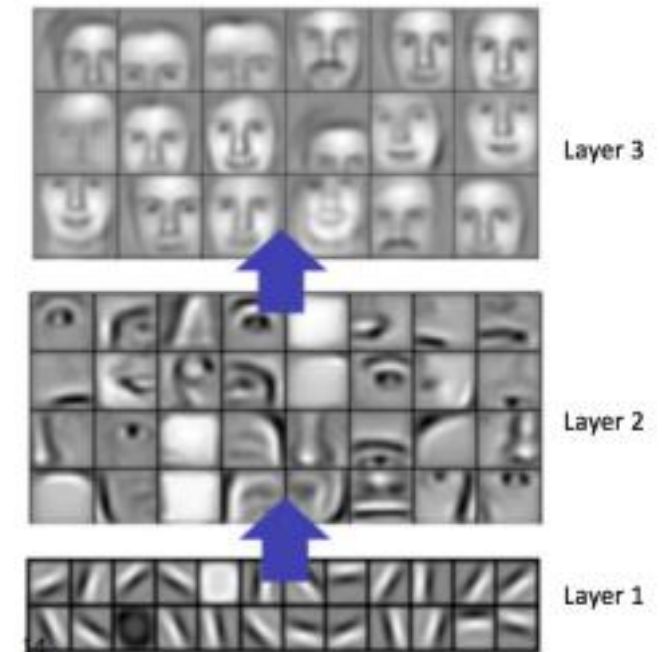
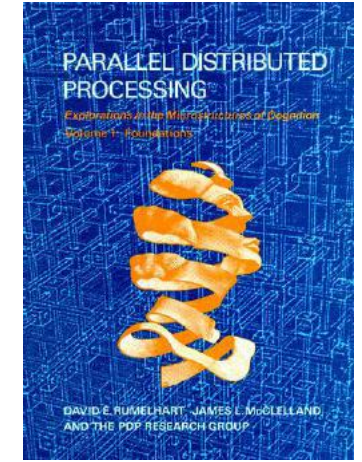
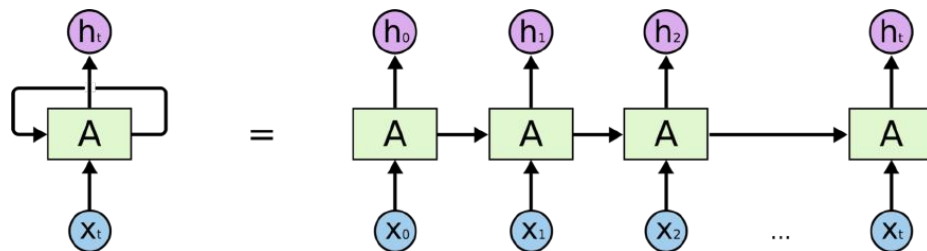
(Early) Spring

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



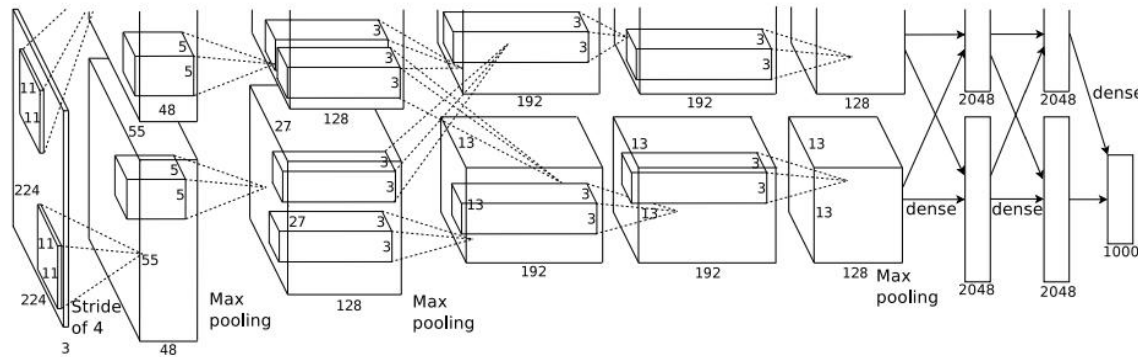
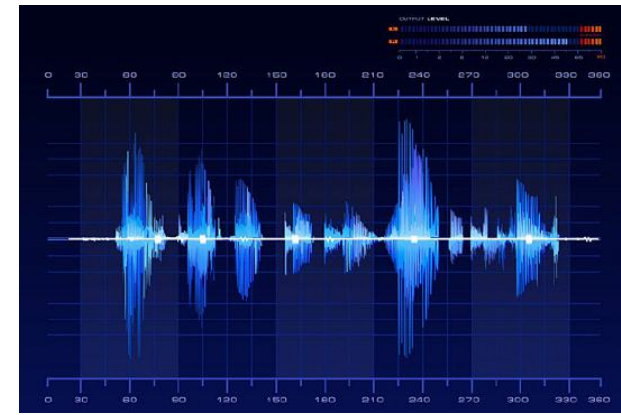
- Convolutional layers [LeCun et al. '90]

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

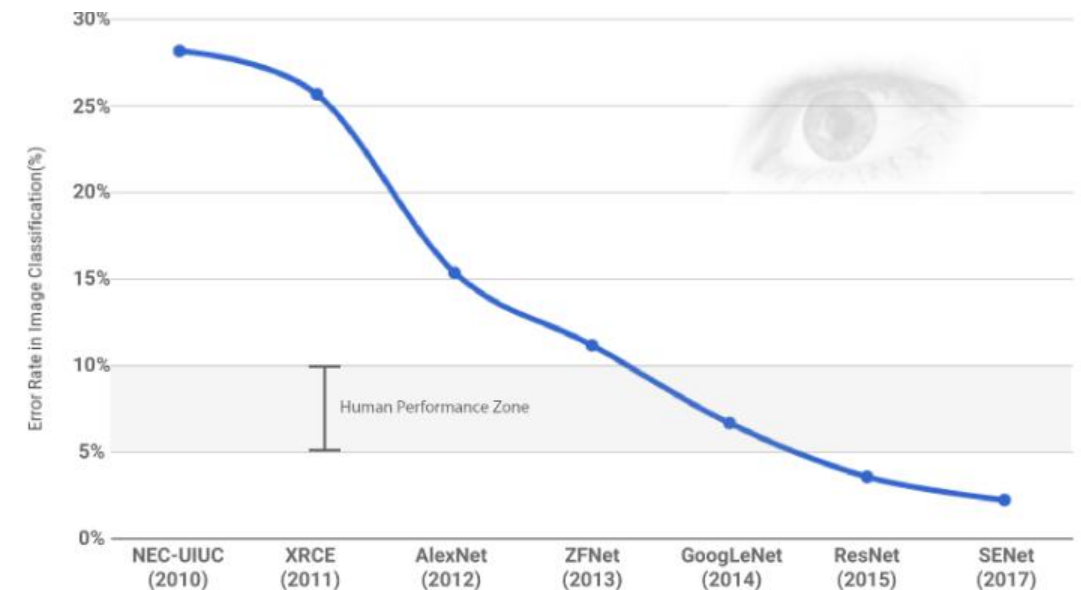


Summer

- 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



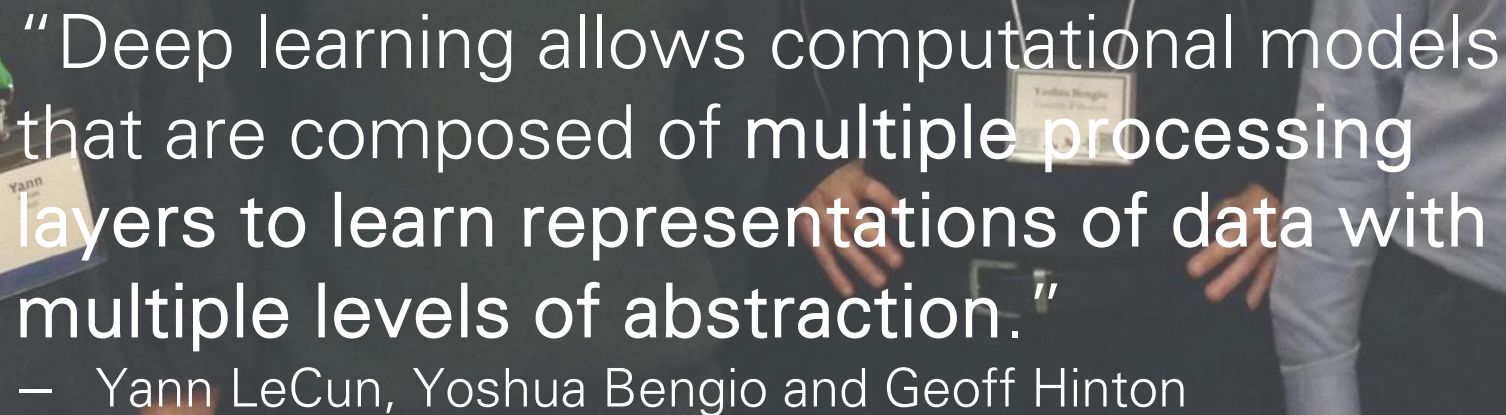
- Enough computational power



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**.

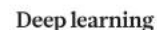
Deep neural networks!



“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

Yann LeCun^{1,2}, Yoshua Bengio² & Geoffrey Hinton^{3,4}

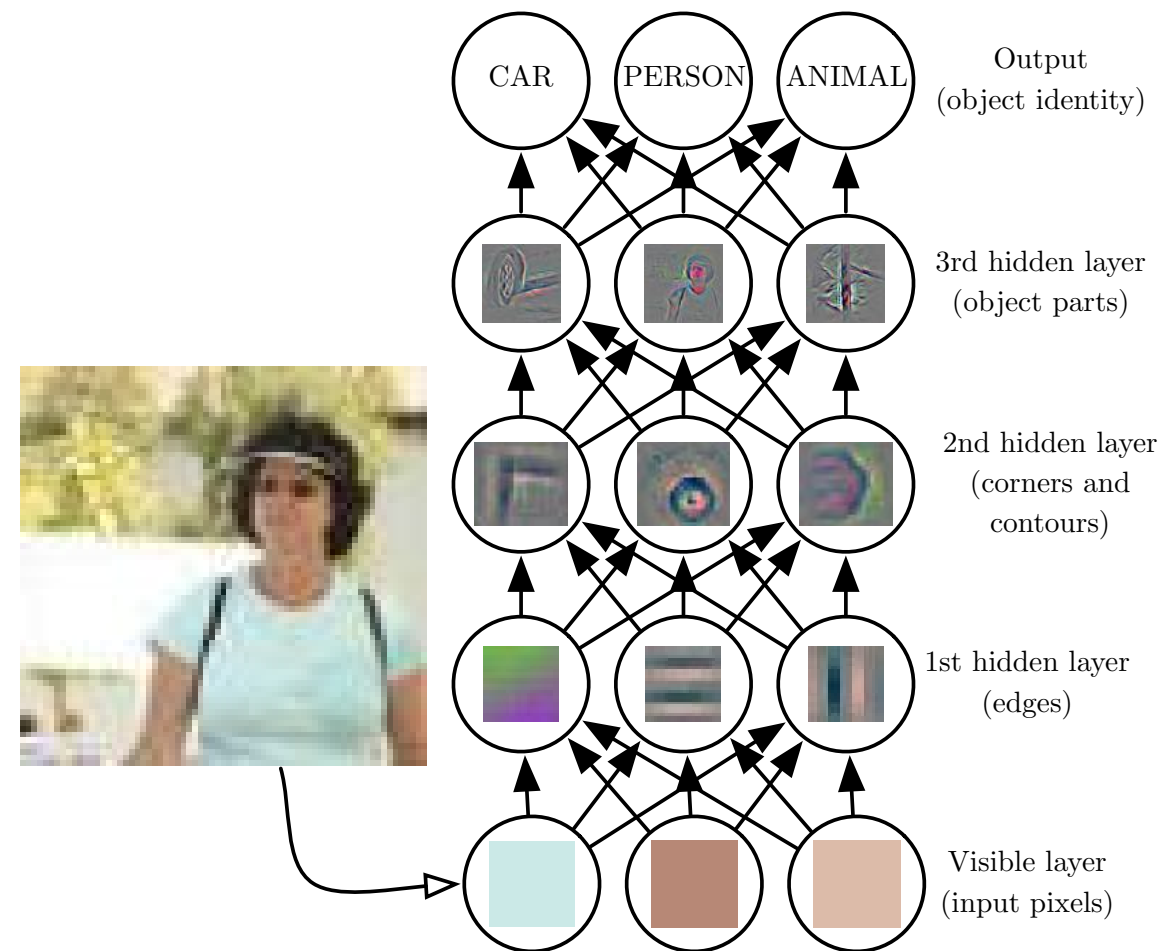
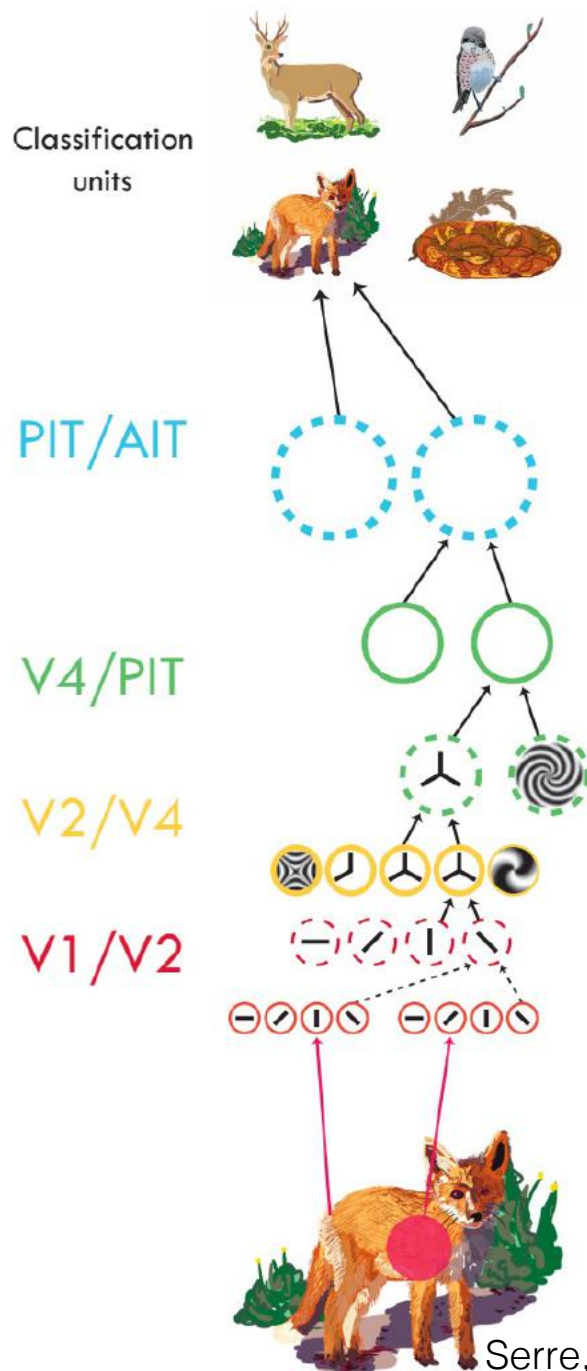
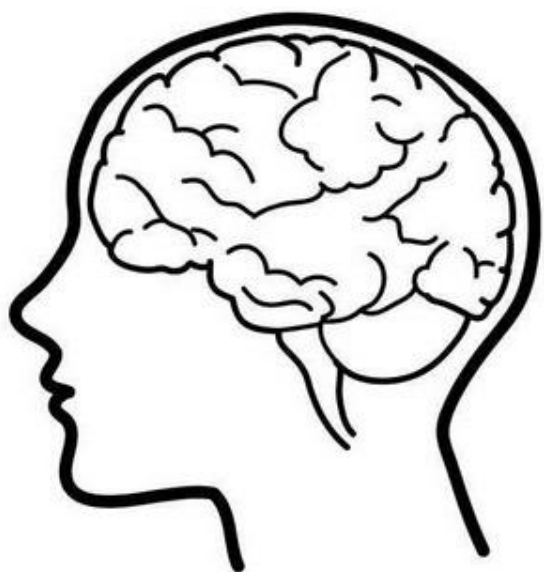
Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction. These methods have dramatically improved the state-of-the-art in speech recognition, visual object recognition, object detection and many other domains such as drug discovery and genomics. Deep learning discovers intricate structure in large data sets by using the backpropagation algorithm to indicate how a machine should change its internal parameters that are used to compute the representation in each layer from the representation in the previous layer. Deep convolutional nets have brought about breakthroughs in processing images, video, speech and audio, whereas recurrent nets have shone light on sequential data such as text and speech.

Machine learning technology powers many aspects of modern science, from web search engines to cancer drug discovery. In addition to its use in many scientific fields, machine learning is also used in the security, financial, and marketing industries. Machine learning systems are used to identify objects in images, transcribe speech into text, match new items, predict product sales, and more. In recent years, machine learning has been used to predict the success of new drugs, and to identify potential drug targets. Machine learning is also used in the design of new drugs, and in the analysis of drug safety data. Machine learning is used in the design of new drugs, and in the analysis of drug safety data. Machine learning is used in the design of new drugs, and in the analysis of drug safety data.

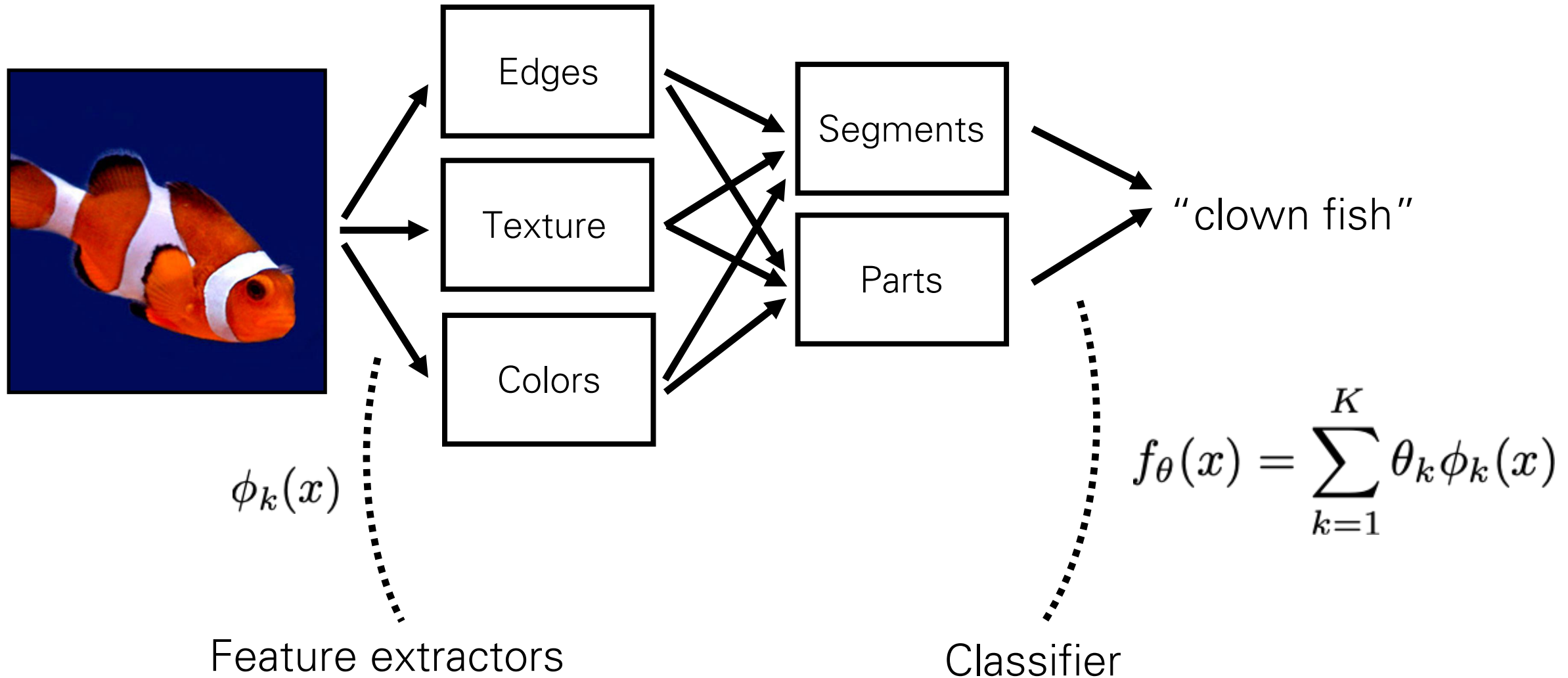
Conventional machine learning techniques were limited in their ability to process natural data in their own form. For decades, constructing a pattern recognition or machine learning system required careful engineering and considerable data expertise to design a feature space that transformed the raw data (such as the pixel values of an image) into a suitable linear representation of feature vectors from which the learning sub-system, often a classifier, could learn.

[illegible][illegible]

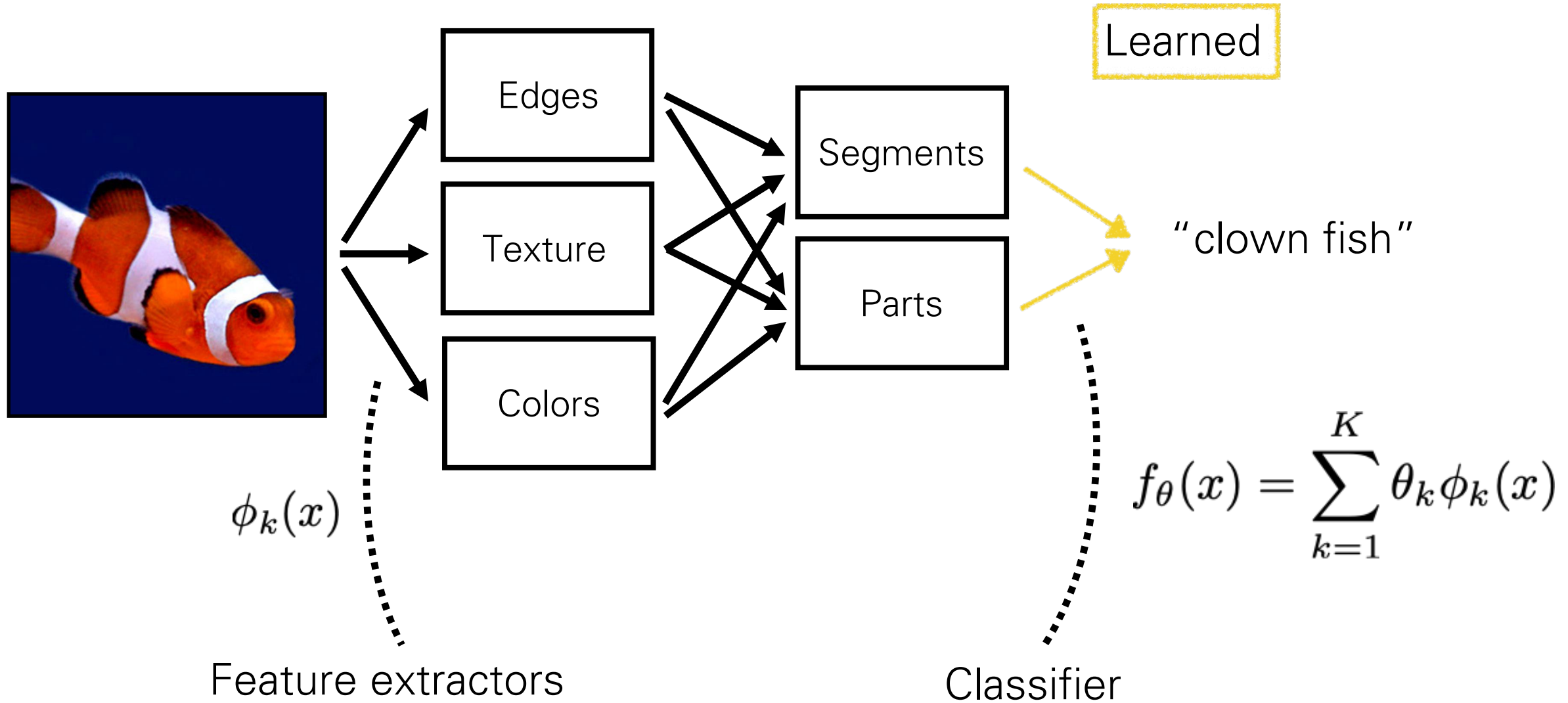
816 / WATERBURY • 2005, 2006 • 14 MAY 2012



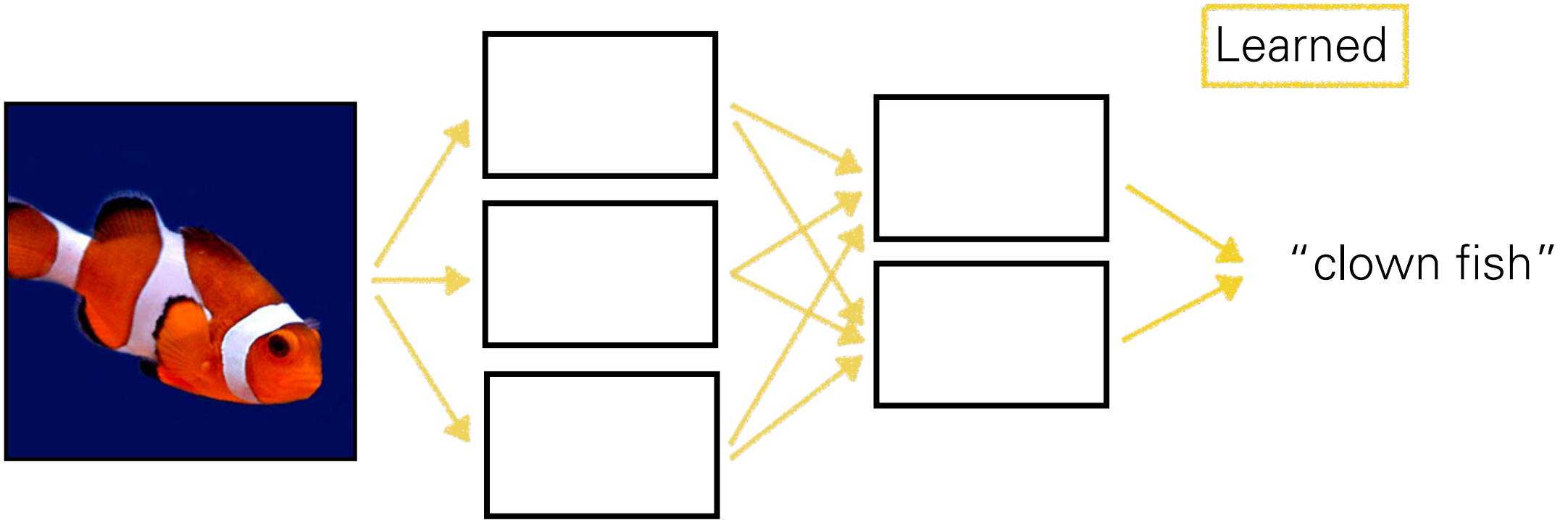
Object recognition



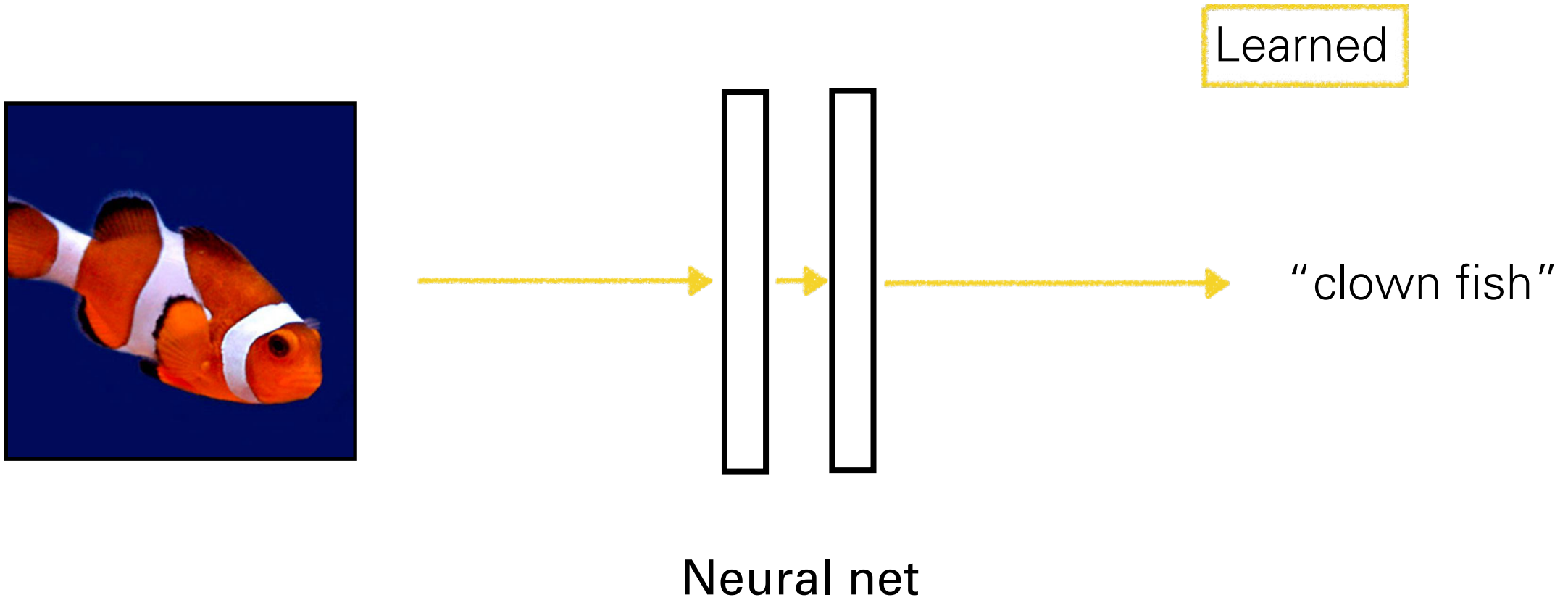
Object recognition



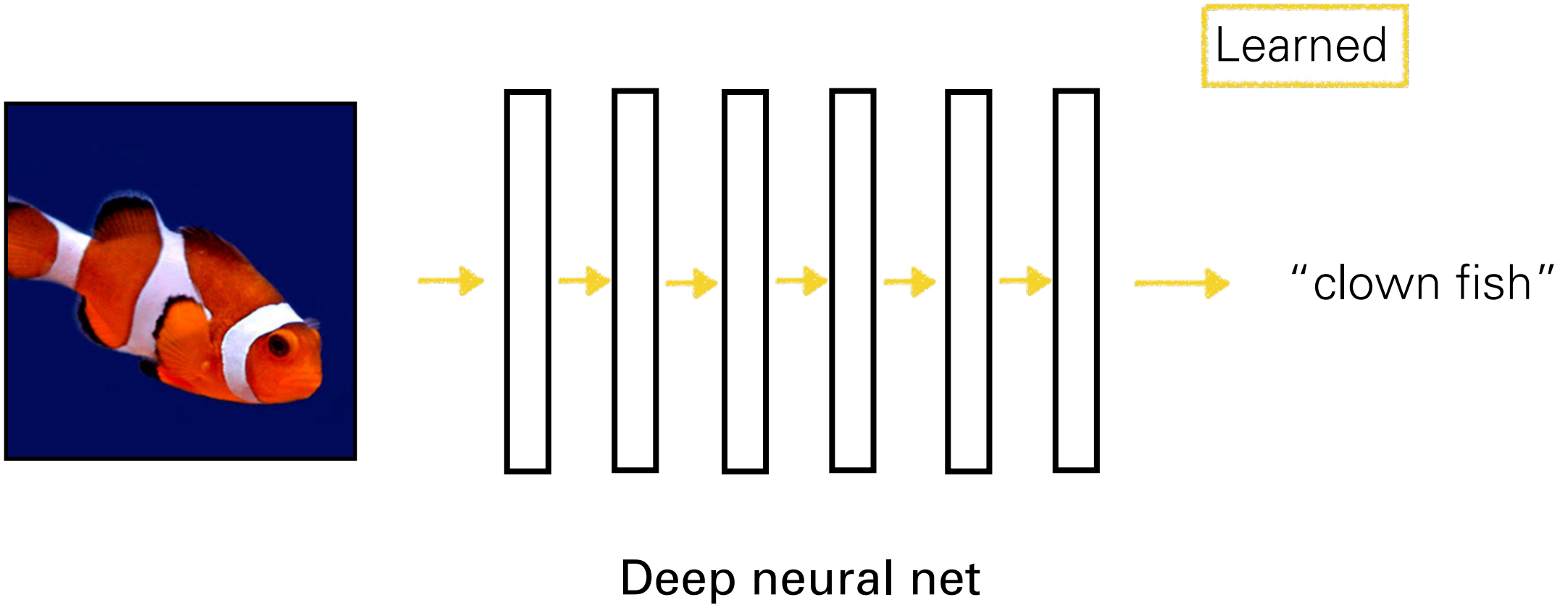
Object recognition



Object recognition

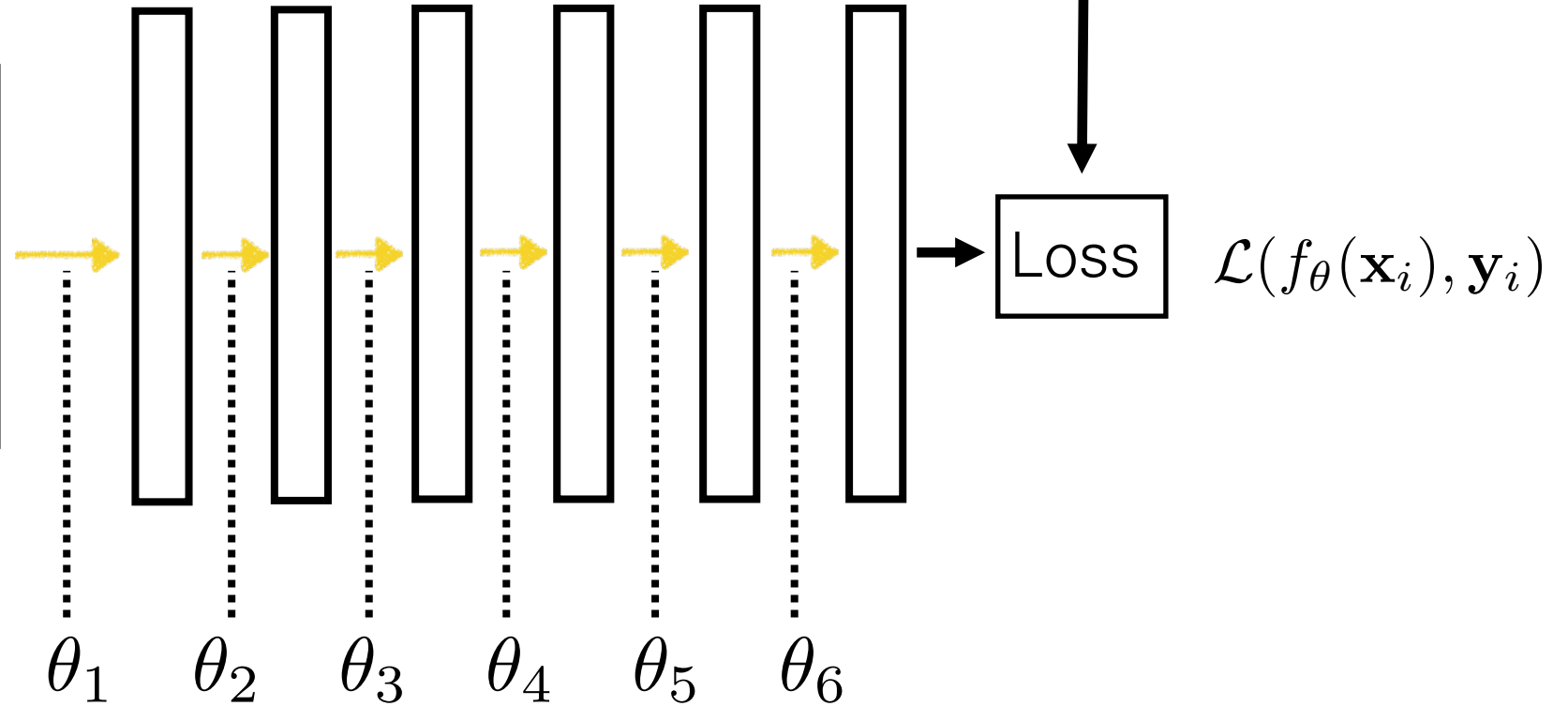
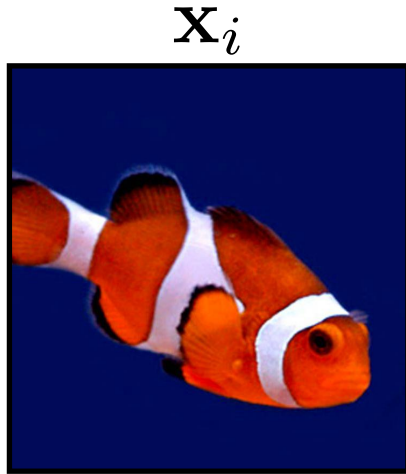


Object recognition



Deep learning

y_i
"clown fish"

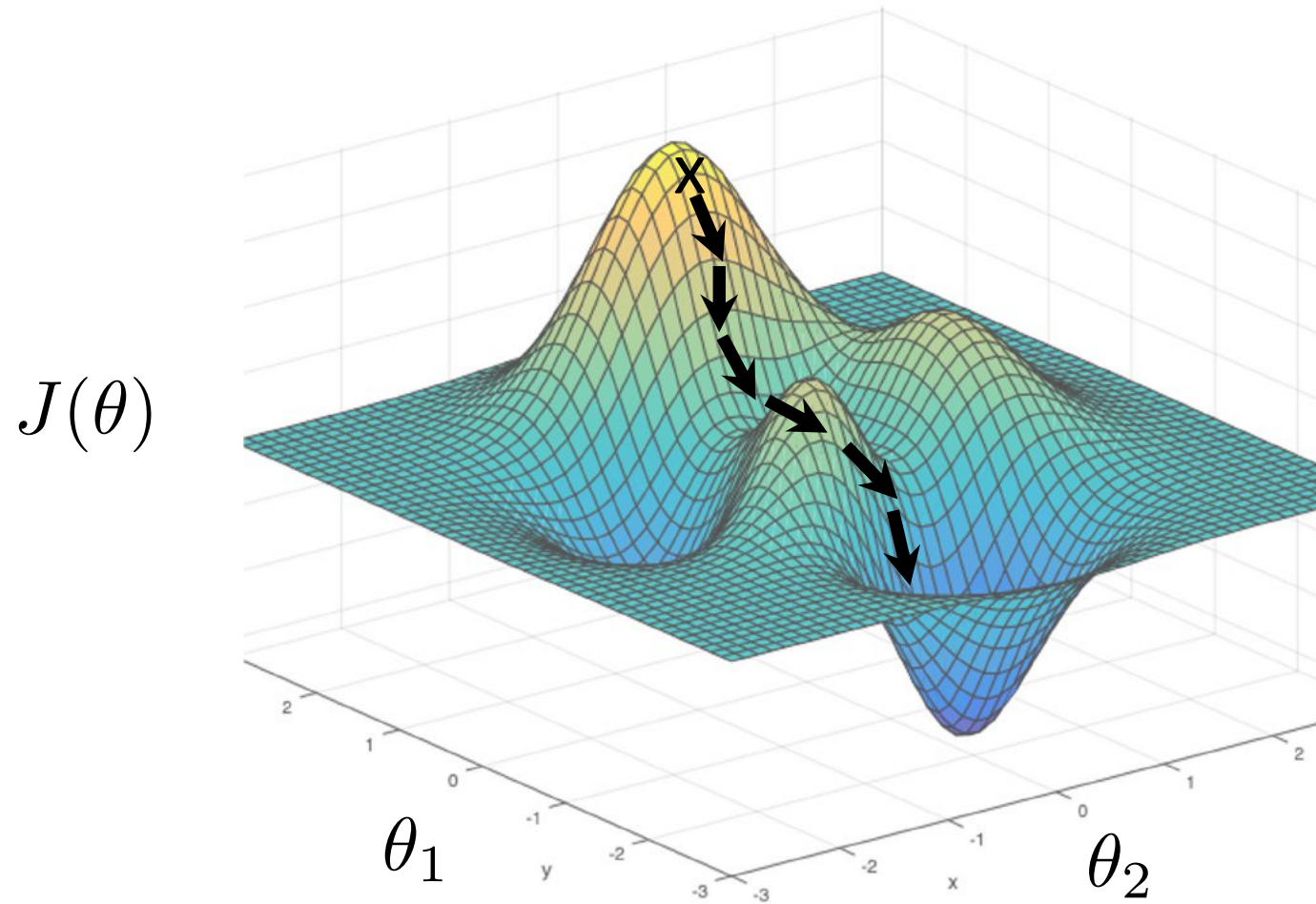


$$\theta^* = \arg \min_{\theta} \sum_{i=1}^N \mathcal{L}(f_{\theta}(\mathbf{x}_i), \mathbf{y}_i)$$

Gradient descent

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

Gradient descent



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

Gradient descent

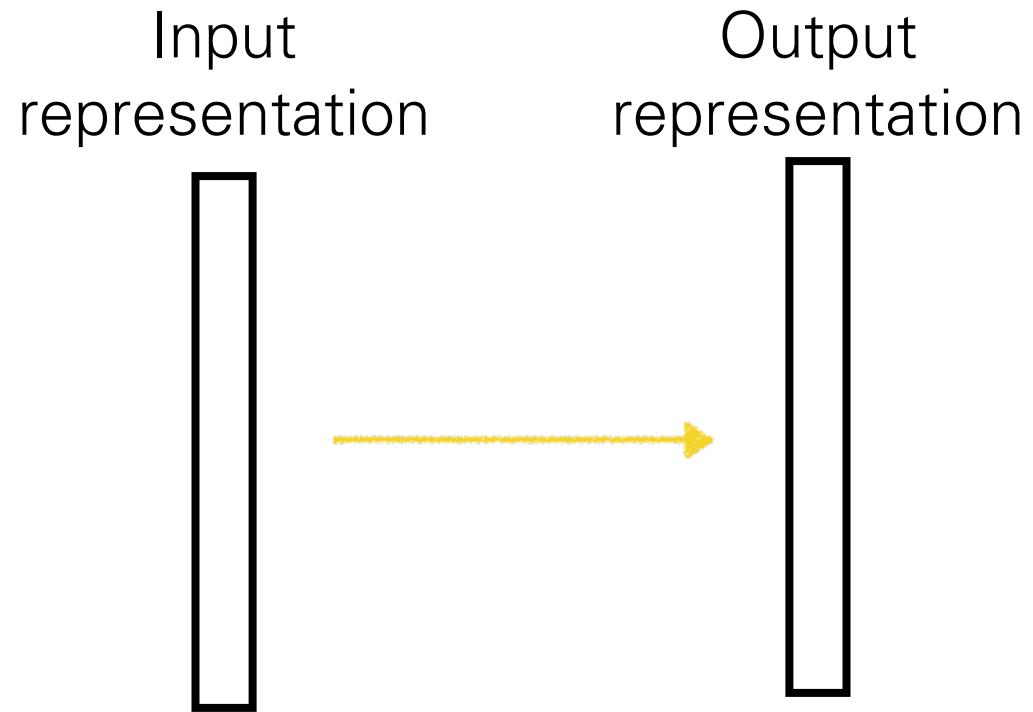
$$\theta^* = \arg \min_{\theta} \underbrace{\sum_{i=1}^N \mathcal{L}(f_{\theta}(\mathbf{x}_i), \mathbf{y}_i)}_{J(\theta)}$$

One iteration of gradient descent:

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

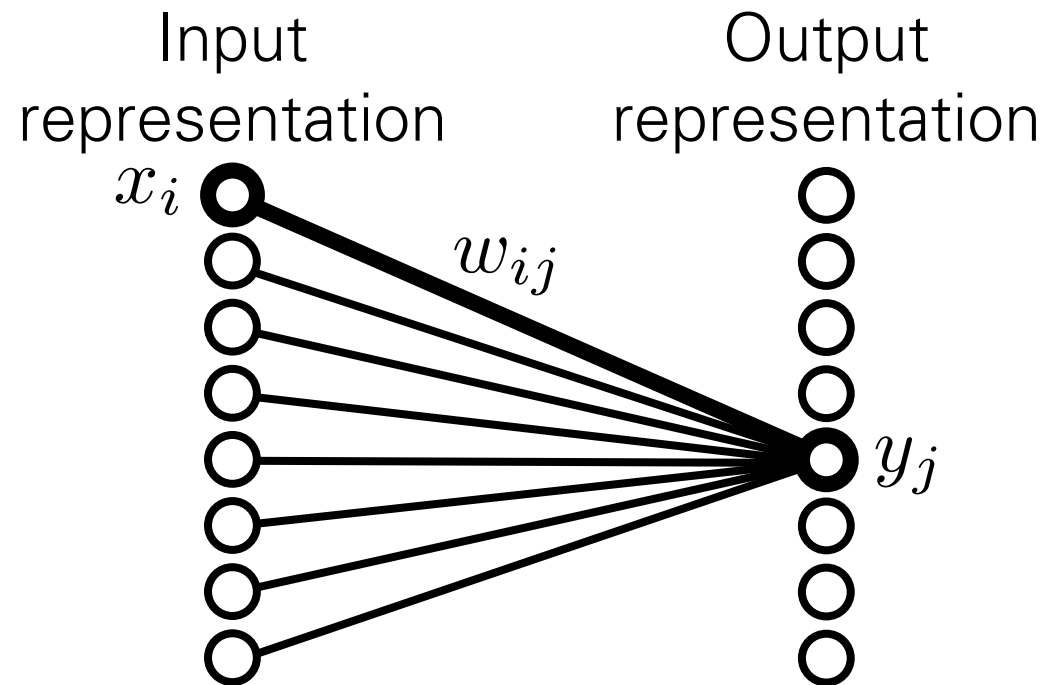
learning rate

Computation in a neural net



Computation in a neural net

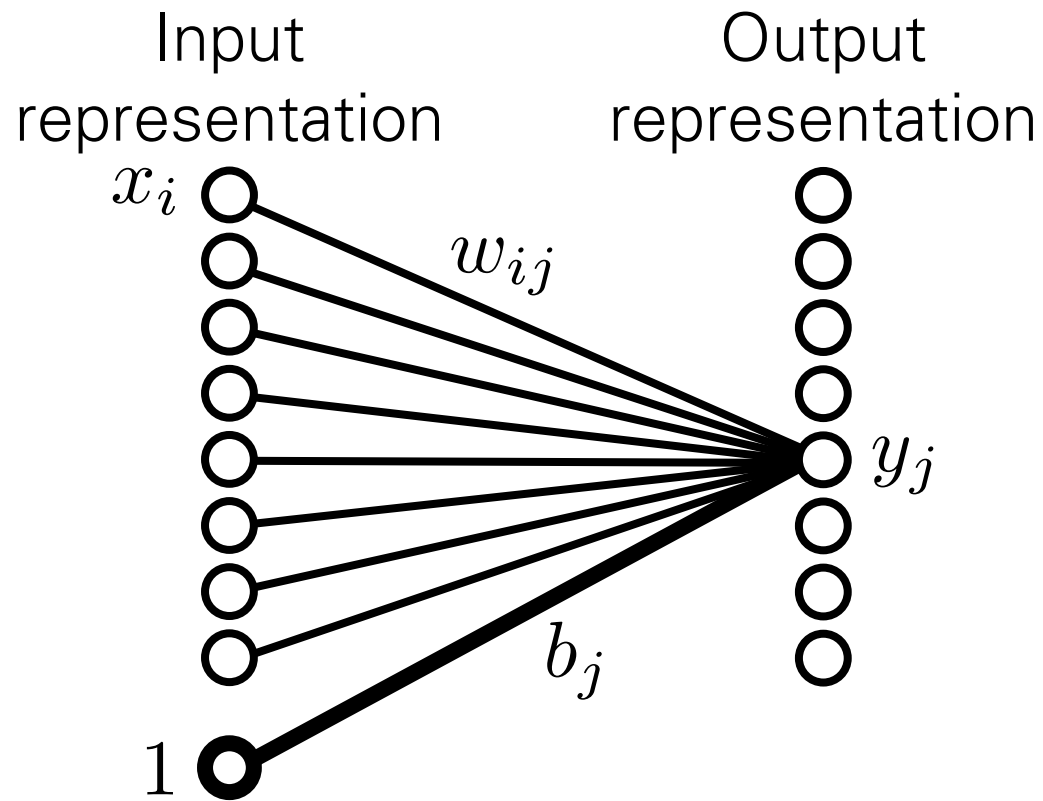
Linear layer



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

Computation in a neural net

Linear layer



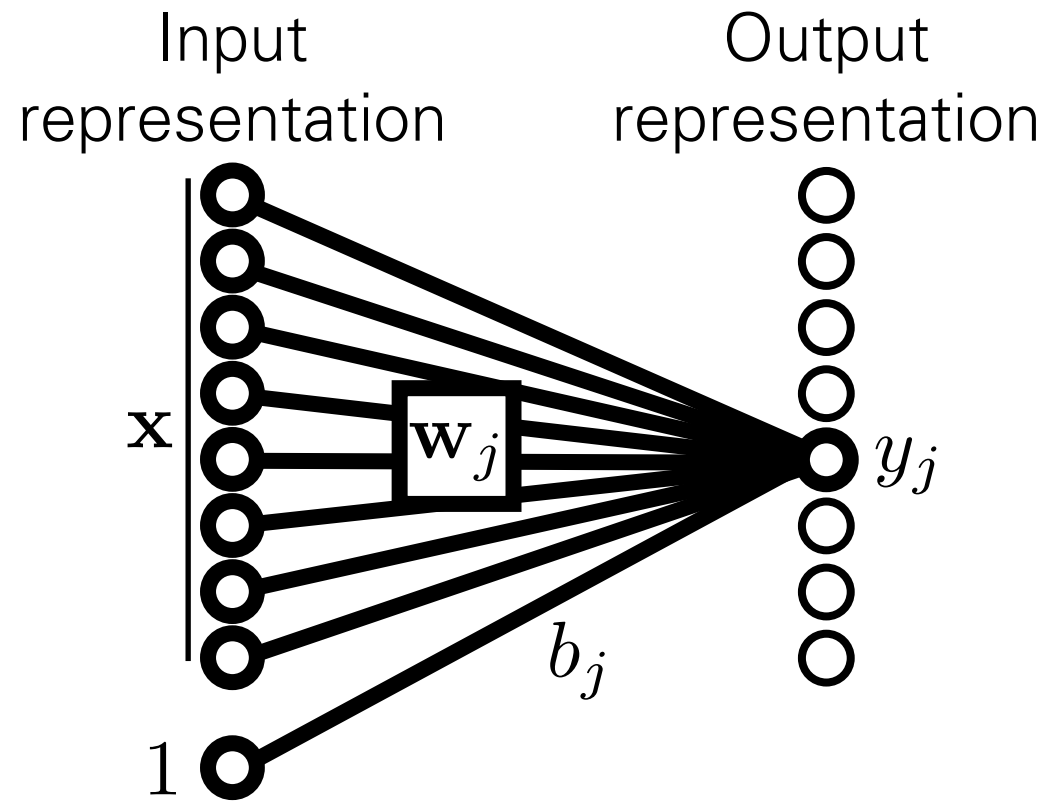
$$y_j = \sum_i w_{ij} x_i + b_j$$

weights

bias

Computation in a neural net

Linear layer



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

weights

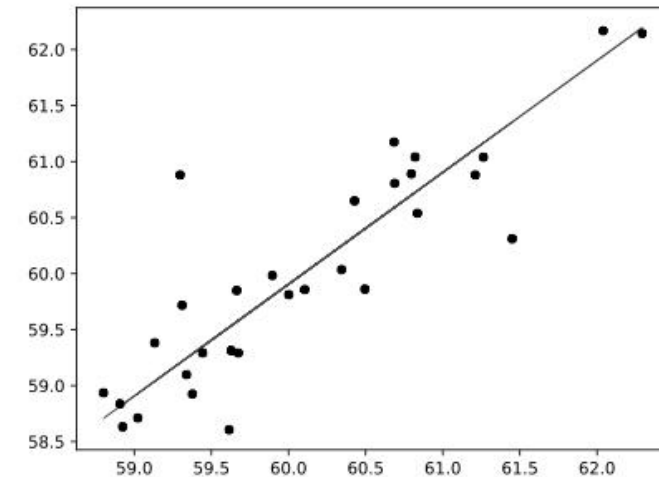
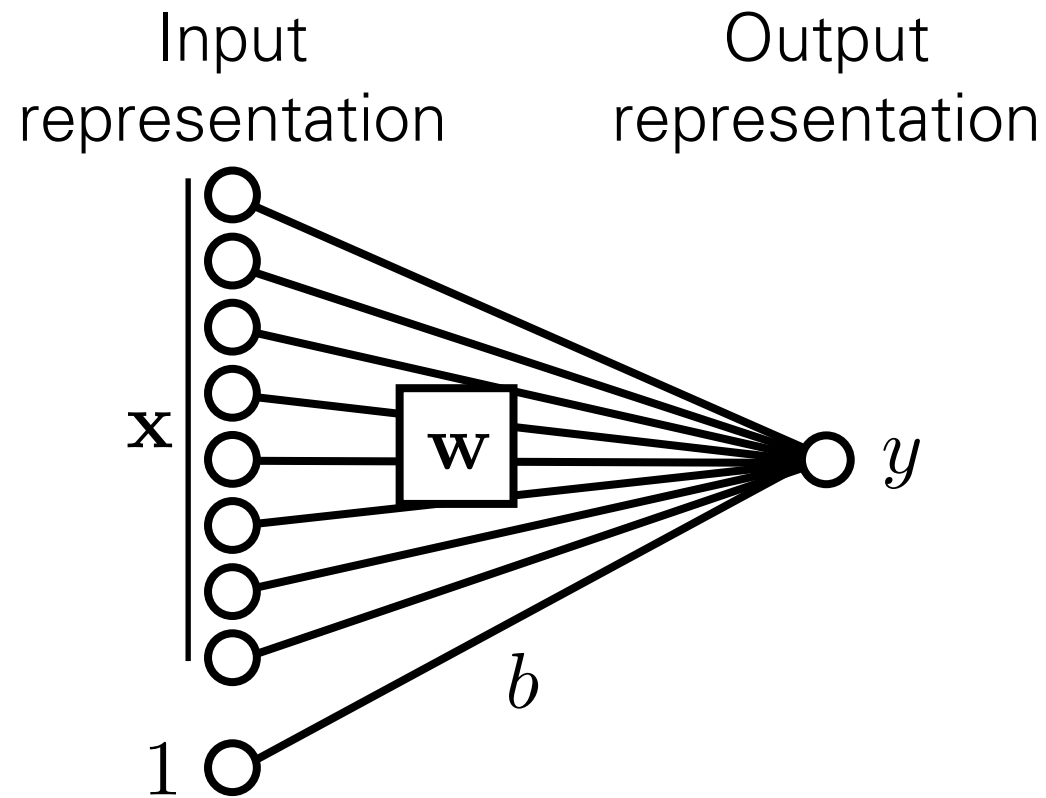
bias

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

parameters of the model

Example: linear regression with a neural net

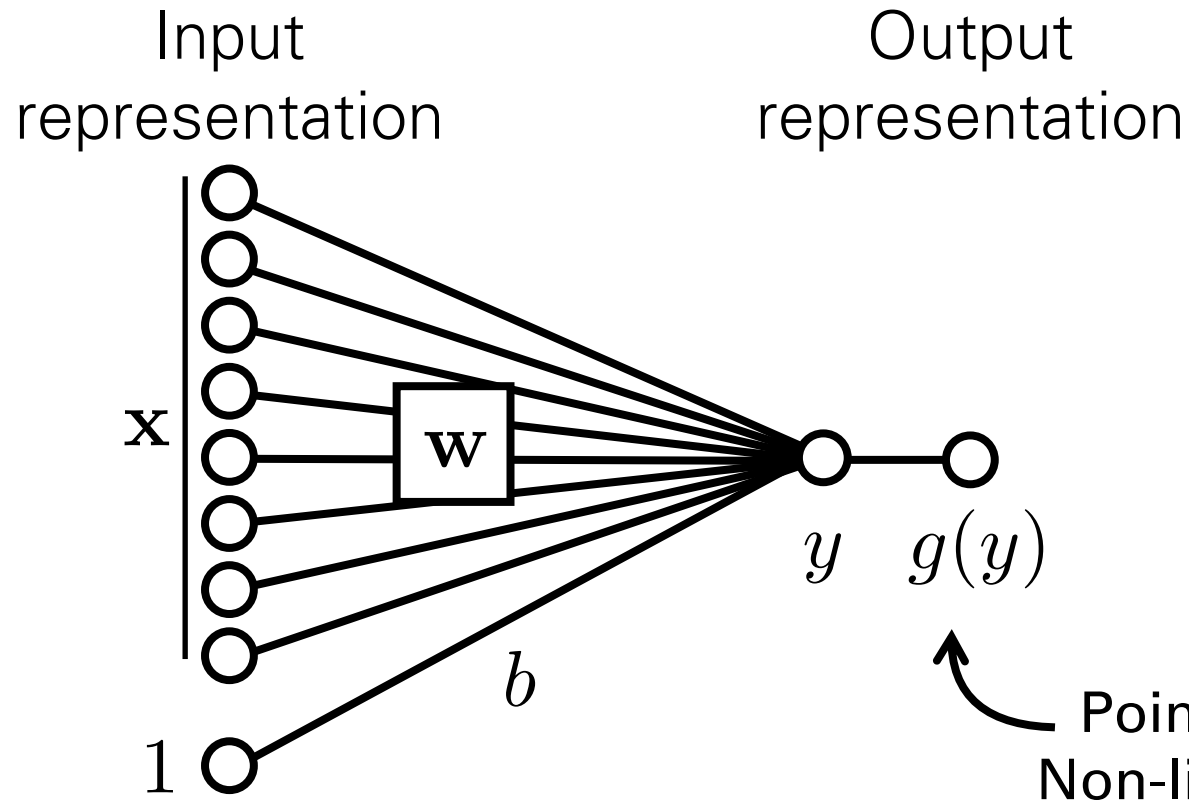
Linear layer



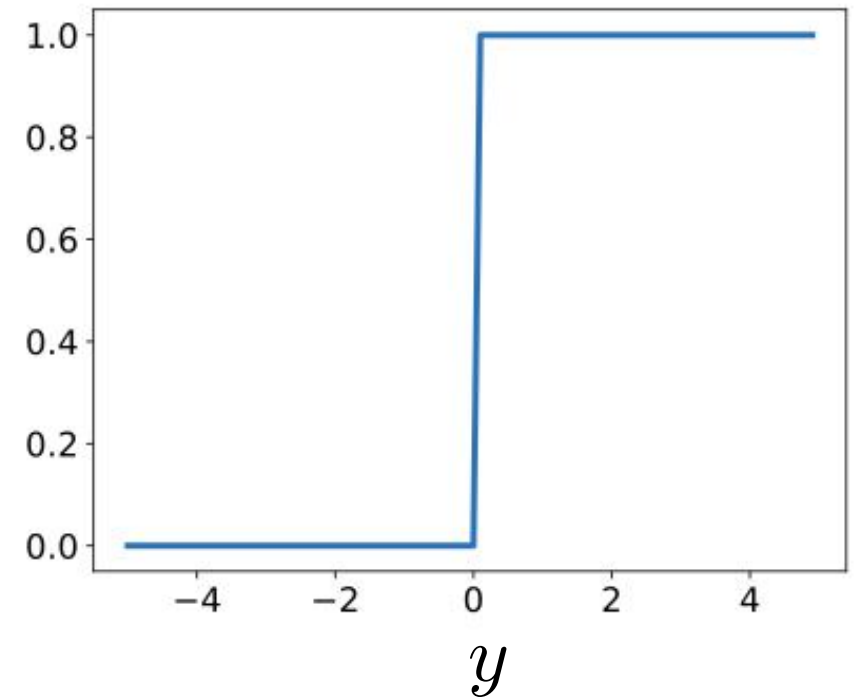
$$f_{\mathbf{w},b}(\mathbf{x}) = \mathbf{x}^T \mathbf{w} + b$$

Computation in a neural net

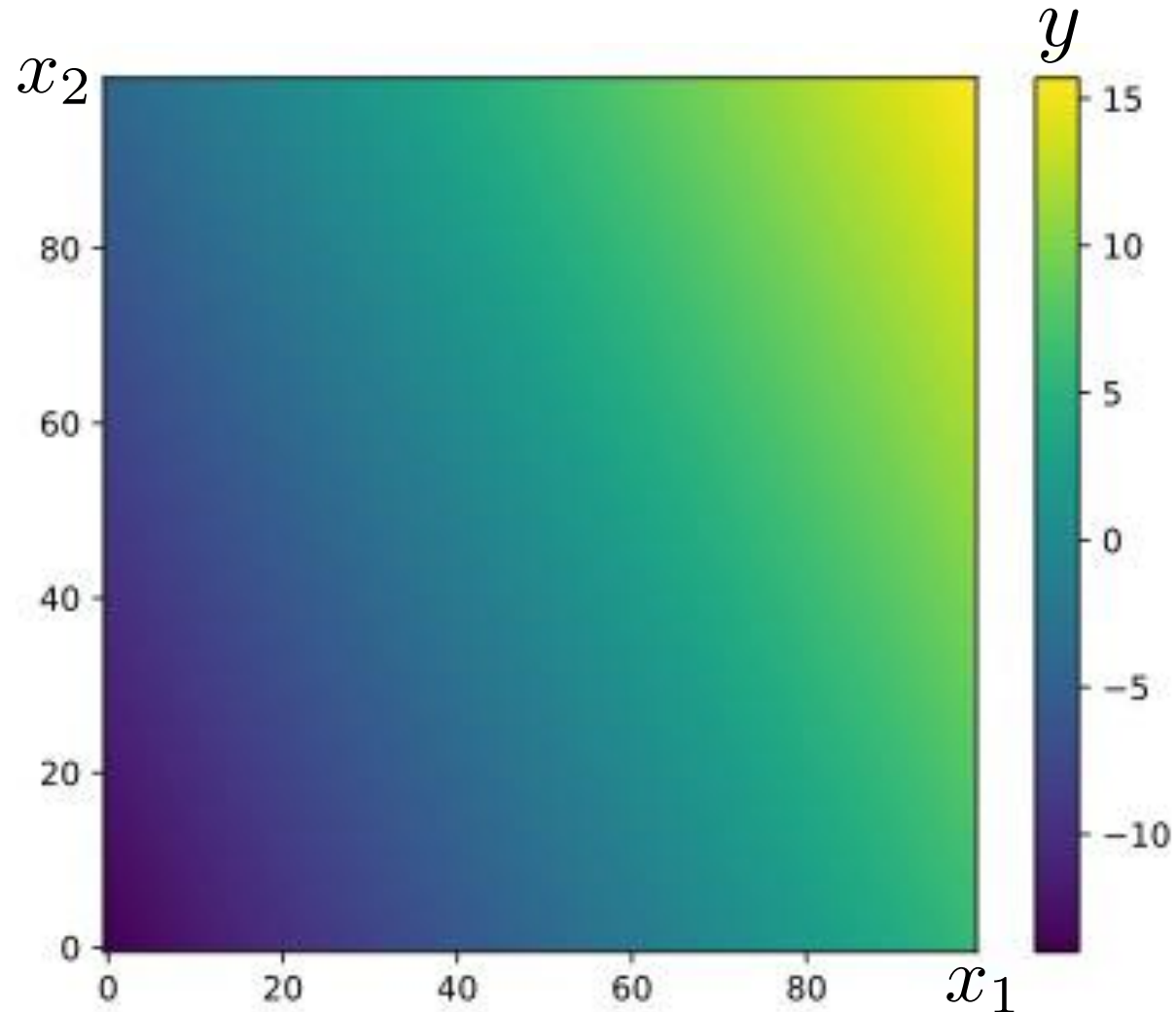
“Perceptron”



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

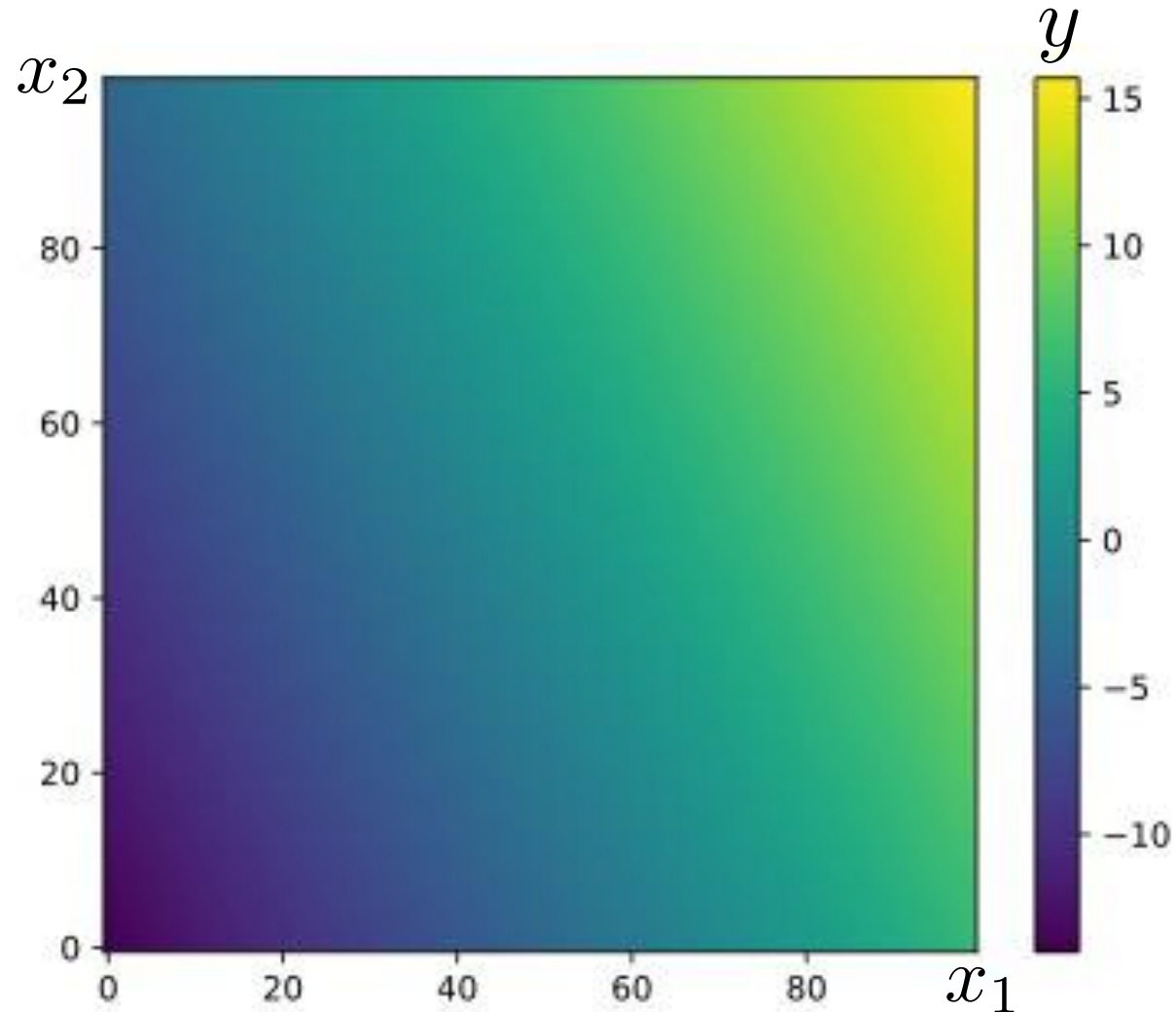


Example: linear classification with a perceptron



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

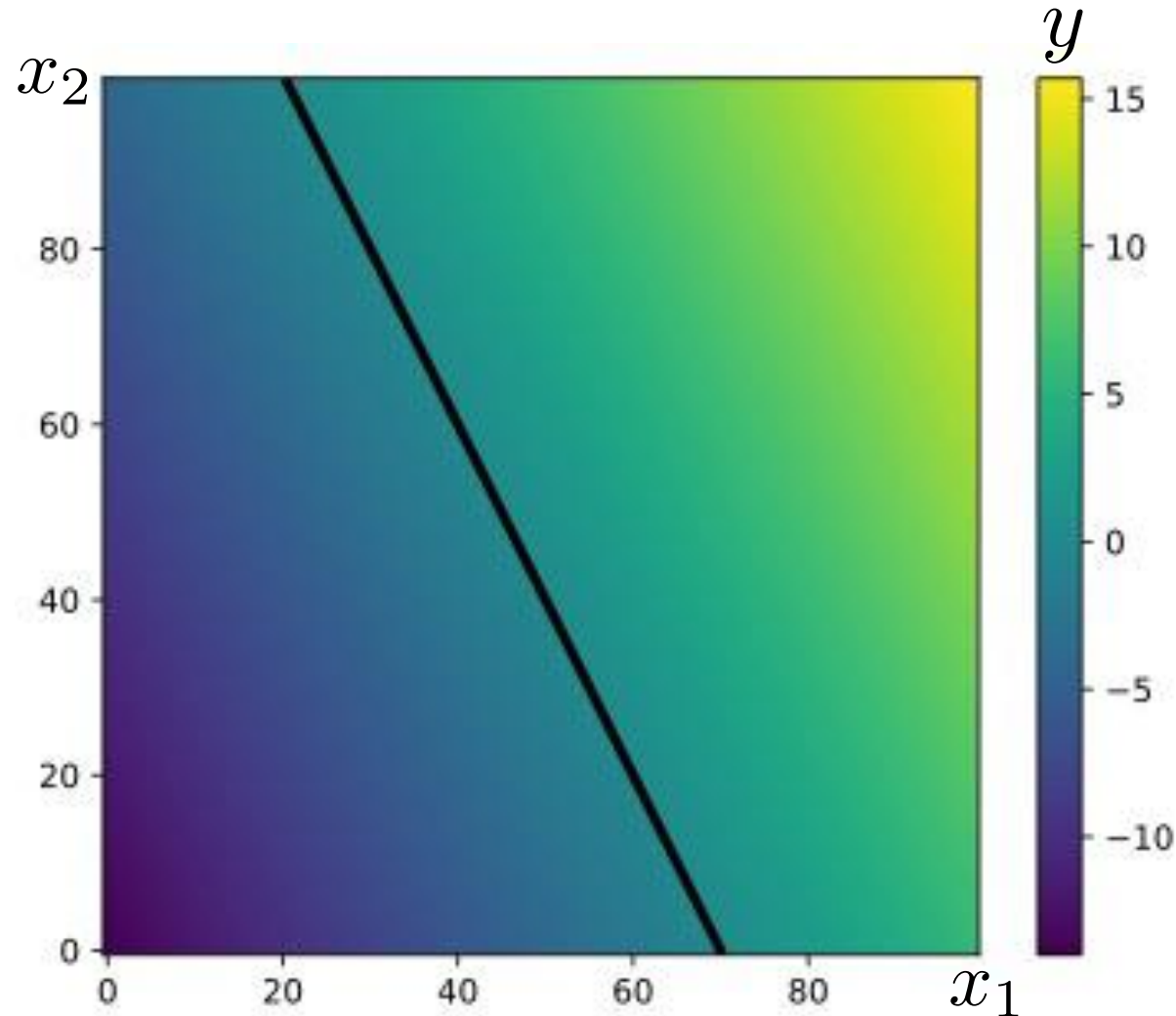
Example: linear classification with a perceptron



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

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

Example: linear classification with a perceptron

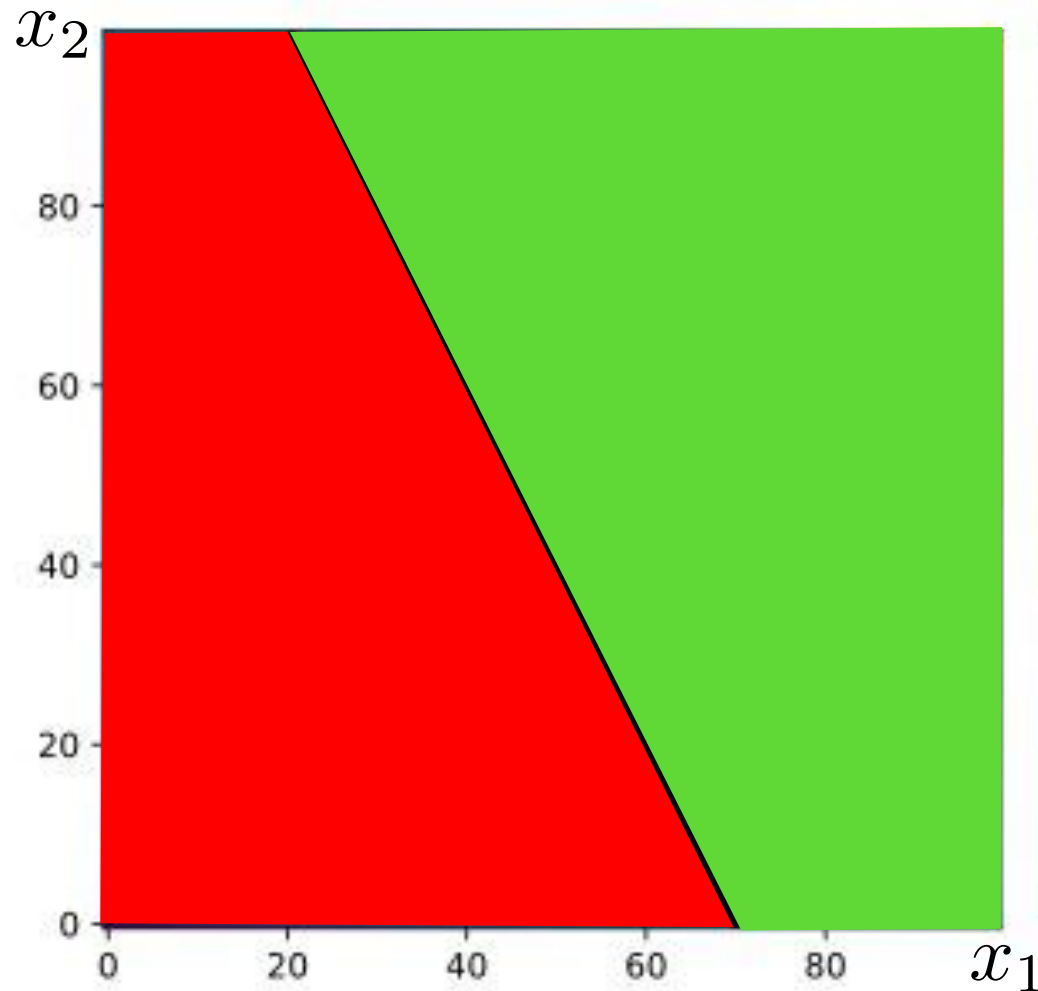


$$y = \mathbf{x}^T \mathbf{w} + b$$

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

Example: linear classification with a perceptron

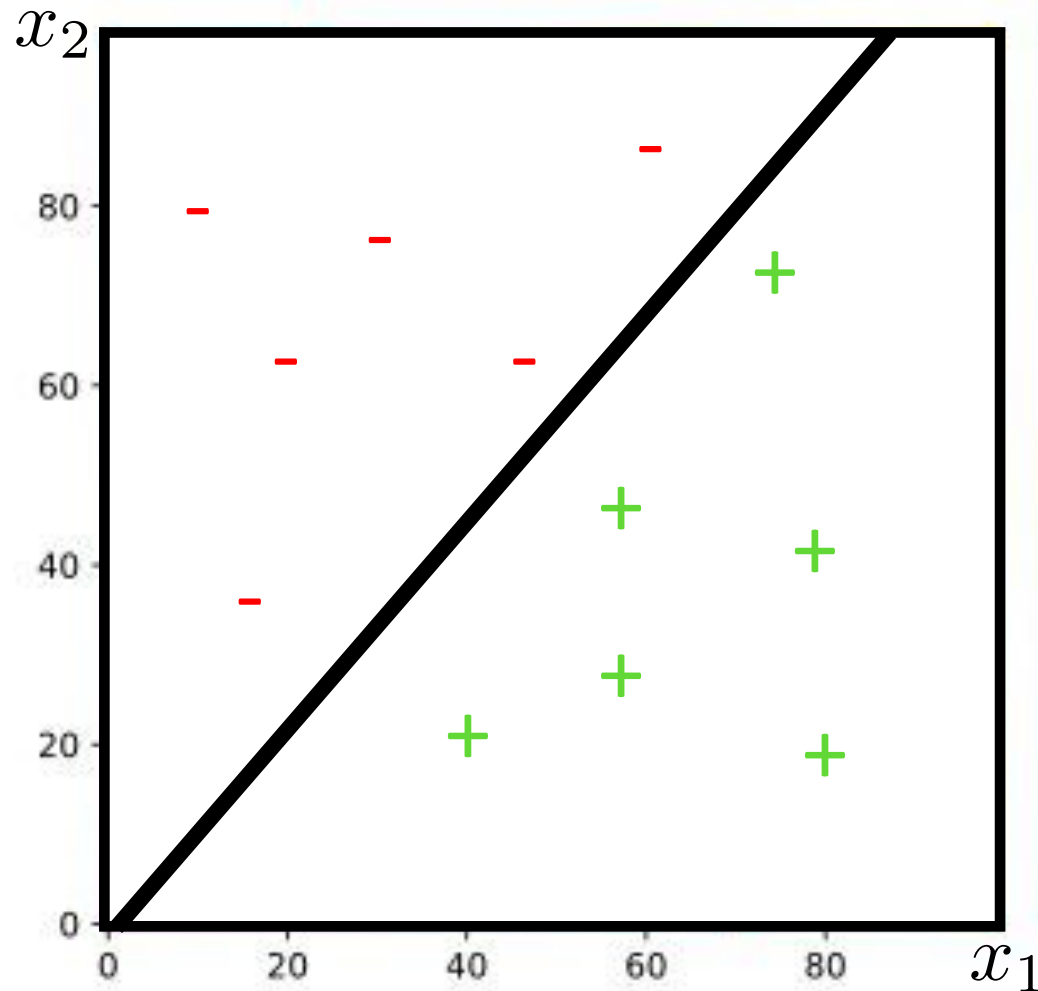
$g(y)$



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

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

Example: linear classification with a perceptron

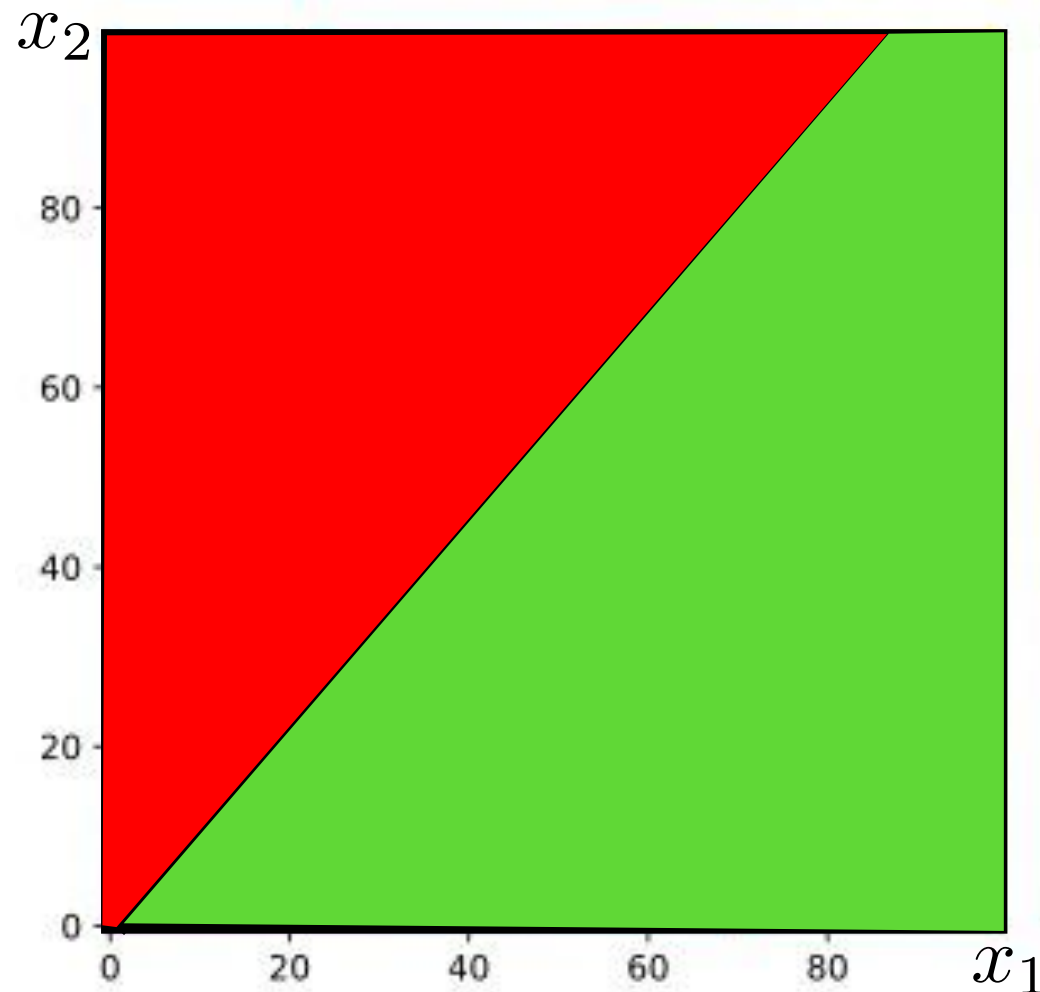


$$\hat{y} = \mathbf{x}^T \mathbf{w} + b$$

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

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

Example: linear classification with a perceptron

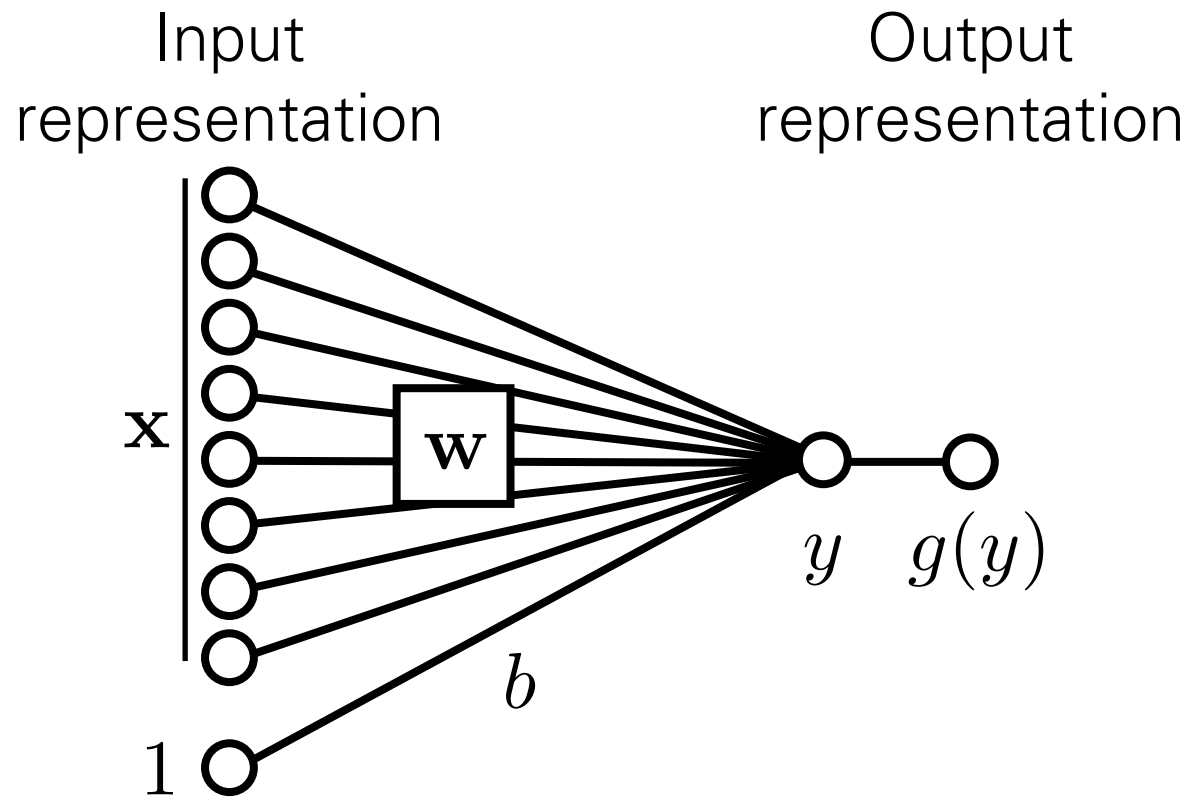


$$\hat{y} = \mathbf{x}^T \mathbf{w} + b$$

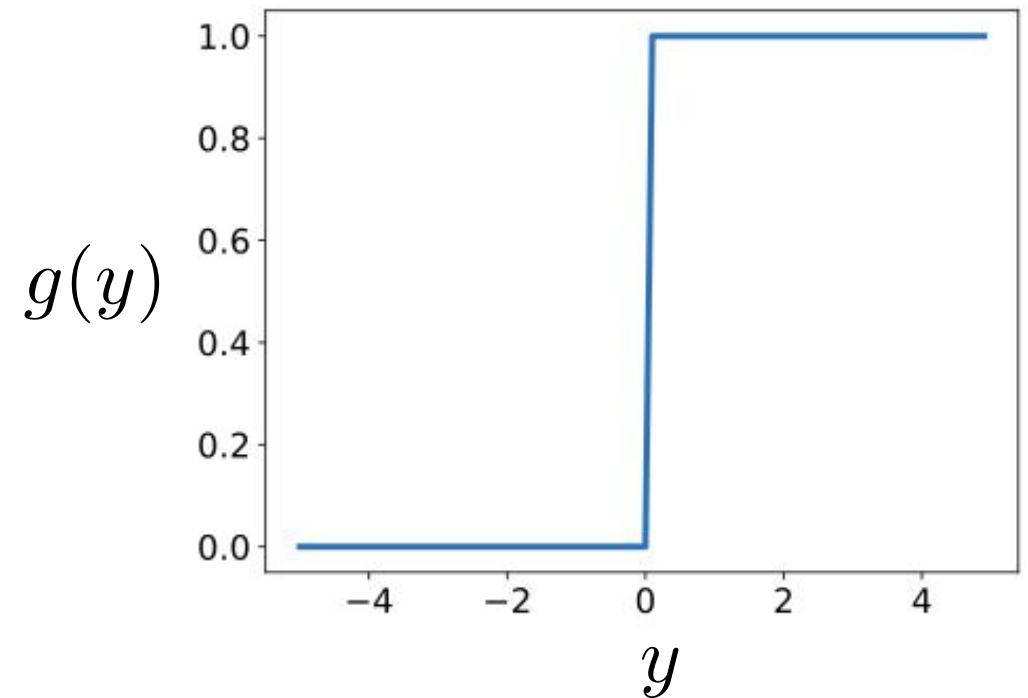
$$g(\hat{y}) = \begin{cases} 1, & \text{if } \hat{y} > 0 \\ 0, & \text{otherwise} \end{cases}$$

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

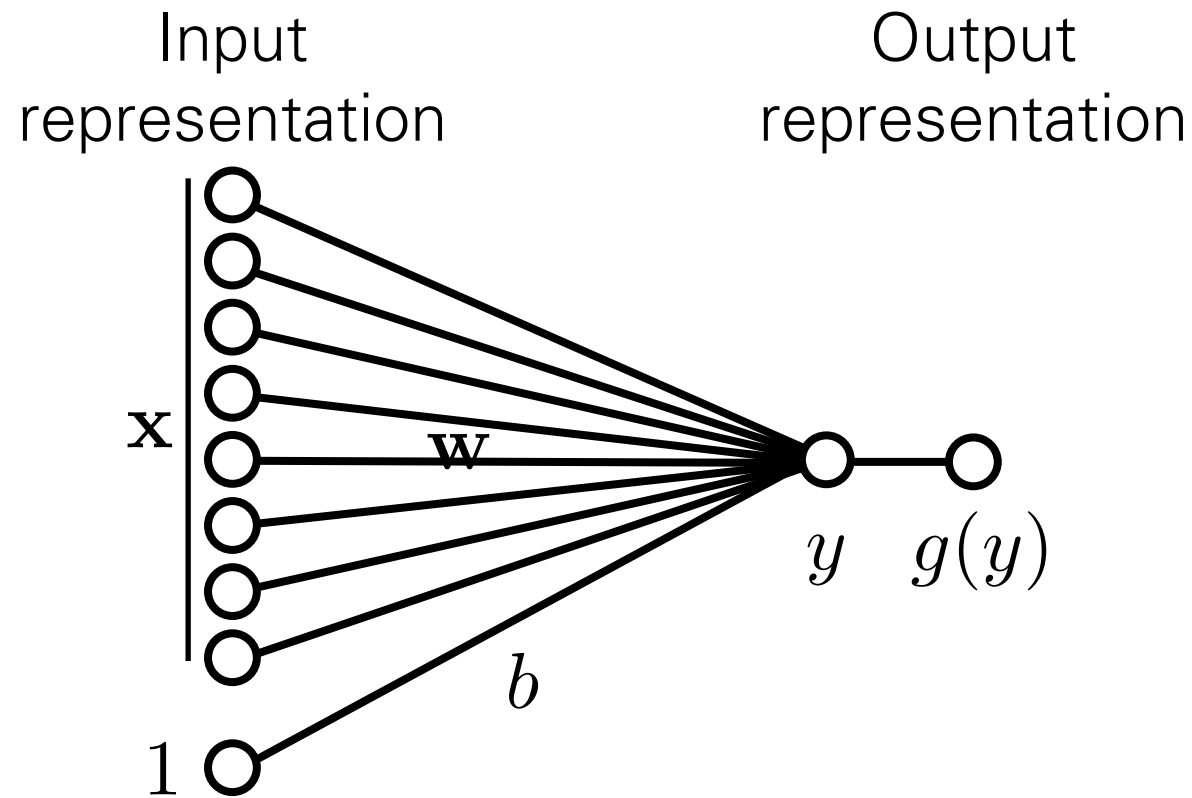
Computation in a neural net



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



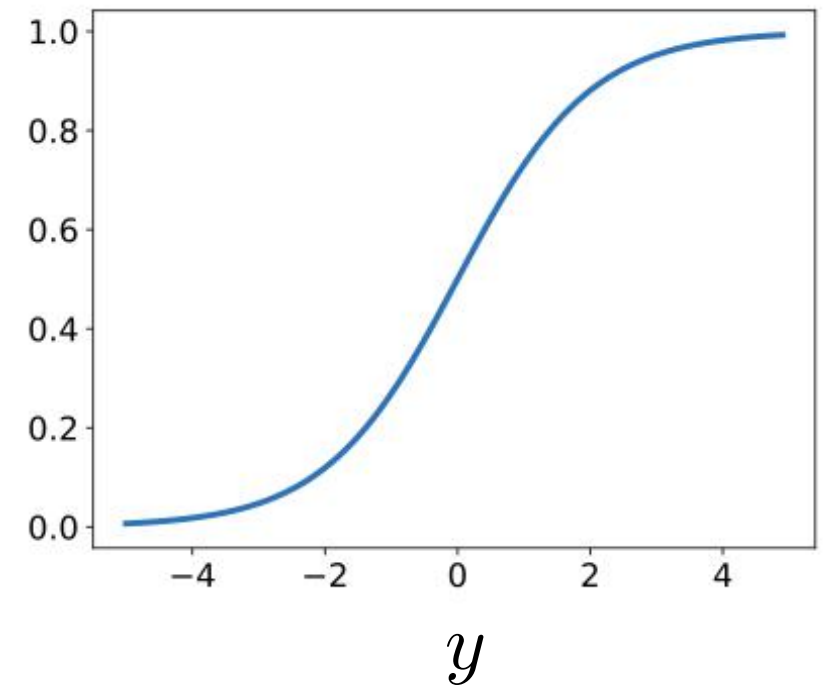
Computation in a neural net – nonlinearity



Sigmoid

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

$g(y)$

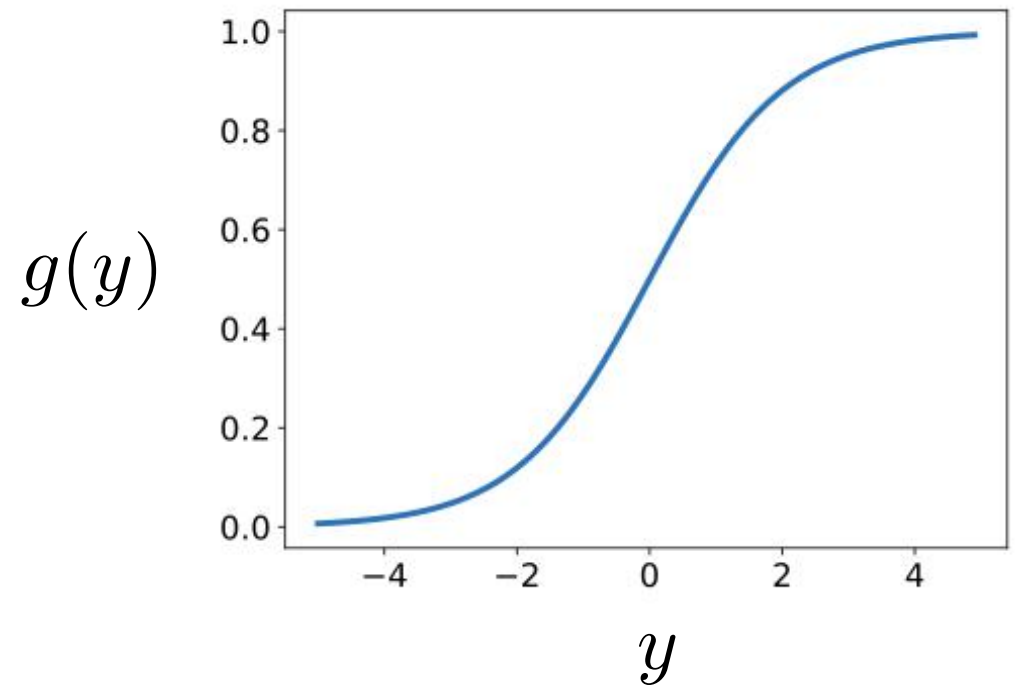


Computation in a neural net – nonlinearity

- 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

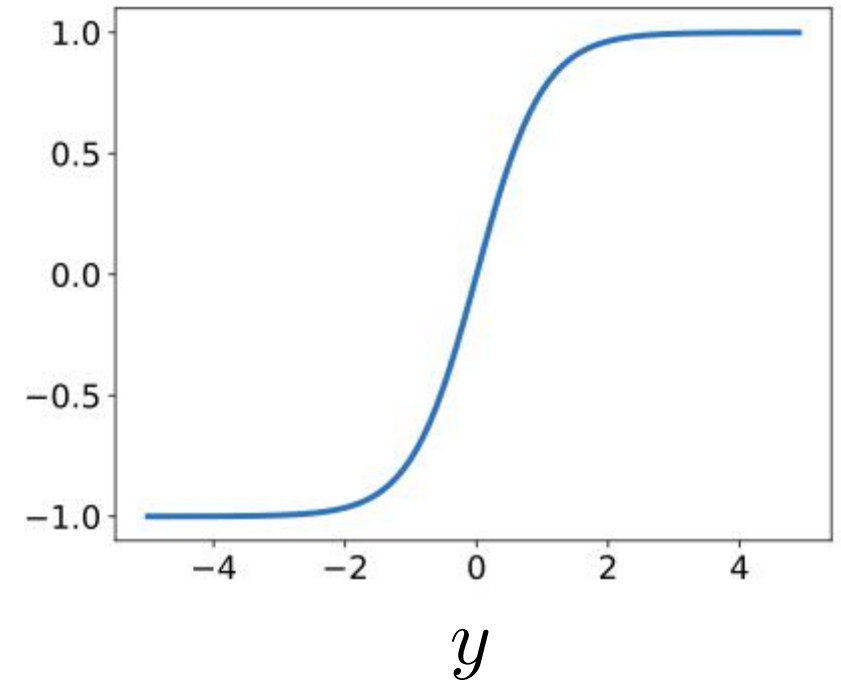
- Bounded between $[-1, +1]$
- Saturation for large +/- inputs
- Gradients go to zero
- Outputs centered at 0
- Preferable to sigmoid

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

Tanh

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

$g(y)$

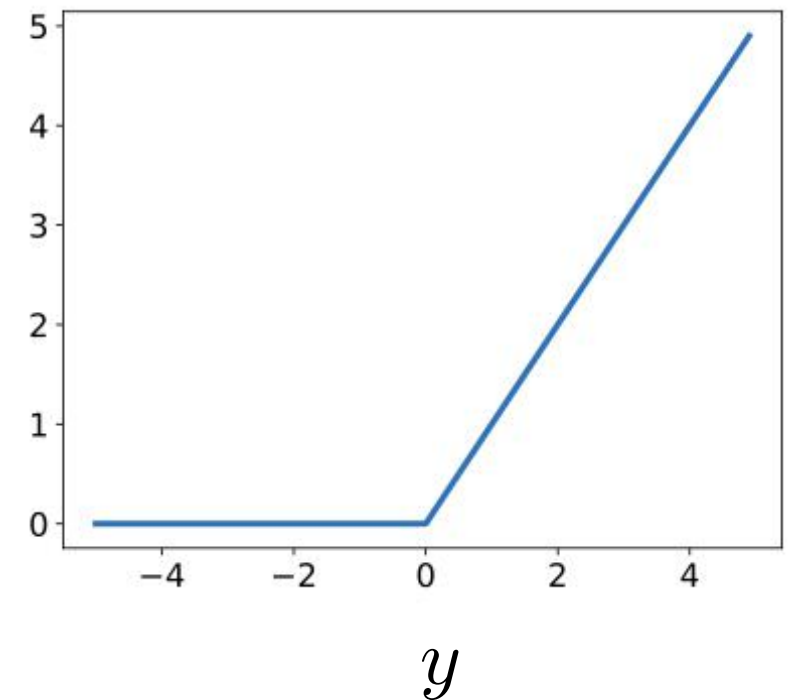


Computation in a neural net – nonlinearity

- Unbounded output (on positive side)
- Efficient to implement: $\frac{\partial g}{\partial y} = \begin{cases} 0, & \text{if } y < 0 \\ 1, & \text{if } y \geq 0 \end{cases}$
- 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.

Rectified linear unit (ReLU)

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



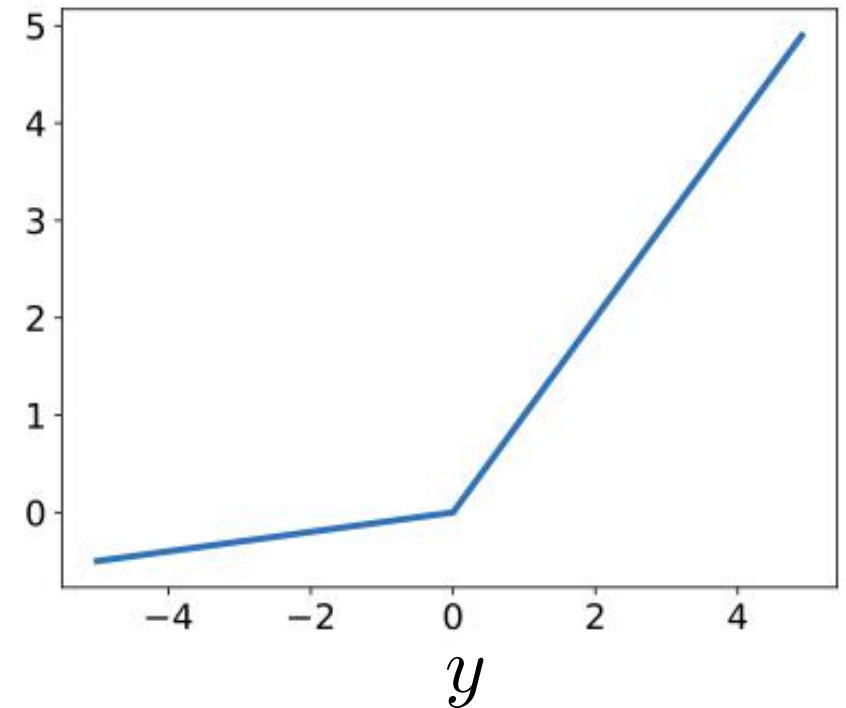
Computation in a neural net – nonlinearity

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

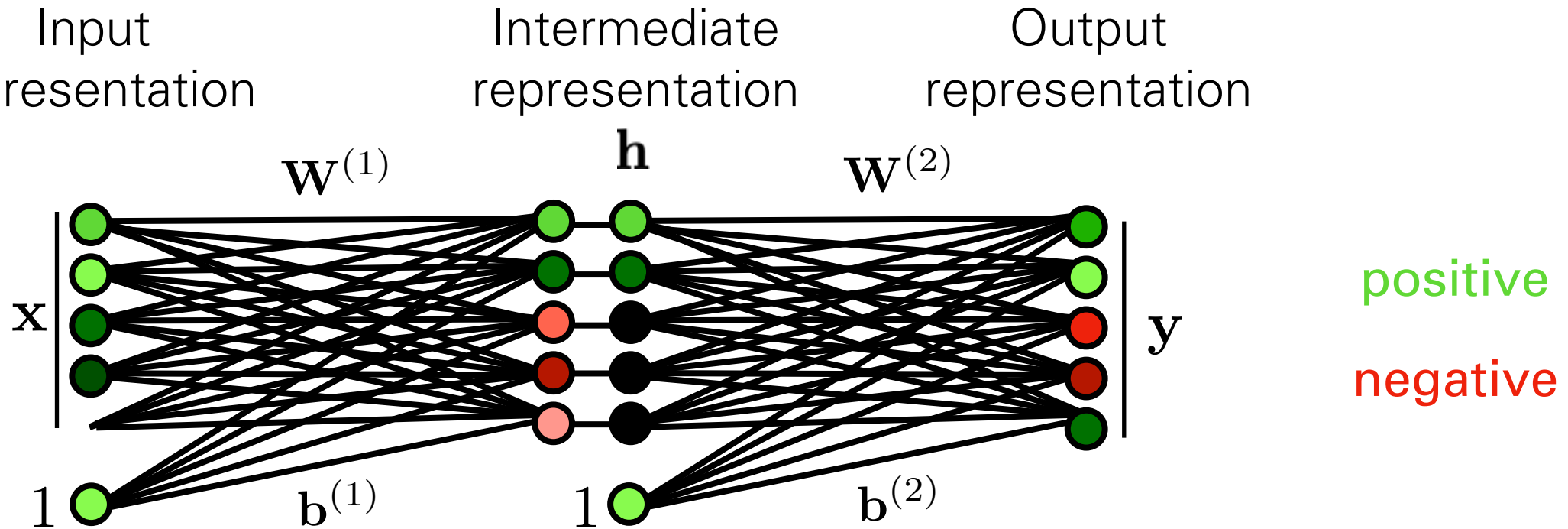
Leaky ReLU

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

$g(y)$



Stacking layers

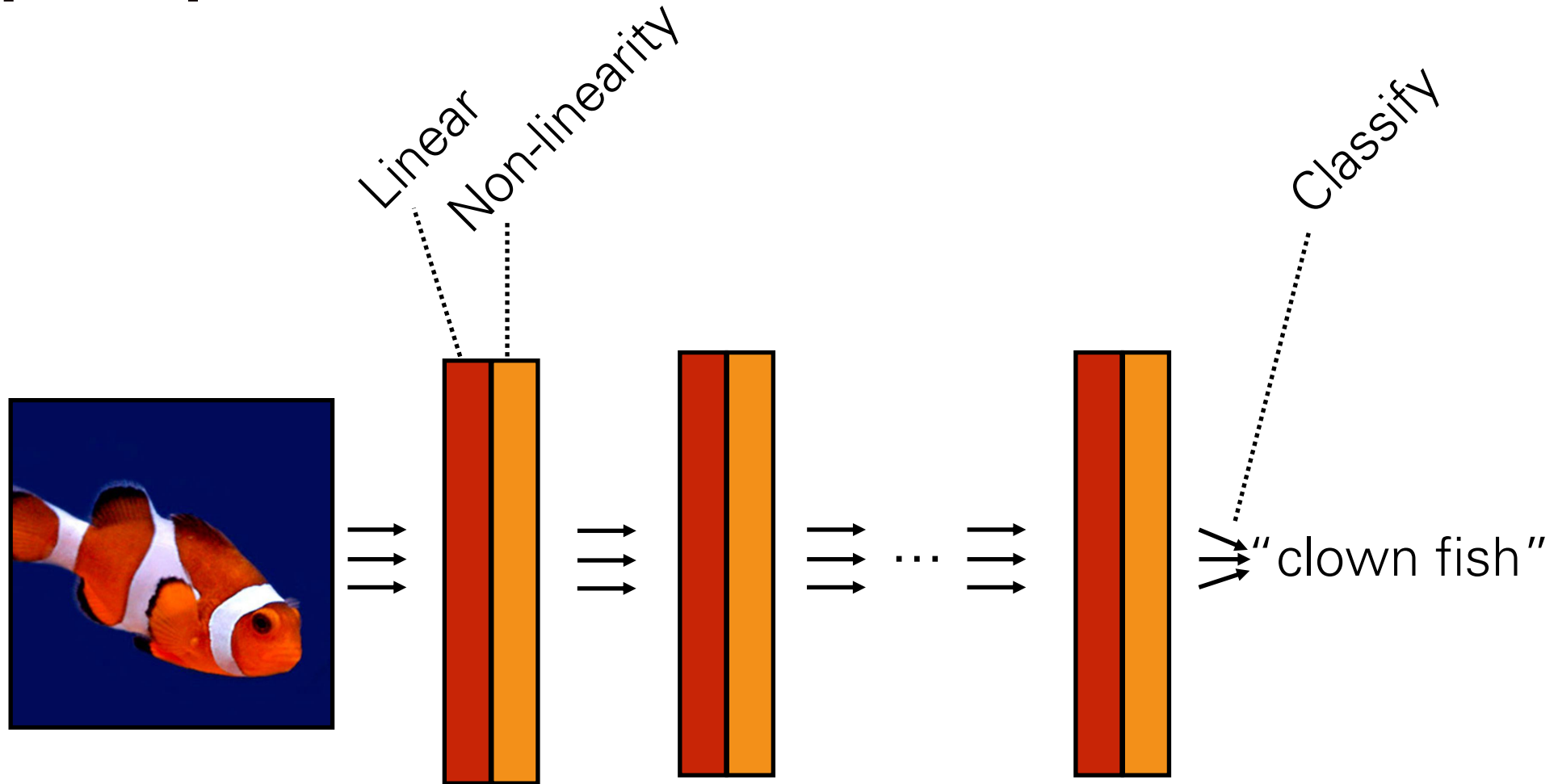


$$\theta = \{\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(L)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(L)}\}$$

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. Neilsen
<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

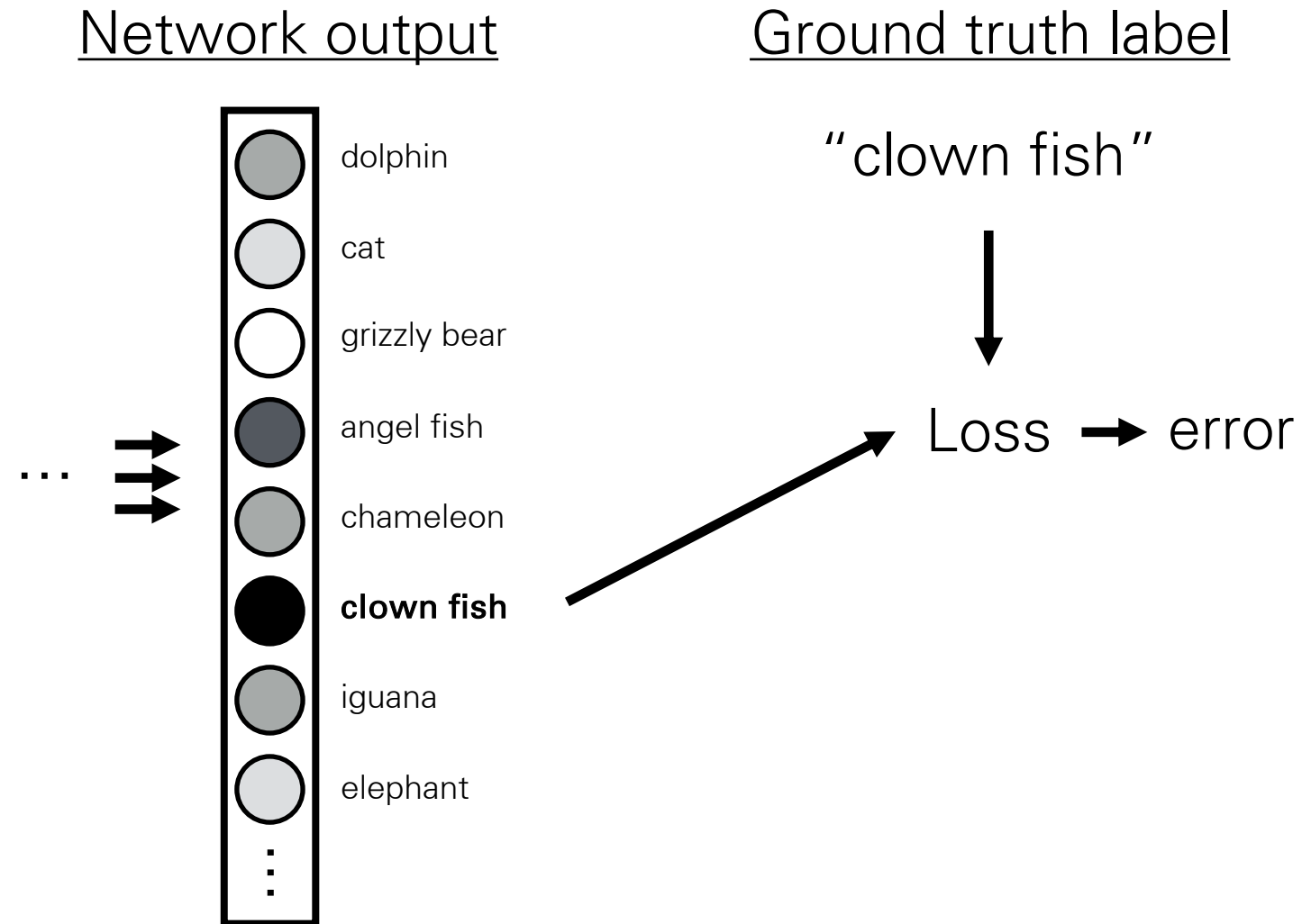


$$f(\mathbf{x}) = f_L(\dots(f_2(f_1(\mathbf{x}))))$$

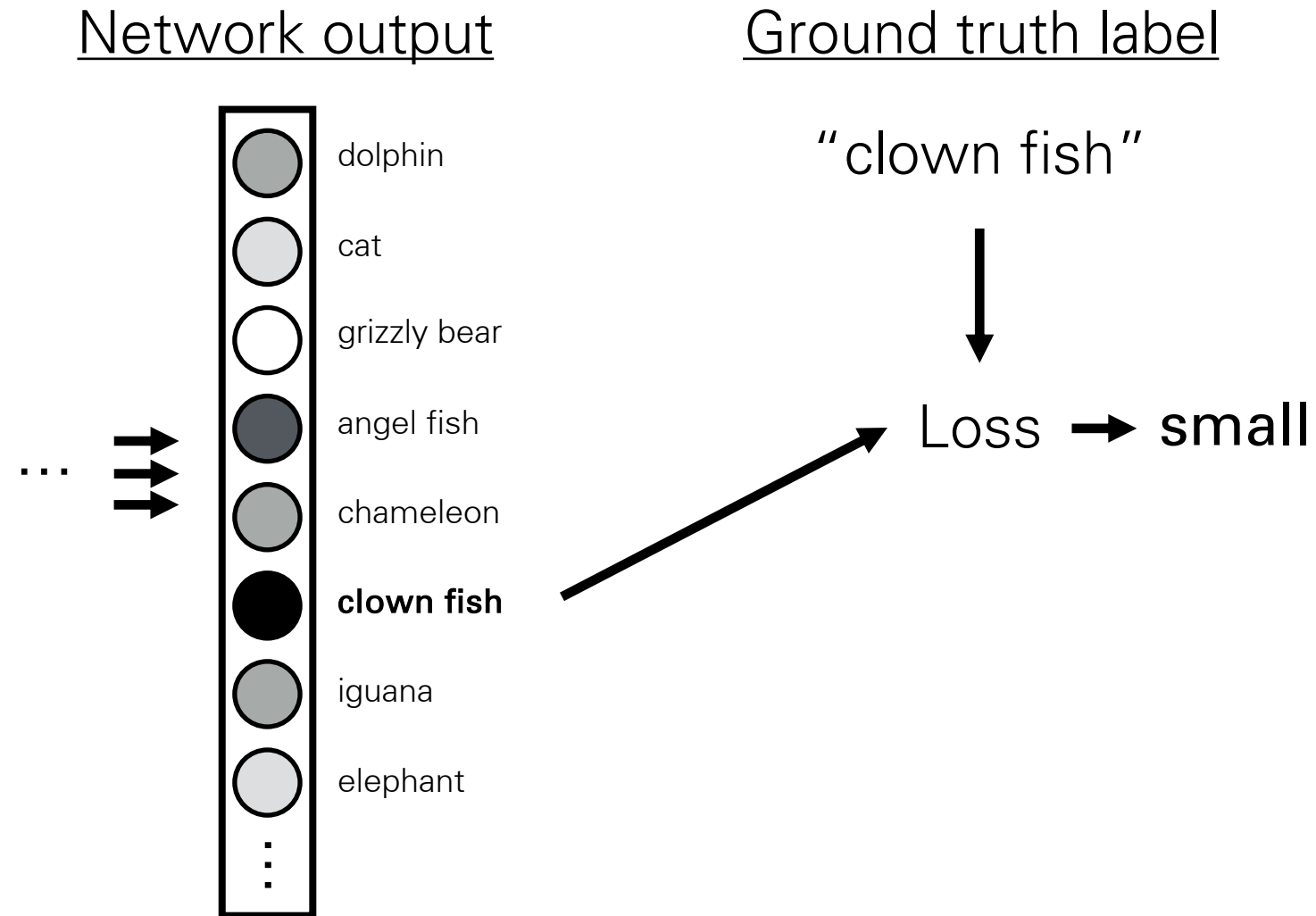
Classifier layer



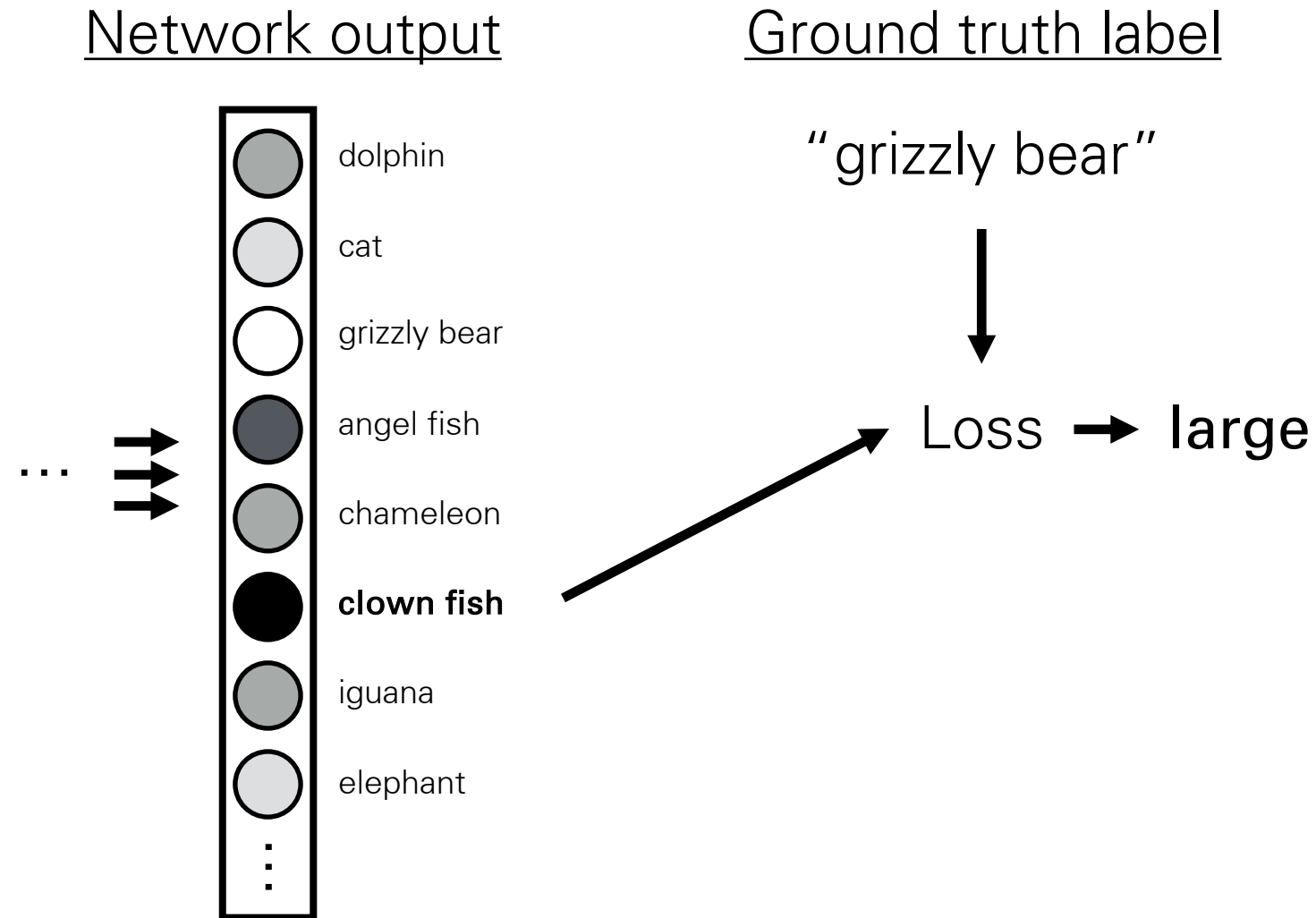
Loss function

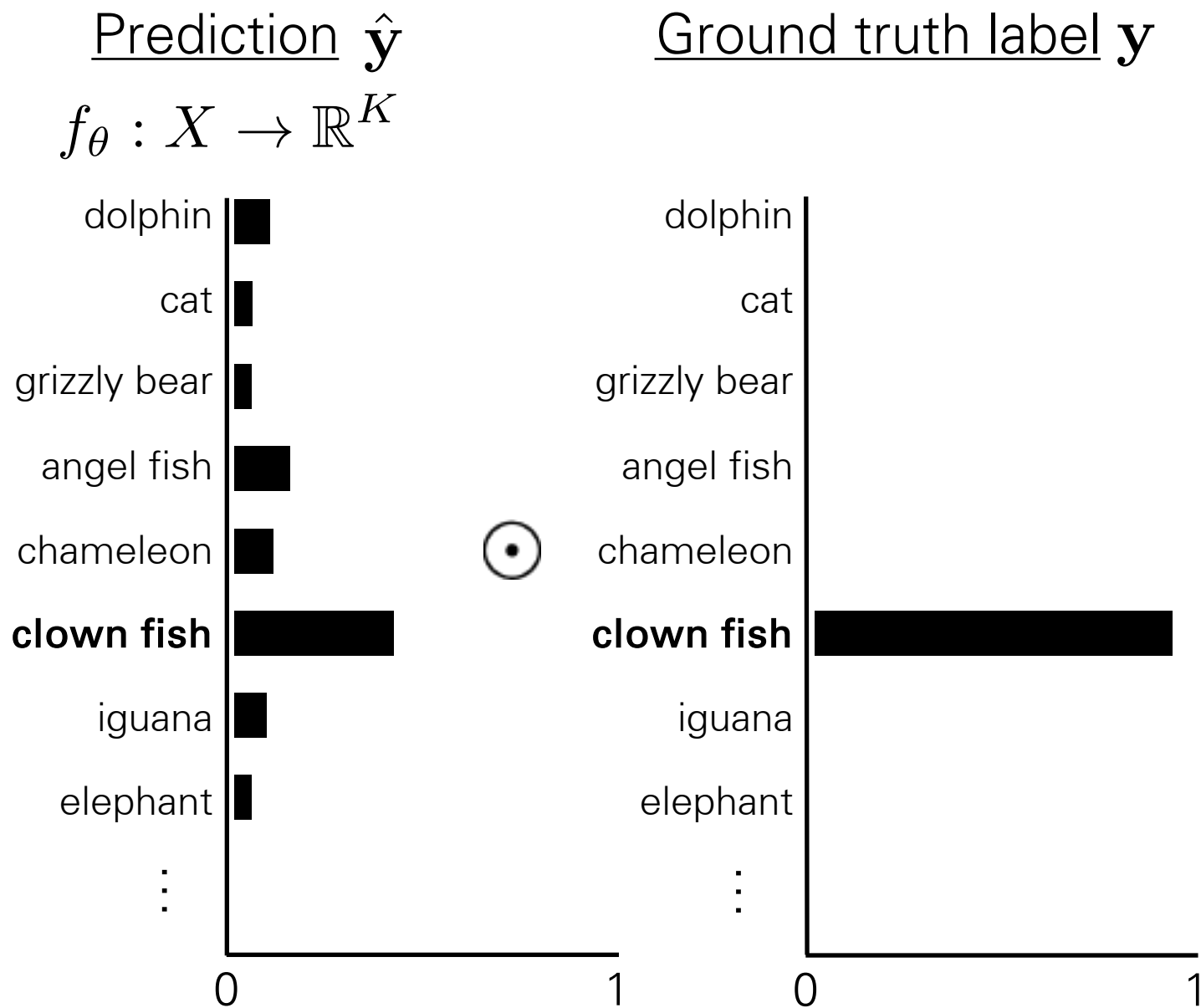
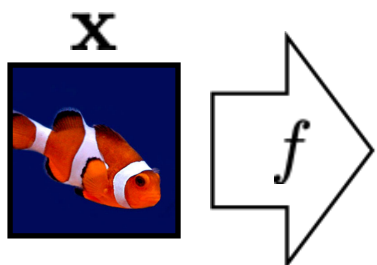


Loss function



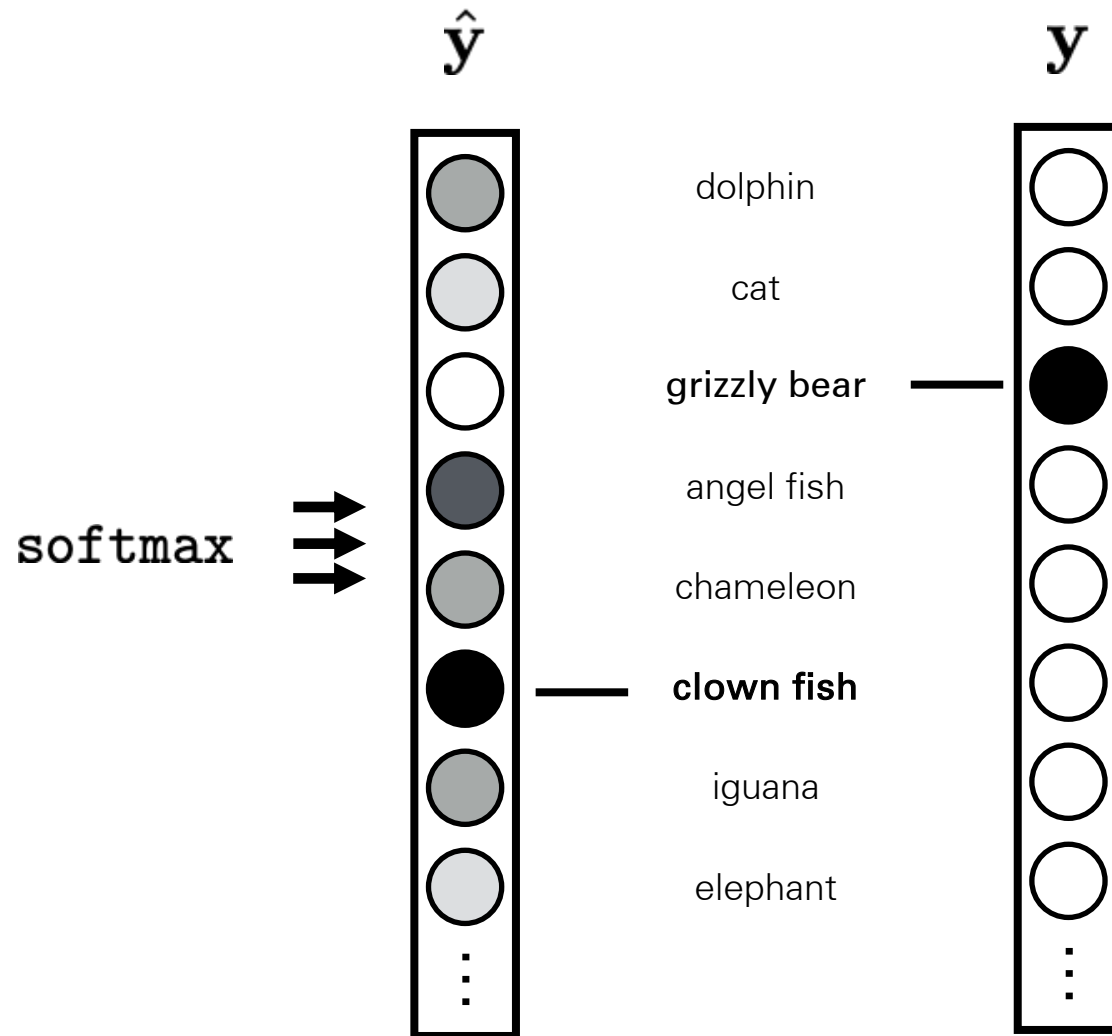
Loss function





Network output

Ground truth label

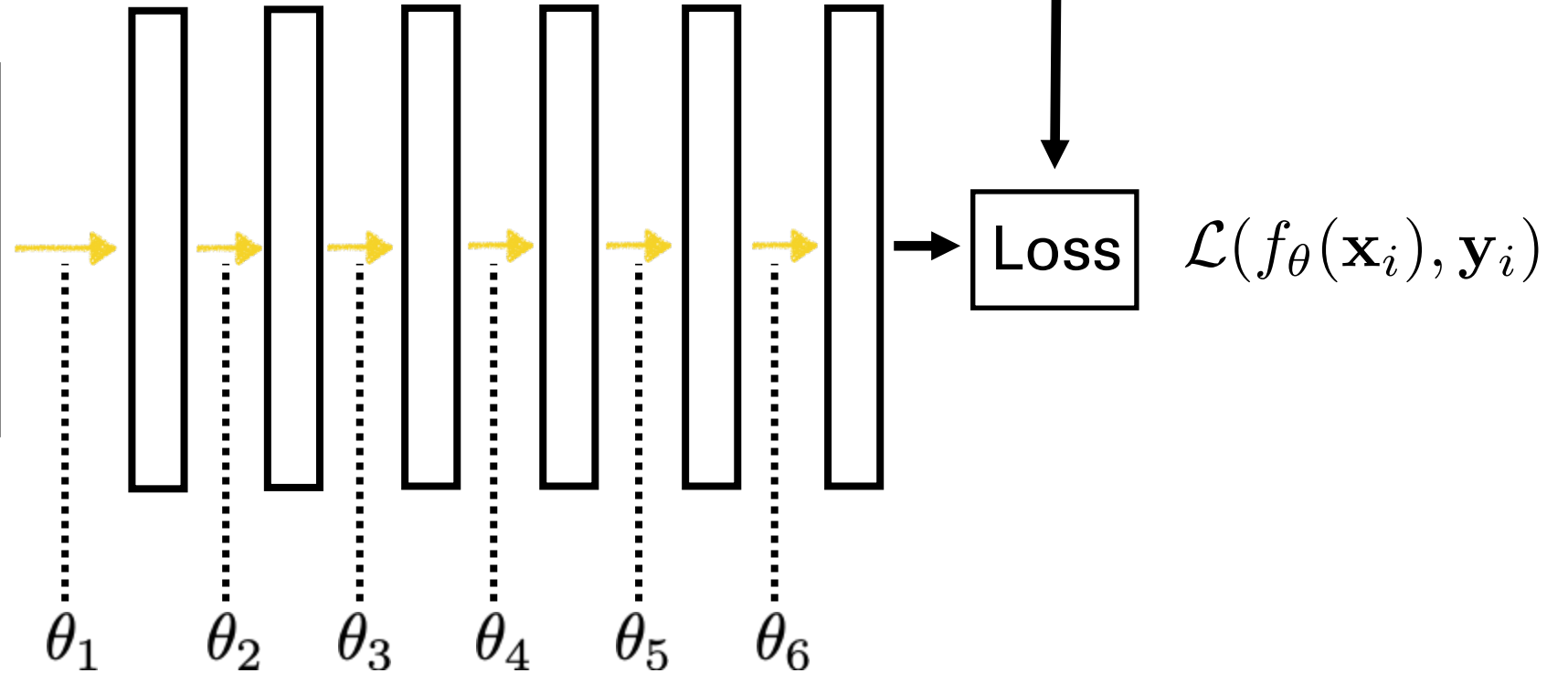
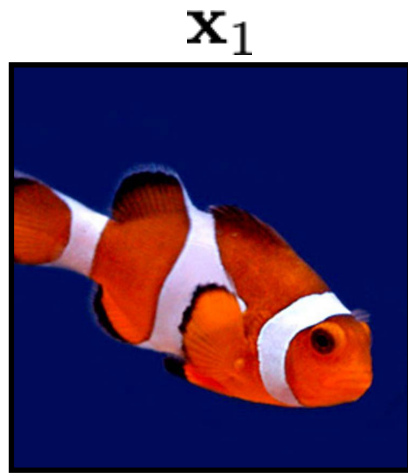


Probability of the observed data under the model

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

Deep learning

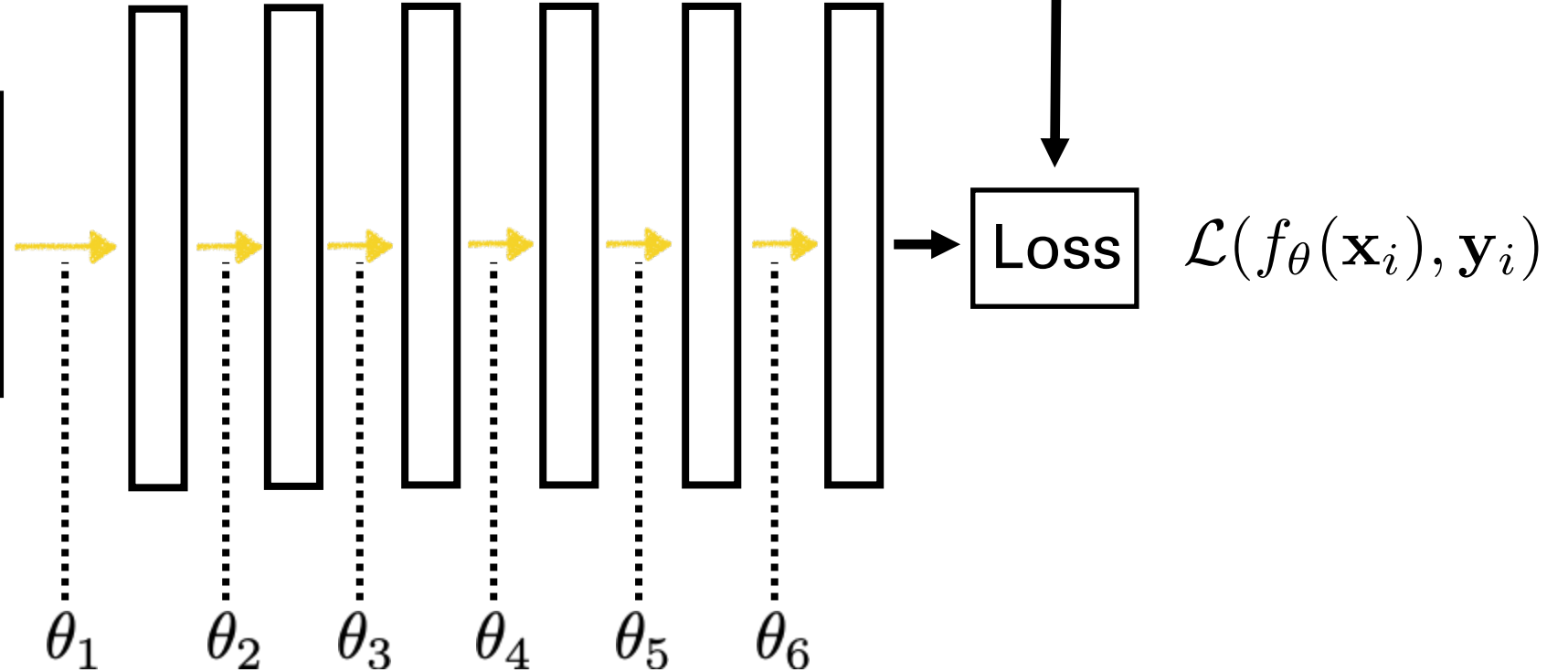
y_1
"clown fish"



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

Deep learning

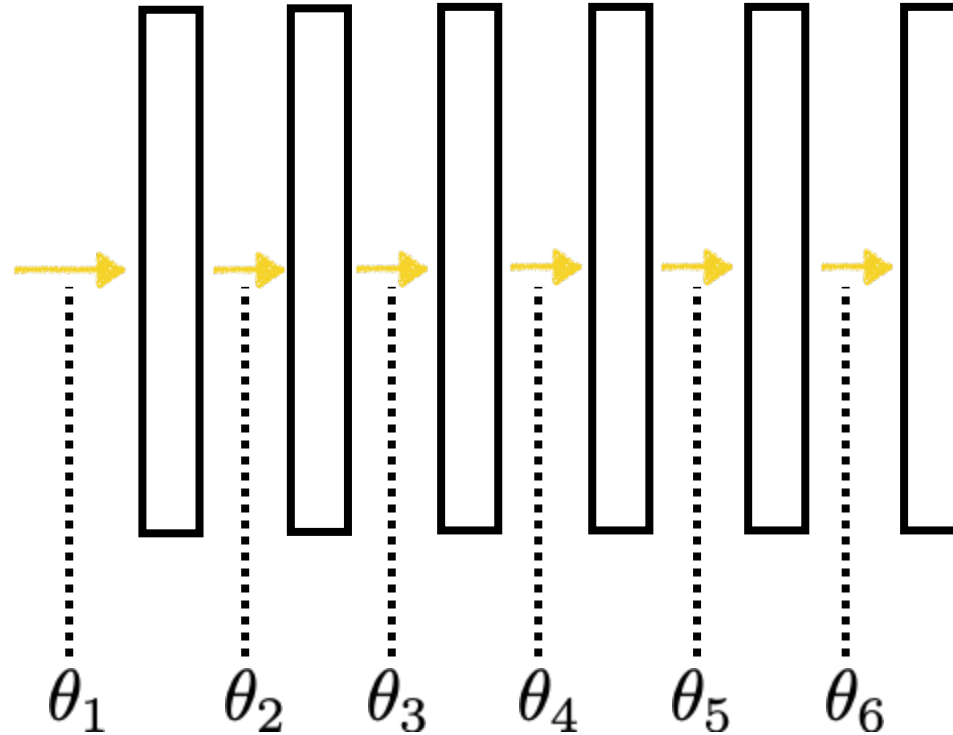
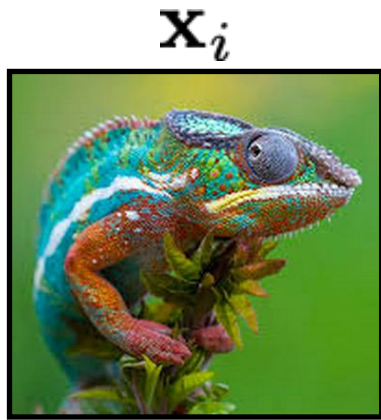
"grizzly bear"



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

Deep learning

y_i
"chameleon"



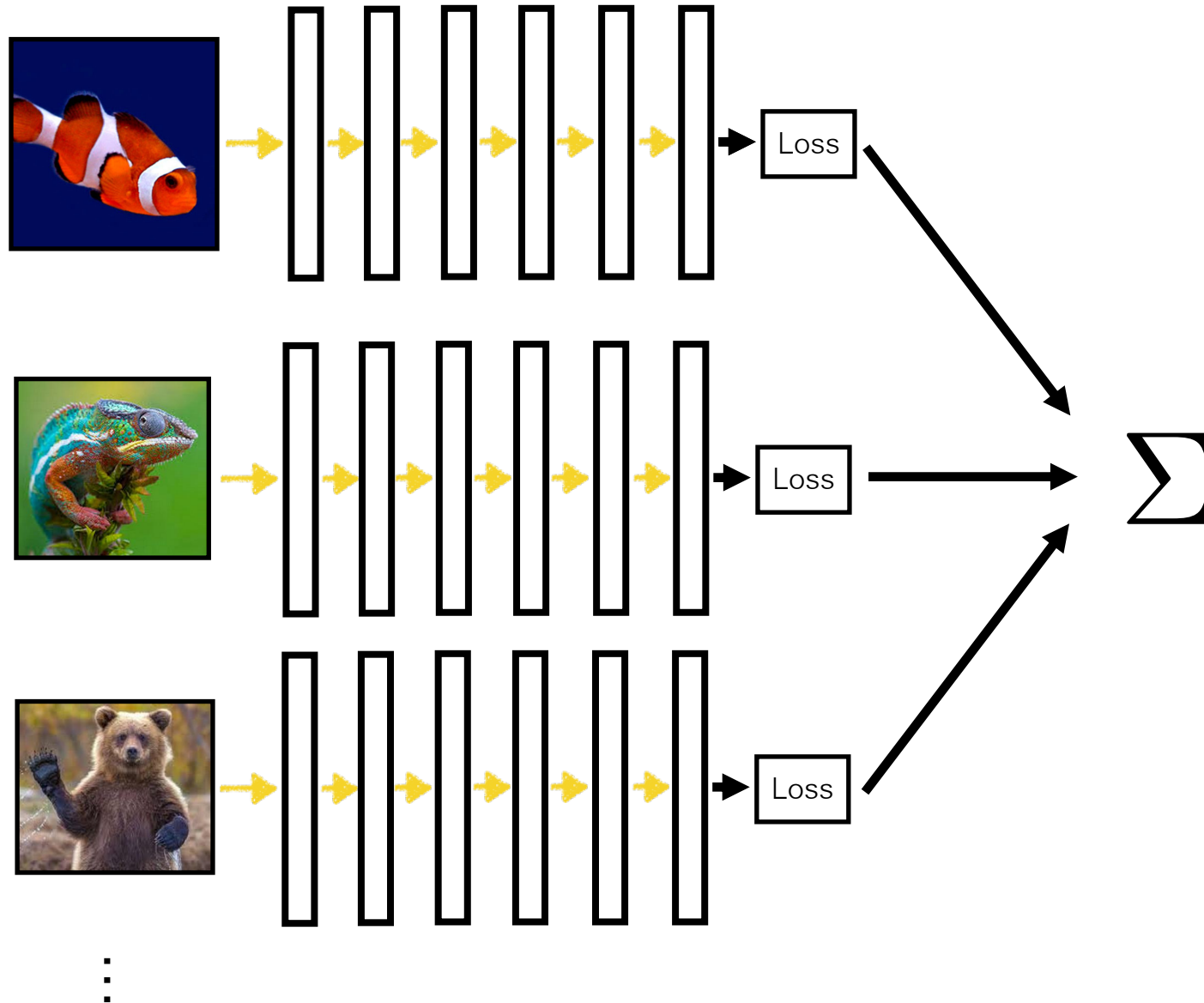
Loss

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

Learned

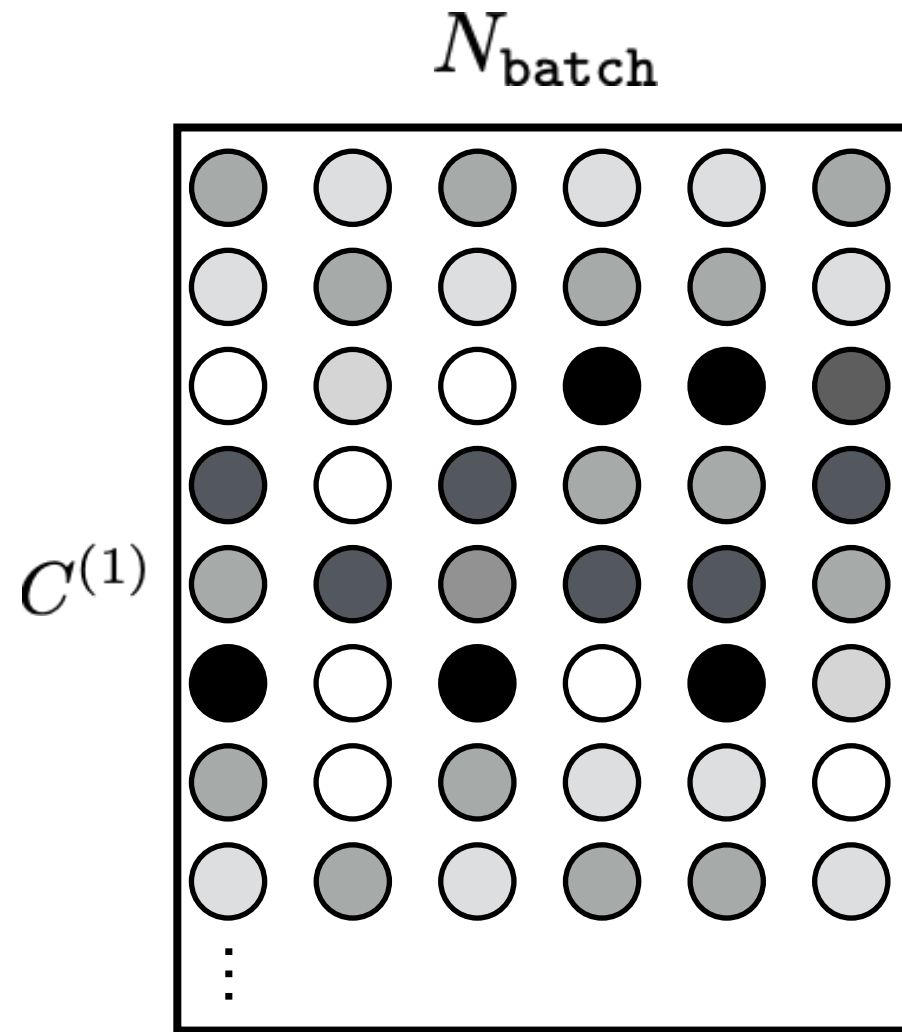
$$\theta^* = \arg \min_{\theta} \sum_{i=1}^N \mathcal{L}(f_{\theta}(\mathbf{x}_i), \mathbf{y}_i)$$

Batch (parallel) processing



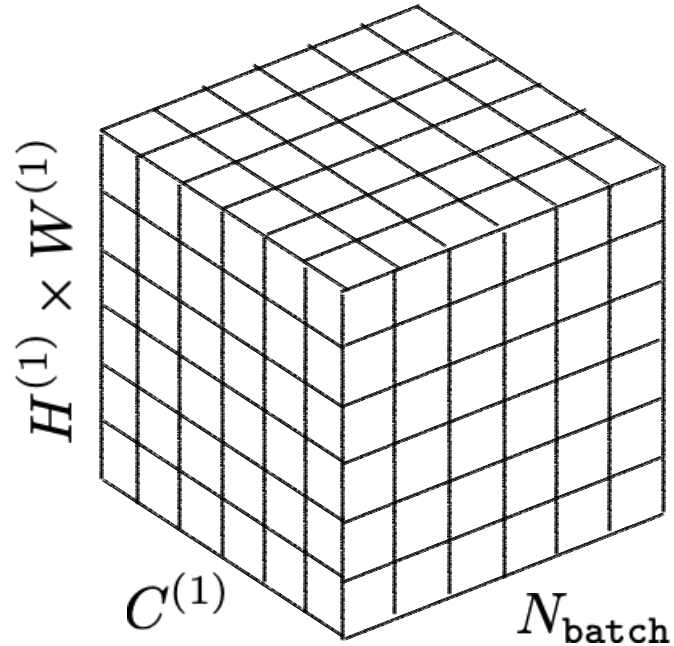
Tensors

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

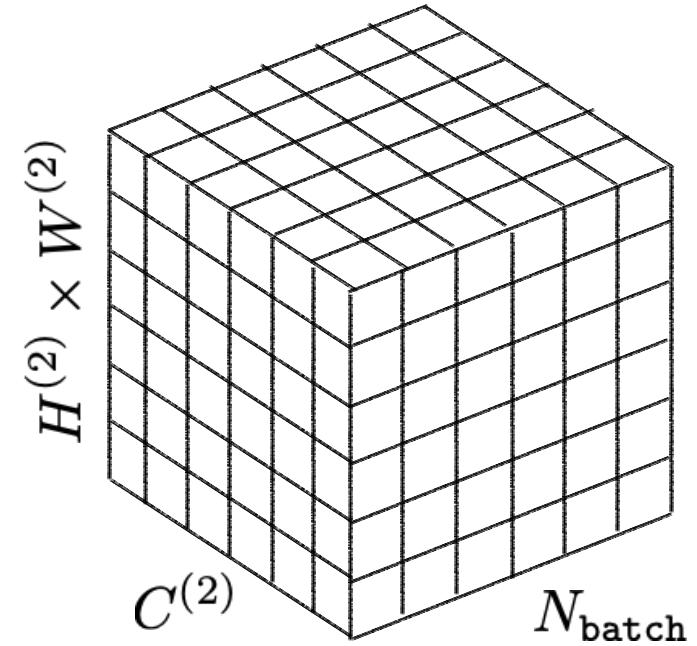


"Tensor flow"

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



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



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

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

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



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.

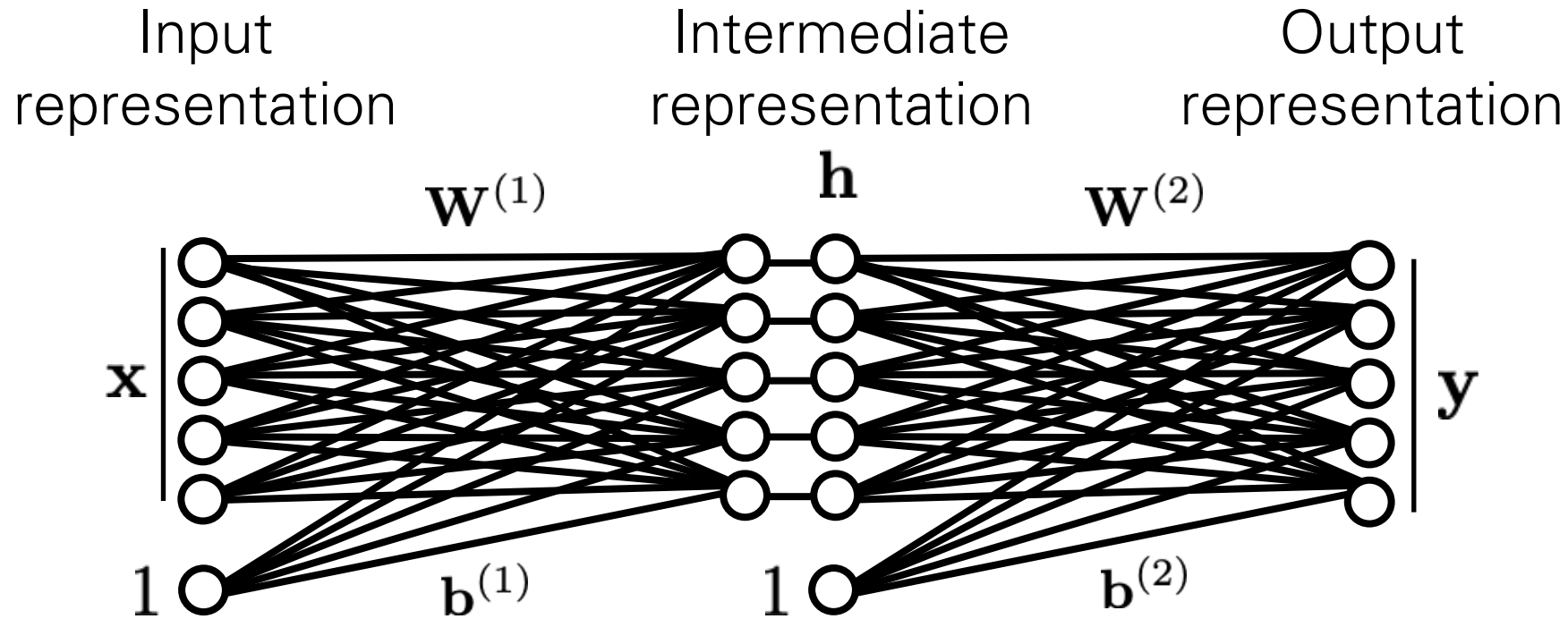
Recall: regularized least squares

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

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

“We prefer to keep weights small.”

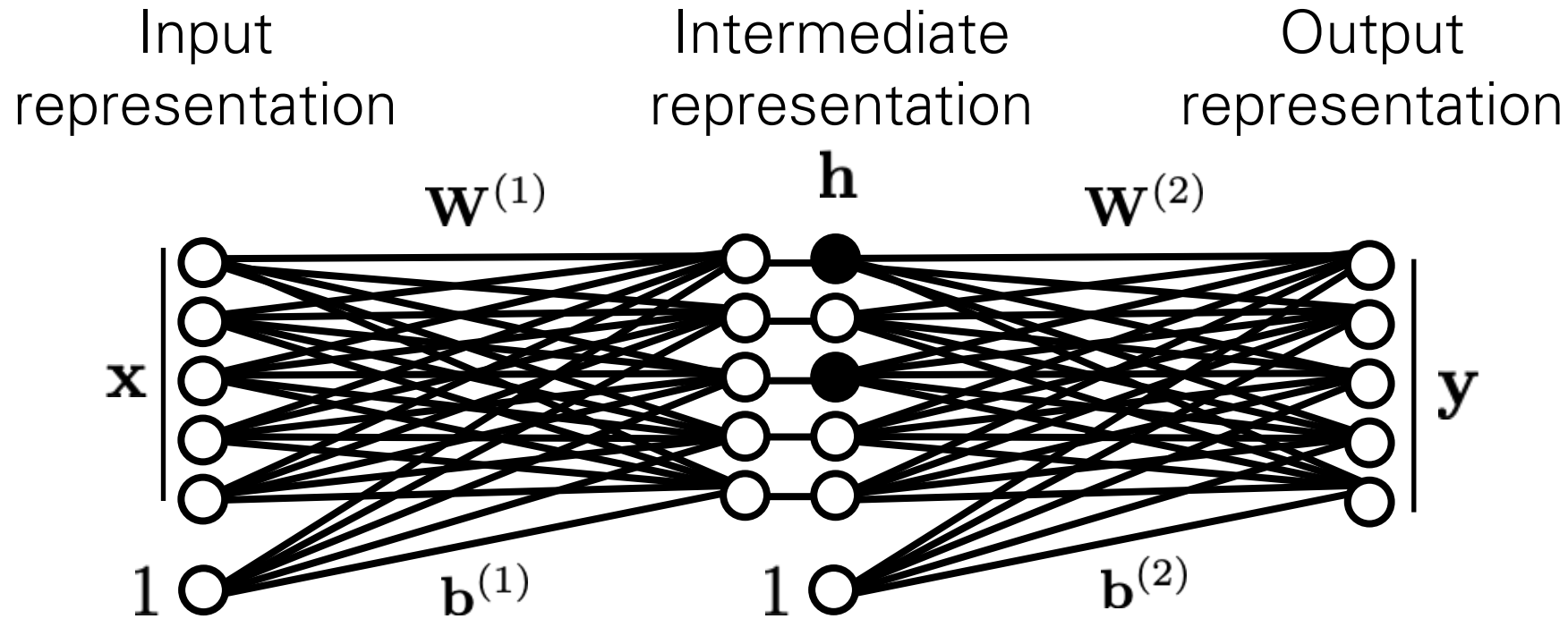
Dropout



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

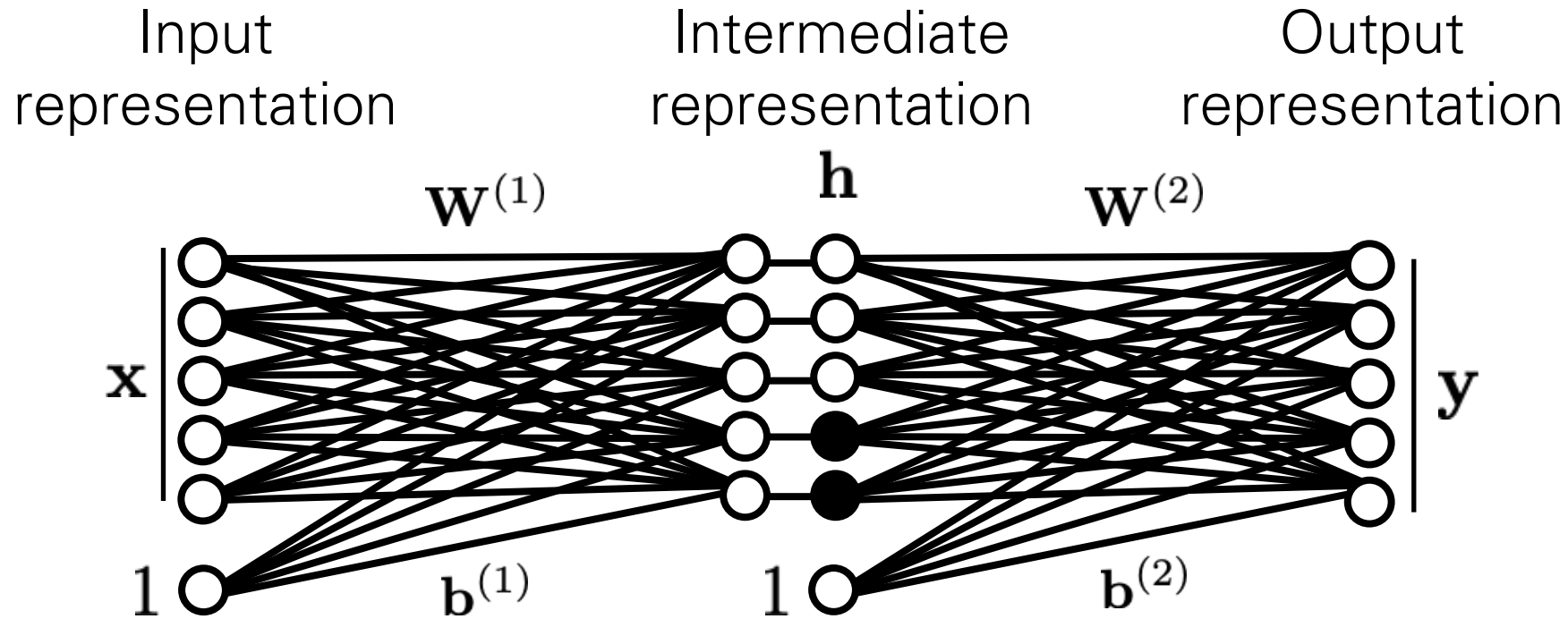
Randomly zero out
hidden units.

Dropout



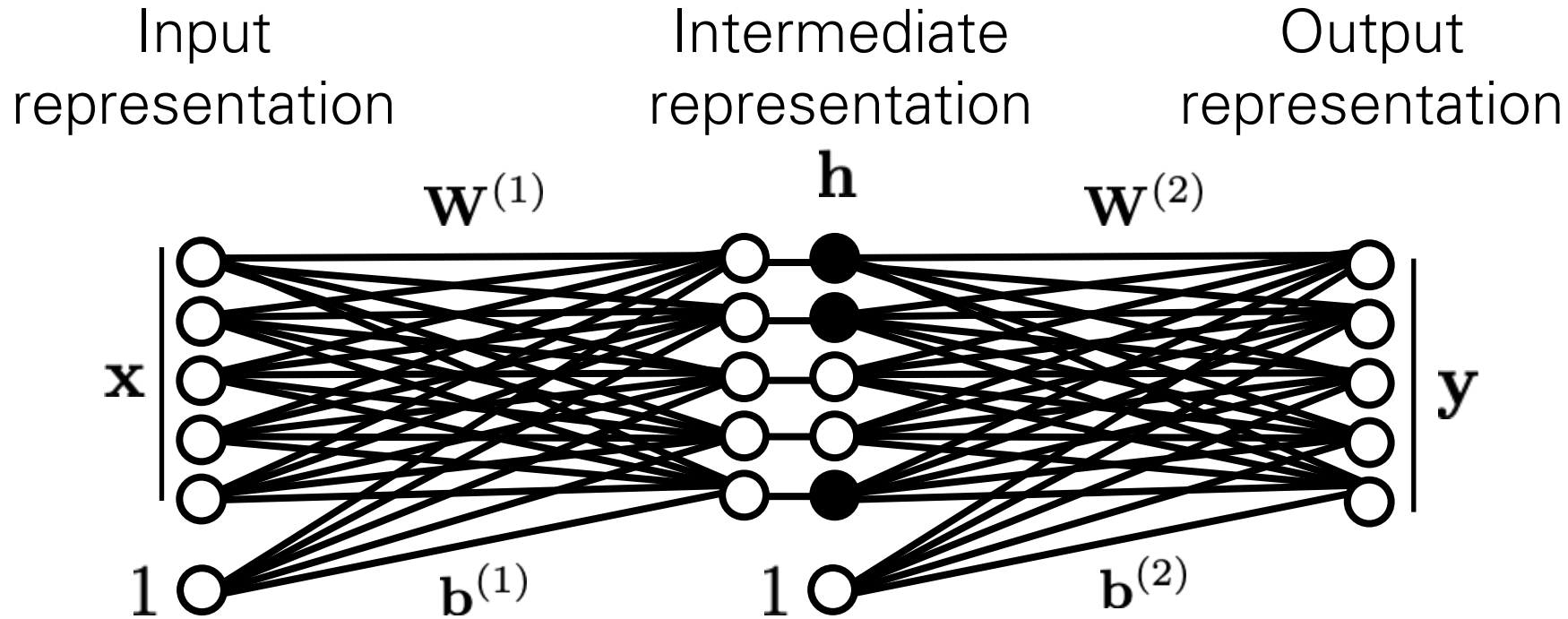
$$\theta = \{\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(L)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(L)}\}$$

Dropout



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

Dropout

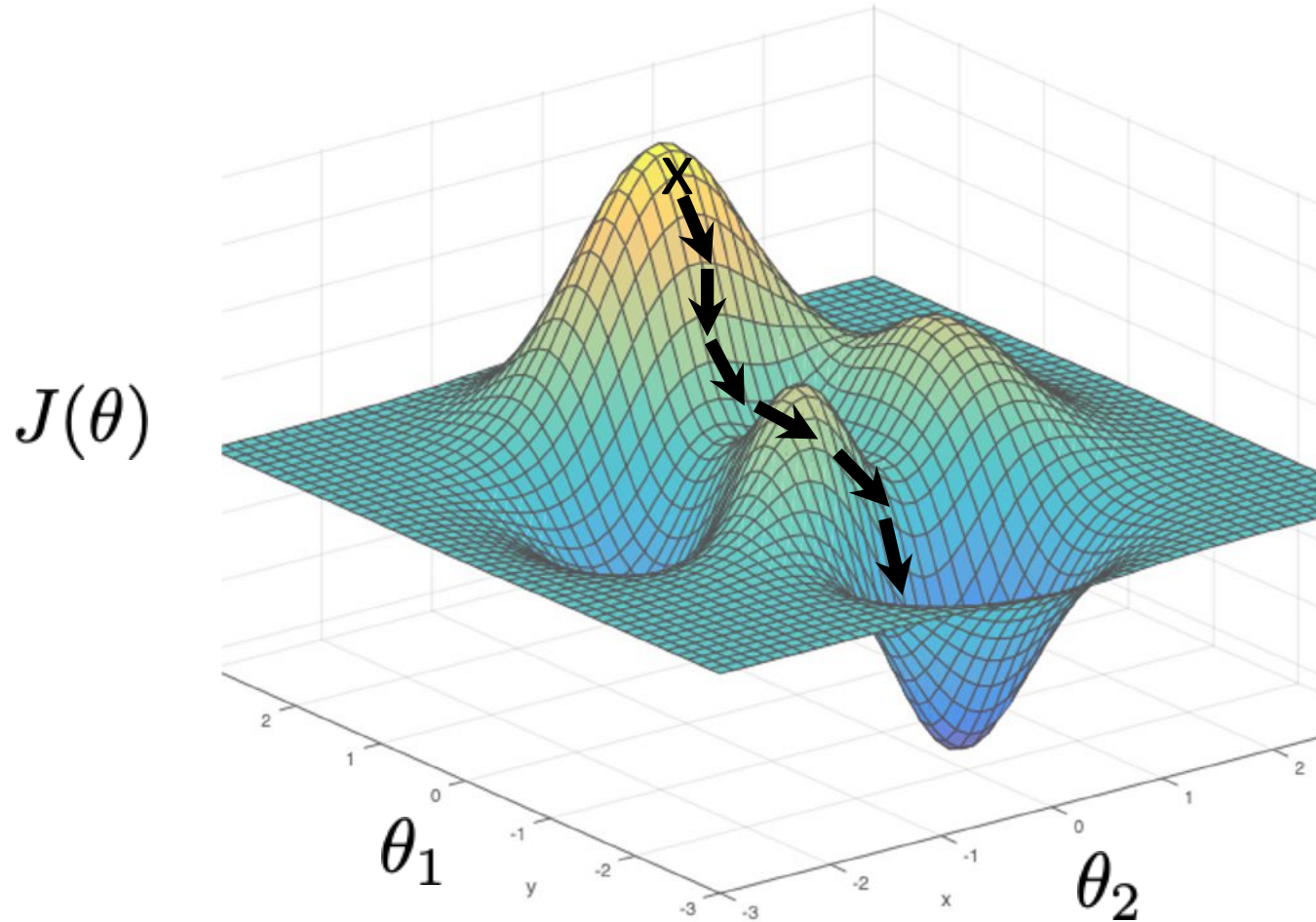


$$\theta = \{\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(L)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(L)}\}$$

Prevents network from relying too much on spurious correlations between different hidden units.

Can be understood as averaging over an exponential **ensemble** of subnetworks. This averaging smooths the function, thereby reducing the effective capacity of the network.

Gradient descent



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

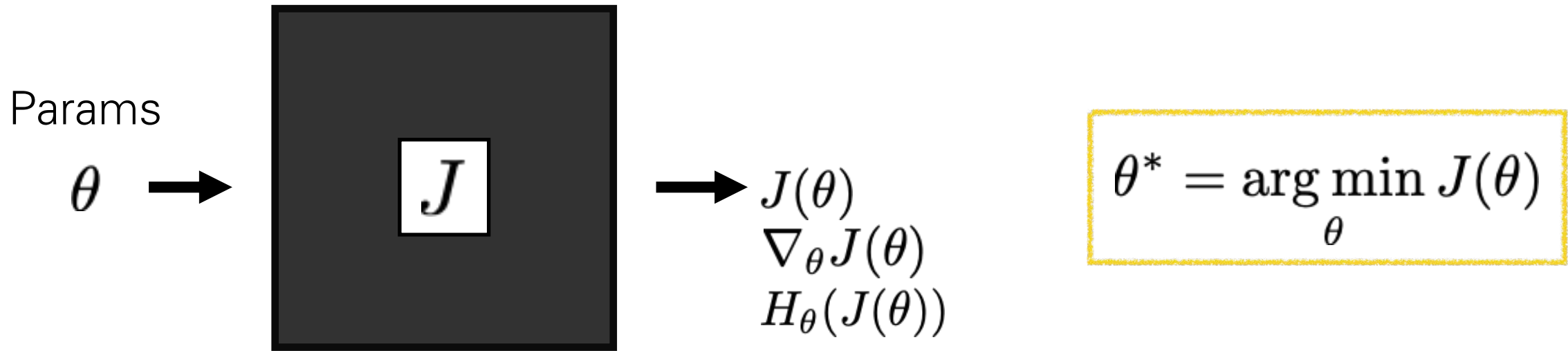
$$\theta^* = \arg \min_{\theta} \underbrace{\sum_{i=1}^N \mathcal{L}(f_{\theta}(\mathbf{x}_i), \mathbf{y}_i)}_{J(\theta)}$$

One iteration of gradient descent:

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

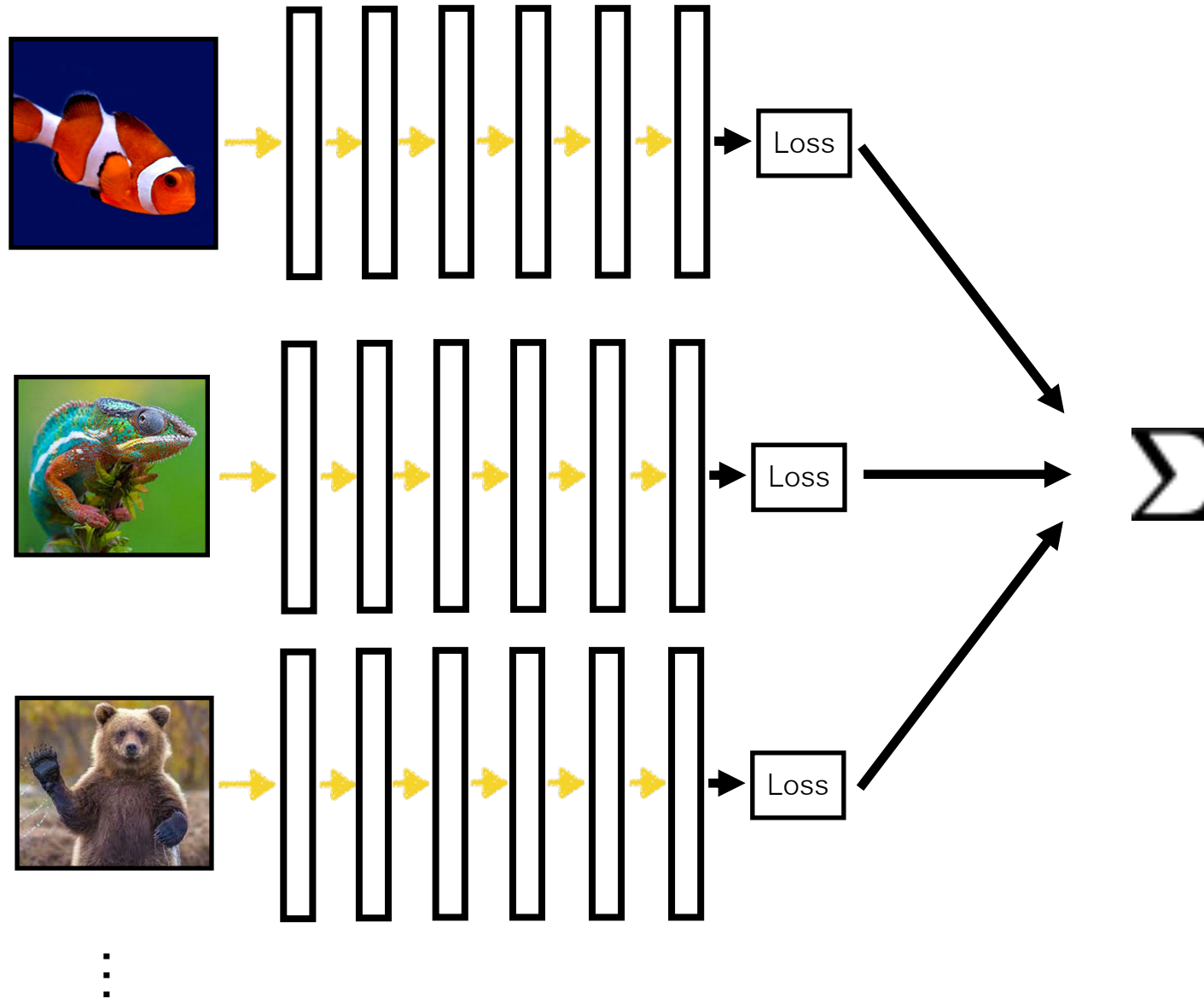
learning rate

Optimization



- What's the knowledge we have about J ?
 - We can evaluate $J(\theta)$
 - We can evaluate $J(\theta)$ and $\nabla_{\theta} J(\theta)$
 - We can evaluate $J(\theta)$, $\nabla_{\theta} J(\theta)$, and $H_{\theta}(J(\theta))$
- Gradient
- Black box optimization
- First order optimization
- 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

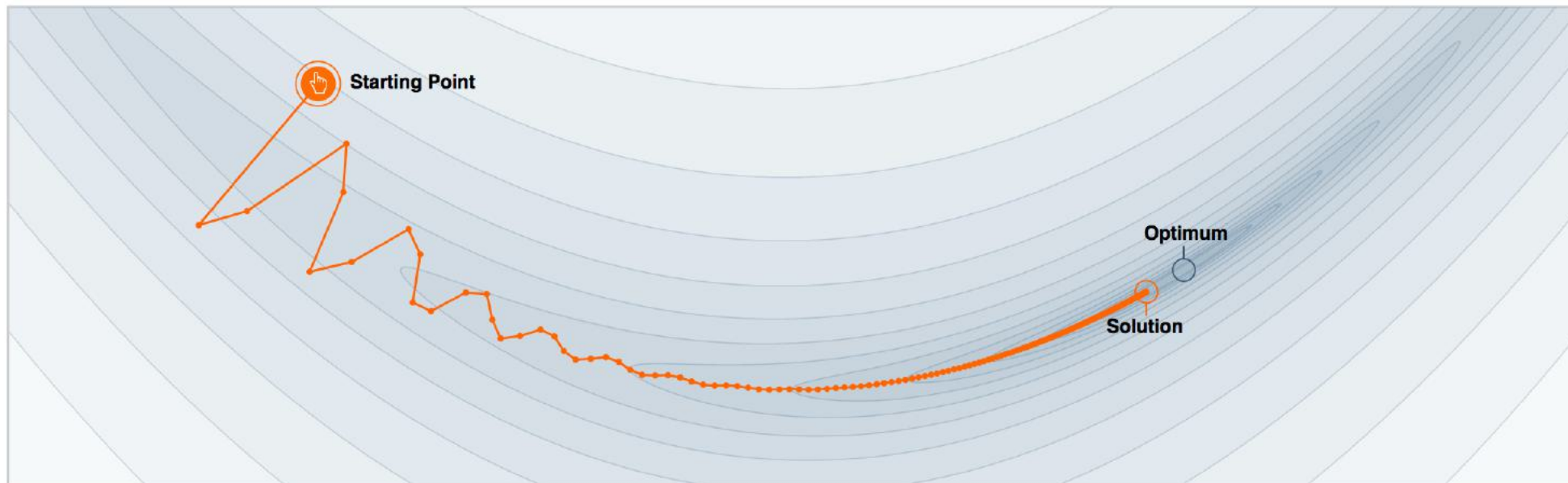
Momentum

- **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?

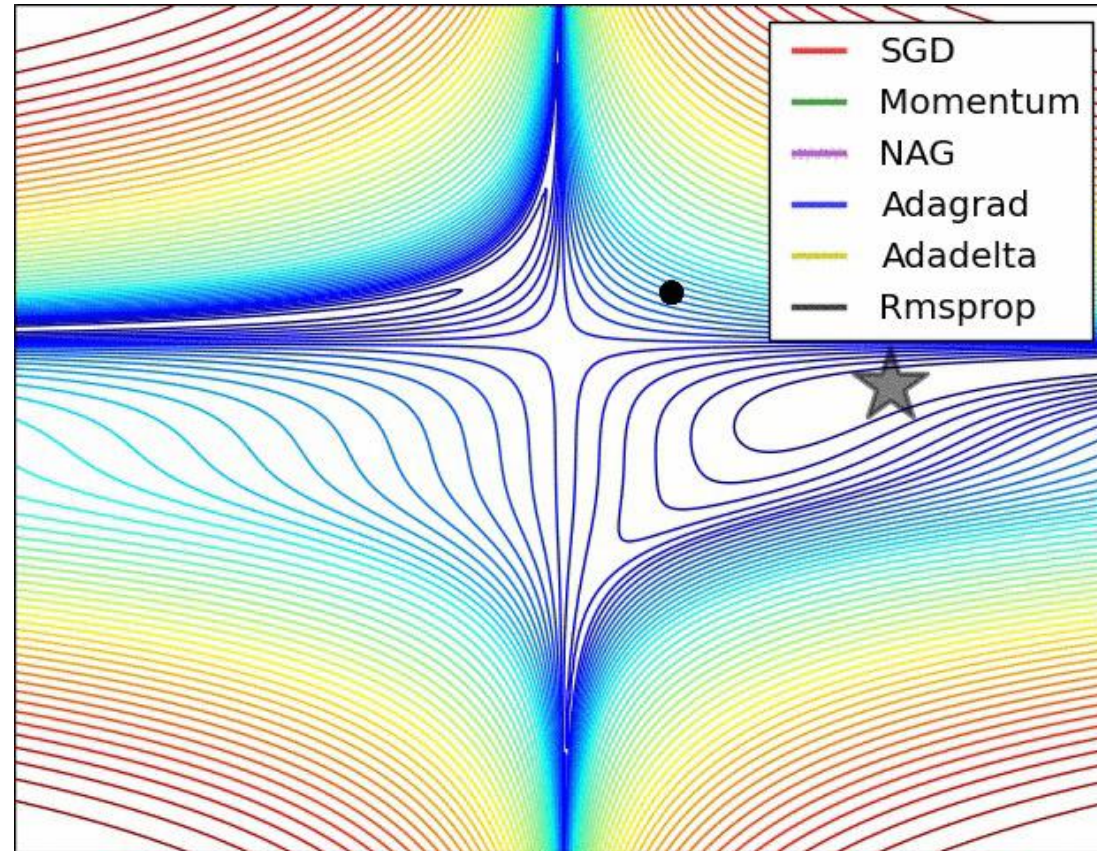
GABRIEL GOH
UC Davis

April. 4
2017

Citation:
Goh, 2017

[<https://distill.pub/2017/momentum/>]

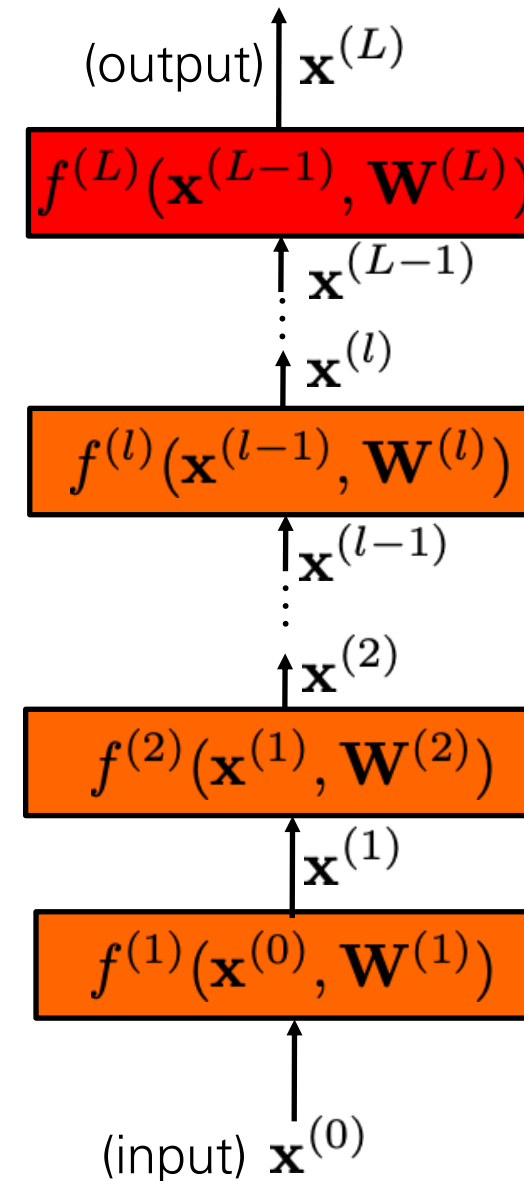
Comparison of gradient descent variants



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

Forward pass

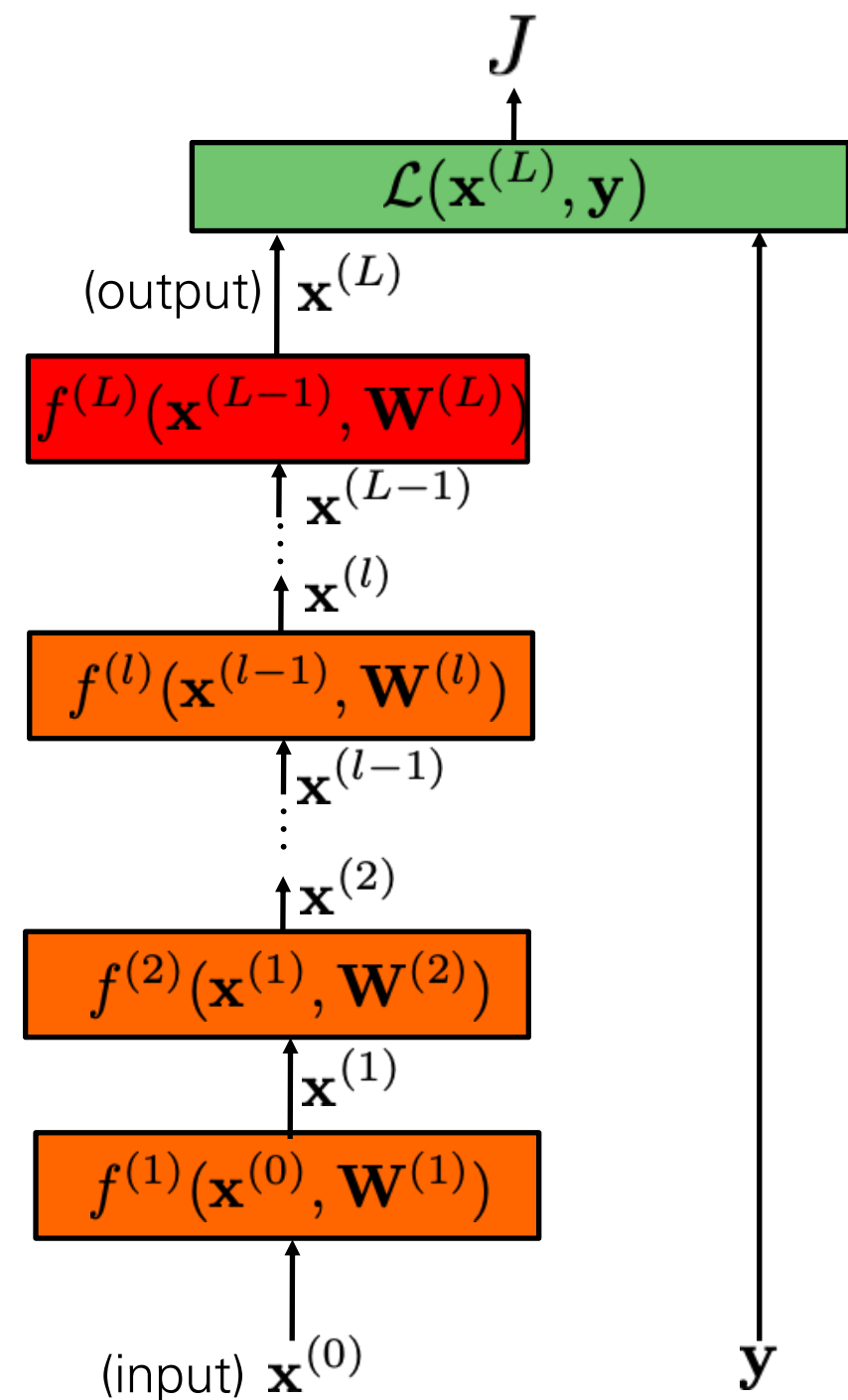
- Consider model with L layers. Layer l has vector of weights $\mathbf{W}^{(l)}$
- **Forward pass:** takes input $\mathbf{x}^{(l-1)}$ and passes it through each layer $f^{(l)}$:
$$\mathbf{x}^{(l)} = f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})$$
- Output of layer l is $\mathbf{x}^{(l)}$.
- Network output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

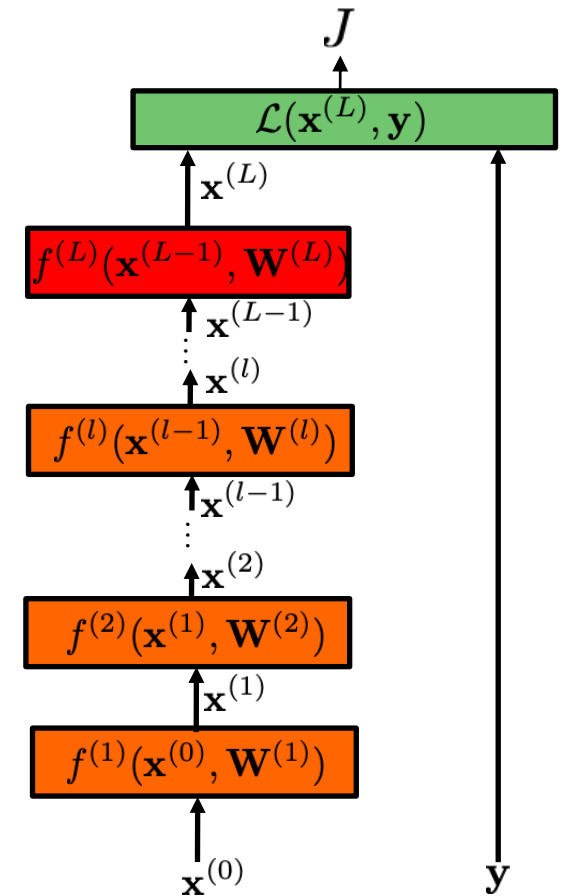
- We need to compute gradients of the cost with respect to model parameters $\mathbf{W}^{(l)}$.
- By design, each layer is differentiable with respect to its parameters and input.

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:

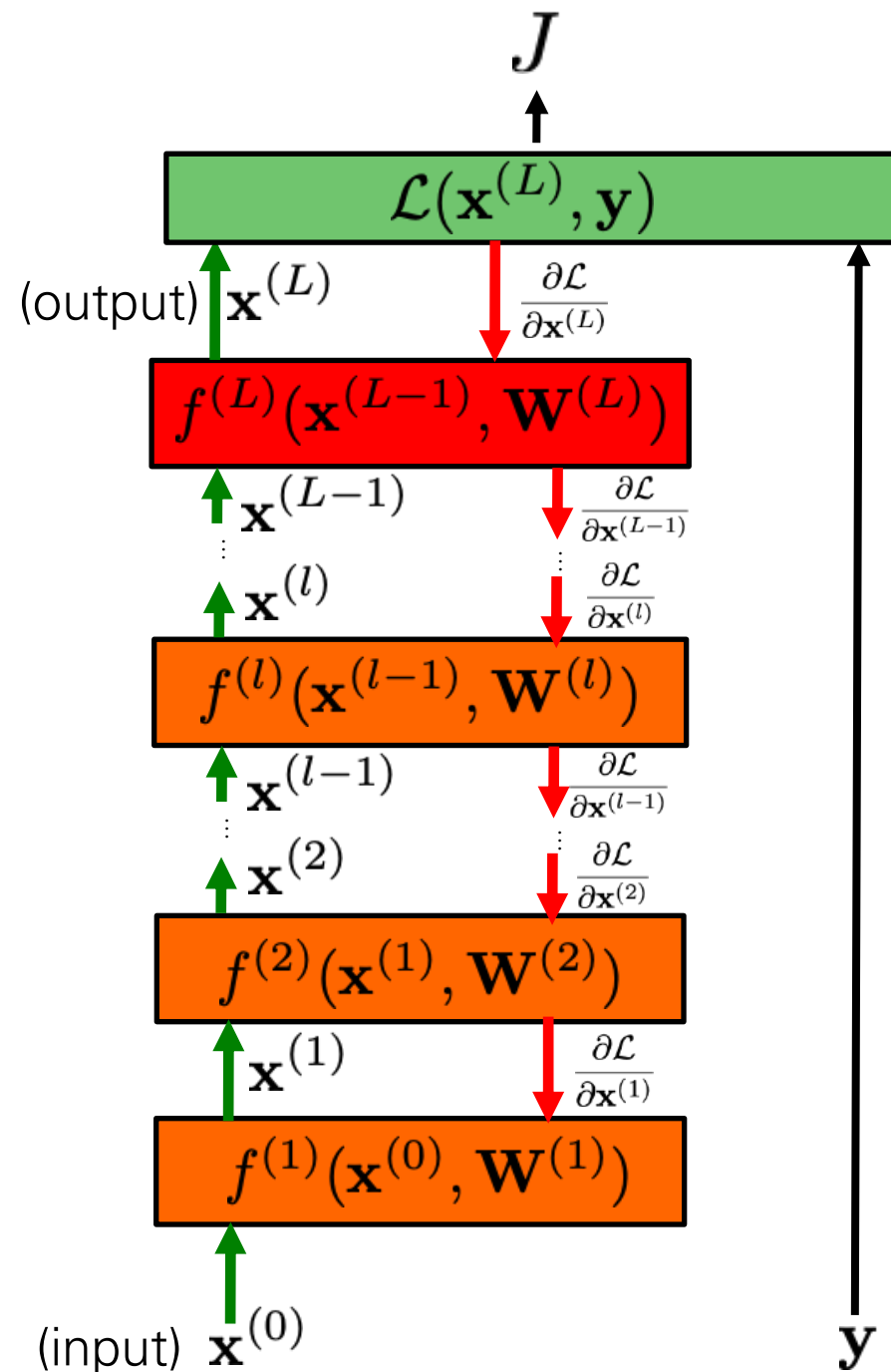
$$\mathbf{x}^{(l)} = f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})$$

- 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).

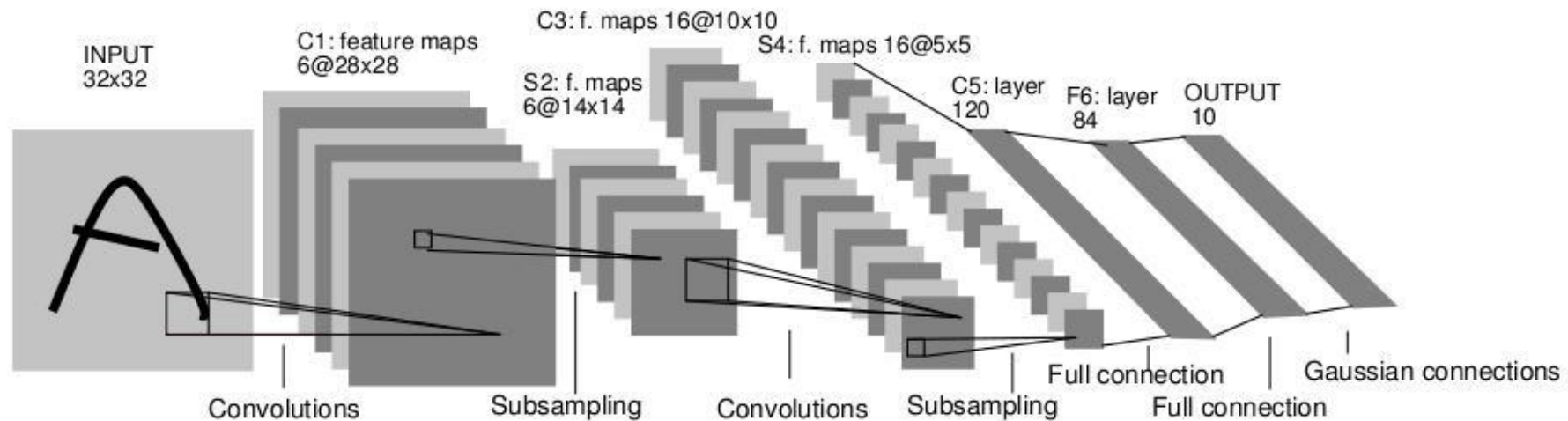


Image classification

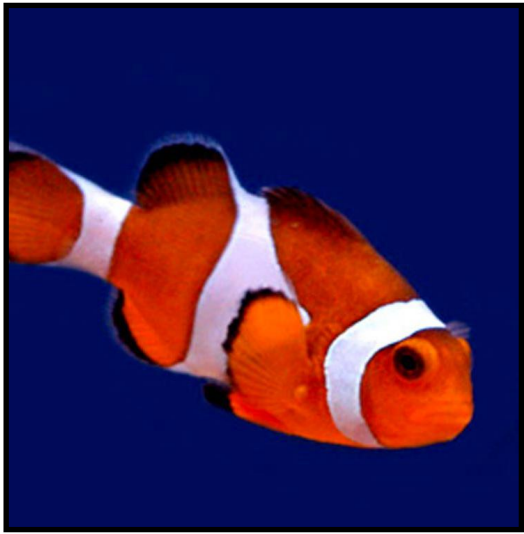


image x

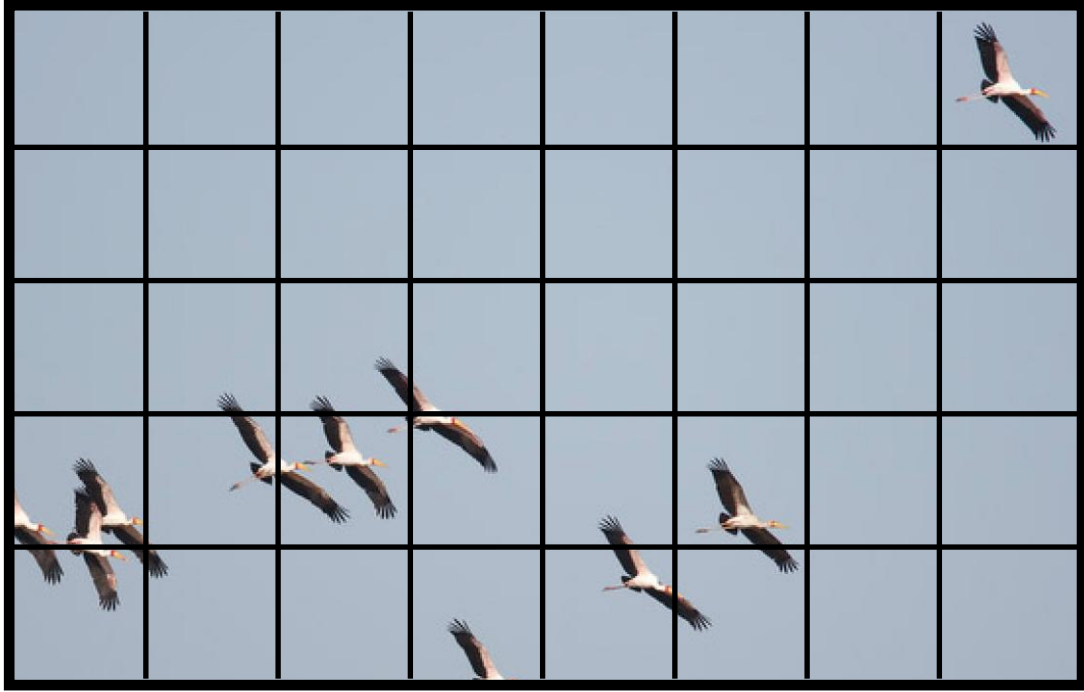


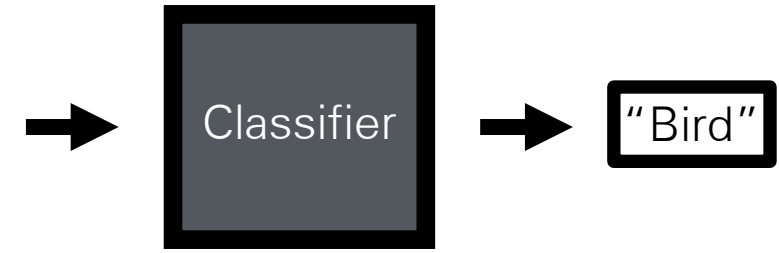
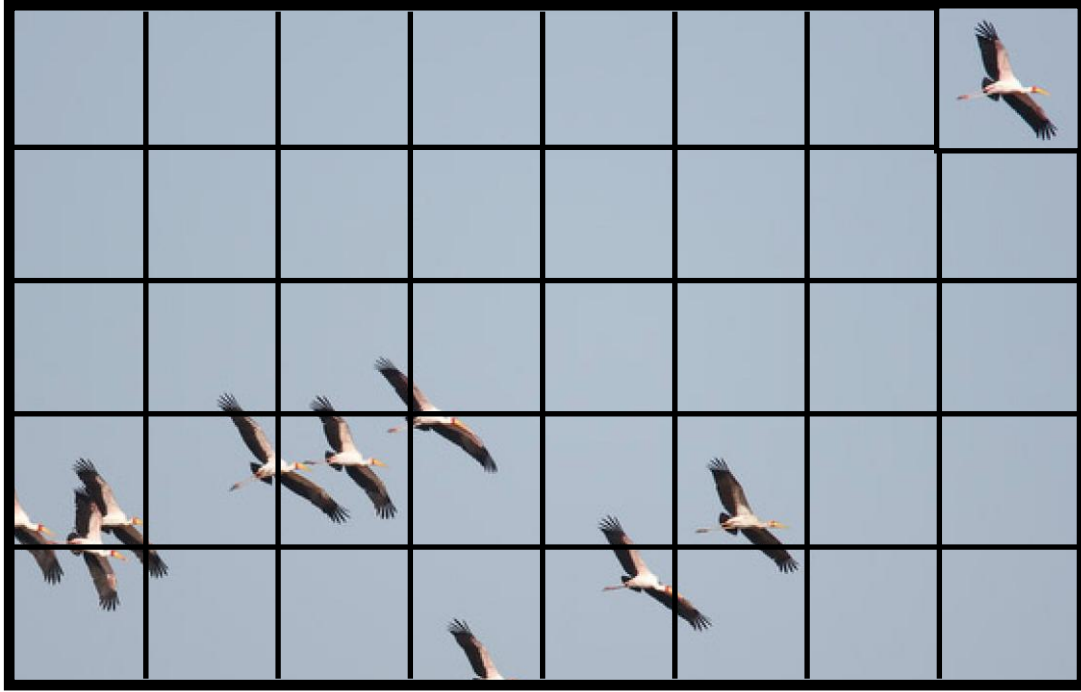
"Fish"

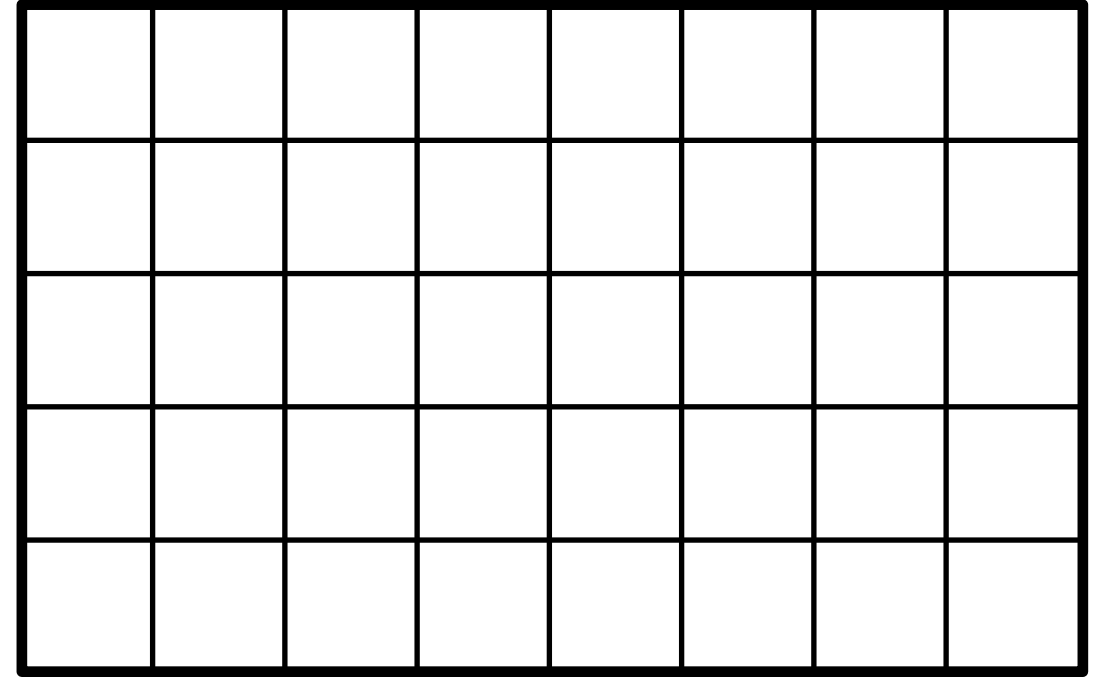
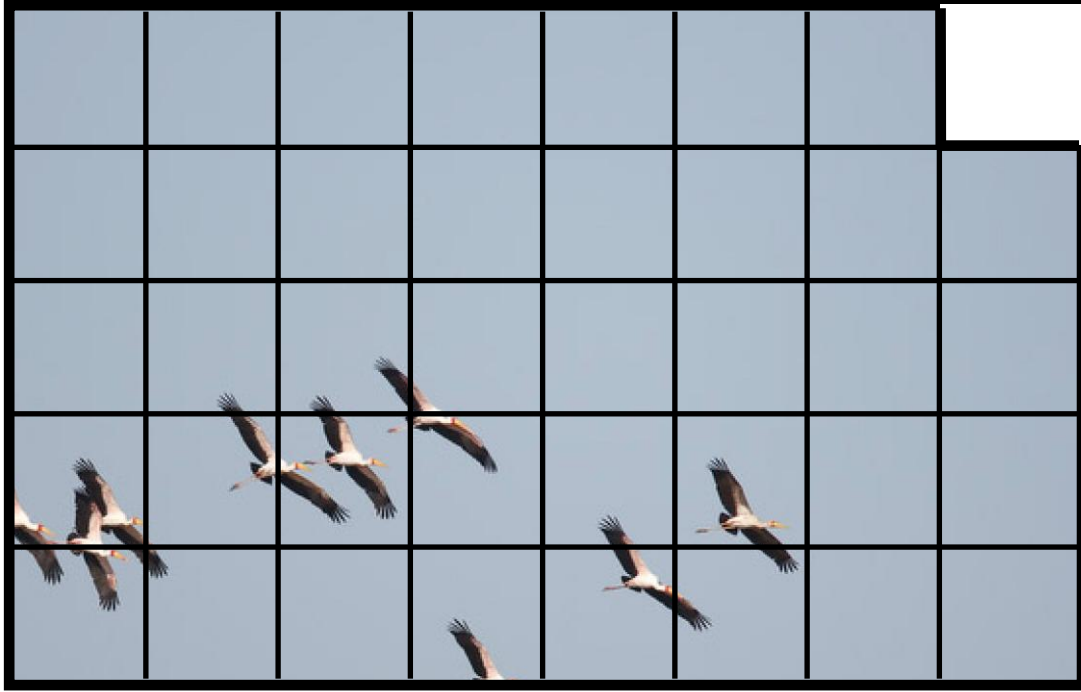
label y

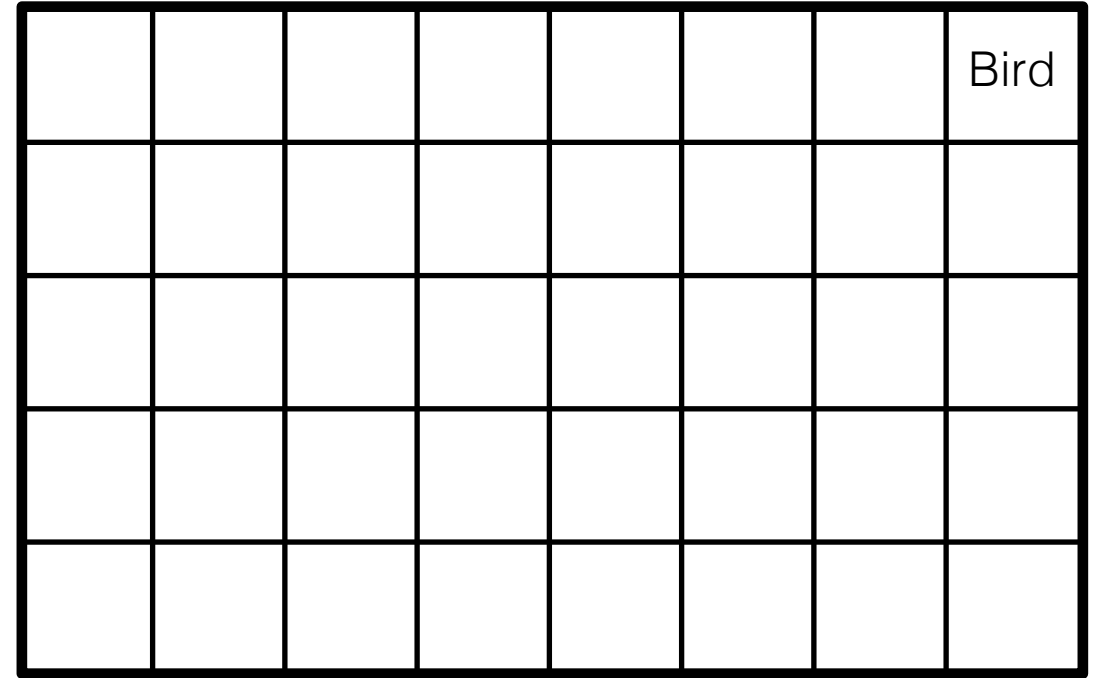
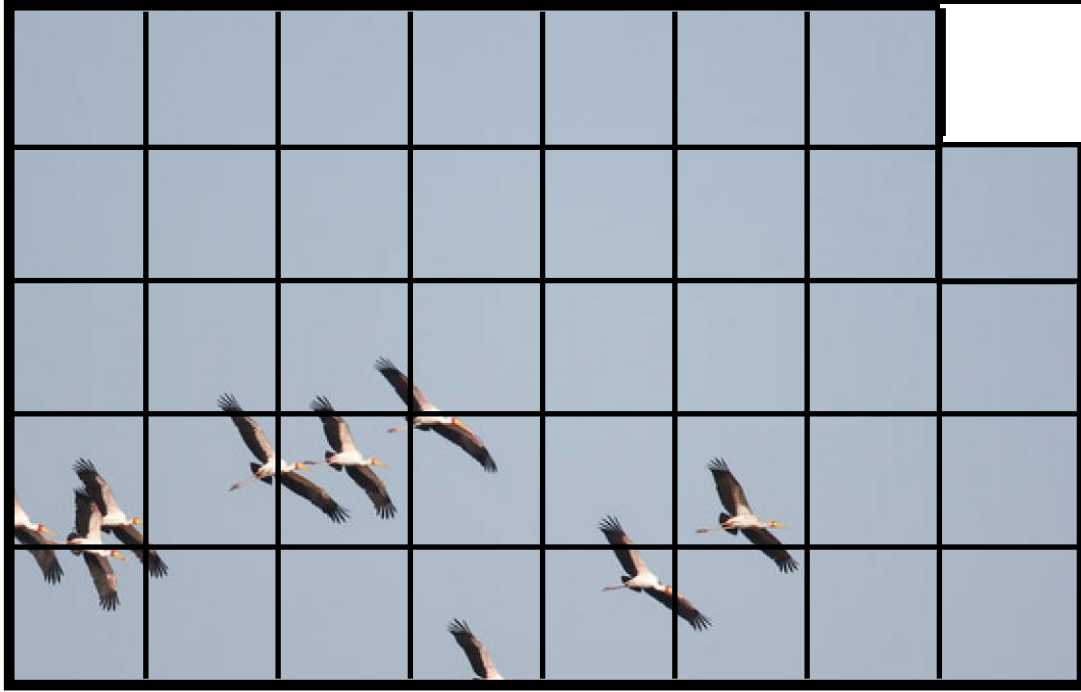


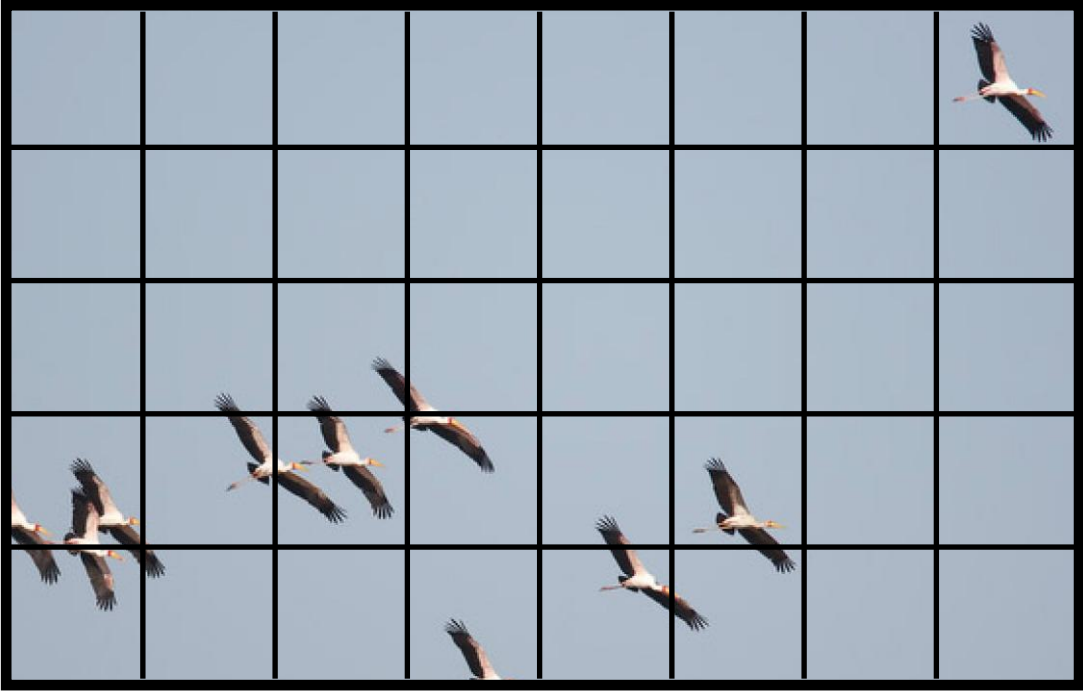
Photo credit: Fredo Durand



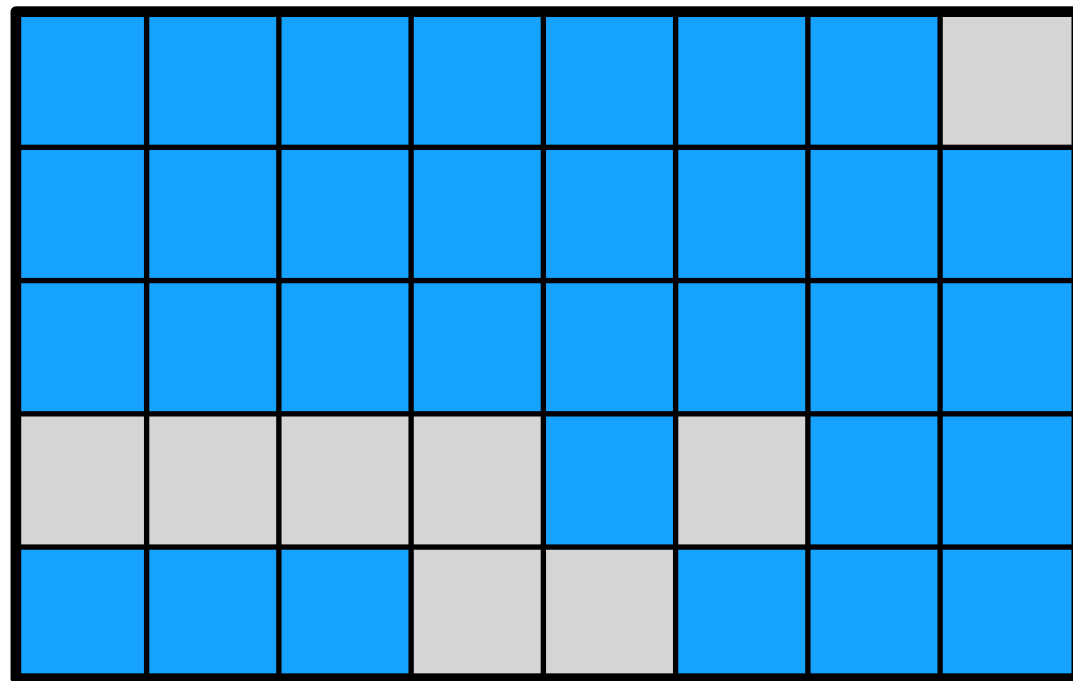
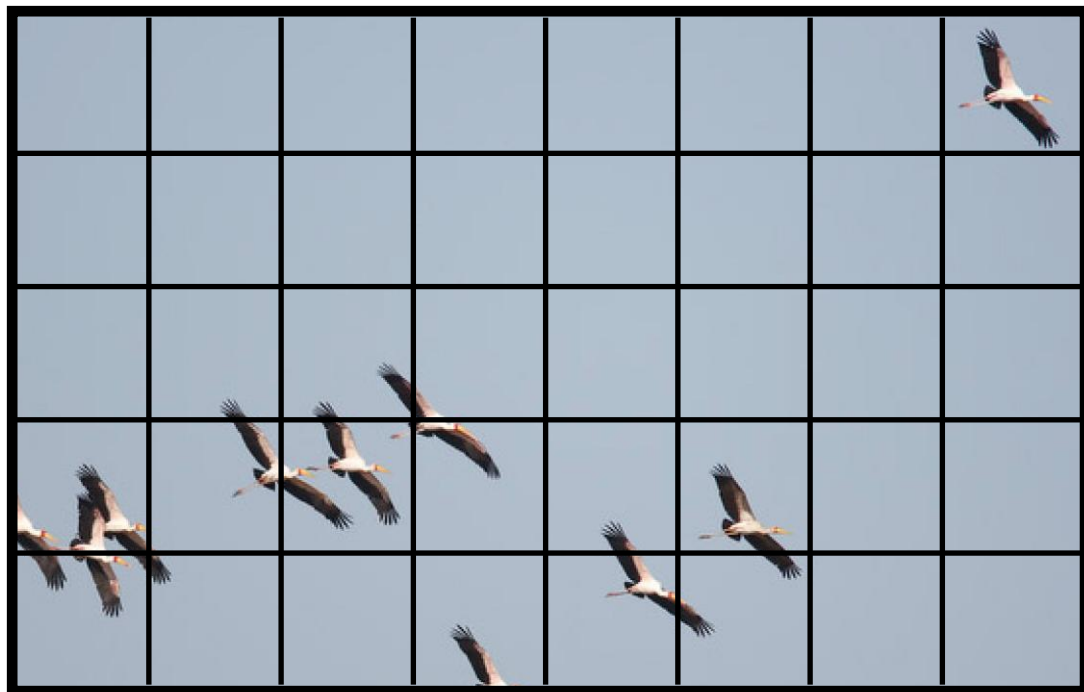


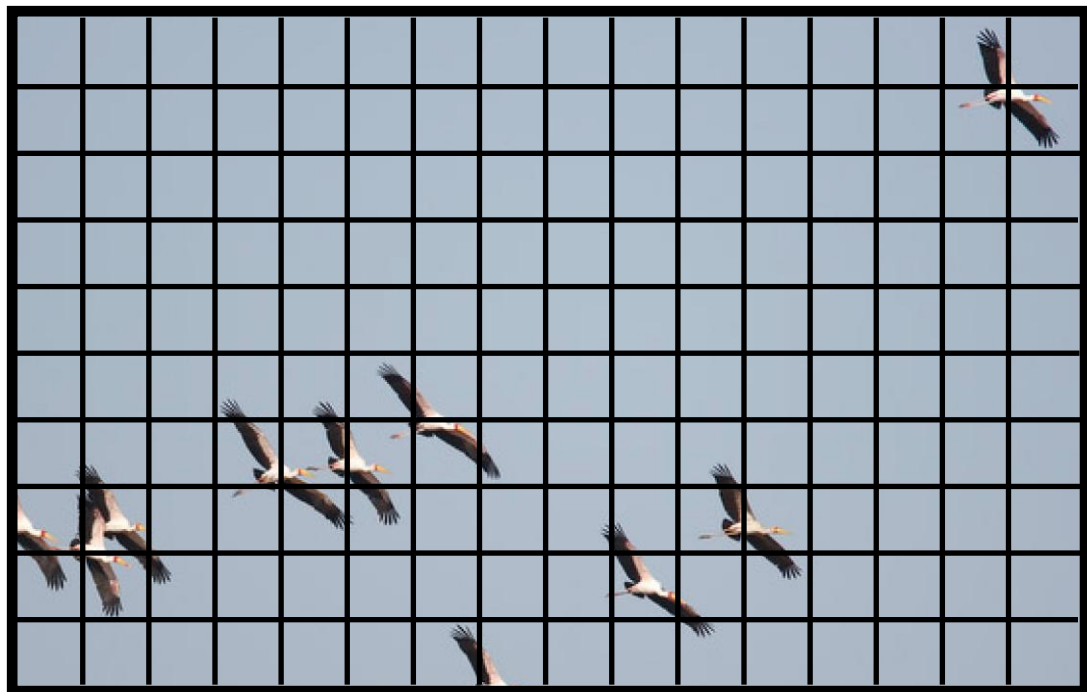


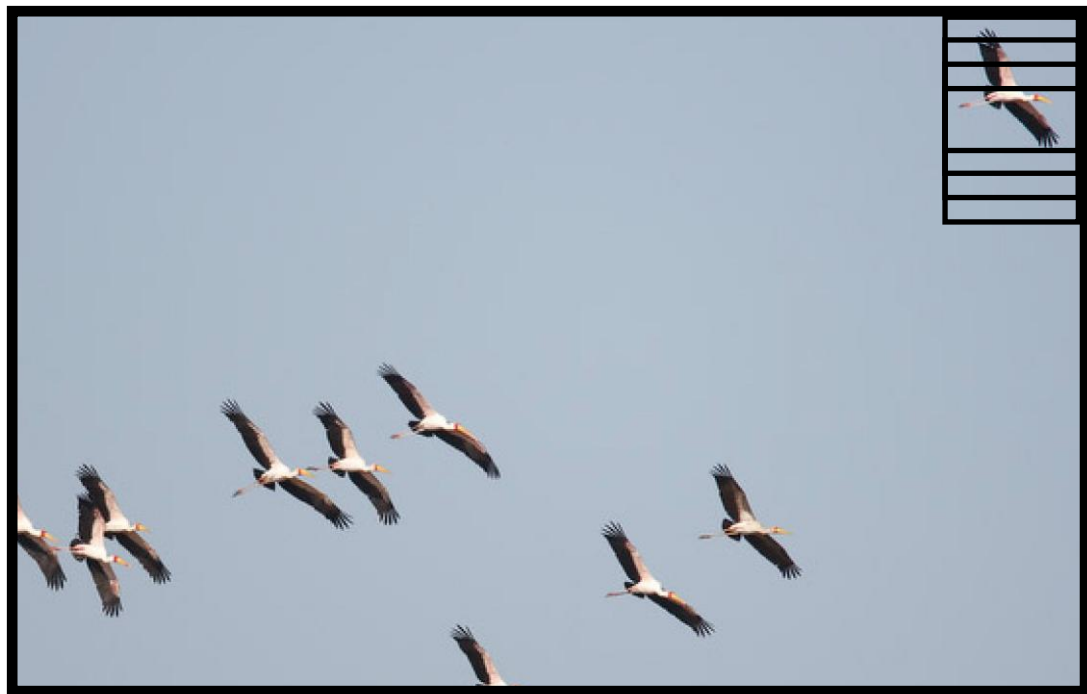




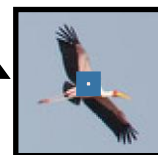
Sky	Sky	Sky	Sky	Sky	Sky	Sky	Bird
Sky	Sky	Sky	Sky	Sky	Sky	Sky	Sky
Sky	Sky	Sky	Sky	Sky	Sky	Sky	Sky
Bird	Bird	Bird	Sky	Bird	Sky	Sky	Sky
Sky	Sky	Sky	Bird	Sky	Sky	Sky	Sky



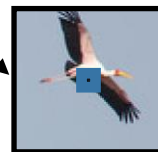




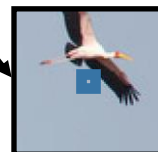
What's the object class of the center pixel?



"Bird"



"Bird"



"Sky"



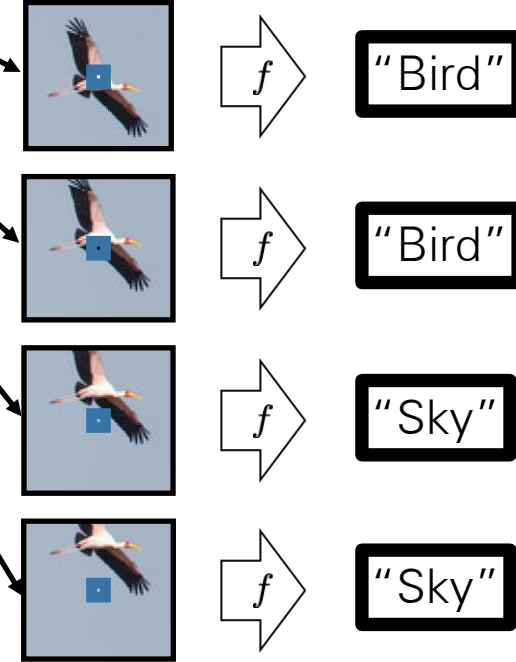
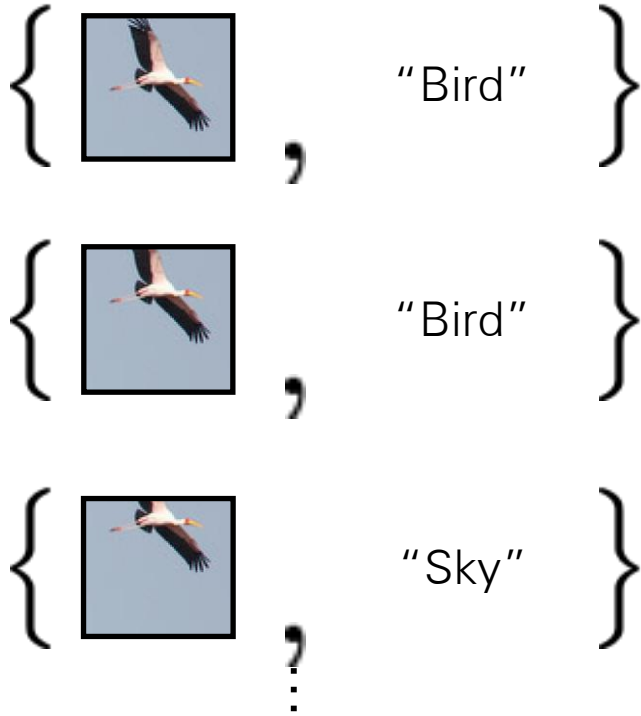
"Sky"

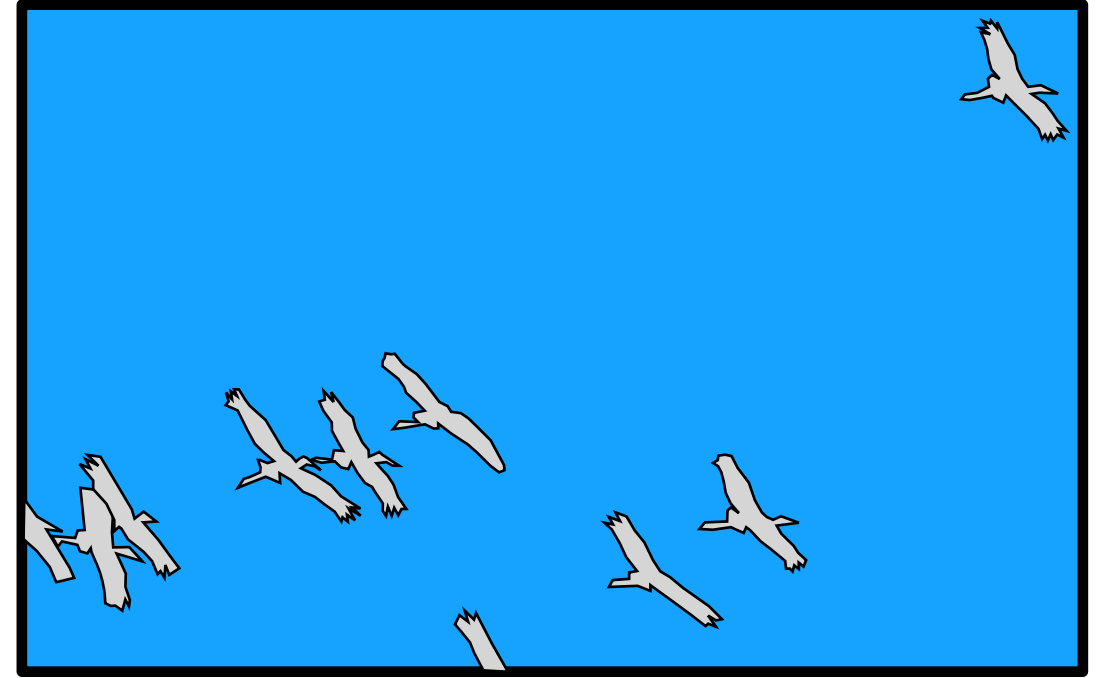
What's the object class of the center pixel?

Training data

x

y



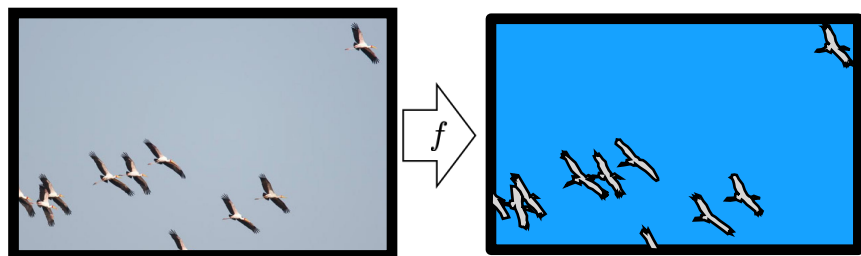
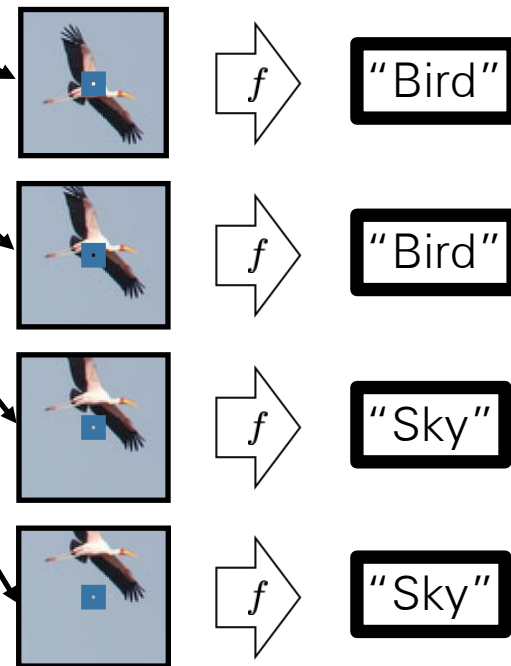


(Colors represent one-hot codes)

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



What's the object class of the center pixel?

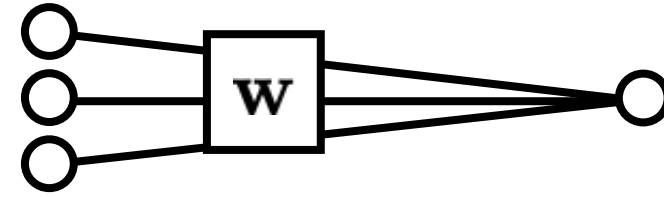


Translation invariance: process each patch in the same way.

An equivariant mapping:

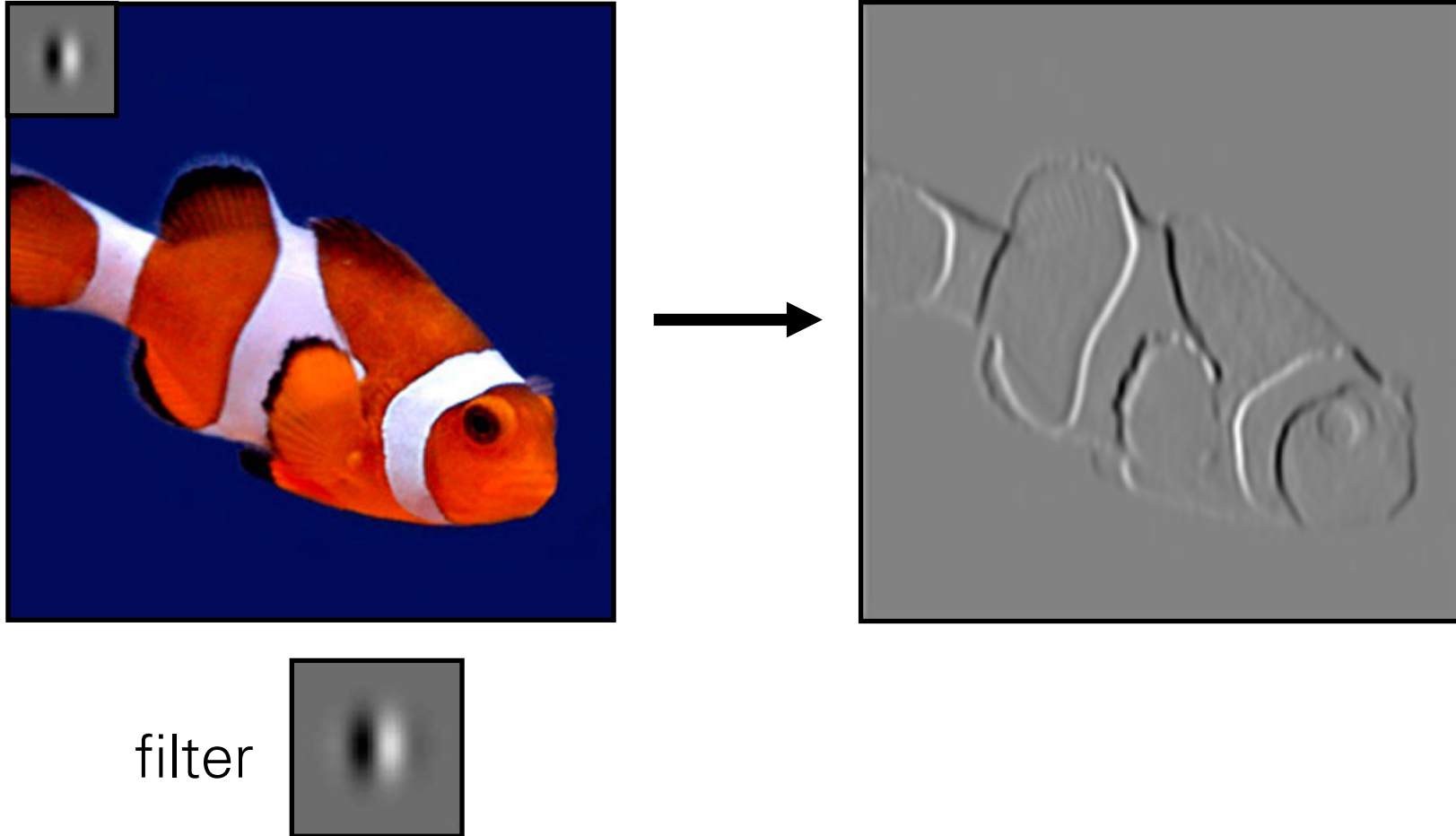
$$f(\text{translate}(x)) = \text{translate}(f(x))$$

W computes a weighted sum of all pixels in the patch



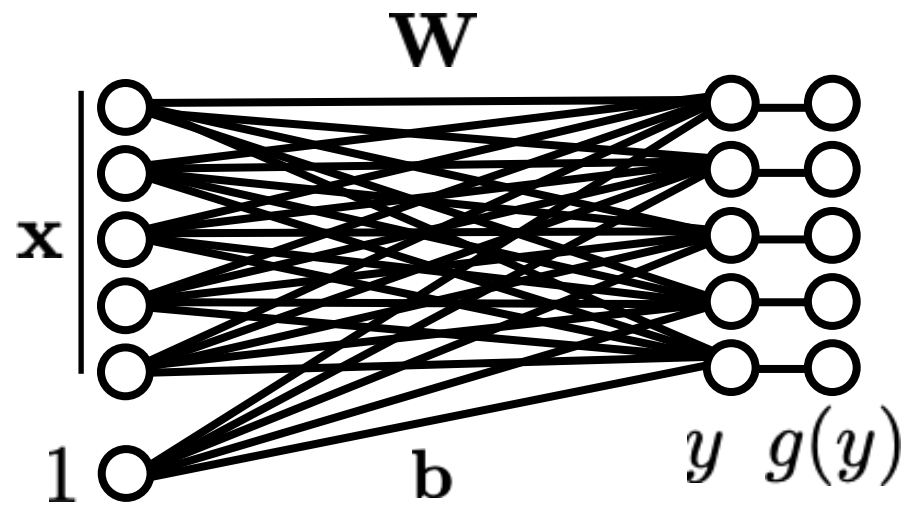
W is a convolutional kernel applied to the full image!

Convolution

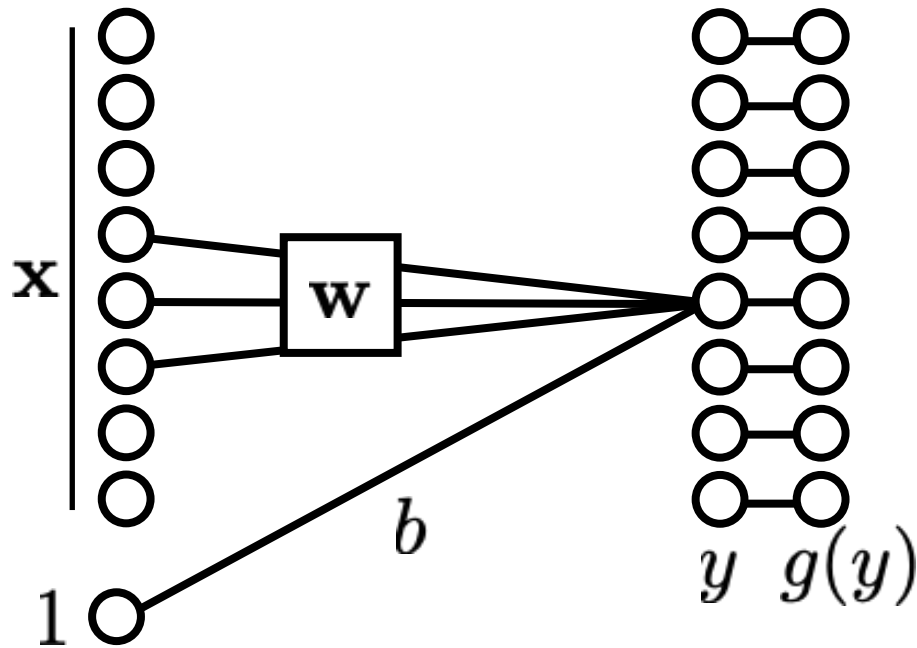


Fully-connected network

Fully-connected (fc) layer



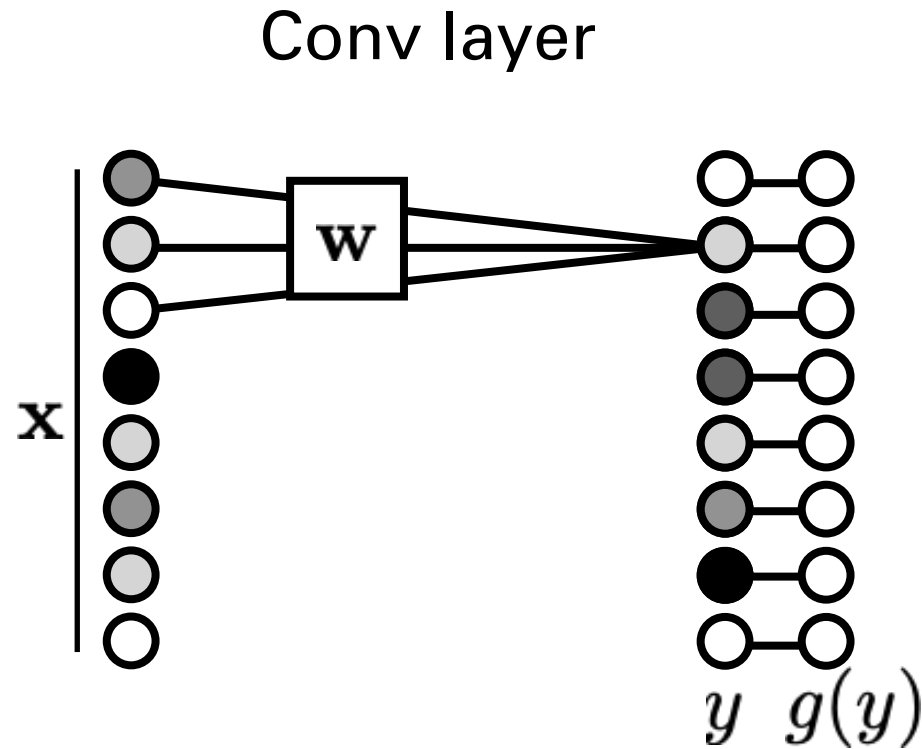
Locally connected network



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.

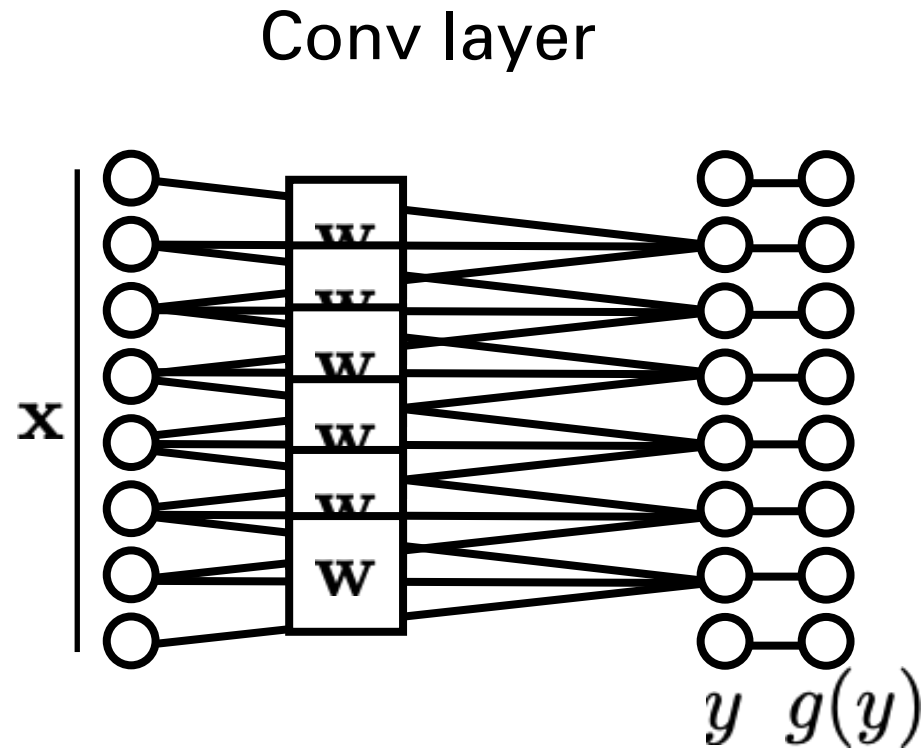
Convolutional neural network



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



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

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

$$\mathbf{x}^{(l+1)} = \begin{matrix} \text{[Toeplitz Matrix]} \end{matrix} * \mathbf{x}^{(l)}$$

The diagram illustrates the equation $\mathbf{x}^{(l+1)} = \text{[Toeplitz Matrix]} * \mathbf{x}^{(l)}$. On the left, a vertical rectangle represents the vector $\mathbf{x}^{(l+1)}$. In the center, an equals sign is followed by a square matrix with a diagonal band of alternating black and white pixels, representing the Toeplitz matrix. To the right of the matrix is an asterisk $*$, followed by another vertical rectangle representing the vector $\mathbf{x}^{(l)}$.

e.g., pixel image

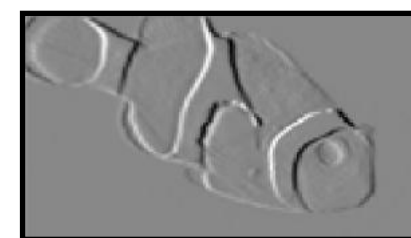
- Constrained linear layer (infinitely strong regularization)
- Fewer parameters \rightarrow easier to learn, less overfitting

$$\mathbf{x}^{(l+1)} = \begin{bmatrix} \text{matrix} \end{bmatrix} * \mathbf{x}^{(l)}$$

The diagram illustrates a vector-matrix multiplication. On the left, a vertical white rectangle with a black border represents the vector $\mathbf{x}^{(l+1)}$. In the center, a square matrix is shown with a dark gray background and a diagonal line of alternating black and white squares, representing a sparse matrix. To the right of the matrix is a multiplication symbol $*$. On the far right, another vertical white rectangle with a black border represents the vector $\mathbf{x}^{(l)}$. An equals sign $=$ is positioned between the vector $\mathbf{x}^{(l+1)}$ and the matrix.

$$\mathbf{x}^{(l+1)} = \begin{bmatrix} \text{diagonal matrix} \end{bmatrix} * \mathbf{x}^{(l)}$$

The diagram illustrates a vector-matrix multiplication. On the left, a vertical vector is labeled $\mathbf{x}^{(l+1)}$. In the center, a square matrix is shown with a dark gray background and a light gray diagonal line of pixels, representing a diagonal matrix. To the right of the matrix is an asterisk $*$, indicating multiplication. On the far right, another vertical vector is labeled $\mathbf{x}^{(l)}$. An equals sign $=$ is positioned between the vector and the matrix.

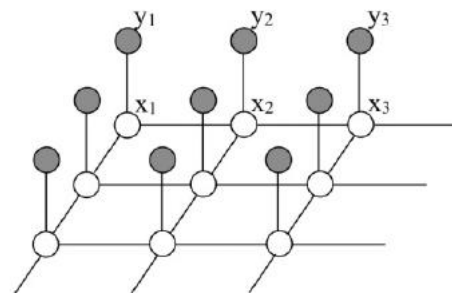


Conv layers can be applied to arbitrarily-sized inputs

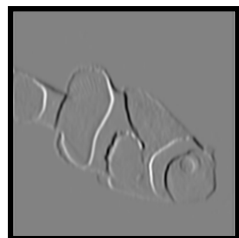
Five views on convolutional layers

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

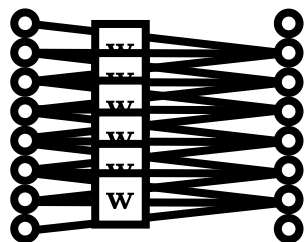
2. Patch processing (Markov assumption)



3. Image filter

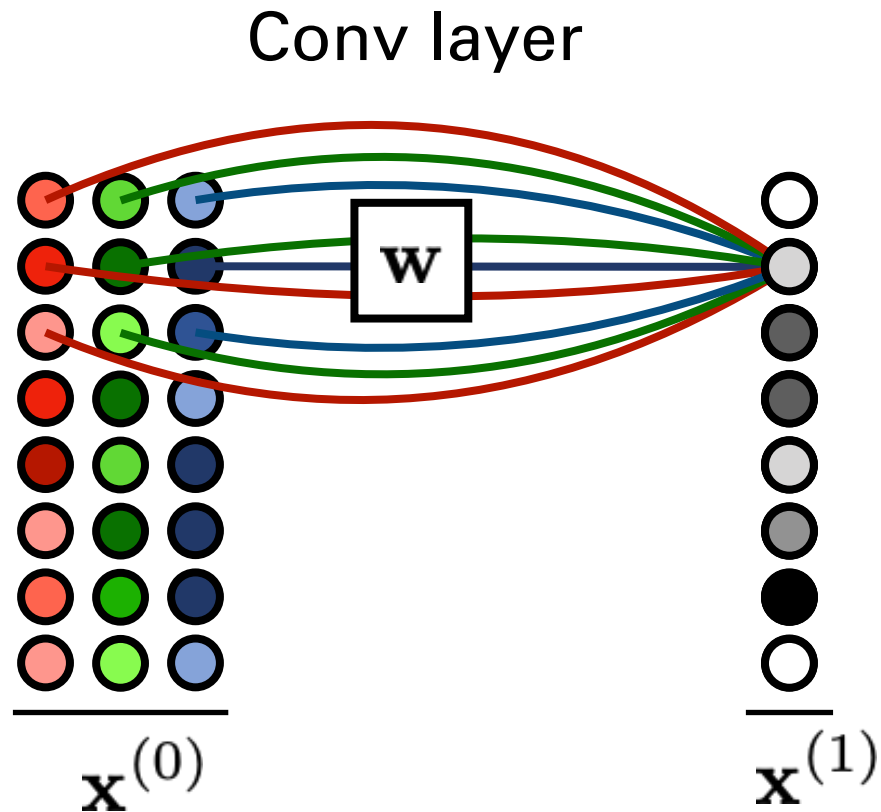


4. Parameter sharing



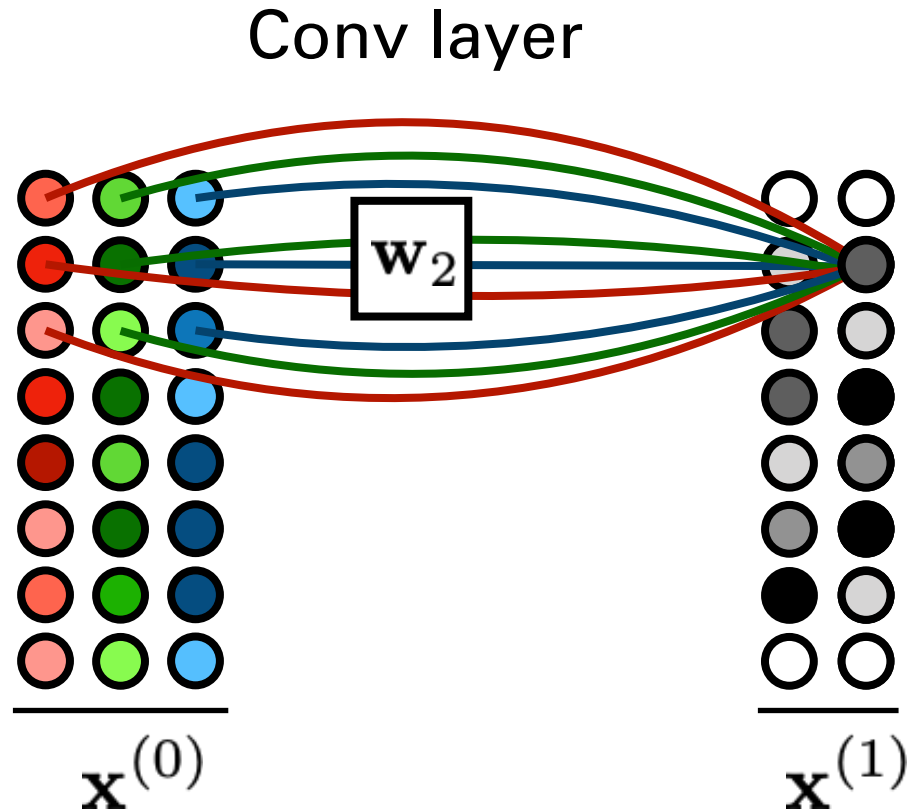
5. A way to process variable-sized tensors

Multiple channels



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

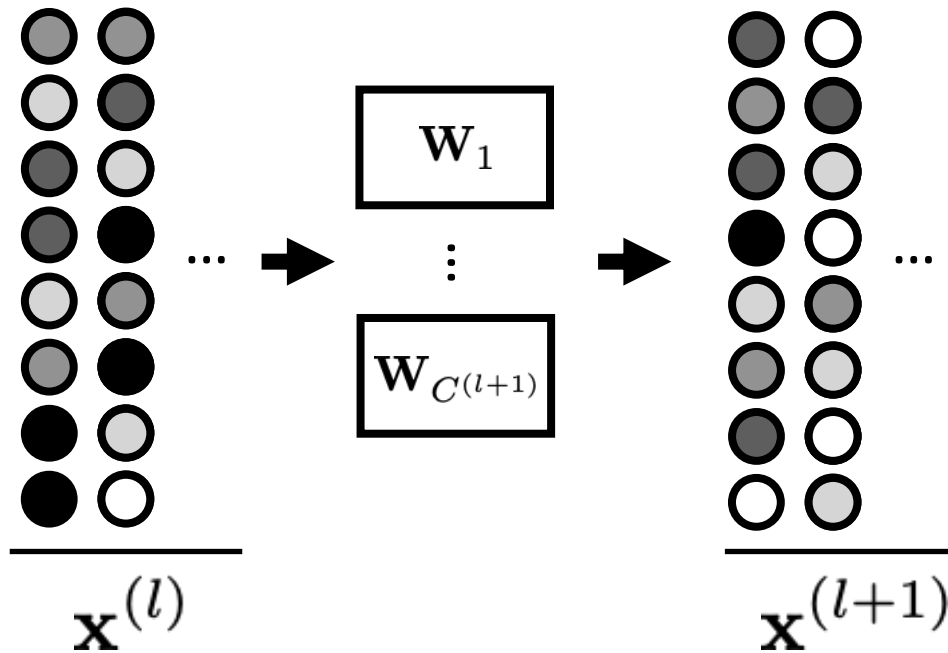
Multiple channels



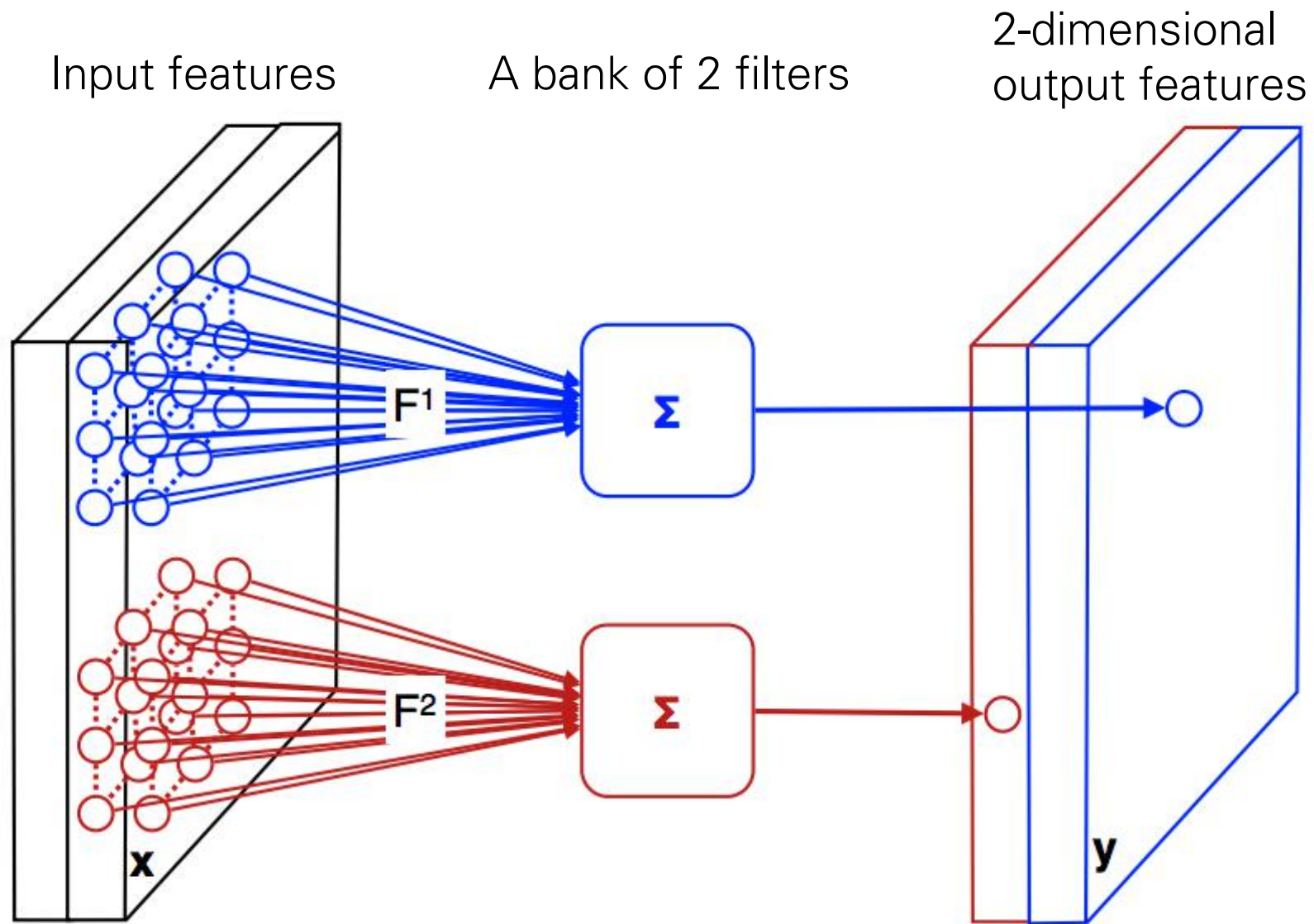
$$\mathbb{R}^{N \times C^{(0)}} \rightarrow \mathbb{R}^{N \times C^{(1)}}$$

Multiple channels

Conv layer



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

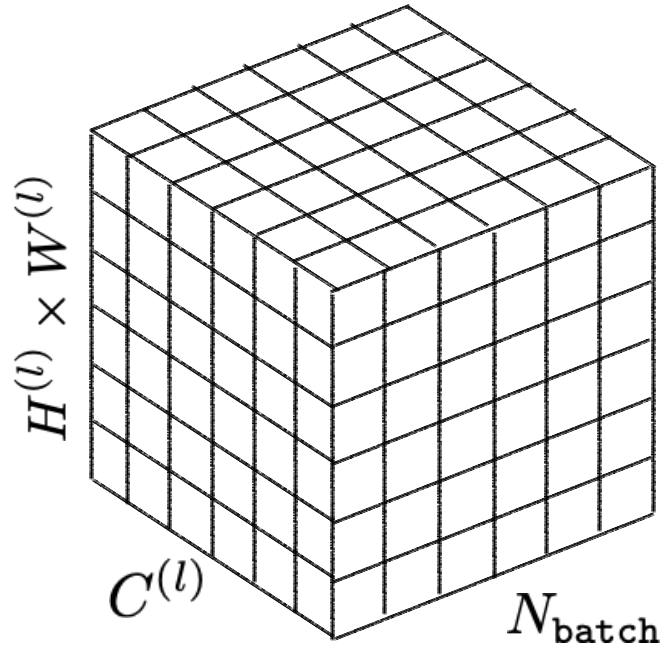


$$\mathbb{R}^{H \times W \times C^{(l)}} \rightarrow \mathbb{R}^{H \times W \times C^{(l+1)}}$$

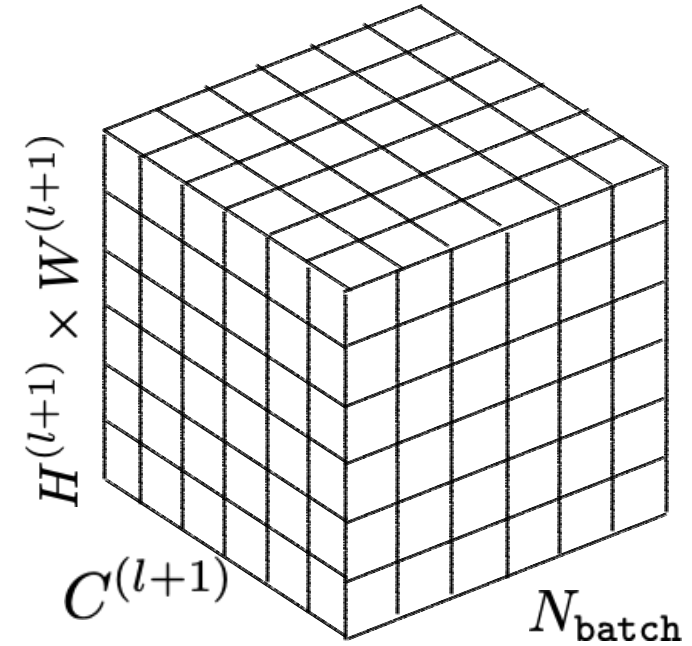
[Figure from Andrea Vedaldi]

"Tensor flow"

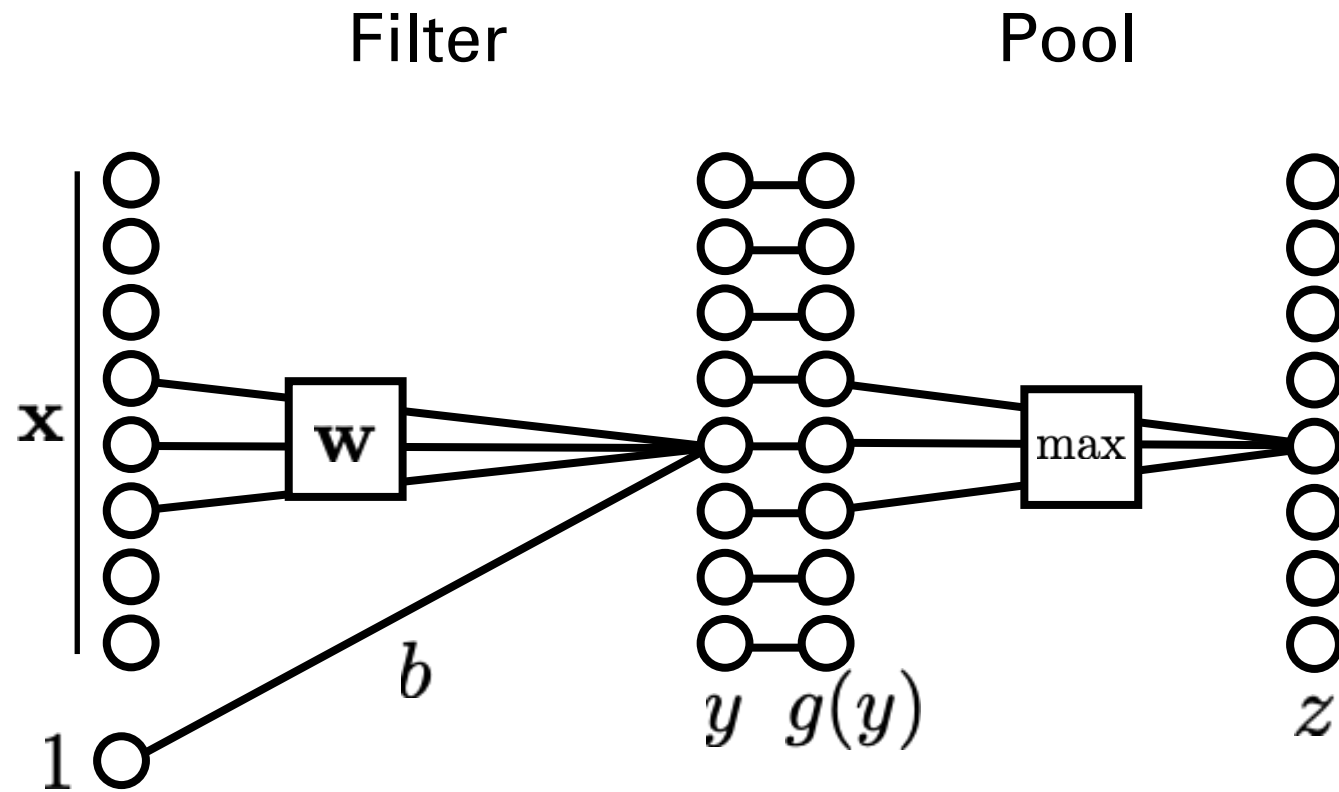
$$\mathbf{x}^{(l)} \in \mathbb{R}^{N_{\text{batch}} \times H^{(l)} \times W^{(l)} \times C^{(l)}}$$



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



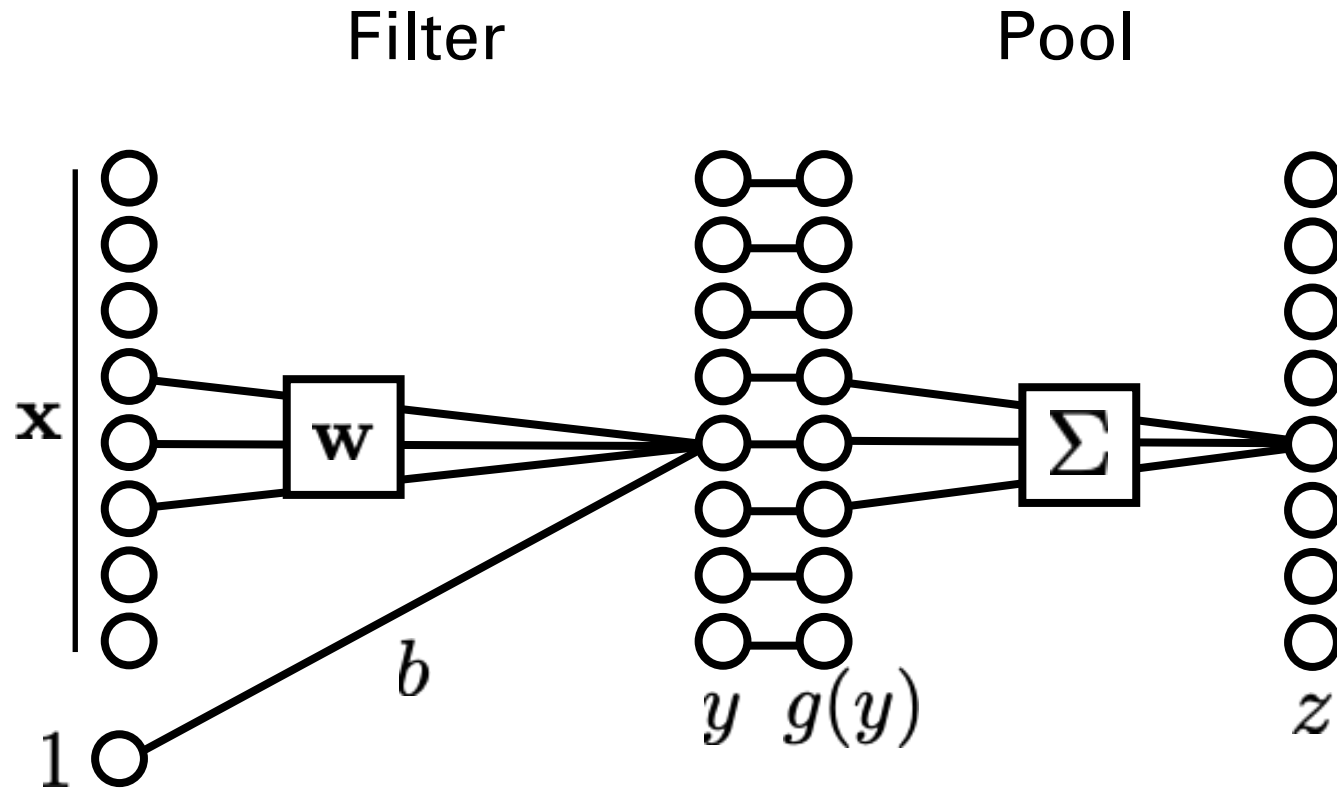
Pooling



Max pooling

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

Pooling



Max pooling

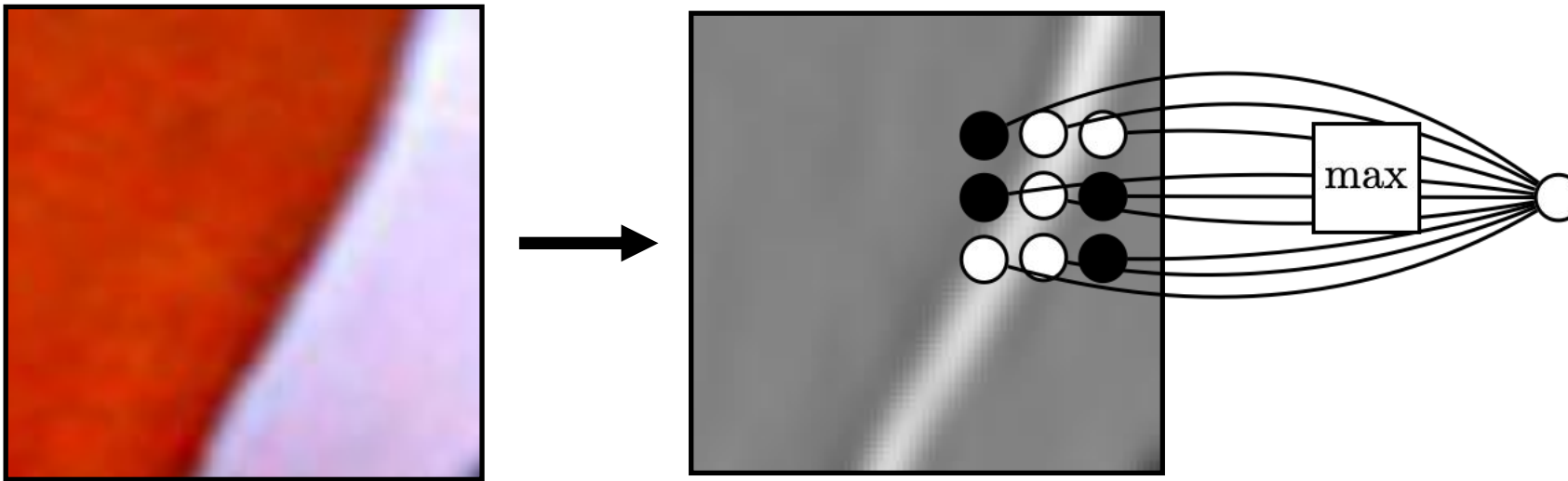
$$z_k = \max_{j \in \mathcal{N}(j)} g(y_j)$$

Mean pooling

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

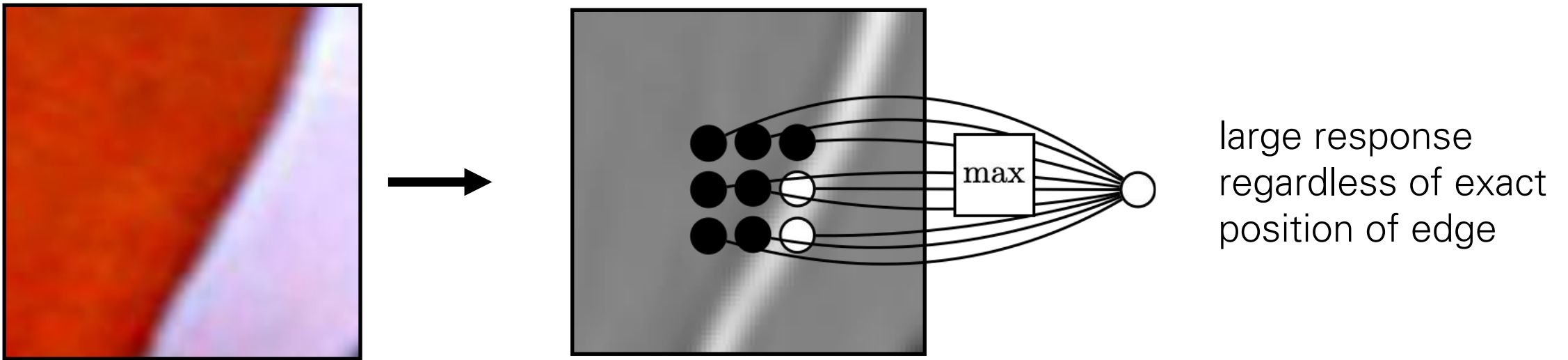
Pooling – Why?

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



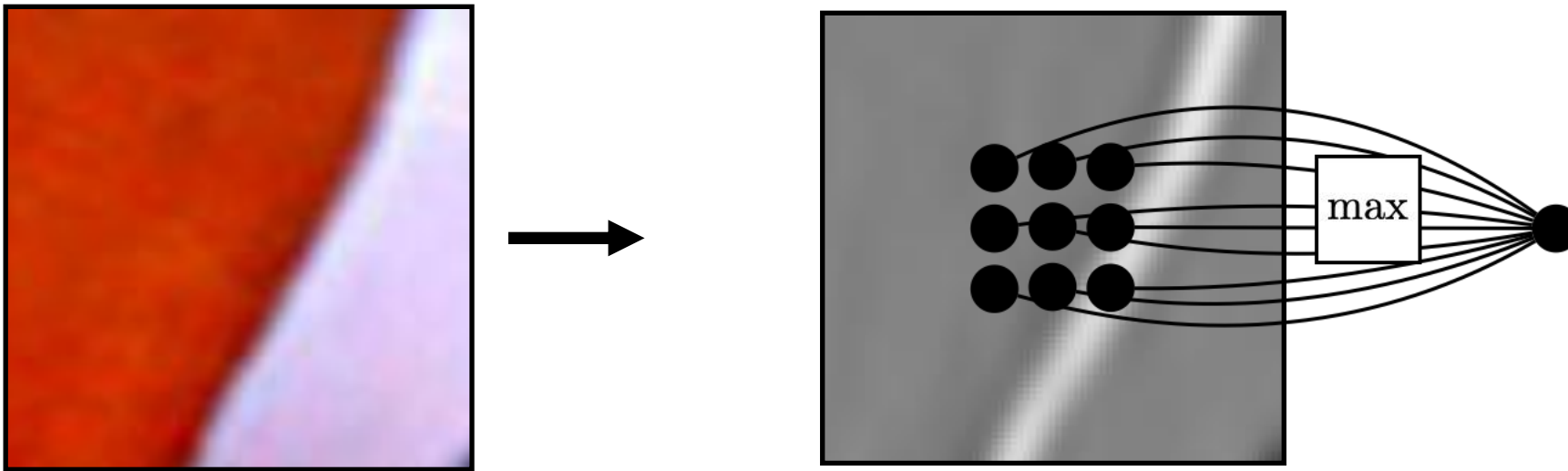
Pooling – Why?

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

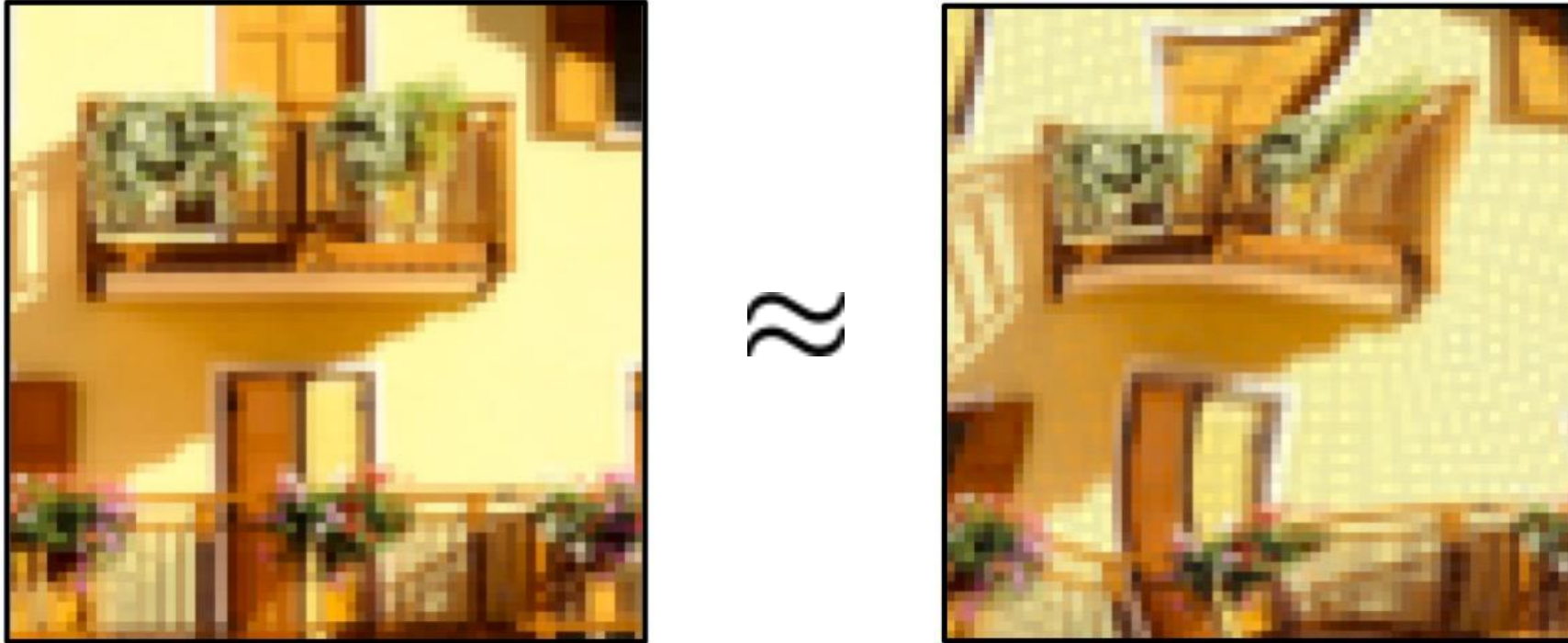


Pooling – Why?

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



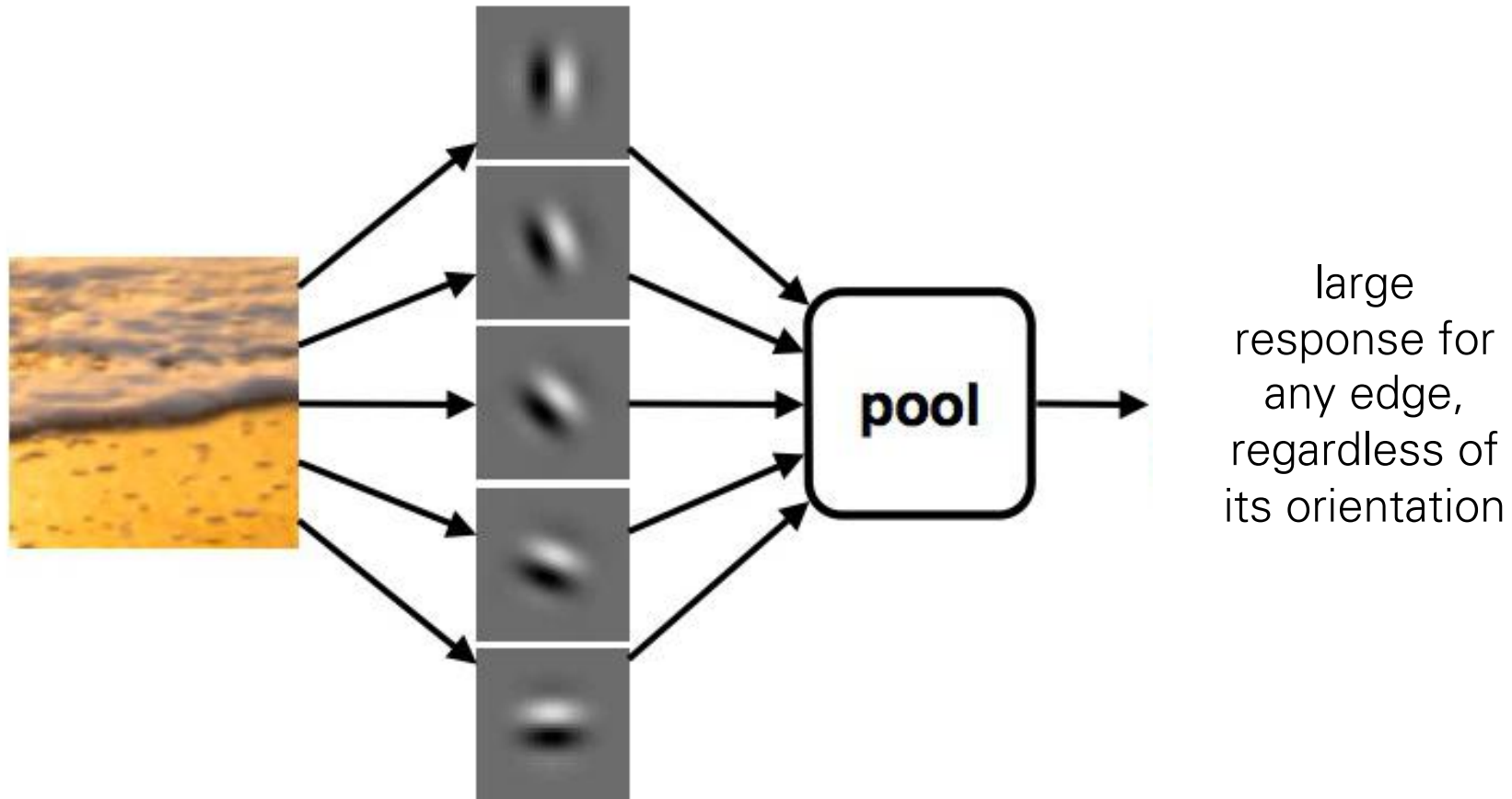
CNNs are stable w.r.t. diffeomorphisms



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

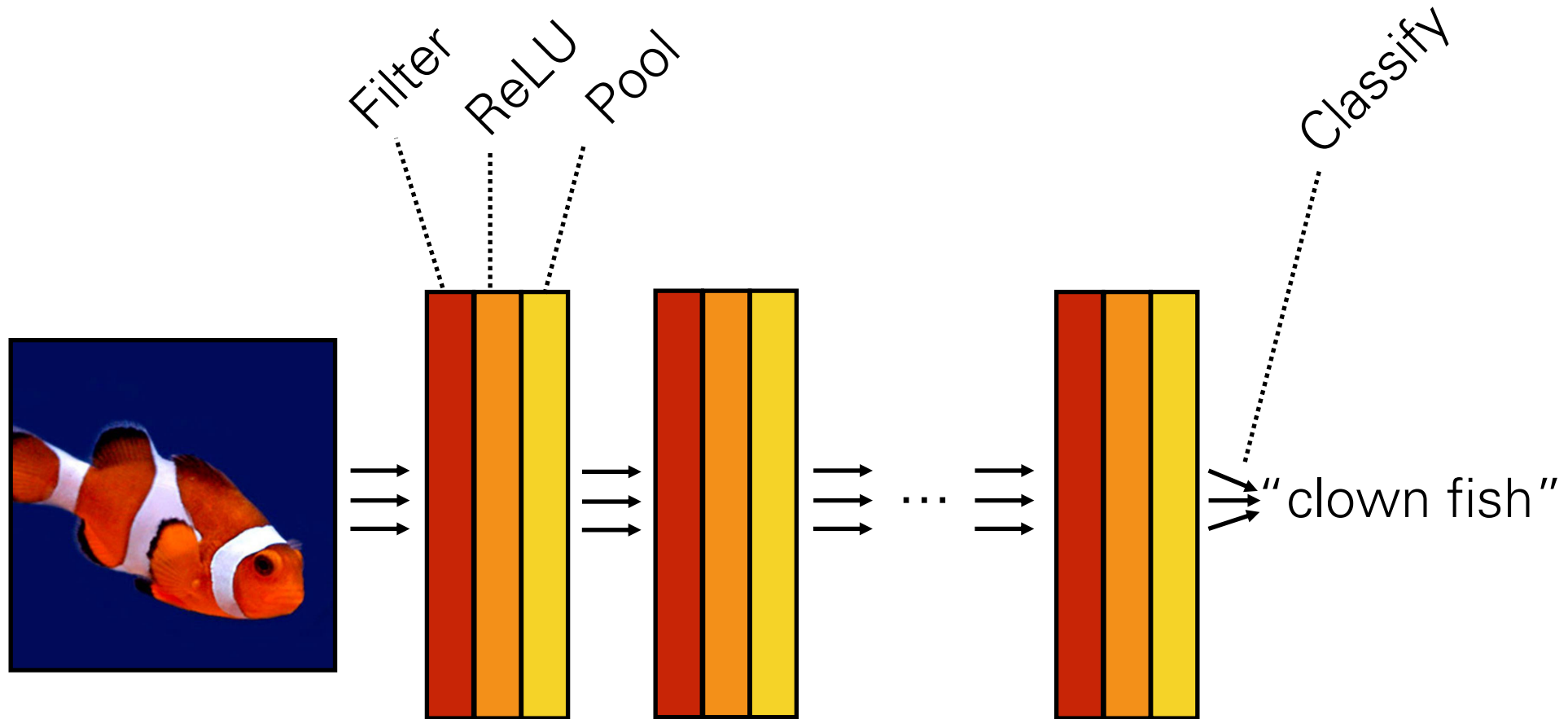
Pooling – Why?

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



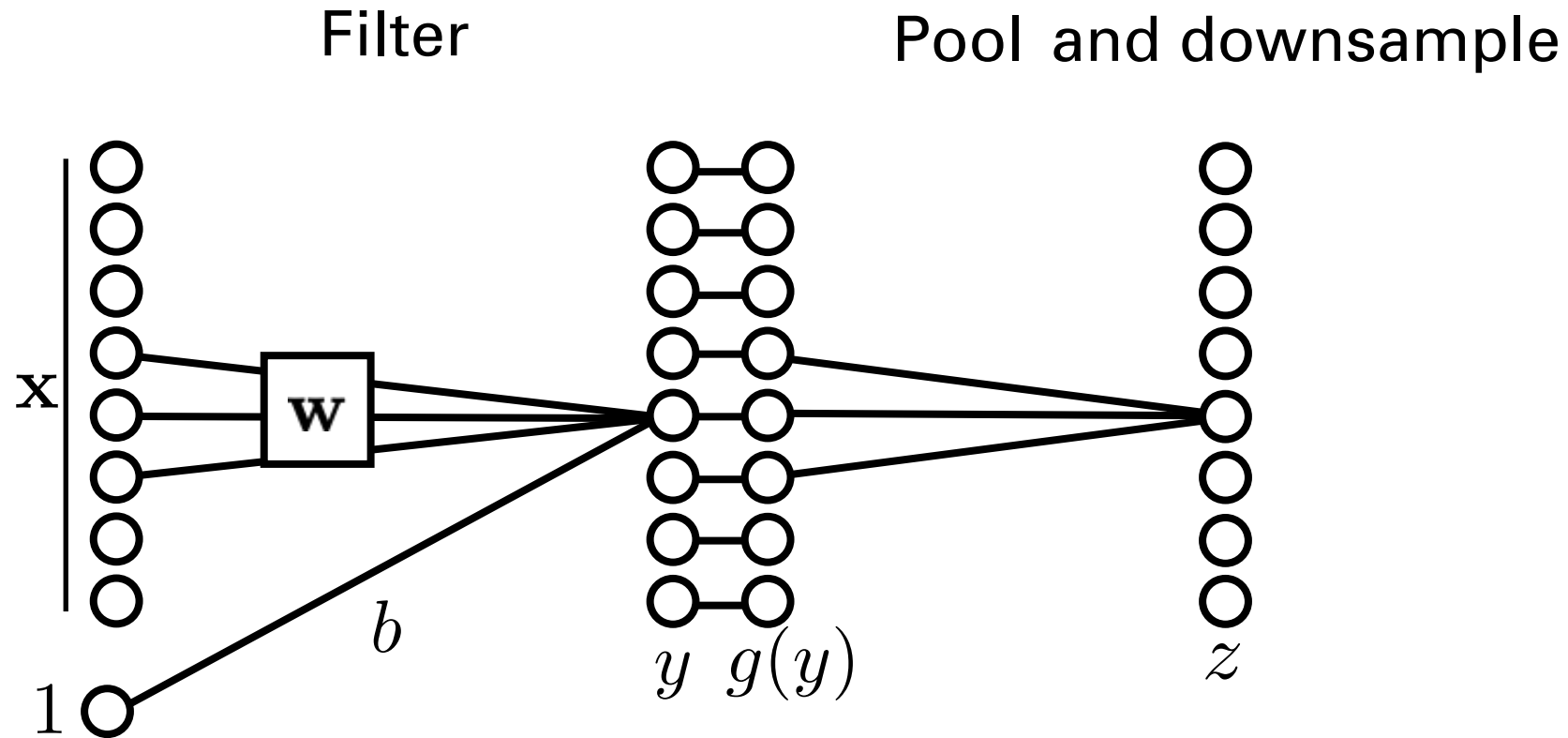
[Derived from slide by Andrea Vedaldi]

Computation in a neural net

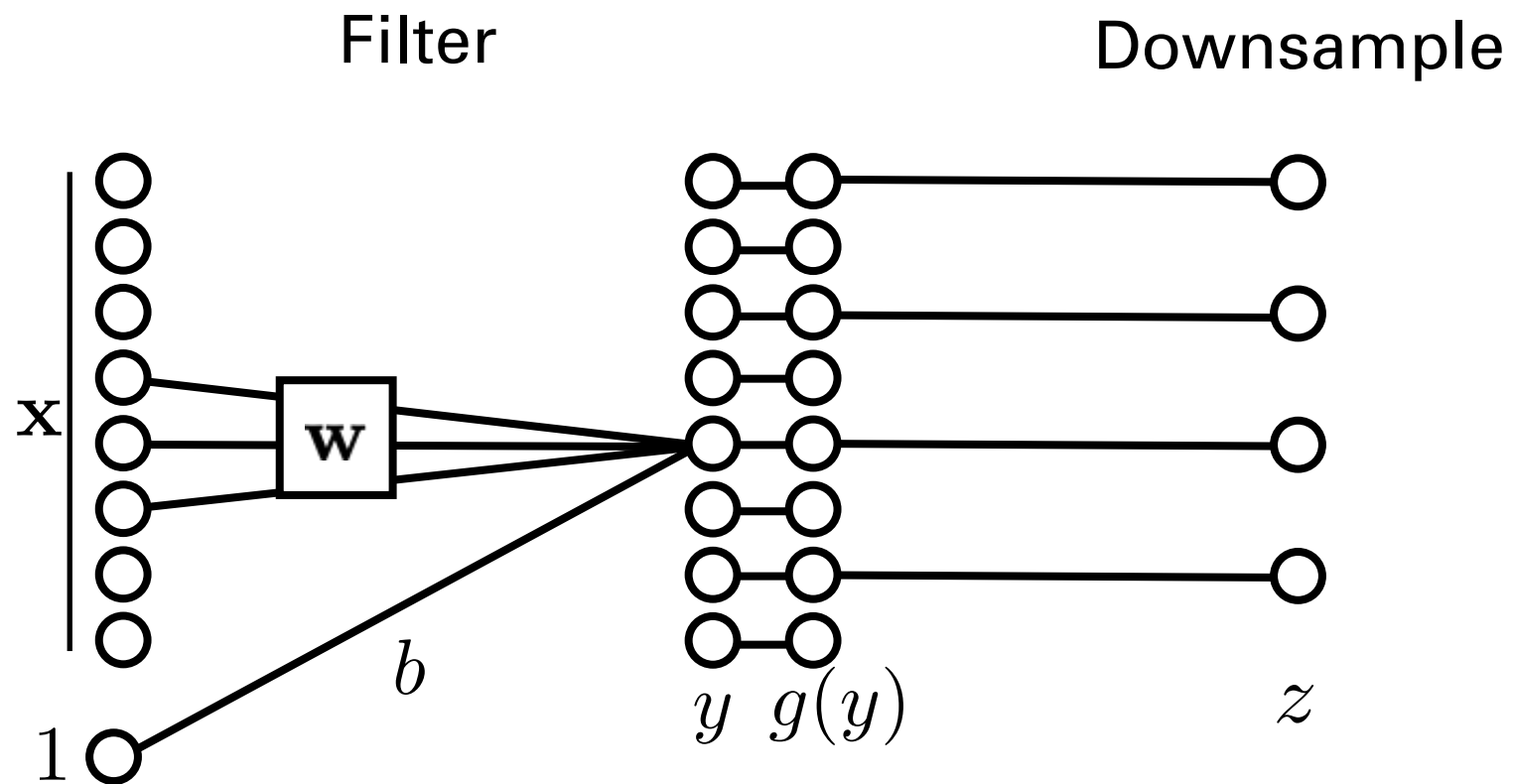


$$f(\mathbf{x}) = f_L(\dots(f_2(f_1(\mathbf{x}))))$$

Downsampling

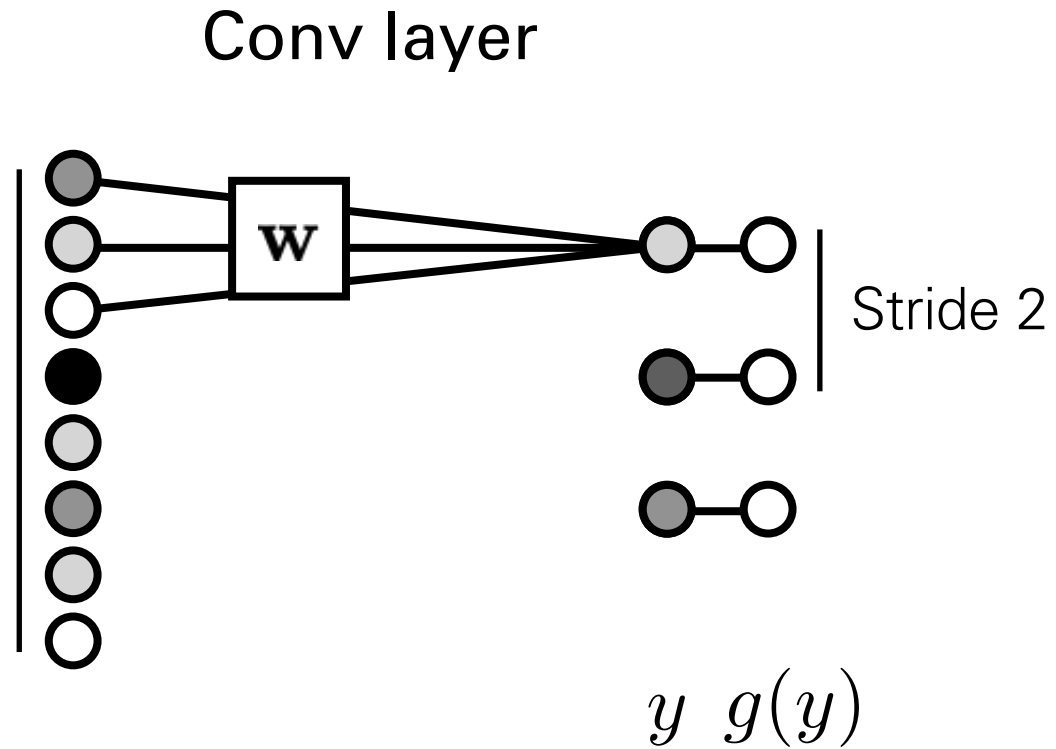


Downsampling



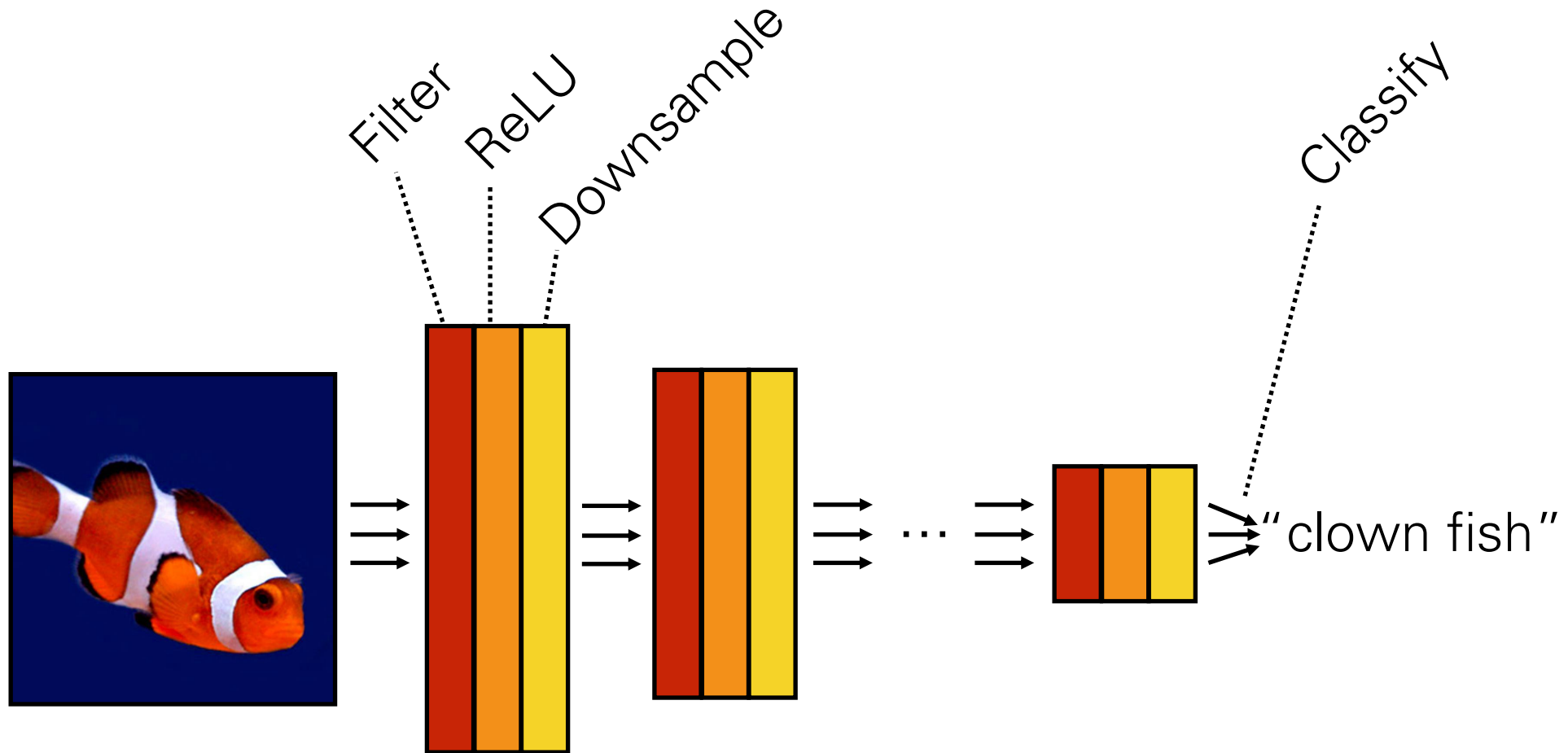
$$\mathbb{R}^{H^{(l)} \times W^{(l)} \times C^{(l)}} \rightarrow \mathbb{R}^{H^{(l+1)} \times W^{(l+1)} \times C^{(l+1)}}$$

Strided convolution



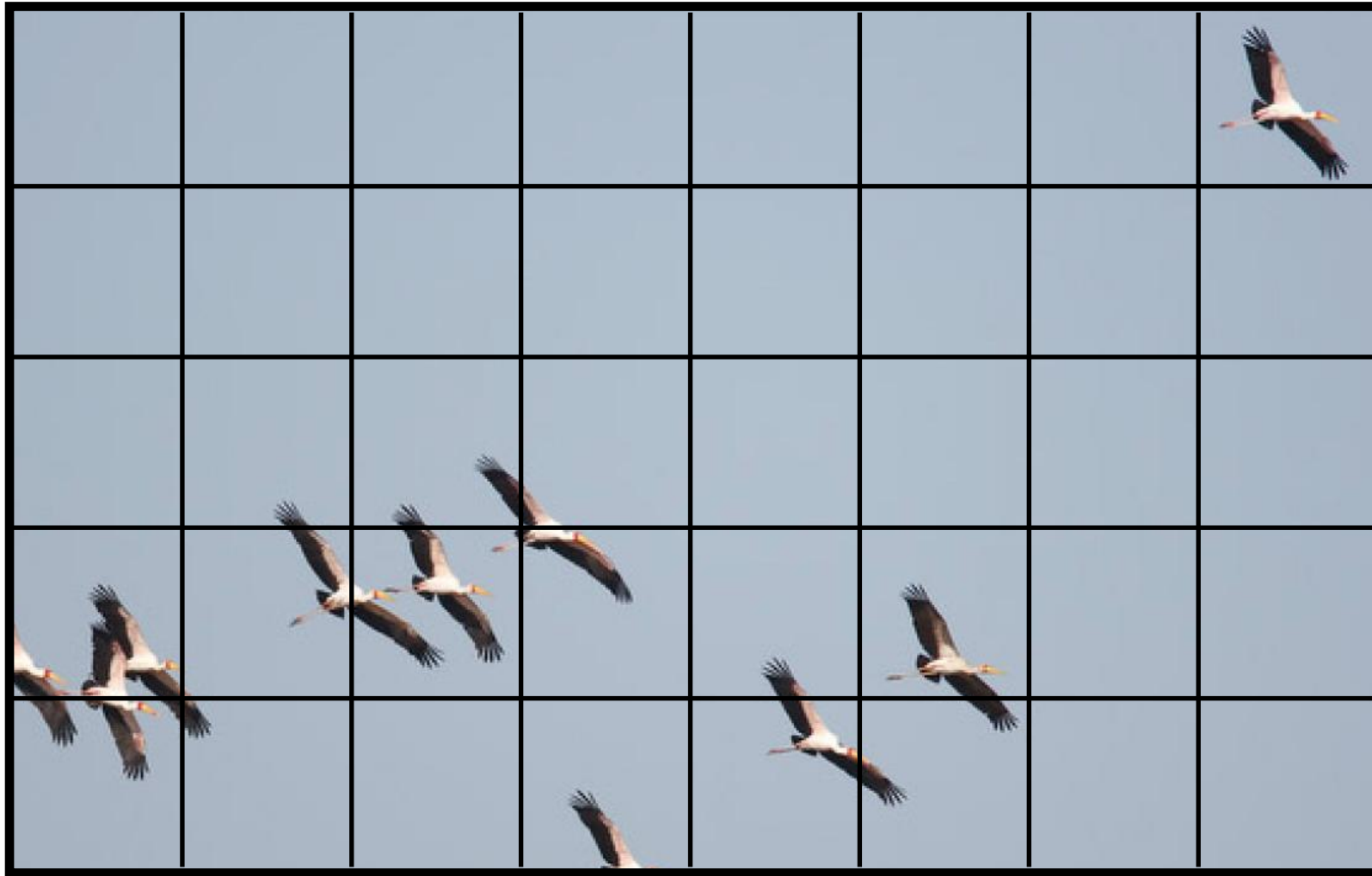
Strided convolutions combine convolution and downsampling into a single operation.

Computation in a neural net

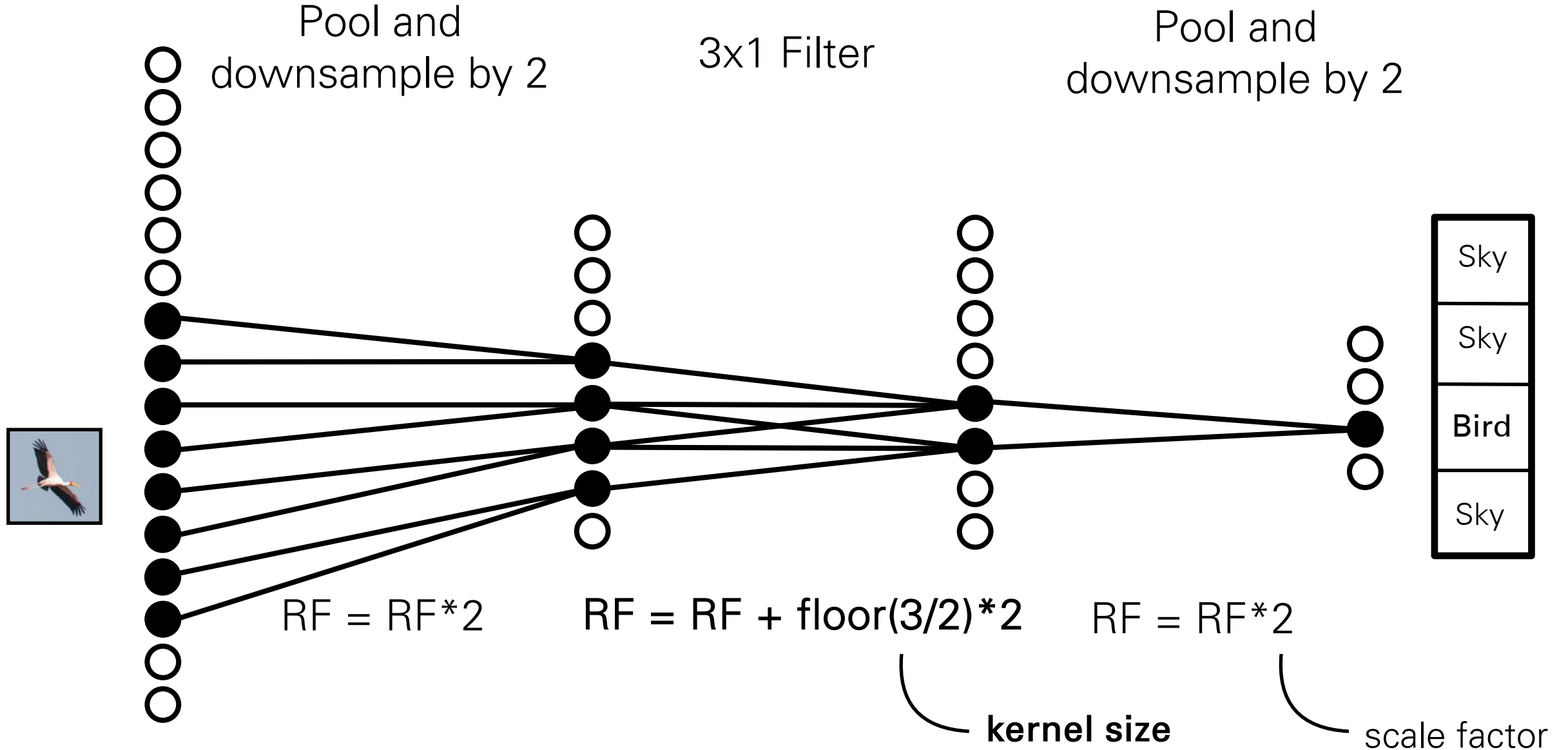


$$f(\mathbf{x}) = f_L(\dots(f_2(f_1(\mathbf{x}))))$$

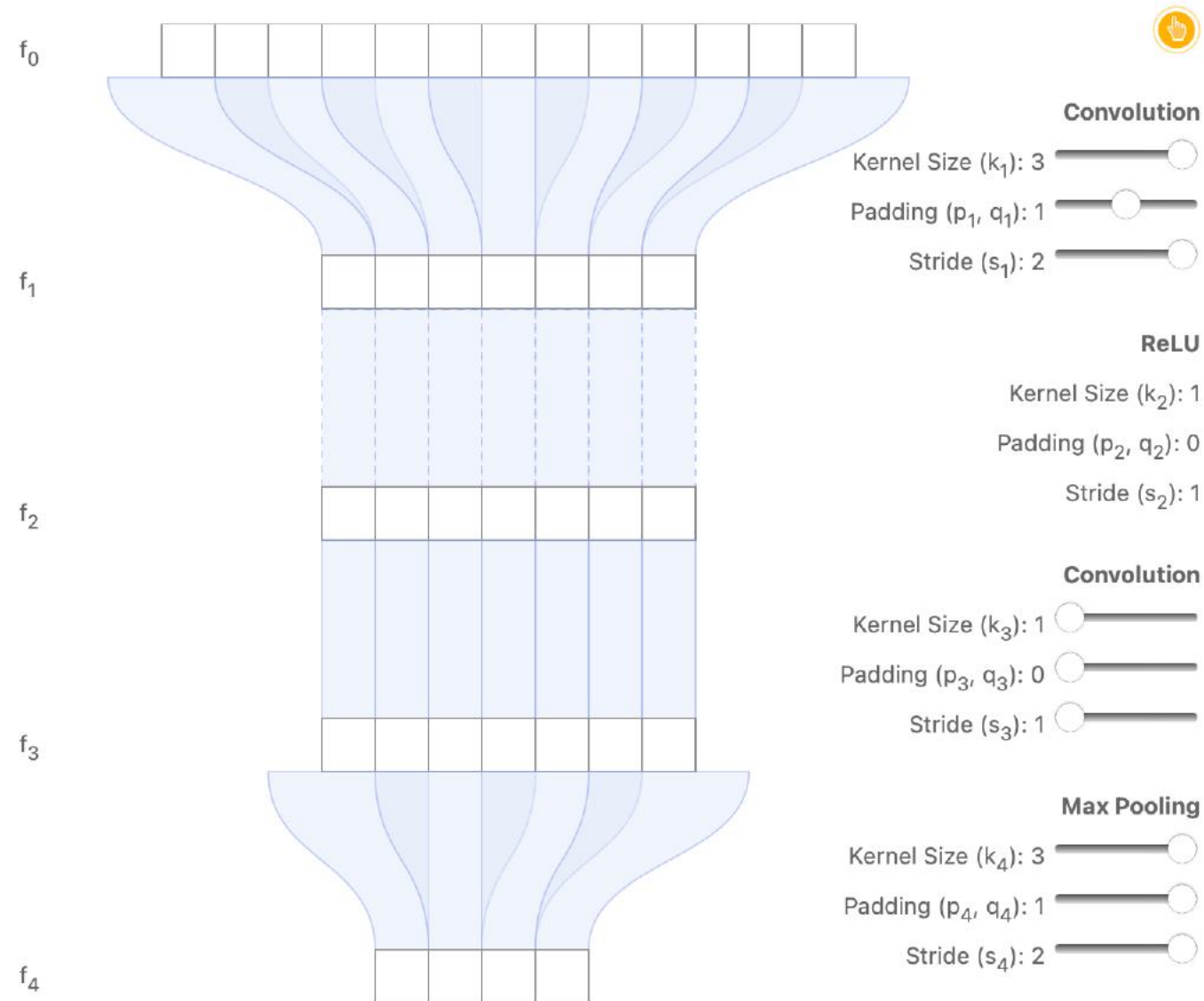
Receptive fields



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?

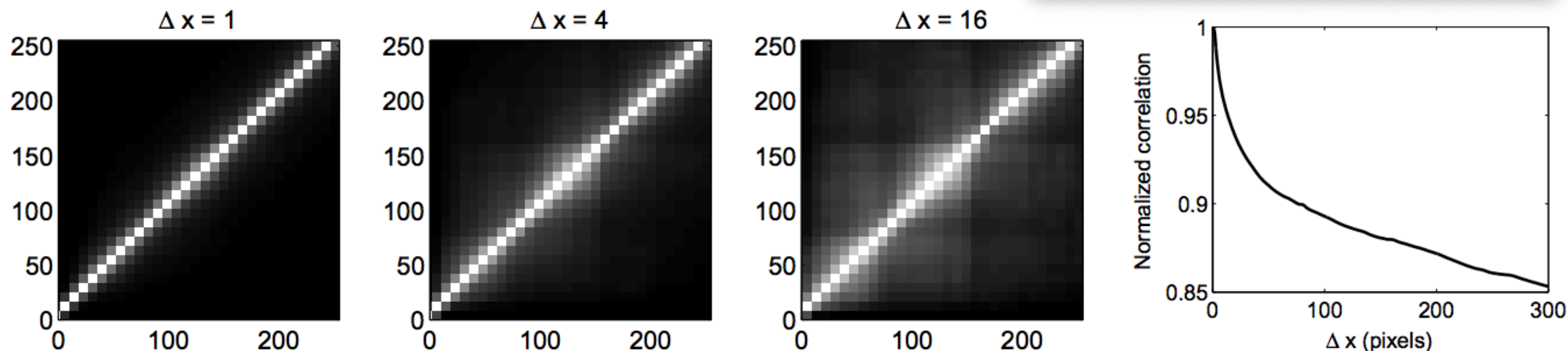
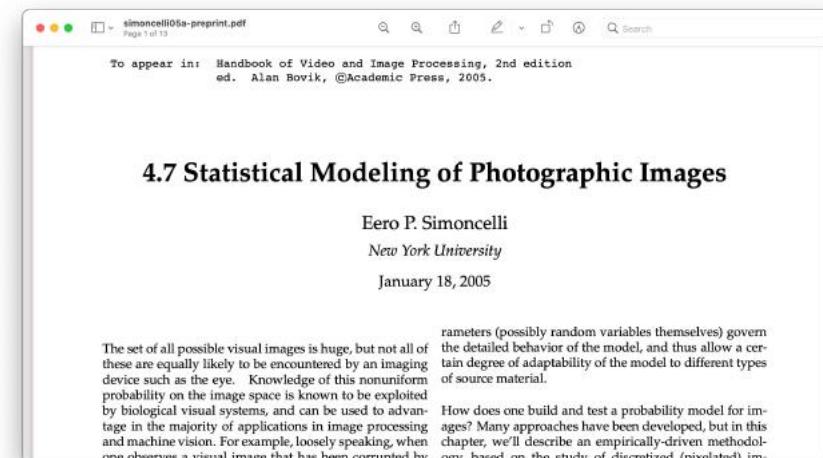


Fig. 1. (a) Scatterplots of pairs of pixels at three different spatial displacements, averaged over five examples images. (b) Autocorrelation function. Photographs are of New York City street scenes, taken with a Canon 10D digital camera, and processed in RAW linear sensor mode (producing pixel intensities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

Why CNNs?

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

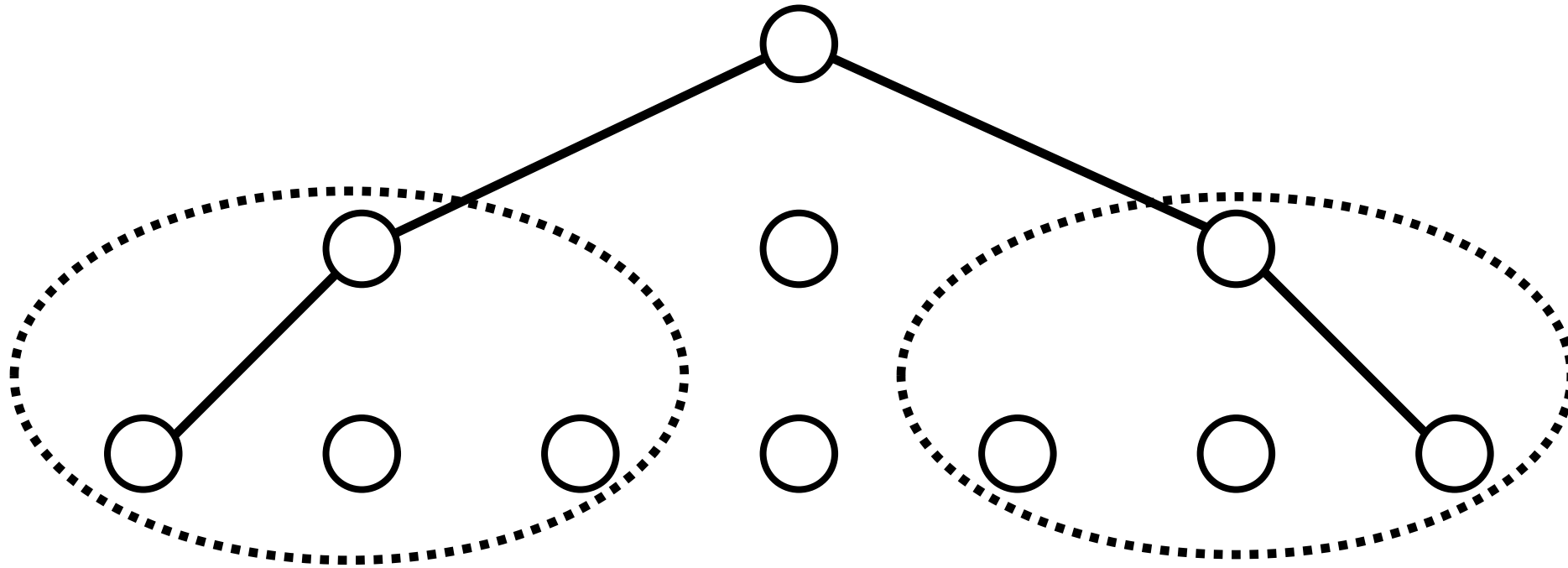
It is therefore often sufficient to model local dependences only. → **Convolution**

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]

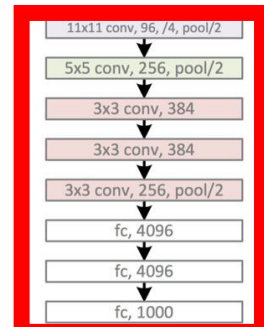
Why CNNs?

Capturing long-range dependences:



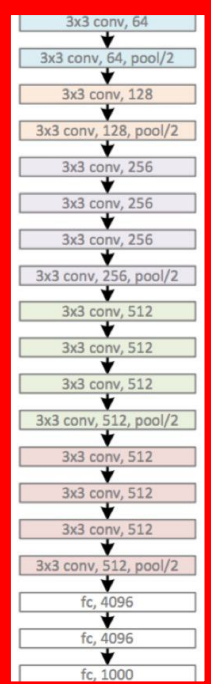
Deep Neural Networks for Visual Recognition

2012: AlexNet
5 conv. layers



Error: 15.3%

2014: VGG
16 conv. layers



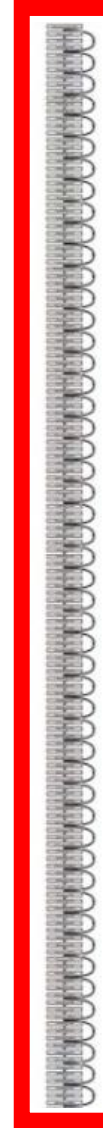
Error: 8.5%

2015: GoogLeNet
22 conv. layers



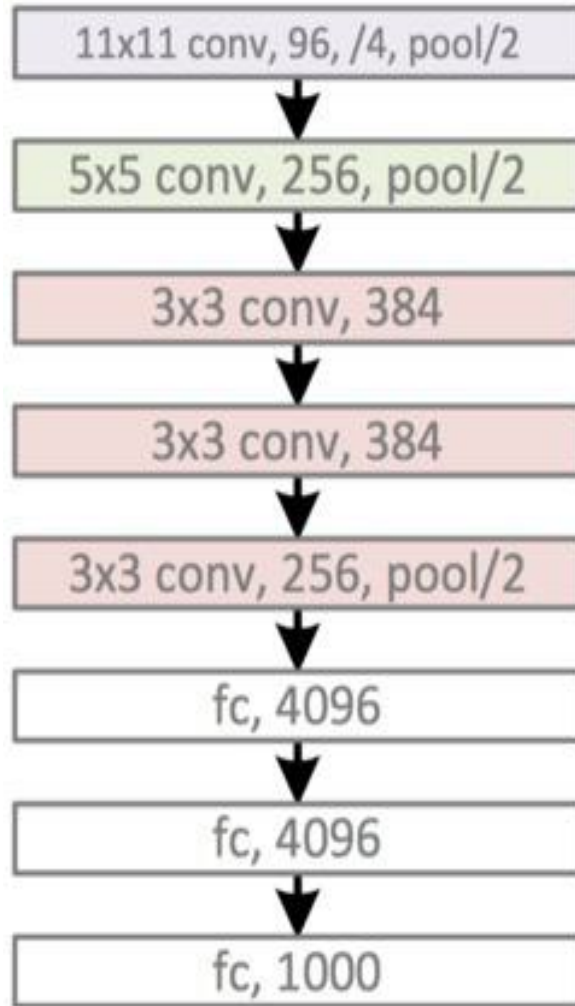
Error: 7.8%

2016: ResNet
>100 conv. layers



Error: 4.4%

2012: AlexNet 5 conv. layers

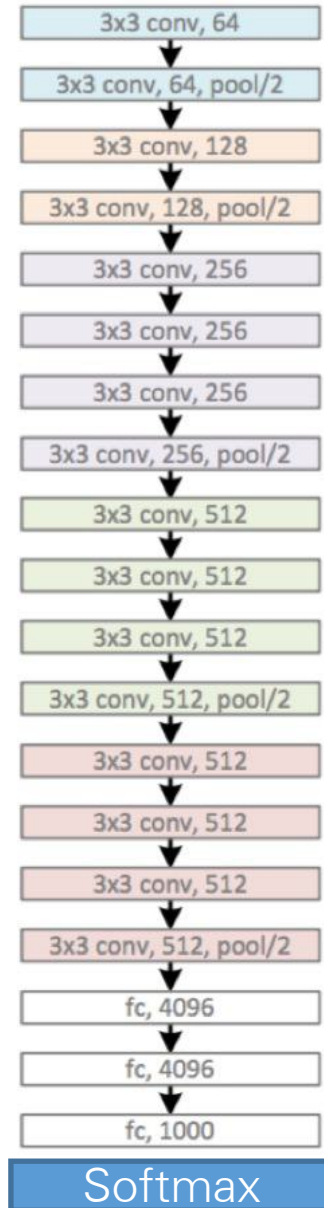


Error: 15.3%

VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION

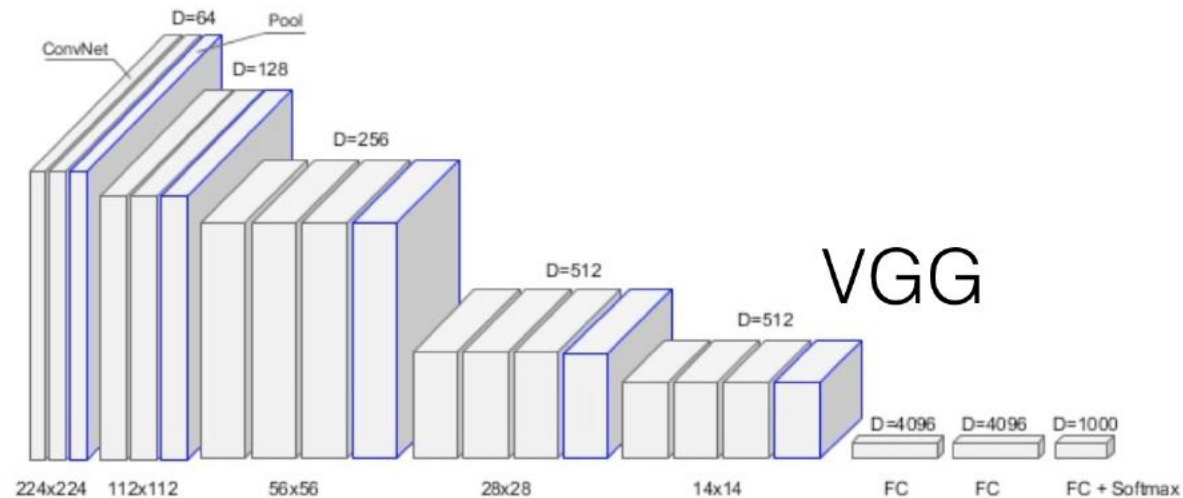
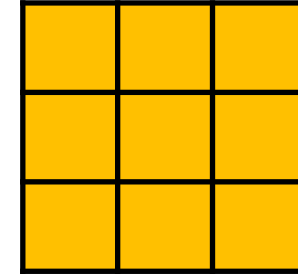
<https://arxiv.org/pdf/1409.1556.pdf>

2014: VGG
16 conv.
layers

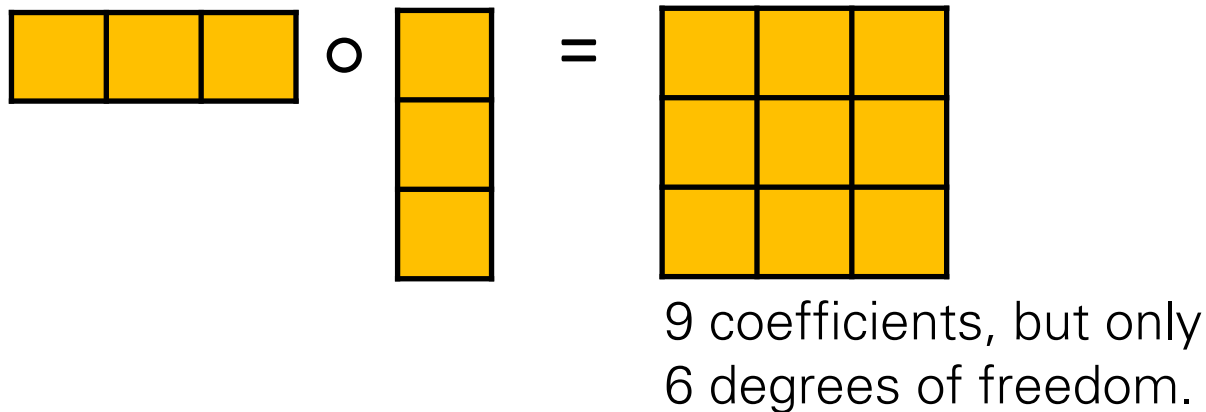
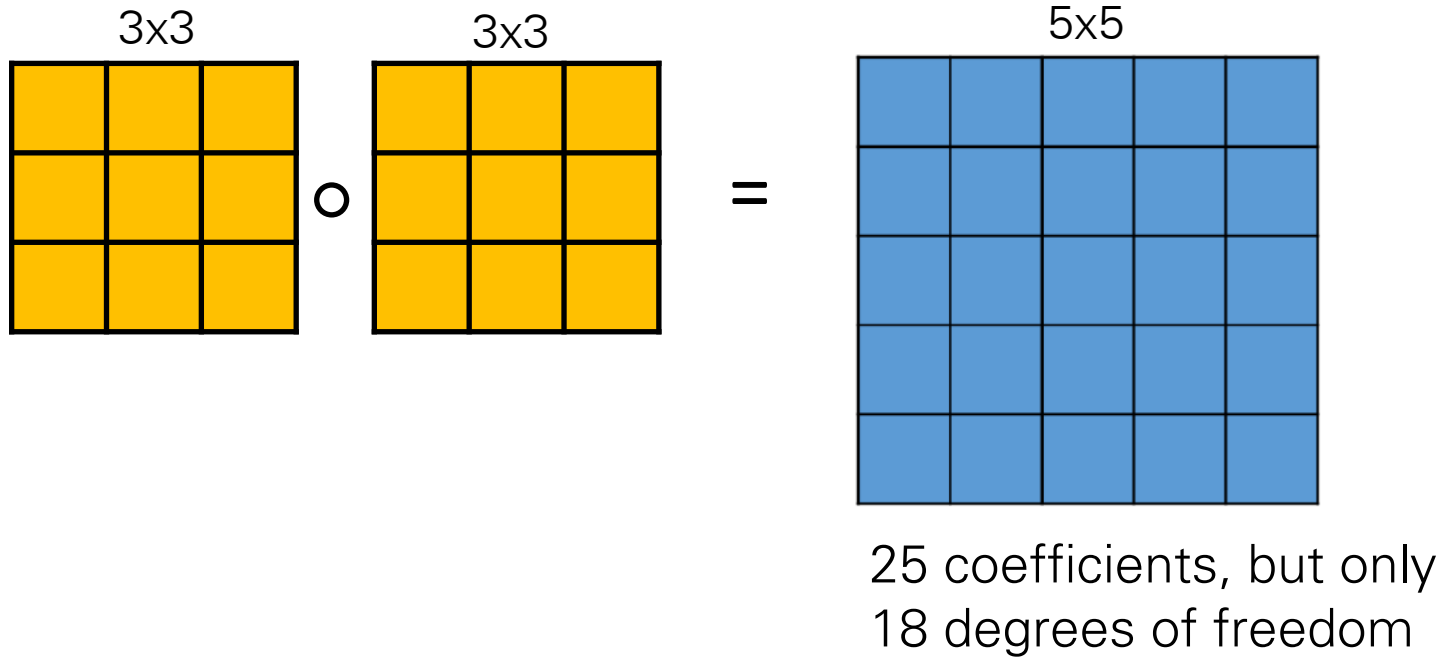


Error: 8.5%

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

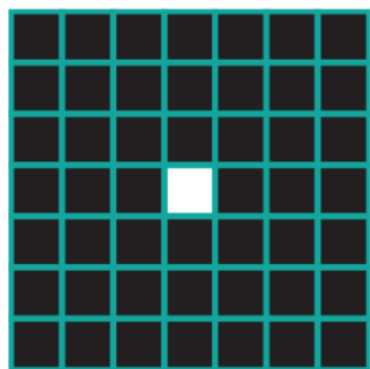
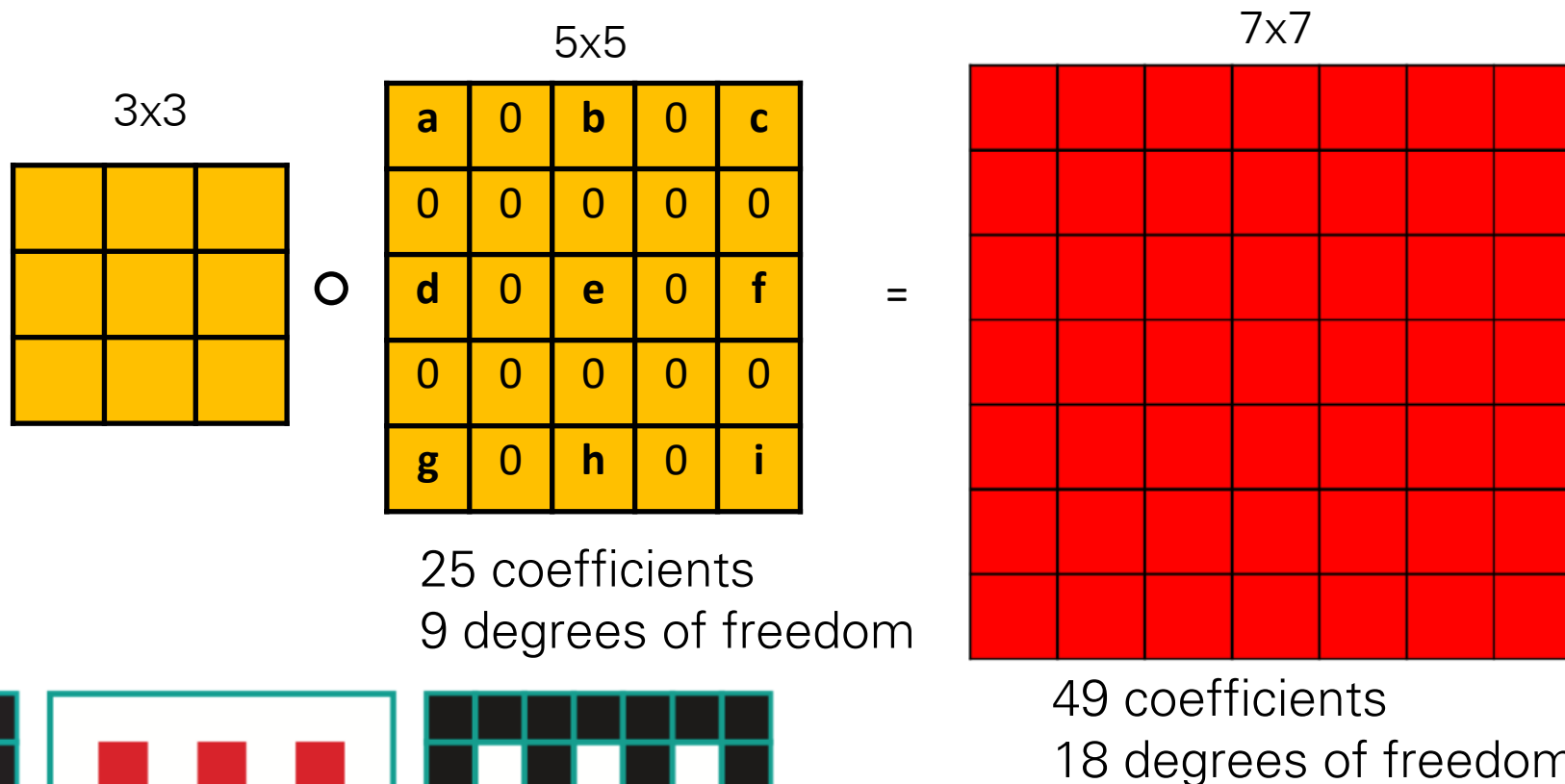


Chaining convolutions

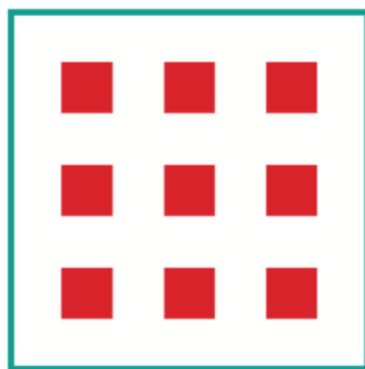


Only separable filters... would this be enough?

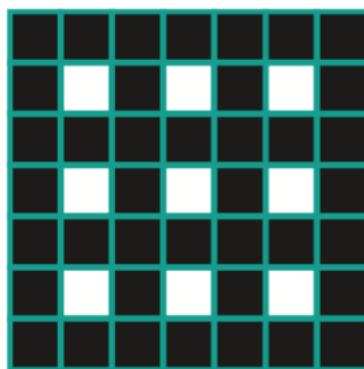
Dilated convolutions



(a) Input



(b) Dilation 2



(c) Output

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

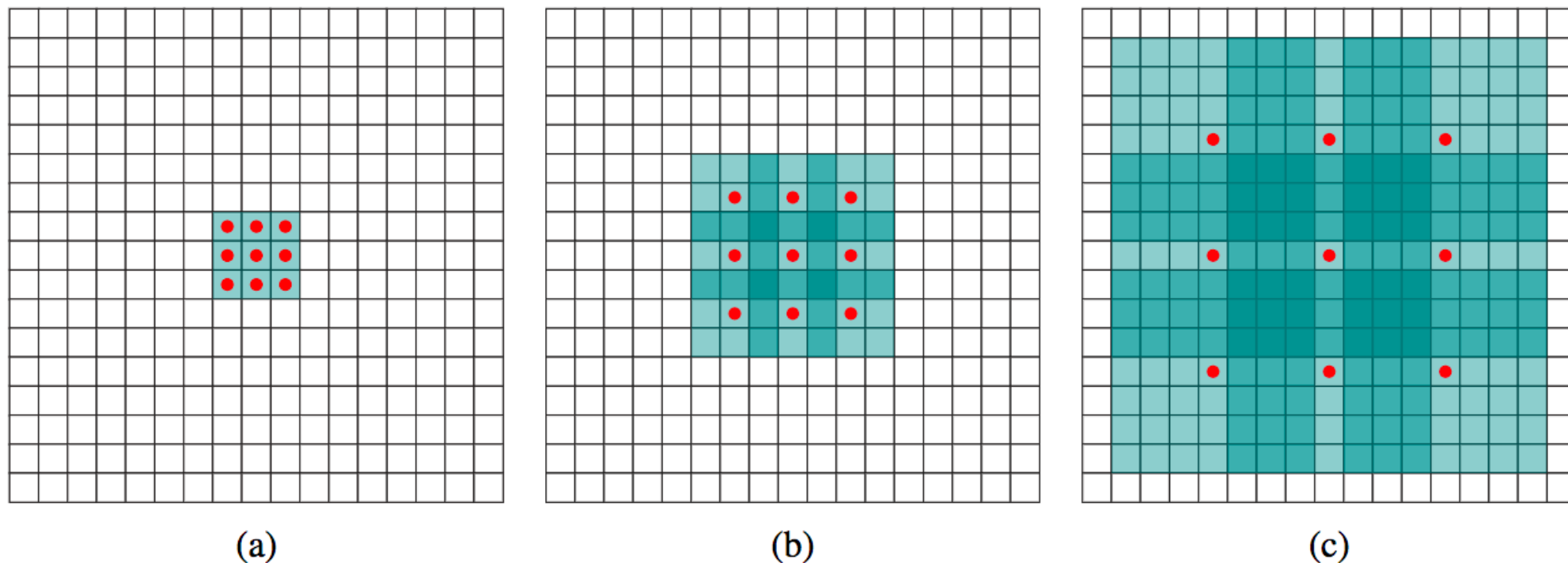
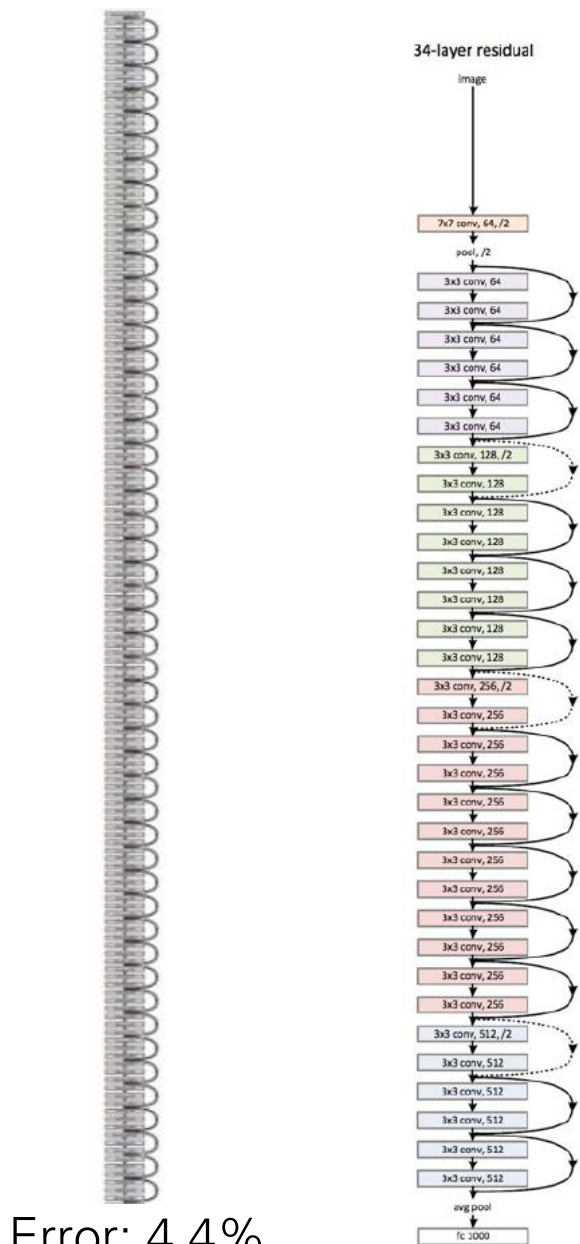


Figure 1: Systematic dilation supports exponential expansion of the receptive field without loss of resolution or coverage. (a) F_1 is produced from F_0 by a 1-dilated convolution; each element in F_1 has a receptive field of 3×3 . (b) F_2 is produced from F_1 by a 2-dilated convolution; each element 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.

2016: ResNet
>100 conv. layers

Deep Residual Learning for Image Recognition

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



Error: 4.4%

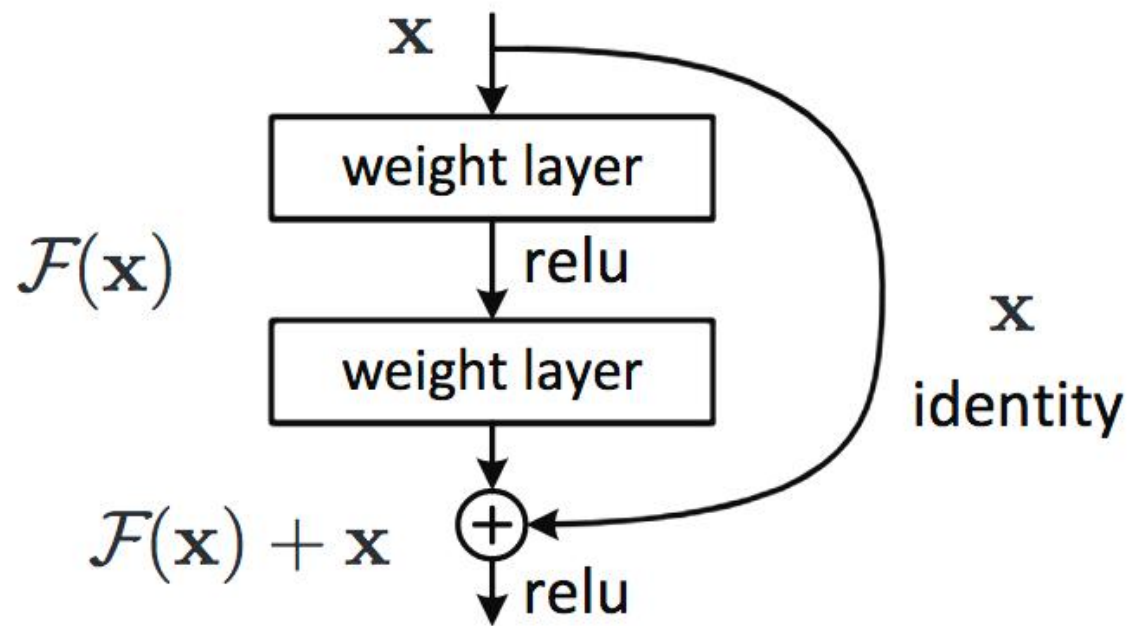
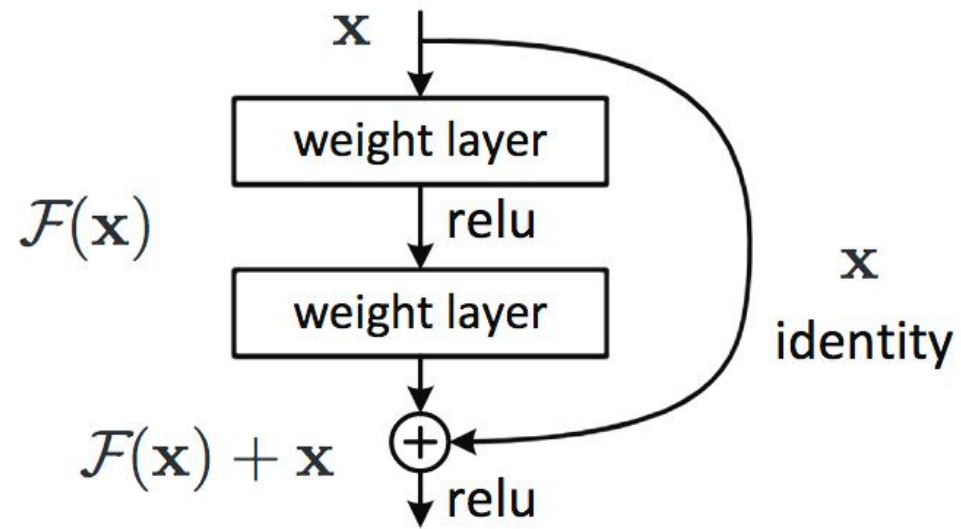
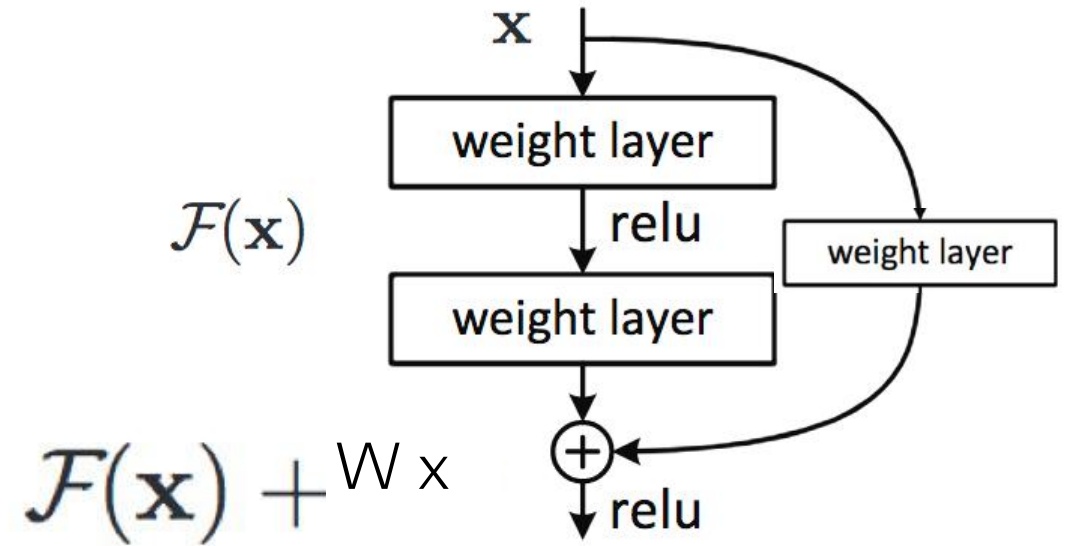


Figure 2. Residual learning: a building block.

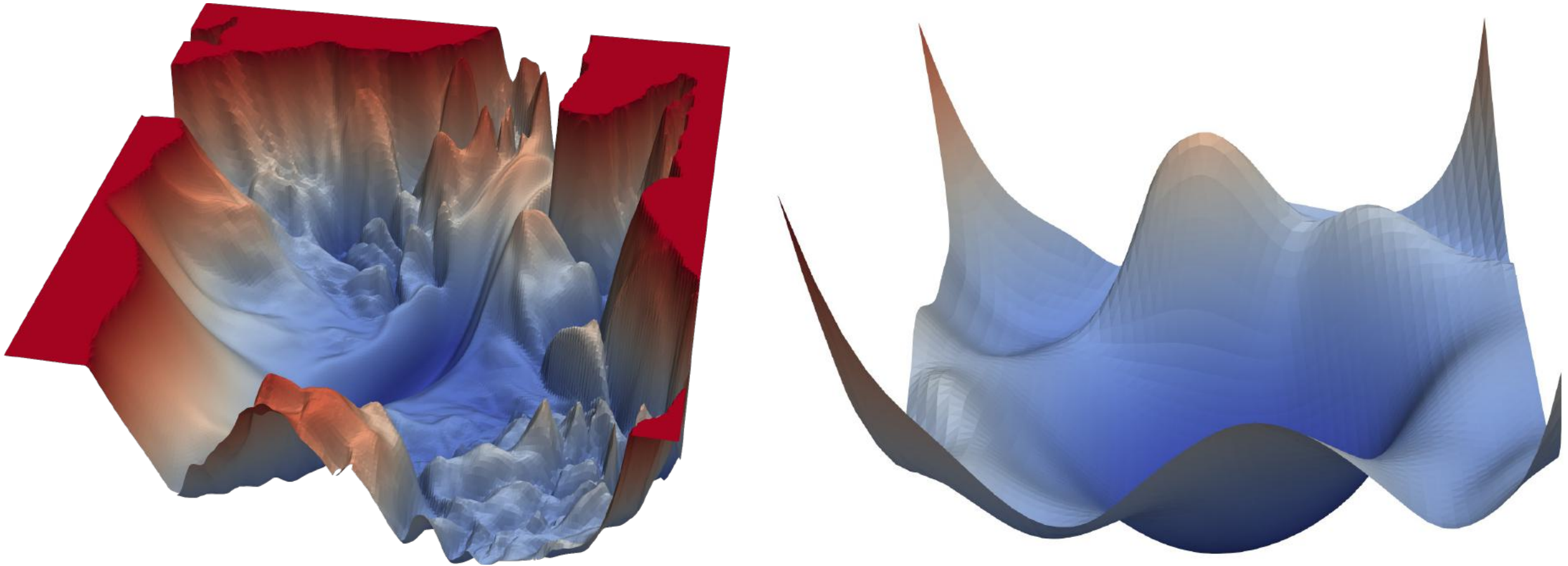
If output has same size as input:



If output has a different size:

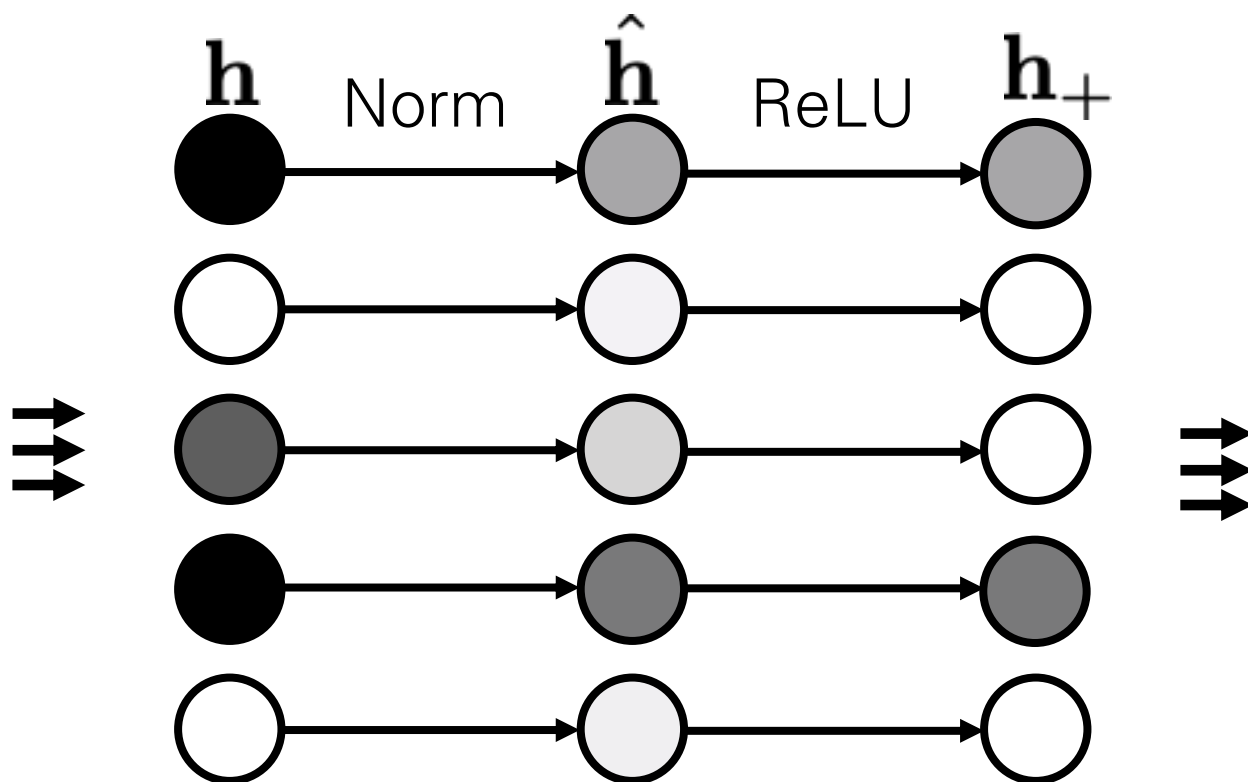


Residual Learning



- 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.

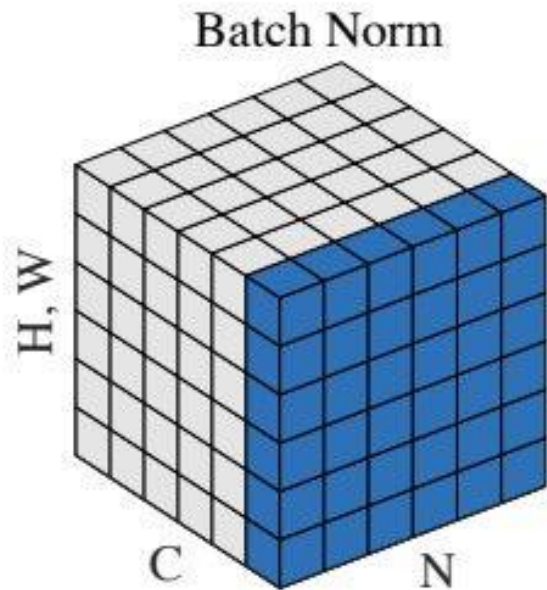
Squashes units into a **standard range**, avoiding overflow.

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

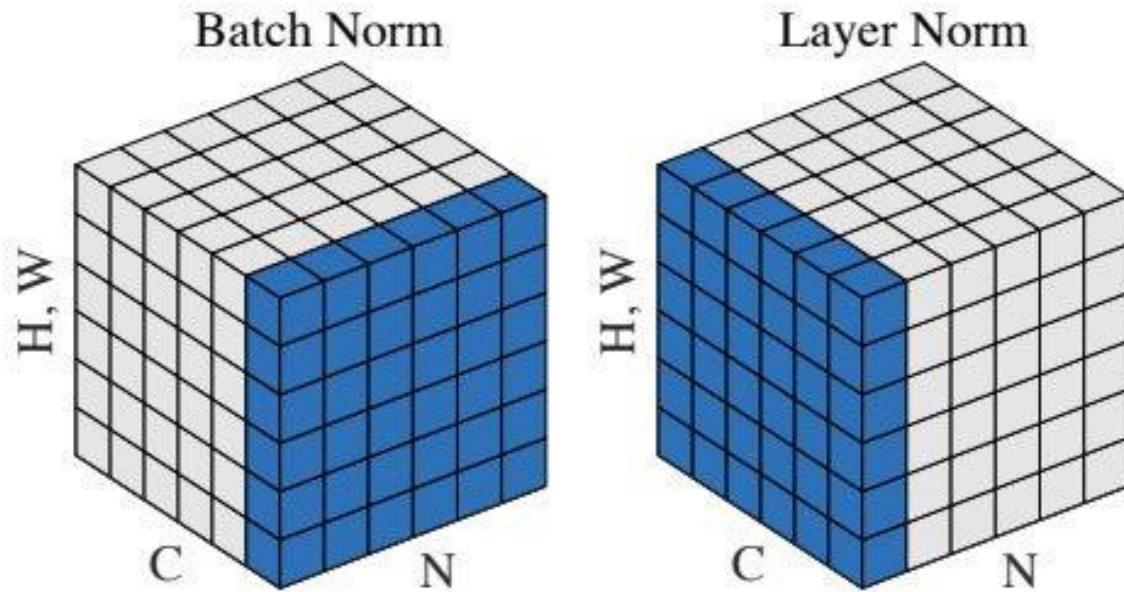
Normalization layers



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]

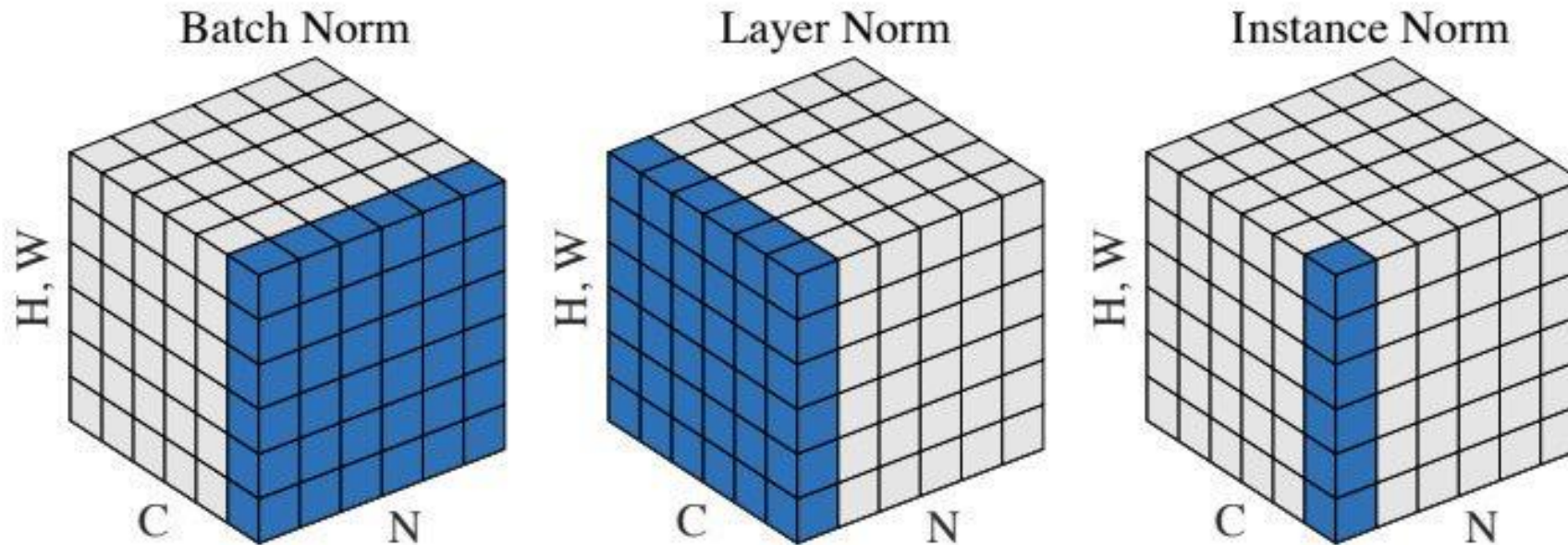
Normalization layers



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]

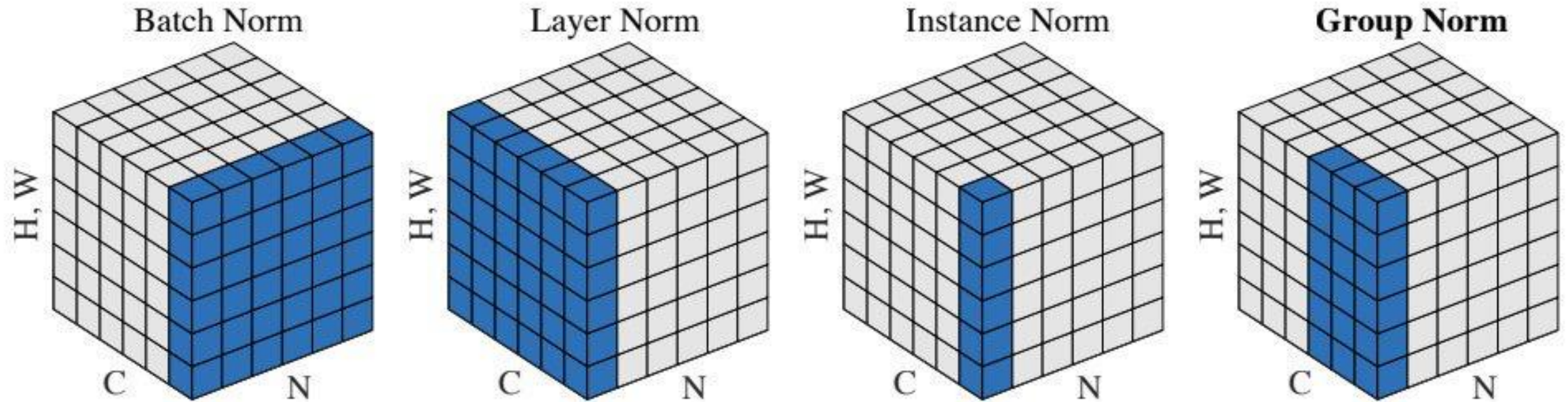
Normalization layers



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

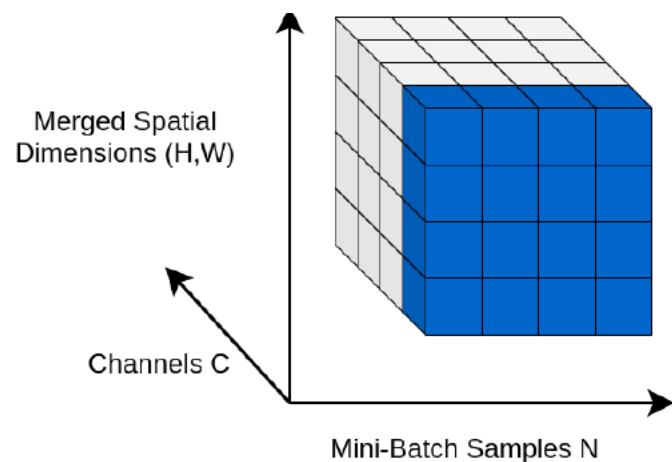


Might as well...

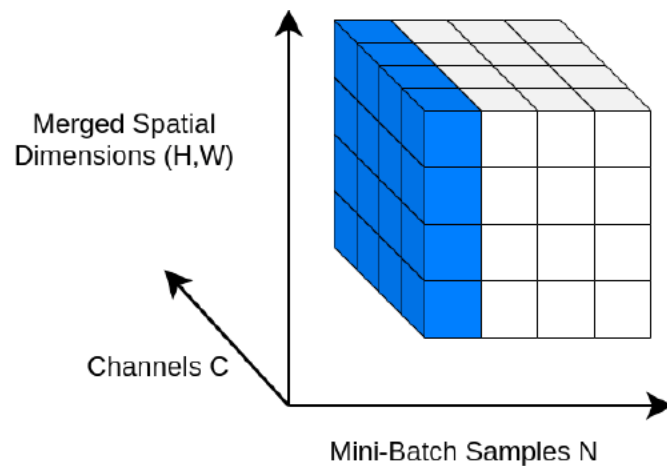
[Figure from Wu & He, arXiv 2018]

Normalization layers

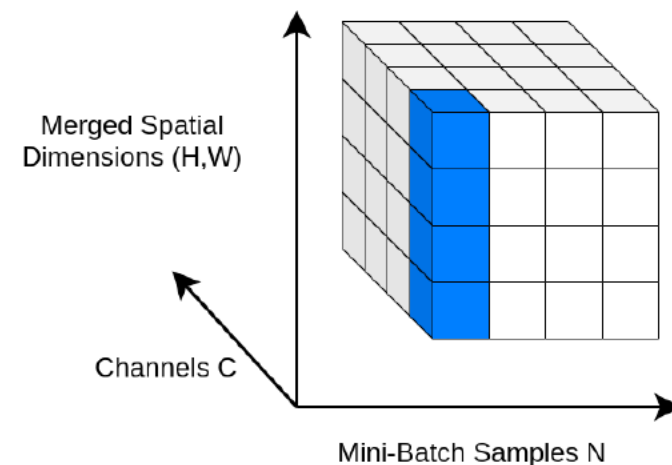
Batch Normalization (2015)



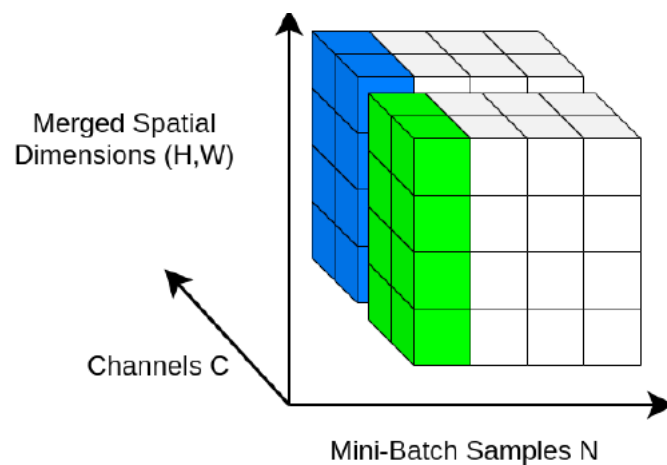
Layer Normalization (2016)



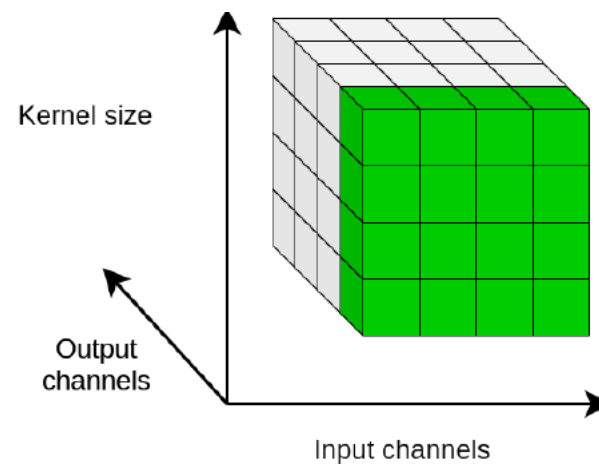
Instance Normalization (2016)



Group Normalization (2018)

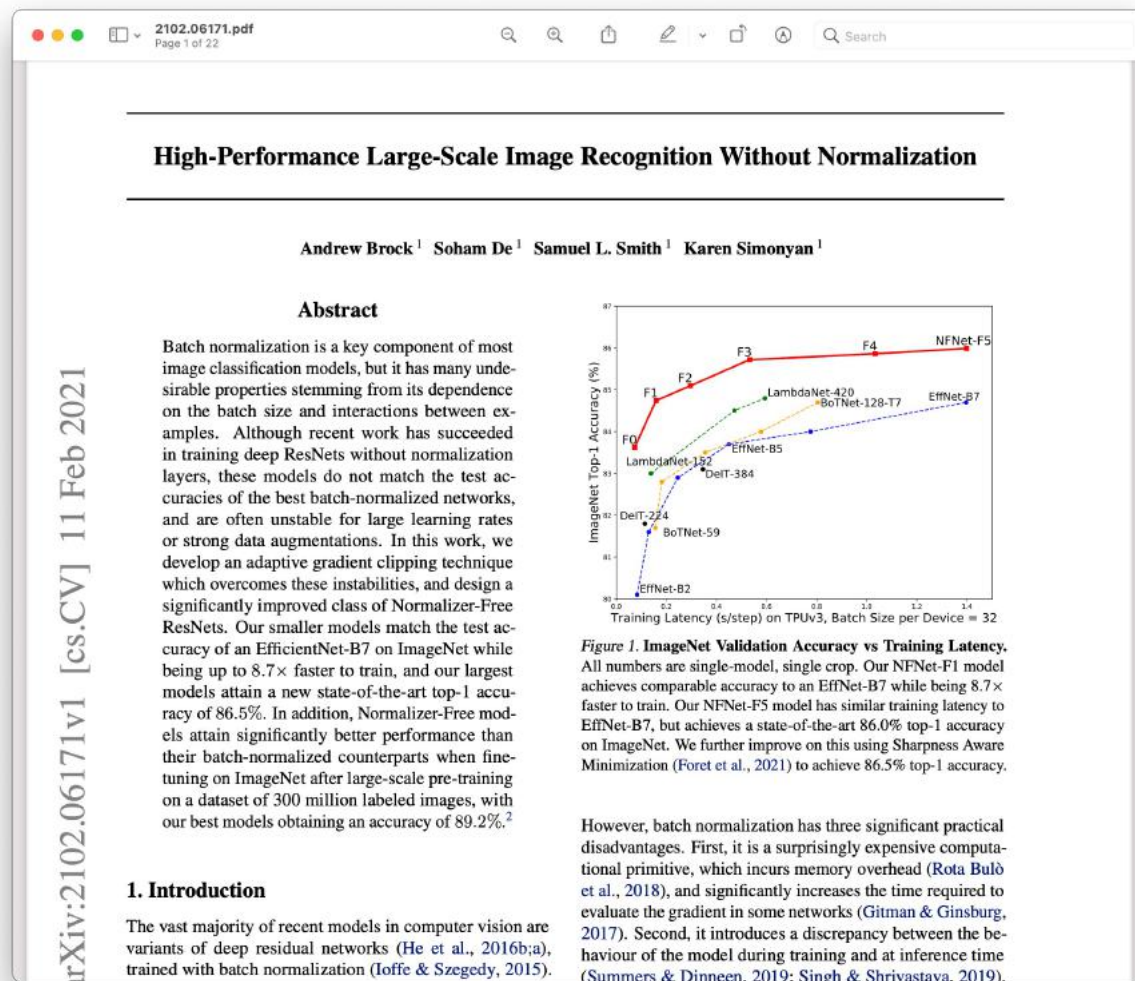
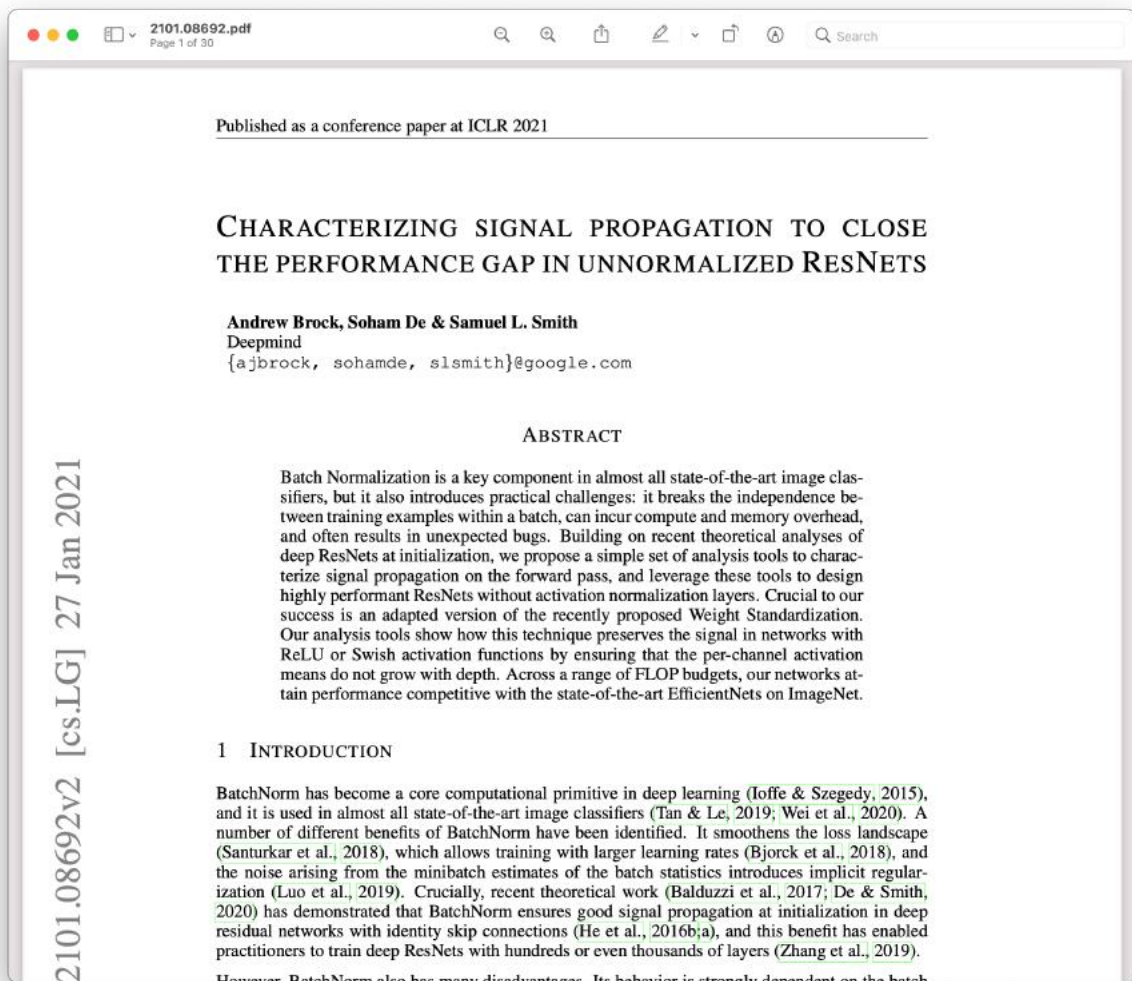


Weight Standardization (2019)



[\[https://theaisummer.com/normalization\]](https://theaisummer.com/normalization)

No normalization layers



Next Lecture:

Sequential Processing with RNNs