

# Lecture 6: Linear classifiers

# Announcements

- Goodfellow book
- Szeliski 5.1-5.3 (can skip unsupervised learning)
- PS1 grades won't be out for a while, due to late submissions

# Announcements

- Guest lecture Weds. (9/20)
- Zoom lectures next week (9/25 and 9/20)
- Guest lectures the following week (10/2 and 10/4)



Sarah Jabbour

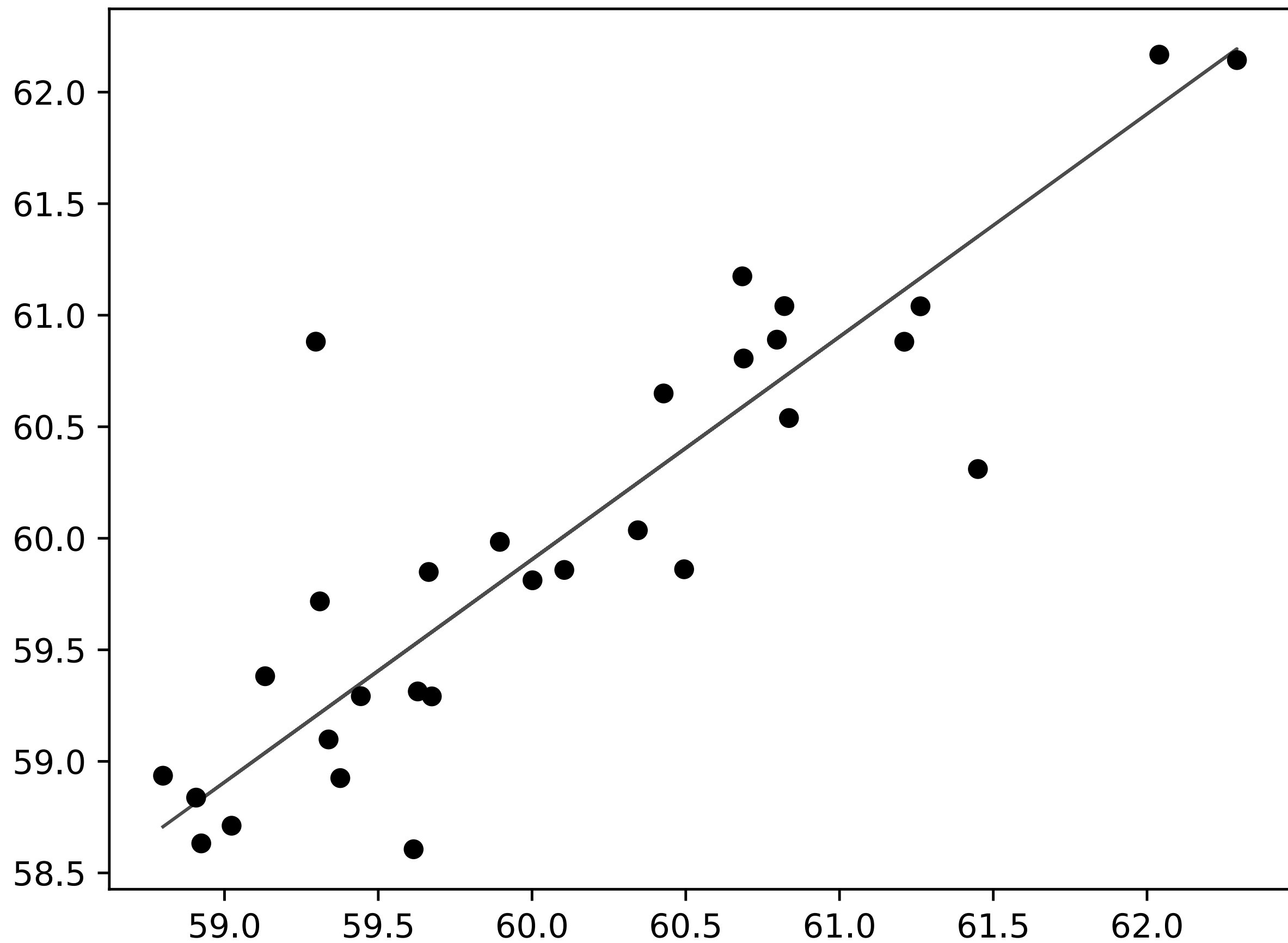


Yiming Dou



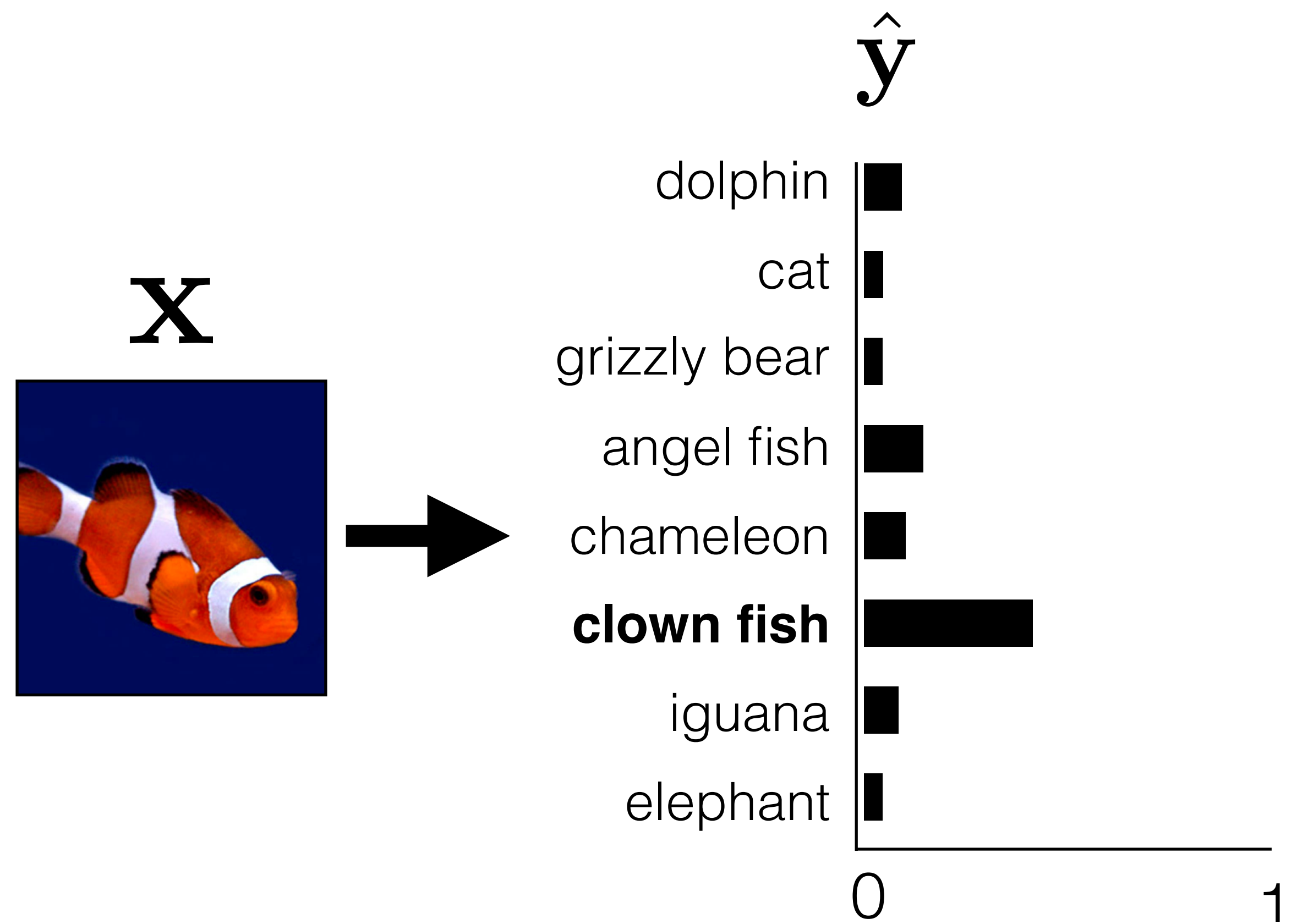
Daniel Geng

# Linear regression



Predict scalar  $y$  from vector  $x$

# Logistic regression



Predict probability of classes from vector  $x$

# Image classification

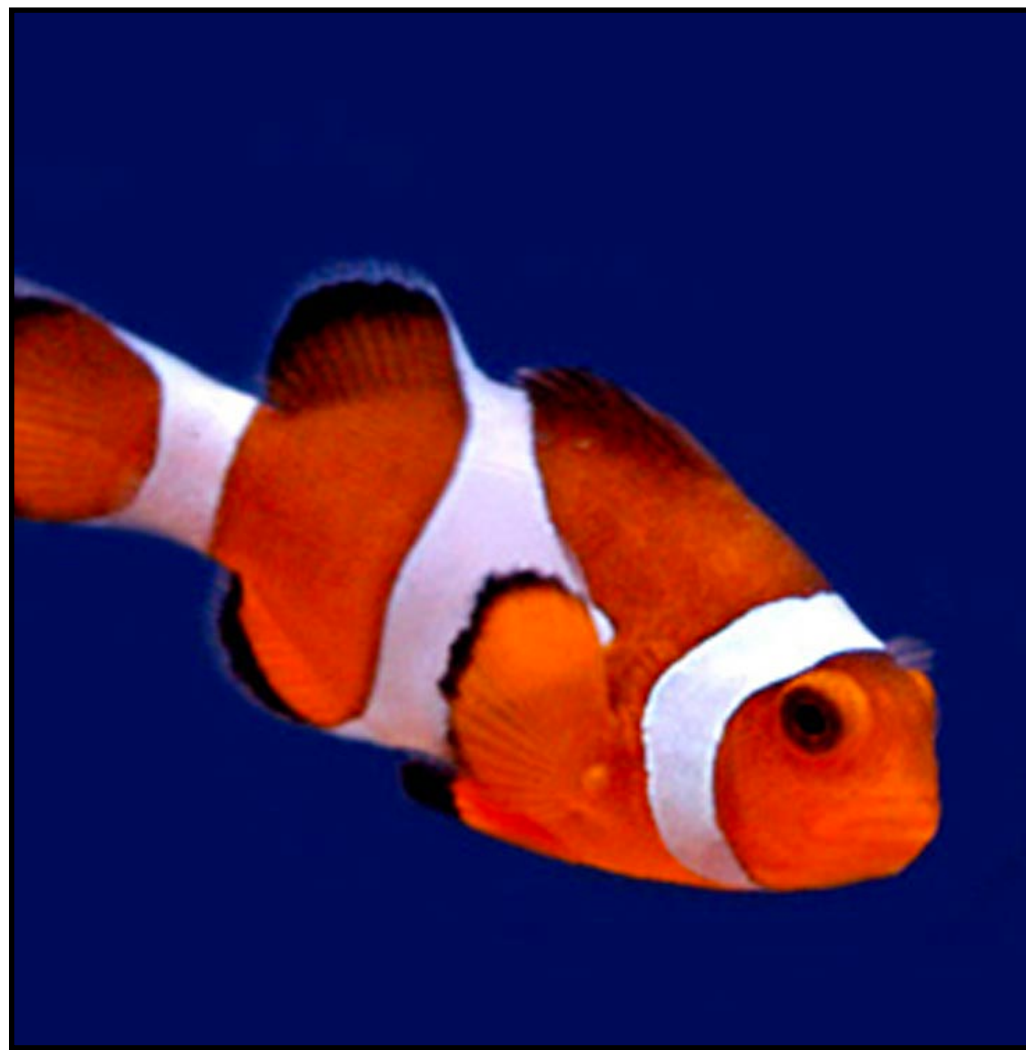


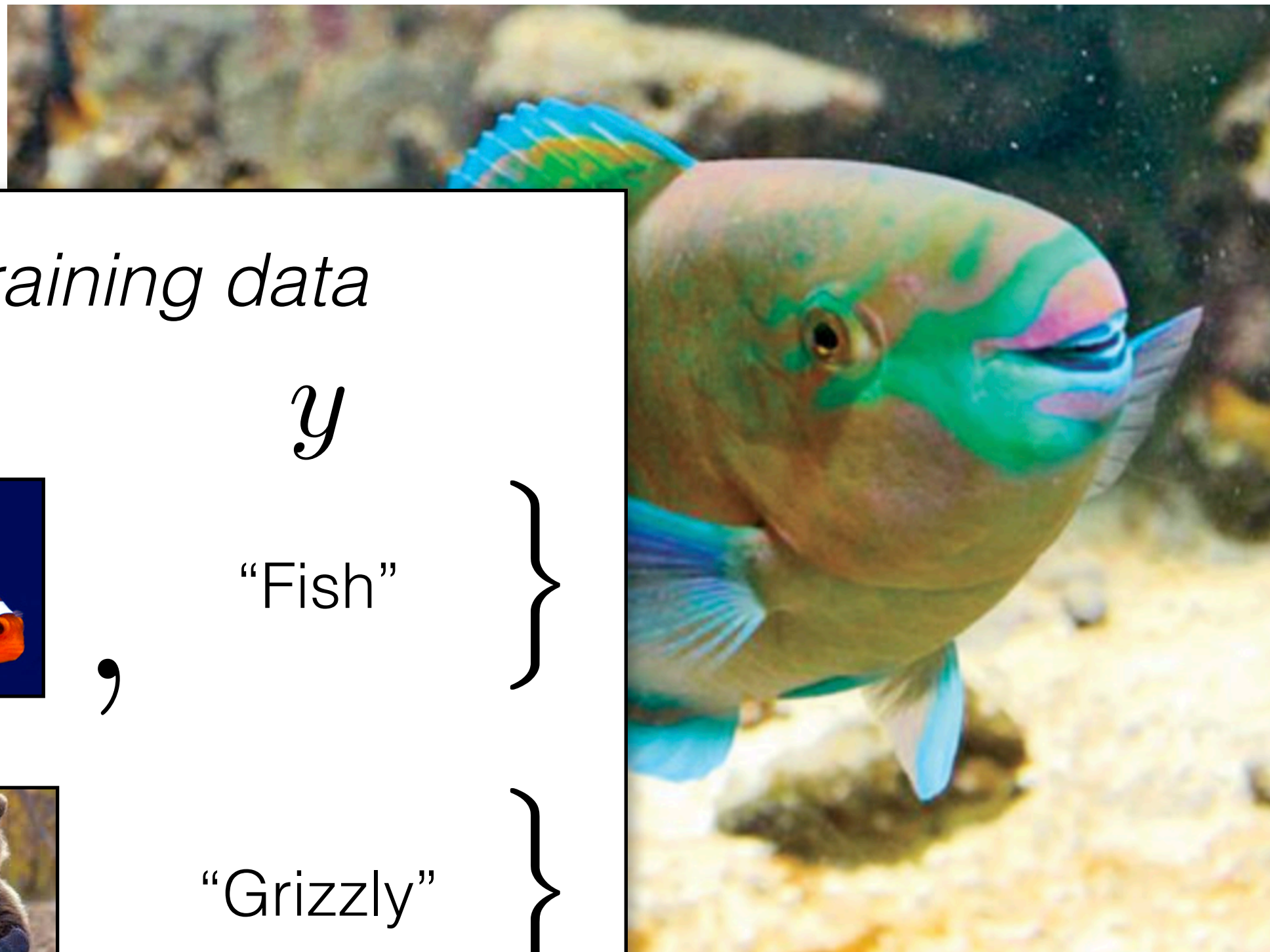
image  $x$



"Fish"

label  $y$

$\mathbf{x}$



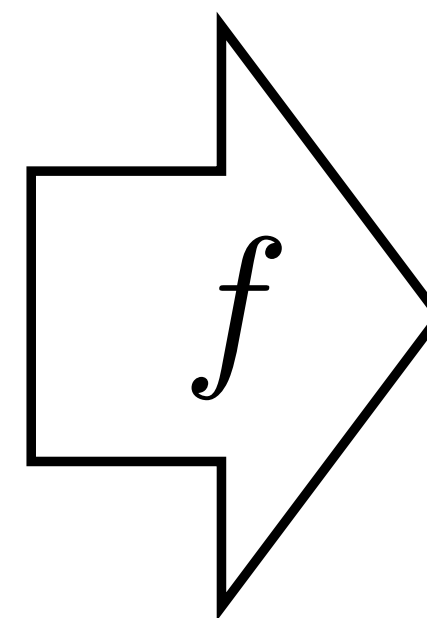
Training data

$\mathbf{x}$

$y$



⋮



$y$

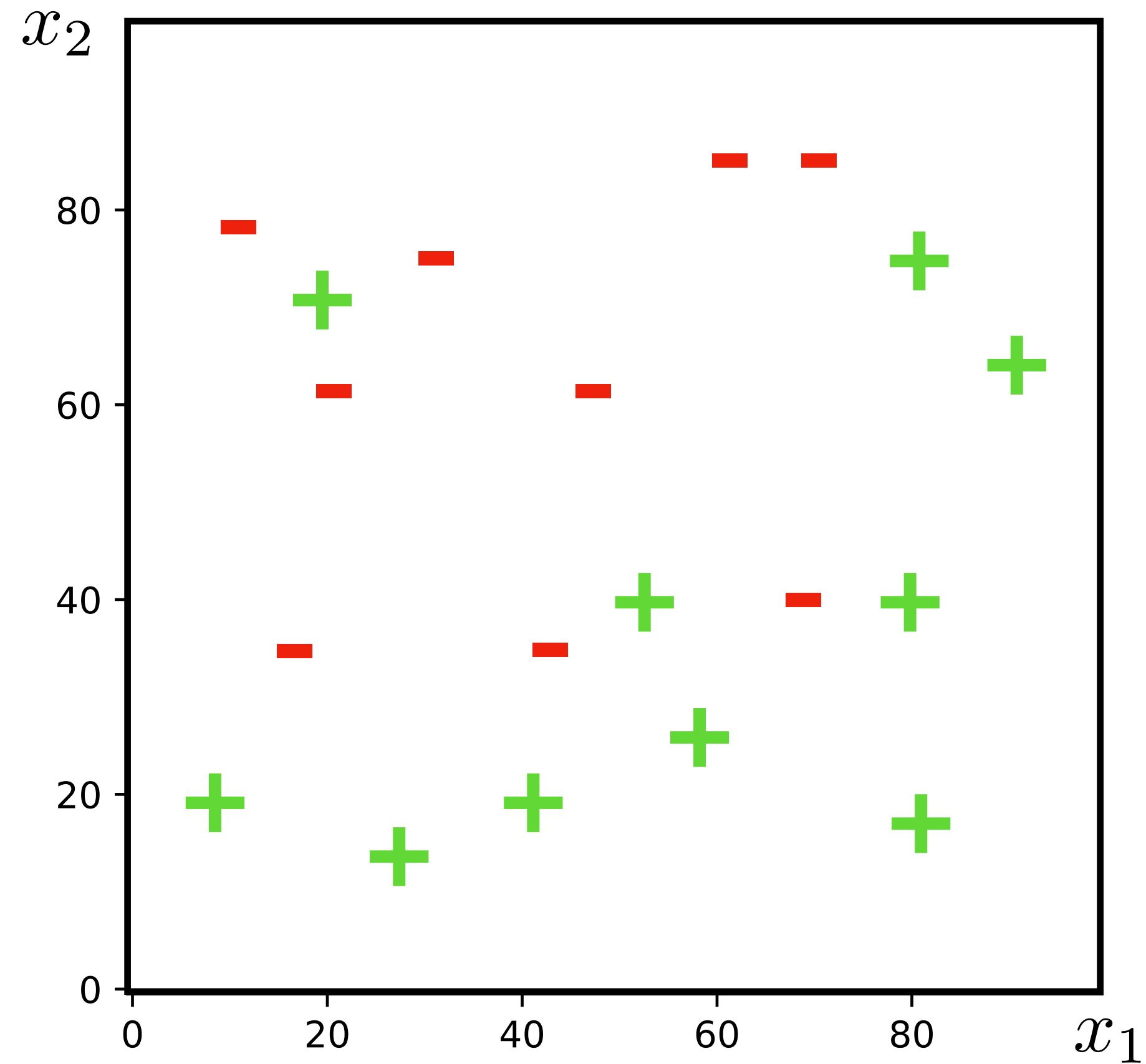
“Fish”

$$\arg \min_{f \in \mathcal{F}} \sum_{i=1}^N \mathcal{L}(f(\mathbf{x}_i), y_i)$$

How can we use a linear model for classification?

# Linear decision boundaries

Consider a **binary** classification problem.

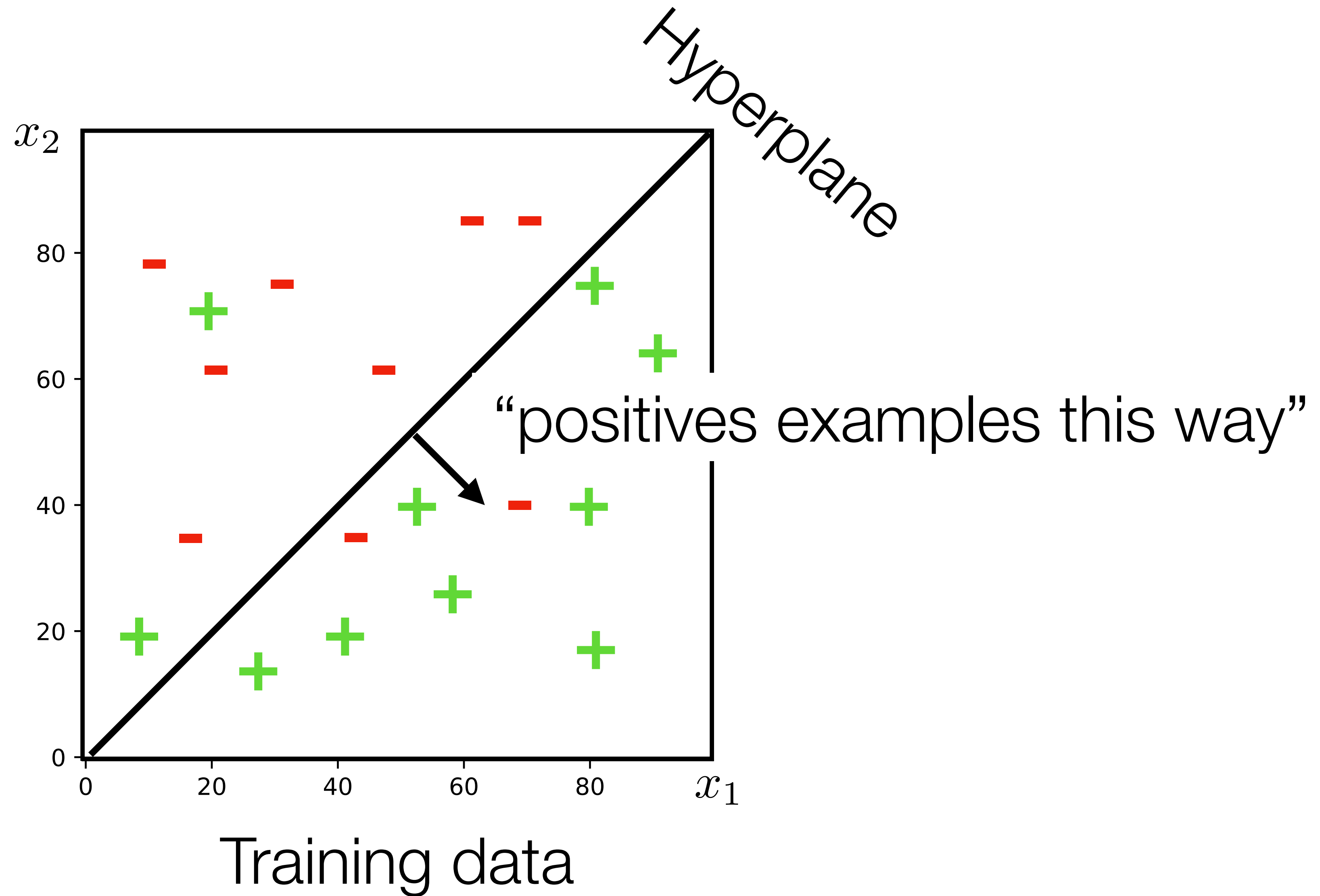


Training data

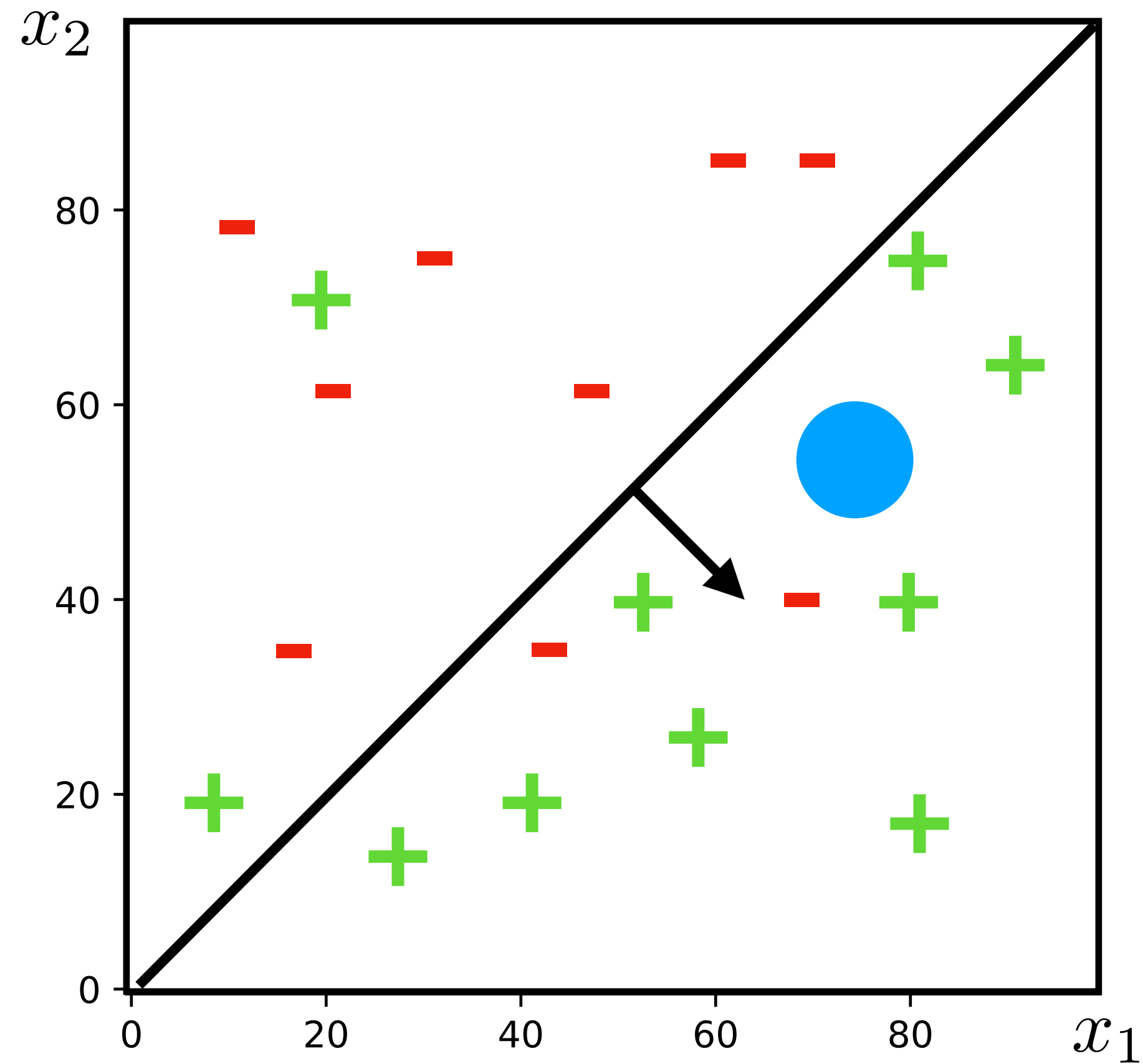


# Linear decision boundaries

Consider a **binary** classification problem.



# Linear decision boundaries



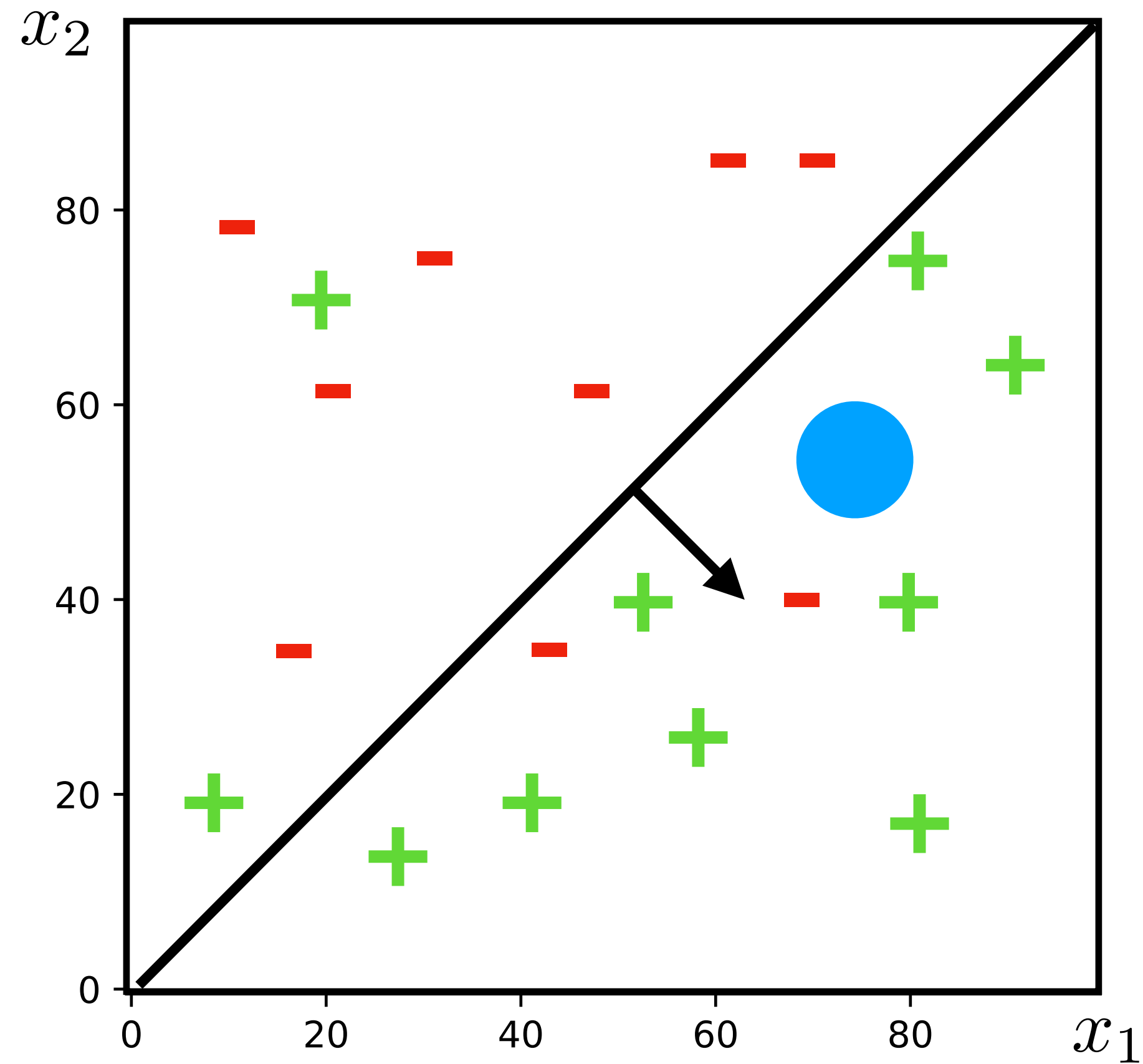
“What label is point?”

Which side of the hyperplane is  $\mathbf{x}$  on?

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

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

# Linear decision boundaries



“What label is point?”

Notational simplification:

Can get rid of  $b$  by “tacking on” a 1 to  $x$

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

$$\tilde{\mathbf{x}} = \begin{bmatrix} x \\ 1 \end{bmatrix}$$

$$\tilde{\mathbf{w}} = \begin{bmatrix} w \\ b \end{bmatrix}$$

# Multiway classification

For a  $k$ -class problem, we'll make a matrix “stacking”  $k$  hyperplanes as rows of a matrix  $W \in \mathbb{R}^{k \times d}$ :

$$W = \begin{bmatrix} \text{-----} & w_1 & \text{-----} \\ \text{-----} & w_2 & \text{-----} \\ & \dots & \\ \text{-----} & w_k & \text{-----} \end{bmatrix}$$

To classify an example: which row has the highest dot product with  $\mathbf{x}$ ?

$$z = Wx + b$$

$$g(\hat{y}) = \operatorname{argmax}_k z_k$$

# Example: handwritten digits

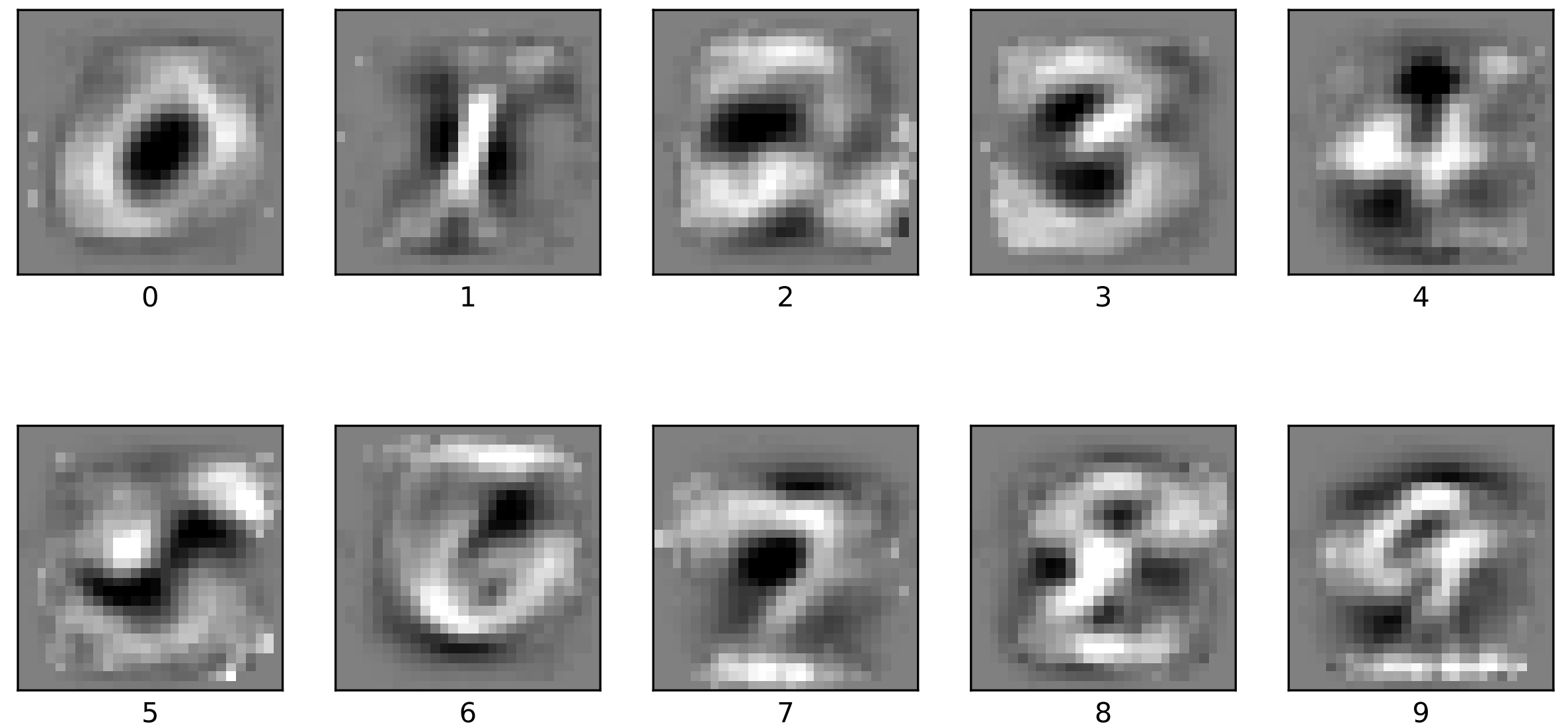
$$z = Wx + b$$

$$g(\hat{y}) = \operatorname{argmax}_k z_k$$

$$W = \begin{bmatrix} \text{---} & w_1 & \text{---} \\ \text{---} & w_2 & \text{---} \\ & \dots & \\ \text{---} & w_k & \text{---} \end{bmatrix}$$

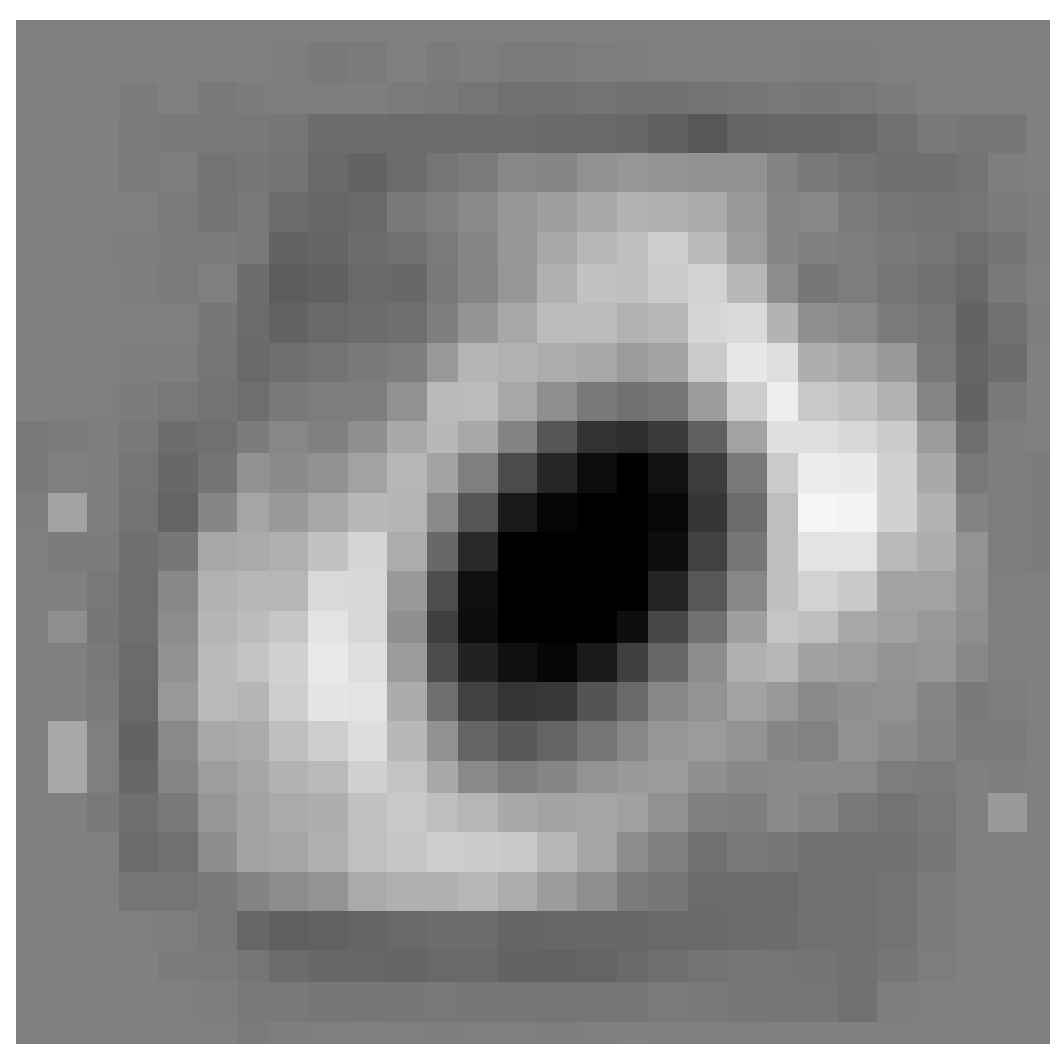


inputs

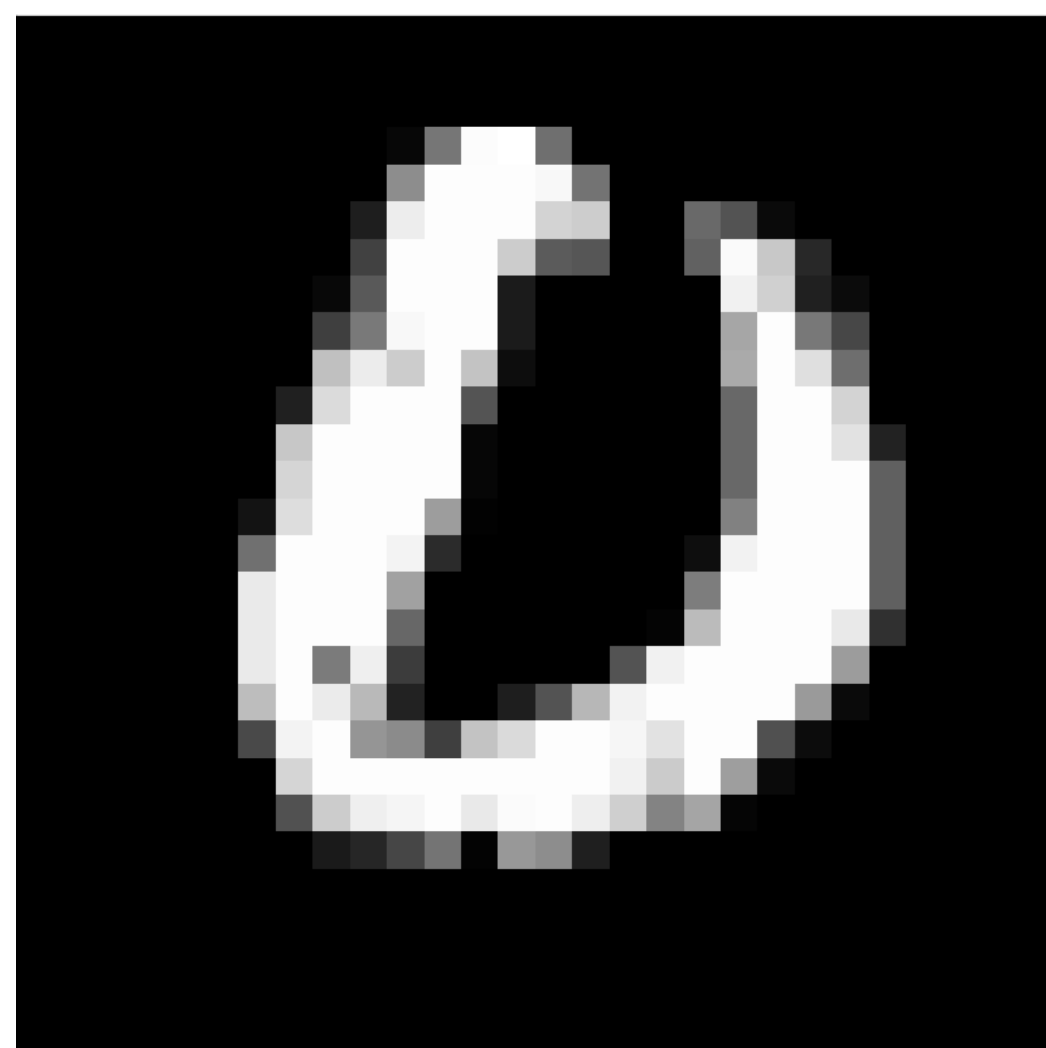
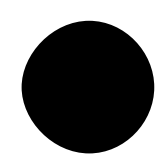


example of learned weights

# Example: handwritten digits



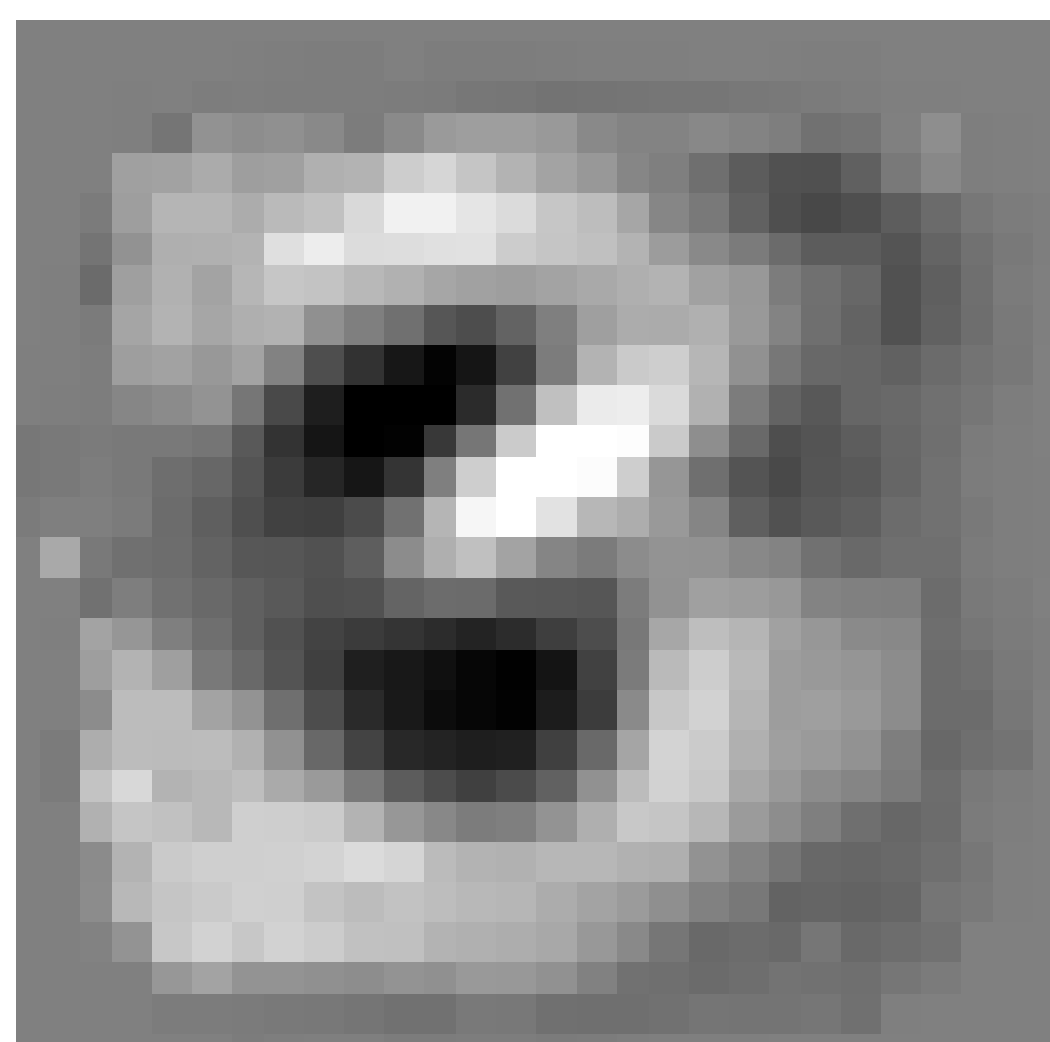
$w_0$



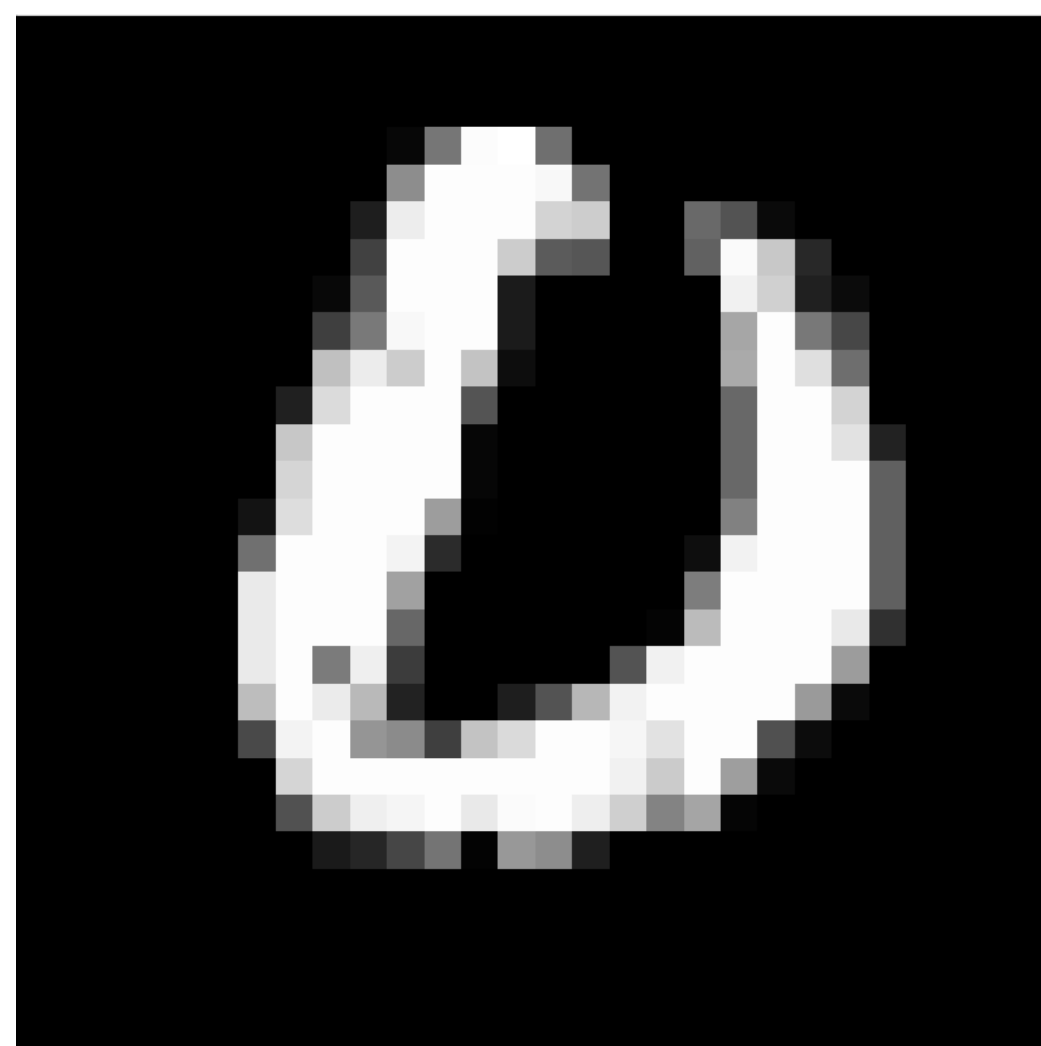
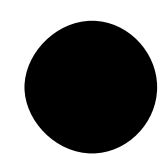
$x$

= high number

# Example: handwritten digits



$w_3$



$x$

= low number

# Converting to probabilities

We want probabilistic predictions:  $p(y_k | x)$

i.e.  $\sum_k p(y_k | x) = 1$ , and  $p(y_k | x) \geq 0$

Solution: use the **softmax** function:

$$\hat{y} = \text{softmax}(Wx + b)$$

$$p(y_k | x) = \hat{y}_k$$

$$\text{softmax}(z)_k = \frac{\exp(z_k)}{\sum_i \exp(z_i)}$$

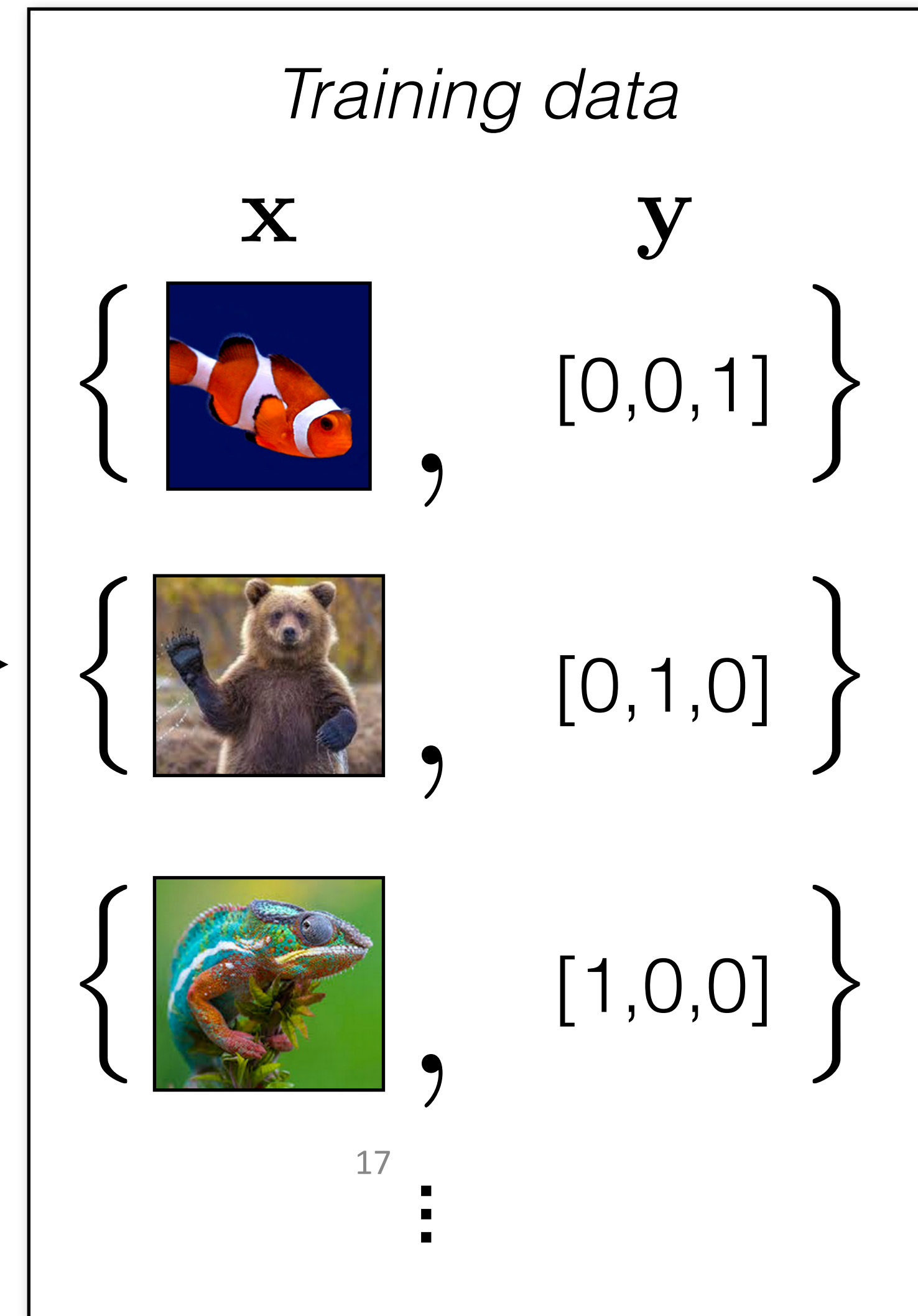
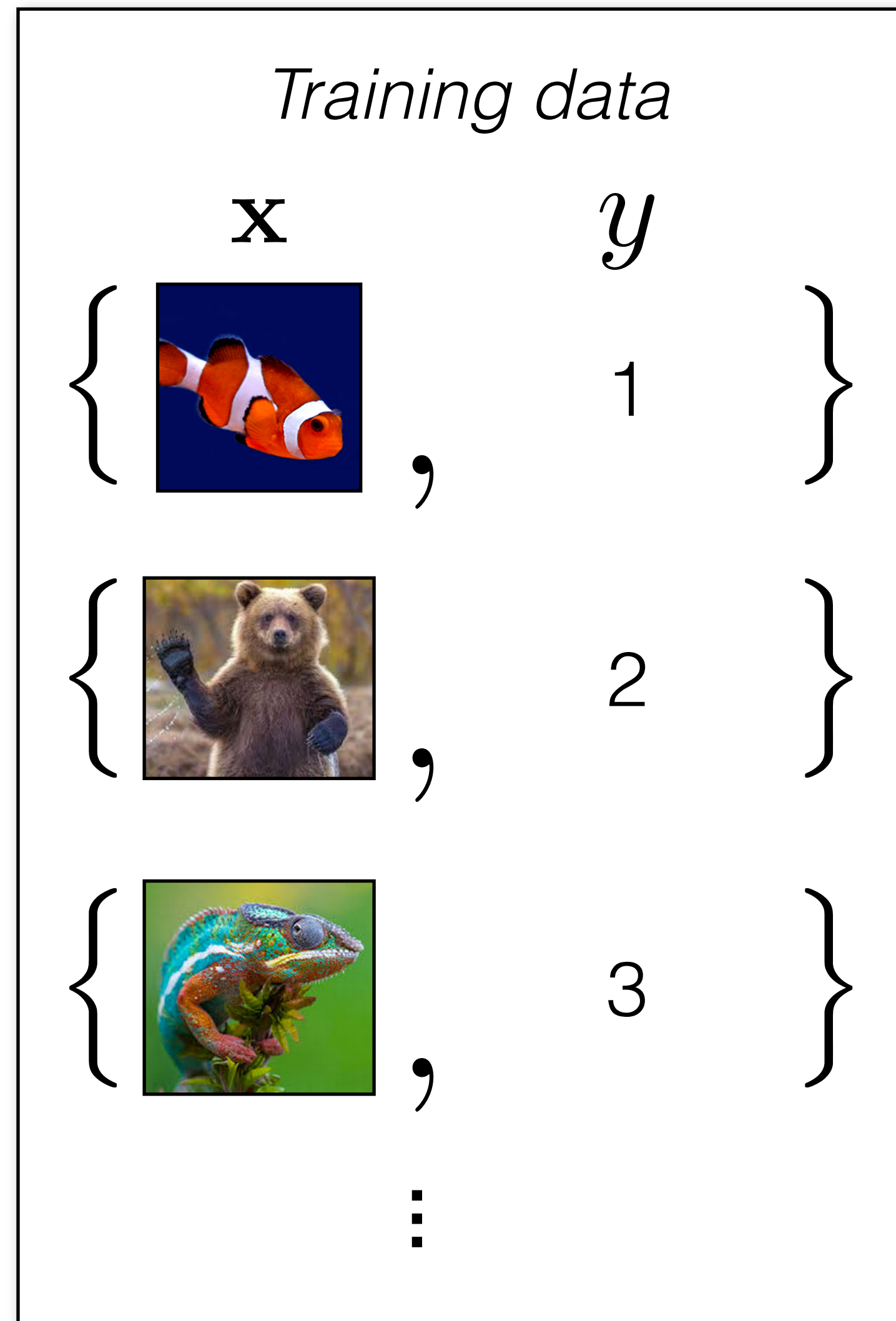
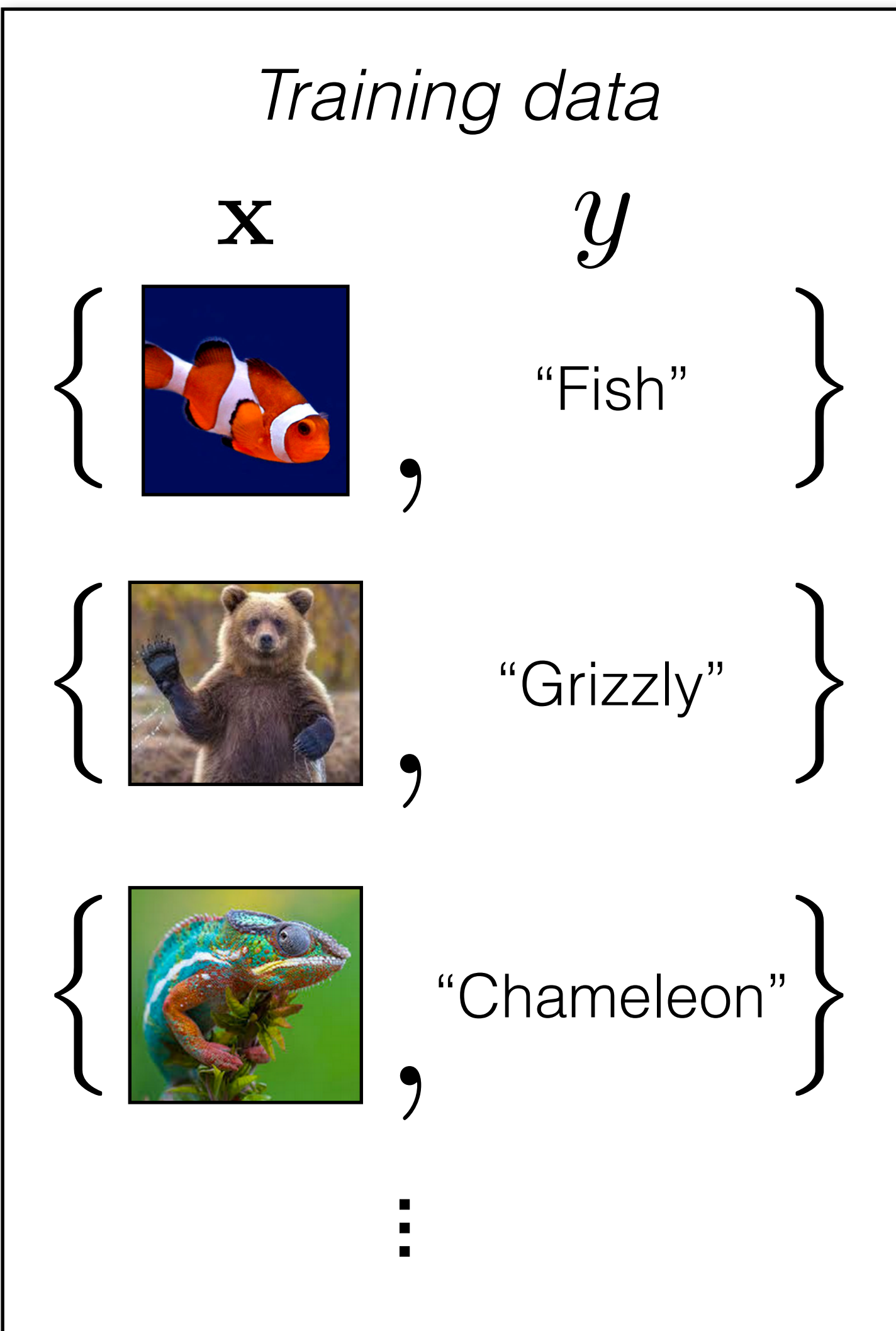
non-negative

sums-to-1



# How do we represent class labels?

## One-hot vector



# Loss function

**0-1 loss:** the loss we'd ideally want

$$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = 1 - \mathbb{1}(\hat{\mathbf{y}}, \mathbf{y}) \leftarrow \text{number of misclassifications}$$

**Square loss:** predict 1 for true class, 0 for others)

$$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = \sum_{k=1}^K (y_k - \hat{y}_k)^2$$

**Cross entropy:** a good surrogate that we'll be able to optimize:

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

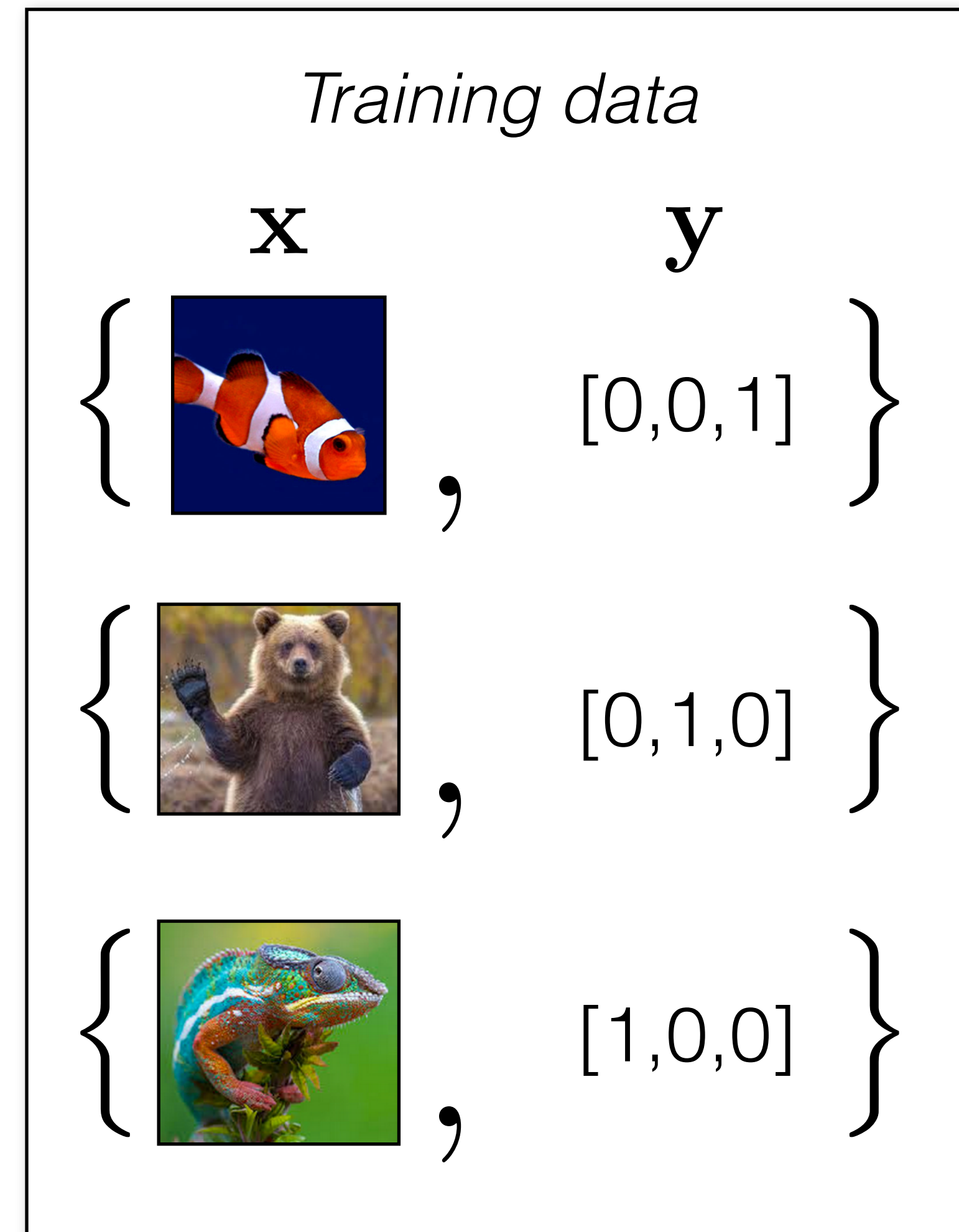
# Loss function

Simpler than it looks:

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

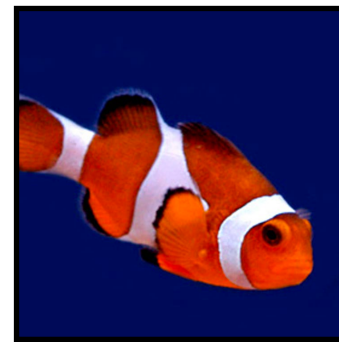
Therefore:

## One-hot vectors

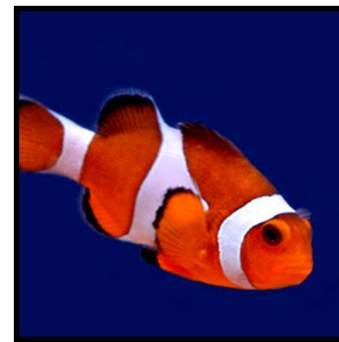


Ground truth label  $y$

$x$

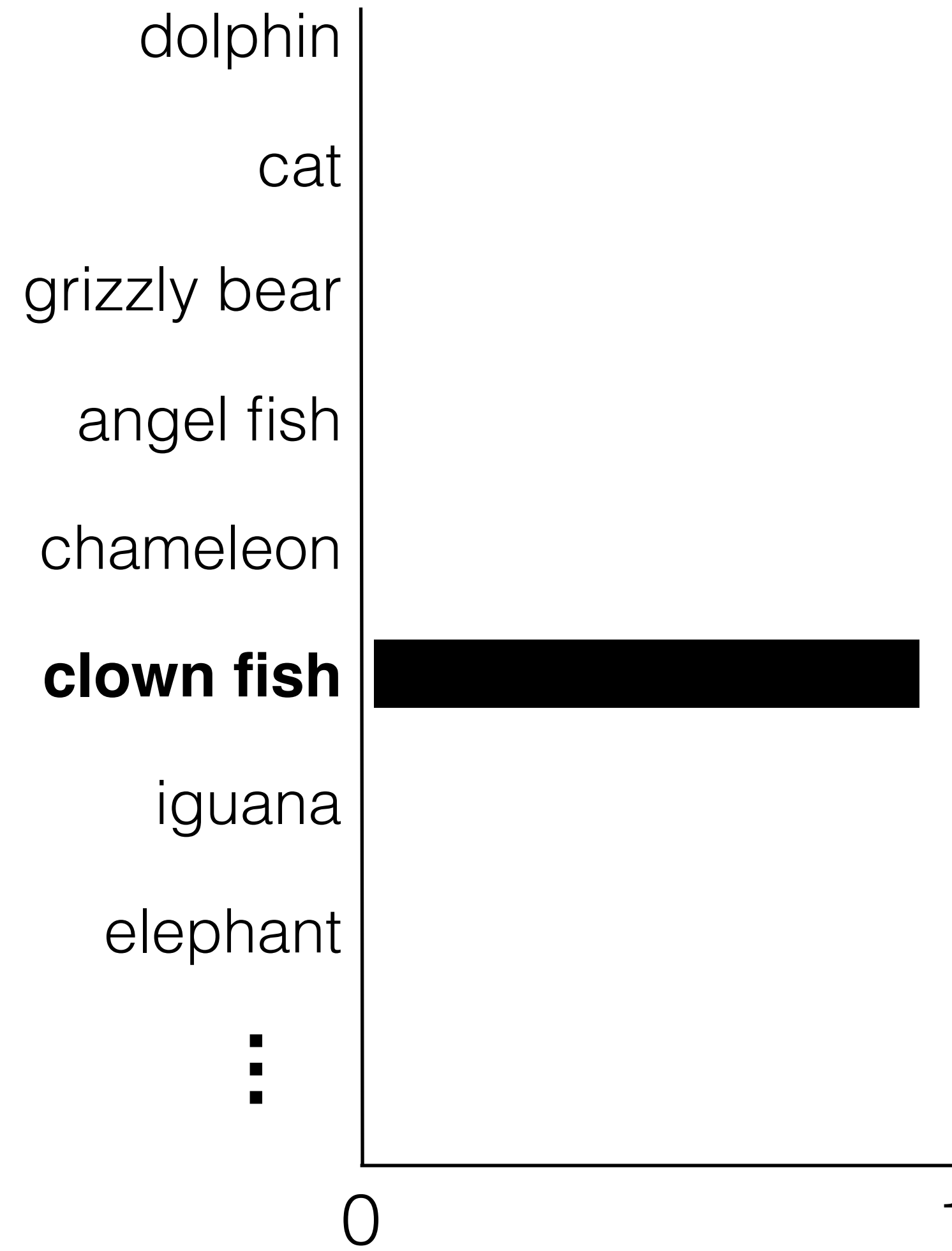


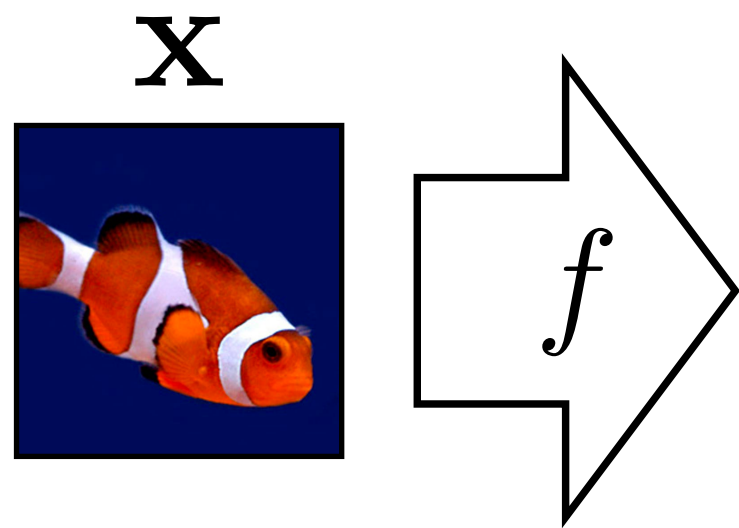
$[0,0,0,0,0,1,0,0,\dots]$



**X**

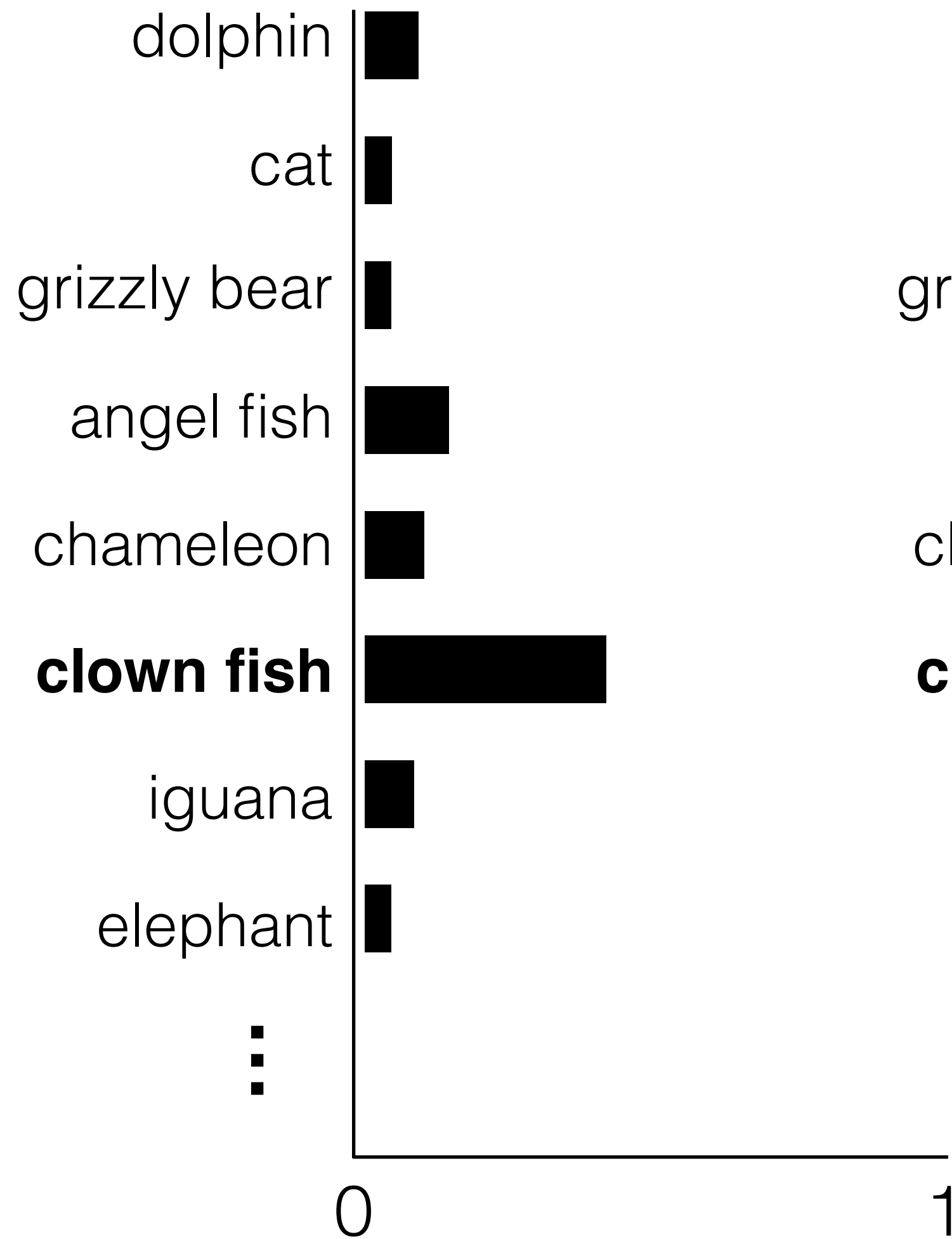
Ground truth label **y**



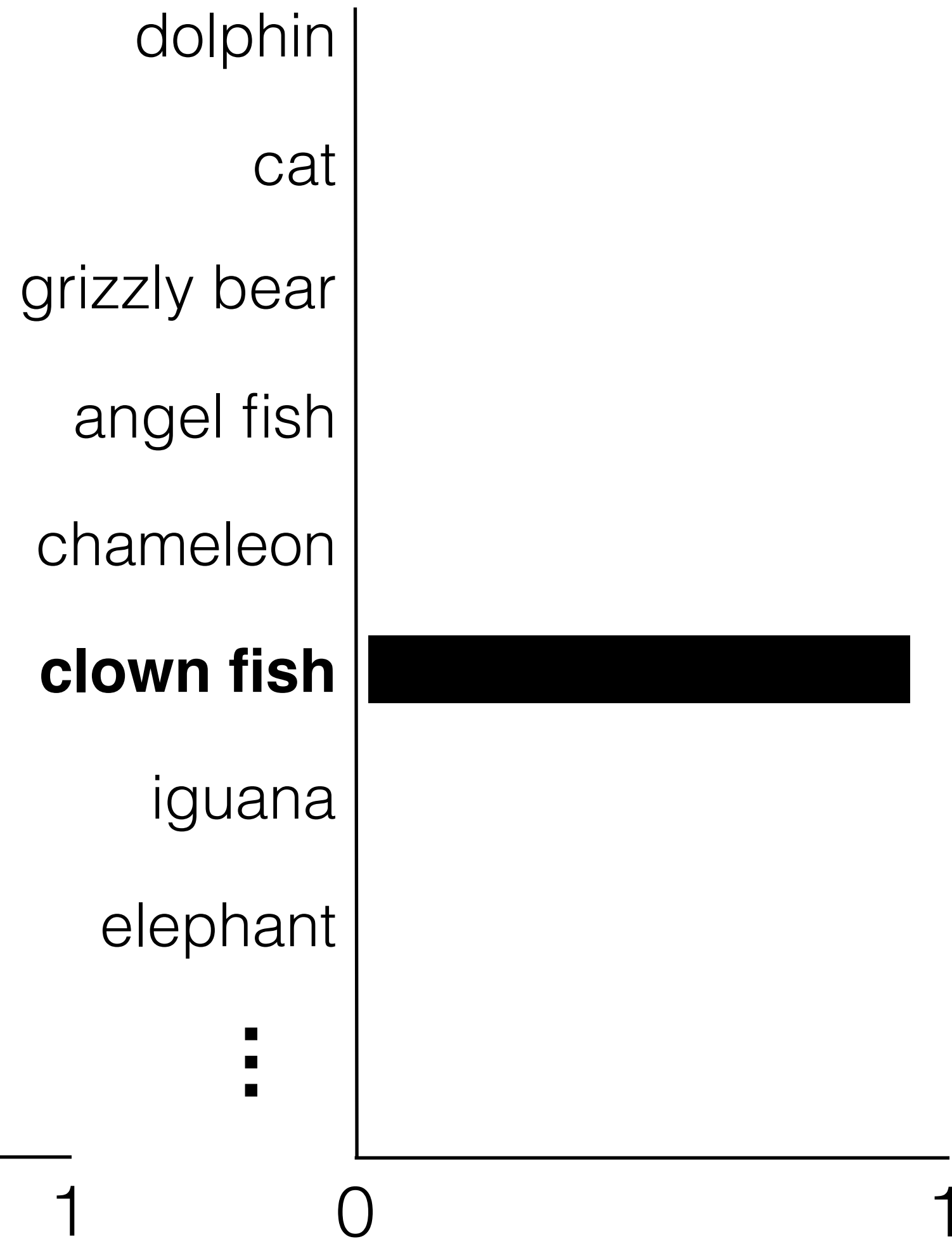


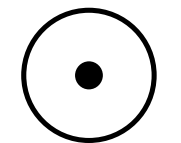
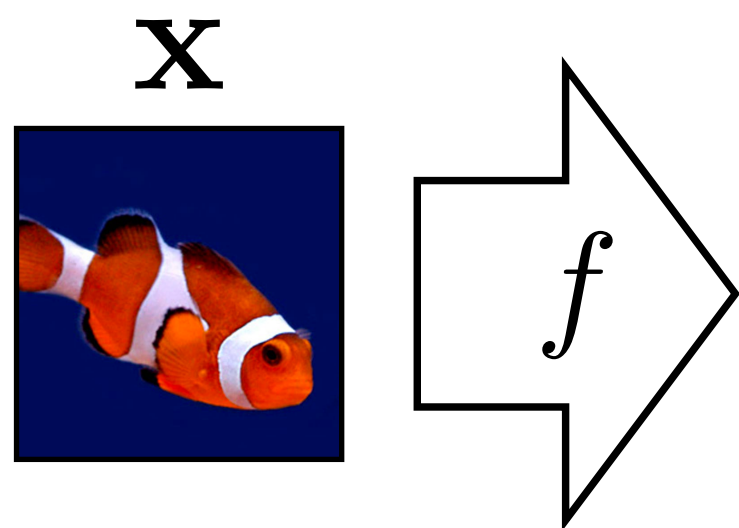
Prediction  $\hat{y}$

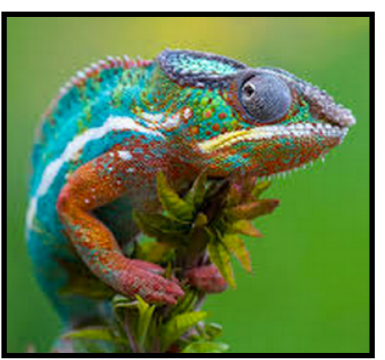
$$f_{\theta} : X \rightarrow \mathbb{R}^K$$



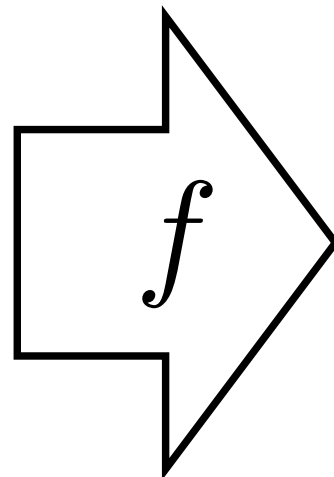
Ground truth label  $y$





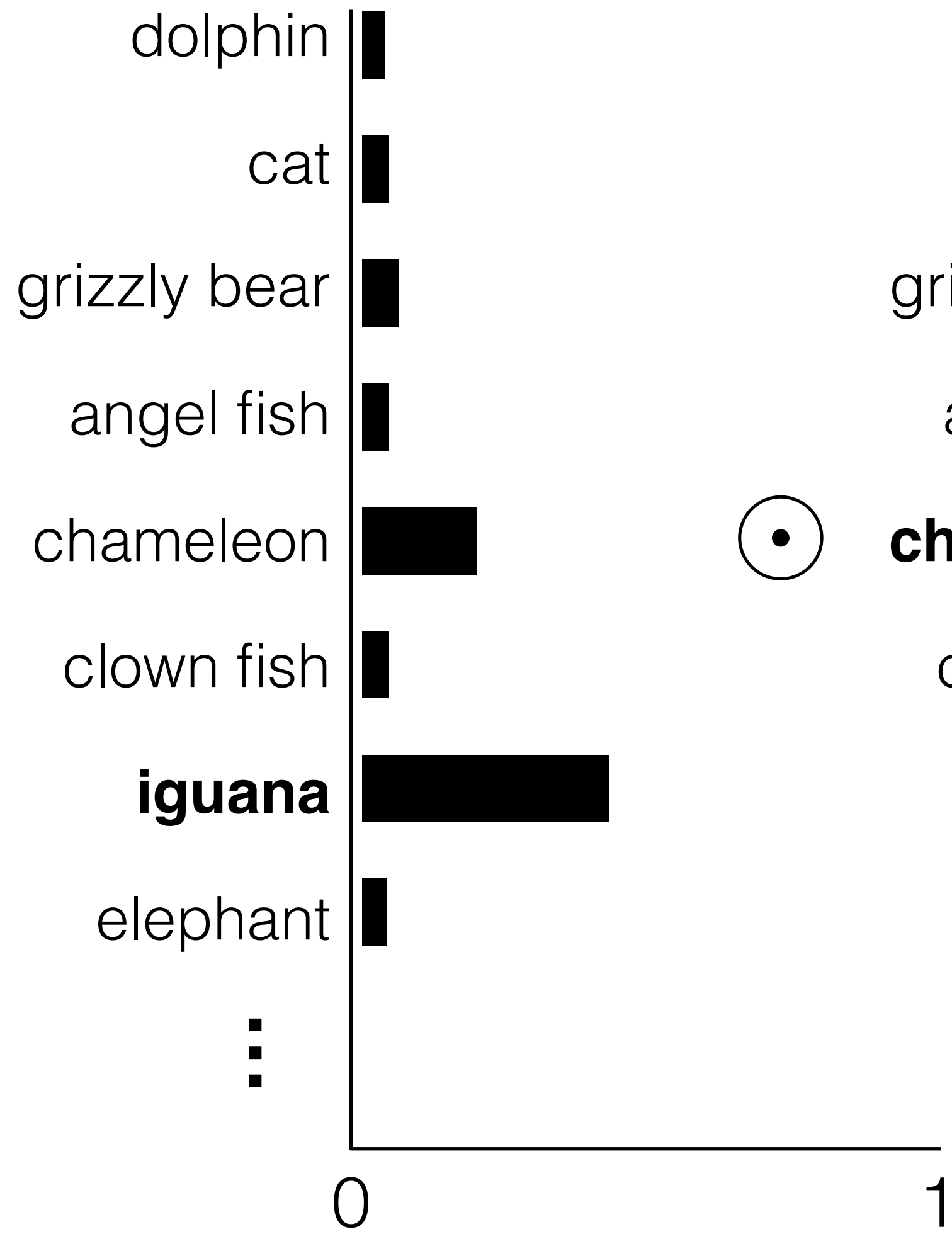


**X**

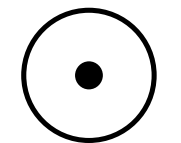


Prediction  $\hat{y}$

$$f_{\theta} : X \rightarrow \mathbb{R}^K$$

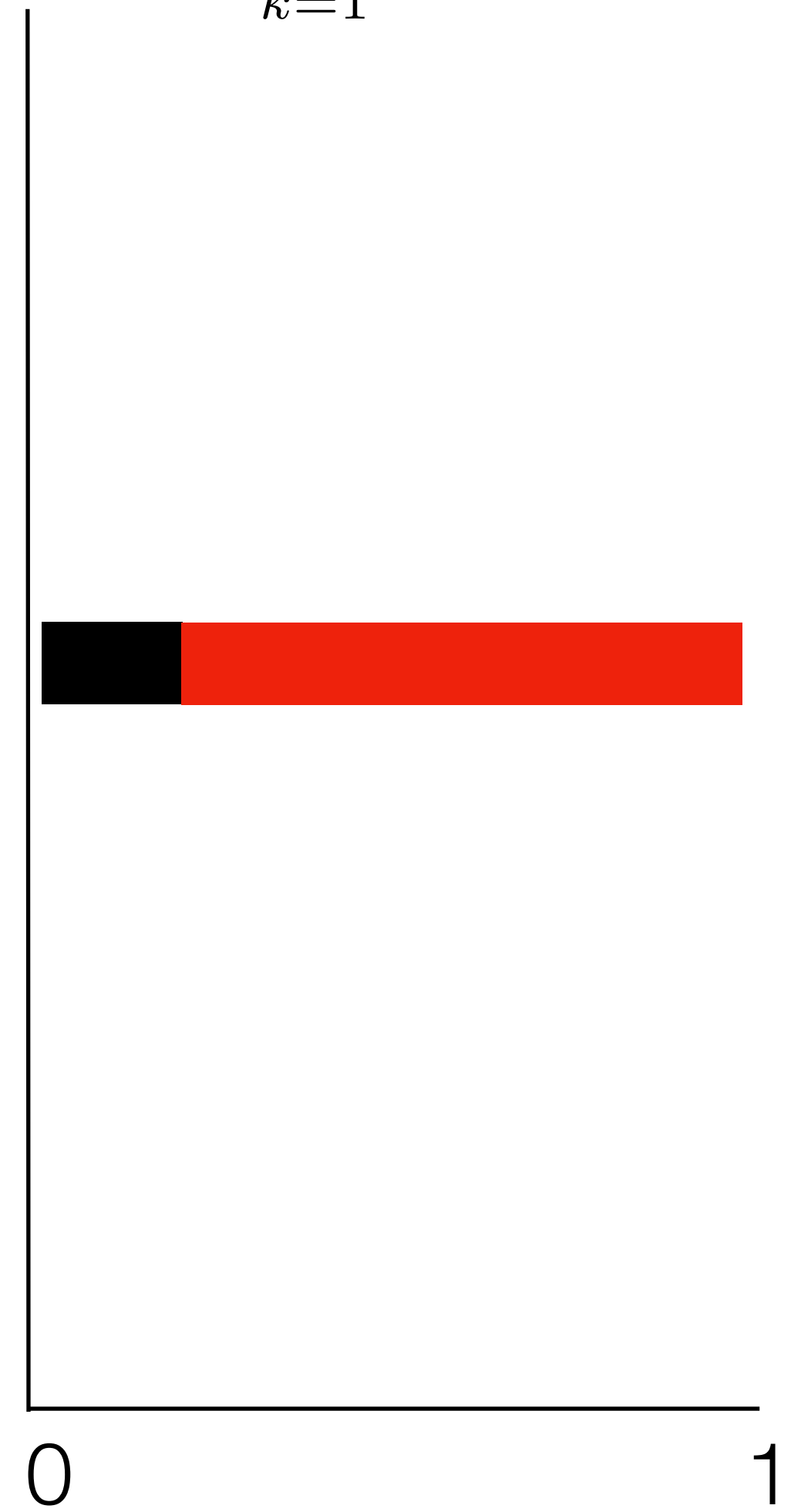


Ground truth label  $y$



Loss

$$-\sum_{k=1}^K y_k \log \hat{y}_k$$





# Hinge loss

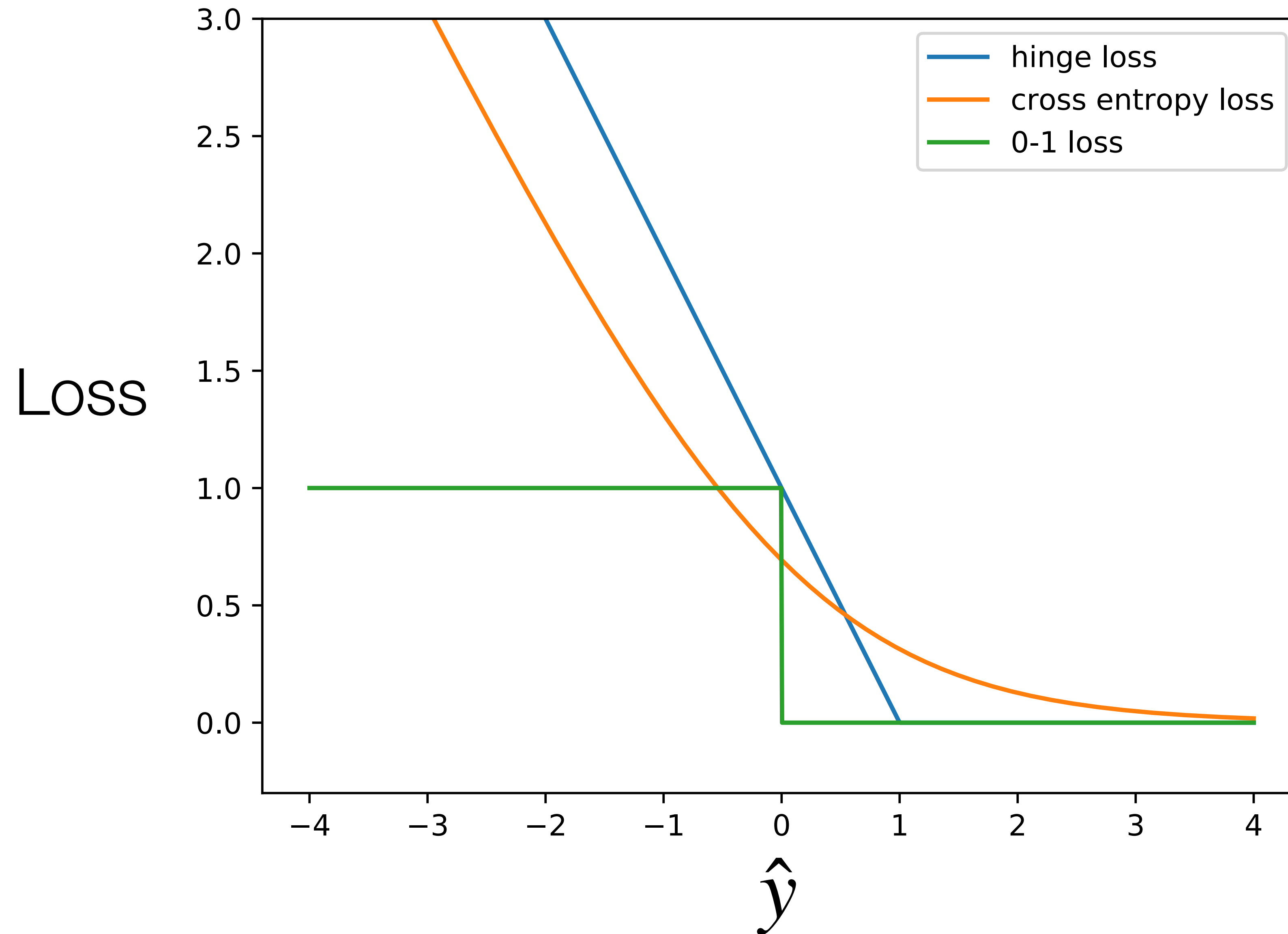
One more you might see: **hinge loss**

$$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = \max(0, 1 - \mathbf{y}\hat{\mathbf{y}}, 0) \quad \leftarrow \text{upper bound on 0-1 loss (true class } \mathbf{y} \text{ is } \{+1, -1\} \text{ instead of } \{0, 1\})$$

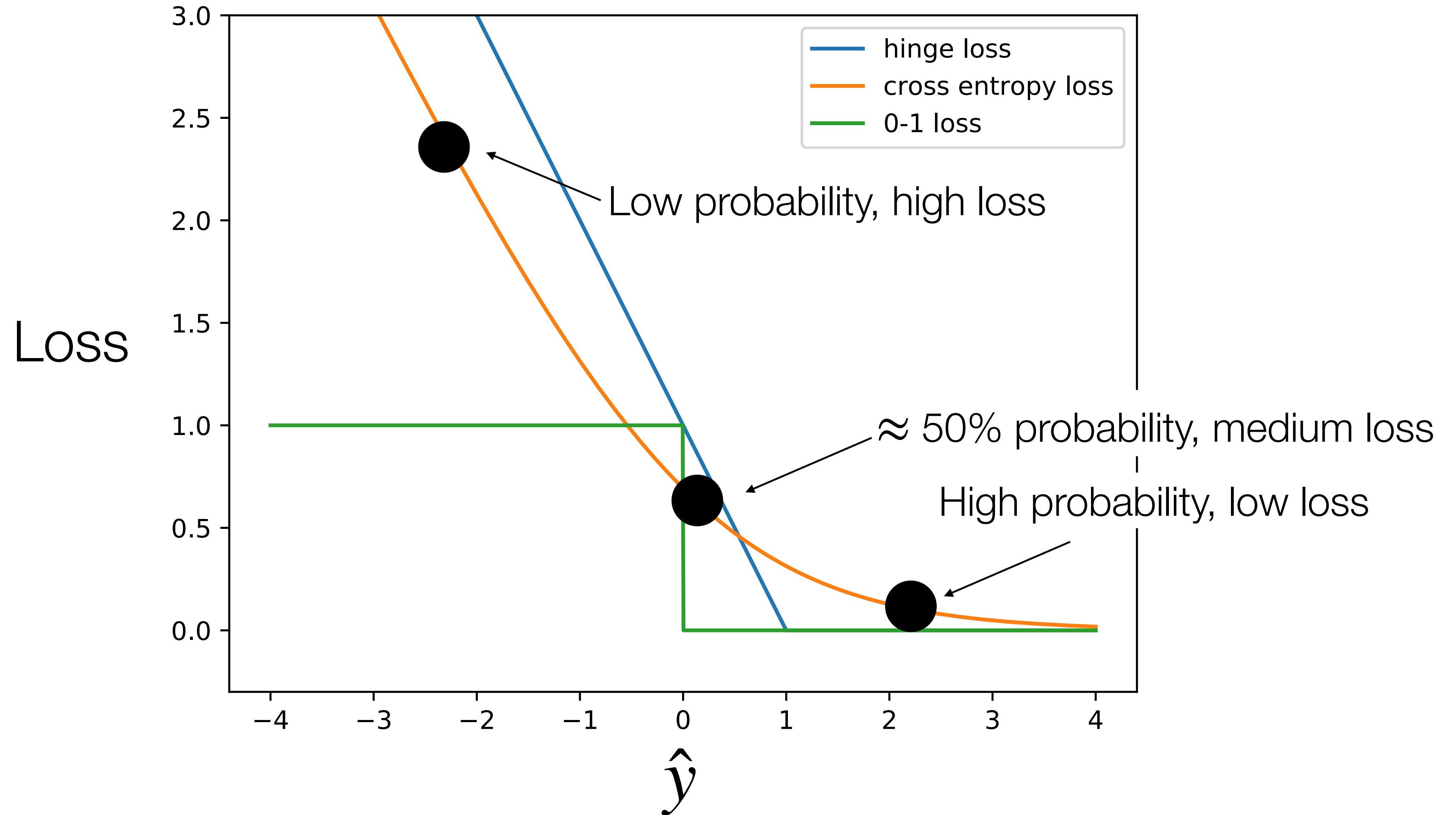
A linear classifier that uses a hinge loss is called a **support vector machine (SVM)**.

It has some very nice properties (they can be made nonlinear with “kernels”). Generally performs similar to logistic regression.

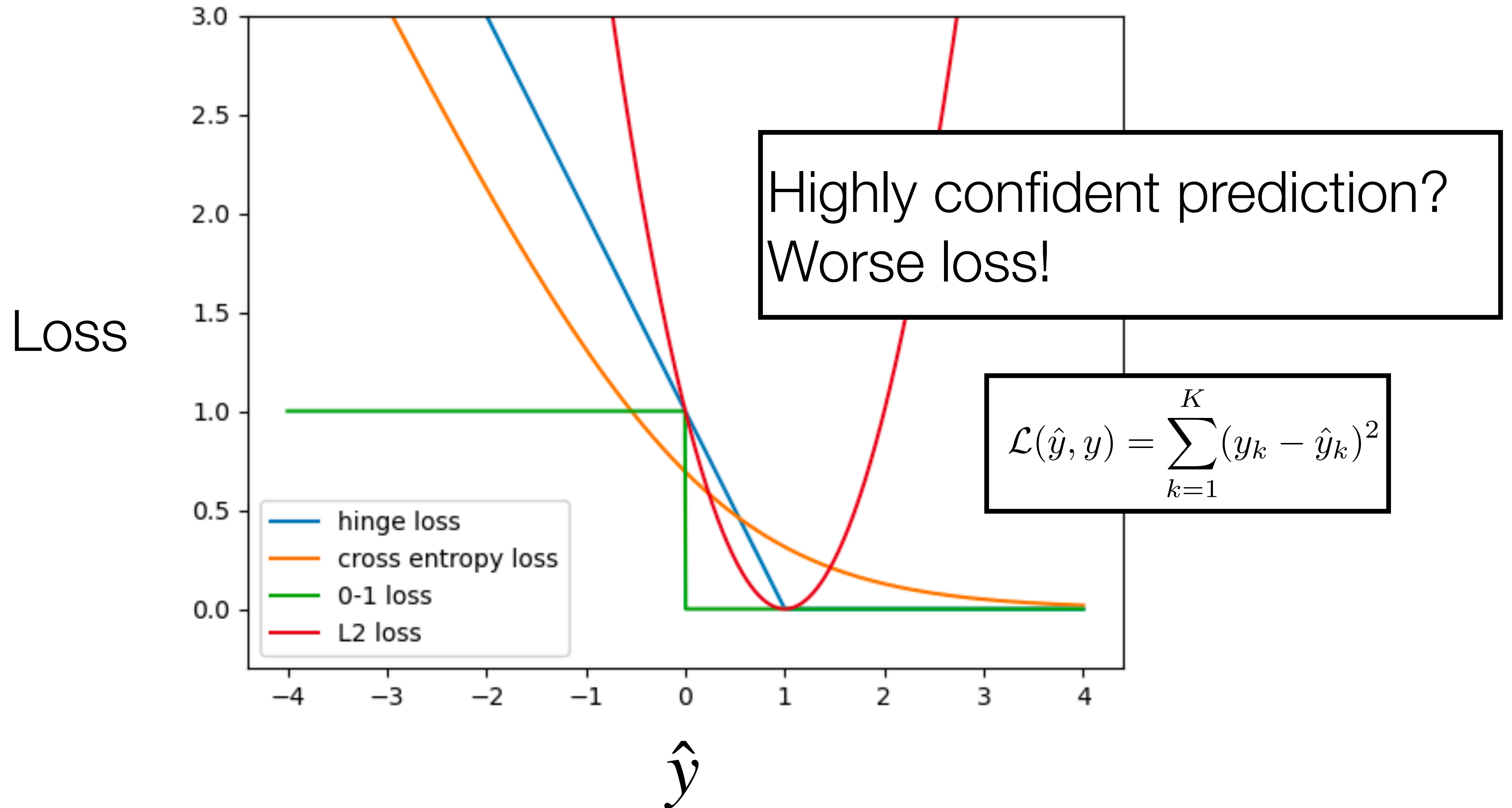
# Loss functions



# Loss functions



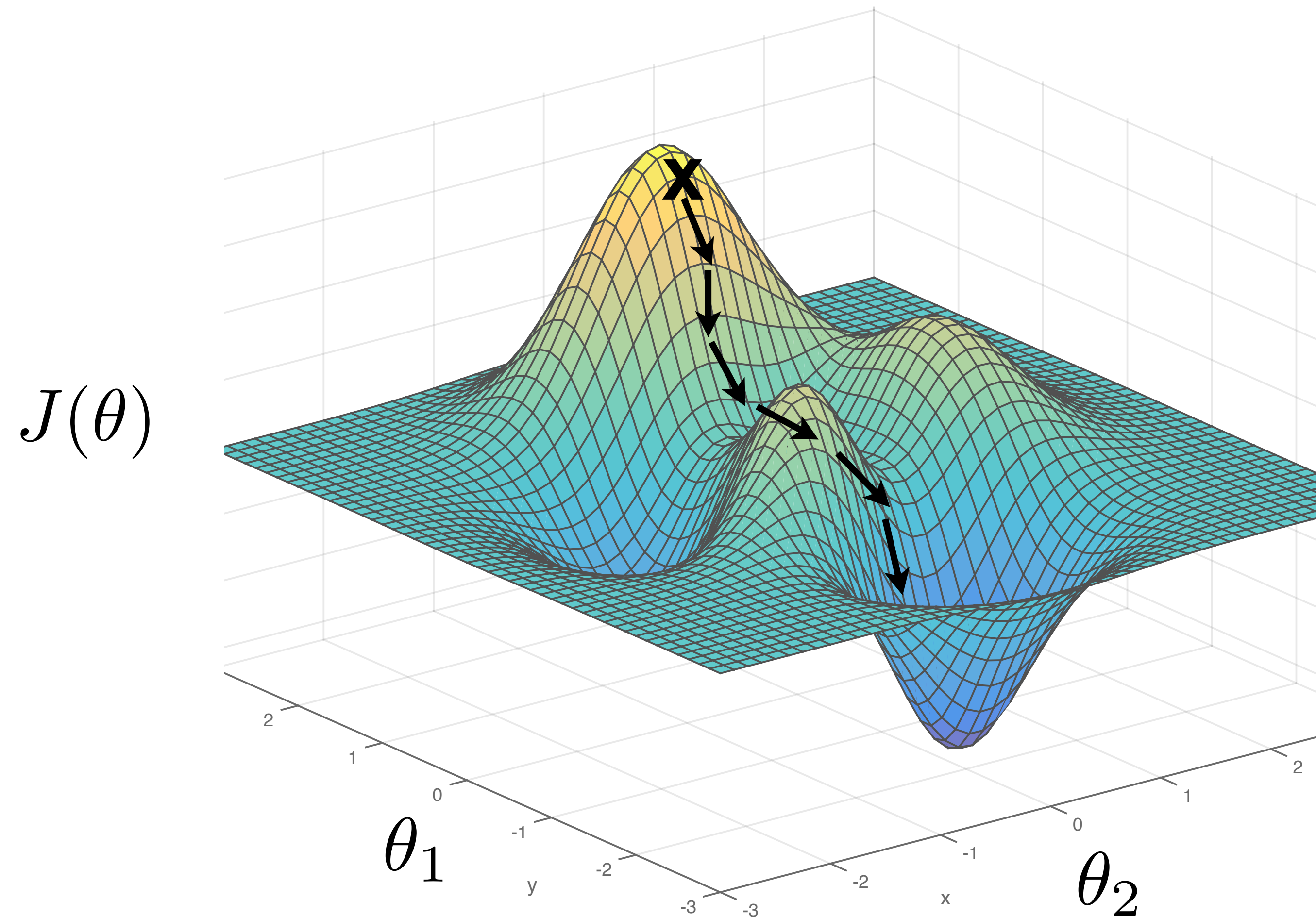
# Why not squared loss?



# How do we learn a classifier?

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

# Gradient descent



Take direction  
of “steepest  
descent”

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

# Gradient descent

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

One iteration of gradient descent:

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

(learning rate)

# Gradient descent

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

What's this again?

1. Gradient of the loss w.r.t. the classifier's parameters
2. Direction of steepest descent
3. "Local" linear approximation to the function

For a refresher on gradients and partial derivatives:

<https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives#partial-derivatives>



# Gradient descent

Take gradient at current  $\theta$

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

$$\nabla_{\theta} J(\theta) = \begin{bmatrix} \frac{\partial J}{\partial \theta_1} \\ \frac{\partial J}{\partial \theta_2} \\ \dots \\ \frac{\partial J}{\partial \theta_d} \end{bmatrix}$$

Vector of partial derivatives for loss

# Estimating the gradient

Idea #1: finite differences  $\frac{f(x + \epsilon) - f(x)}{\epsilon}$

$$\nabla J(\theta) = \begin{bmatrix} \frac{\partial J}{\partial \theta_1} \\ \frac{\partial J}{\partial \theta_2} \\ \dots \\ \frac{\partial J}{\partial \theta_d} \end{bmatrix} \approx \frac{J(\theta + \epsilon e_2) - J(\theta - \epsilon e_2)}{2\epsilon}$$

where  $e_2$  is vector of zeros except for a 1 in 2nd component

# Estimating the gradient

Idea #1: finite differences  $\frac{f(x + \epsilon) - f(x)}{\epsilon}$

$$\nabla J(\theta) = \begin{bmatrix} \frac{\partial J}{\partial \theta_1} \\ \frac{\partial J}{\partial \theta_2} \\ \dots \\ \frac{\partial J}{\partial \theta_d} \end{bmatrix}$$

- Easy to compute but **slow**.
- Inaccurate in some cases, unless careful
- Useful for **checking** other methods

# Estimating the gradient

Idea #2: compute it analytically using calculus.

**Simple example:** binary labels, squared loss, and regularization on  $\theta$

$$J(\theta) = \lambda \|\theta\|^2 + \frac{1}{N} \sum_{i=1}^N (y_i - \theta^\top x_i)^2$$

$\nabla_{\theta}$

$$\nabla_{\theta} J(\theta) = 2\lambda\theta - \frac{1}{N} \sum_{i=1}^N 2(y_i - \theta^\top x_i)x_i$$

# Analyzing the gradient descent update

Recall update rule:  $\theta^{t+1} = \theta^t - \eta_t \nabla_{\theta} J(\theta) \Big|_{\theta=\theta^t}$

How does this change  $\theta$  ?

$$-\nabla_{\theta} J(\theta) = -2\lambda\theta - \frac{1}{N} \sum_{i=1}^N 2(y_i - \theta^{\top} x_i) x_i$$

“Decay” toward 0      Scalar  $\alpha$

# Analyzing the gradient descent update

What happens at each example?

$$-\nabla_{\theta} J(\theta) = -2\lambda\theta - \frac{1}{N} \sum_{i=1}^N 2(y_i - \theta^{\top} x_i) x_i$$

Scalar  $\alpha$   
↓

If  $\theta^{\top} x_i < y$  (too low): then  $\theta_{t+1} = \theta + \alpha x_i$  for some  $\alpha > 0$

**Dot product before:**  $\theta^{\top} x_i$

**Dot product after:**  $(\theta + \alpha x_i)^{\top} x_i = \theta^{\top} x_i + \alpha x_i^{\top} x_i$

# Computational issues

# Batch gradient descent

Loss function:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^N L(x_i, y_i, \theta)$$

Its gradient is the **sum of gradients** for each example:

$$\nabla J(\theta) = \frac{1}{N} \sum_{i=1}^N \nabla L(x_i, y_i, \theta)$$

Problem: requires iterating over every training example each gradient step!

Can we speed this up?



# Stochastic gradient descent

This is just an **average**:

$$\nabla J(\theta) = \frac{1}{N} \sum_{i=1}^N \nabla L(x_i, y_i, \theta)$$

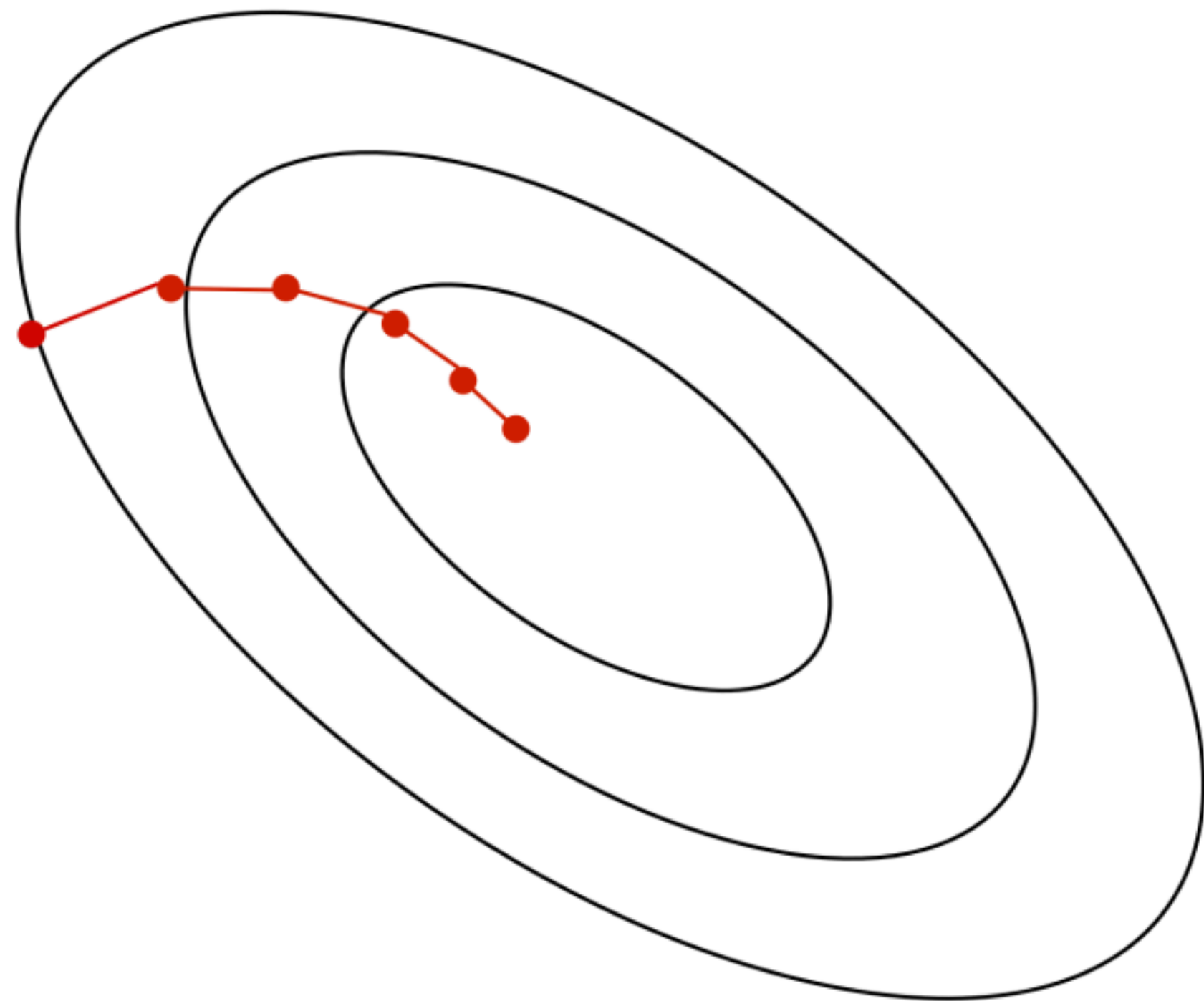
We know from statistics that we can estimate the average of a full “population” from a **sample**.

$$\nabla J(\theta) \approx \frac{1}{|B|} \sum_{i \in B} \nabla L(x_i, y_i, \theta)$$

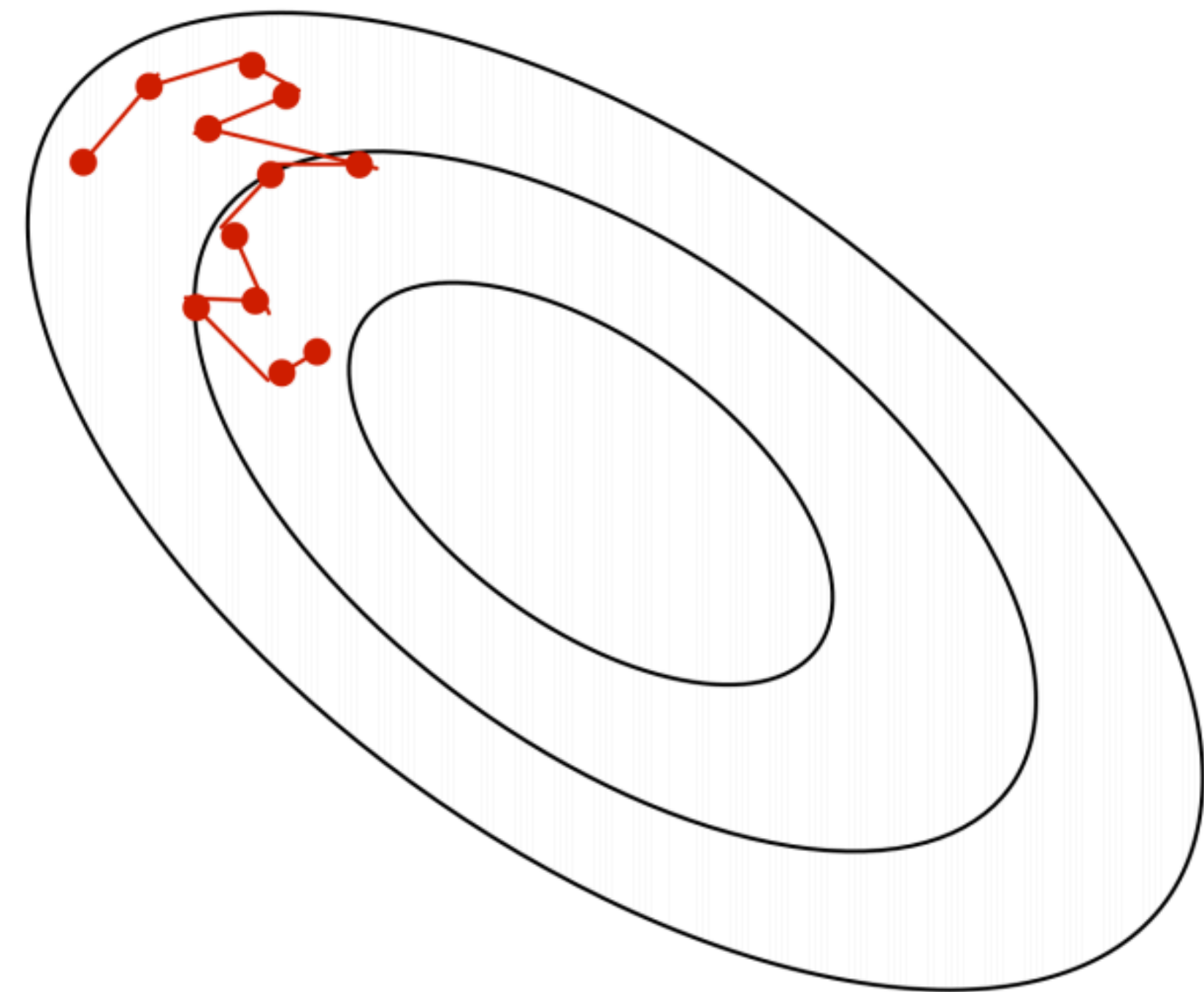
where  $B$  is a **minibatch**: a random subset of examples.

This is called **stochastic gradient descent (SGD)**.

# Stochastic gradient descent



Batch gradient descent



Stochastic gradient descent

# Stochastic gradient descent



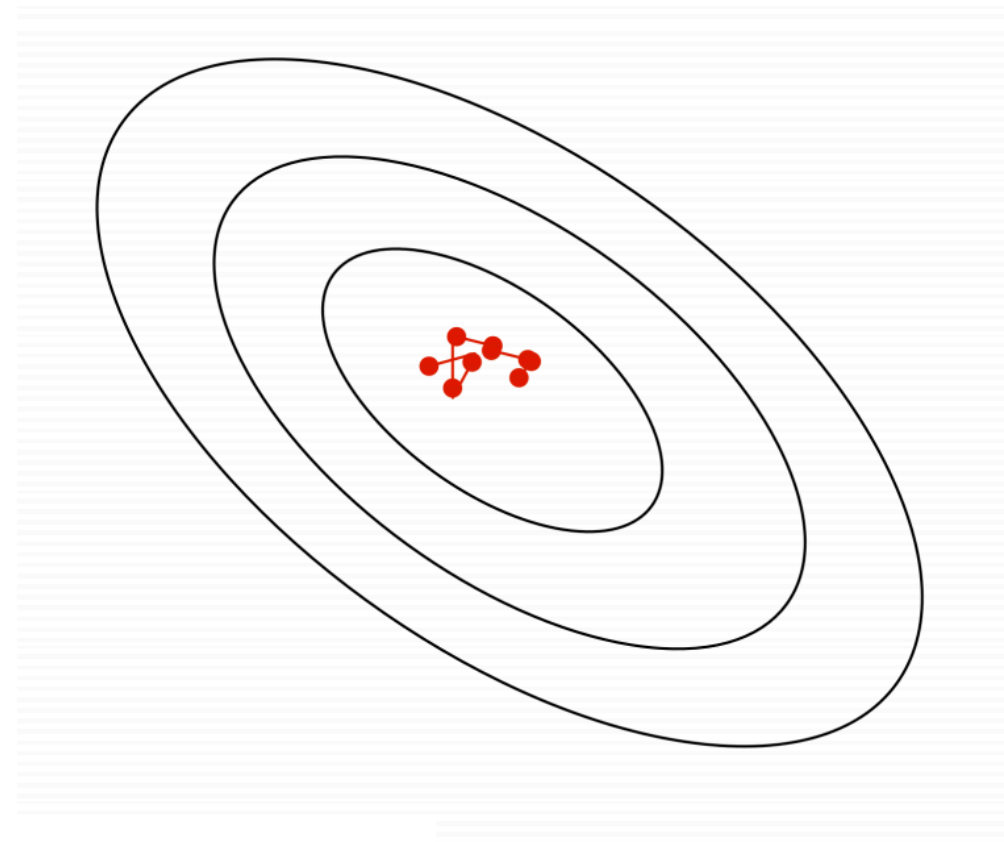
**ML Hipster** @ML\_Hipster · Jul 20, 2012

"Oh sure, going in that direction will totally minimize the objective function" —Sarcastic Gradient Descent.

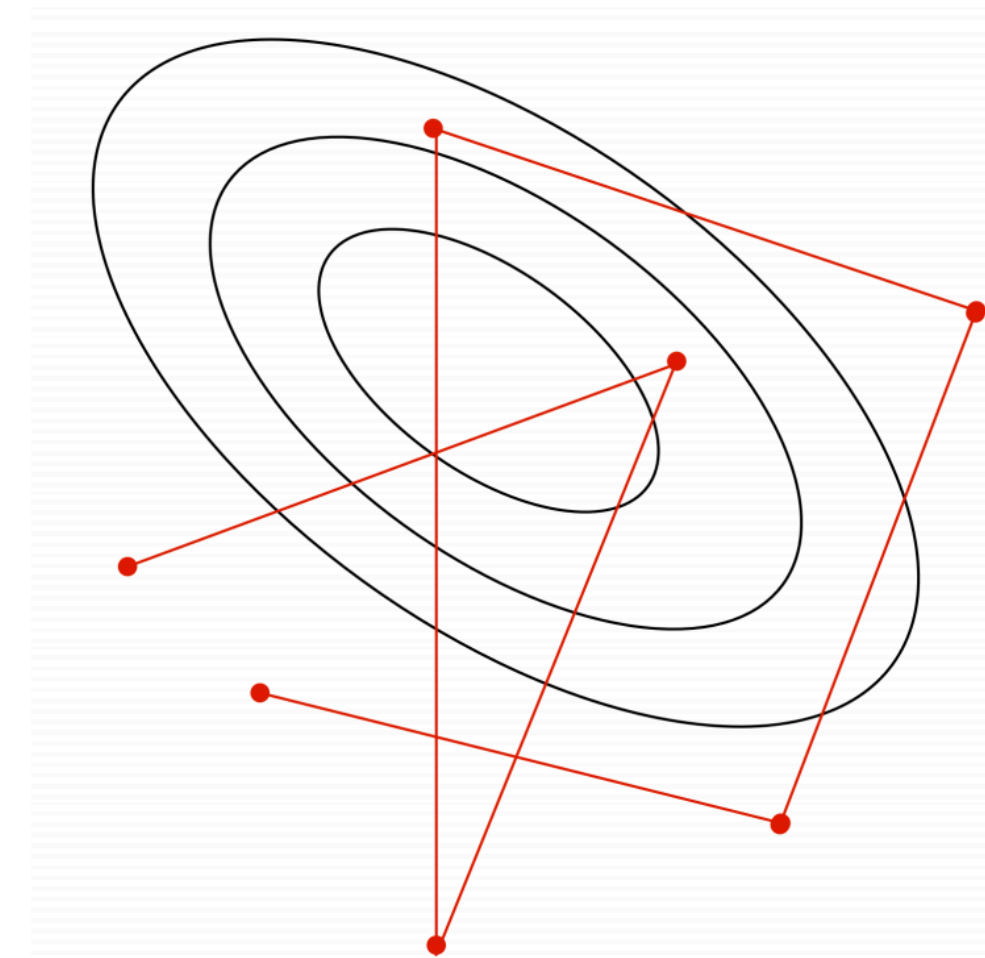


# Learning rate

- Sensitive to the learning rate:  $\theta^{t+1} = \theta^t - \eta_t \nabla_{\theta} J(\theta) \Big|_{\theta=\theta^t}$



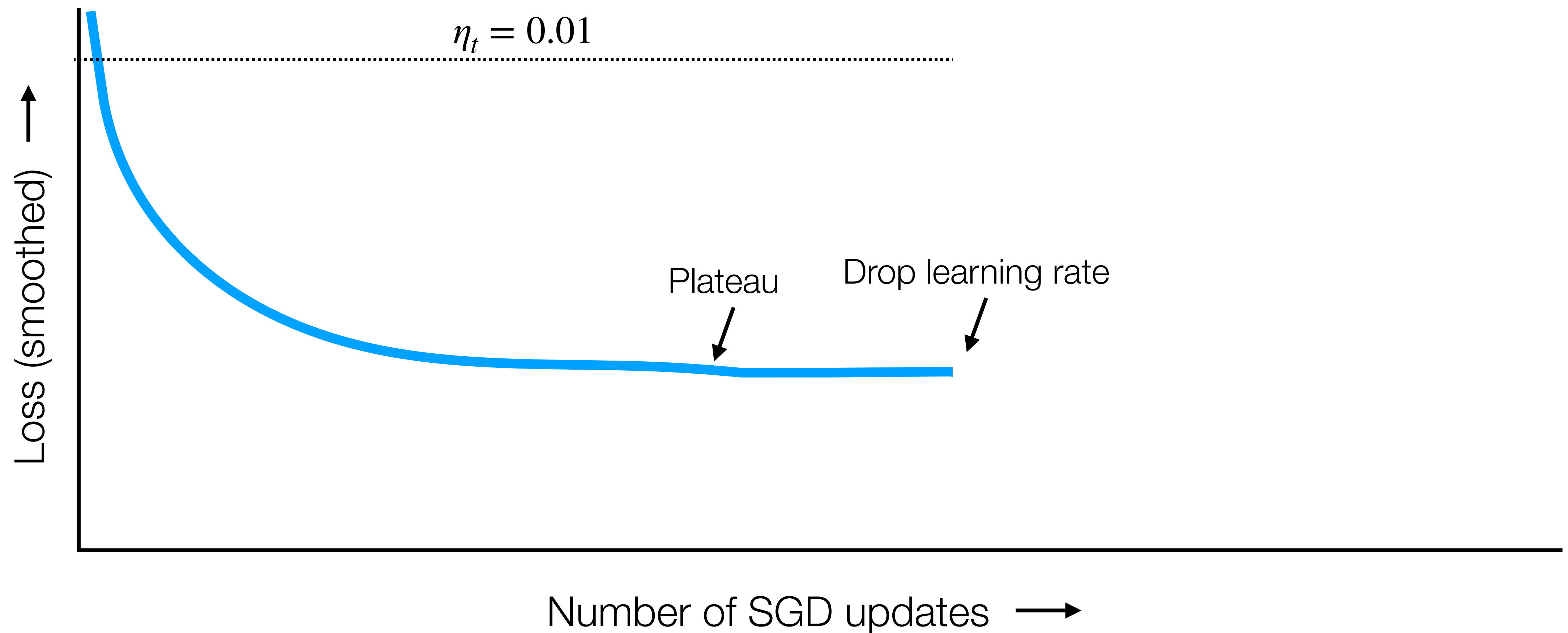
Small learning rate



Large learning rate

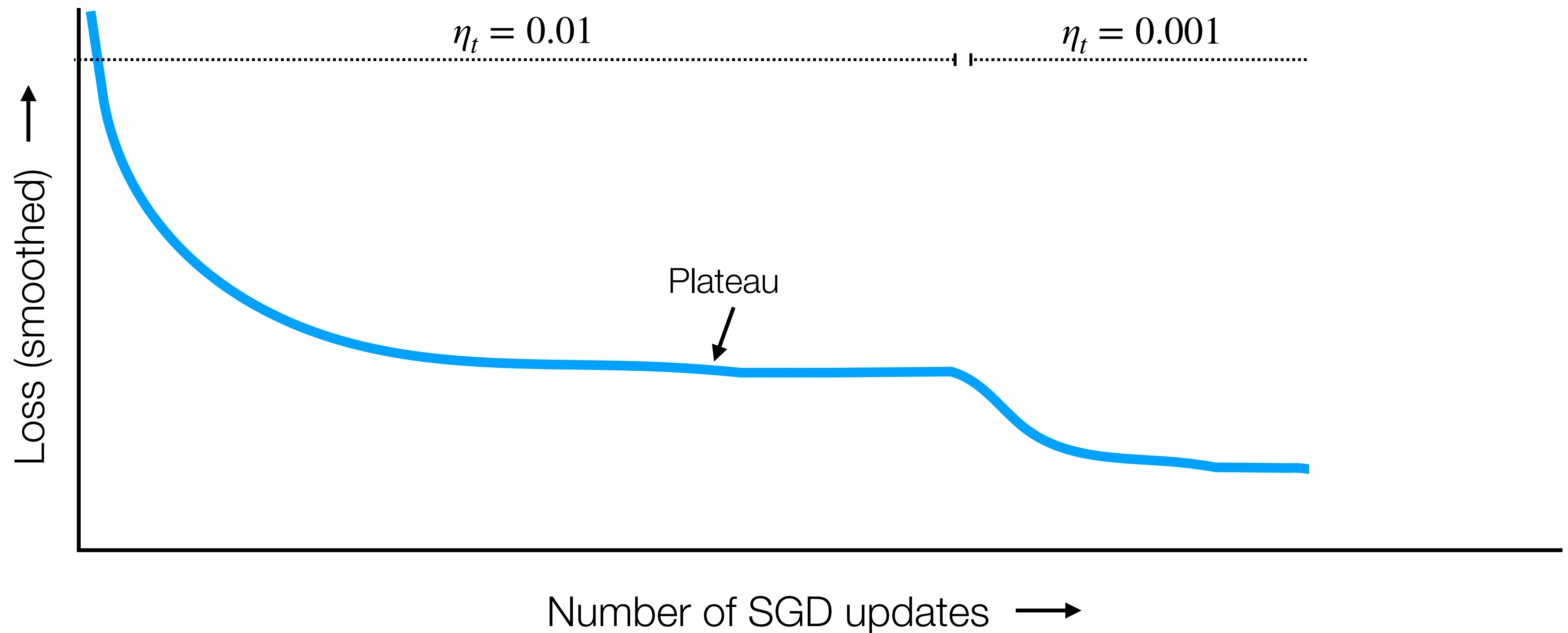
# Learning rate schedules

- Start with high learning rate, and decrease over time.



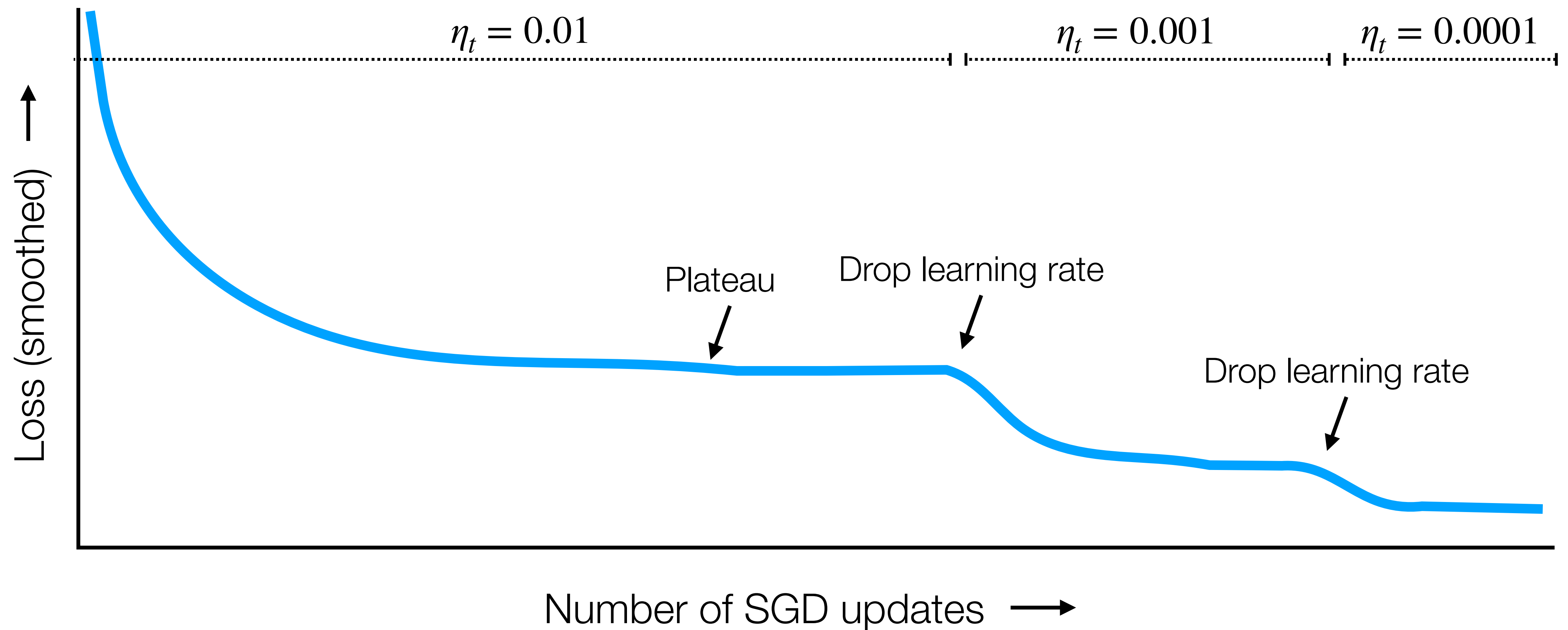
# Learning rate schedules

- Start with high learning rate, and decrease over time.



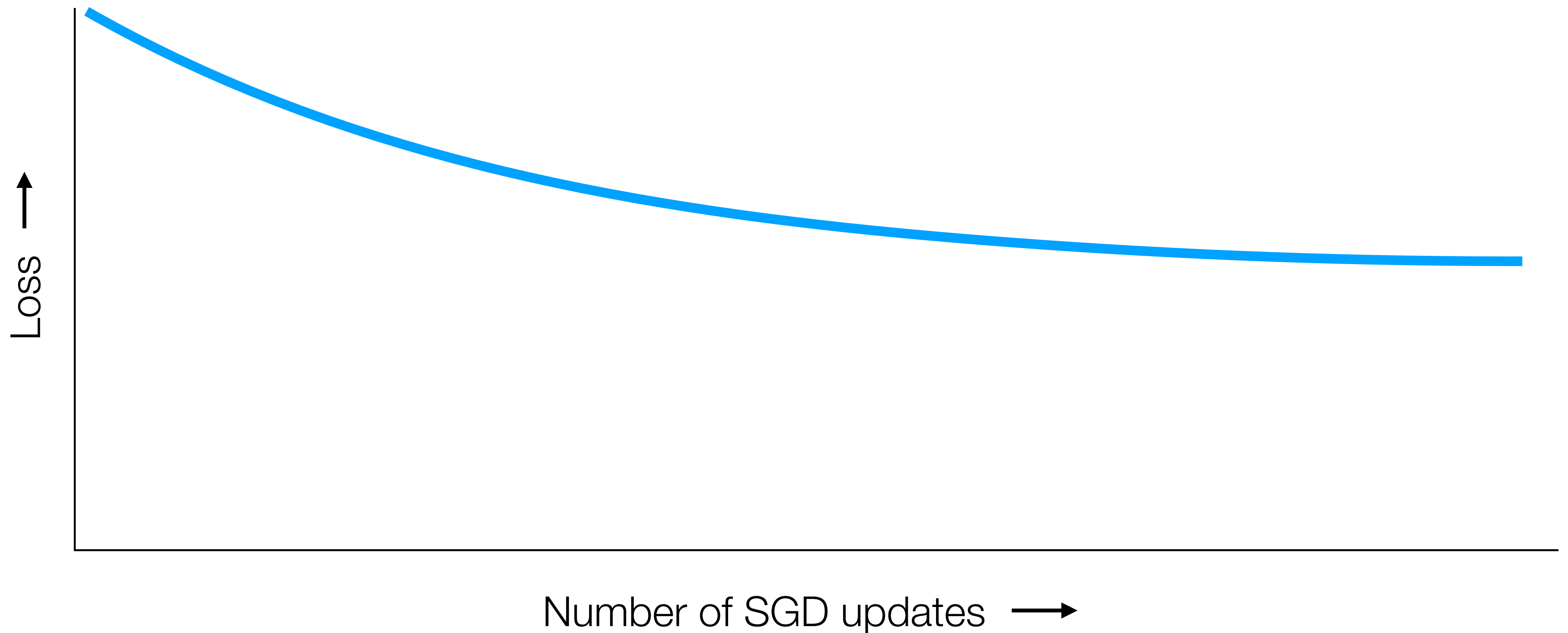
# Learning rate schedules

- Start with high learning rate, and decrease over time.



# Learning rate schedules

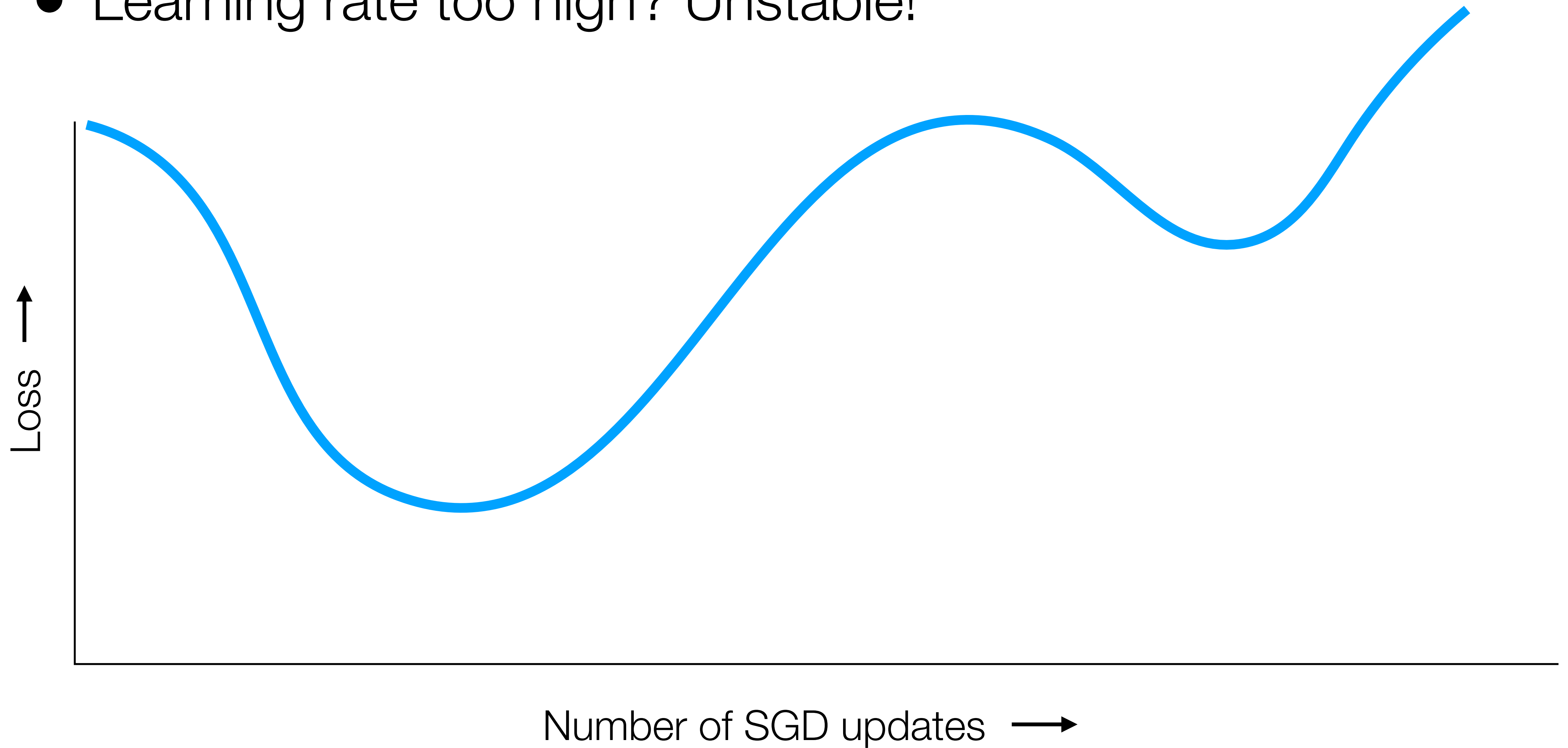
- What if you always use a small learning rate?
- Loss goes down extremely slowly





# Learning rate schedules

- Learning rate too high? Unstable!

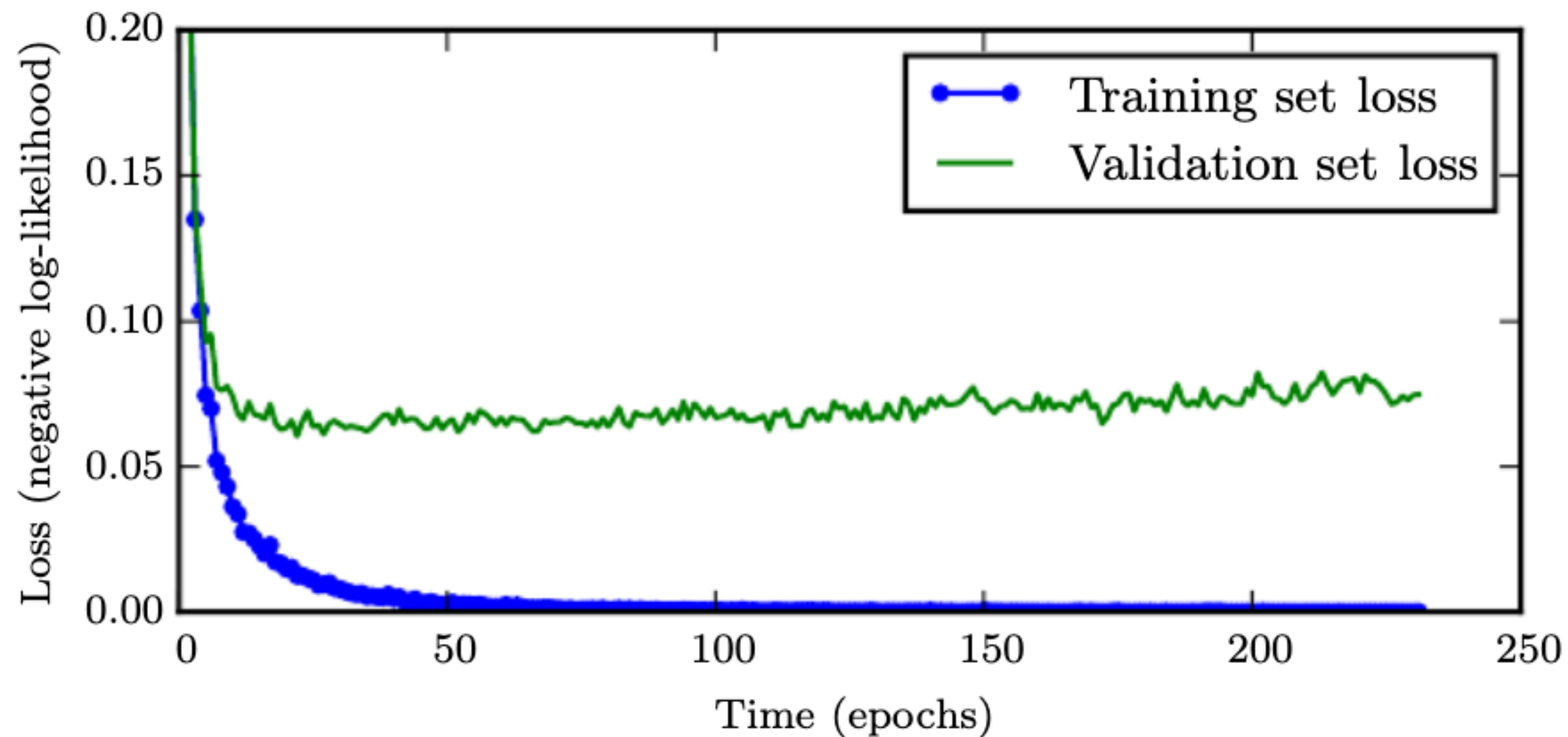


# Learning rate schedules

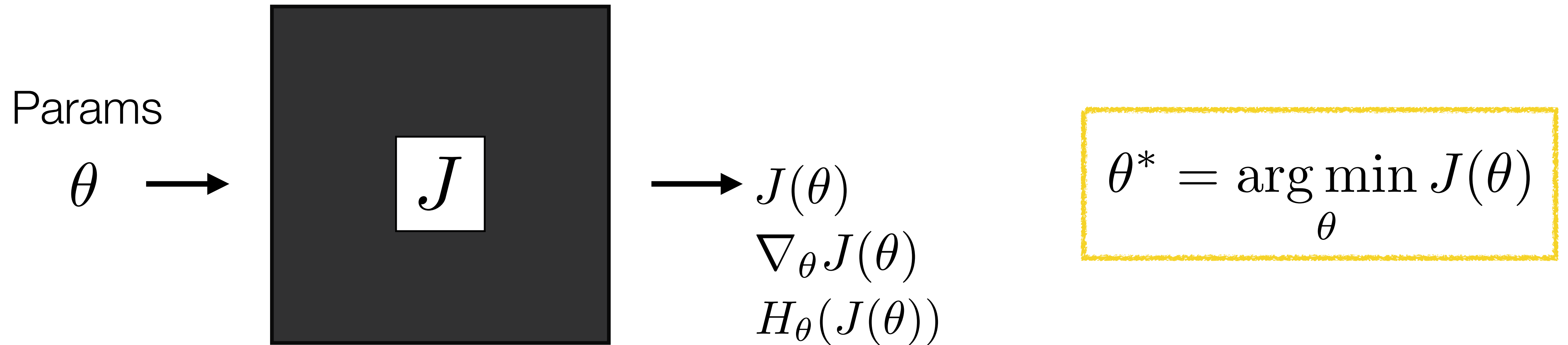
- Choosing the initial learning rate is surprisingly hard. Often requires grid search, i.e., trying many rates and choosing the best.
- When do you drop the learning rate? Some strategies:
  - Wait until validation or training loss plateaus, then drop it (e.g., by a factor of 10).
  - Smoothly drop the learning rate over time. Requires choosing the rate of dropping.
- Another option: decrease *and* increase the learning rate using a periodic function (e.g., cosine) [Loshchilov & Hutter, 2017]. Perhaps surprisingly, this works well.

# Regularization

- As before, add  $L_2$  regularization  $R(\theta) = \|\theta\|^2$  model parameters.
- **Early stopping:** stop training when validation loss increases, and revert to previous checkpoint.



# Optimization beyond gradient descent



- 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**

# PS3: intro to machine learning

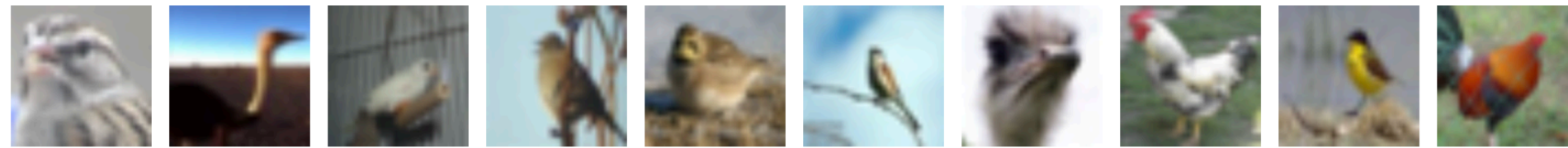
**airplane**



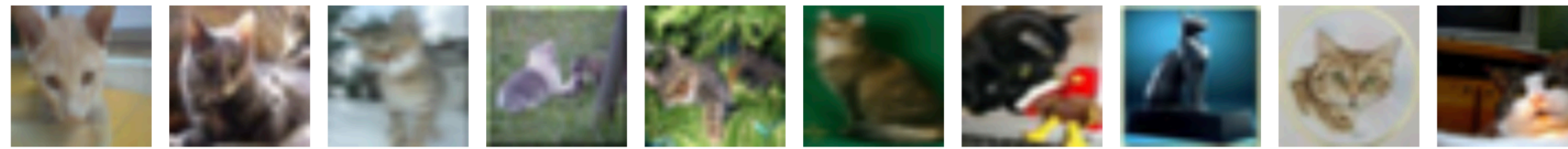
**automobile**



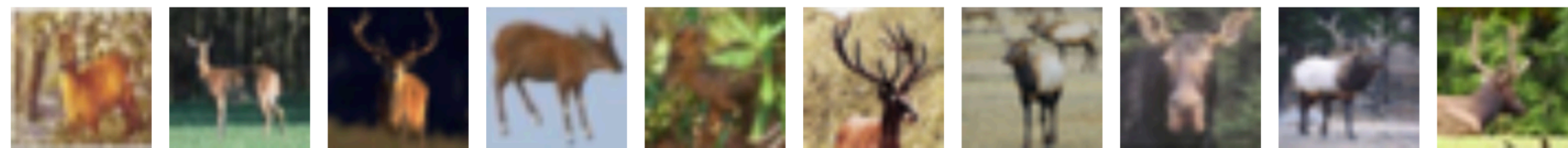
**bird**



**cat**



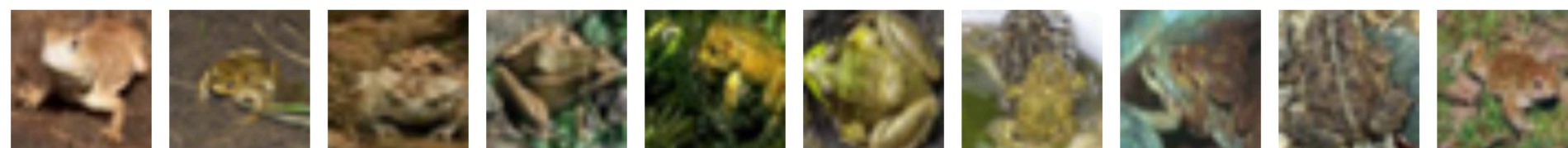
**deer**



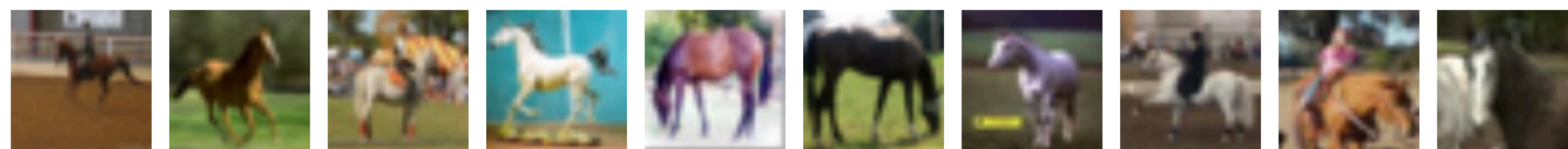
**dog**



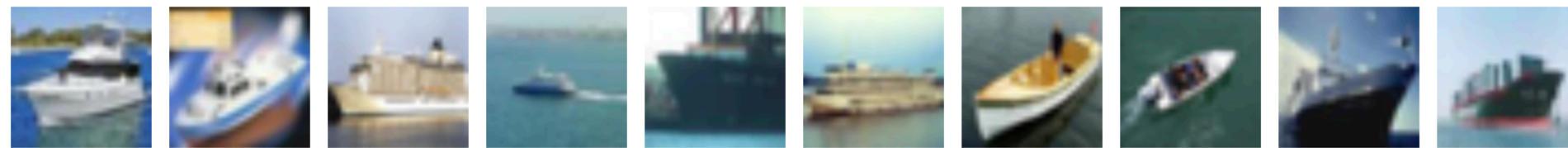
**frog**



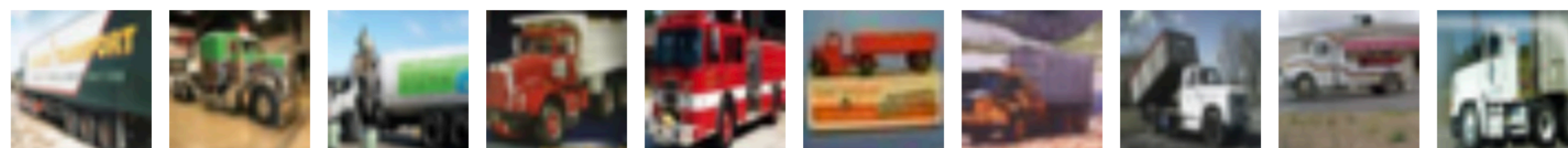
**horse**



**ship**

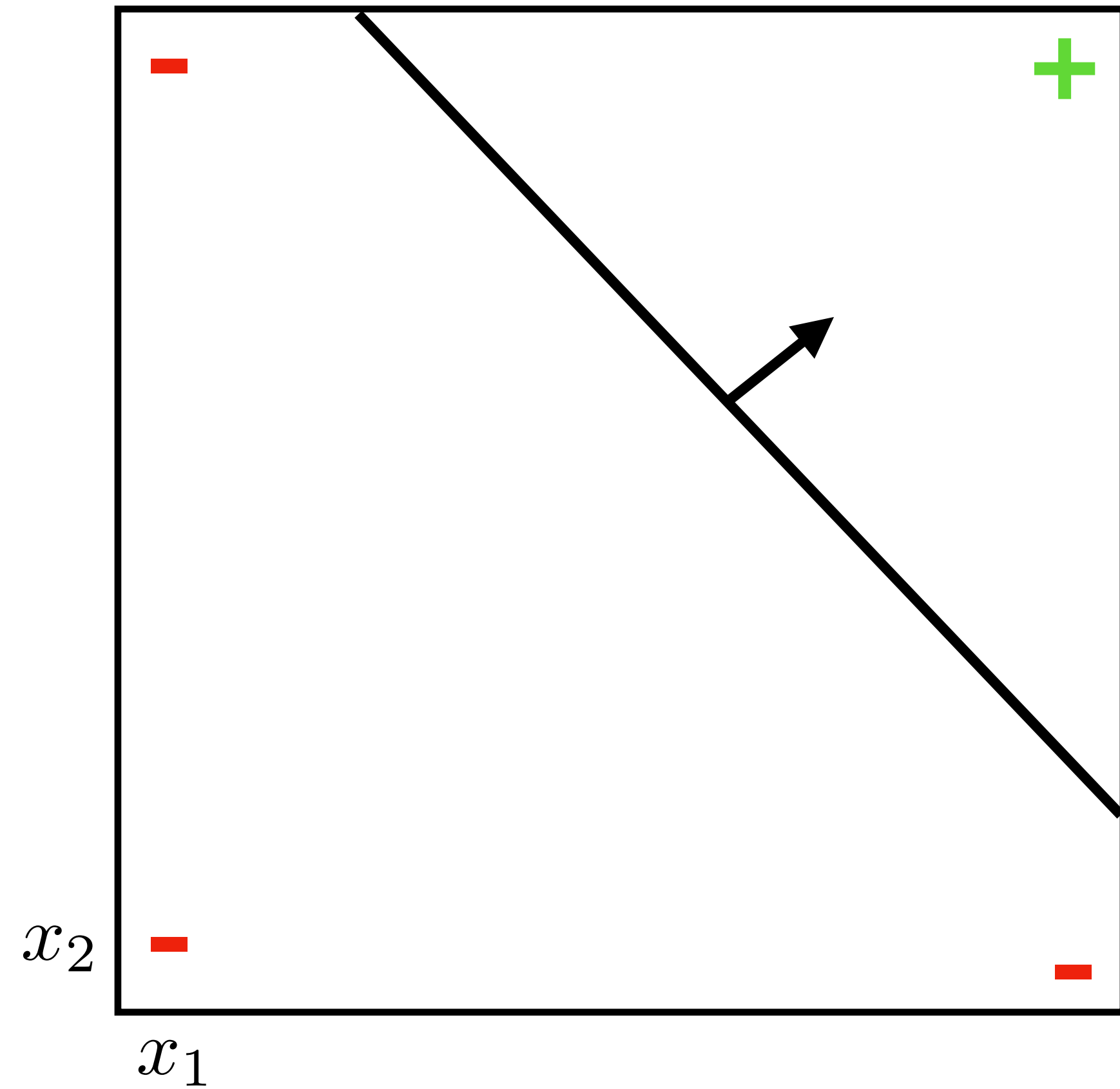


**truck**



- Object recognition with small images
- Hand-crafted image features
- K-nearest neighbor
- Logistic regression with SGD
- Cross-validation

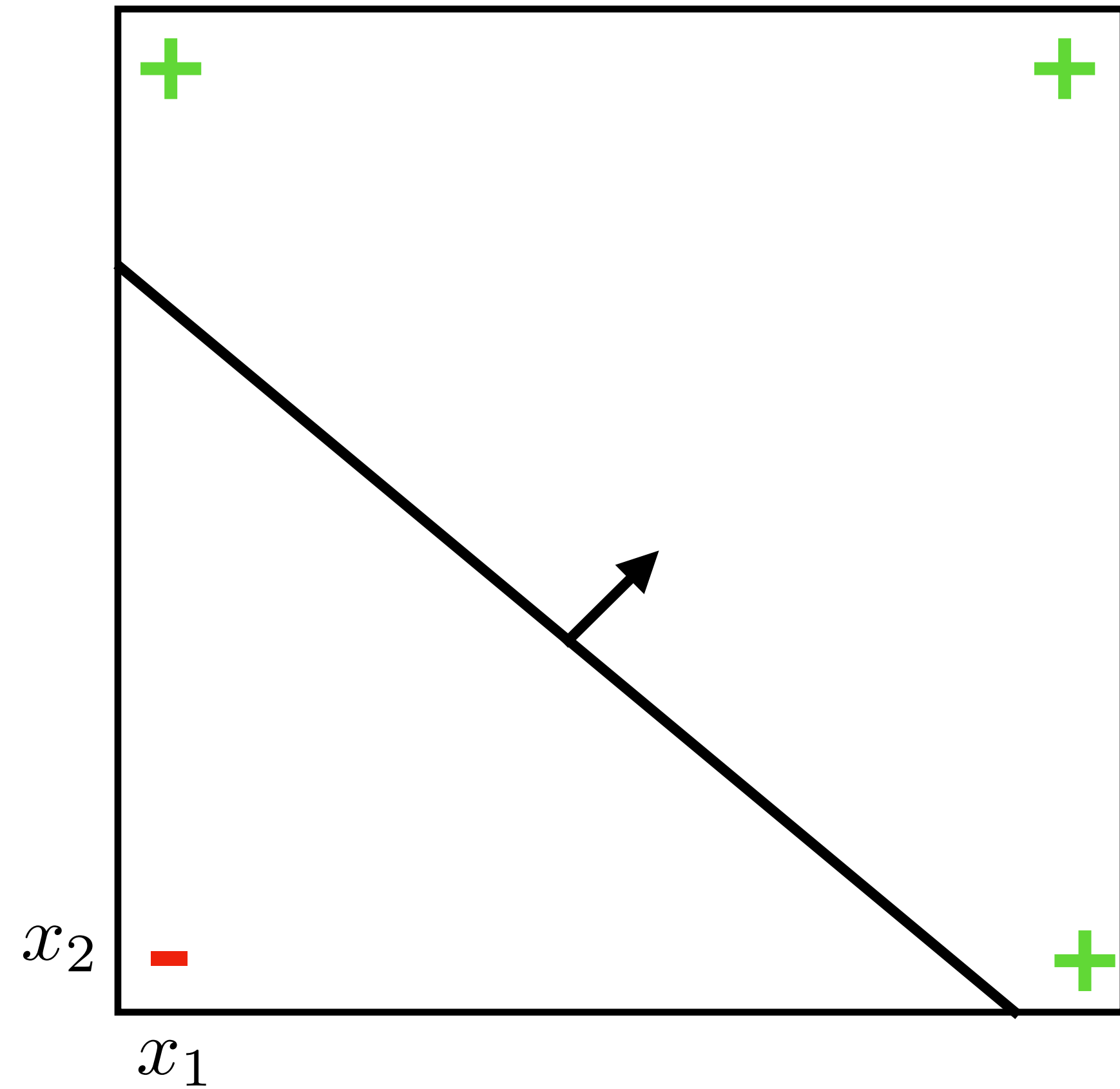
# Limitations to linear classifiers



		$x_2$	
		0	1
$x_1$	0	0	0
	1	0	1

AND

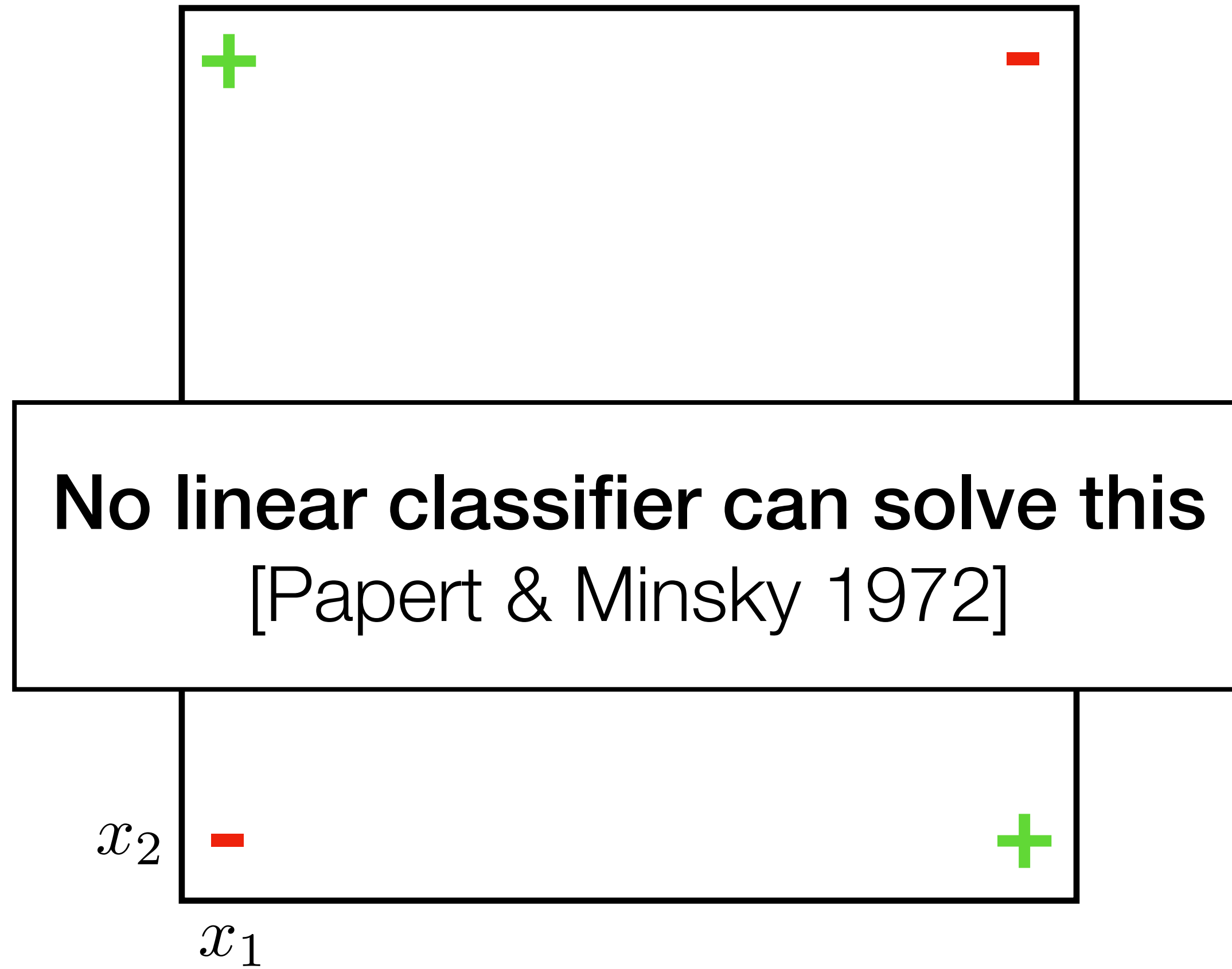
# Limitations to linear classifiers



		$x_2$	
		0	1
$x_1$	0	0	1
	1	1	1

OR

# Limitations to linear classifiers



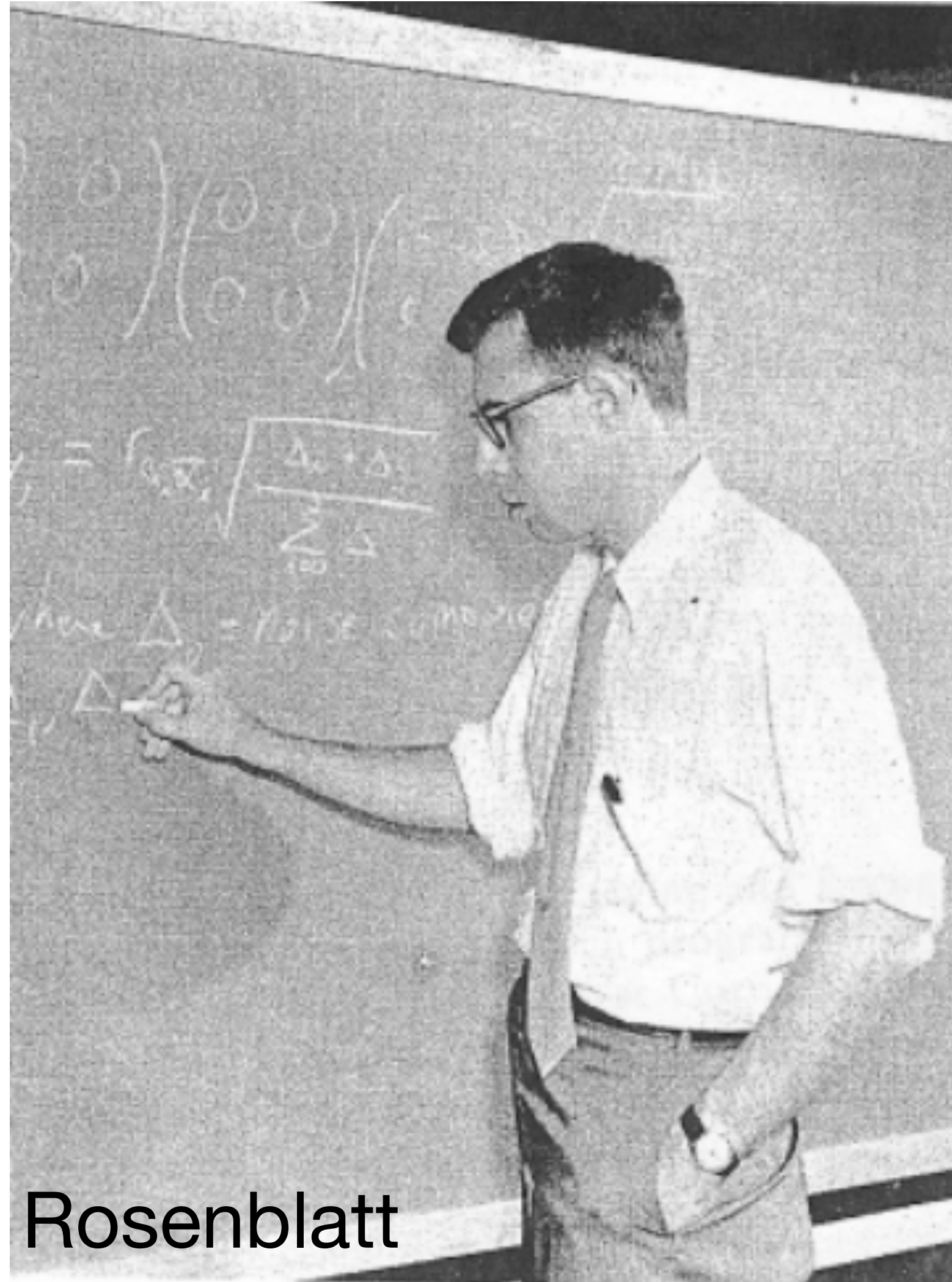
	$x_2$	
	0	1
$x_1$	0	0
	1	1

XOR



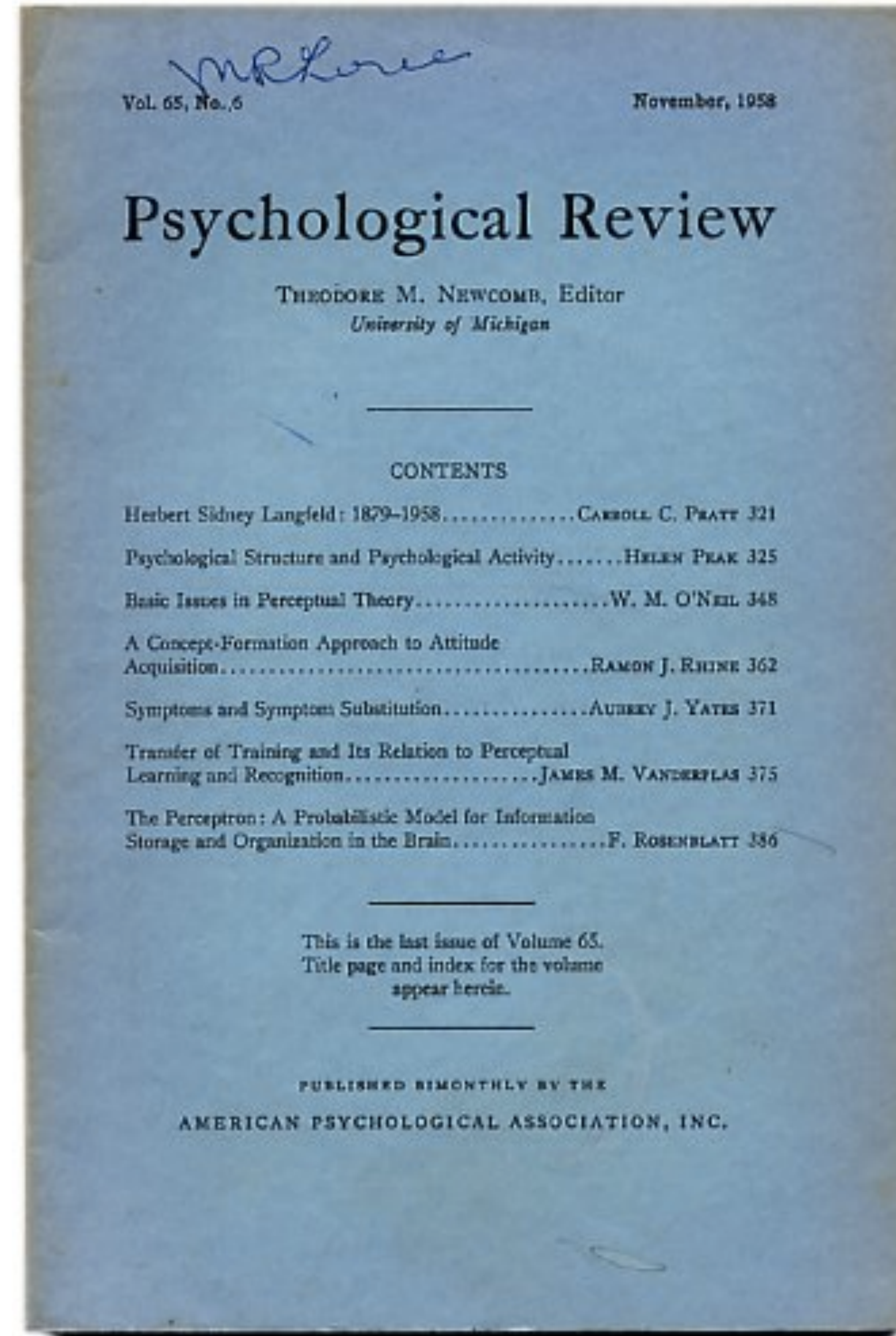
# Neural networks

# Perceptrons, 1958



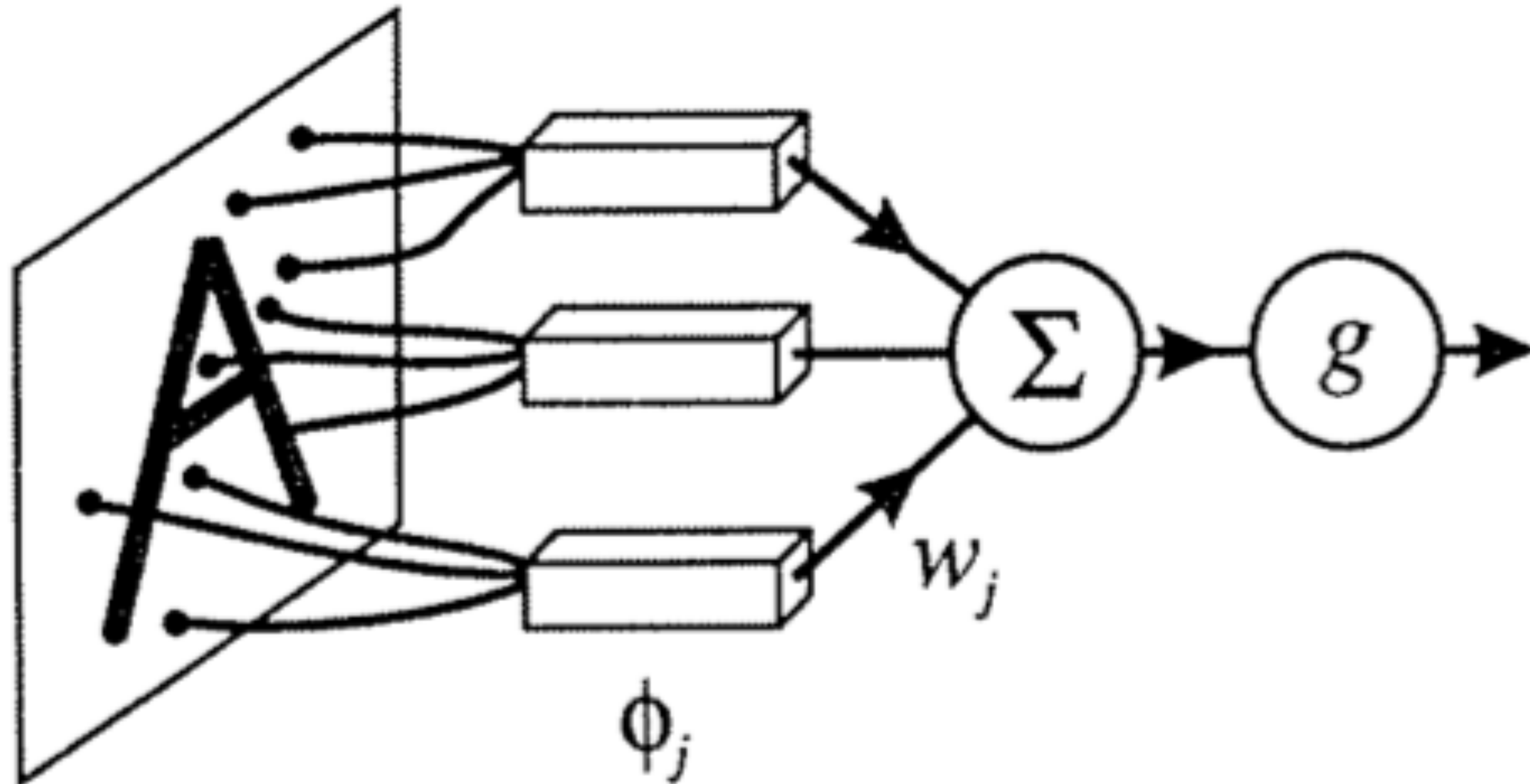
Rosenblatt

[http://www.ecse.rpi.edu/homepages/nagy/PDF\\_chrono/2011\\_Nagy\\_Pace\\_FR.pdf](http://www.ecse.rpi.edu/homepages/nagy/PDF_chrono/2011_Nagy_Pace_FR.pdf). Photo by George Nagy



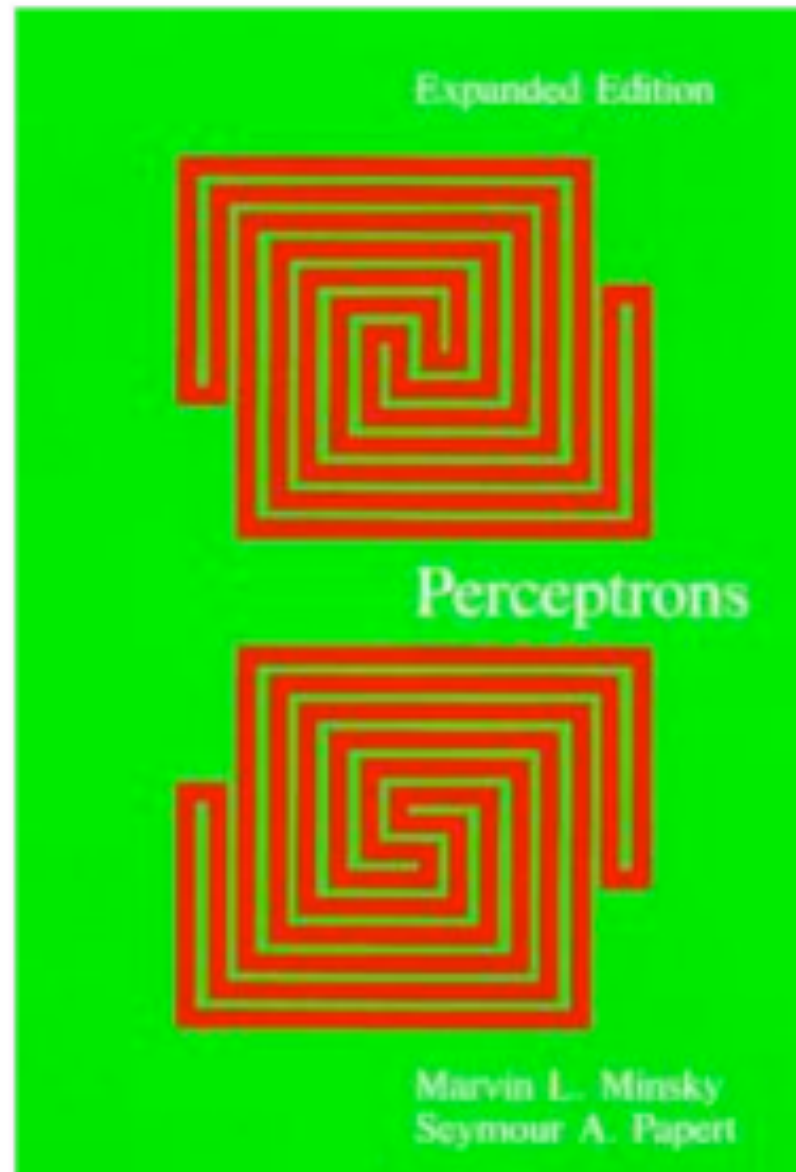
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.335.3398&rep=rep1&type=>

# Perceptrons, 1958



Very similar to the linear models we've seen.

# Minsky and Papert, Perceptrons, 1972



FOR BUYING OPTIONS, START HERE

Select Shipping Destination

Paperback | \$35.00 Short | £24.95 | ISBN: 9780262631112 | 308 pp. | 6 x 8.9 in | December 1987

## Perceptrons, expanded edition

An Introduction to Computational Geometry

By [Marvin Minsky](#) and [Seymour A. Papert](#)

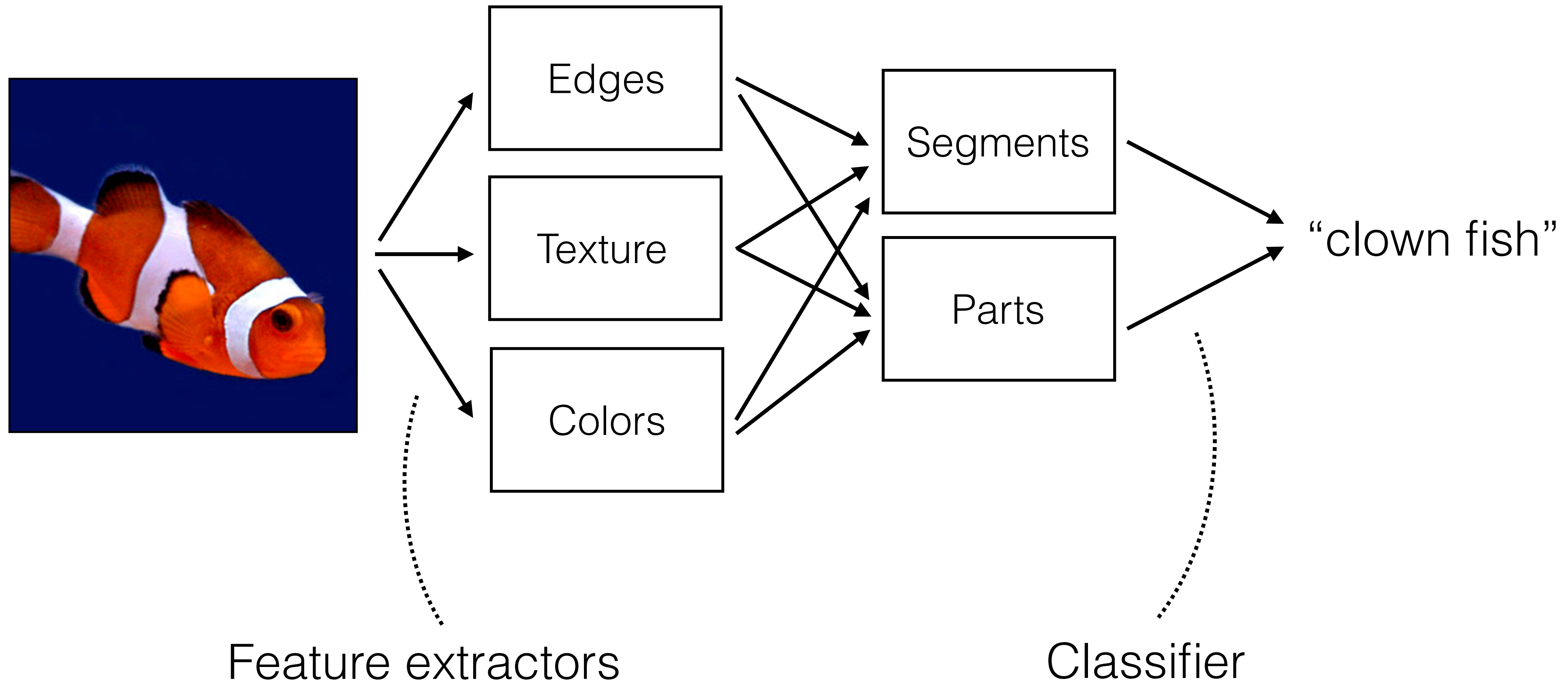
### Overview

*Perceptrons* - the first systematic study of parallelism in computation - has remained a classical work on threshold automata networks for nearly two decades. It marked a historical turn in artificial intelligence, and it is required reading for anyone who wants to understand the connectionist counterrevolution that is going on today.

Artificial-intelligence research, which for a time concentrated on the programming of ton Neumann computers, is swinging back to the idea that intelligence might emerge from the activity of networks of neuronlike entities. Minsky and Papert's book was the first example of a mathematical analysis carried far enough to show the exact limitations of a class of computing machines that could seriously be considered as models of the brain. Now the new developments in mathematical tools, the recent interest of physicists in the theory of disordered matter, the new insights into and psychological models of how the brain works, and the evolution of fast computers that can simulate networks of automata have given *Perceptrons* new importance.

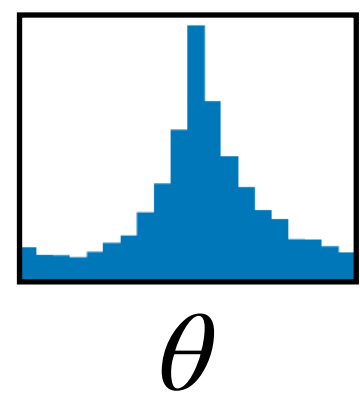
Witnessing the swing of the intellectual pendulum, Minsky and Papert have added a new chapter in which they discuss the current state of parallel computers, review developments since the appearance of the 1972 edition, and identify new research directions related to connectionism. They note a central theoretical challenge facing connectionism: the challenge to reach a deeper understanding of how "objects" or "agents" with individuality can emerge in a network. Progress in this area would link connectionism with what the authors have called "society theories of mind."

# “Classic” recognition **without** neural nets

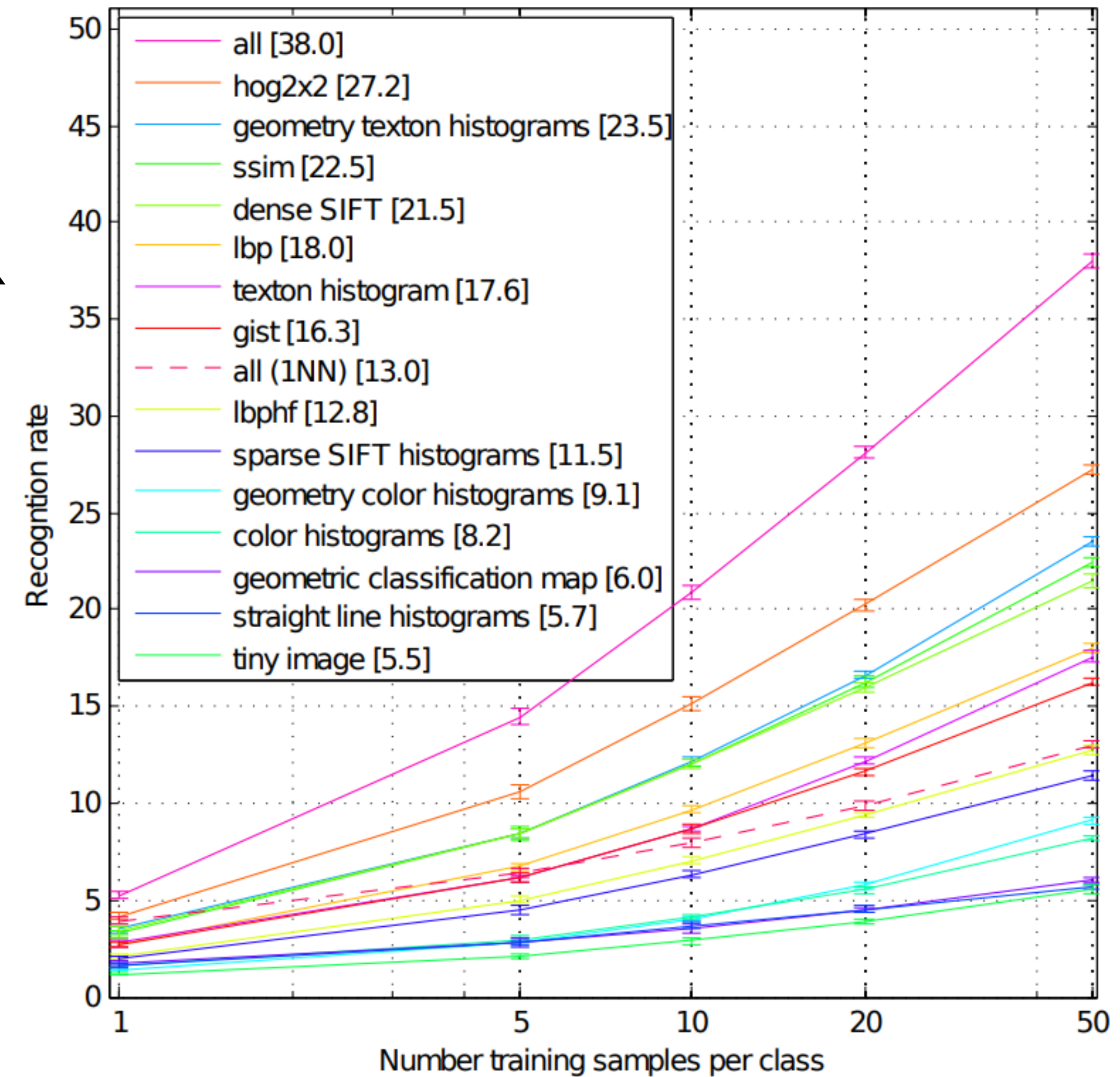


# “Classic” recognition **without** neural nets

Hand-crafted features

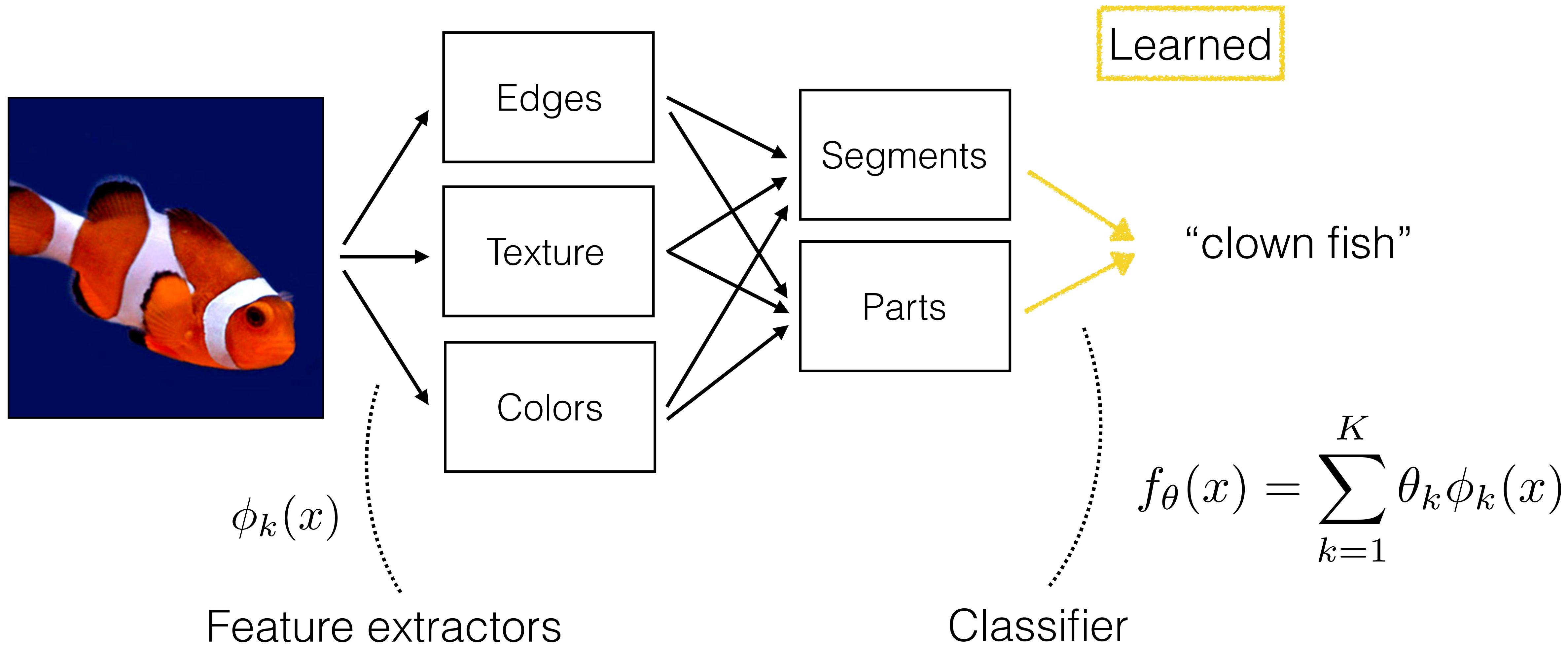


edge orientation histogram

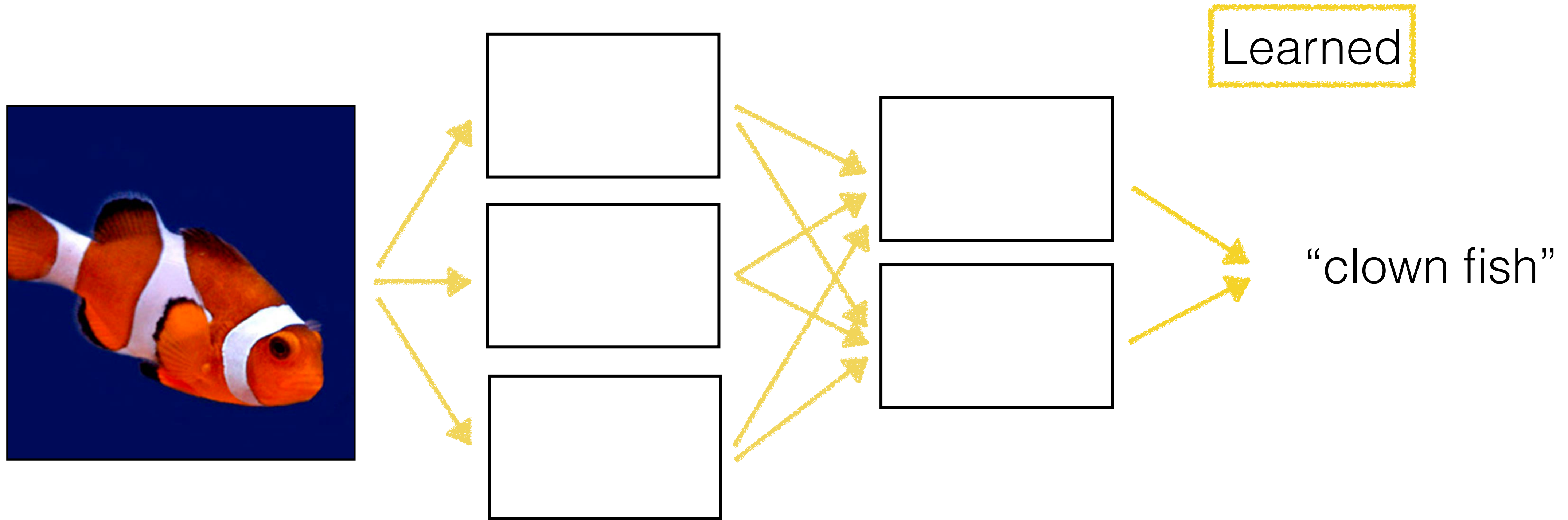


[Xiao et al., “SUN database”, 2010]

# “Classic” object recognition **without** neural nets

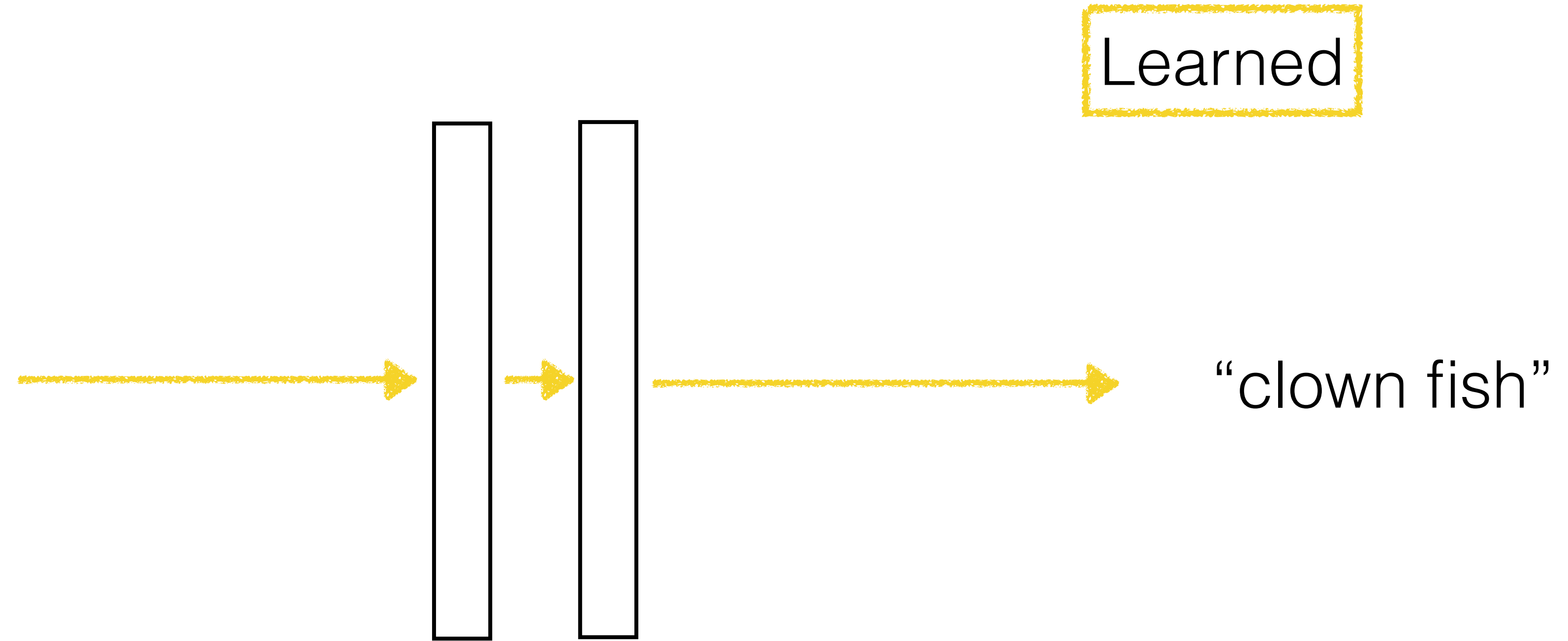
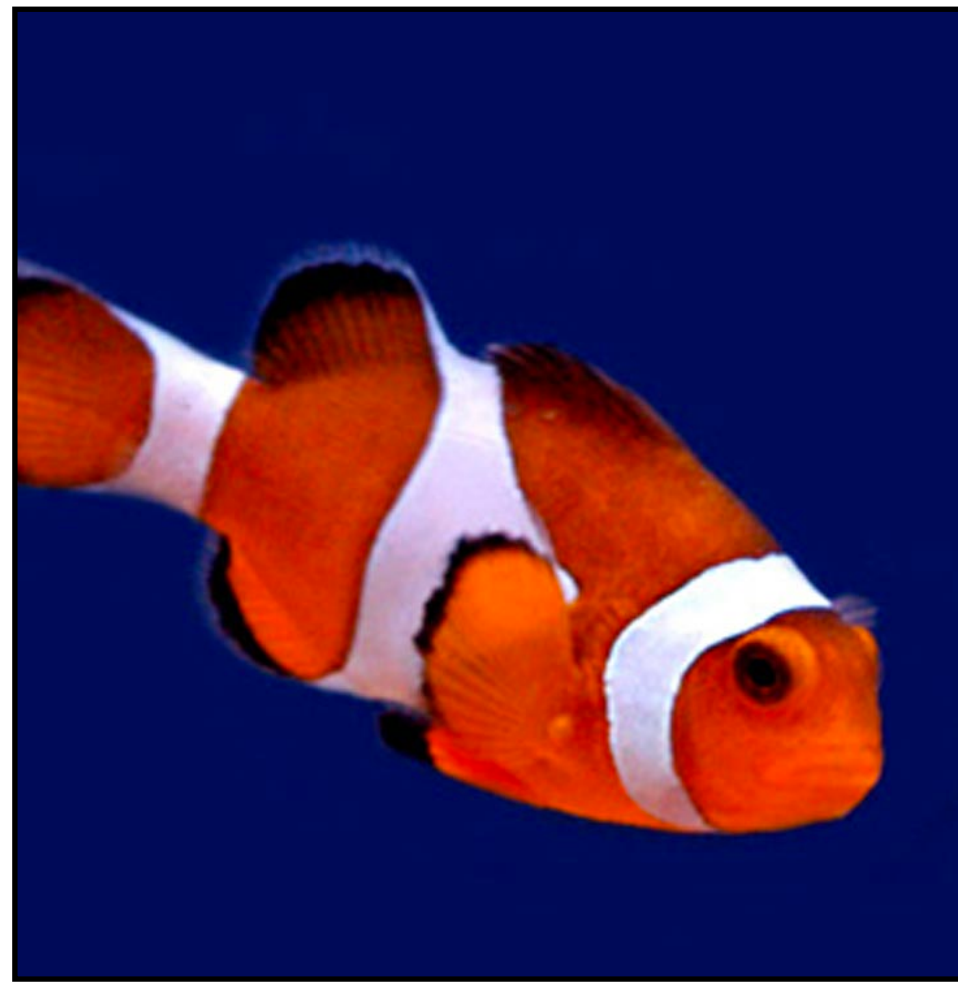


# Object recognition





# Object recognition

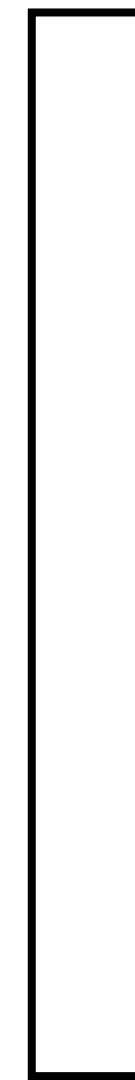
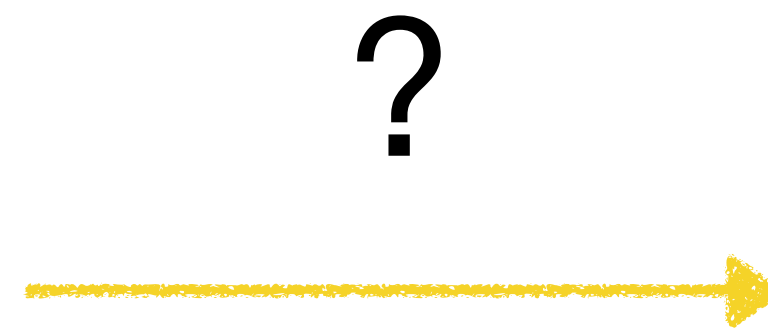
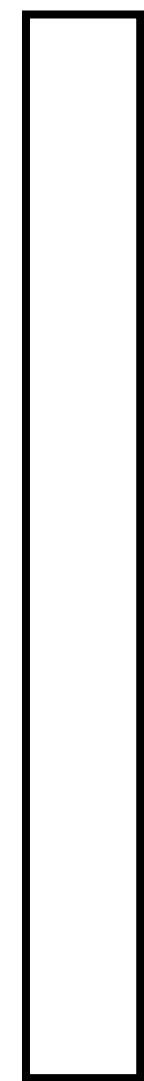


**Neural net**

# Computation in a neural net

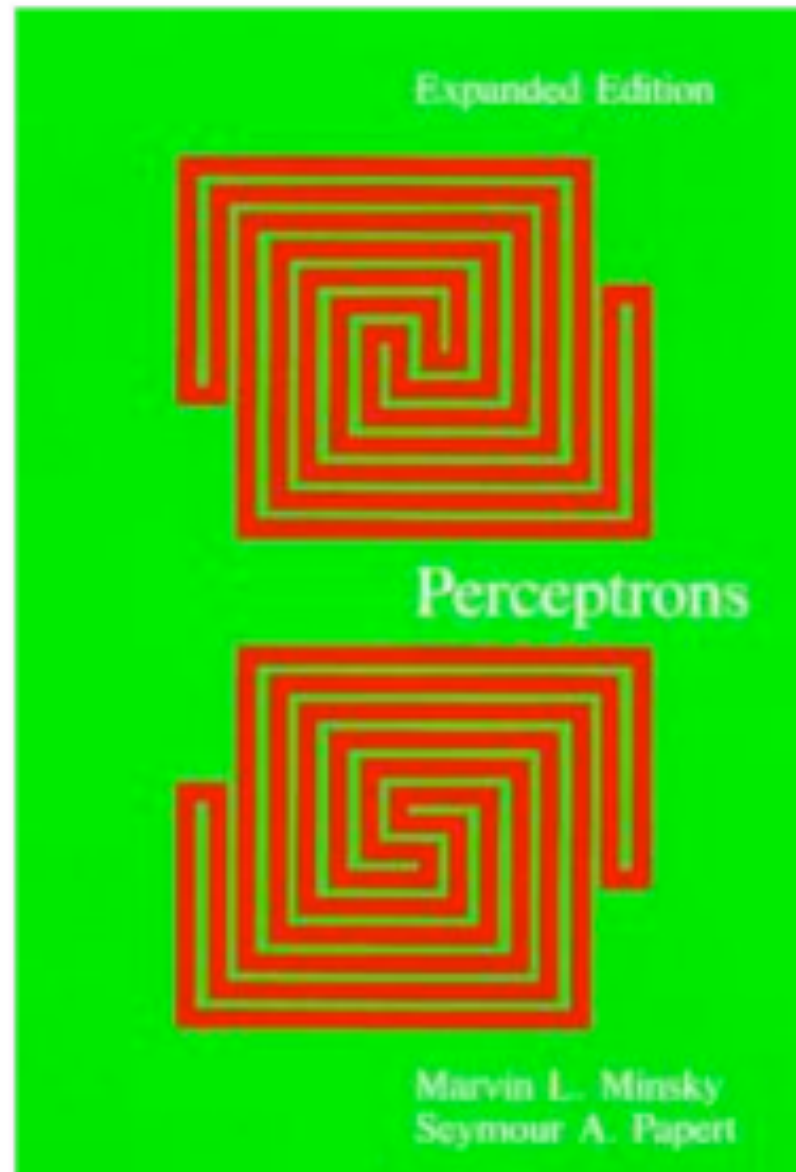
Input vector

Output vector



Next lecture: neural networks

# Minsky and Papert, Perceptrons, 1972



FOR BUYING OPTIONS, START HERE

Select Shipping Destination

Paperback | \$35.00 Short | £24.95 | ISBN: 9780262631112 | 308 pp. | 6 x 8.9 in | December 1987

## Perceptrons, expanded edition

An Introduction to Computational Geometry

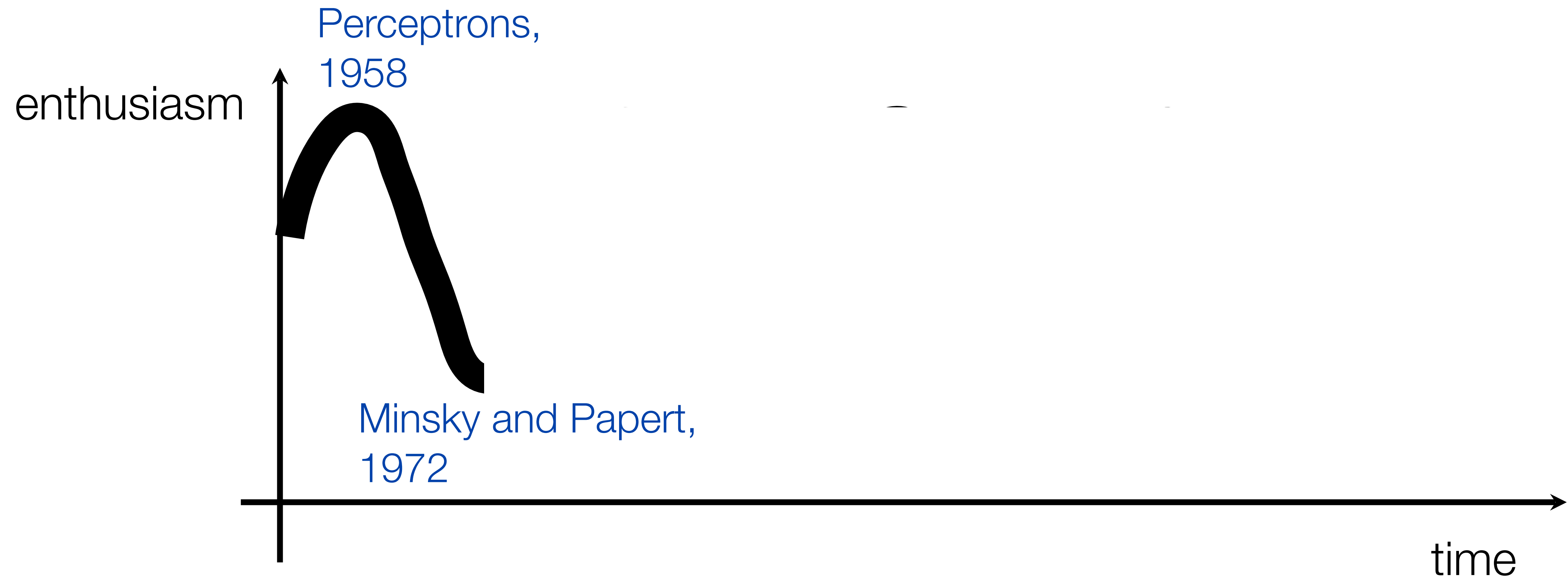
By [Marvin Minsky](#) and [Seymour A. Papert](#)

### Overview

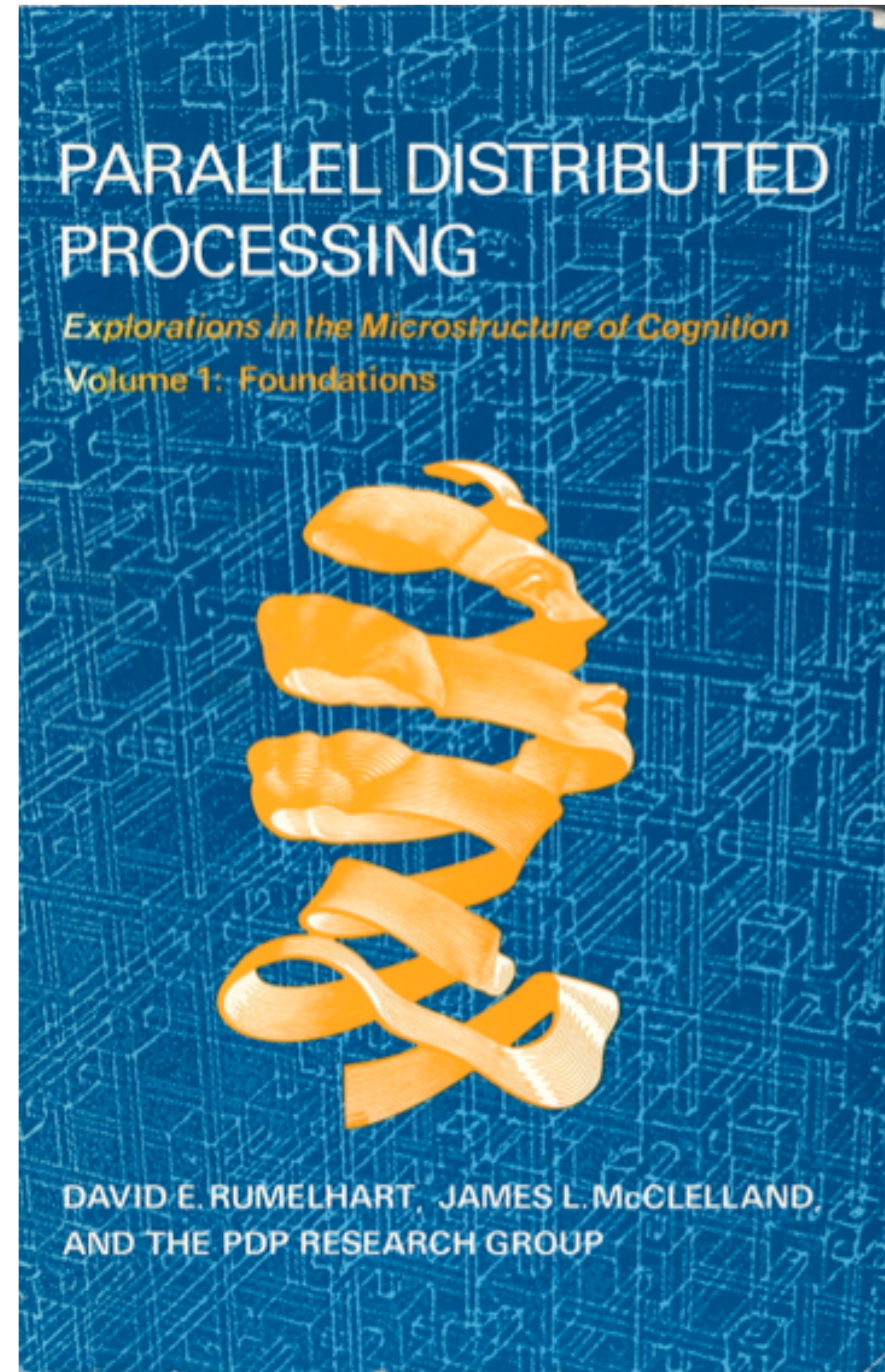
*Perceptrons* - the first systematic study of parallelism in computation - has remained a classical work on threshold automata networks for nearly two decades. It marked a historical turn in artificial intelligence, and it is required reading for anyone who wants to understand the connectionist counterrevolution that is going on today.

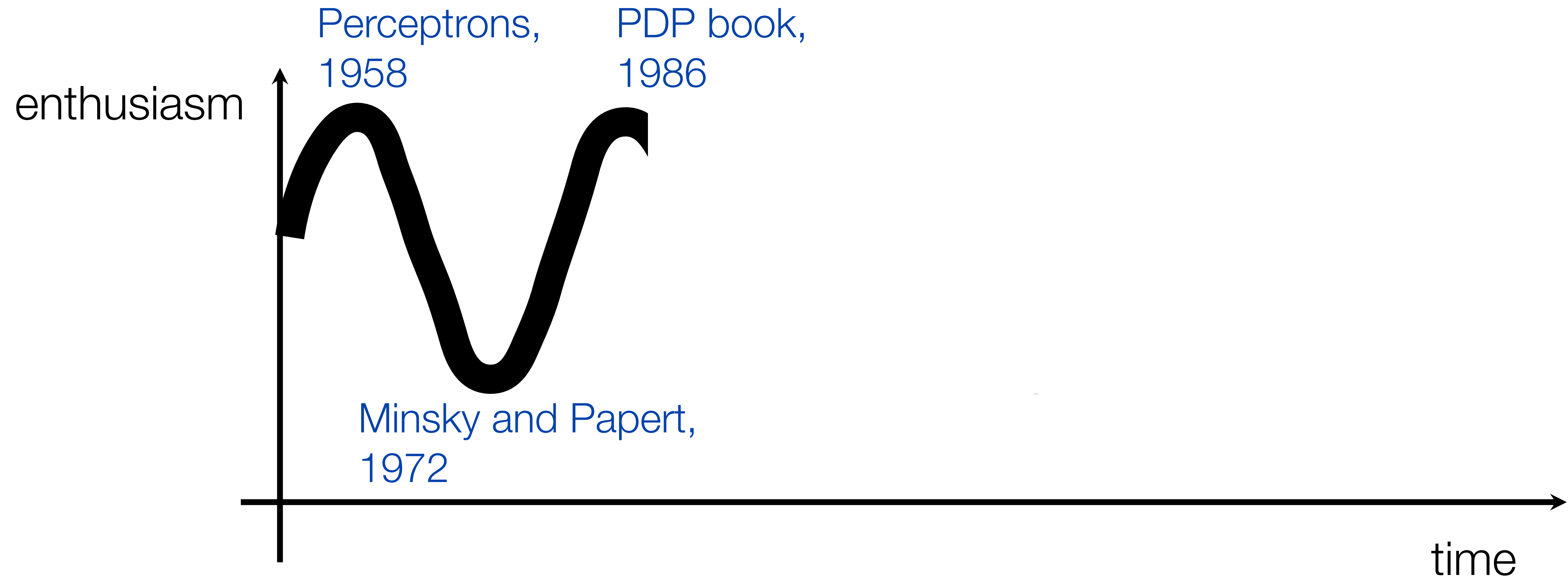
Artificial-intelligence research, which for a time concentrated on the programming of ton Neumann computers, is swinging back to the idea that intelligence might emerge from the activity of networks of neuronlike entities. Minsky and Papert's book was the first example of a mathematical analysis carried far enough to show the exact limitations of a class of computing machines that could seriously be considered as models of the brain. Now the new developments in mathematical tools, the recent interest of physicists in the theory of disordered matter, the new insights into and psychological models of how the brain works, and the evolution of fast computers that can simulate networks of automata have given *Perceptrons* new importance.

Witnessing the swing of the intellectual pendulum, Minsky and Papert have added a new chapter in which they discuss the current state of parallel computers, review developments since the appearance of the 1972 edition, and identify new research directions related to connectionism. They note a central theoretical challenge facing connectionism: the challenge to reach a deeper understanding of how "objects" or "agents" with individuality can emerge in a network. Progress in this area would link connectionism with what the authors have called "society theories of mind."



# Parallel Distributed Processing (PDP), 1986





# LeCun convolutional neural networks

PROC. OF THE IEEE, NOVEMBER 1998

7

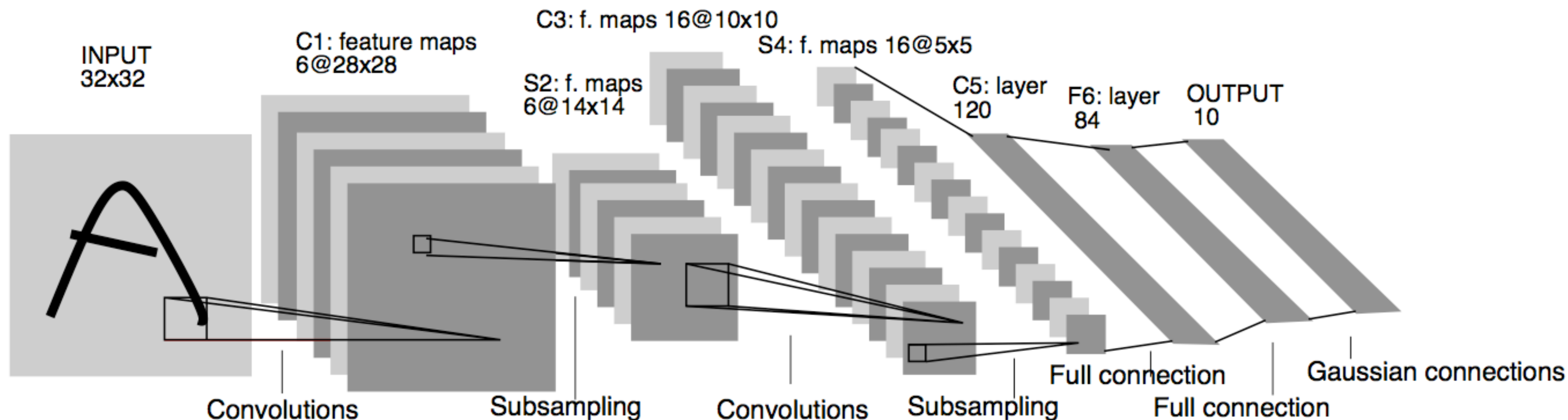


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Demos:

<http://yann.lecun.com/exdb/lenet/index.html>



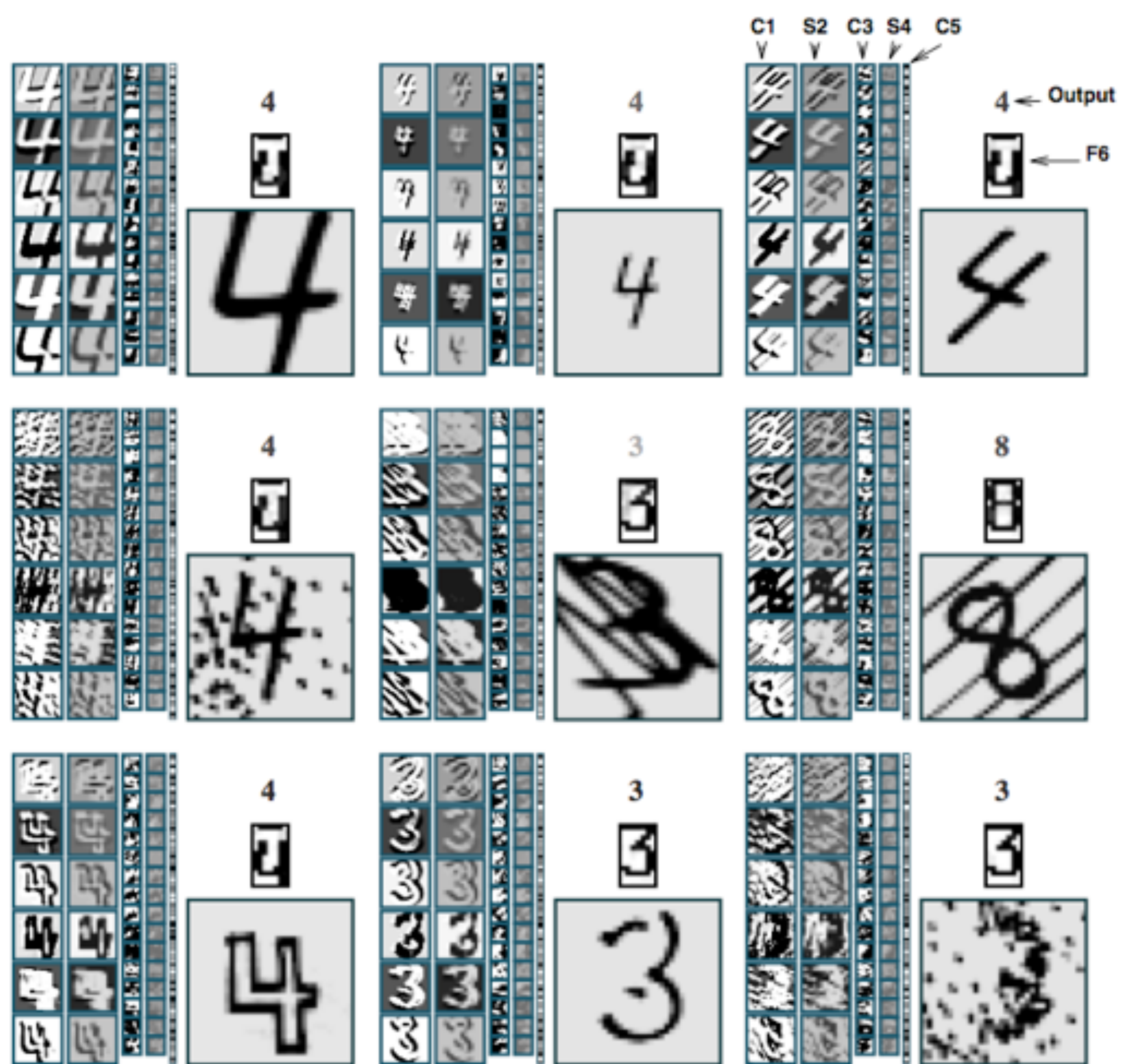
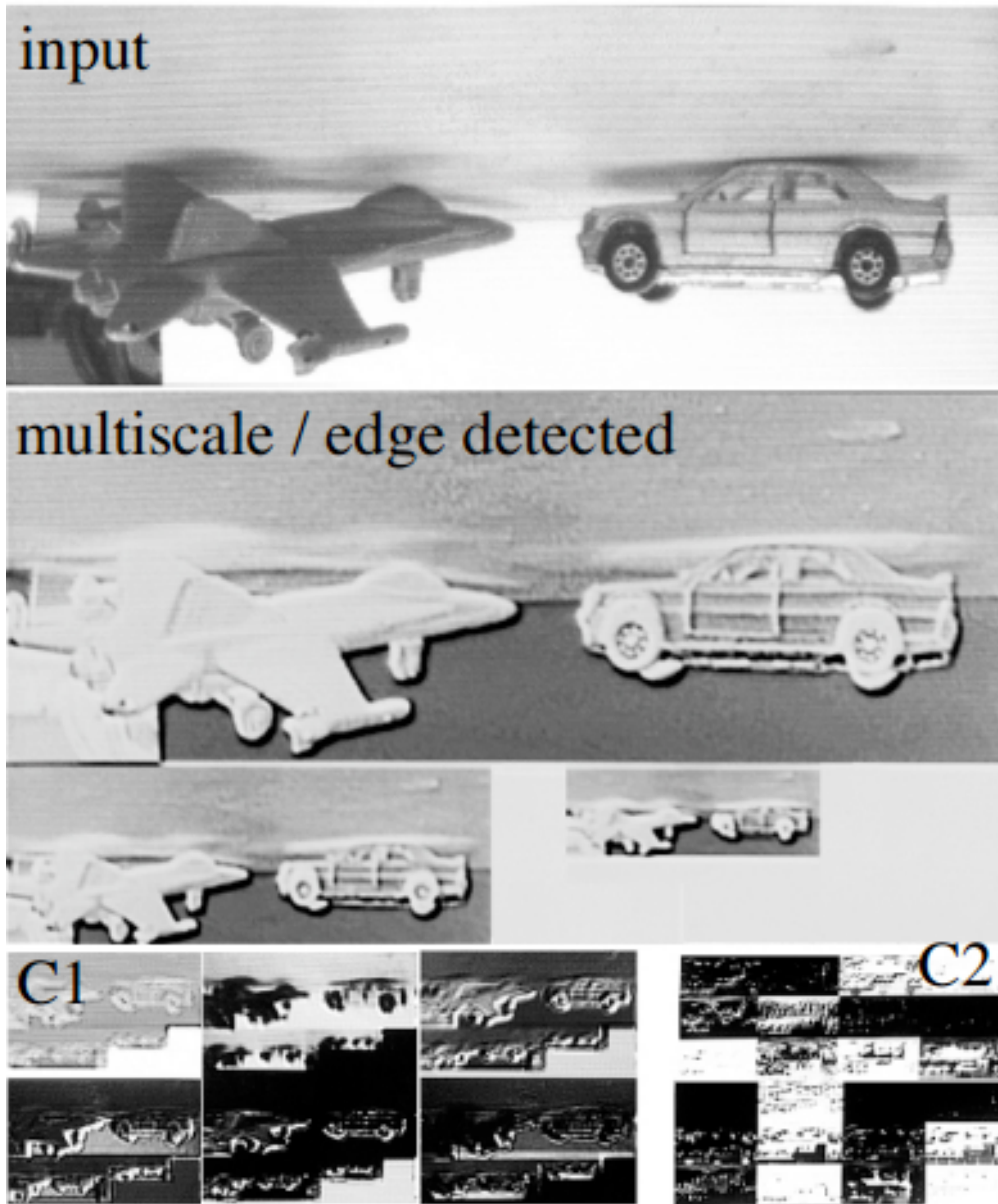


Fig. 13. Examples of unusual, distorted, and noisy characters correctly recognized by LeNet-5. The grey-level of the output label represents the penalty (lighter for higher penalties).



Neural networks to recognize handwritten digits and human faces?

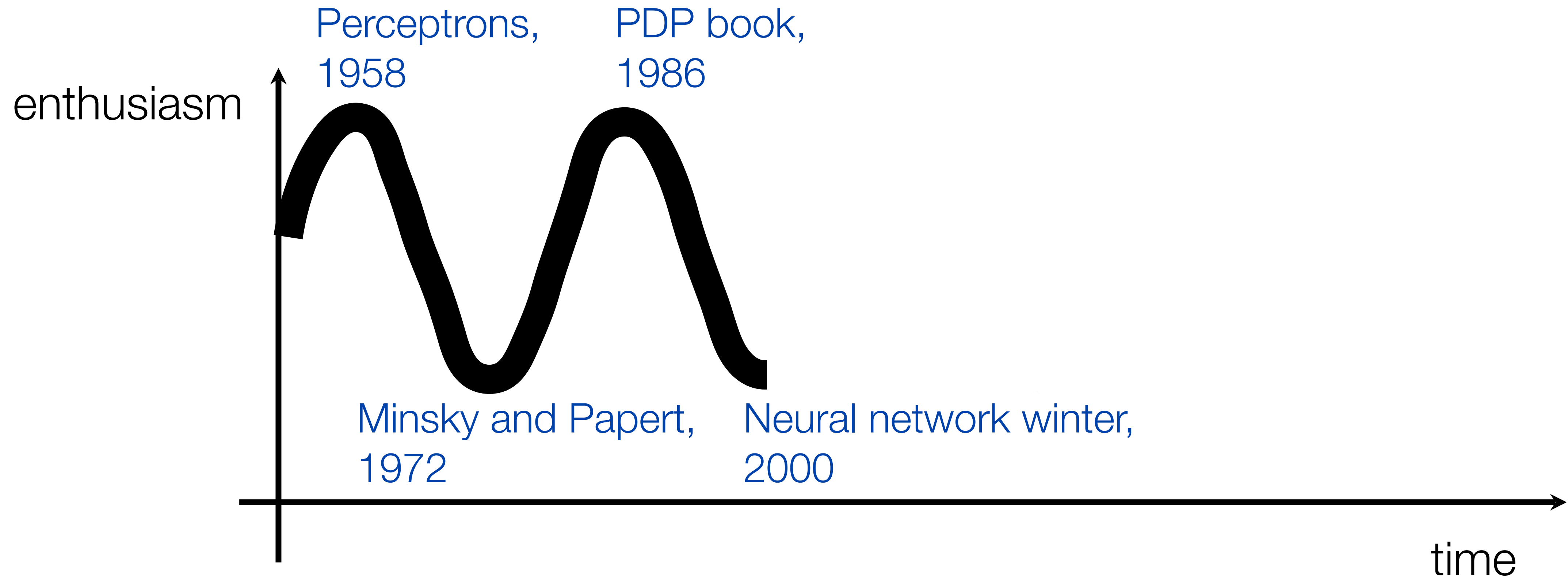
yes

Neural networks for tougher problems?

not really

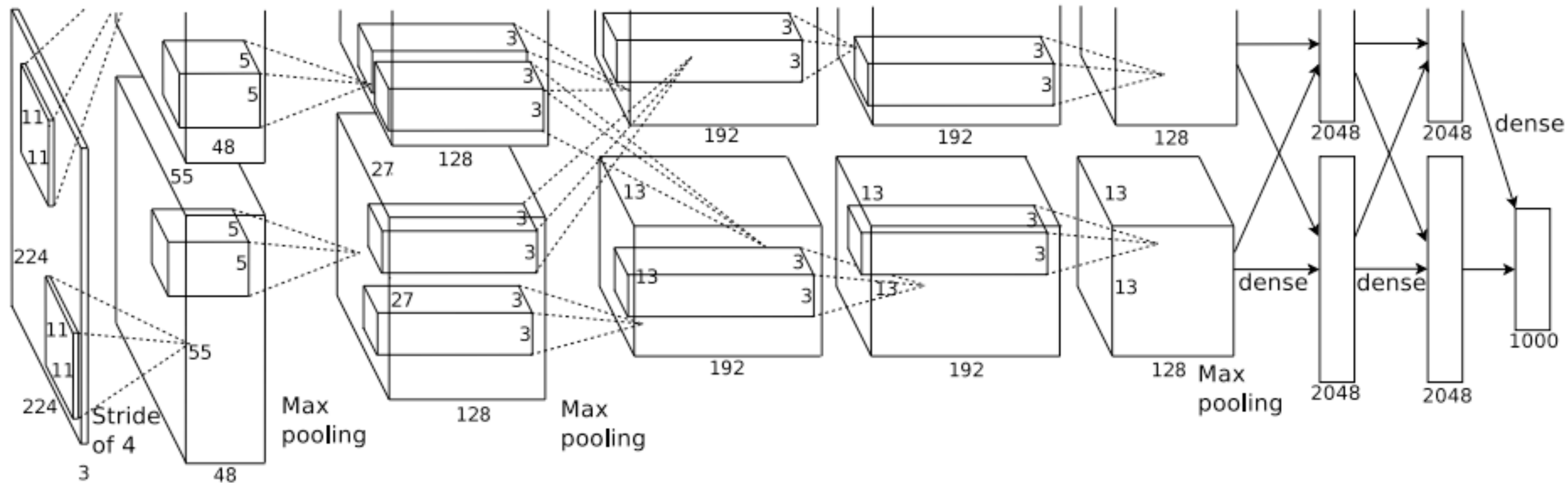
# Machine learning circa 2000

- Neural Information Processing Systems (NeurIPS), is a top conference on machine learning.
- For the 2000 conference:
  - title words predictive of paper acceptance: “Belief Propagation” and “Gaussian”.
  - title words predictive of paper rejection: “Neural” and “Network”.



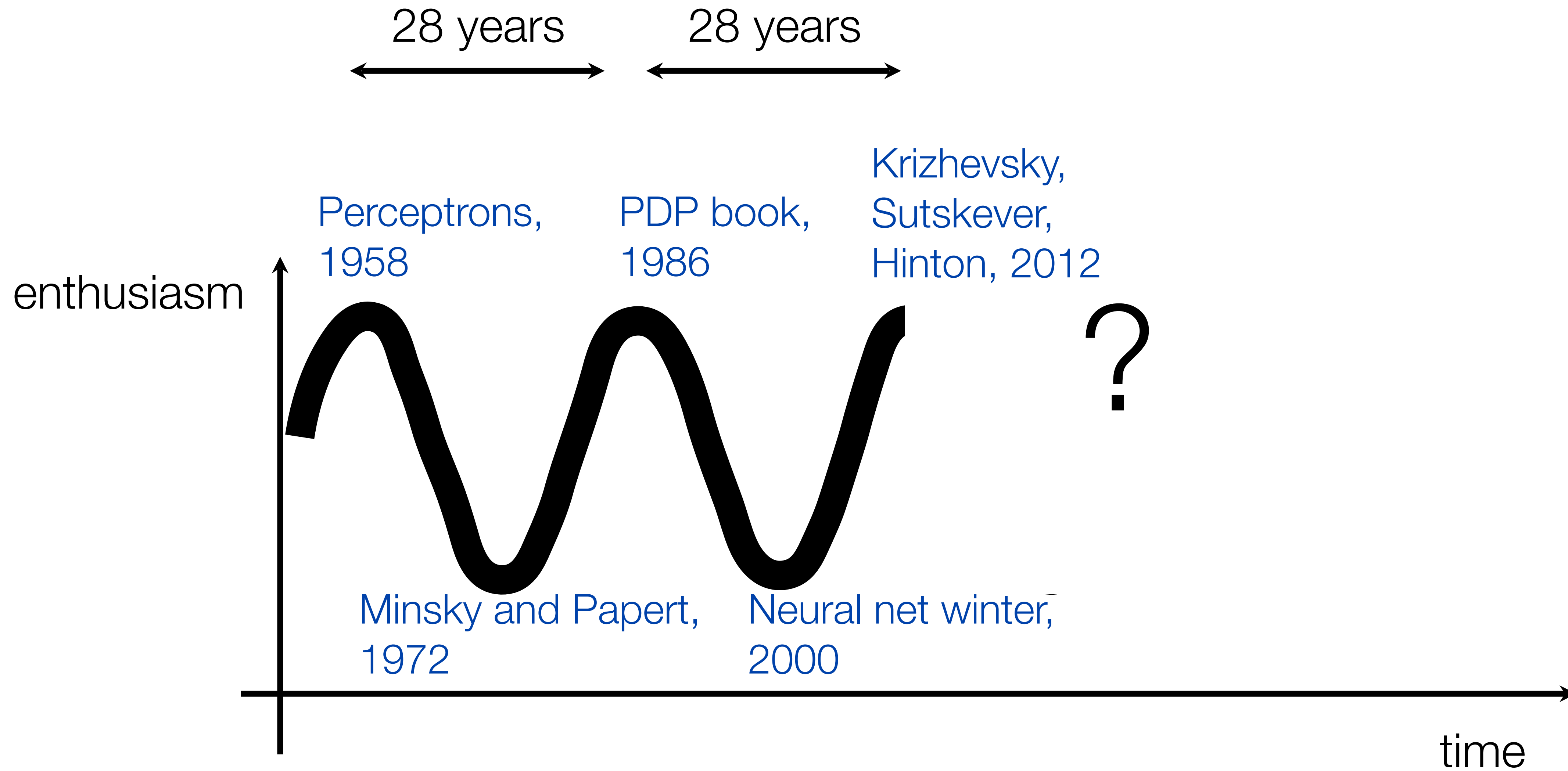
# Krizhevsky, Sutskever, and Hinton, NeurIPS 2012

## “AlexNet”

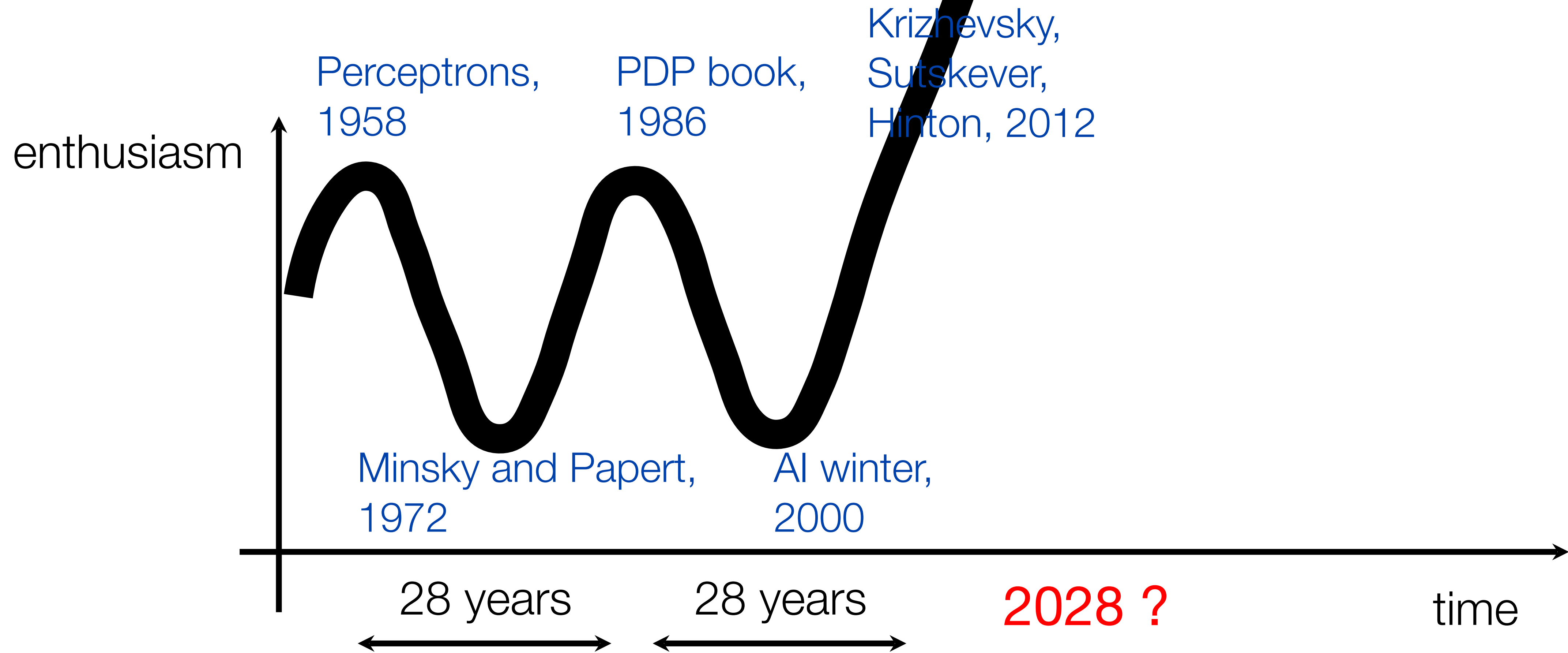


# Krizhevsky, Sutskever, and Hinton, NeurIPS 2012



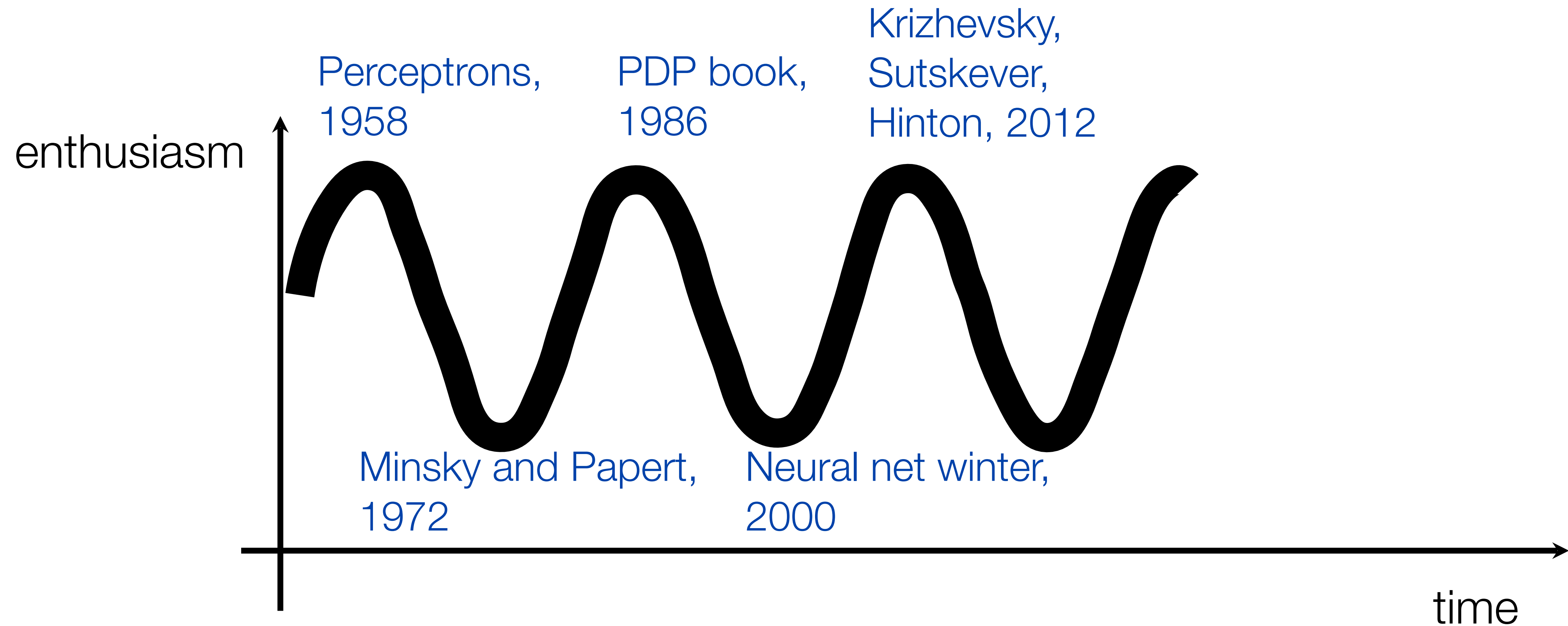


# What comes next?

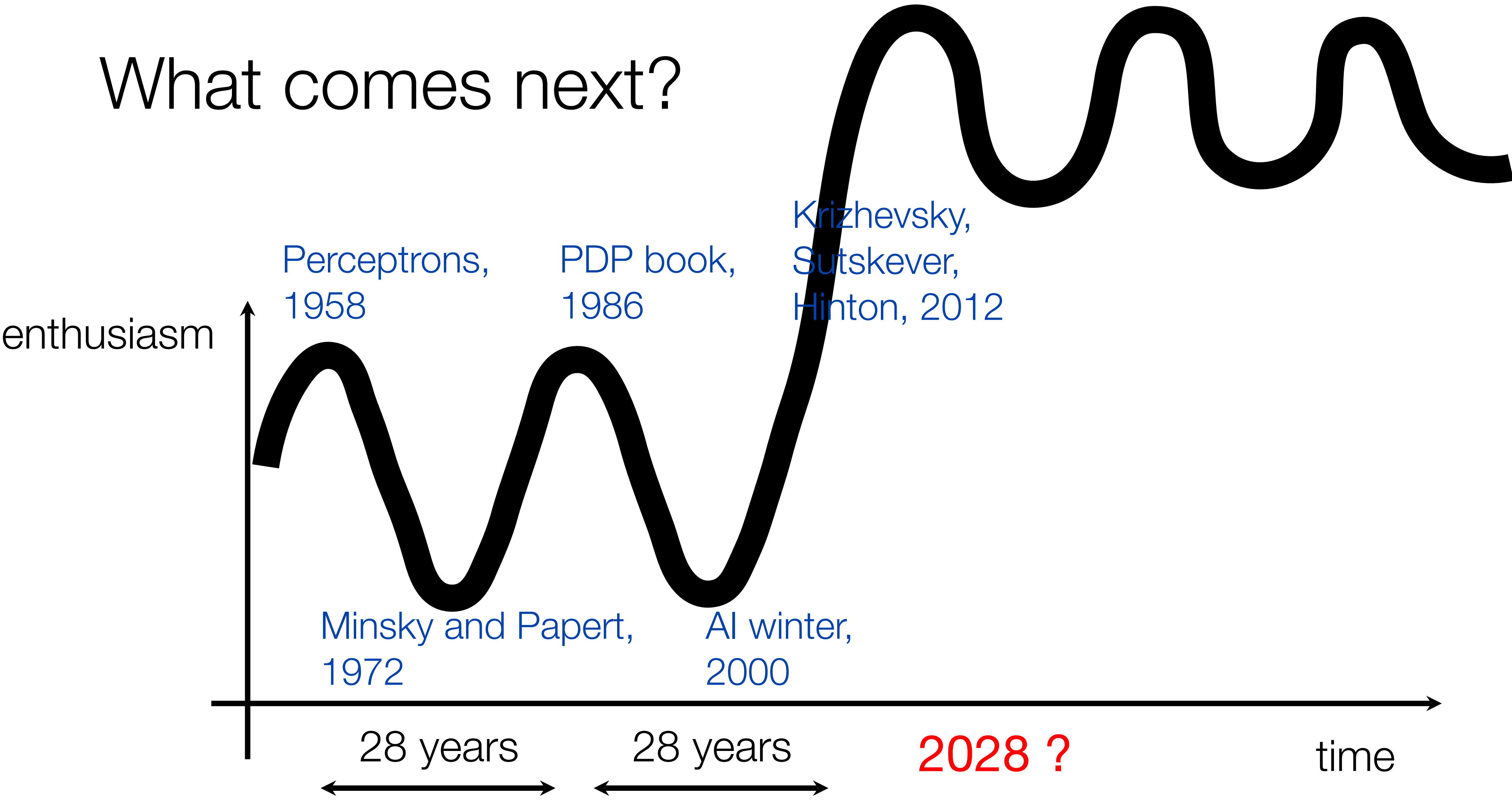




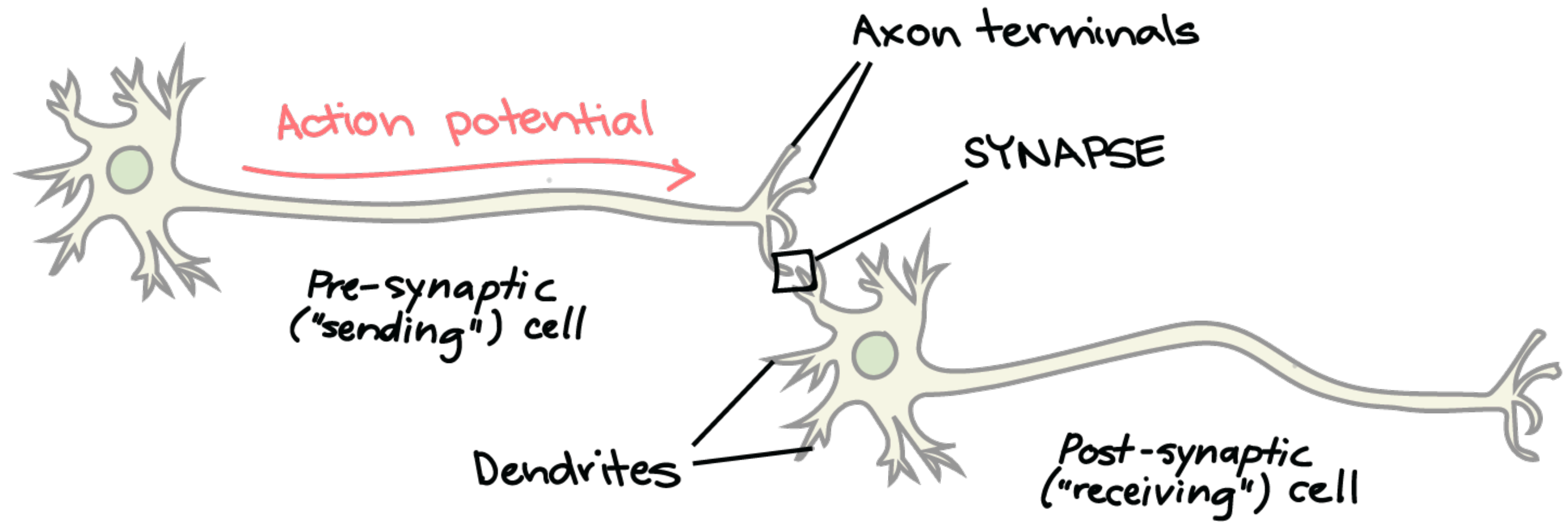
# What comes next?



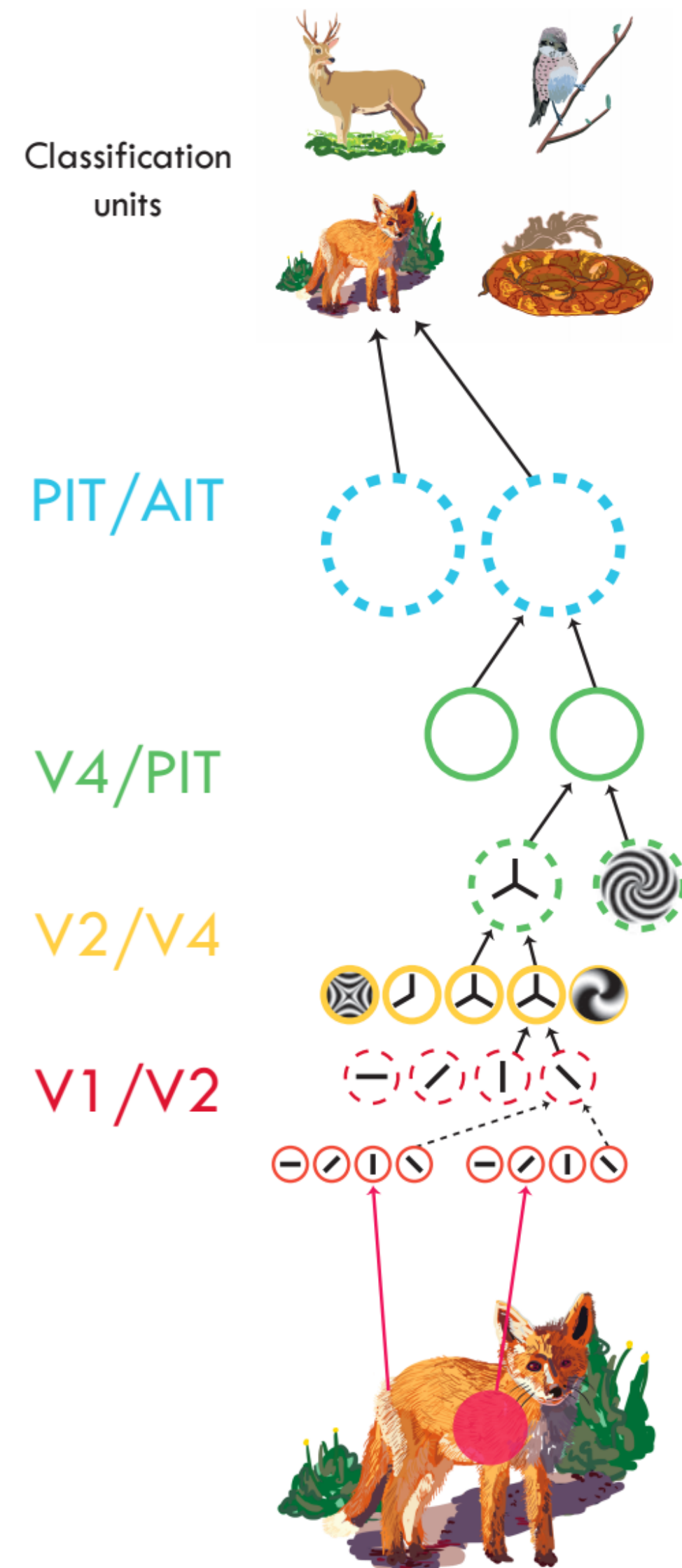
# What comes next?



# Inspiration: Neurons



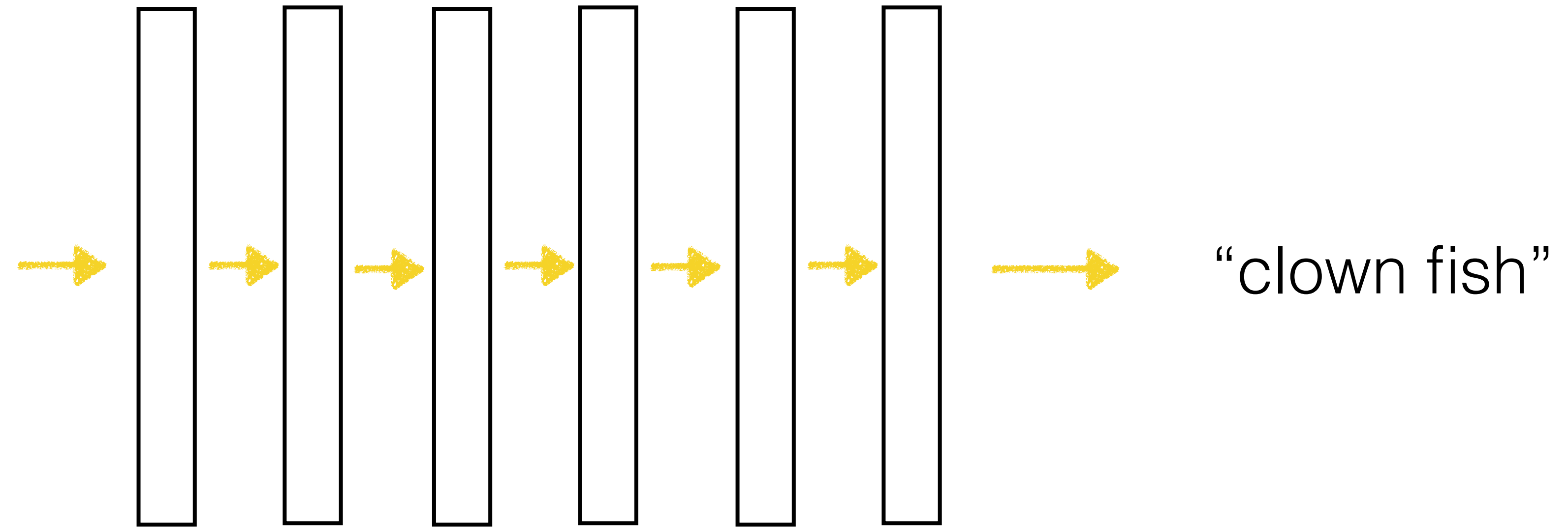
# Inspiration: Hierarchical Representations



Best to treat as *inspiration*. The neural nets we'll talk about aren't very biologically plausible.

[Serre, 2014]

# Object recognition

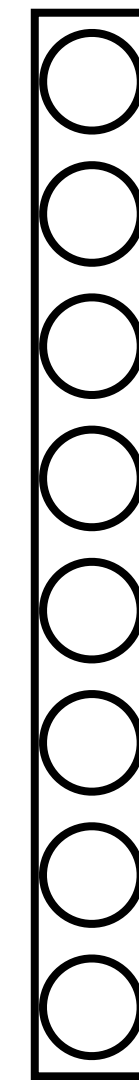
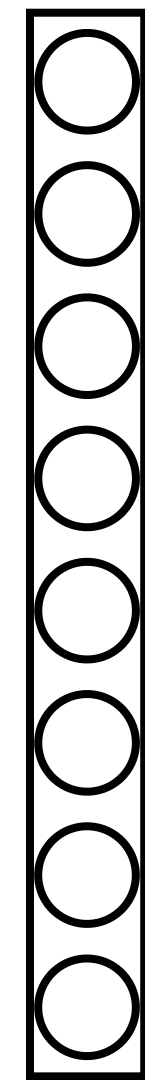


Neural network

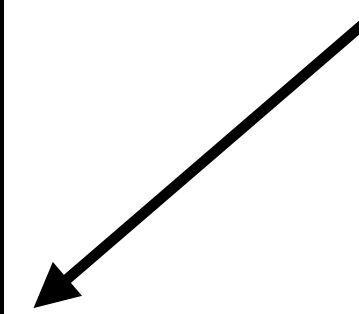
# Computation in a neural net

Input vector

Output vector

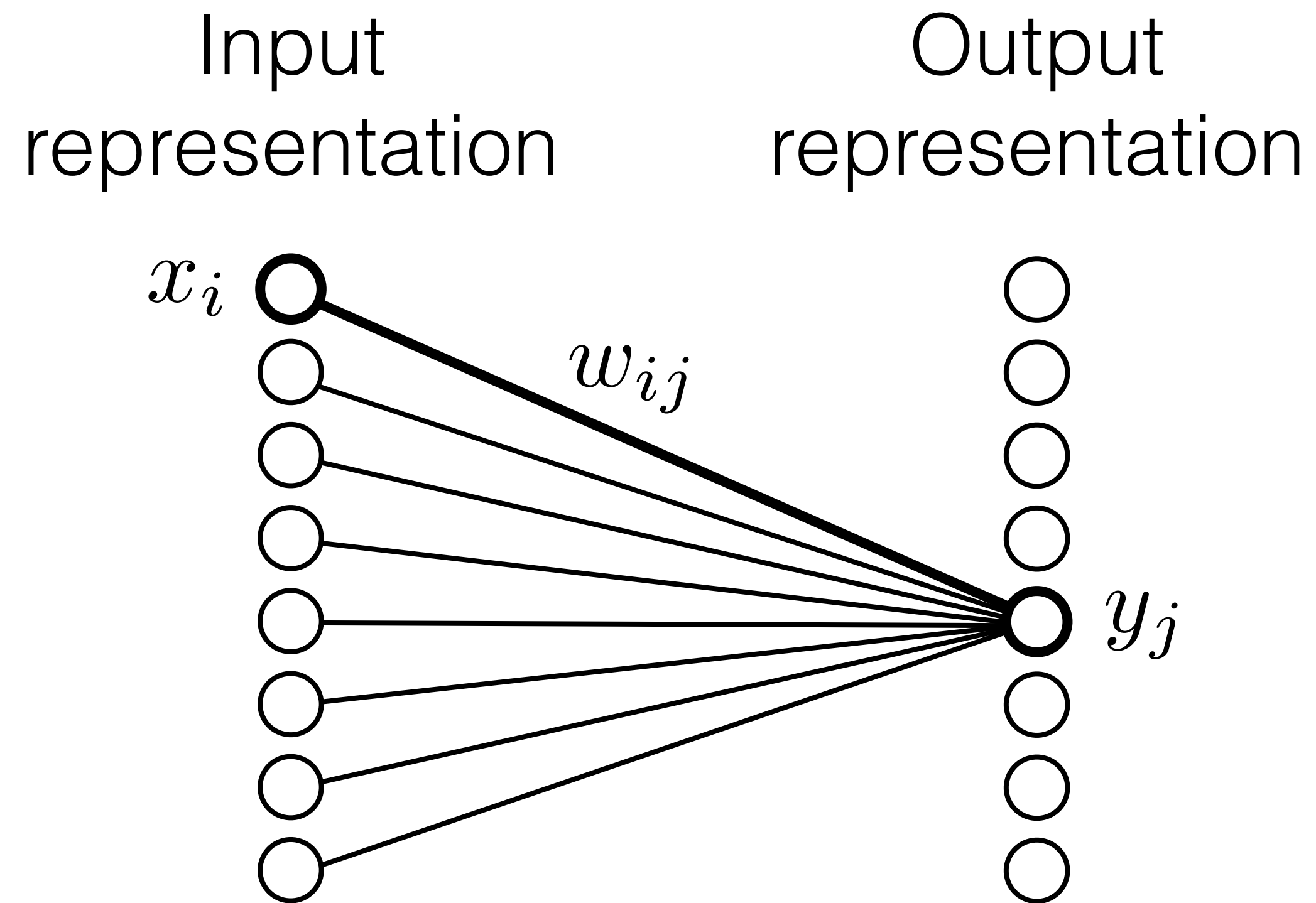


Neuron  
(a.k.a unit)



# Computation in a neural net

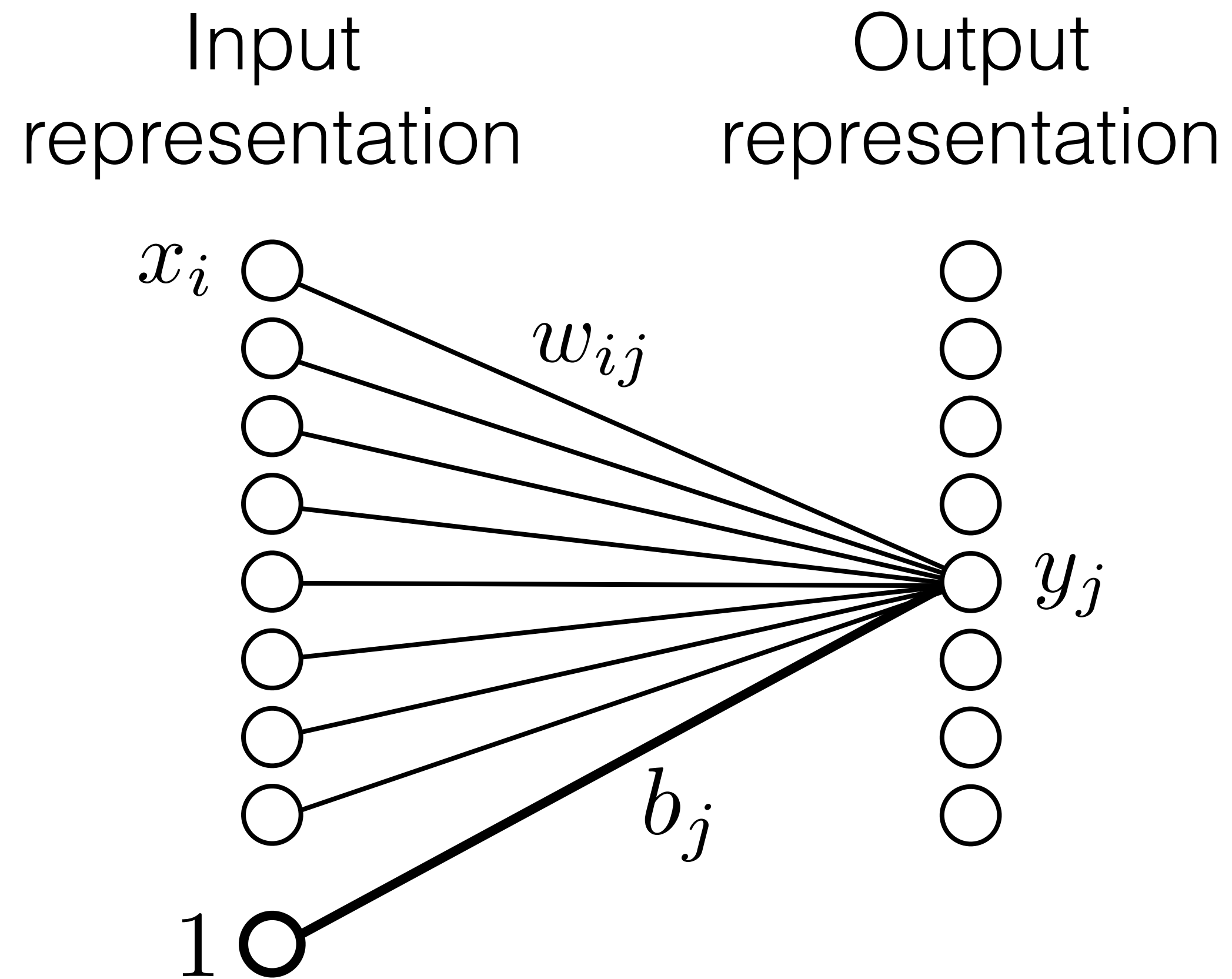
## Linear layer



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

# Computation in a neural net

## Linear layer



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

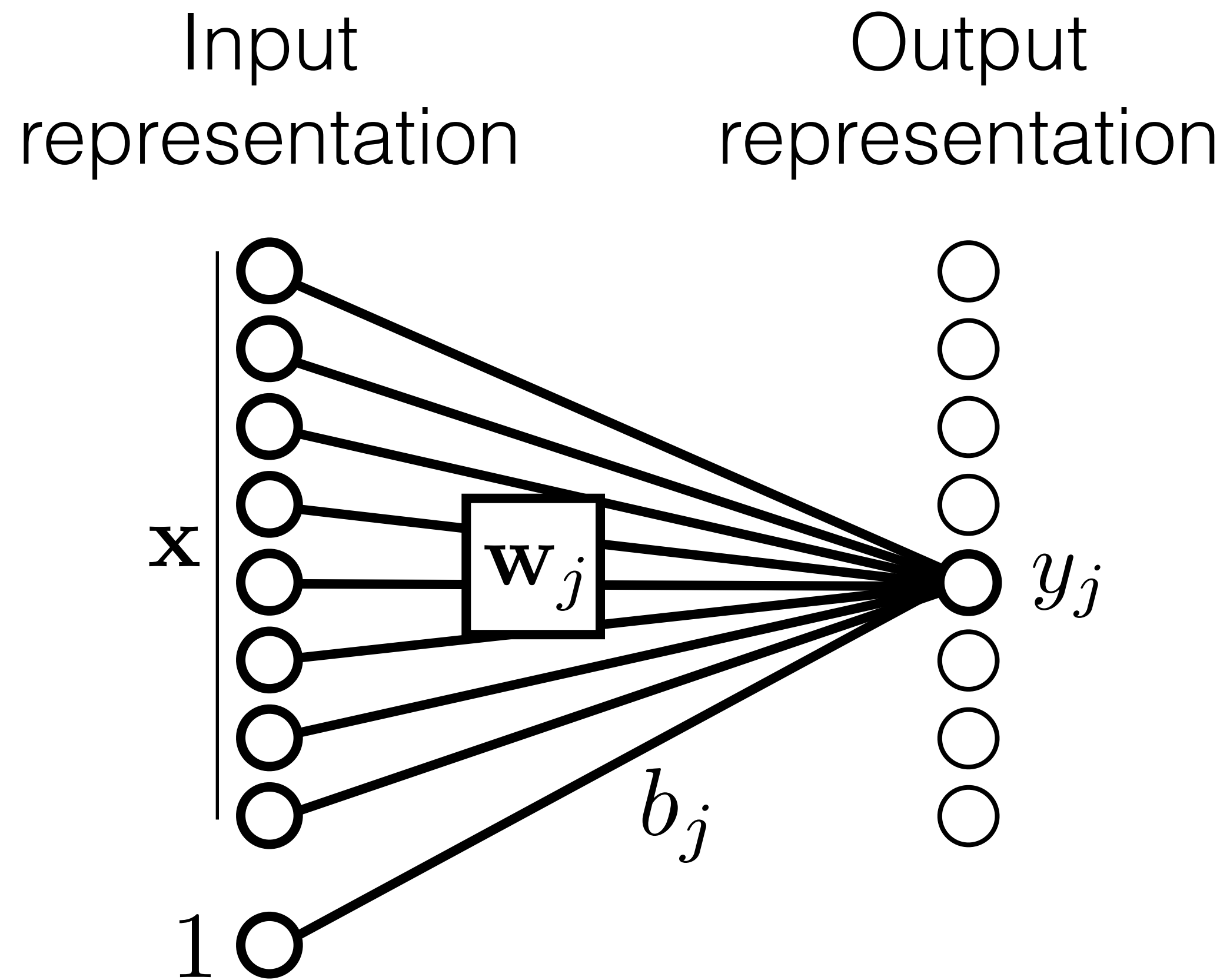
**weights**

**bias**



# Computation in a neural net

## Linear layer



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

weights

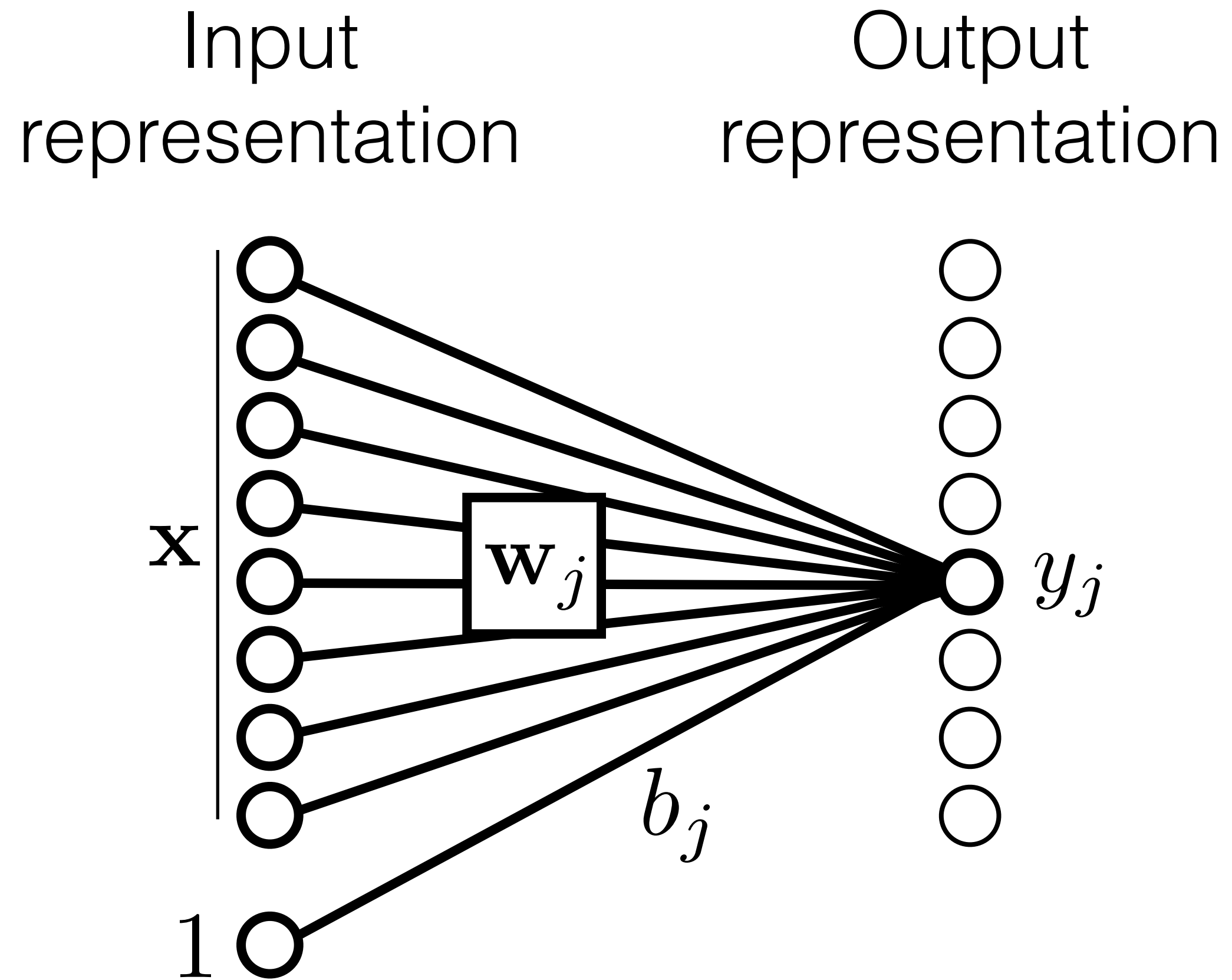
bias

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

parameters of the model

# Computation in a neural net

## Linear layer



## Full layer

weights

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

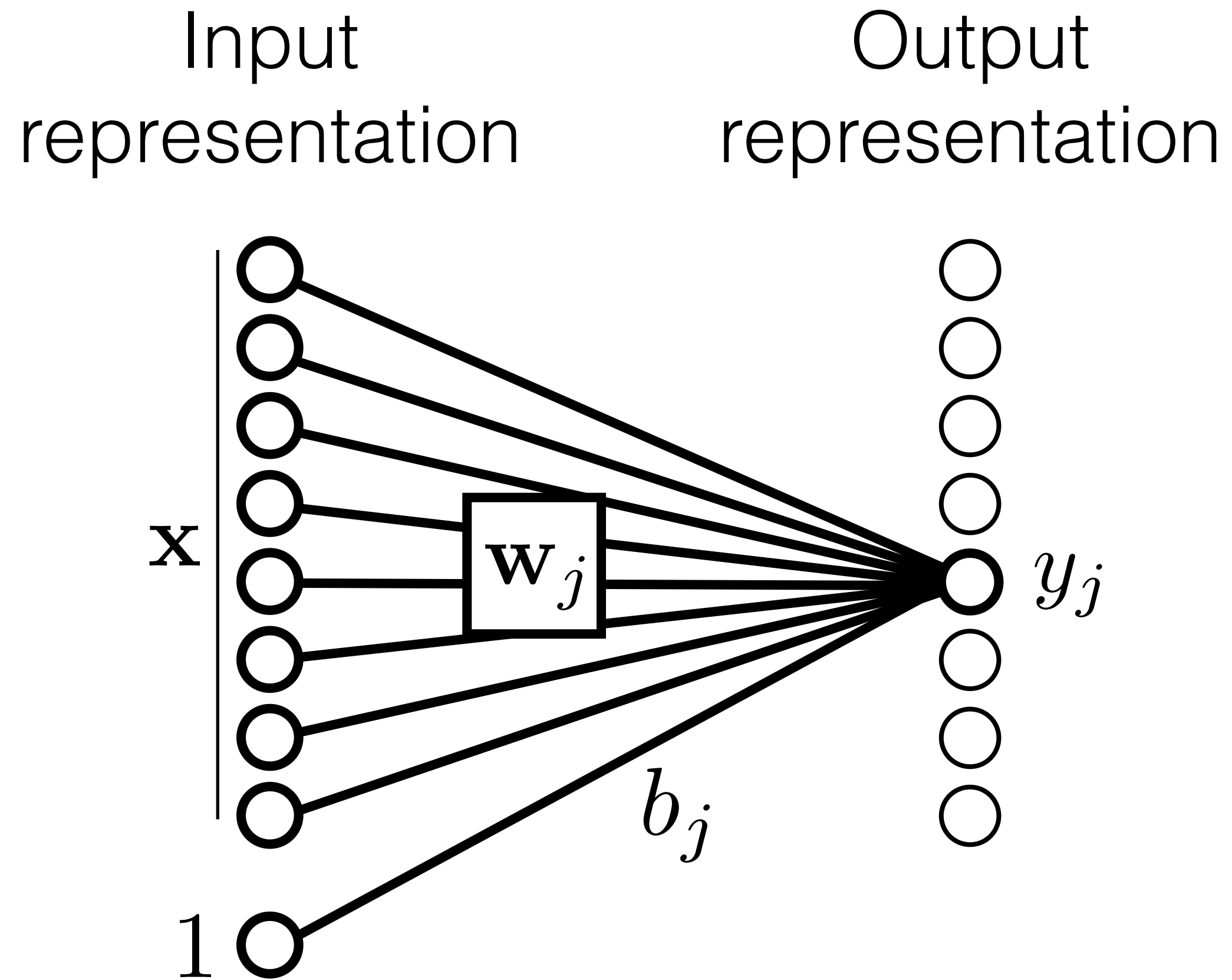
bias

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

parameters of the model

# Computation in a neural net

## Linear layer



## Full layer

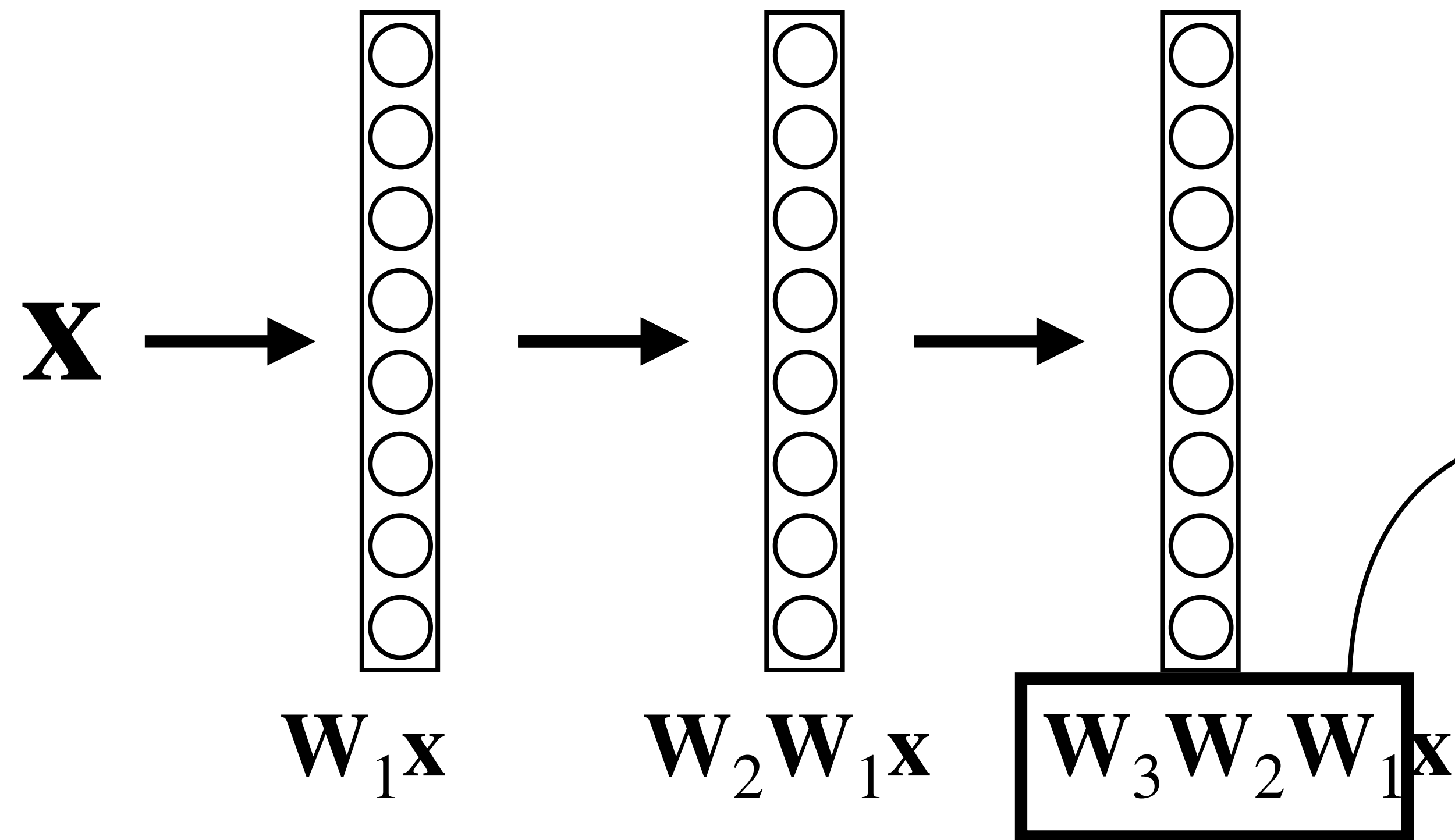
**weights + bias**

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

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

# What's the problem with this idea?

Consider stacking multiple layers:



Can be expressed as single linear layer!

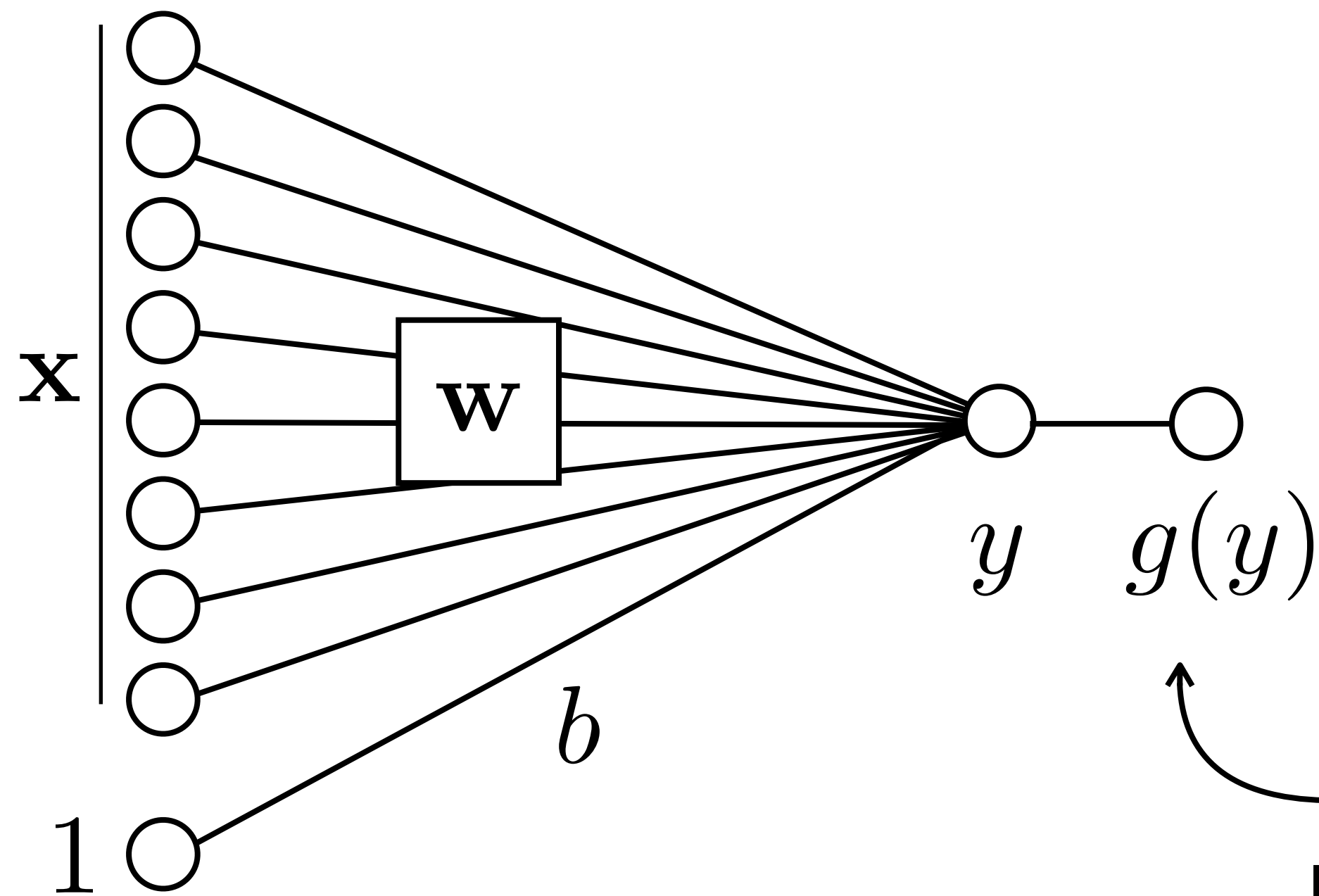
$$\left( \prod_i \mathbf{W}_i \right) \mathbf{x} = \hat{\mathbf{W}} \mathbf{x}$$

Limited power, e.g. can't solve XOR.

# Solution: simple nonlinearity

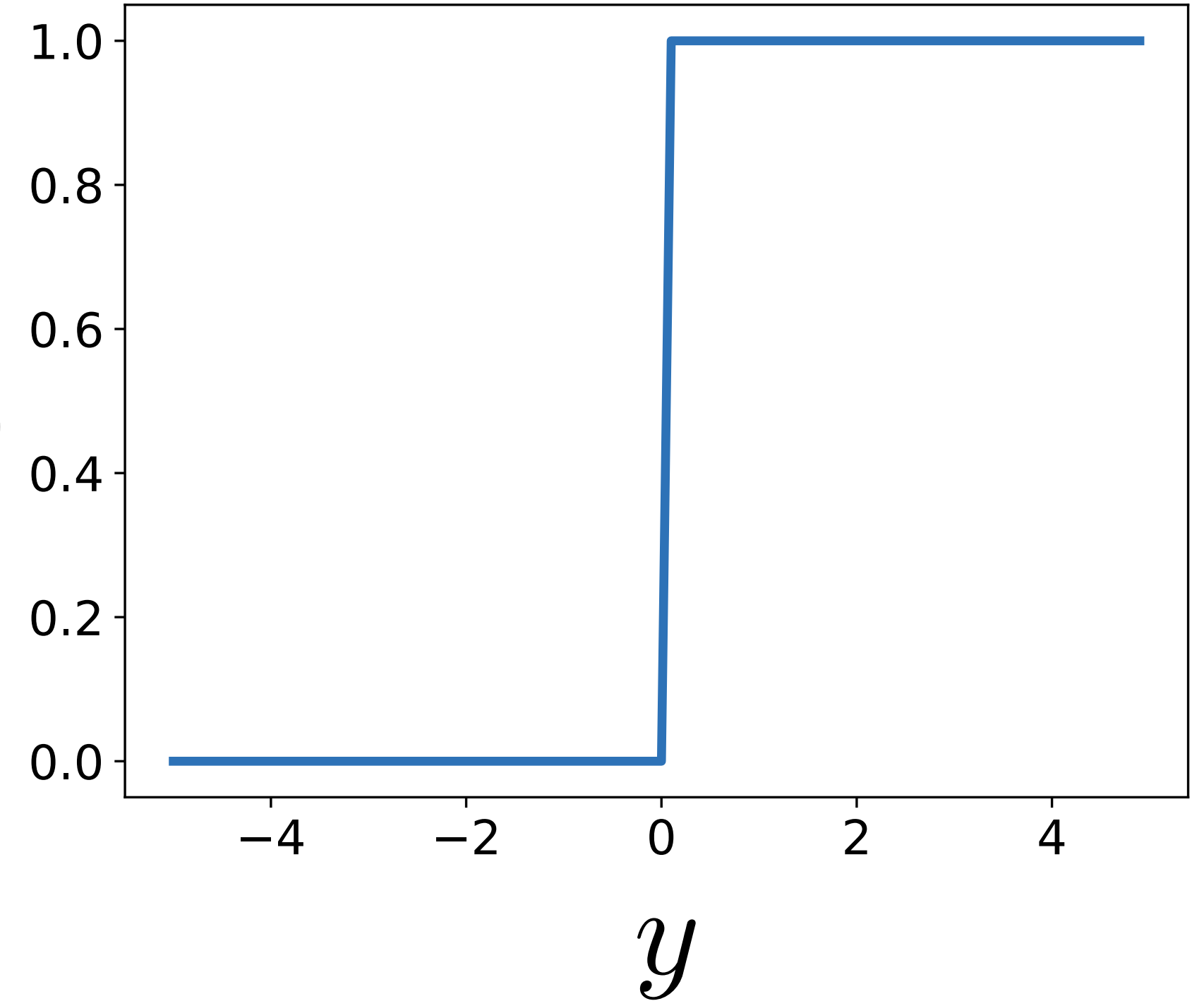
Input representation

Output representation



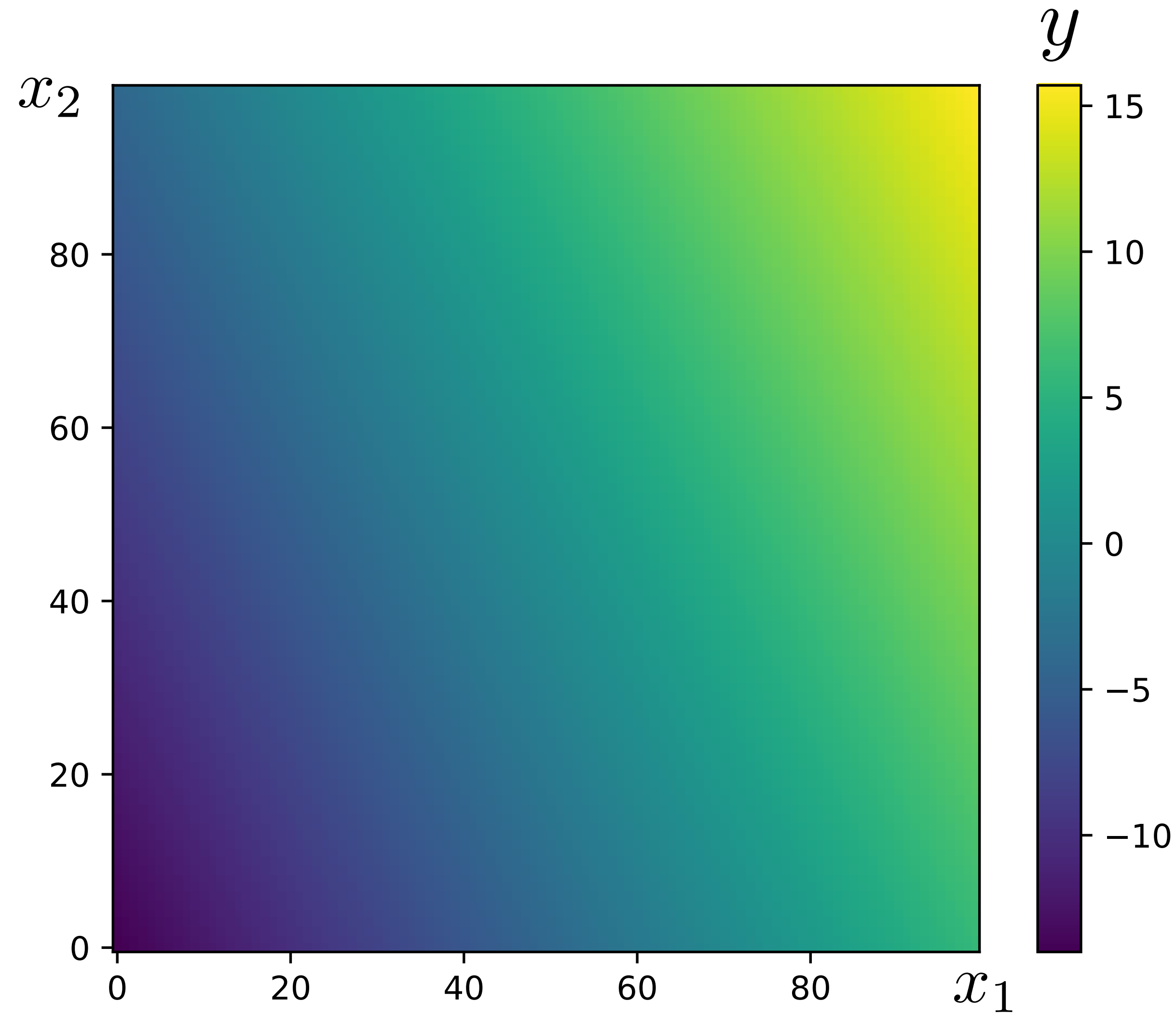
$$g(y) = \begin{cases} 1, & \text{if } y > 0 \\ 0, & \text{otherwise} \end{cases}$$

$g(y)$



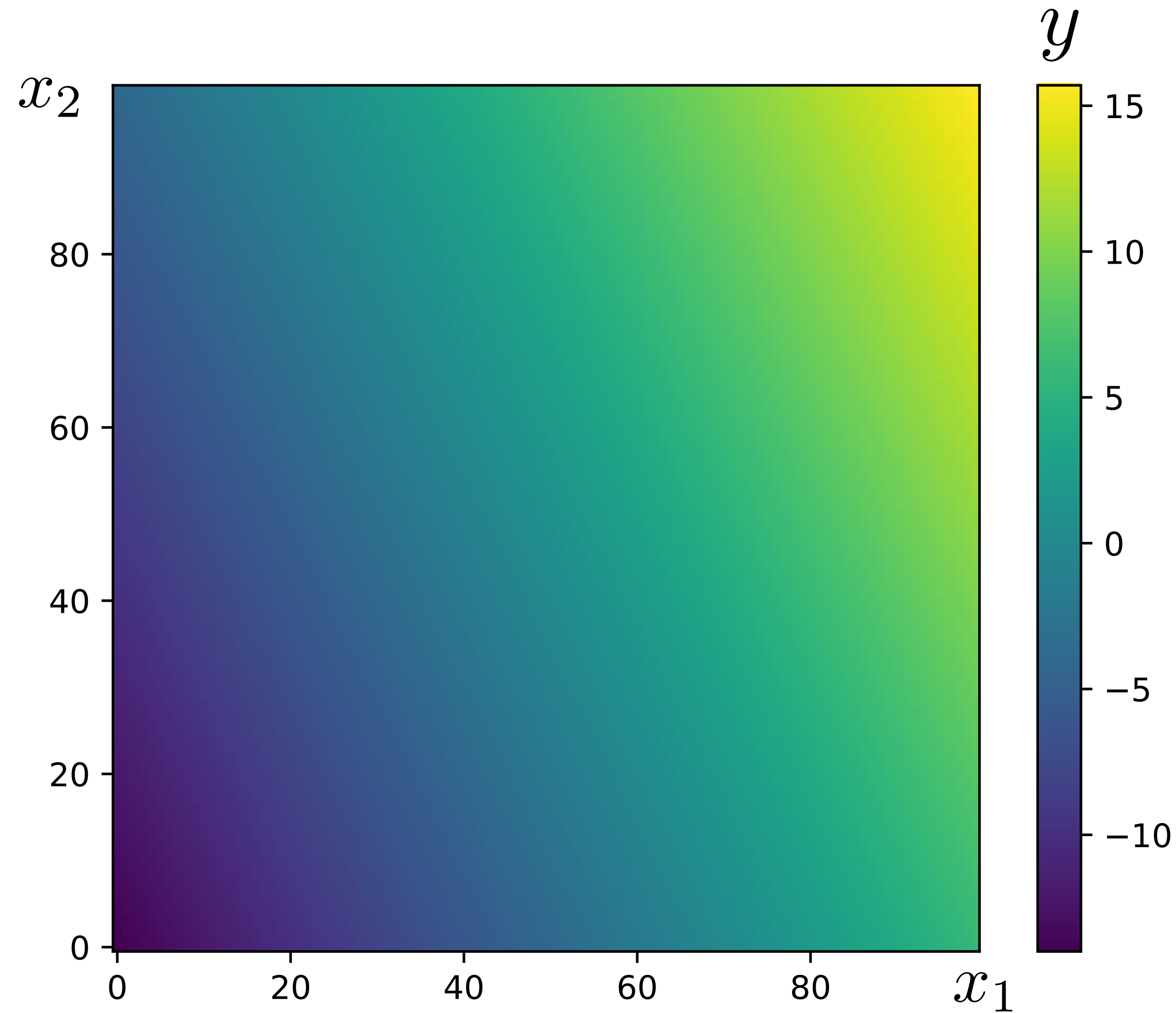
**Pointwise  
Non-linearity**

# Example: linear classification with a perceptron



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

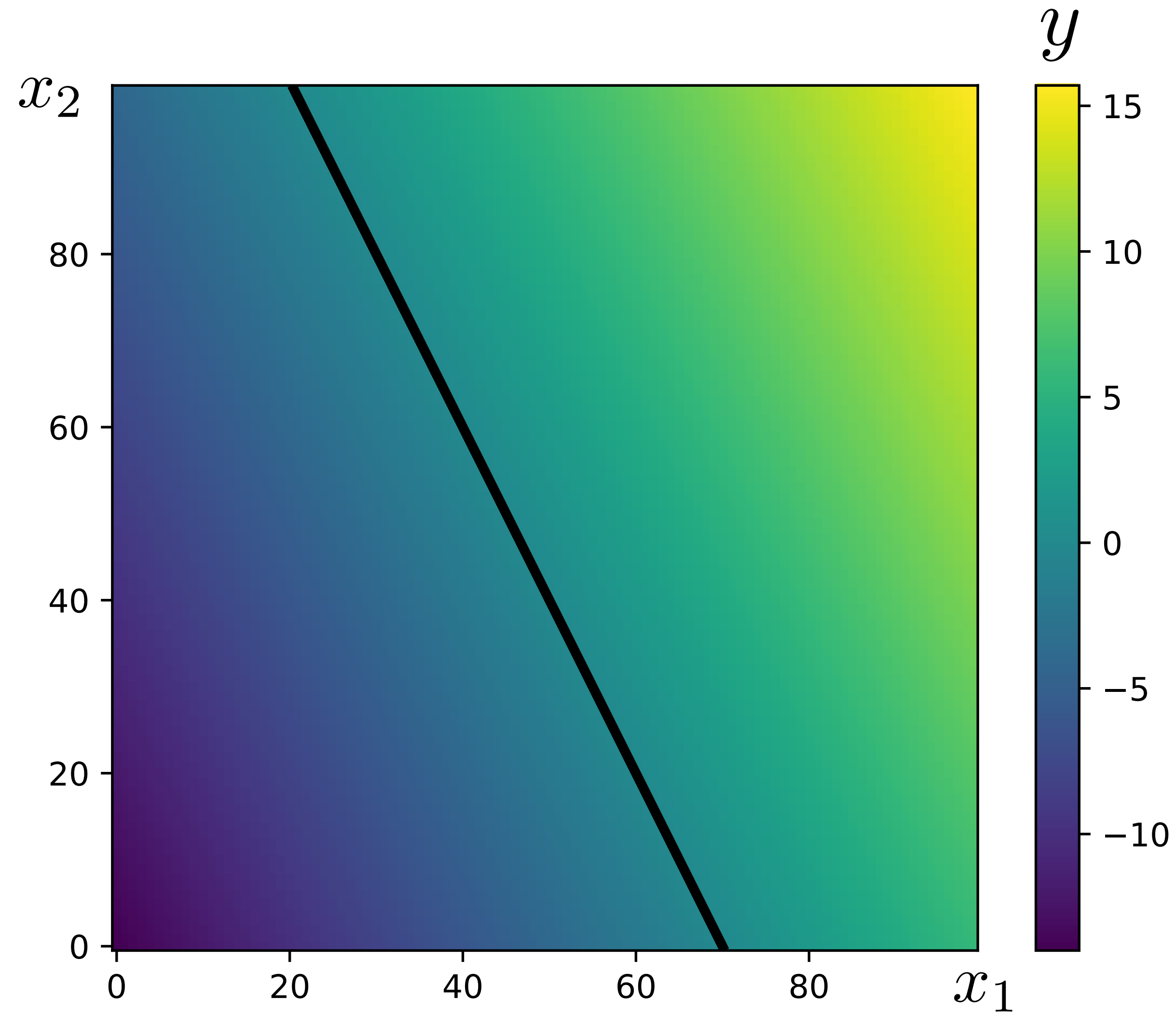
# Example: linear classification with a perceptron



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

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

# Example: linear classification with a perceptron



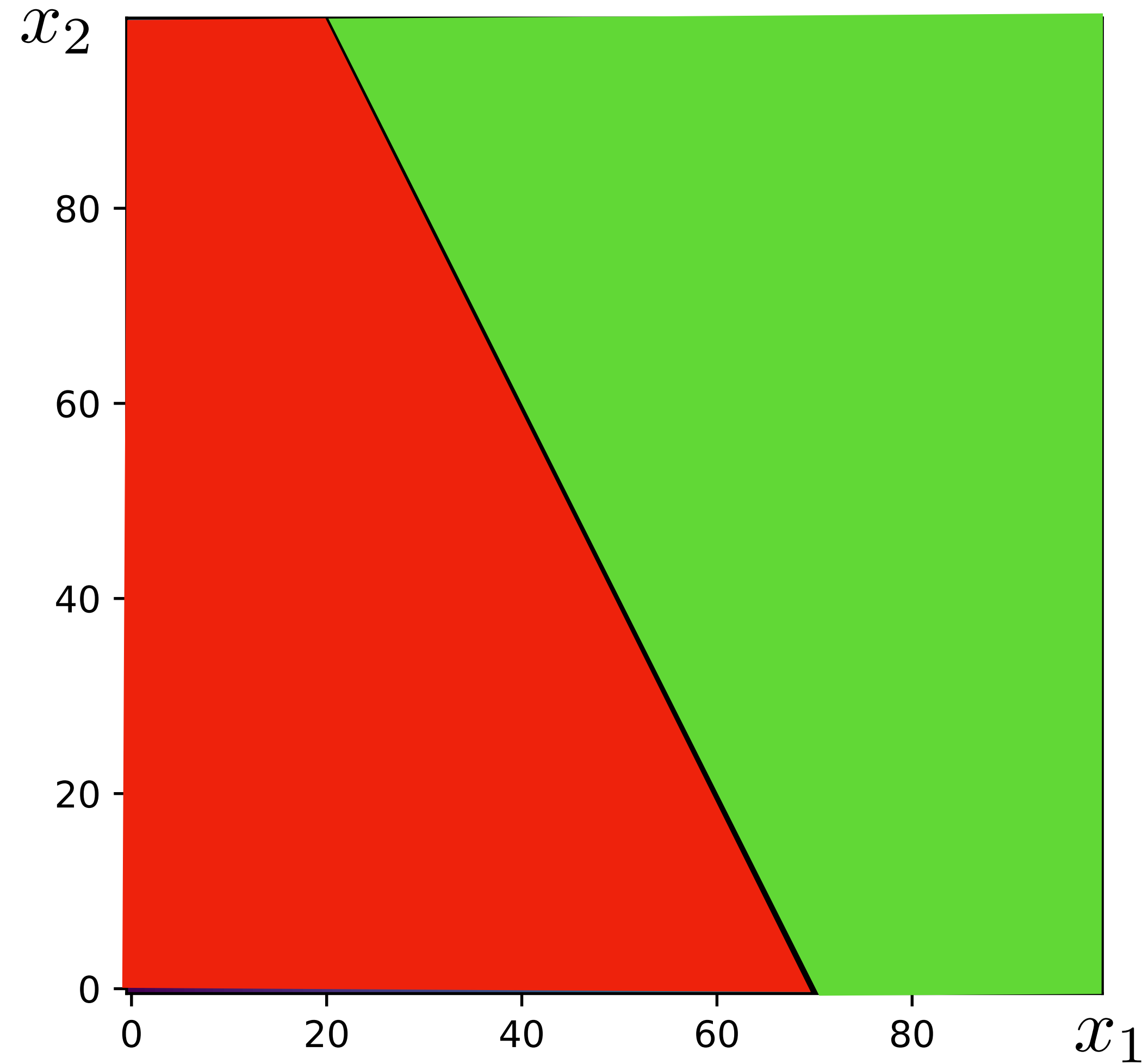
$$y = \mathbf{x}^T \mathbf{w} + b$$

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



# Example: linear classification with a perceptron

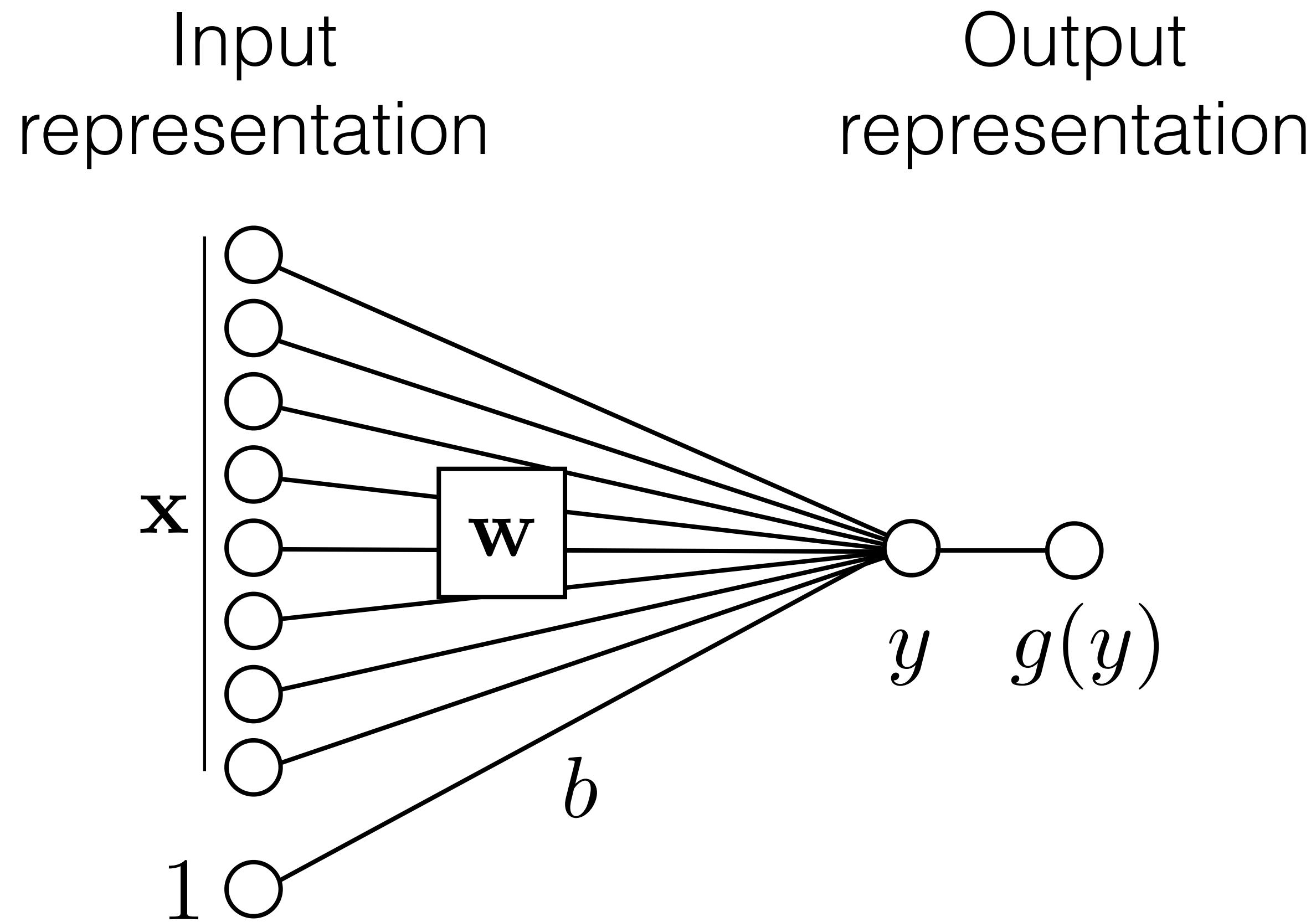
$$g(y)$$



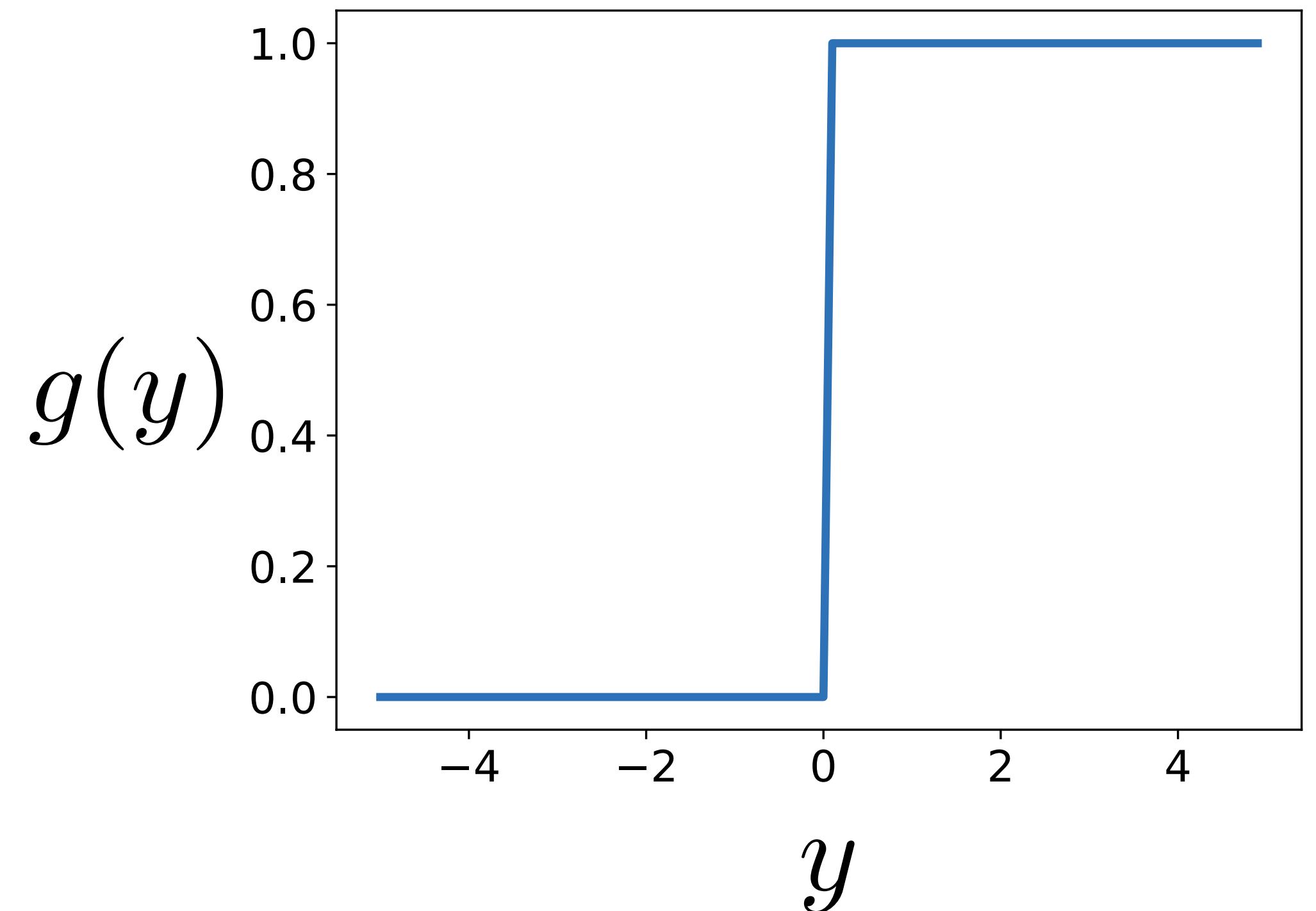
$$y = \mathbf{x}^T \mathbf{w} + b$$

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

# Computation in a neural net — nonlinearity

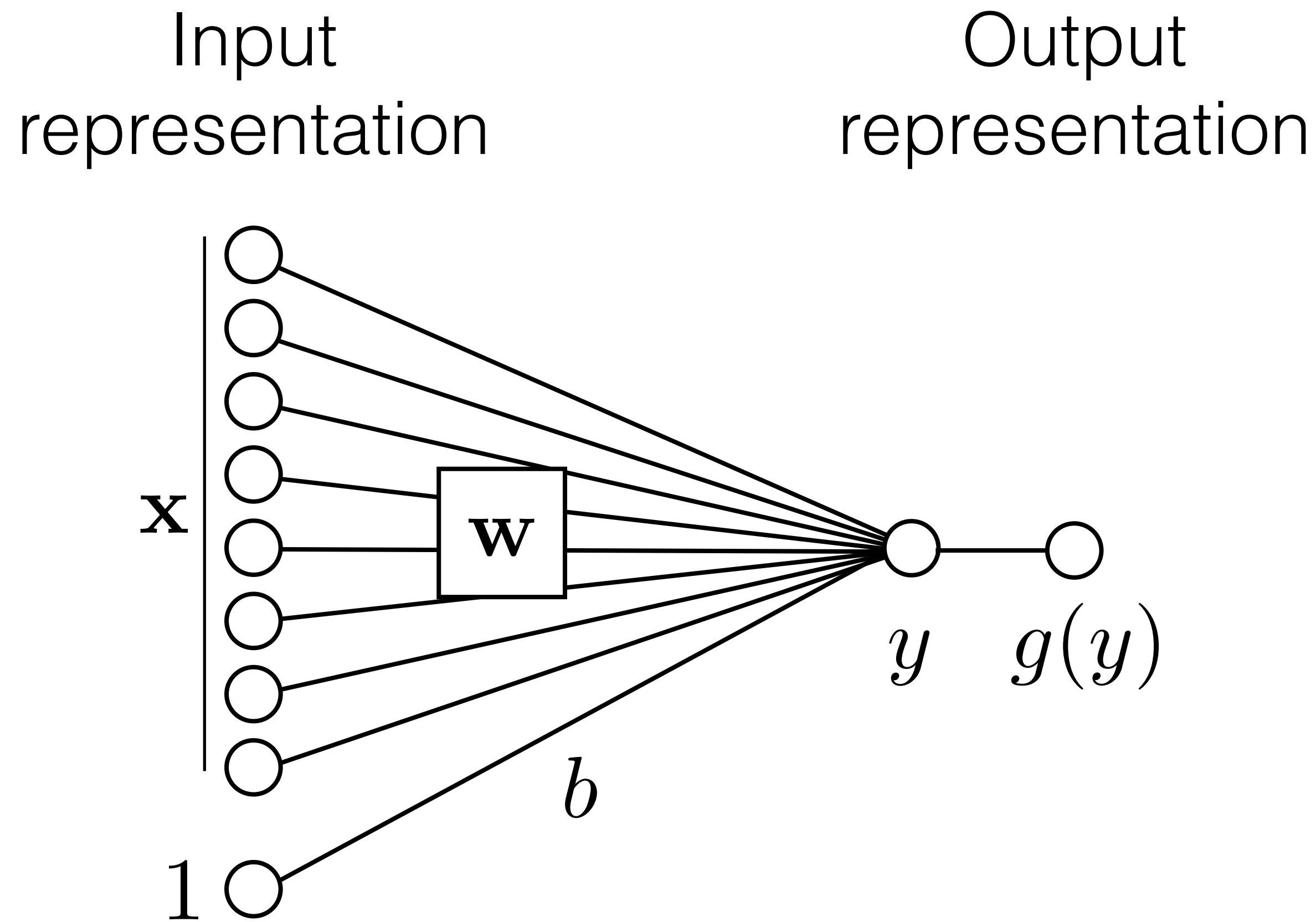


$$g(y) = \begin{cases} 1, & \text{if } y > 0 \\ 0, & \text{otherwise} \end{cases}$$



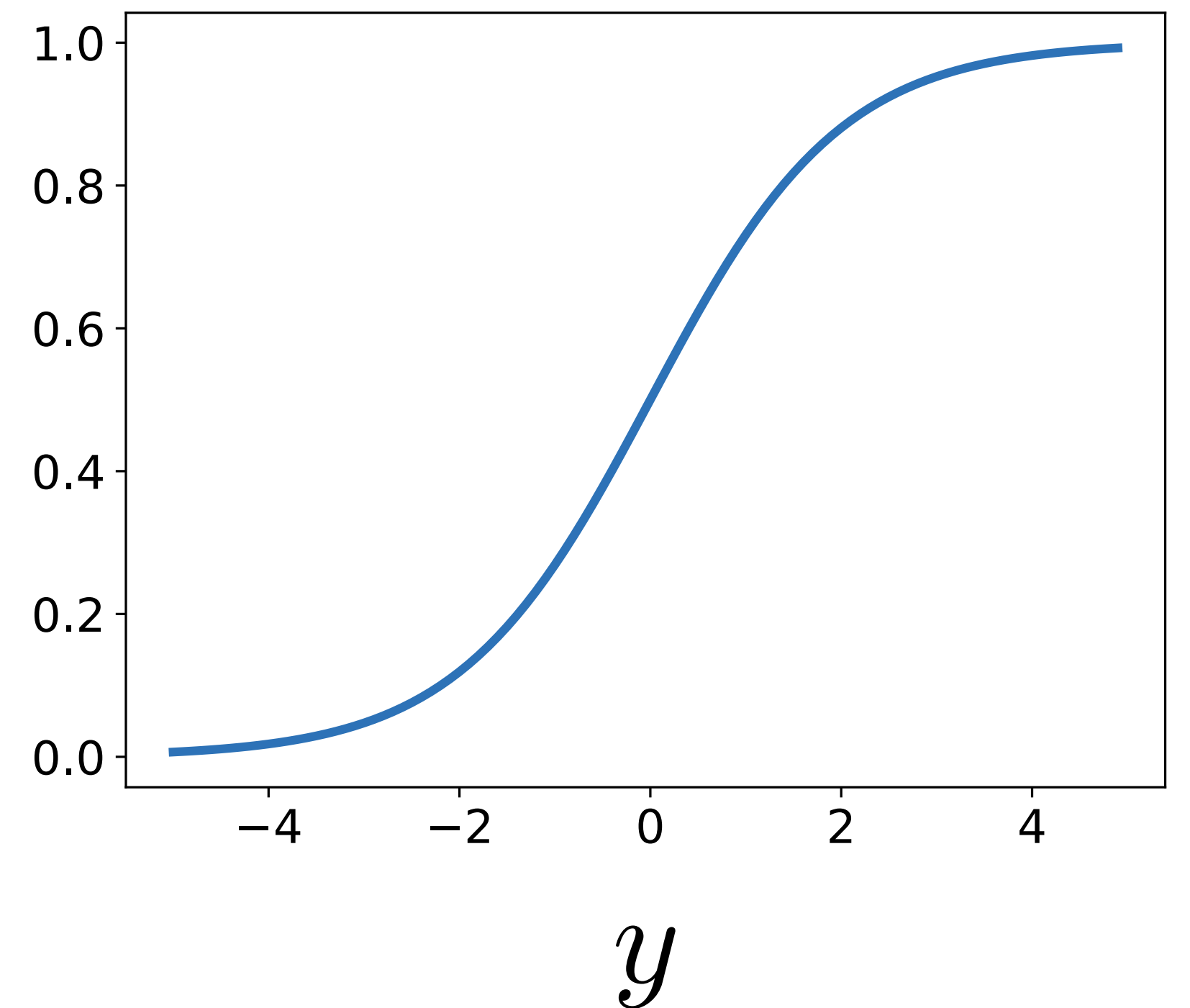
Can't use with gradient descent,  $\nabla g = 0$

# Computation in a neural net — nonlinearity



## Sigmoid

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

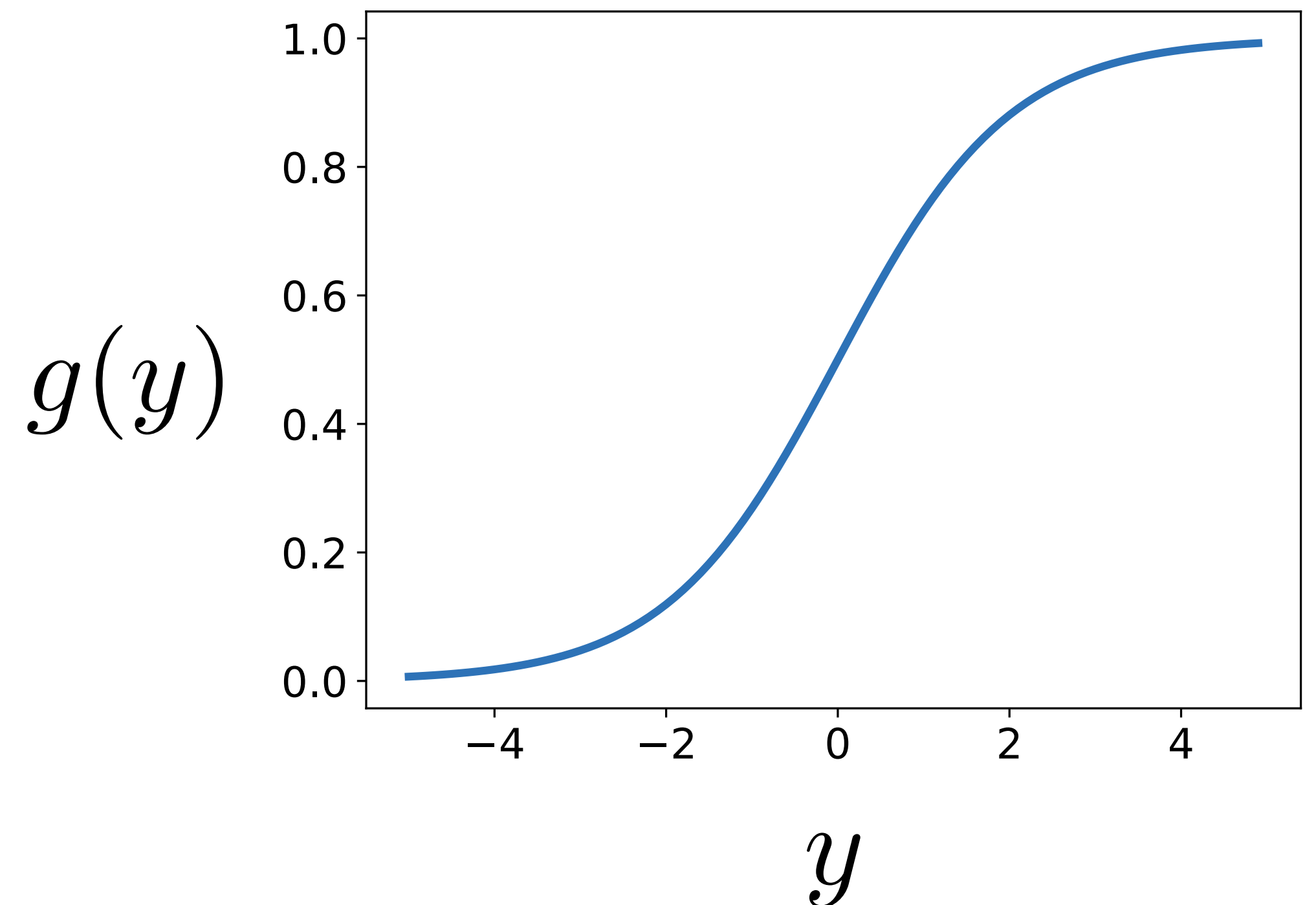


# Computation in a neural net — nonlinearity

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

## Sigmoid

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



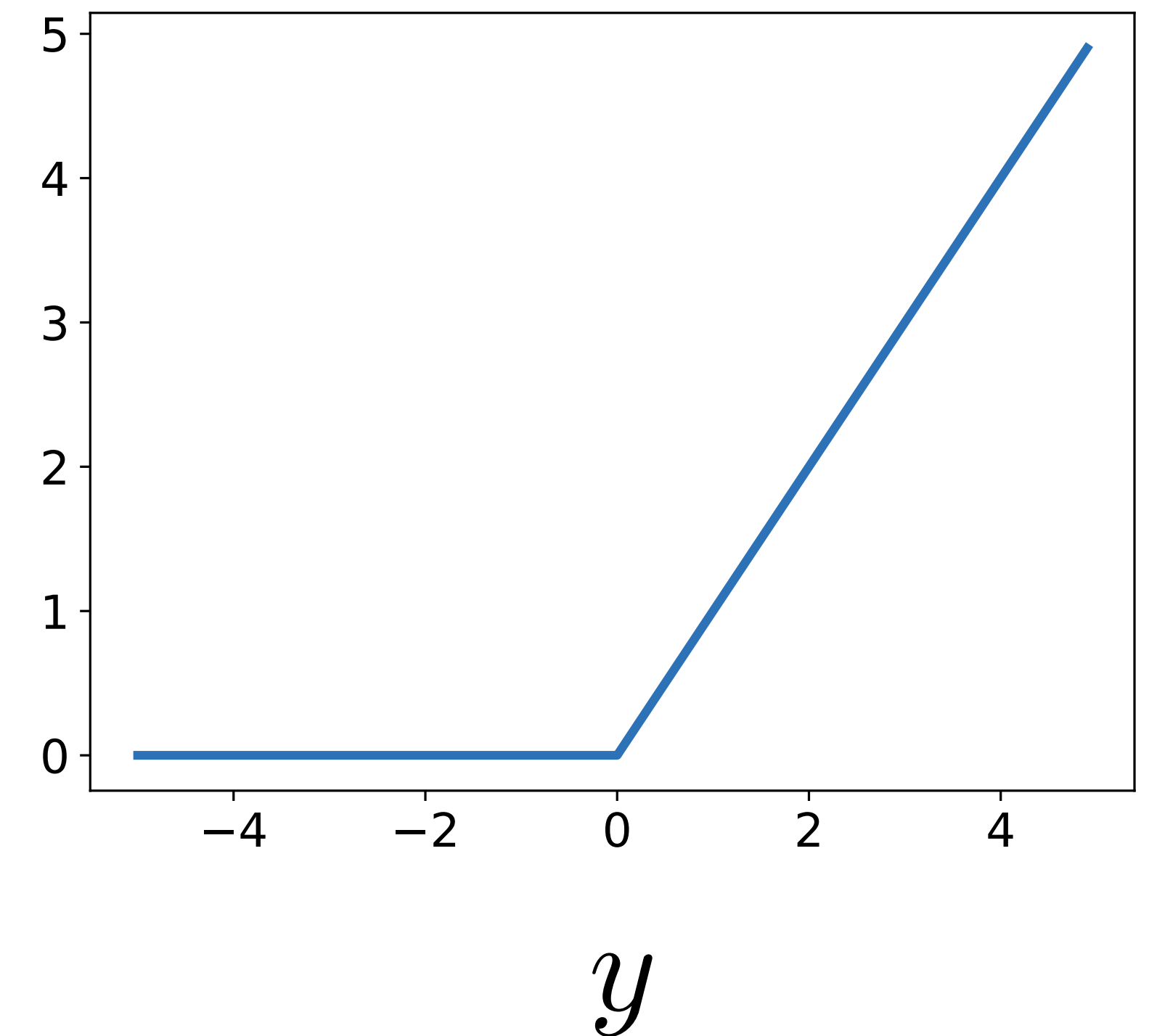
# Computation in a neural net — nonlinearity

- 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)

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

$g(y)$



# Computation in a neural net — nonlinearity

- where  $a$  is small (e.g. 0.02)

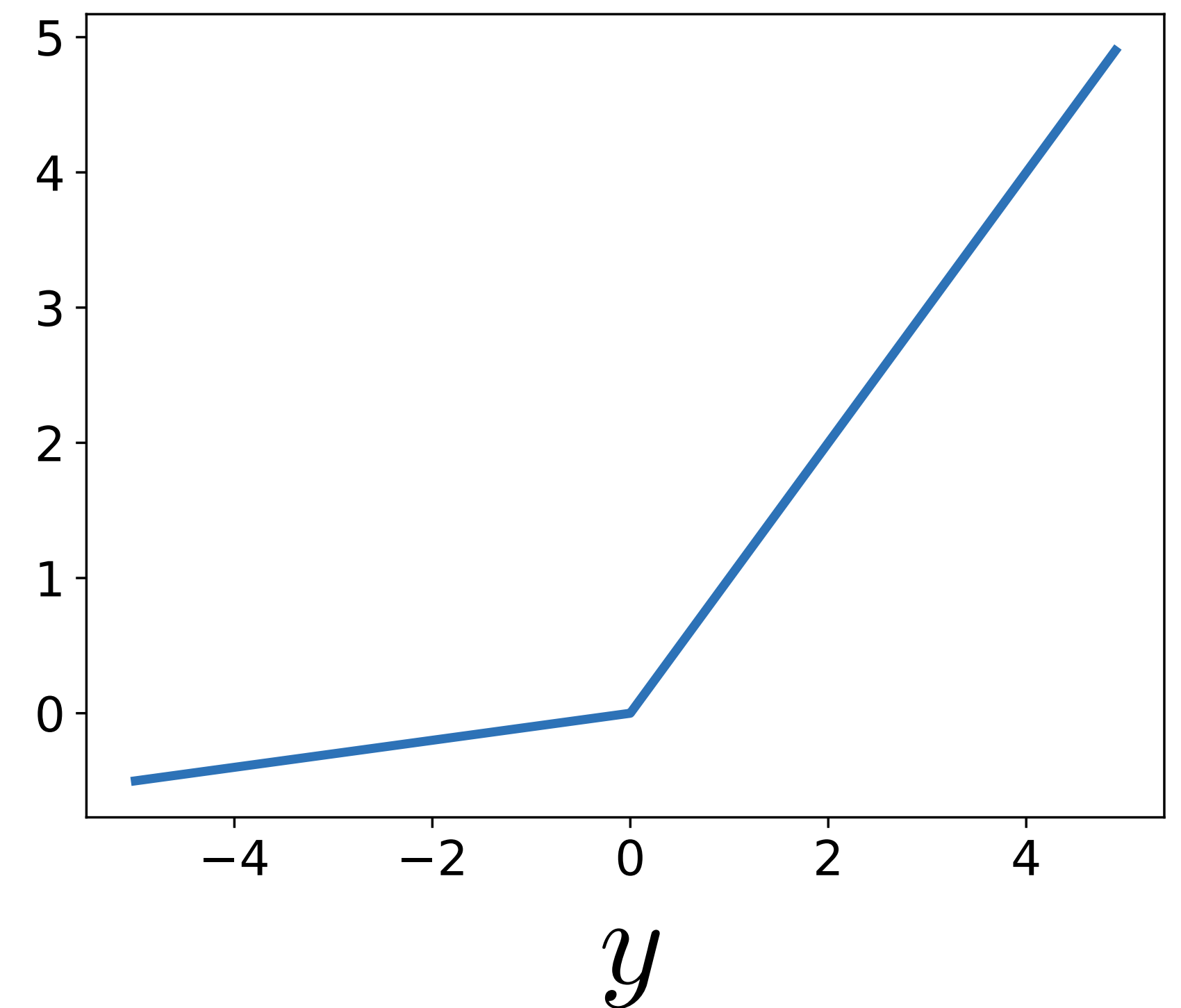
- Efficient to implement:  $\frac{\partial g}{\partial y} = \begin{cases} -a, & \text{if } y < 0 \\ 1, & \text{if } y \geq 0 \end{cases}$

- Has non-zero gradients everywhere (unlike ReLU)

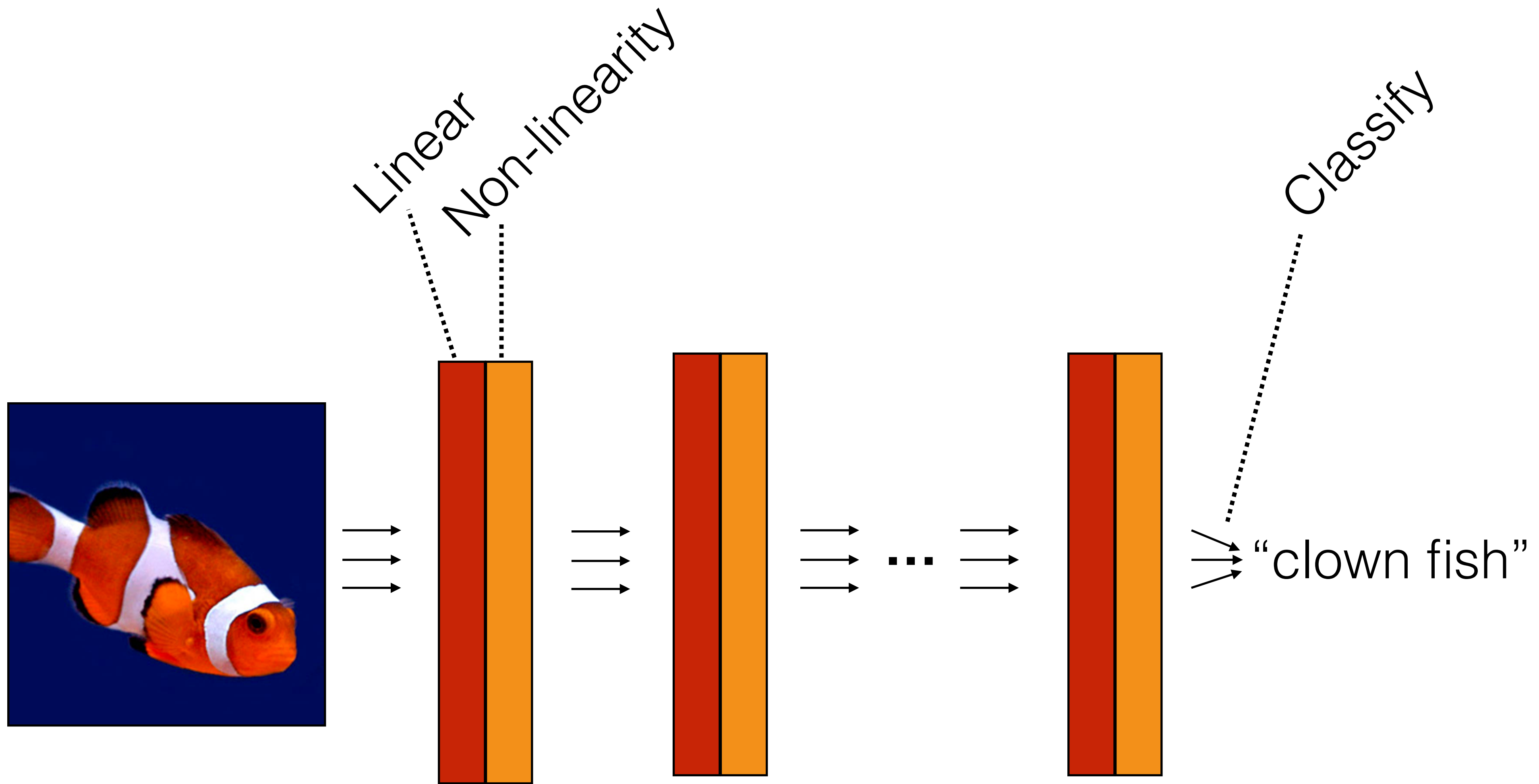
## Leaky ReLU

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

$g(y)$



# Deep nets



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

Computation has a simple form

$$\mathbf{y} = \mathbf{W}^{(n)} g(\mathbf{W}^{(n-1)} \dots g(\mathbf{W}^{(3)} g(\mathbf{W}^{(2)} (g(\mathbf{W}^{(1)} \mathbf{x}))))))$$

- Composition of linear functions with nonlinearities in between
- E.g. matrix multiplications with ReLU,  $\max(0, \mathbf{x})$  afterwards
- Do a matrix multiplication, set all negative values to 0, repeat

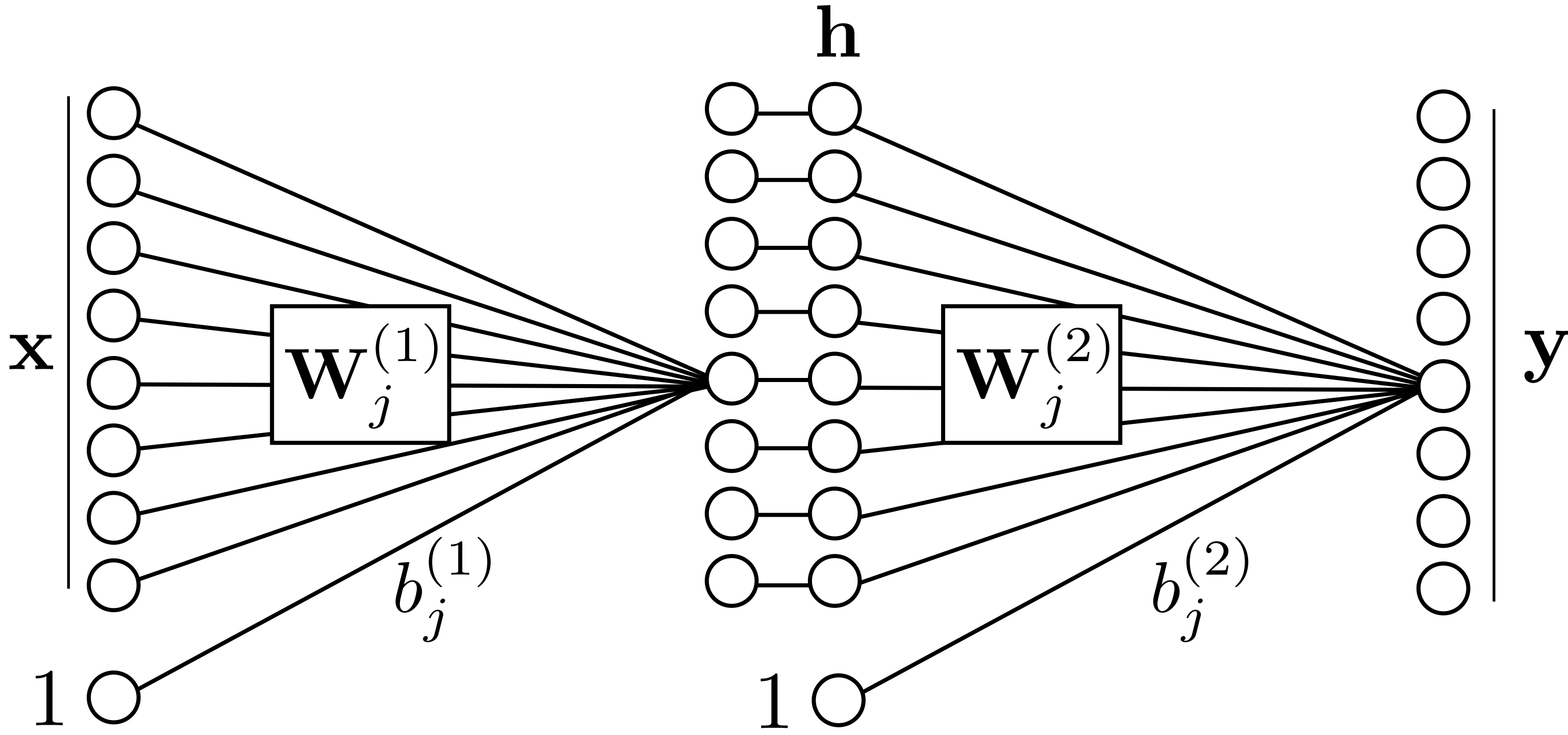


# Stacking layers

Input representation

Intermediate representation

Output representation



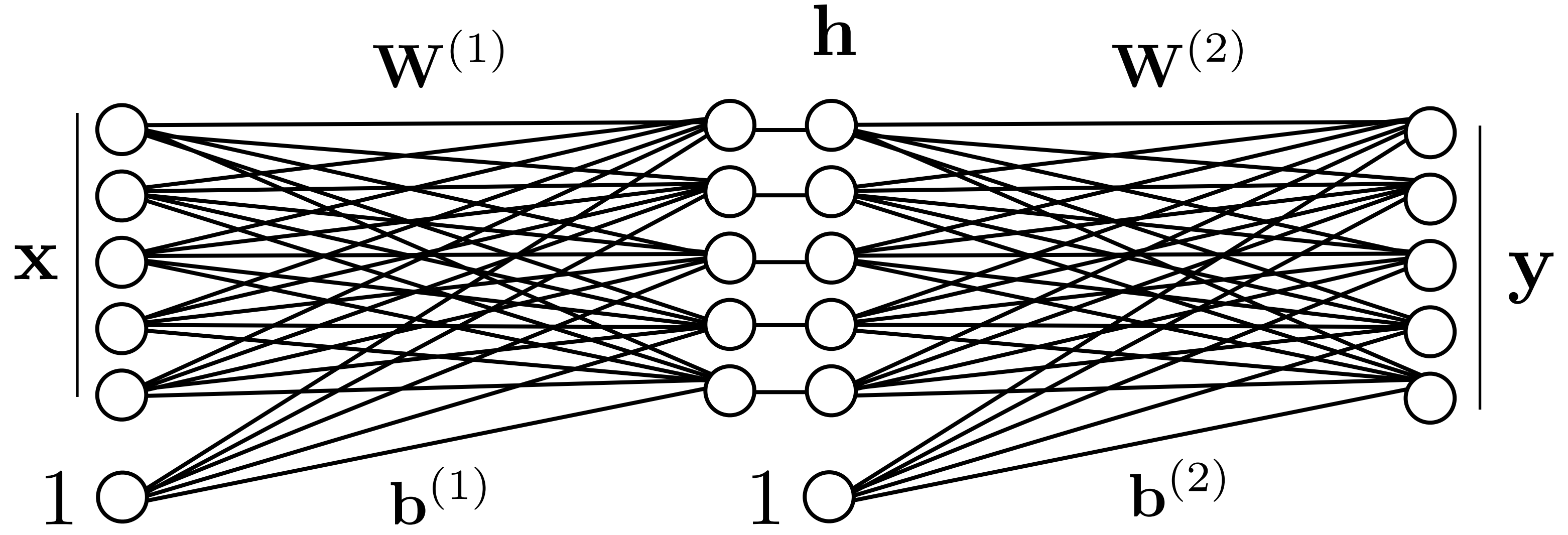
$\mathbf{h}$  = "hidden units"

# Stacking layers

Input representation

Intermediate representation

Output representation

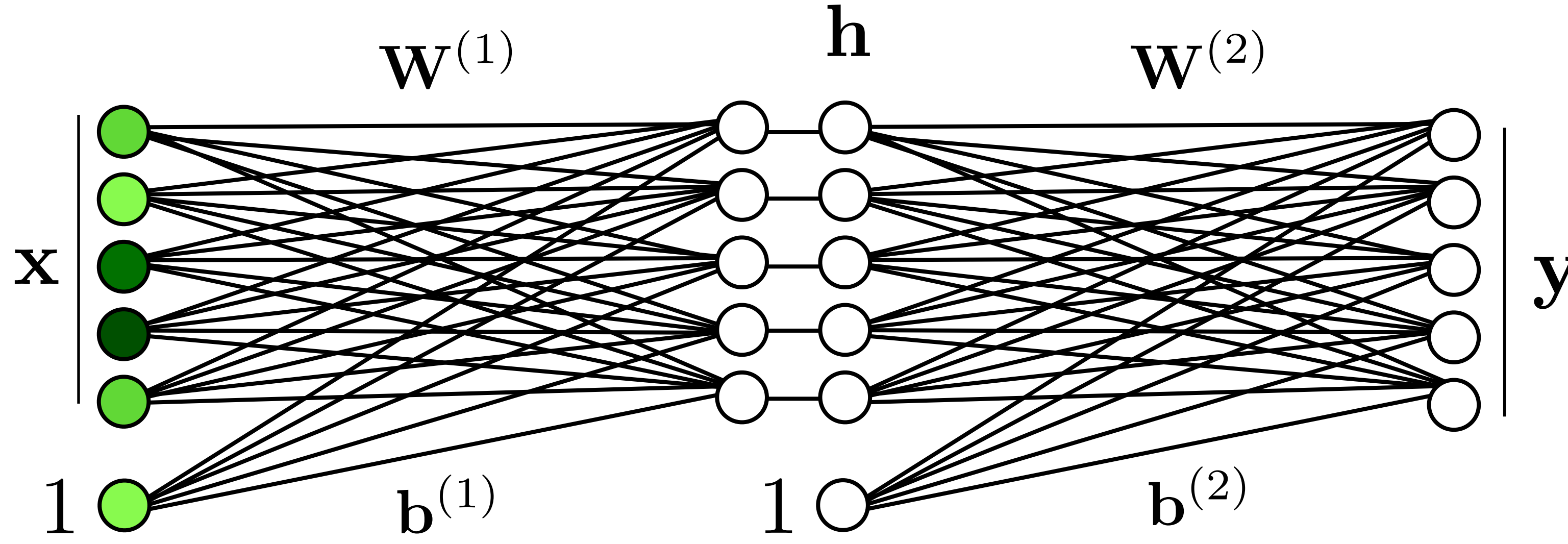


$$\mathbf{h} = g(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \quad \mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$$

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

# Stacking layers

Input representation      Intermediate representation      Output representation



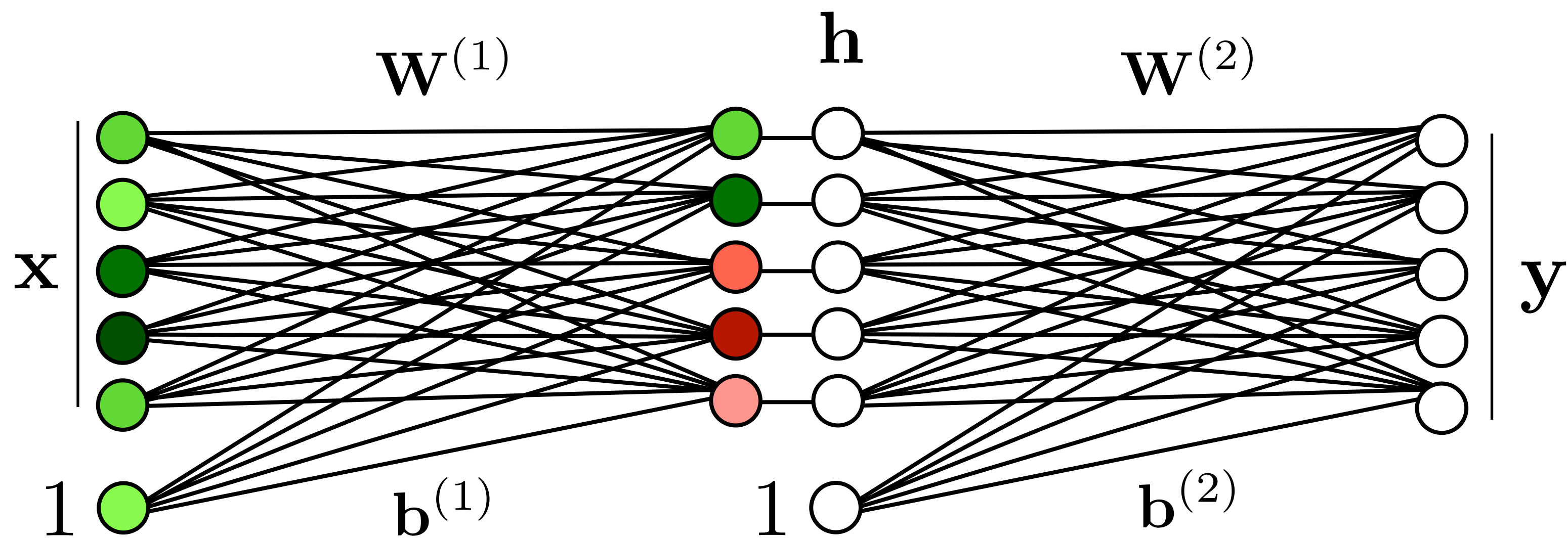
positive  
negative

$$\mathbf{h} = g(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \quad \mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$$

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

# Stacking layers

Input representation      Intermediate representation      Output representation



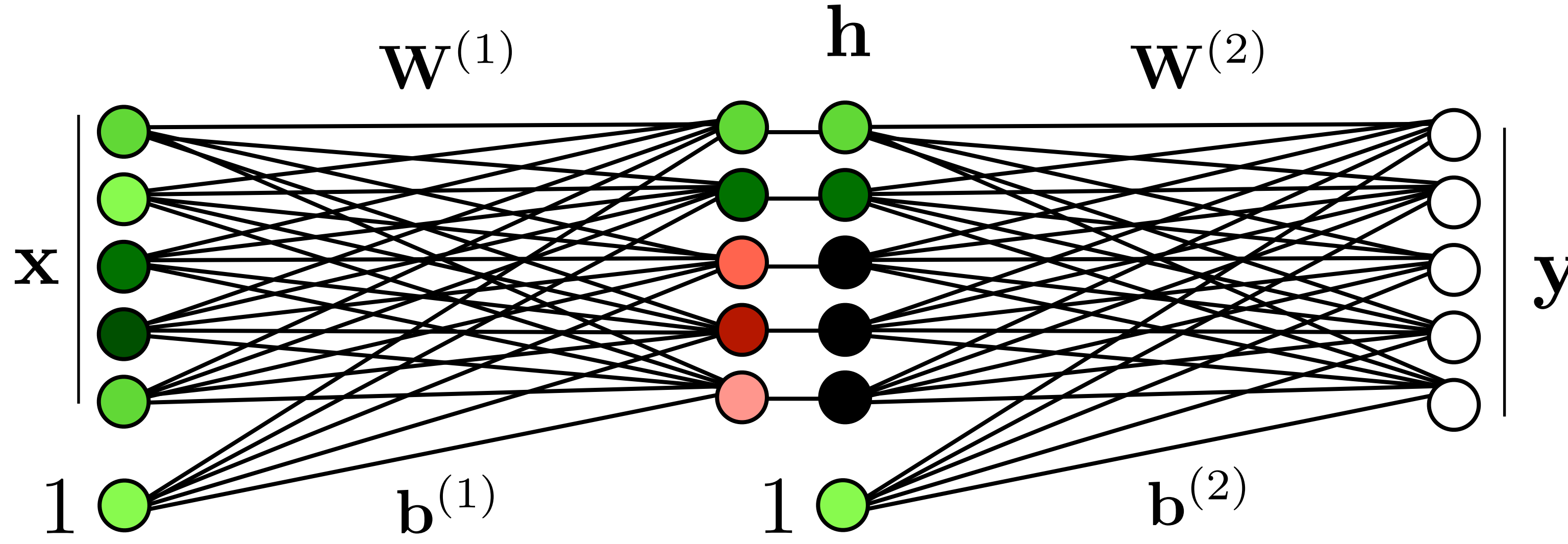
**positive**  
**negative**

$$\mathbf{h} = g(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \quad \mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$$

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

# Stacking layers

Input representation      Intermediate representation      Output representation



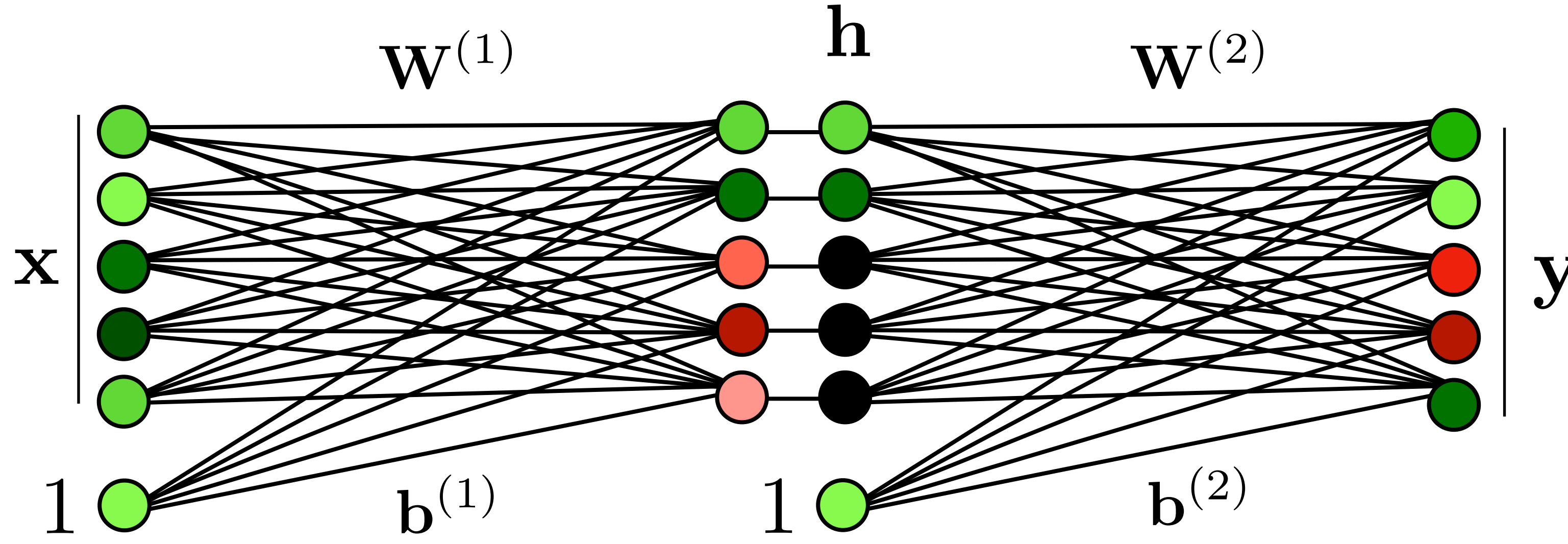
positive  
negative

$$\mathbf{h} = g(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \quad \mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$$

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

# Stacking layers

Input representation      Intermediate representation      Output representation



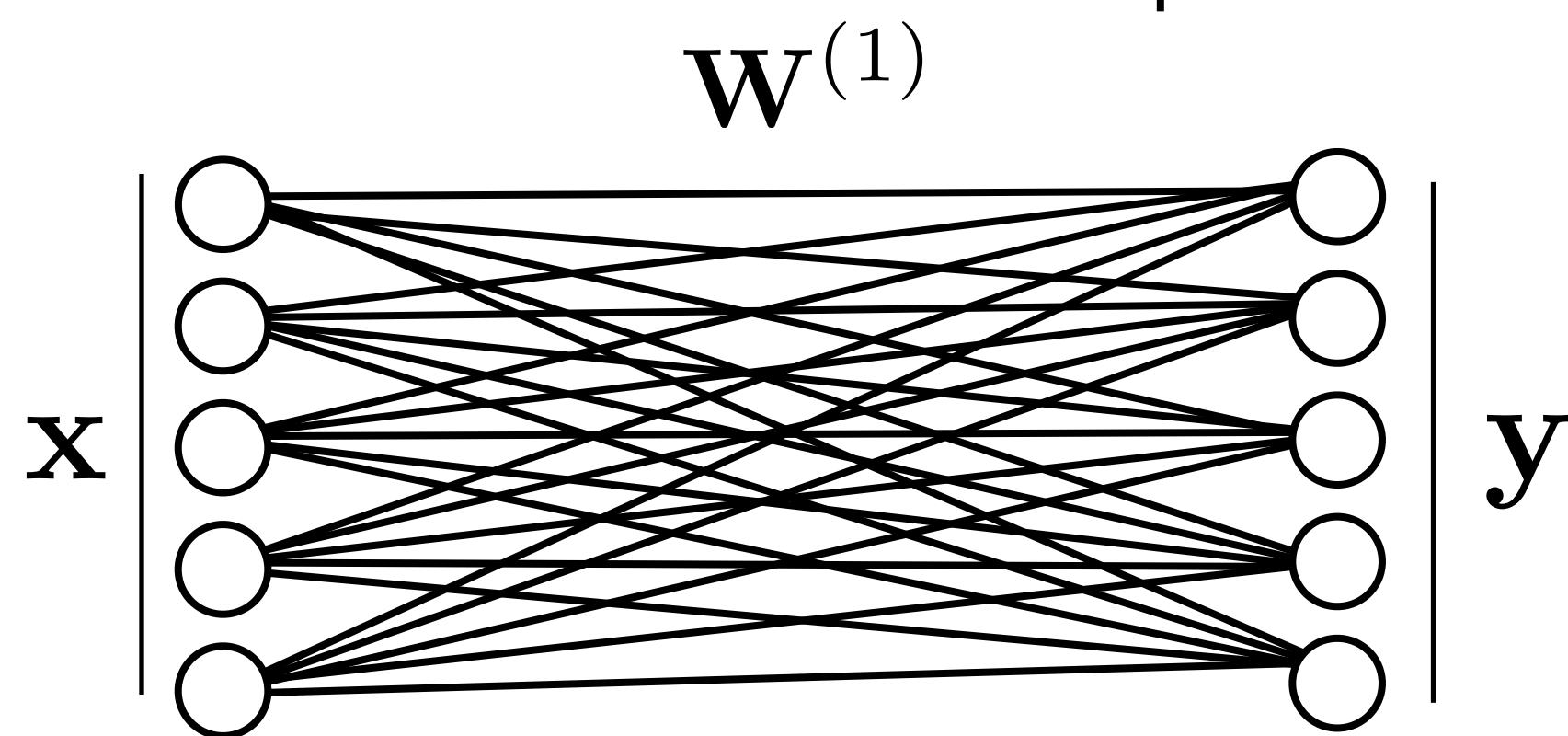
positive  
negative

$$\mathbf{h} = g(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \quad \mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$$

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

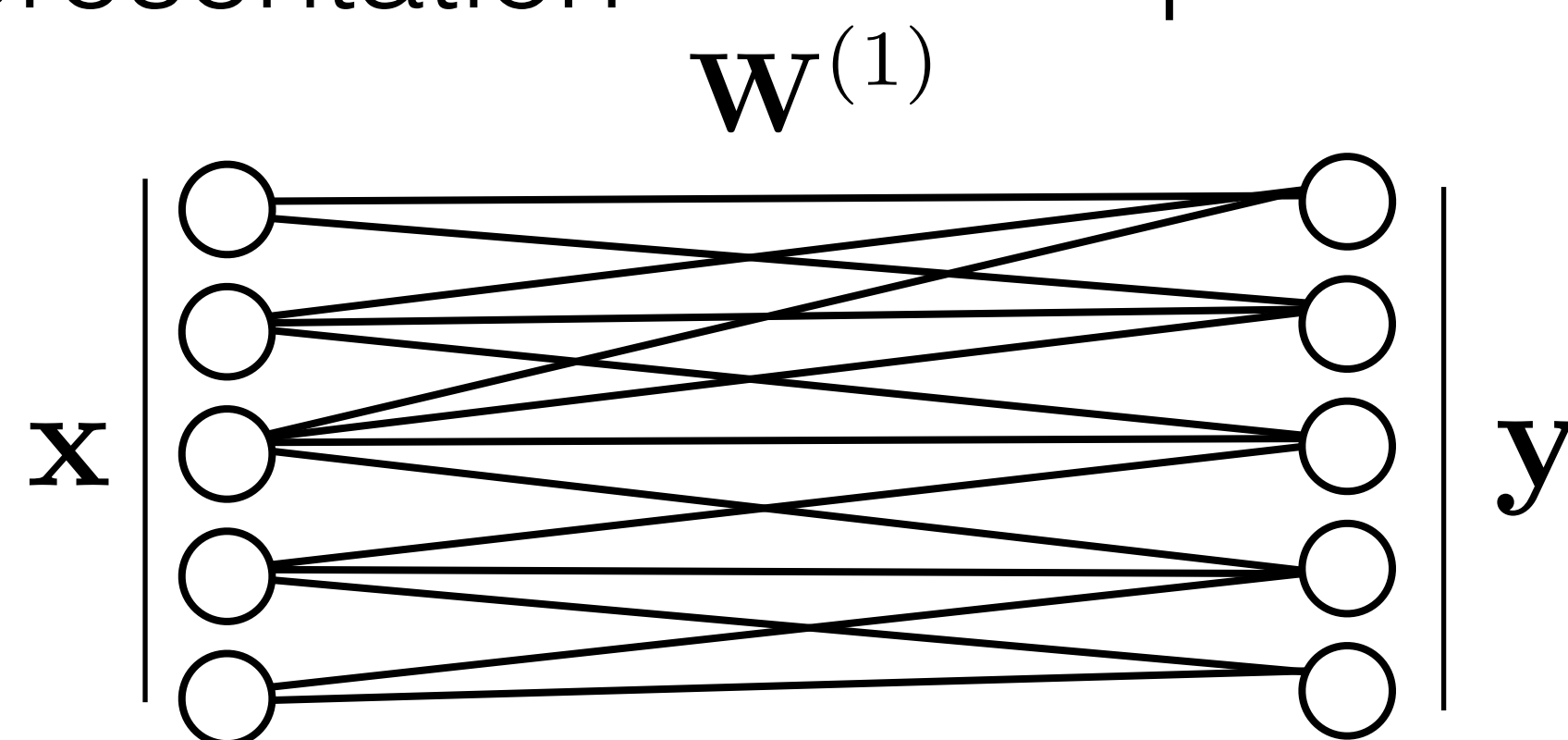
# Connectivity patterns

Input representation      Output representation



*Fully connected layer*

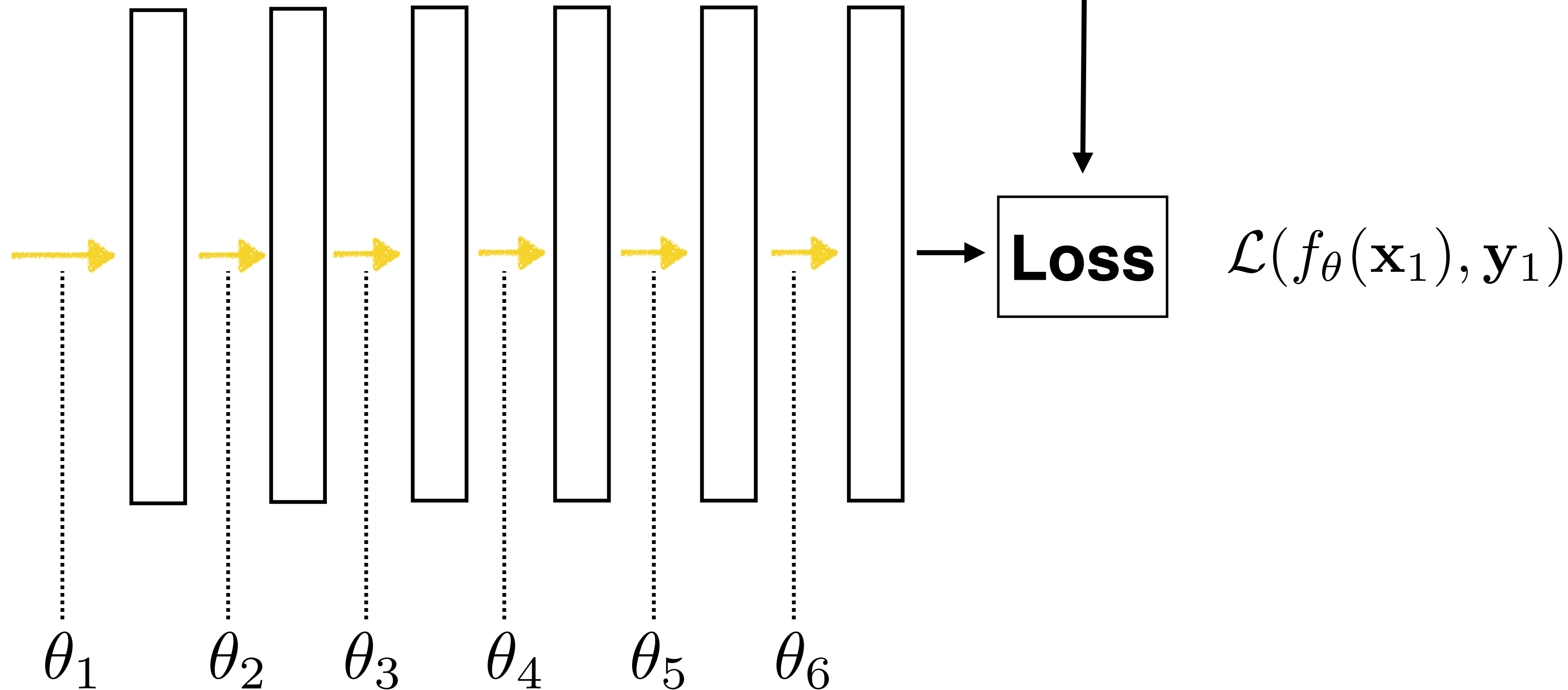
Input representation      Output representation



*Locally connected layer  
(Sparse  $W$ )*

# Deep learning

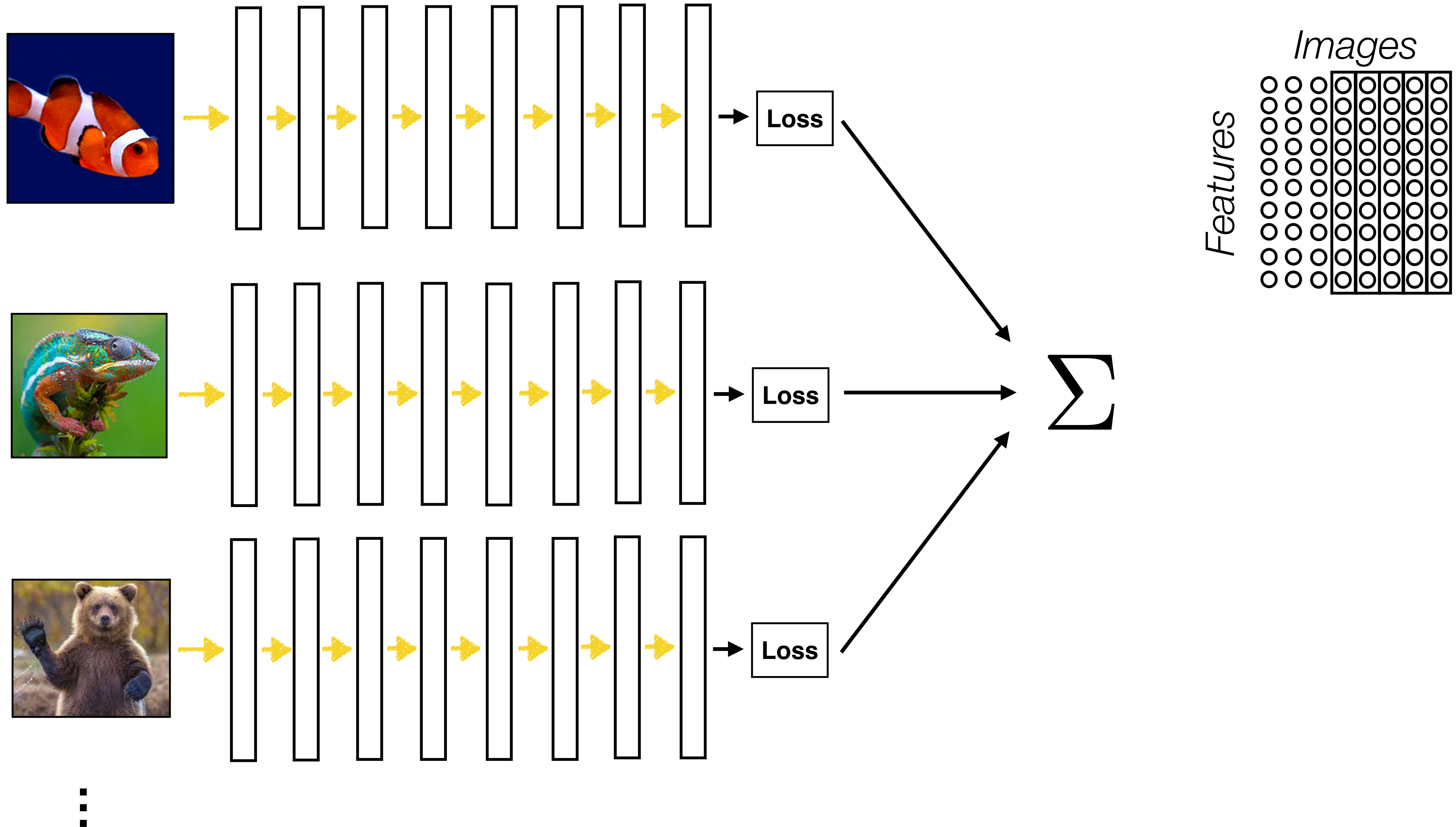
$y_1$   
“clown fish”



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

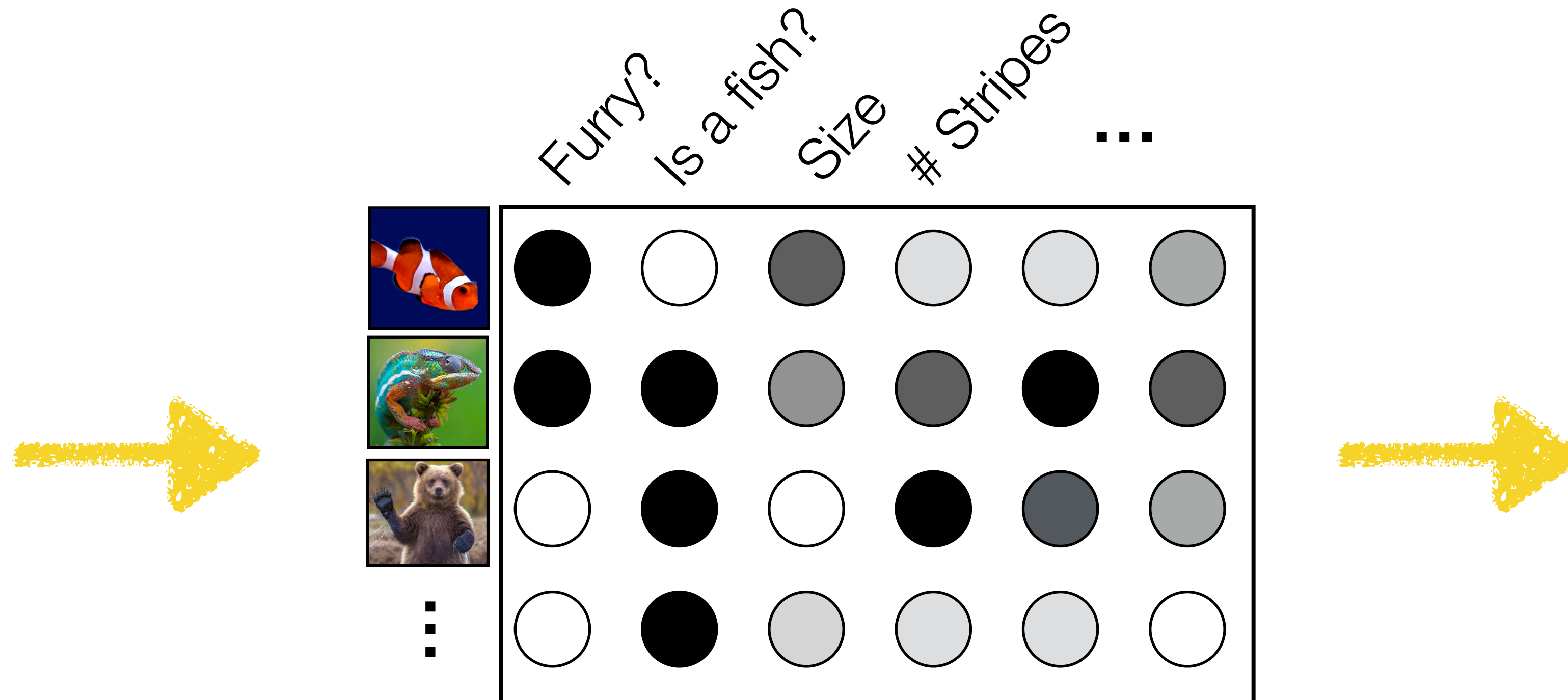


# Batch processing



# Tensors

(multi-dimensional arrays)



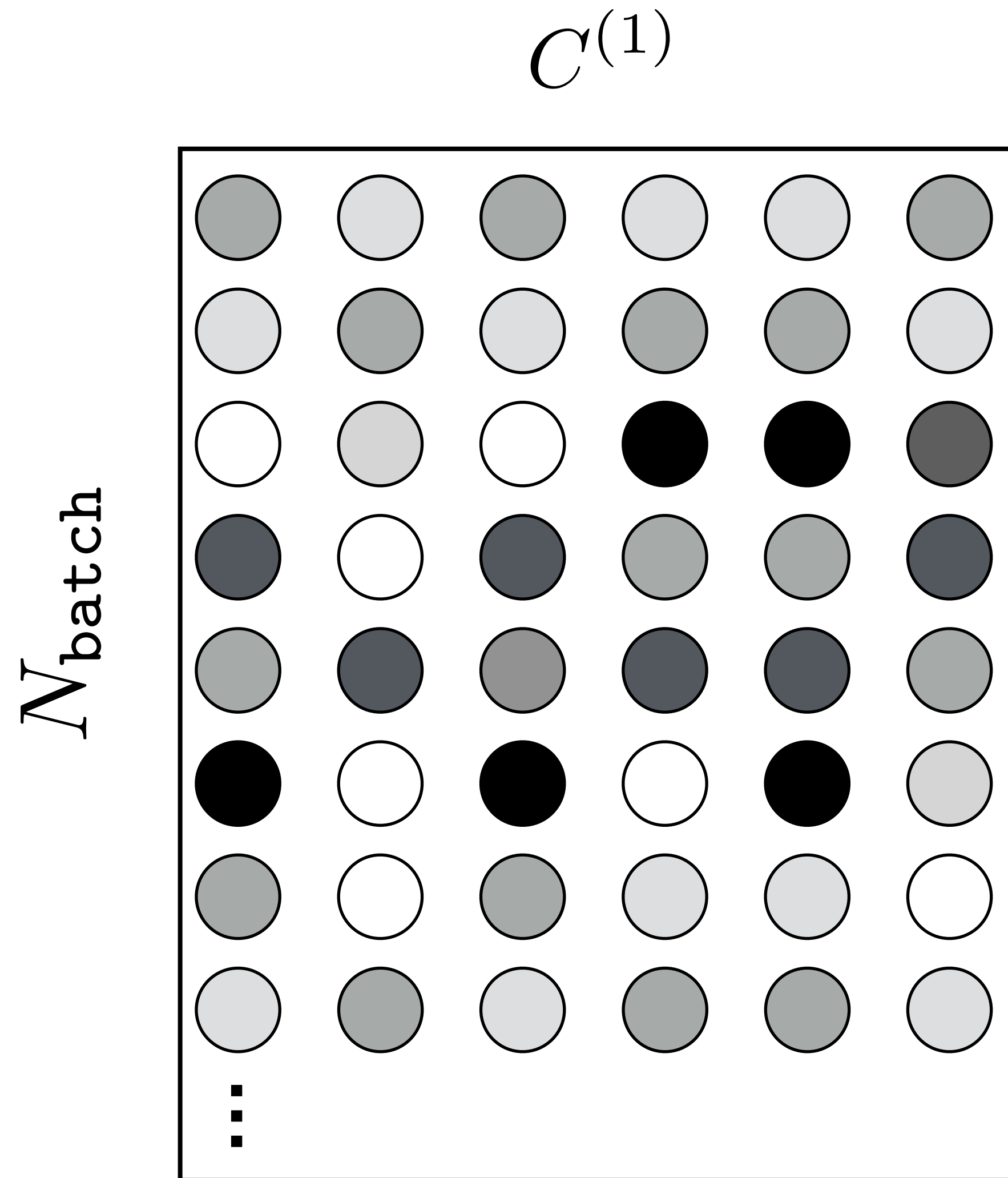
Each layer is a representation of the data

# Tensors

(multi-dimensional arrays)

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

# neurons  
# features  
# units  
# “channels”

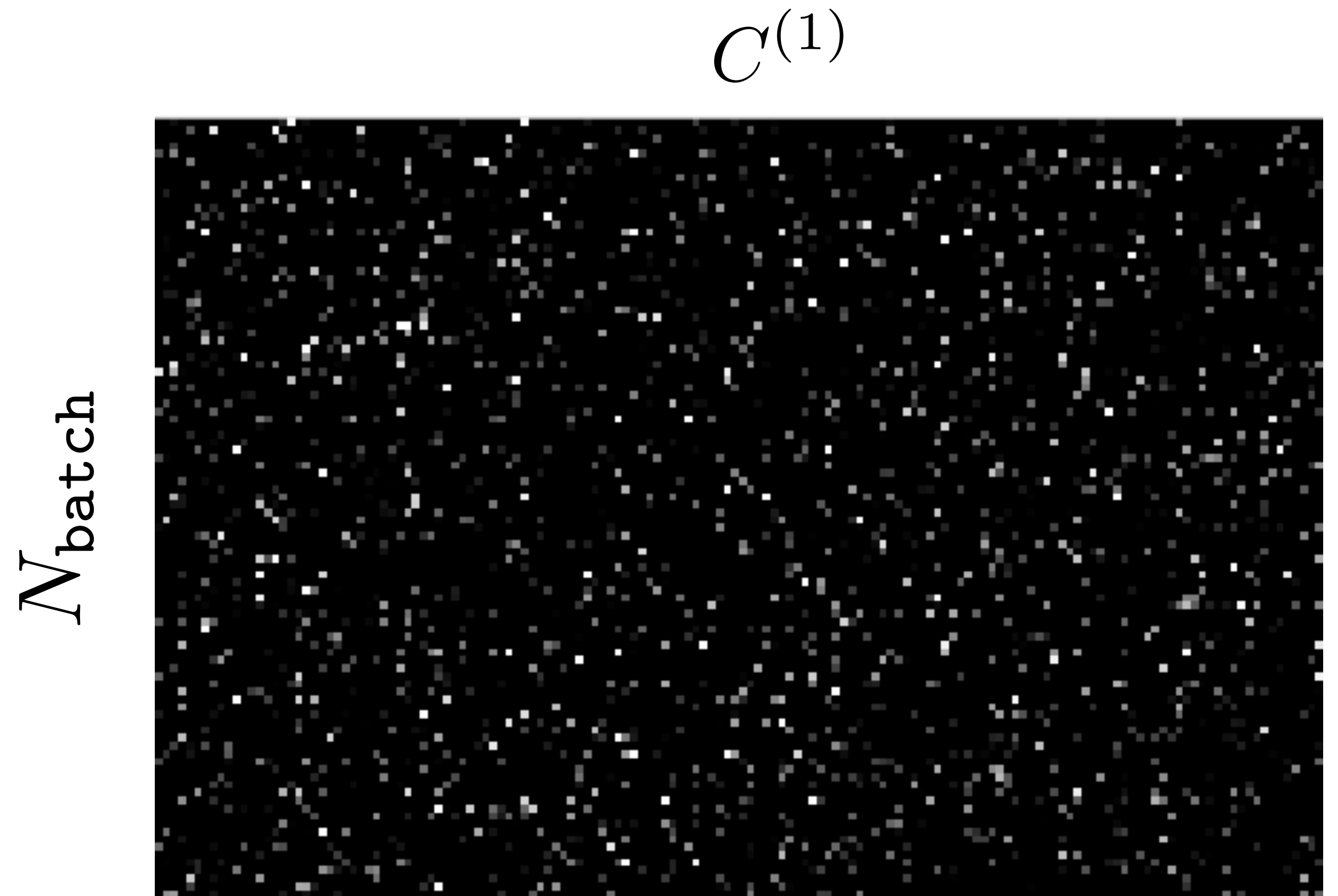


# Tensors

(multi-dimensional arrays)

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

# neurons  
# features  
# units  
# “channels”

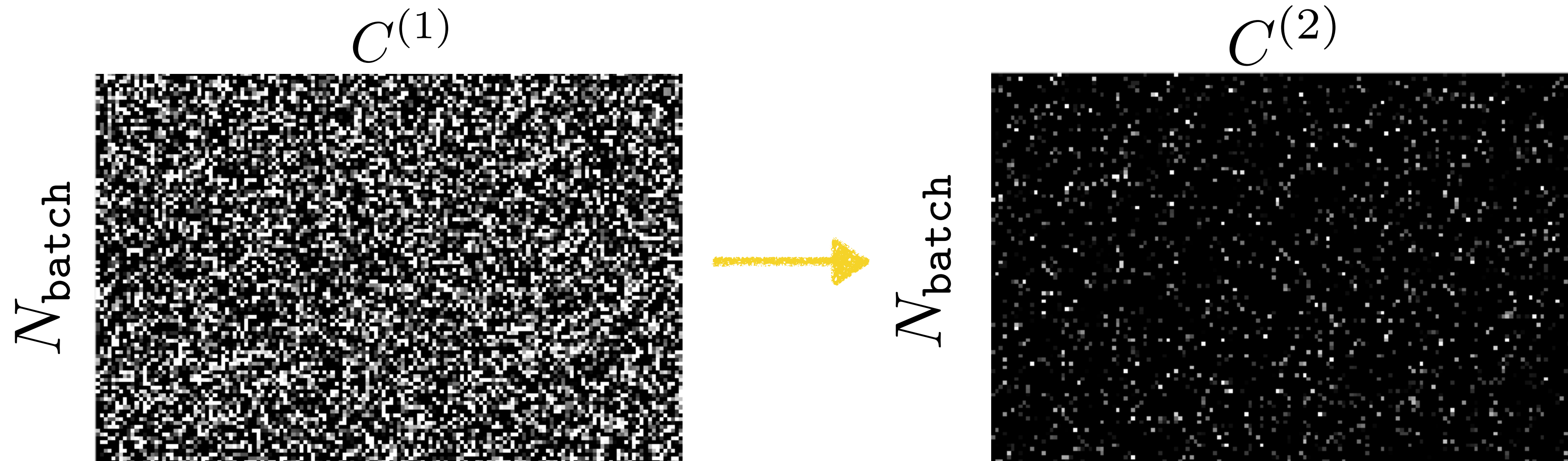


# Tensors

(multi-dimensional arrays)

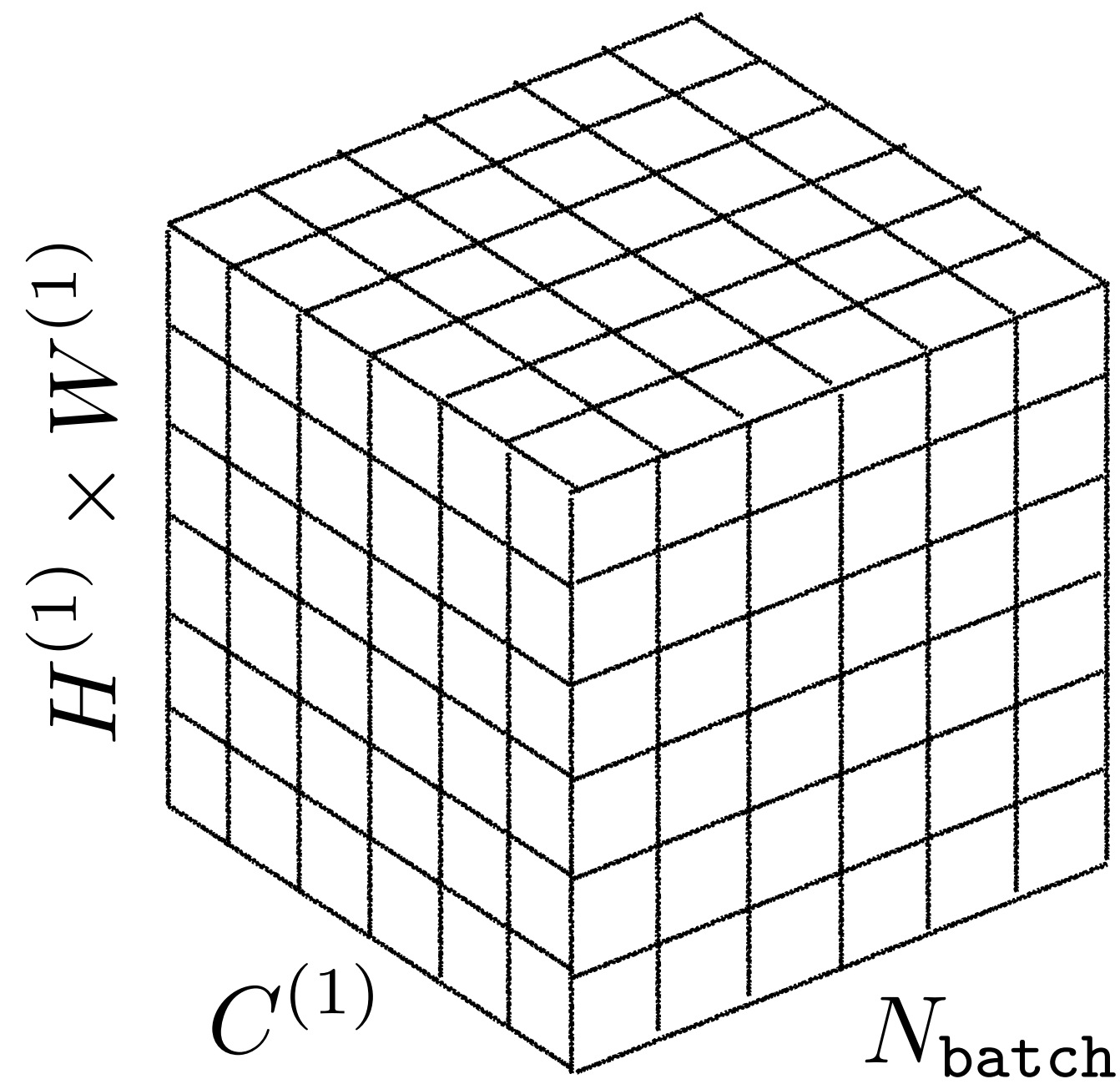
$$\mathbf{h}^{(1)} \in \mathbb{R}^{N_{\text{batch}} \times C^{(1)}}$$

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



# Processing a layer

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



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

