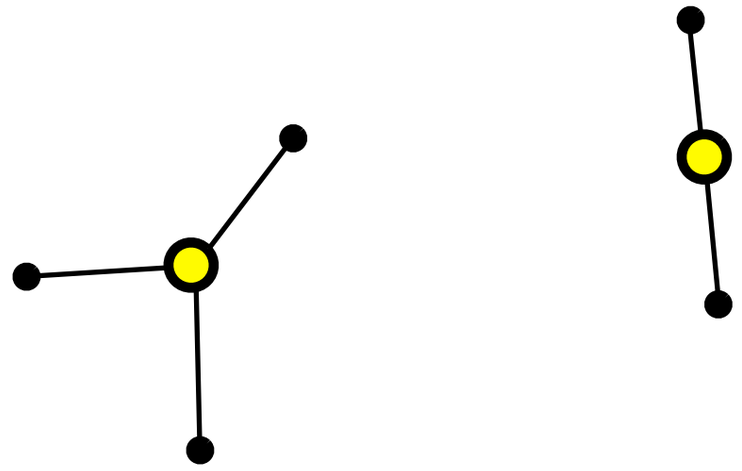
k-means clustering (recap)

- Idea: try to estimate k cluster centers by minimizing "distortion"
- Define distortion as:



$$D = \sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk} \parallel \mathbf{x}_n - \boldsymbol{\mu}_k \parallel^2$$

$$r_{nk} = 1 \text{ if } \mathbf{x}_n \in \text{cluster } k, \text{ 0 otherwise.}$$

- r_{nk} is 1 for the closest cluster mean to $\mathbf{x}_n$.
- Each point $\mathbf{x}_n$ is the minimum distance from its closet center.
- How do we learn the cluster means?
- Need to derive a ***learning rule***.

Deriving a learning rule for the cluster means

- Our objective function is:

$$D = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \parallel \mathbf{x}_n - \boldsymbol{\mu}_k \parallel^2$$

- Differentiate w.r.t. to the mean (the parameter we want to estimate):

$$\frac{\partial D}{\partial \boldsymbol{\mu}_k} = 2 \sum_{n=1}^{N} r_{nk}(\mathbf{x}_n - \boldsymbol{\mu}_k)$$

- We know the optimum is when

$$\frac{\partial D}{\partial \boldsymbol{\mu}_k} = 2 \sum_{n=1}^{N} r_{nk}(\mathbf{x}_n - \boldsymbol{\mu}_k) = 0$$

- Here, we can solve for the mean:

$$\boldsymbol{\mu}_k = \frac{\sum_n r_{nk} \mathbf{x}_n}{\sum_n r_{nk}}$$

- This is simply a weighted mean for each cluster.

- Thus we have a simple estimation algorithm (*k-means clustering*)

  1. select k points at random
  2. estimate (update) means
  3. repeat until converged

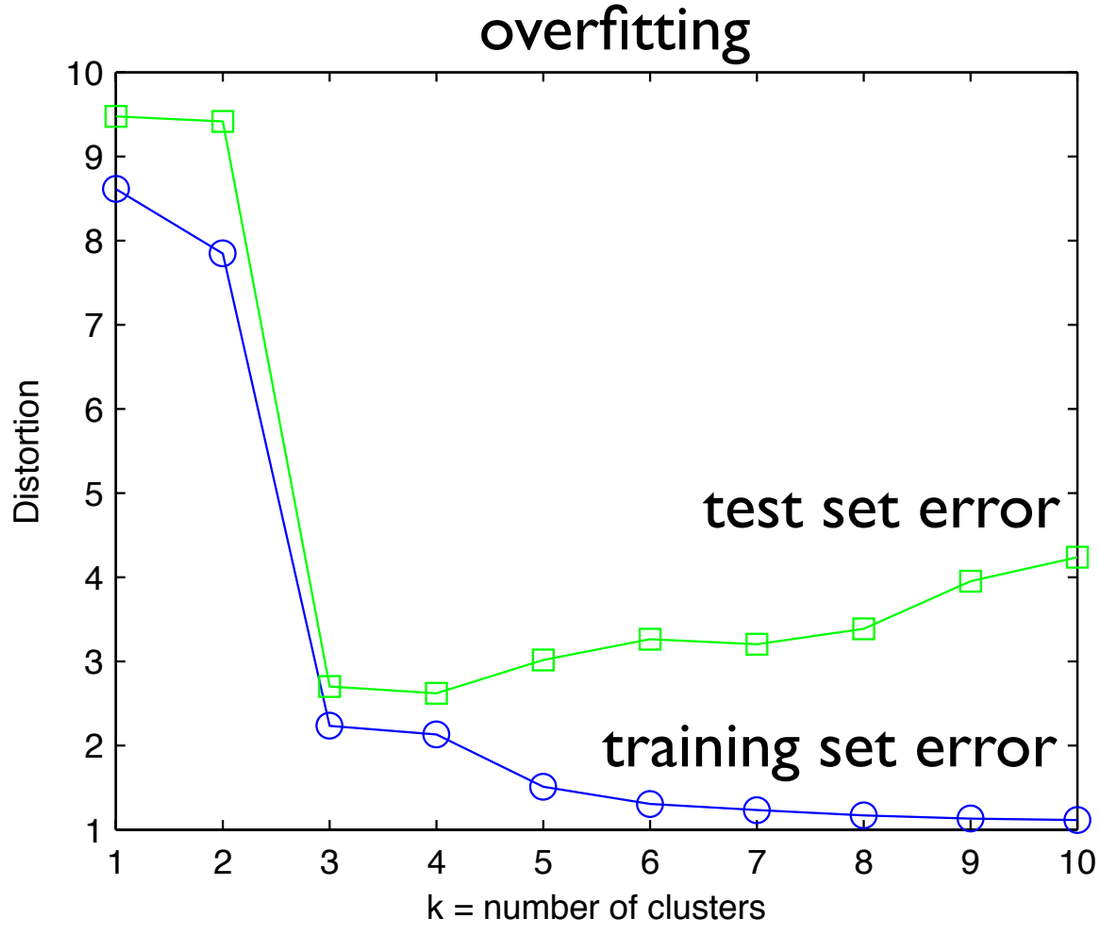
- convergence (to a local minimum) is guaranteed

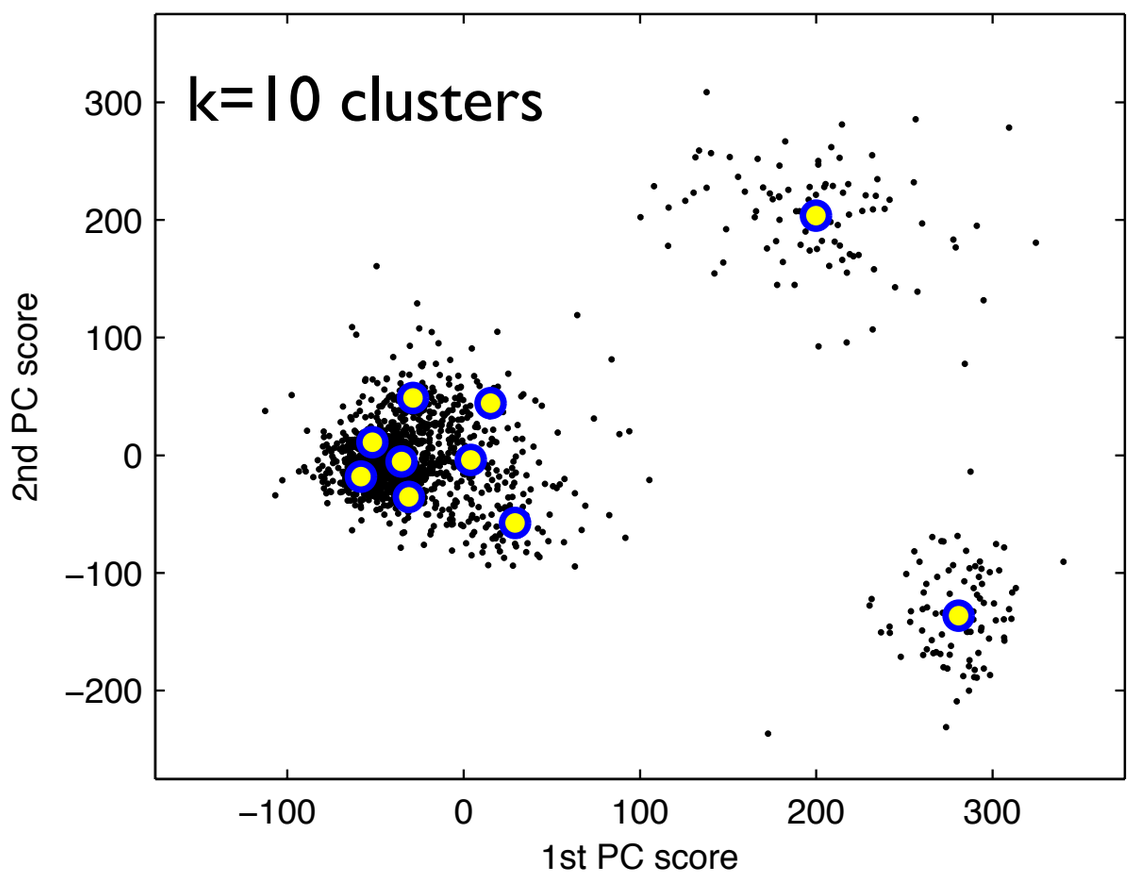
How do we choose k?

- *Increasing* k, will always *decrease* our distortion. This will ***overfit*** the data.
  - How can we avoid this?
  - Or how do we choose the best k?
- One way: ***cross validation***
- Use our distortion metric:

$$D = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \parallel \mathbf{x}_n - \boldsymbol{\mu}_k \parallel^2$$

- Then just measure the distortion on a *test data set*, and stop when we reach a minimum.



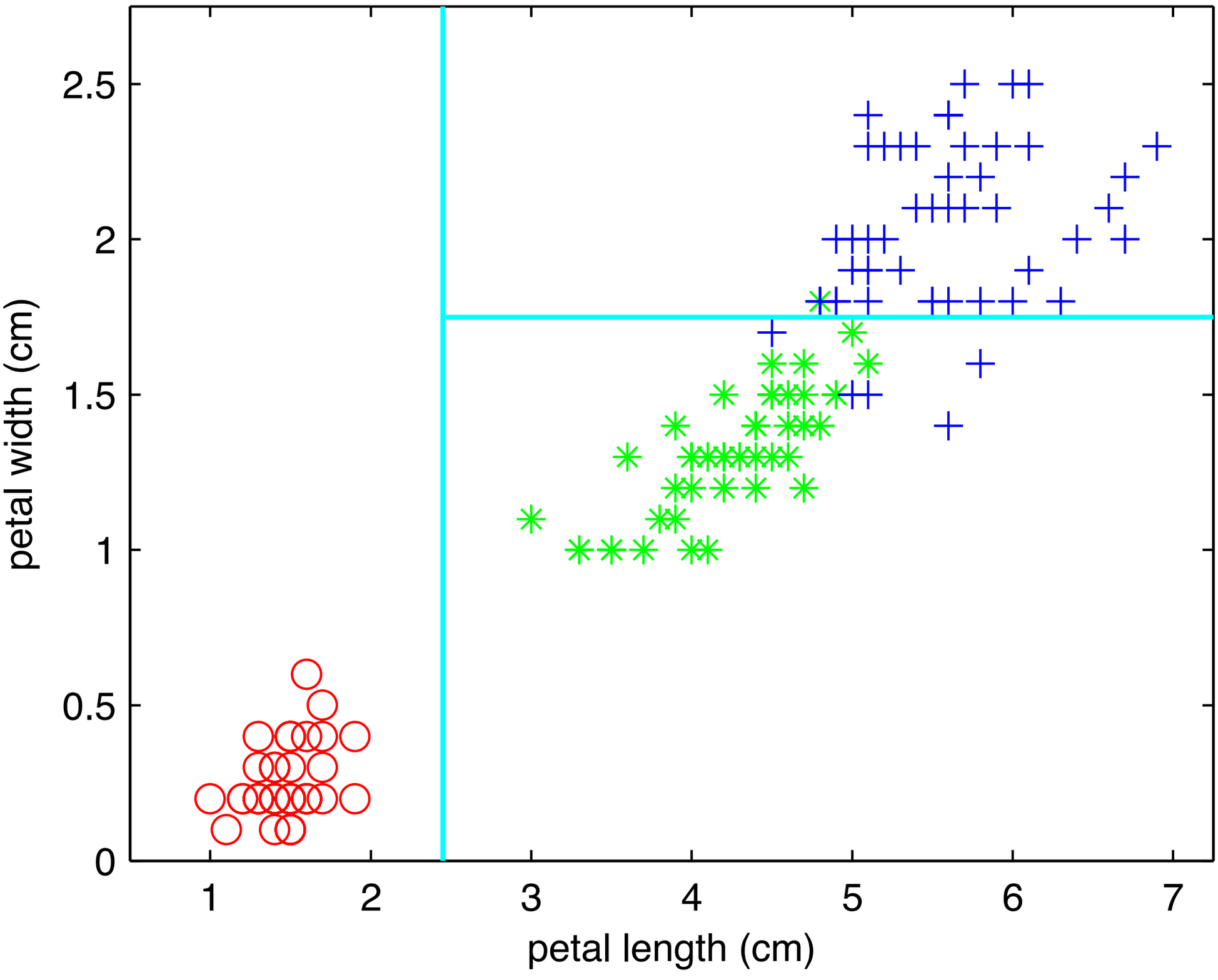


EECS 391
Intro to AI

Neural Networks

L20 Tue Nov 16

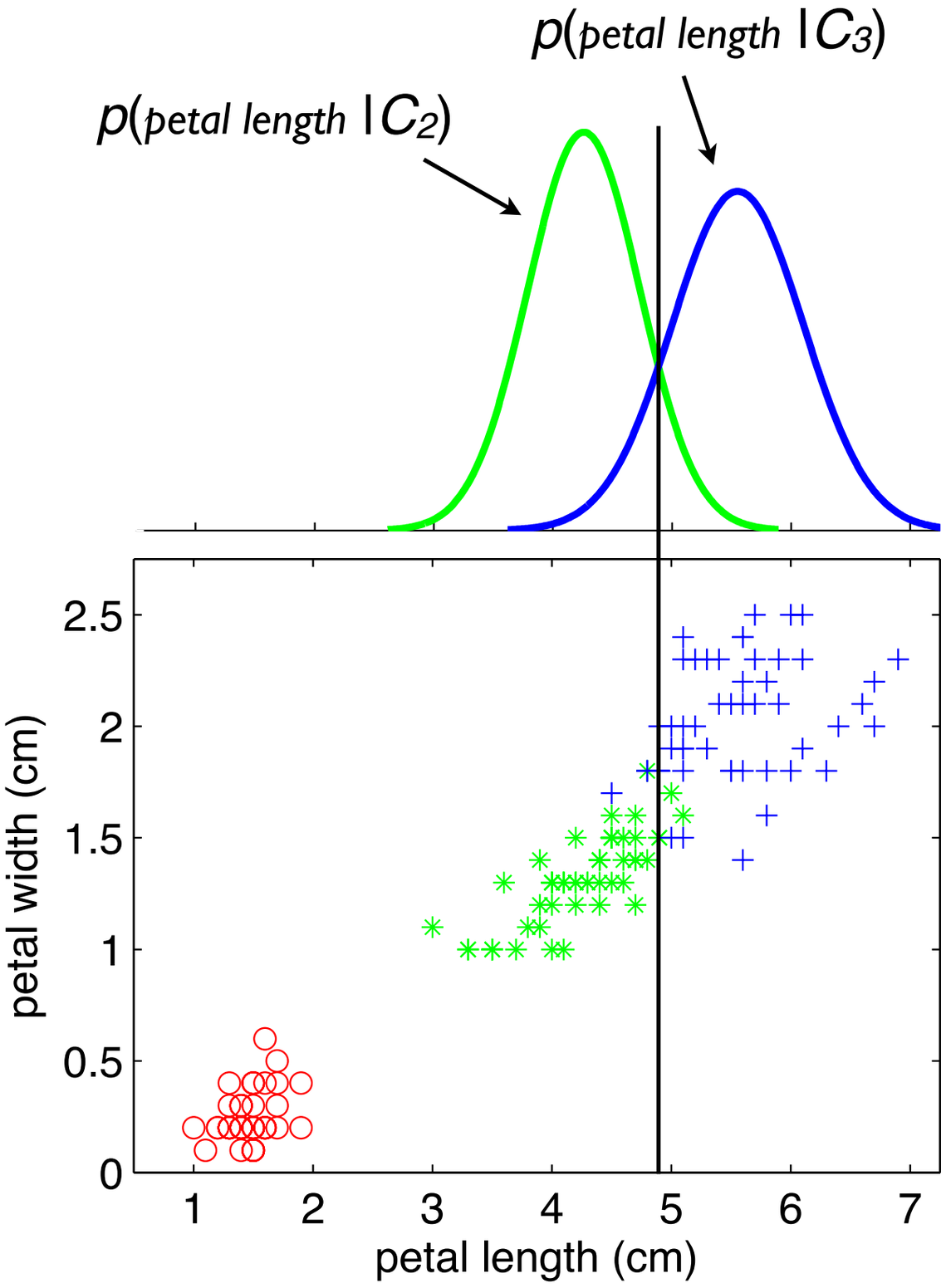
The Iris dataset with decision tree boundaries



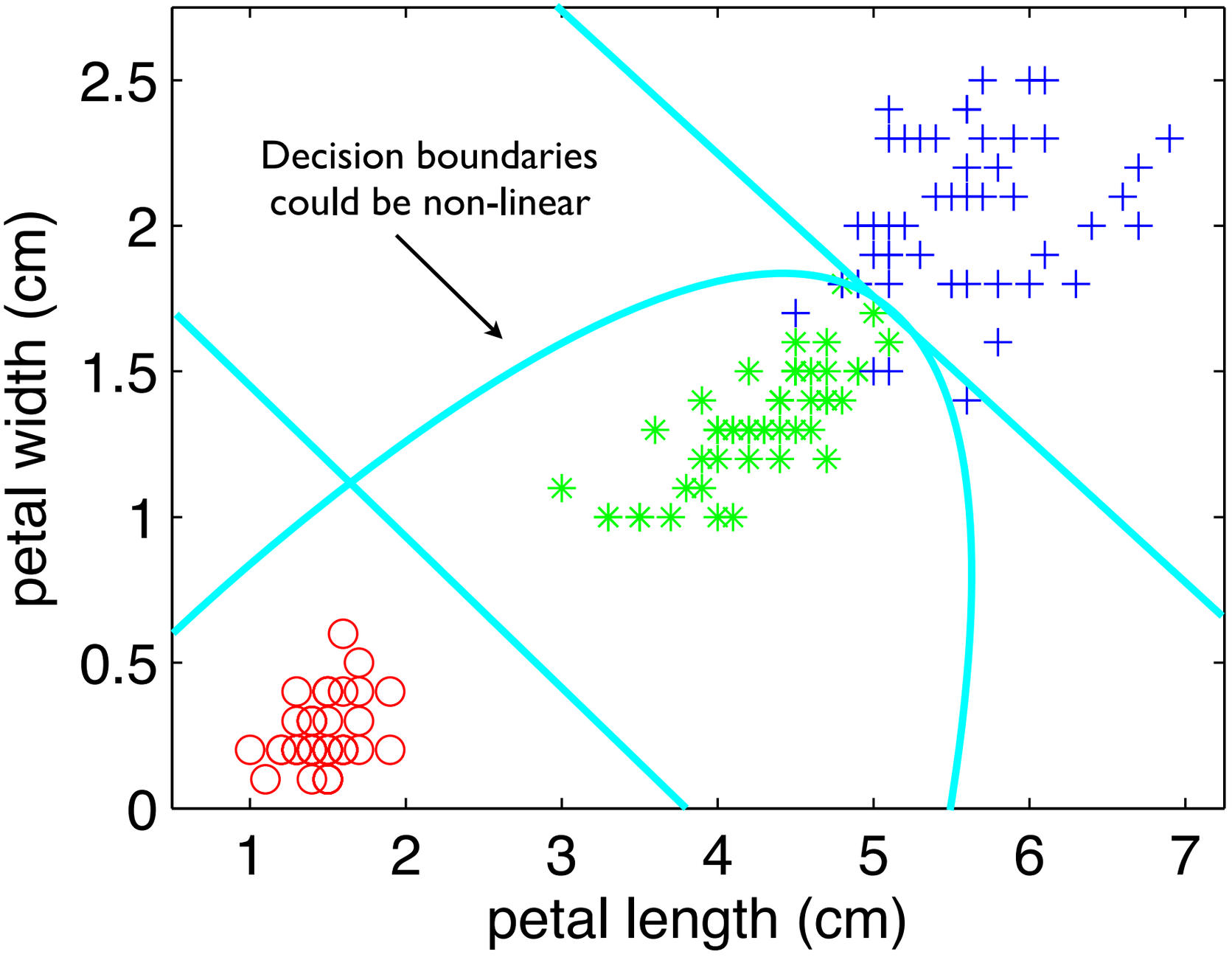
The optimal decision boundary for C_2 vs C_3

Is this really optimal?

Could we do better?



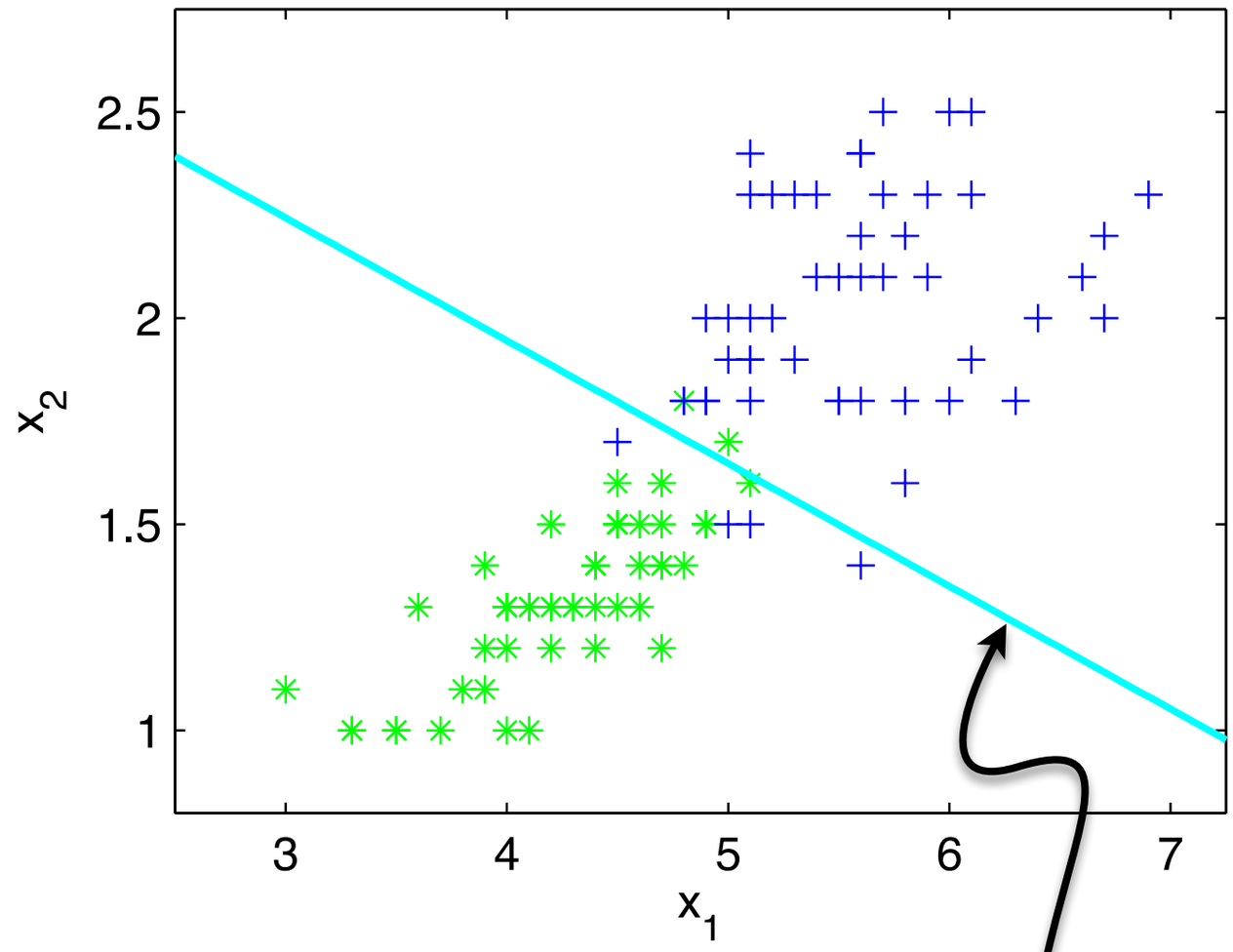
Arbitrary decision boundaries would be more powerful



Defining a decision boundary

$$
\begin{aligned}
y &= \mathbf{m}^T\mathbf{x} + b \\
&= m_1 x_1 + m_2 x_2 + b \\
&= \sum_i m_i x_i + b
\end{aligned}
$$

$$
\mathbf{x} \in \begin{cases} \text{class 1} & \text{if } y \geq 0, \\ \text{class 2} & \text{if } y < 0. \end{cases}
$$



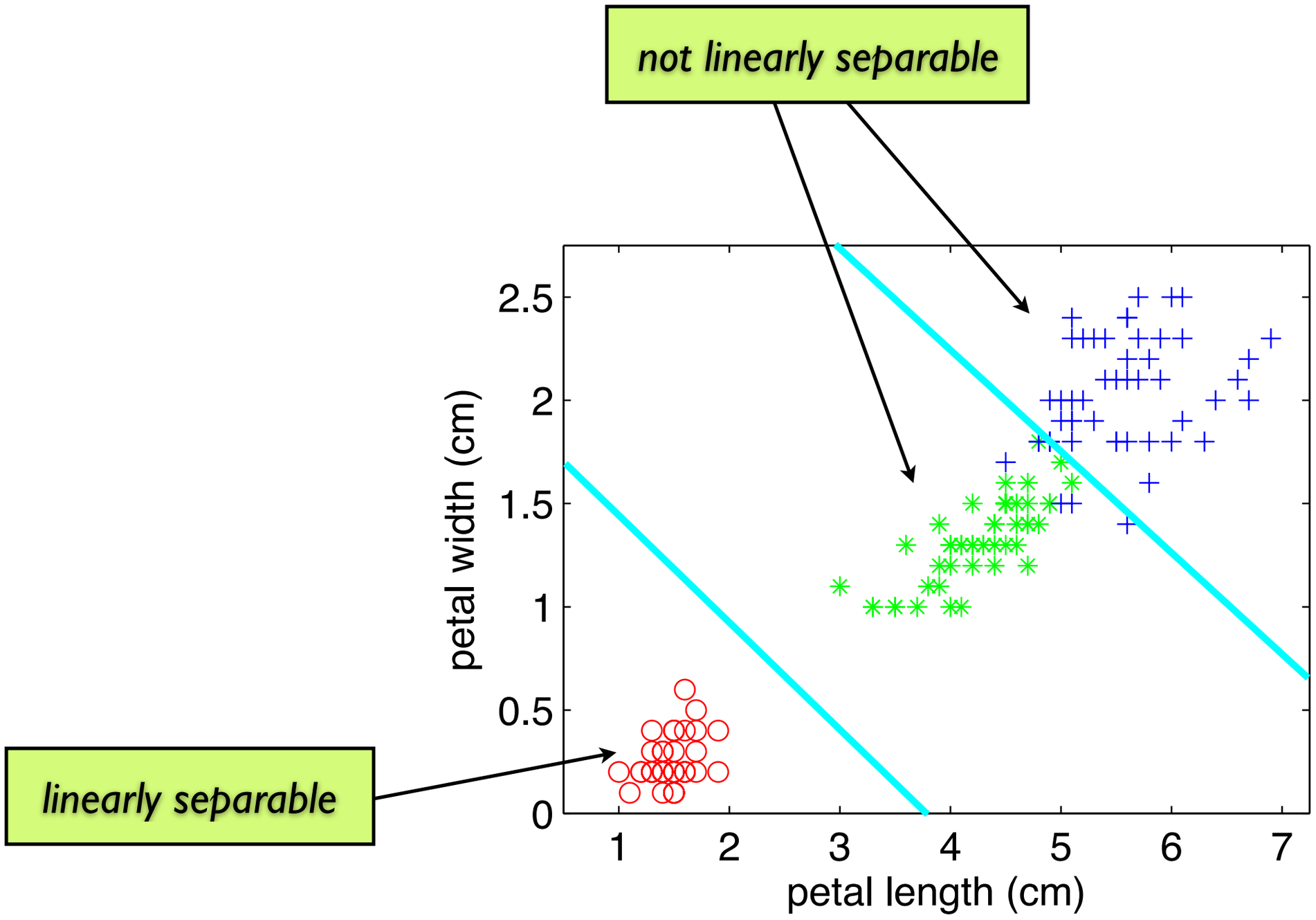
The decision *boundary*:

$$y = \mathbf{m}^T\mathbf{x} + b = 0$$

Or in terms of scalars:

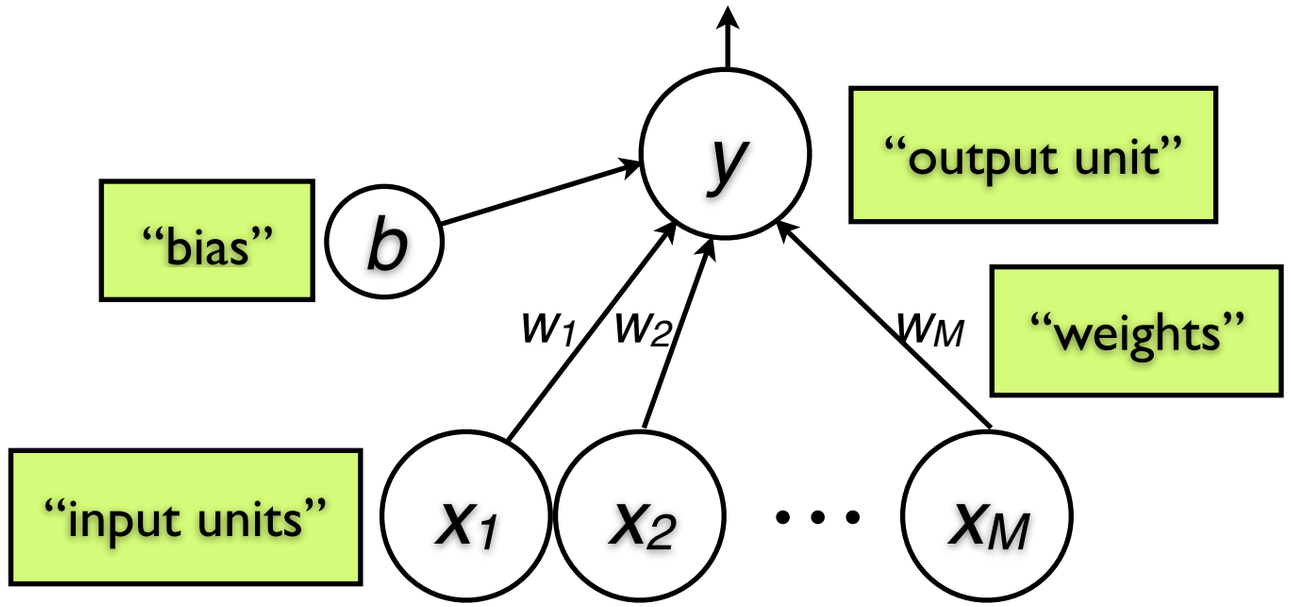
$$
\begin{aligned}
m_1 x_1 + m_2 x_2 &= -b \\
\Rightarrow x_2 &= -\frac{m_1 x_1 + b}{m_2}
\end{aligned}
$$

Linear separability

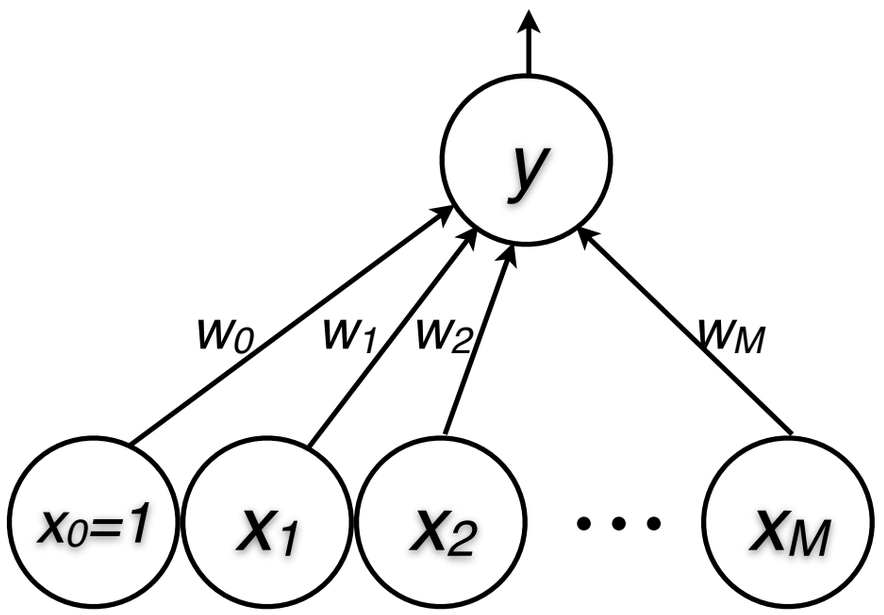


Diagraming the classifier as a "neural" network

$$
\begin{aligned}
y &= \mathbf{w}^T\mathbf{x} + b \\
&= \sum_{i=1}^{M} w_i x_i + b
\end{aligned}
$$



$$
y = \mathbf{w}^T\mathbf{x} = \sum_{i=0}^{M} w_i x_i
$$



Determining, ie learning, the optimal linear discriminant

- First we must define an *objective function*, ie the goal of learning
- Simple idea: adjust weights so that output $y(\mathbf{x}_n)$ matches class c_n
- *Objective*: minimize sum-squared error over all patterns $\mathbf{x}_n$:

$$E = \frac{1}{2}\sum_{n=1}^{N}(\mathbf{w}^T\mathbf{x}_n - c_n)^2$$

- Note the notation $\mathbf{x}_n$ defines a *pattern vector*:

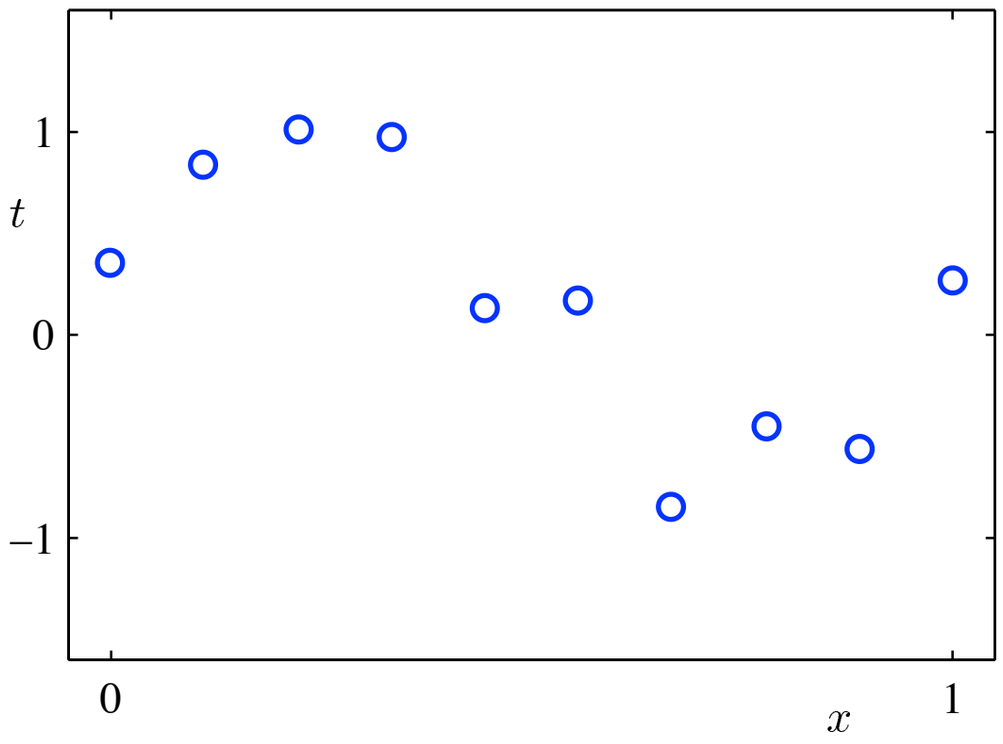
$$\mathbf{x}_n = \{x_1, \ldots, x_M\}_n$$

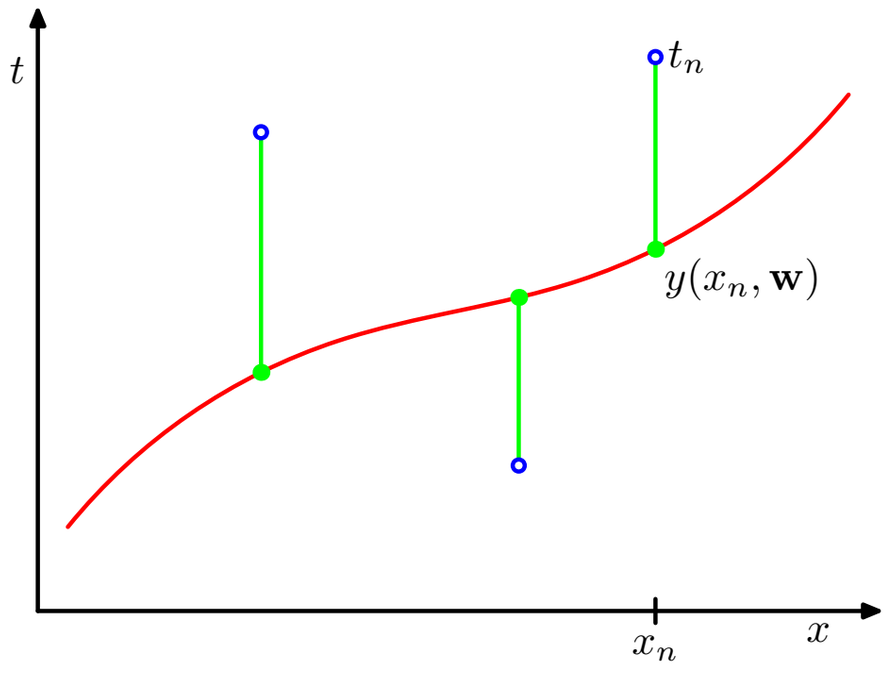
- We can define the desired class as:

$$c_n = \begin{cases} 0 & \mathbf{x}_n \in \text{class } 1 \\ 1 & \mathbf{x}_n \in \text{class } 2 \end{cases}$$

We've seen this before: curve fitting

$$t = \sin(2\pi x) + \text{noise}$$





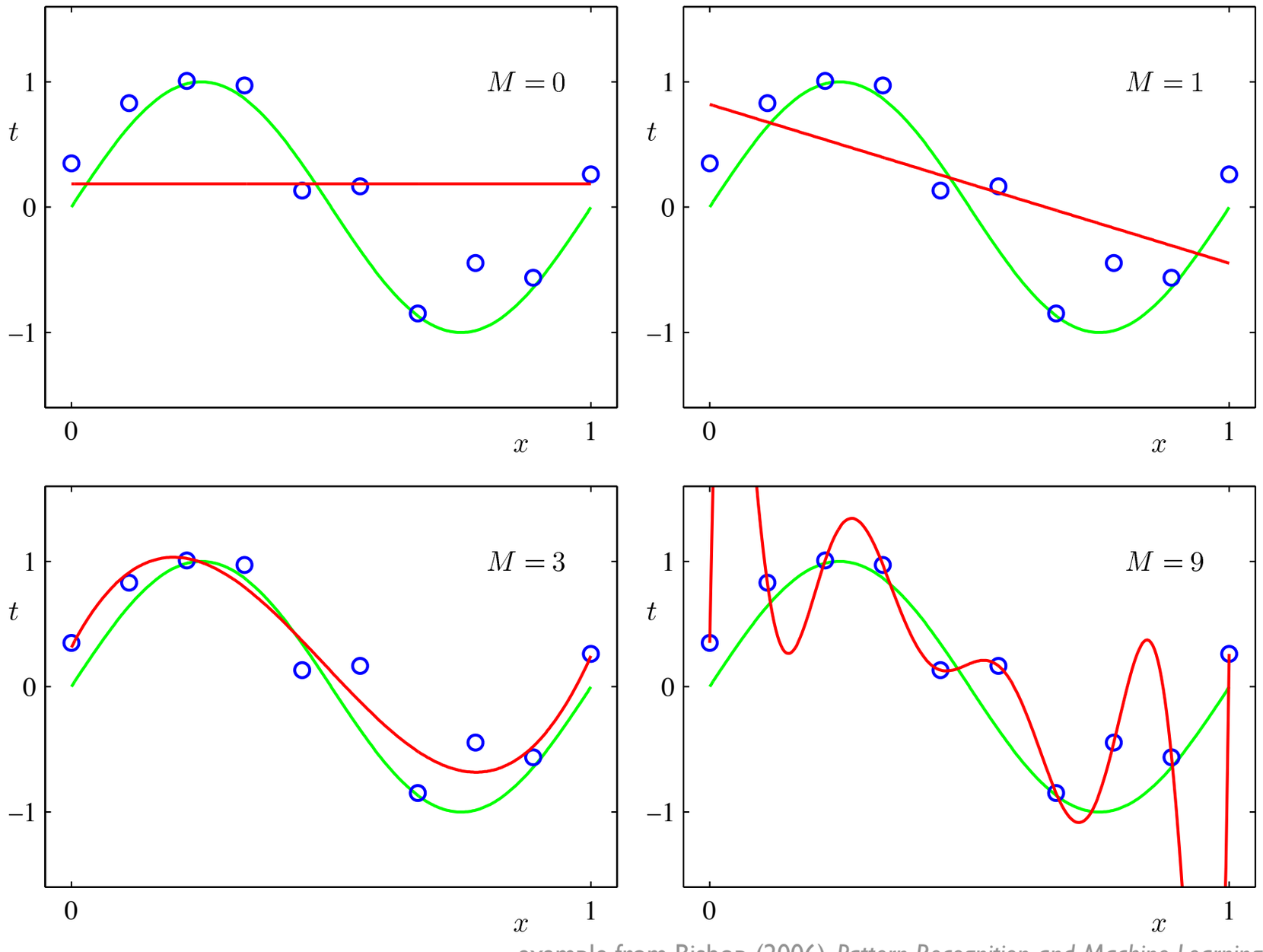
example from Bishop (2006), *Pattern Recognition and Machine Learning*

Neural networks compared to polynomial curve fitting

$$y(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \cdots + w_M x^M = \sum_{j=0}^{M} w_j x^j$$

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} [y(x_n, \mathbf{w}) - t_n]^2$$

For the linear network, M=1 and there are multiple input dimensions

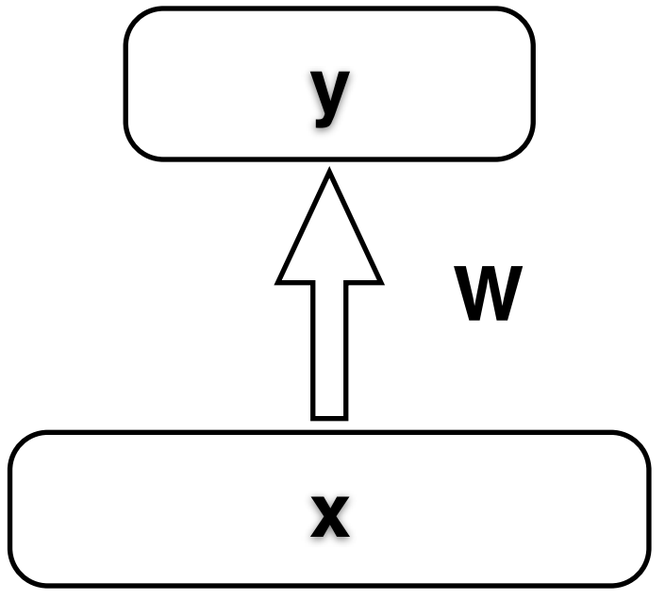


example from Bishop (2006), *Pattern Recognition and Machine Learning*

General form of a linear network

- A linear neural network is simply a linear transformation of the input.

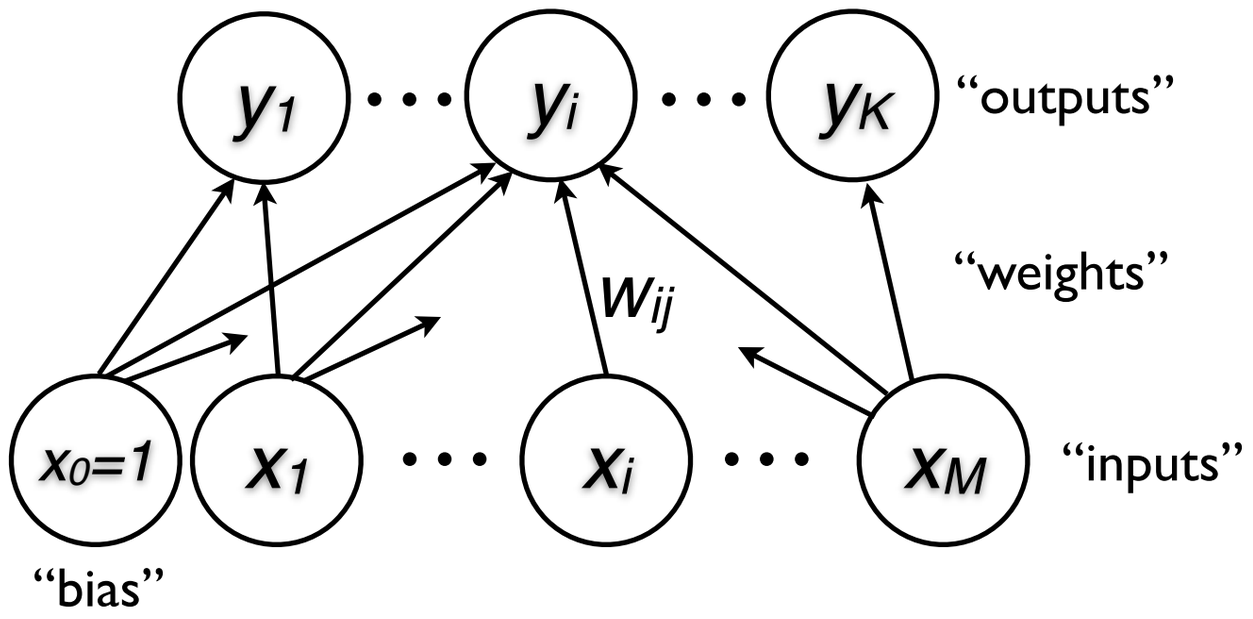
$$y_j = \sum_{i=0}^{M} w_{i,j} x_i$$



- Or, in matrix-vector form:

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

- Multiple outputs corresponds to *multivariate regression*



Training the network: Optimization by gradient descent

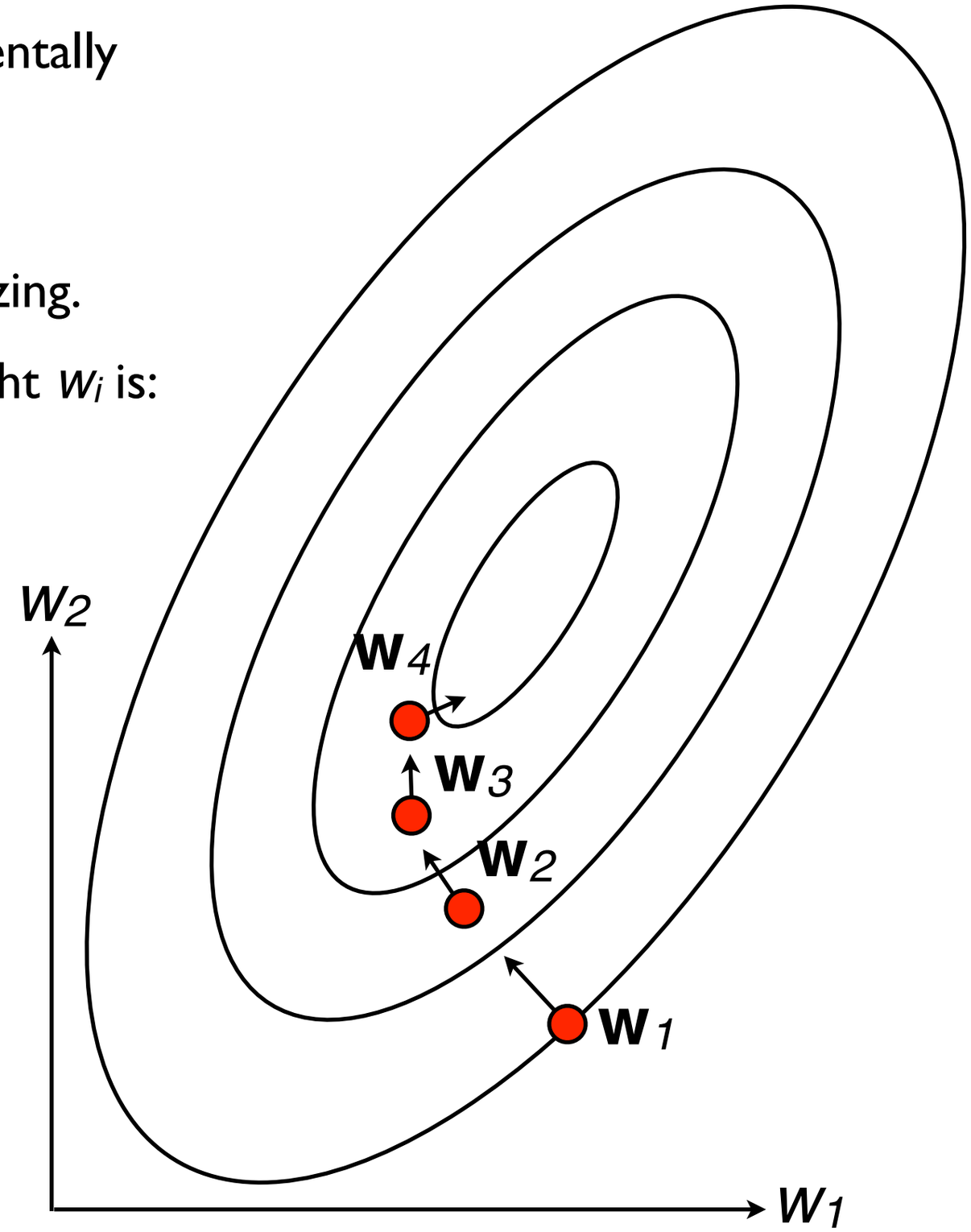
- We can adjust the weights incrementally to minimize the objective function.
- This is called *gradient descent*
- Or gradient *ascent* if we're maximizing.
- The gradient descent rule for weight w_i is:

$$w_i^{t+1} = w_i^t - \epsilon \frac{\partial E}{w_i}$$

- Or in vector form:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \epsilon \frac{\partial E}{\mathbf{w}}$$

- For gradient *ascent*, the *sign* of the gradient step changes.



Computing the gradient

- Idea: minimize error by gradient descent
- Take the derivative of the objective function wrt the weights:

$$
\begin{aligned}
E &= \frac{1}{2}\sum_{n=1}^{N}(\mathbf{w}^T\mathbf{x}_n - c_n)^2 \\
\frac{\partial E}{w_i} &= \frac{2}{2}\sum_{n=1}^{N}(w_0 x_{0,n} + \cdots + w_i x_{i,n} + \cdots + w_M x_{M,n} - c_n)x_{i,n} \\
&= \sum_{n=1}^{N}(\mathbf{w}^T\mathbf{x}_n - c_n)x_{i,n}
\end{aligned}
$$

- And in vector form:

$$
\frac{\partial E}{\mathbf{w}} = \sum_{n=1}^{N}(\mathbf{w}^T\mathbf{x}_n - c_n)\mathbf{x}_n
$$

Simulation: learning the decision boundary

- Each iteration updates the gradient:

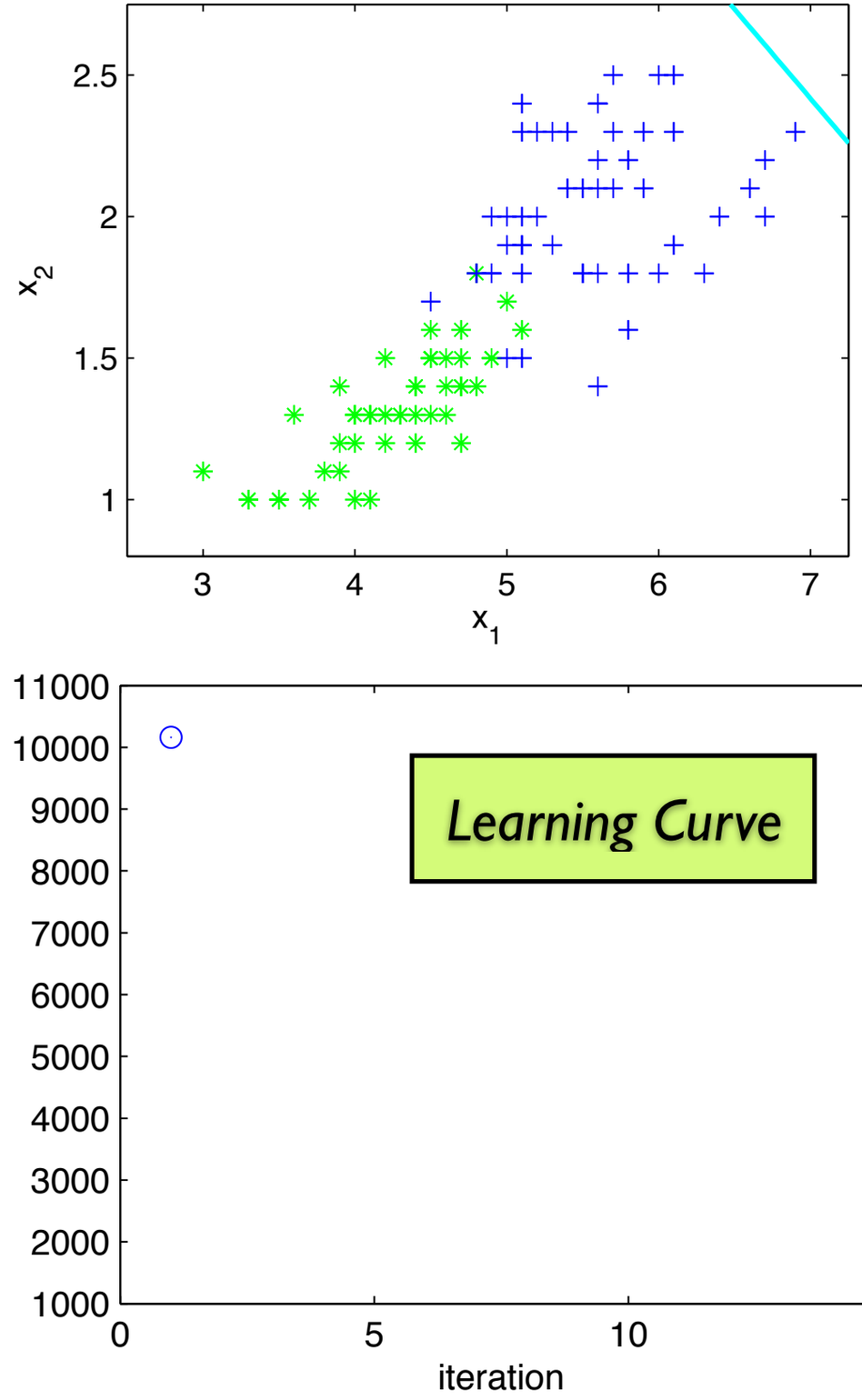
$$w_i^{t+1} = w_i^t - \epsilon \frac{\partial E}{w_i}$$

$$\frac{\partial E}{w_i} = \sum_{n=1}^{N} (\mathbf{w}^T \mathbf{x}_n - c_n) x_{i,n}$$

- Epsilon is a small value:

  ε = 0.1/N

- Epsilon too large:
  - learning diverges
- Epsilon too small:
  - convergence slow



Simulation: learning the decision boundary

- Each iteration updates the gradient:

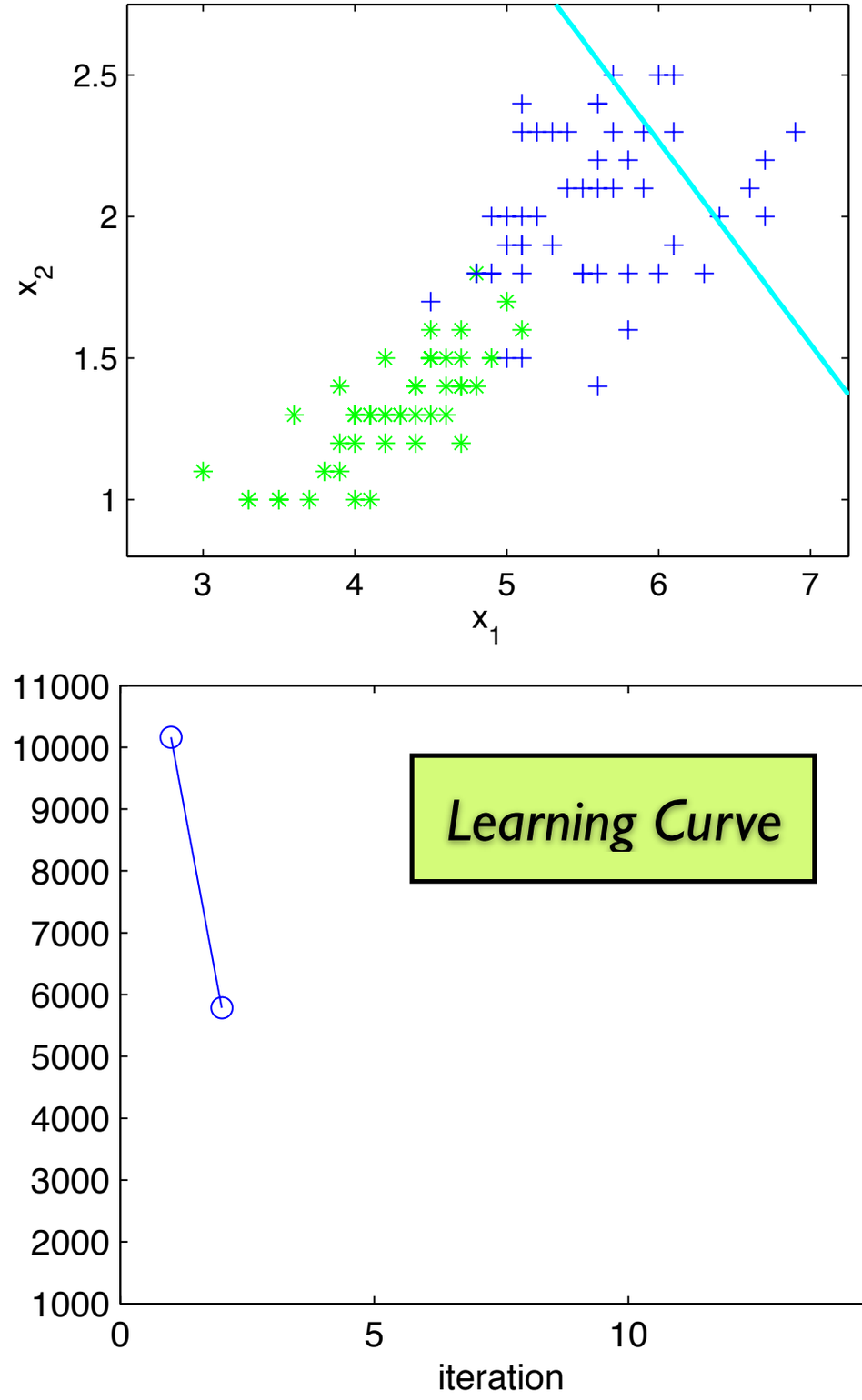
$$w_i^{t+1} = w_i^t - \epsilon \frac{\partial E}{w_i}$$

$$\frac{\partial E}{w_i} = \sum_{n=1}^{N} (\mathbf{w}^T \mathbf{x}_n - c_n) x_{i,n}$$

- Epsilon is a small value:

  ε = 0.1/N

- Epsilon too large:
  - learning diverges
- Epsilon too small:
  - convergence slow



Simulation: learning the decision boundary

- Each iteration updates the gradient:

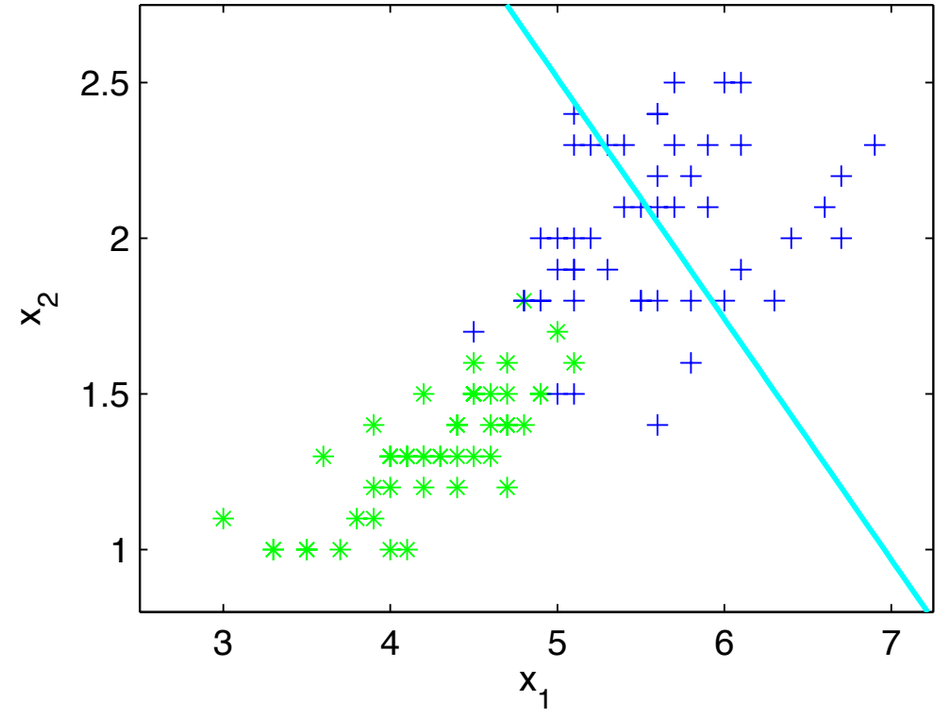
$$w_i^{t+1} = w_i^t - \epsilon \frac{\partial E}{w_i}$$

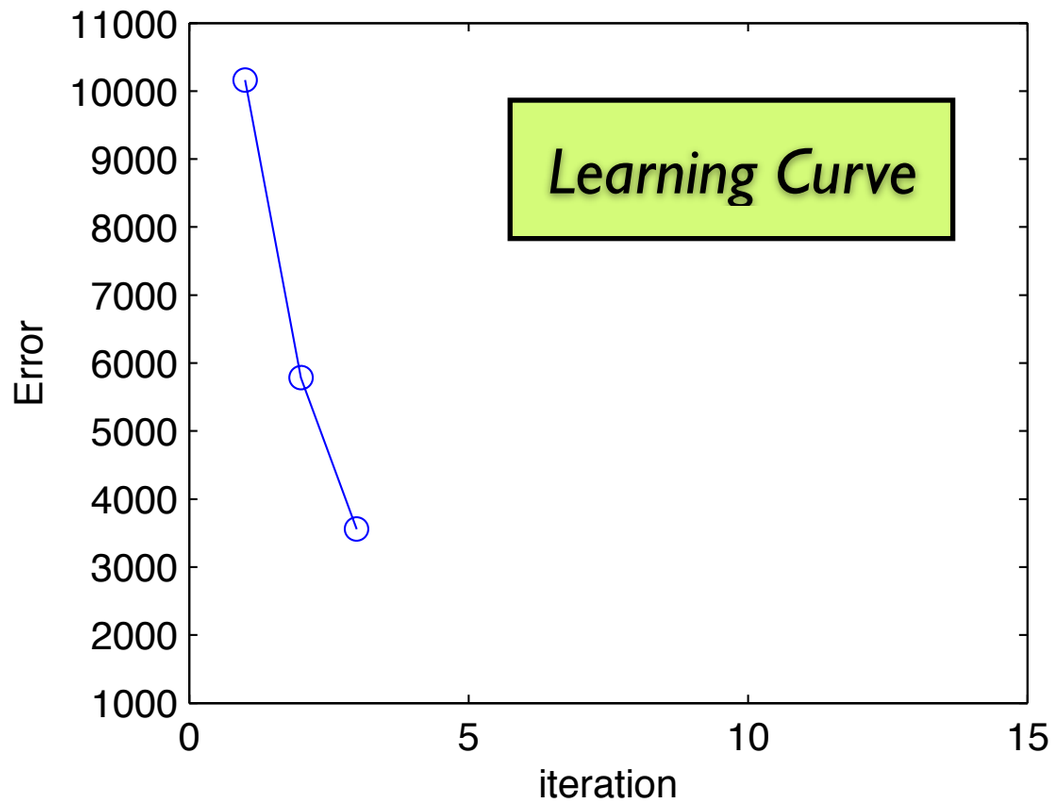
$$\frac{\partial E}{w_i} = \sum_{n=1}^{N} (\mathbf{w}^T \mathbf{x}_n - c_n) x_{i,n}$$

- Epsilon is a small value:

  $\varepsilon = 0.1/N$

- Epsilon too large:
  - learning diverges
- Epsilon too small:
  - convergence slow





Simulation: learning the decision boundary

- Each iteration updates the gradient:

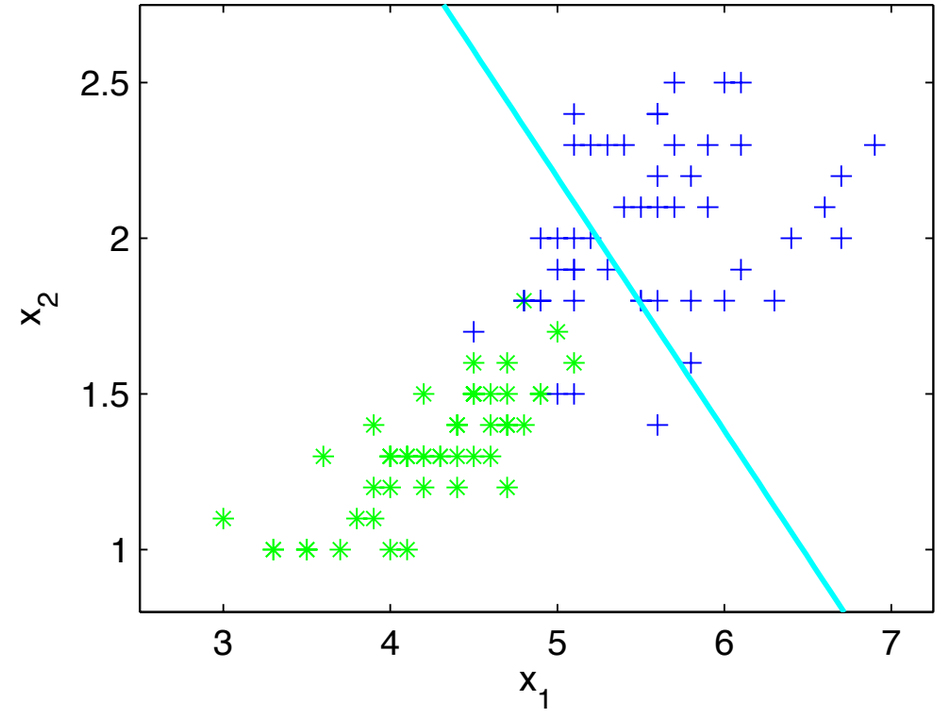
$$w_i^{t+1} = w_i^t - \epsilon \frac{\partial E}{w_i}$$

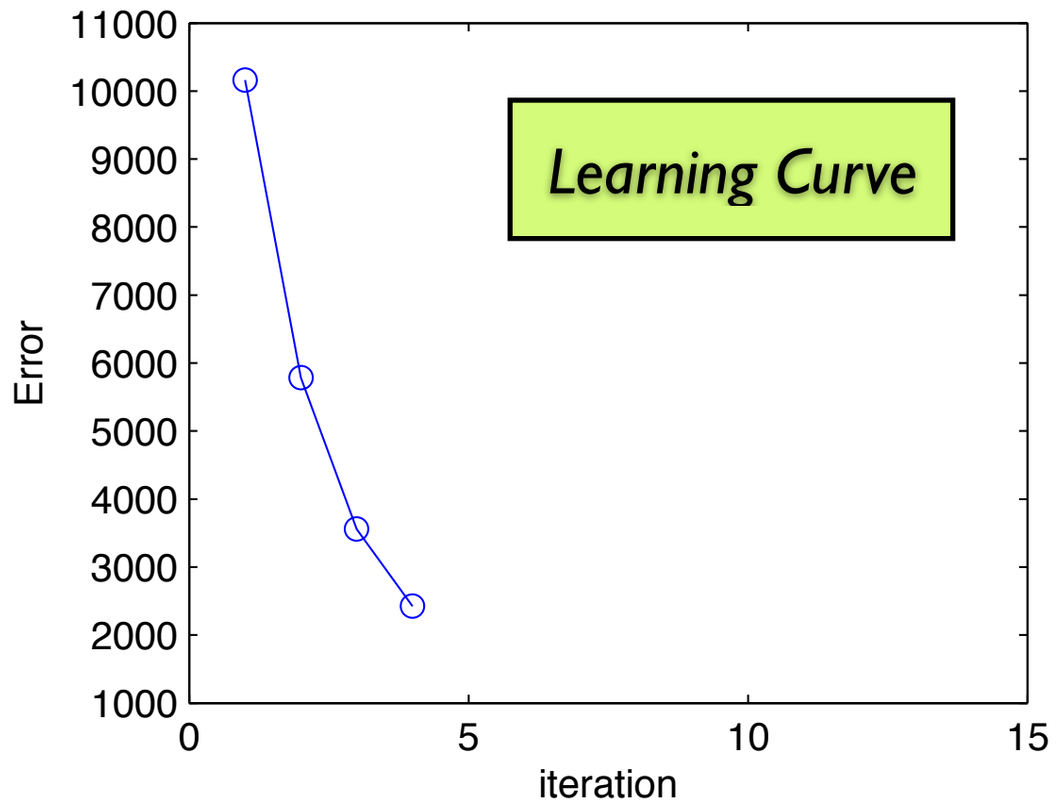
$$\frac{\partial E}{w_i} = \sum_{n=1}^{N} (\mathbf{w}^T \mathbf{x}_n - c_n) x_{i,n}$$

- Epsilon is a small value:

  ε = 0.1/N

- Epsilon too large:
  - learning diverges
- Epsilon too small:
  - convergence slow





Simulation: learning the decision boundary

- Each iteration updates the gradient:

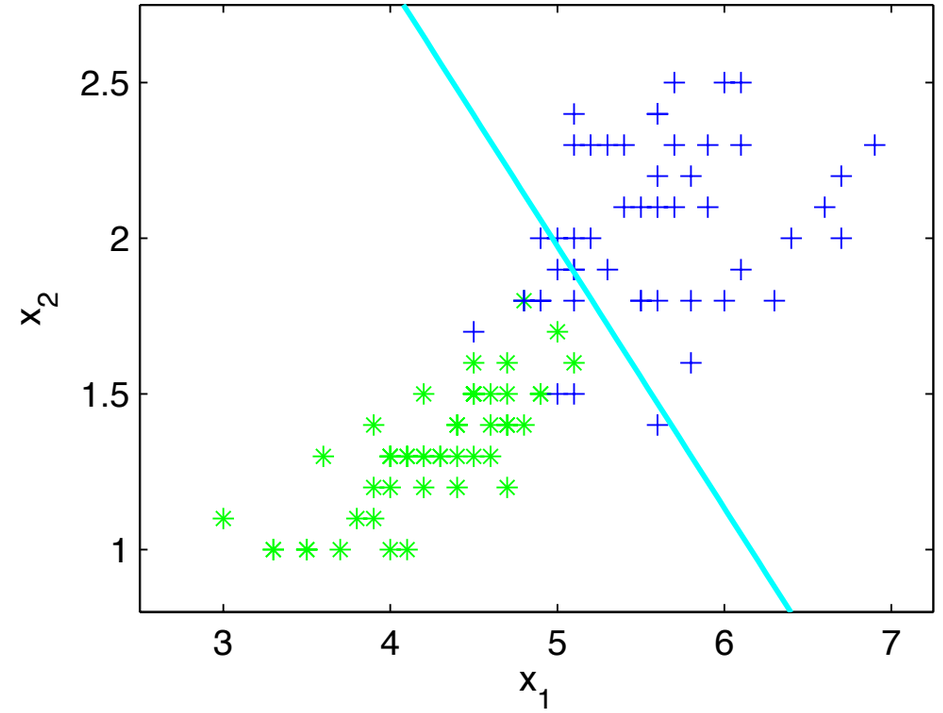
$$w_i^{t+1} = w_i^t - \epsilon \frac{\partial E}{w_i}$$

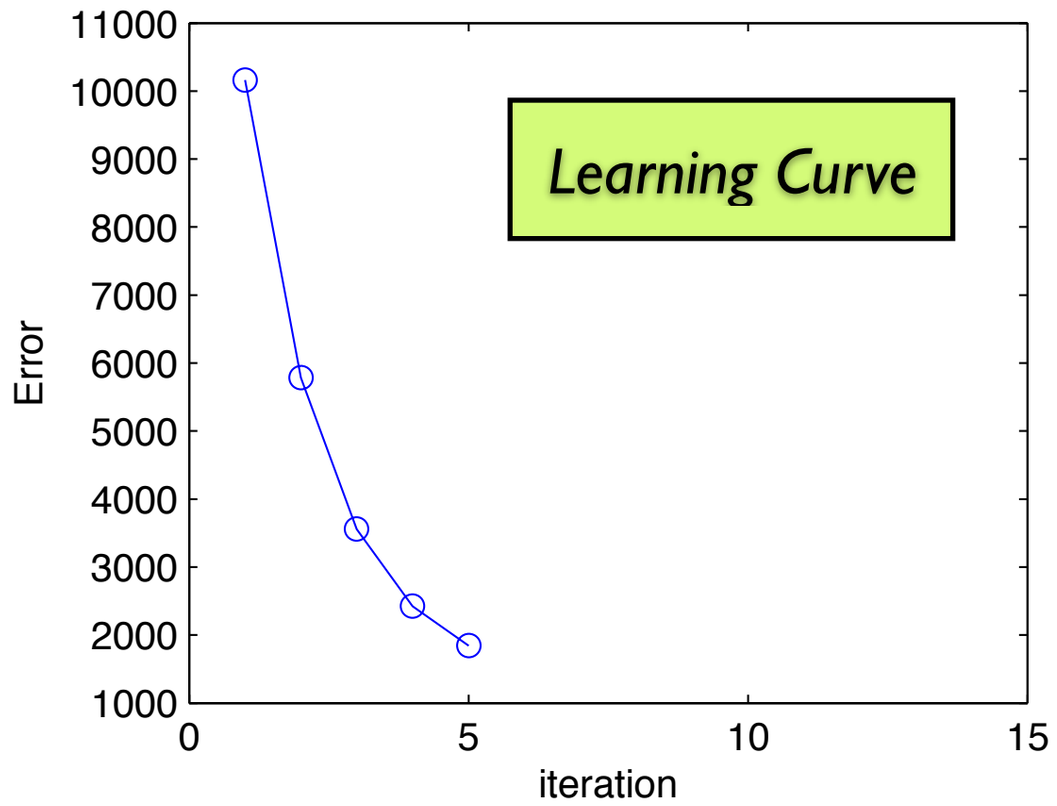
$$\frac{\partial E}{w_i} = \sum_{n=1}^{N} (\mathbf{w}^T \mathbf{x}_n - c_n) x_{i,n}$$

- Epsilon is a small value:

  $\varepsilon = 0.1/N$

- Epsilon too large:
  - learning diverges
- Epsilon too small:
  - convergence slow





Simulation: learning the decision boundary

- Each iteration updates the gradient:

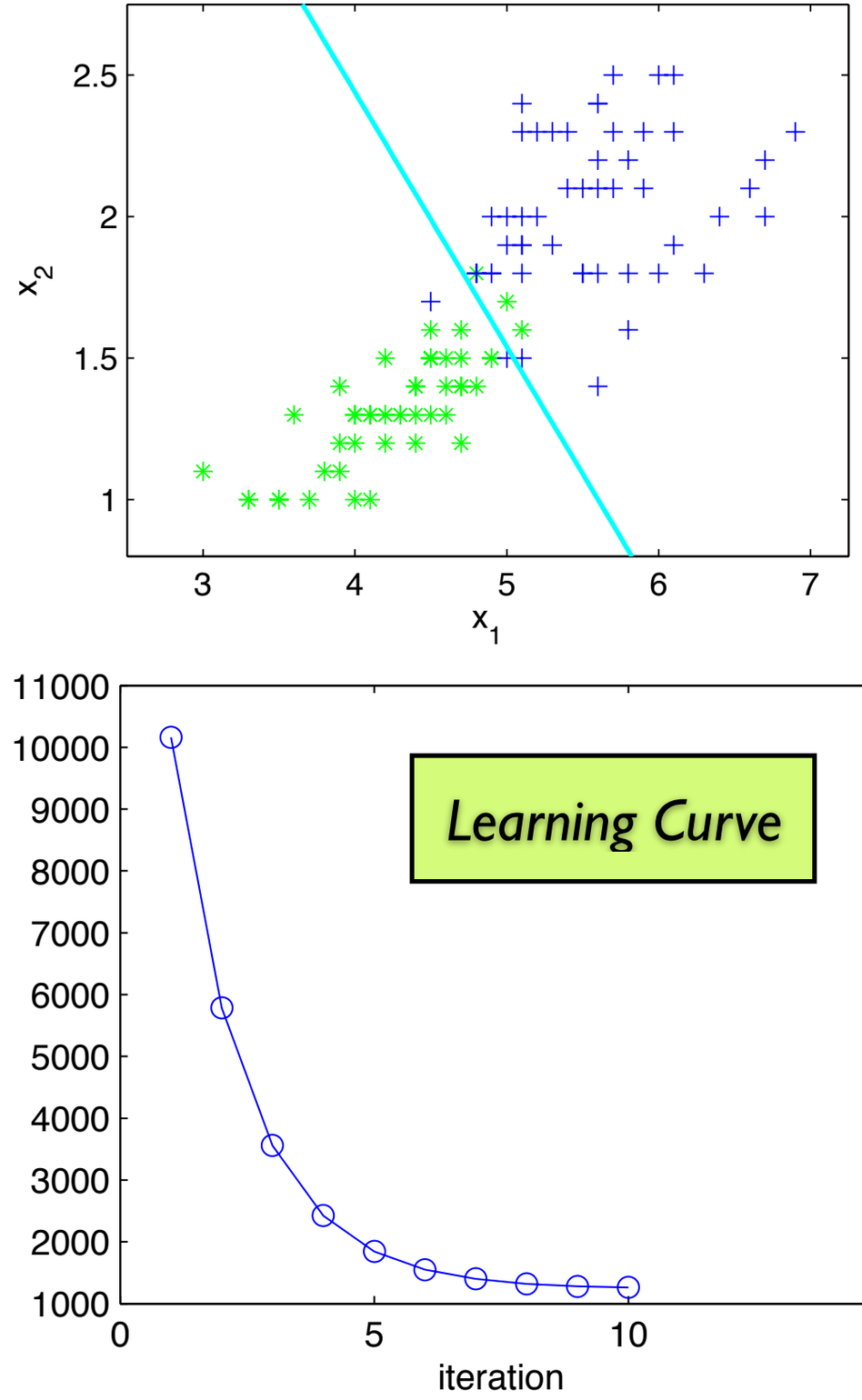
$$w_i^{t+1} = w_i^t - \epsilon \frac{\partial E}{w_i}$$

$$\frac{\partial E}{w_i} = \sum_{n=1}^{N} (\mathbf{w}^T \mathbf{x}_n - c_n) x_{i,n}$$

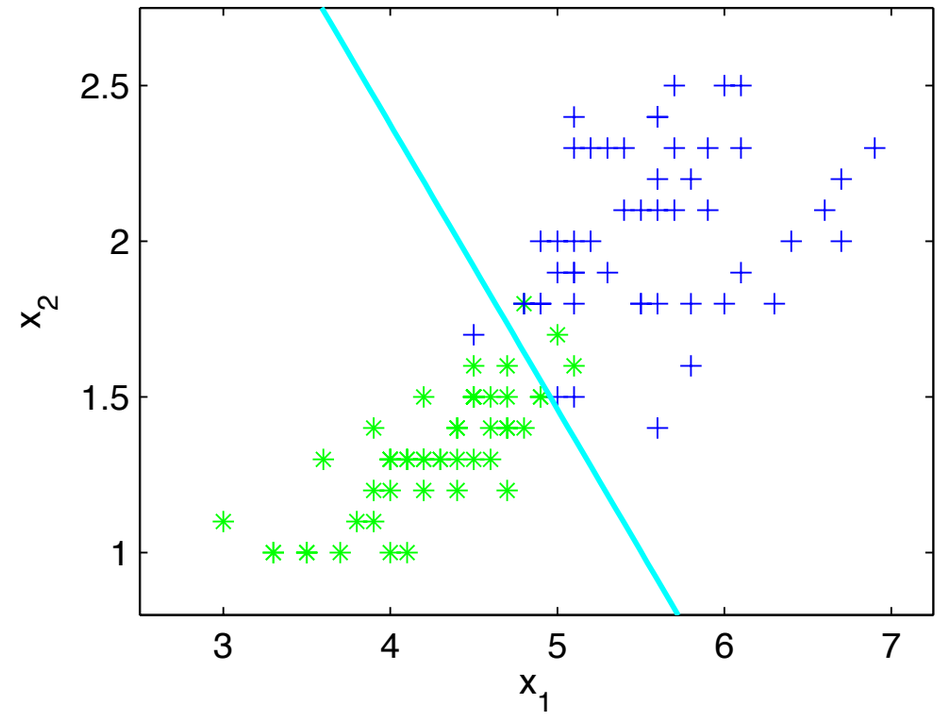
- Epsilon is a small value:

  ε = 0.1/N

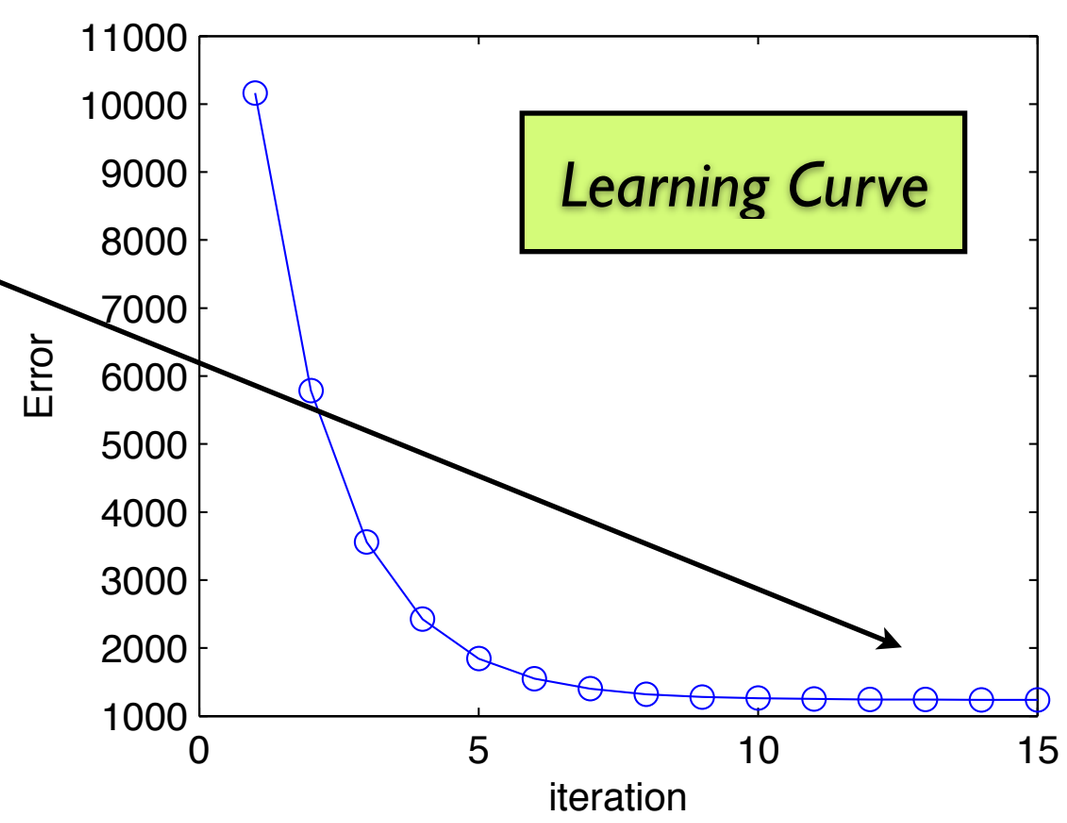
- Epsilon too large:
  - learning diverges
- Epsilon too small:
  - convergence slow



Simulation: learning the decision boundary

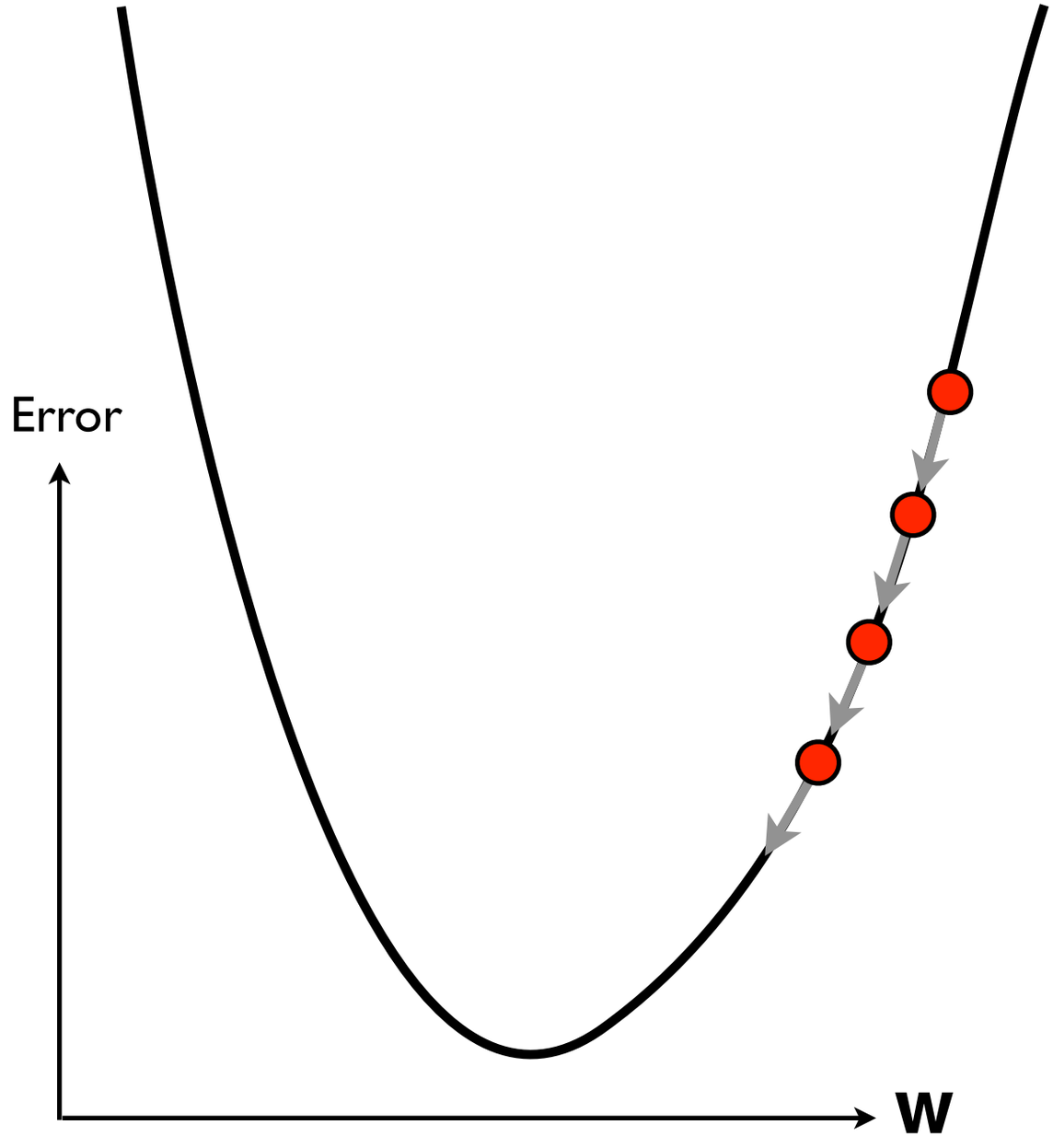


- Learning converges onto the solution that minimizes the error.
- For linear networks, this is guaranteed to converge to the minimum
- It is also possible to derive a closed-form solution (covered later)



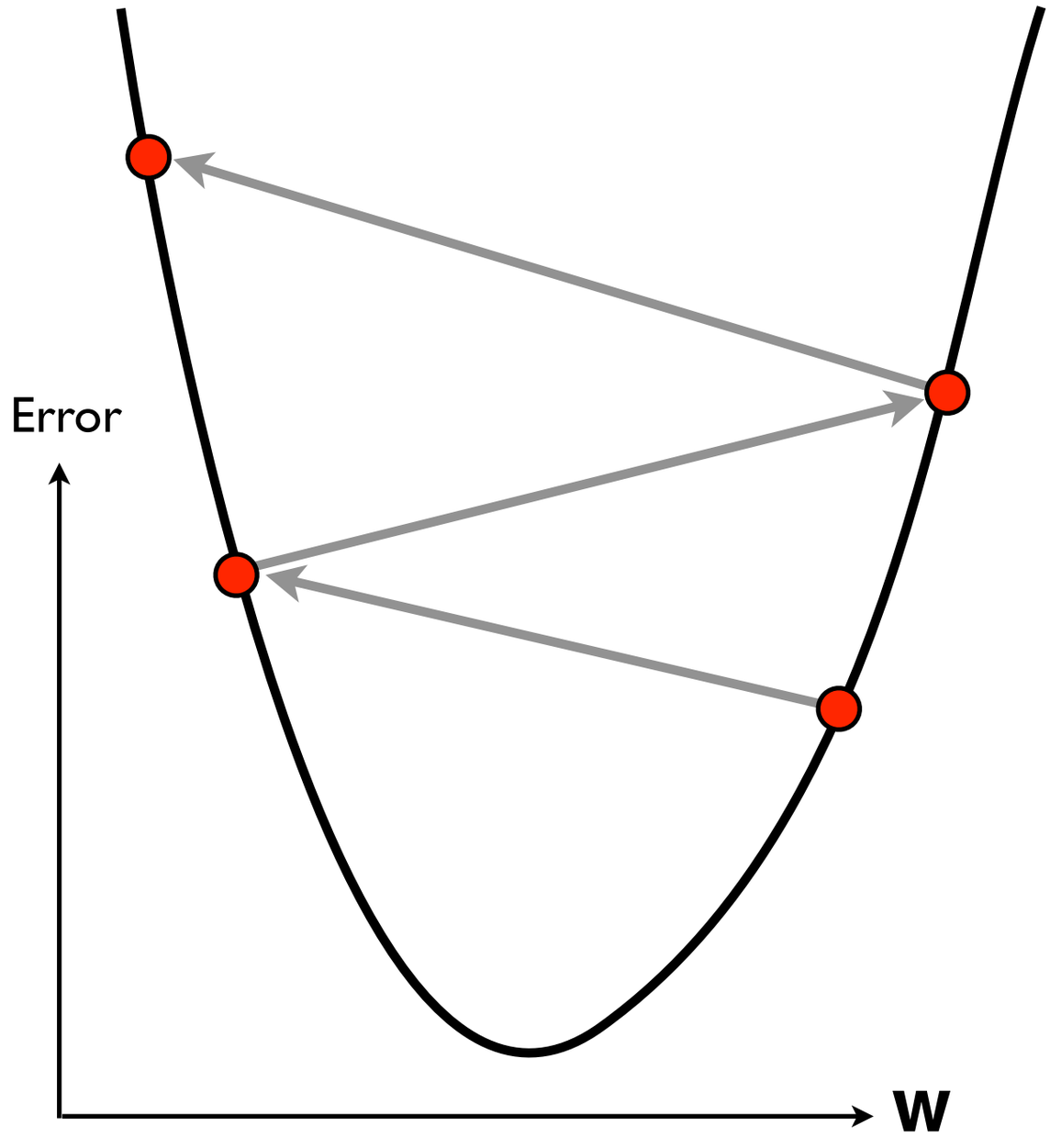
Learning is slow when epsilon is too small

- Here, larger step sizes would converge more quickly to the minimum



Divergence when epsilon is too large

- If the step size is too large, learning can oscillate between different sides of the minimum



Multi-layer networks

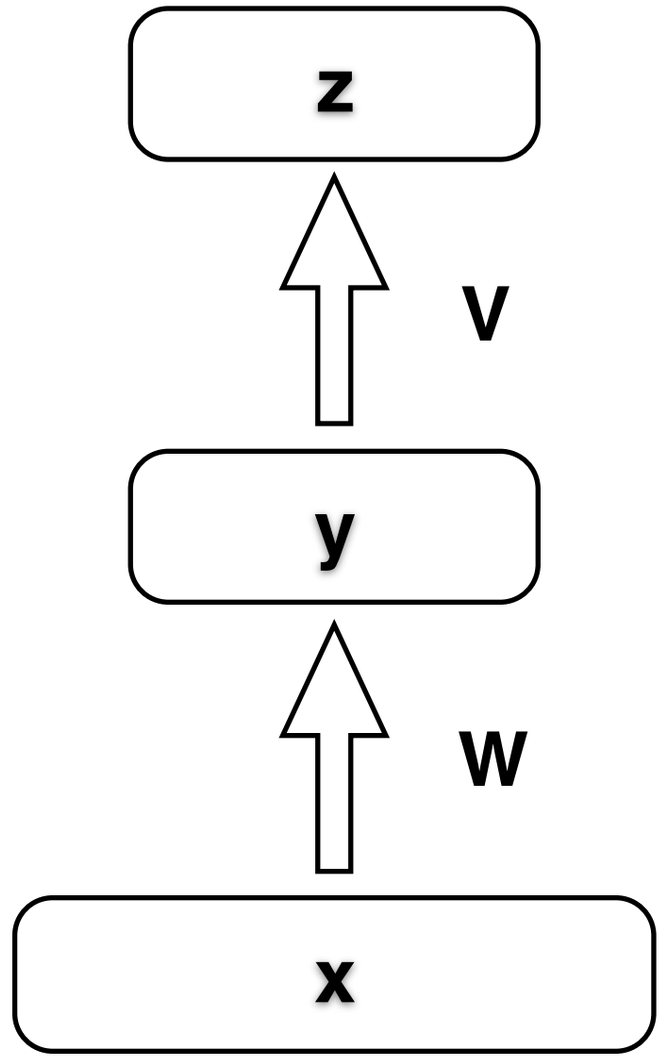
- Can we extend our network to multiple layers? We have:

$$
\begin{aligned}
y_j &= \sum_i w_{i,j} x_i \\
z_j &= \sum_k v_{j,k} y_j \\
&= \sum_k v_{j,k} \sum_i w_{i,j} x_i
\end{aligned}
$$

- Or in matrix form

$$
\begin{aligned}
\mathbf{z} &= \mathbf{V}\mathbf{y} \\
&= \mathbf{V}\mathbf{W}\mathbf{x}
\end{aligned}
$$

- Thus a two-layer linear network is equivalent to a one-layer linear network with weights **U=VW**.
- It is *not* more powerful.

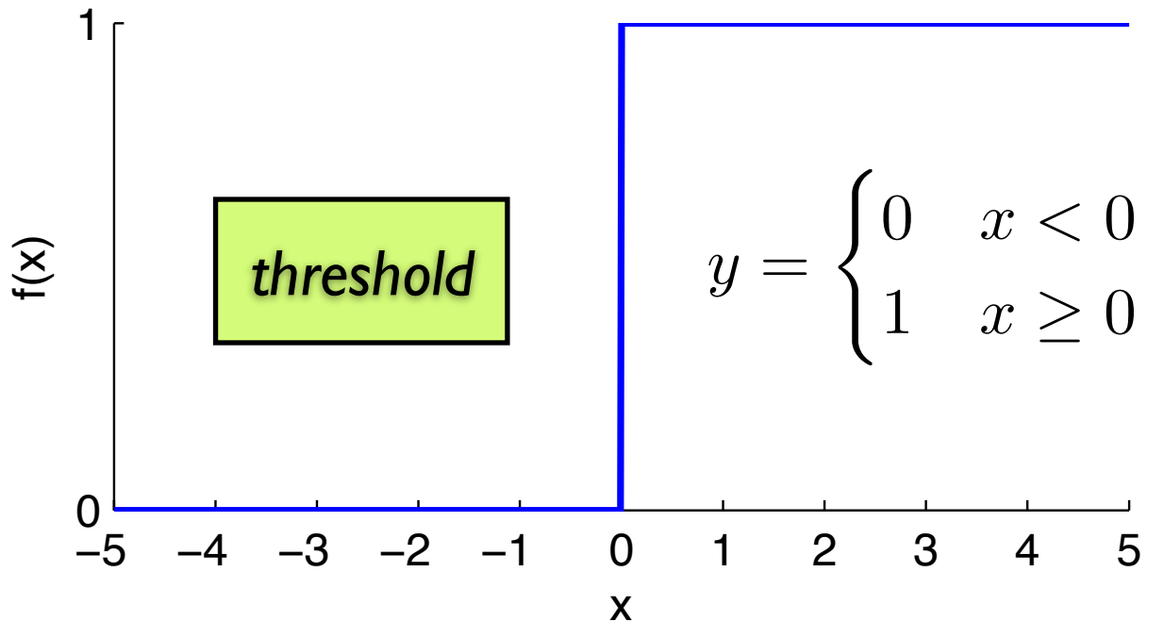


How do we address this?

Non-linear neural networks

- Idea introduce a non-linearity:

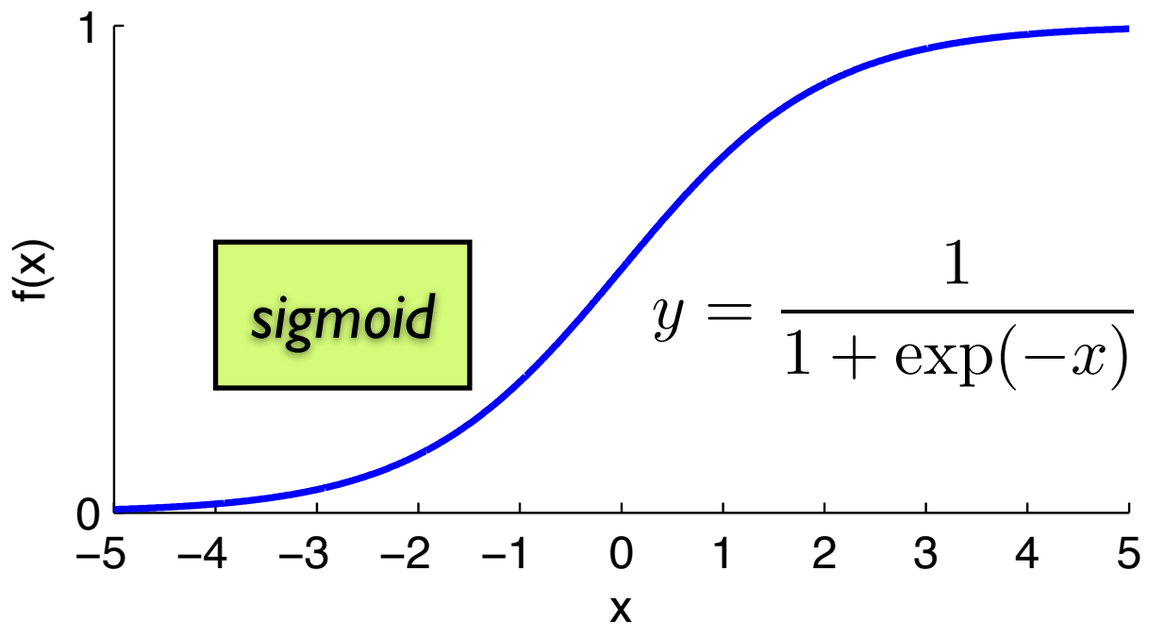
$$y_j = f(\sum_i w_{i,j} x_i)$$



- Now, multiple layers are *not* equivalent

$$\begin{aligned} z_j &= f(\sum_k v_{j,k}\, y_j) \\ &= f(\sum_k v_{j,k}\, f(\sum_i w_{i,j} x_i)) \end{aligned}$$

- Common nonlinearities:
  - threshold
  - sigmoid



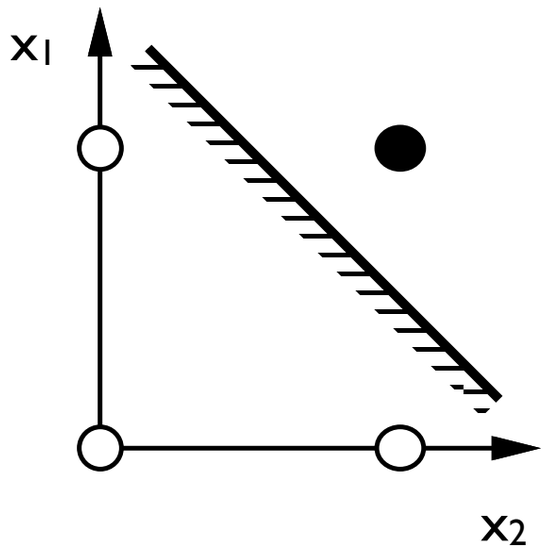
Posterior odds interpretation of a sigmoid

- The sigmoid can be interpreted as the posterior odds between two hypotheses

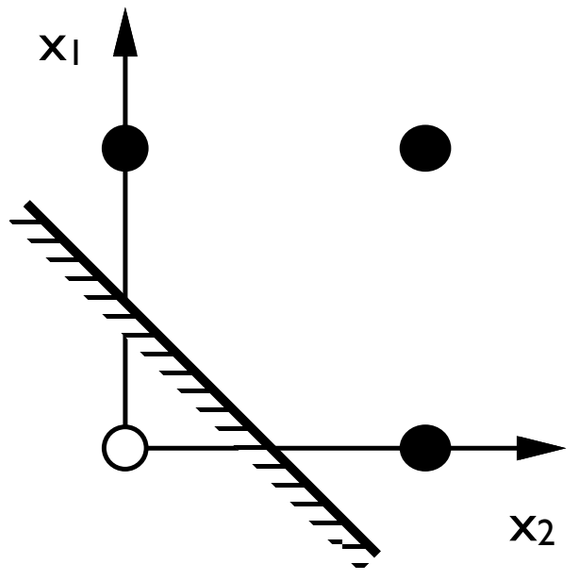
$$
\begin{aligned}
p(C_1|\mathbf{x}) &= \frac{p(\mathbf{x}|C_1)p(C_1)}{p(\mathbf{x}|C_1)p(C_1) + p(\mathbf{x}|C_2)p(C_2)} \\
&= \frac{1}{1+\exp(a)} = \sigma(a) \\
\text{where } a &= \log \frac{p(\mathbf{x}|C_1)p(C_1)}{p(\mathbf{x}|C_2)p(C_2)}
\end{aligned}
$$

- "a" is the log odds ratio of $p(C_1|\mathbf{x})$ and $p(C_2|\mathbf{x})$.

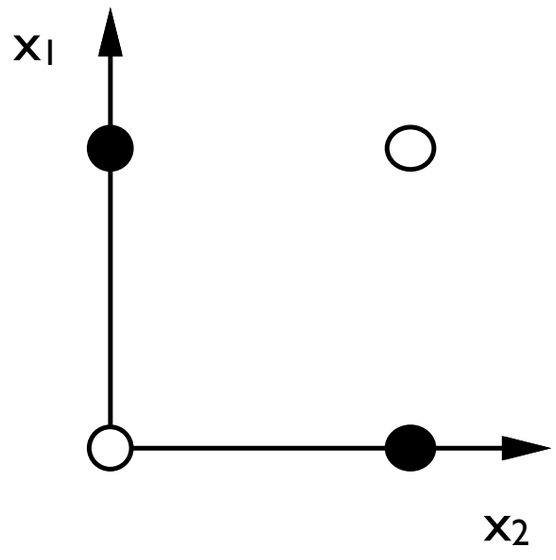
Modeling logical operators



x_1 AND x_2



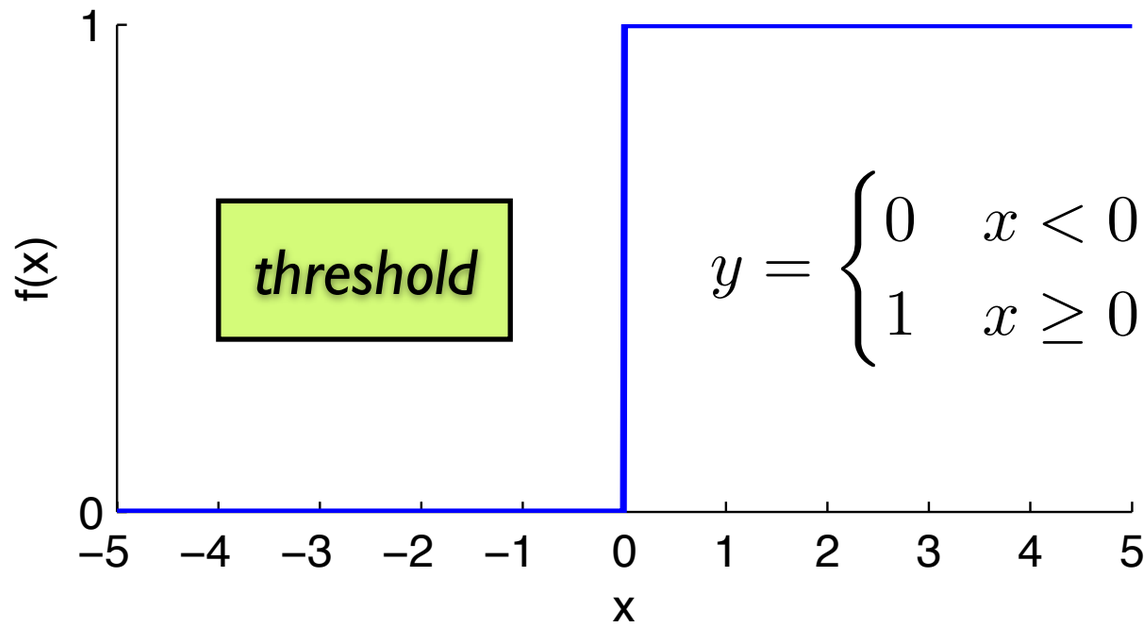
x_1 OR x_2



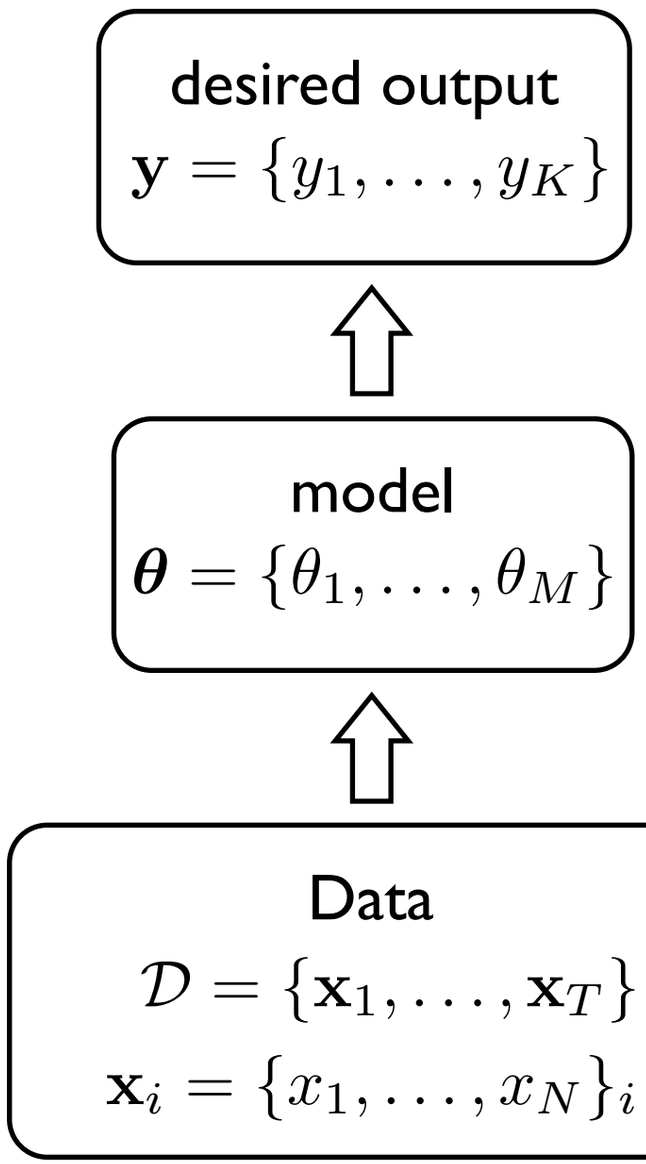
x_1 XOR x_2

- A one-layer binary-threshold network can implement the logical operators AND and OR, but not XOR.
- Why not?

$$y_j = f(\sum_i w_{i,j} x_i)$$



The general classification/regression problem



for classification:

$$y_i = \begin{cases} 1 & \text{if } \mathbf{x}_i \in C_i \equiv \text{class } i, \\ 0 & \text{otherwise} \end{cases}$$

regression for arbitrary y.

model (e.g. a decision tree) is defined by M parameters, **e.g. a multi-layer neural network.**

input is a set of T observations, each an N-dimensional vector (binary, discrete, or continuous)

Given data, we want to learn a model that can correctly classify novel observations **or map the inputs to the outputs**

A general multi-layer neural network

- Error function is defined as before, where we use the target vector t_n to define the desired output for network output y_n.

$$E = \frac{1}{2}\sum_{n=1}^{N}(\mathbf{y}_n(\mathbf{x_n}, \mathbf{W_{1:L}}) - \mathbf{t}_n)^2$$

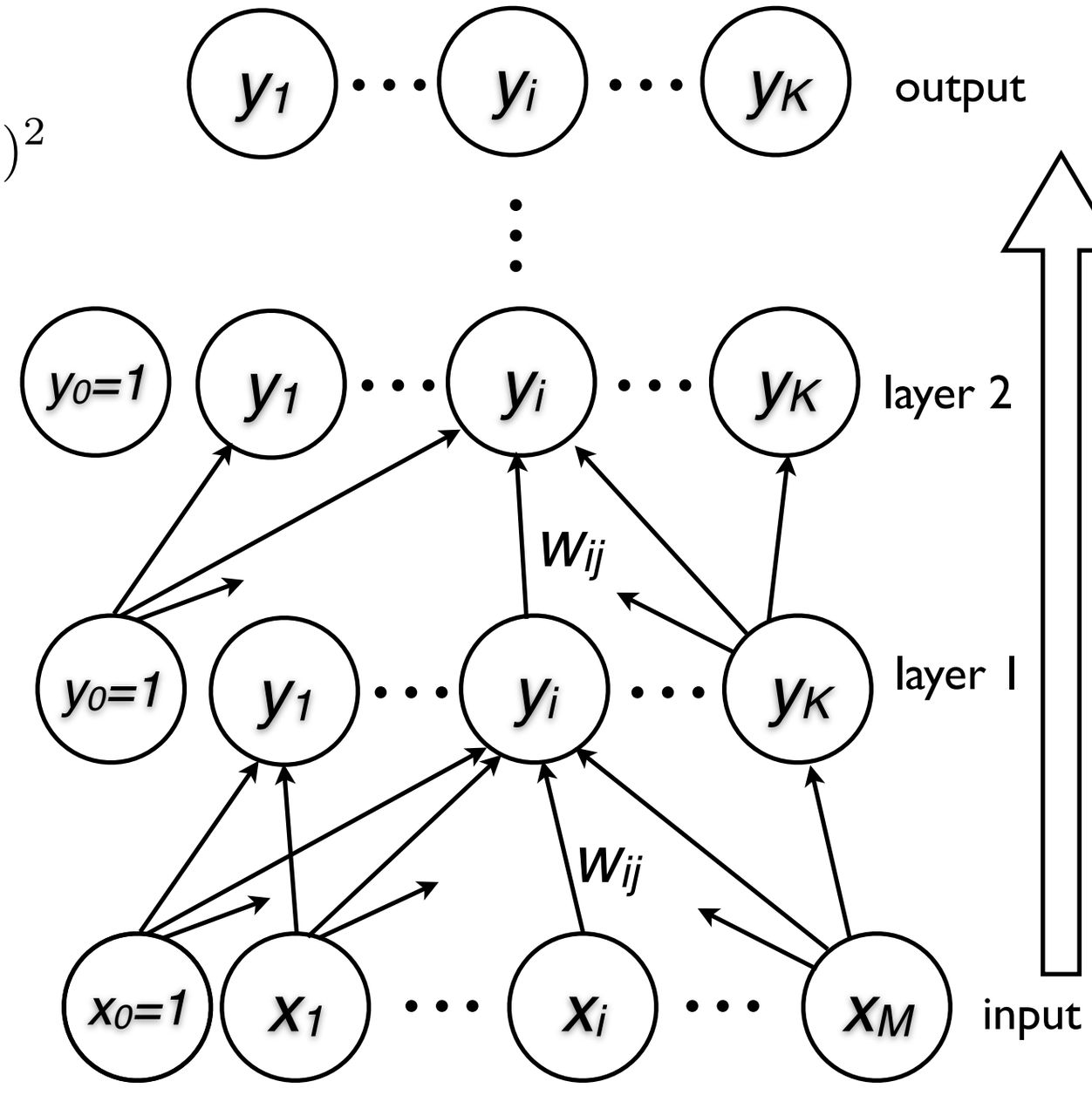
- The "forward pass" computes the outputs at each layer:

$$y_j^l = f(\sum_i w_{i,j}^l \, y_j^{l-1})$$

$$l = \{1, \ldots, L\}$$

$$\mathbf{x} \equiv \mathbf{y}^0$$

$$\text{output} = \mathbf{y}^L$$

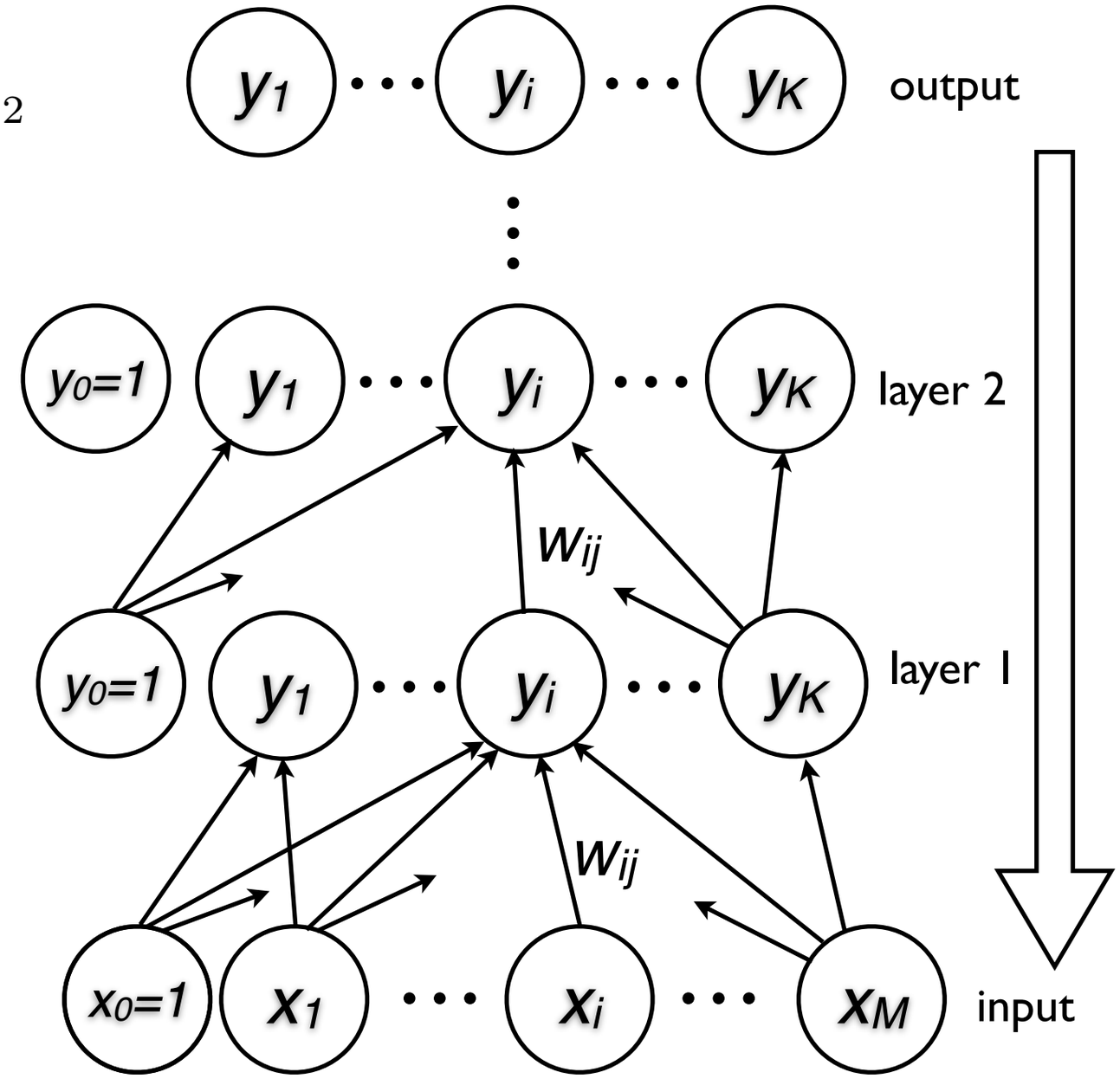


Deriving the gradient for a sigmoid neural network

- Mathematical procedure for train is gradient descient: same as before, except the gradients are more complex to derive.

New problem: local minima

$$E = \frac{1}{2}\sum_{n=1}^{N}(\mathbf{y}_n(\mathbf{x_n}, \mathbf{W_{1:L}}) - \mathbf{t}_n)^2$$



- Convenient fact for the sigmoid non-linearity:

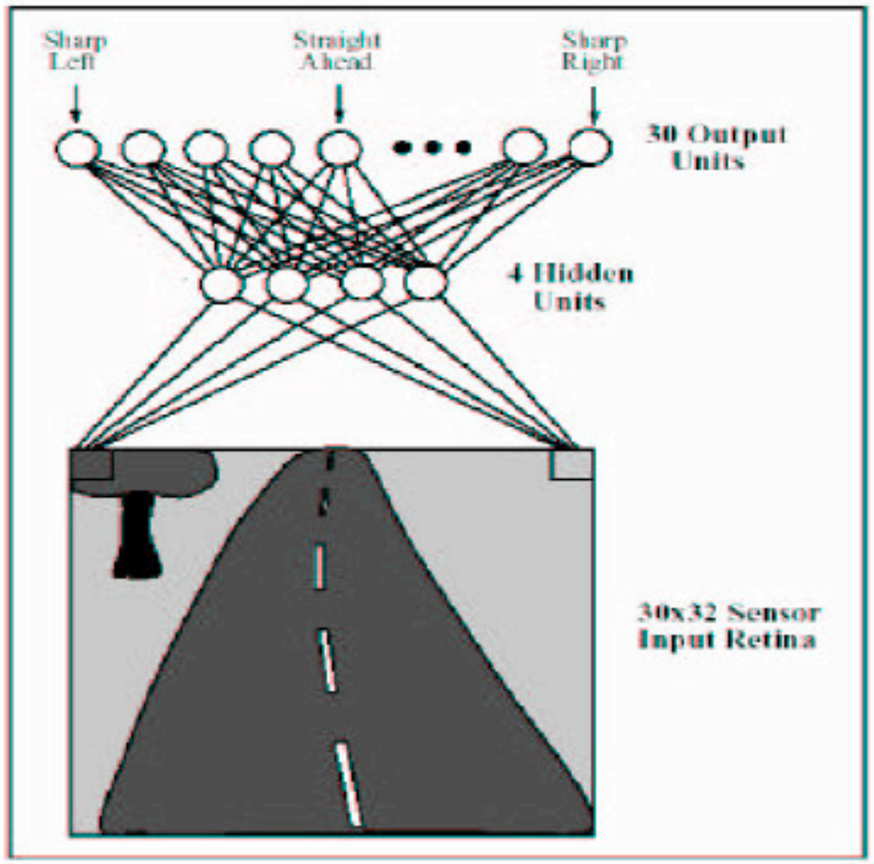
$$\frac{d\sigma(x)}{dx} = \frac{d}{dx}\frac{1}{1+\exp(-x)}$$
$$= \sigma(x)(1-\sigma(x))$$

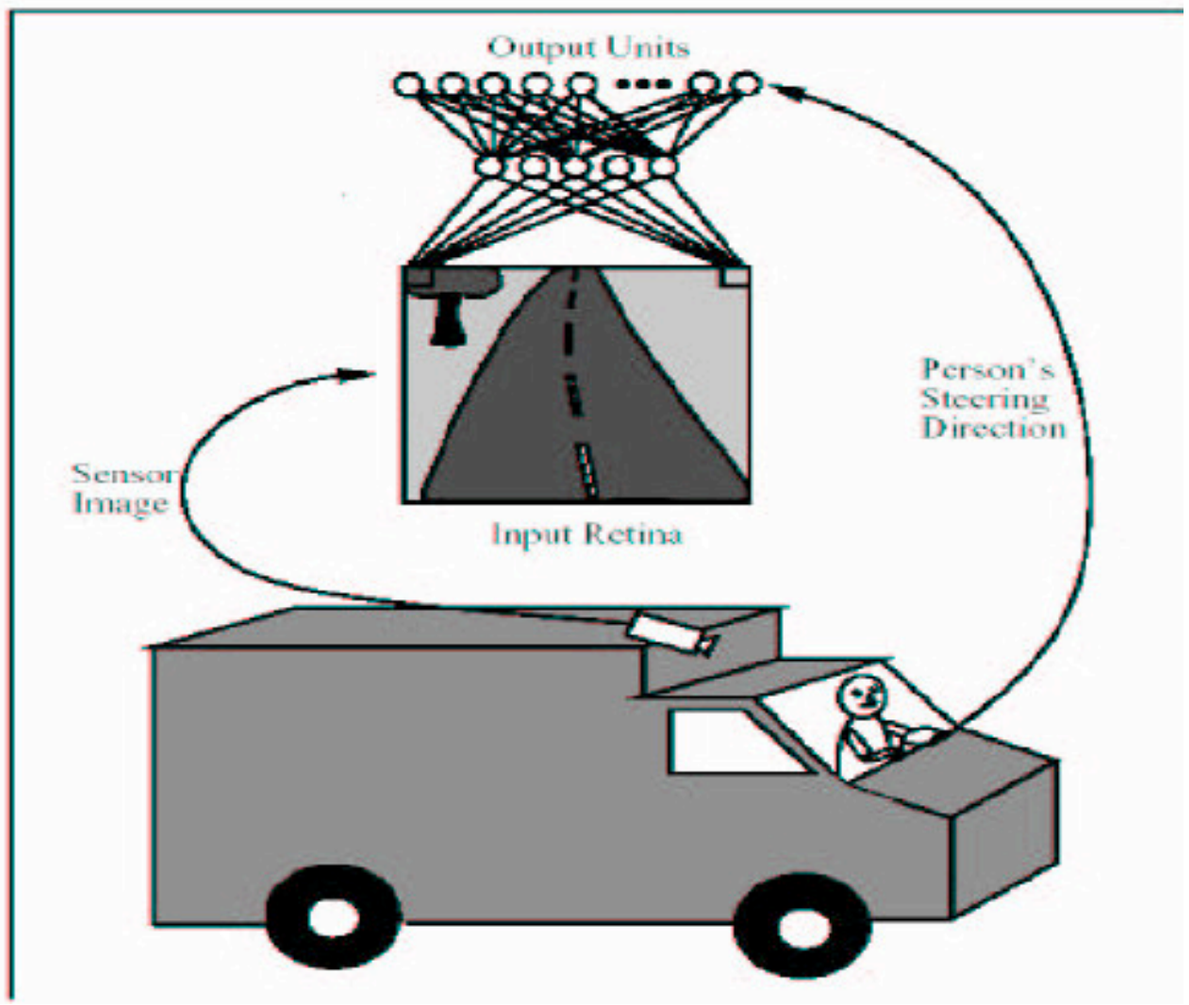
- backward pass computes the gradients: *back-propagation*

$$\mathbf{W}_{t+1} = \mathbf{W}_t + \epsilon\frac{\partial E}{\mathbf{W}}$$

Applications: Driving (output is analog: steering direction)

network with 1 layer
(4 hidden units)





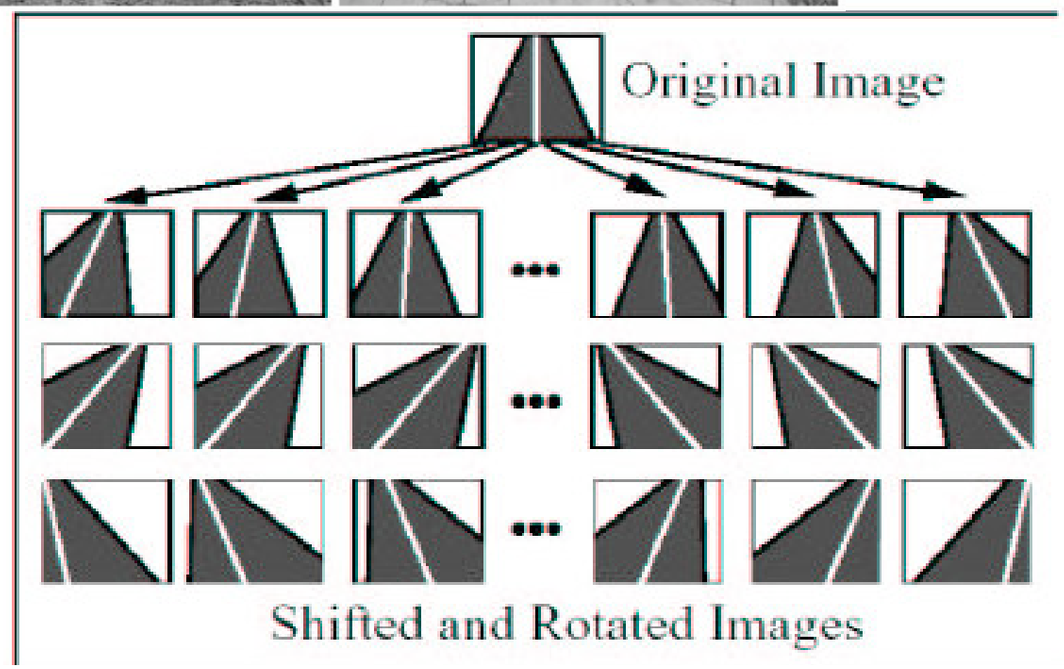
- Learns to drive on roads
- Demonstrated at highway speeds over 100s of miles

D. Pomerleau. *Neural network perception for mobile robot guidance.* Kluwer Academic Publishing, 1993.

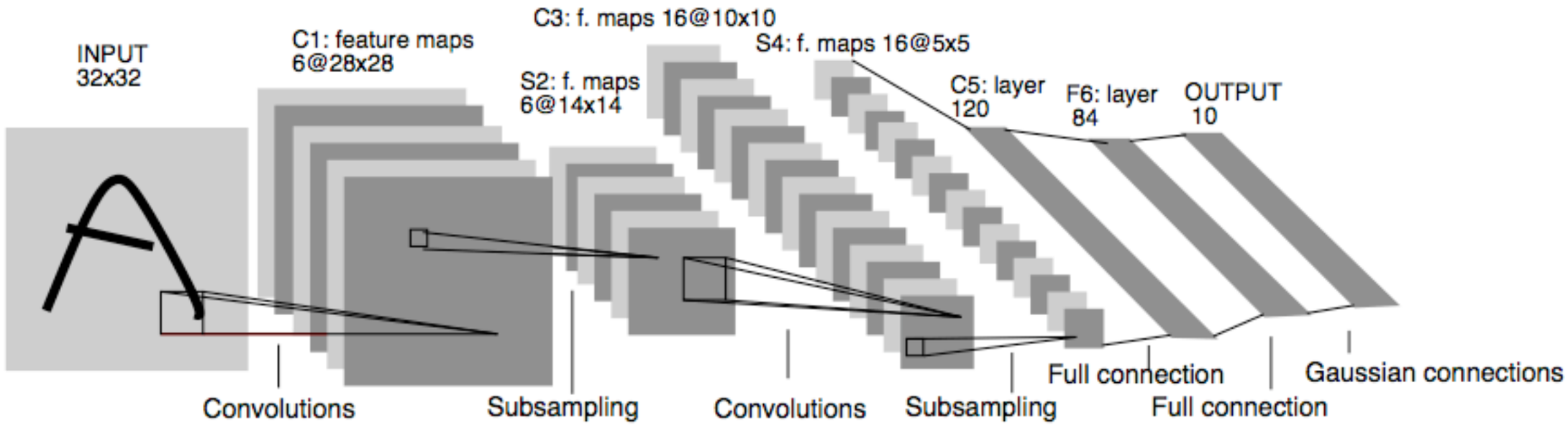
Real image input is augmented to avoid overfitting

Training data:
Images +
corresponding
steering angle

Important:
Conditioning of
training data to
generate new
examples → avoids
overfitting



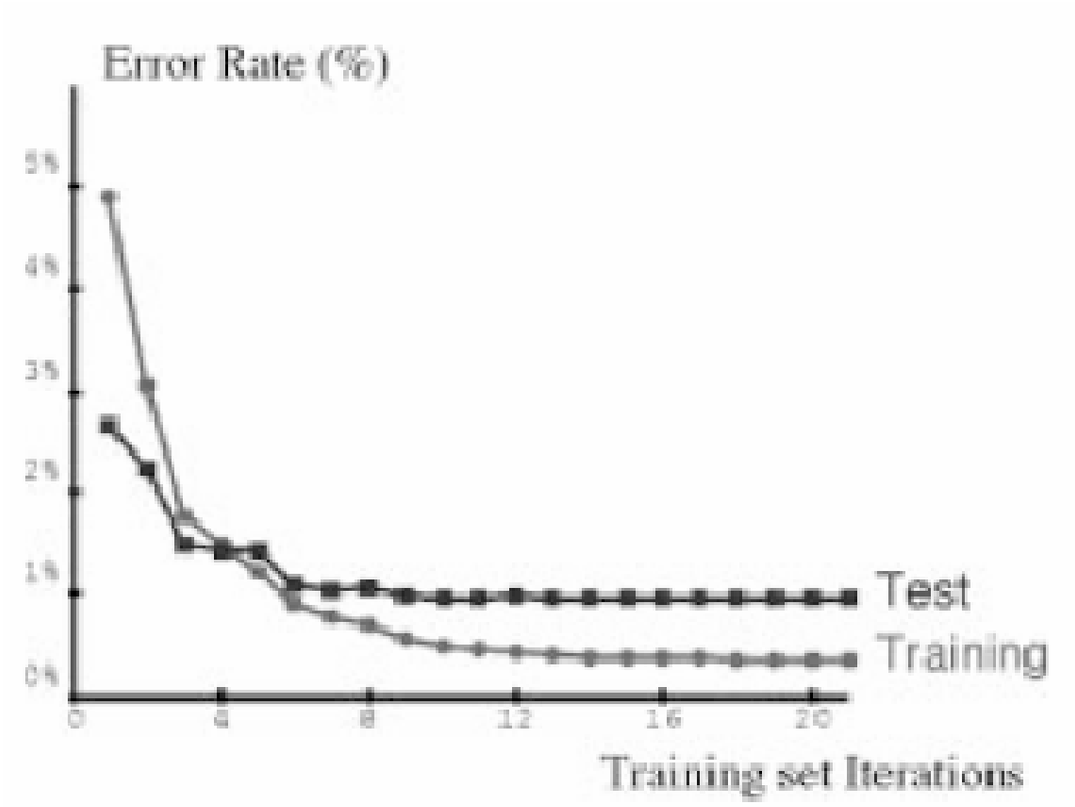
Hand-written digits: LeNet



- Takes as input image of handwritten digit
- Each pixel is an input unit
- Complex network with many layers
- Output is digit class

- Tested on large (50,000+) database of handwritten samples
- Real-time
- Used commercially

LeNet

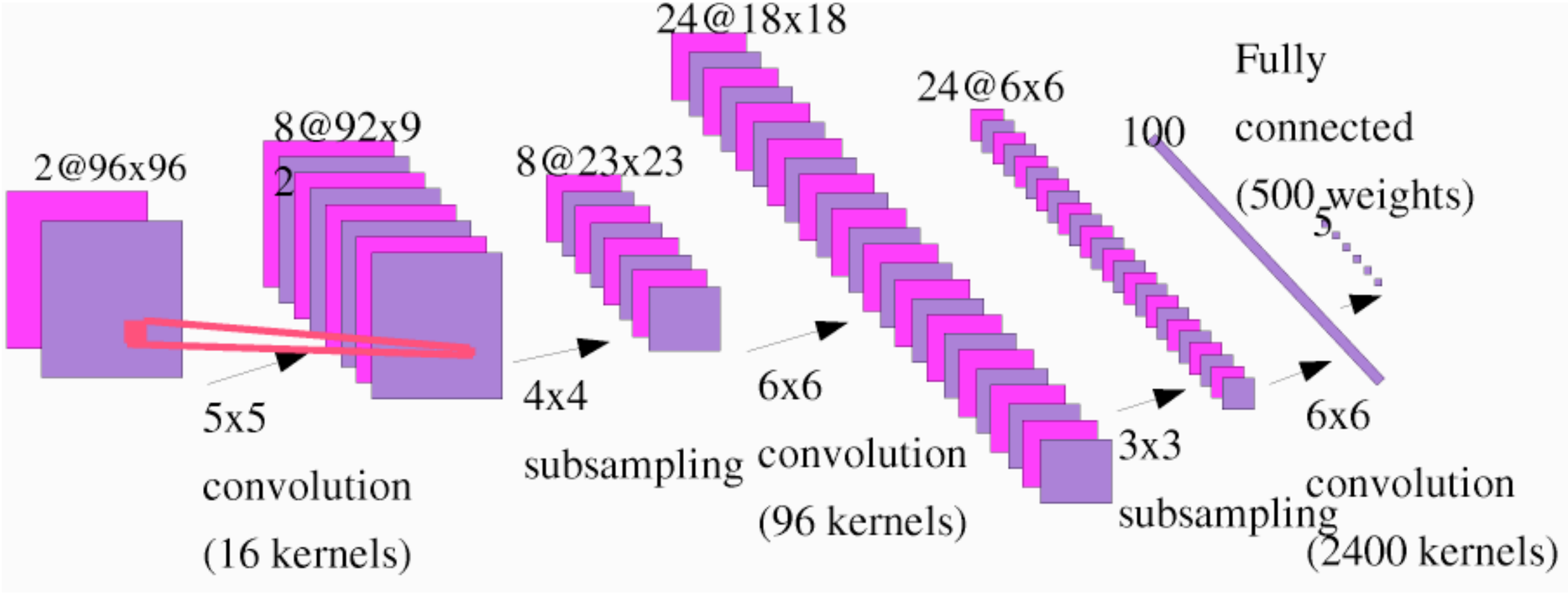


Very low error rate (<< 1%

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, november 1998.

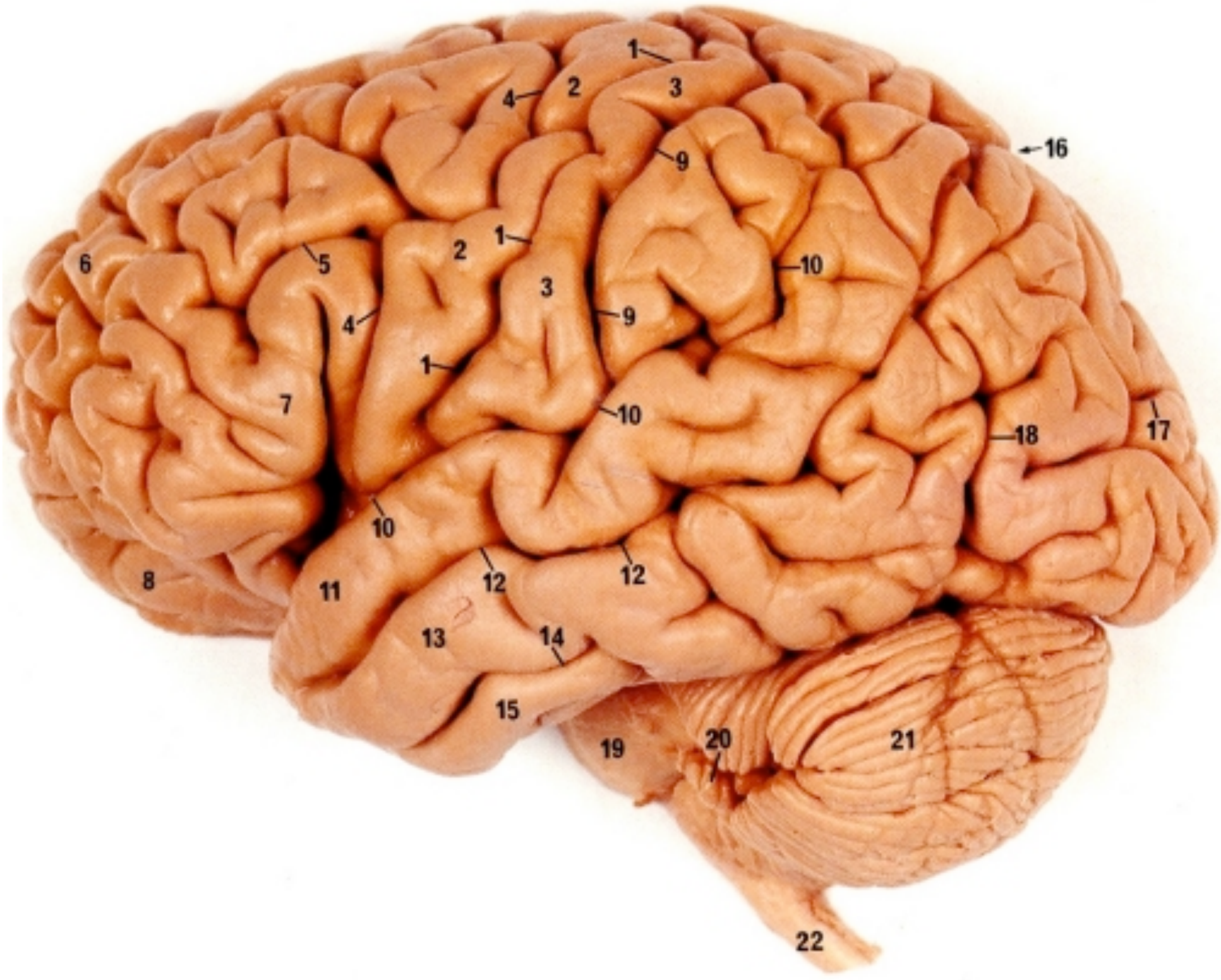
http://yann.lecun.com/exdb/lenet/

Object recognition



- LeCun, Huang, Bottou (2004). Learning Methods for Generic Object Recognition with Invariance to Pose and Lighting. Proceedings of CVPR 2004.
- http://www.cs.nyu.edu/~yann/research/norb/

Is AI is still at the balloon stage?

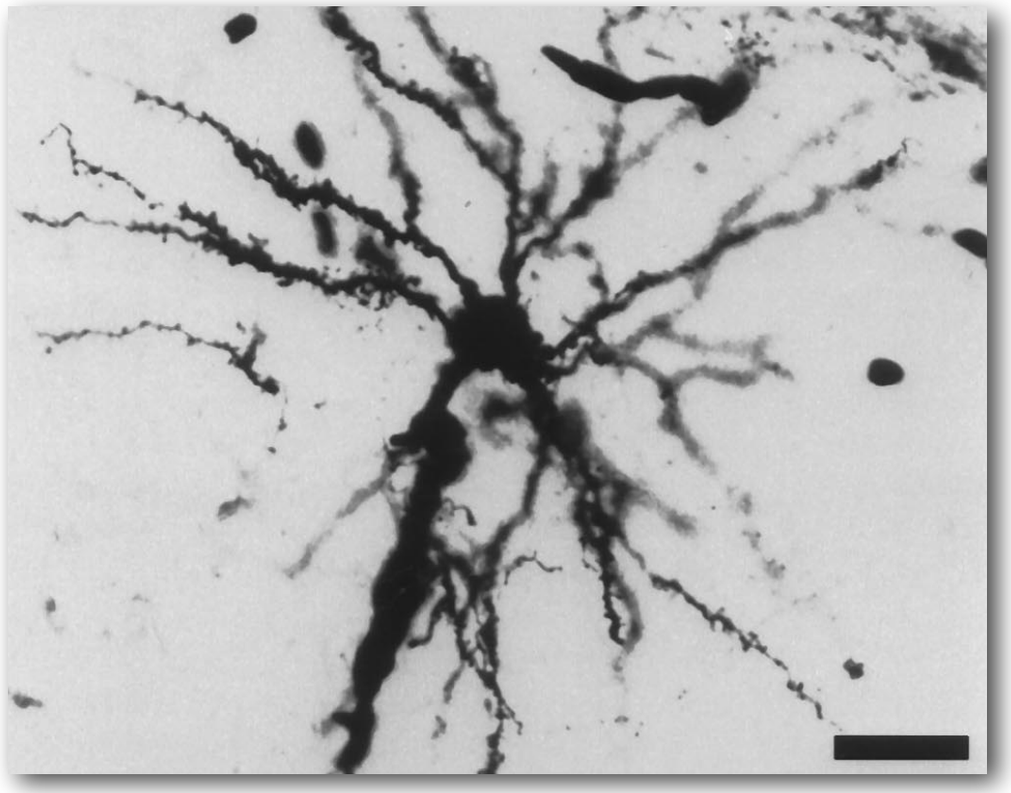


Limitations of this approach

AI research focuses on underlying principles. Russel & Norvig:

Aeronautical engineering texts do not define the goal of their field as making "machines that fly so exactly like pigeons that they can fool even other pigeons."

My own neural recordings from my PhD studies



A neuron in an auditory brain area

Neural response

Sound waveform

Brains vs computers

Brains (adult cortex)

- surface area: 2500 cm^2
- squishy

- neurons: 20 billion
- synapses: 240,000 billion

- neuron size: 15,000 nm
- synapse size: 1,000 nm

- synaptic OPS: 30,000 billion

Intel quad core (Nehalem)

- surface area: 107 mm^2
- crystalline

- transistors: 0.82 billion

- transistor size: 45 nm

- FLOPS: 45 billion

Deep Blue: 512 processors, 1 TFLOP

Brains vs computers: energy efficiency

Brains (adult cortex)

- surface area: 2500 cm^2
- squishy

- neurons: 20 billion
- synapses: 240,000 billion

- neuron size: 15,000 nm
- synapse size: 1,000 nm

- synaptic OPS: 30,000 billion

- power usage: ~12 W
- **2500 GFLOPS/W**

Intel quad core (Nehalem, 2008)

- surface area: 107 mm^2
- crystalline

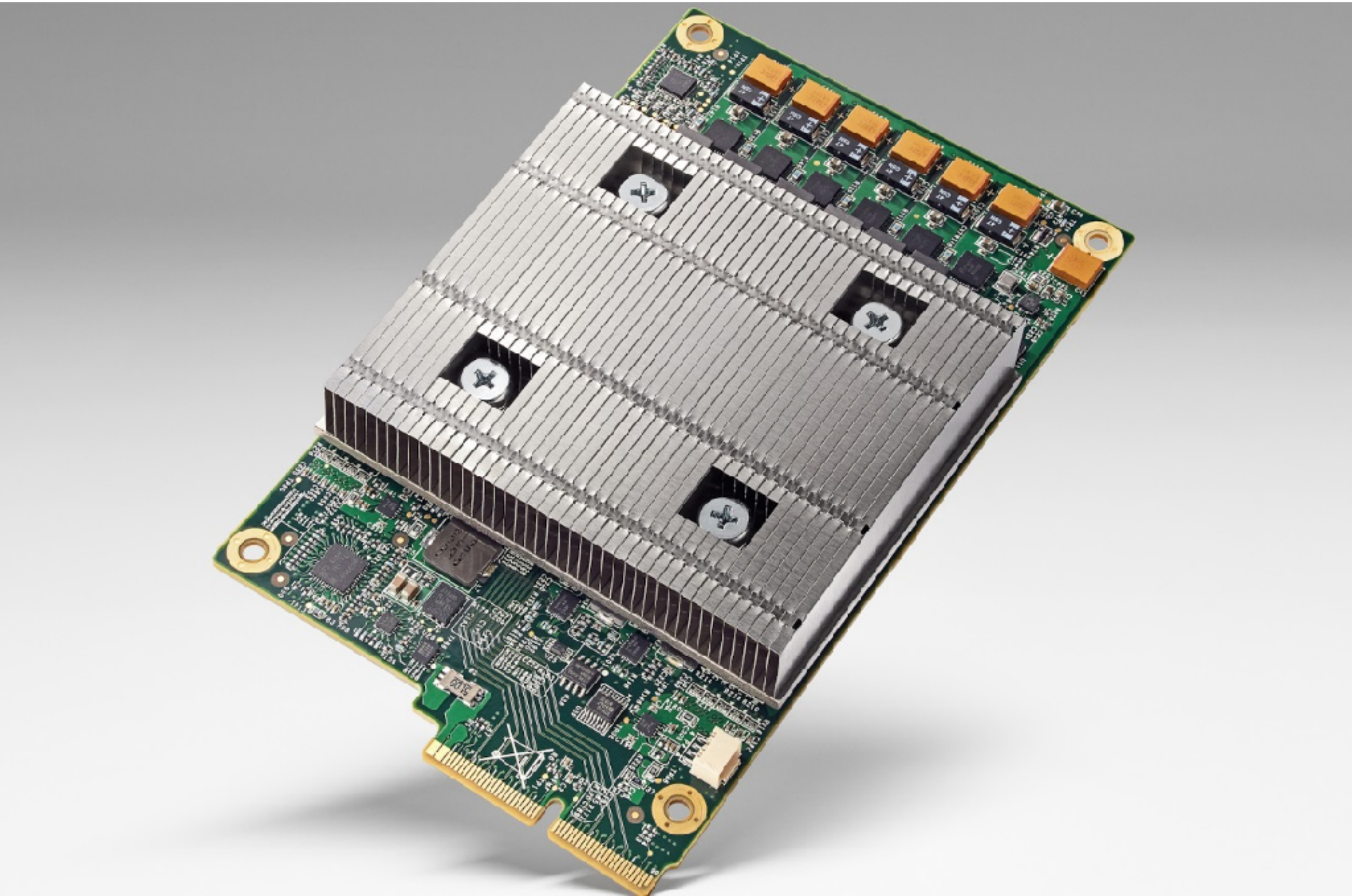
- transistors: 0.82 billion

- transistor size: 45 nm

- FLOPS: 45 billion

- power usage: 60 W
- **0.75 GFLOPS/W**

Google's Deep Learning Chip: The Tensor Processing Unit



Brains vs computers: energy efficiency

Brains (adult cortex)

- surface area: 2500 cm^2
- squishy

- neurons: 20 billion
- synapses: 240,000 billion

- neuron size: 15,000 nm
- synapse size: 1,000 nm

- synaptic Terra OPS: ~30

- power usage: ~12 W
- **2.5 TOPS/W**

Google TPU 2.0

- surface area: ~331 mm^2
- crystalline

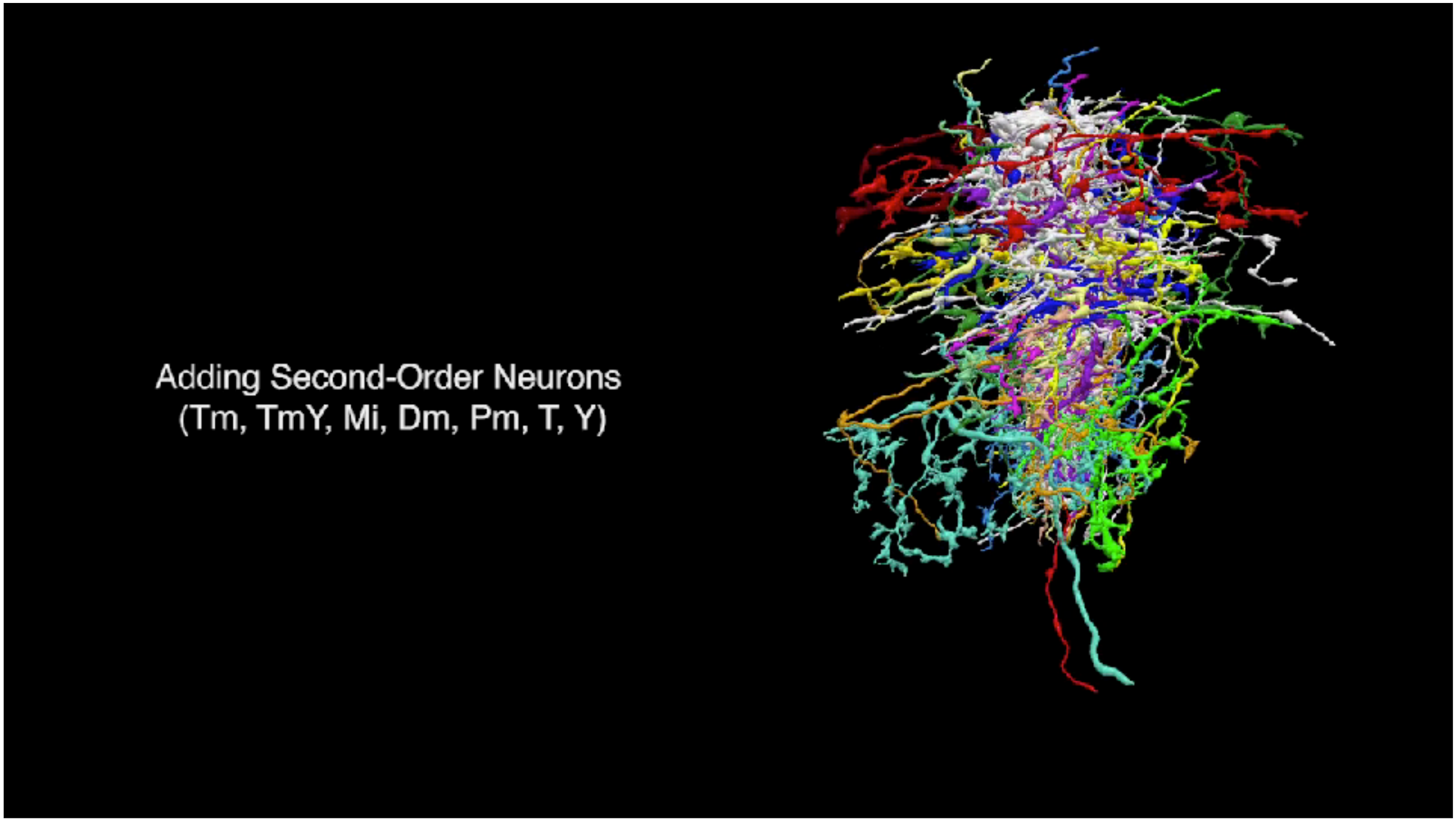
- transistors: 2-2.5 billion

- transistor size: 28 nm

- Terra OPS (8 bit): 92
  (TPU 2.0 is 180 teraops)
- power usage: 40 W
- **2.3 TOPS/W**

  (4.5 for TPU 2.0 assuming 40W)

Reconstruction of neural motion processing circuits in a fly



A visual motion detection circuit suggested by *Drosophila* connectomics

379 neurons; 8,637 chemical synaptic connections

Summary

- Decision boundaries
  - Bayes optimal
  - linear discriminant
  - linear separability
- Classification vs regression
- Optimization by gradient descent
- Degeneracy of a multi-layer linear network
- Non-linearities:: threshold, sigmoid, others?
- Issues:
  - very general architecture, can solve many problems
  - large number of parameters: need to avoid overfitting
  - usually requires a large amount of data, or special architecture
  - local minima, training can be slow, need to set stepsize