

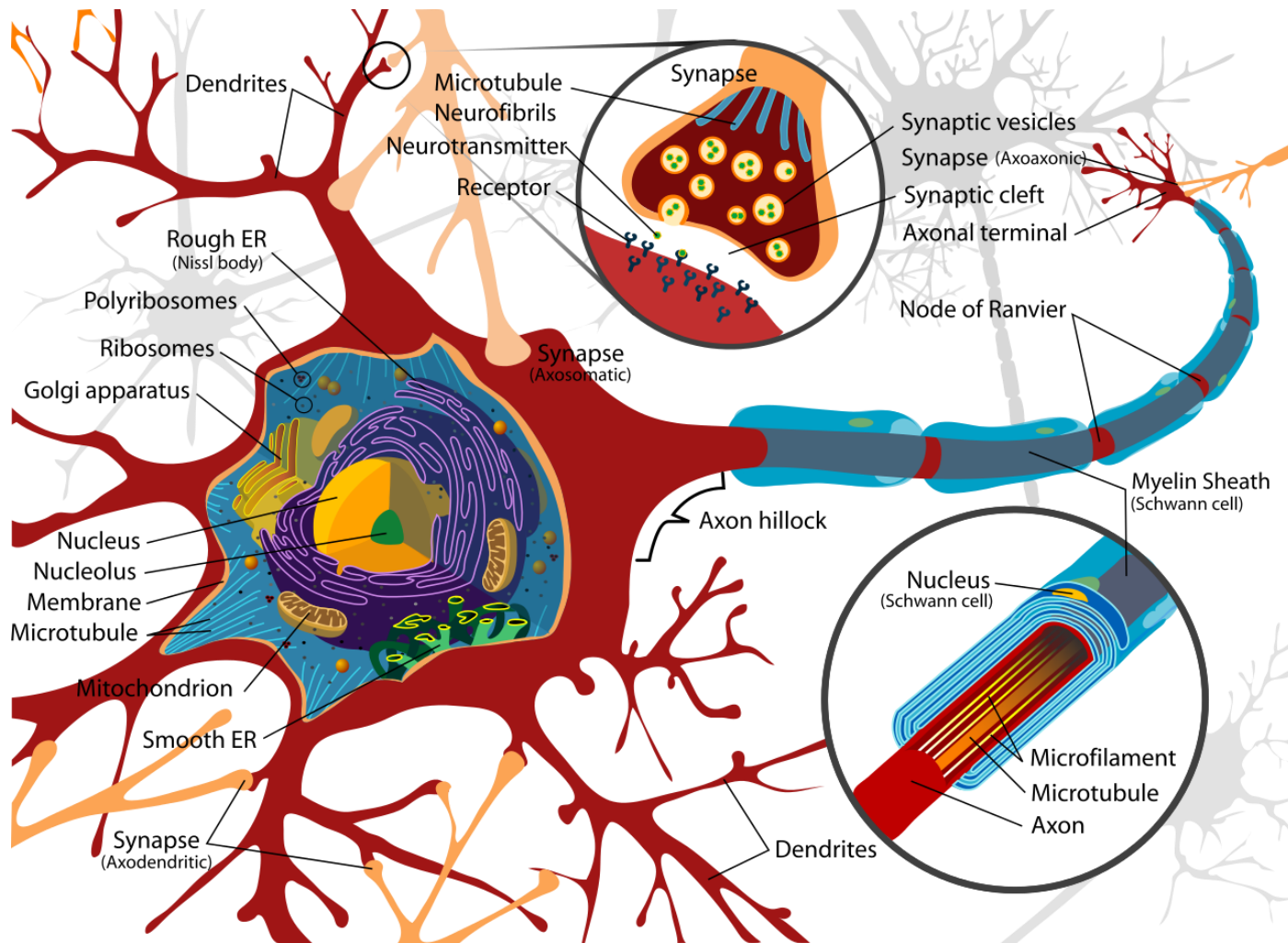


Artificial Neural Networks

Robust approach to approximating

1. Real-valued target functions
 2. Discrete-valued target functions
 3. Vector-valued target functions
-
- Can deal well with noisy, complex sensor data
 - Raw input from cameras or microphones
 - Restriction bias & preference bias
 - Not easy to understand by humans (especially the learned results)

- Threshold units
- Gradient descent
- Multilayer networks
- Backpropagation
- Hidden layer representations
- Example: Face Pose Classification
- Advanced topics



Consider humans:

- Neuron switching time $\sim .001$ second
- Number of neurons $\sim 10^{10}$
- Connections per neuron $\sim 10^{4-5}$
- Scene recognition time $\sim .1$ second
- 100 inference steps doesn't seem like enough
➡ Much parallel computation

Can you write such code?

Properties of artificial neural nets (ANNs)

- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically

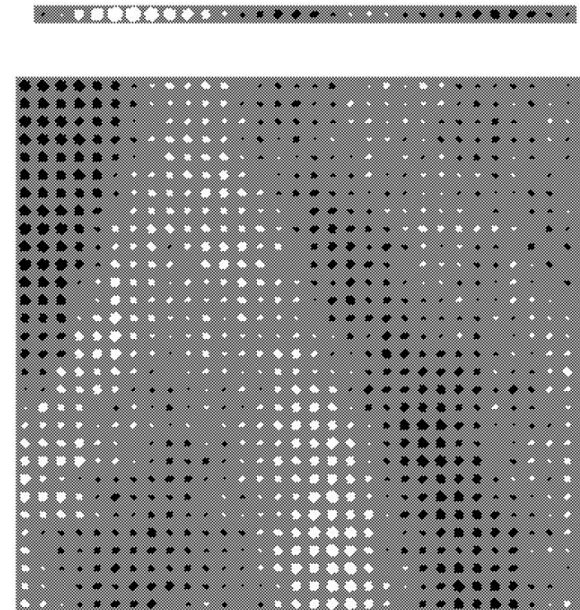
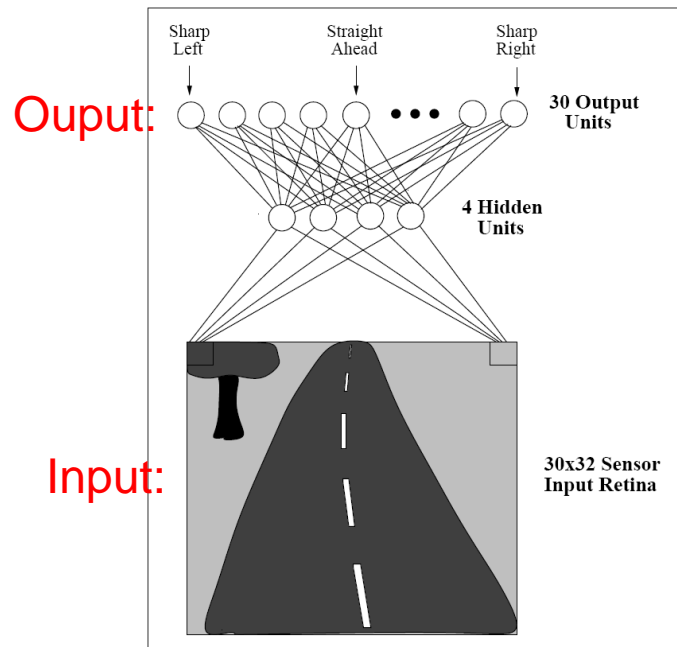
- Input is high-dimensional discrete or real-valued
- Output is discrete or real valued
- Output is one value or a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability of result is unimportant
- Long learning times are acceptable

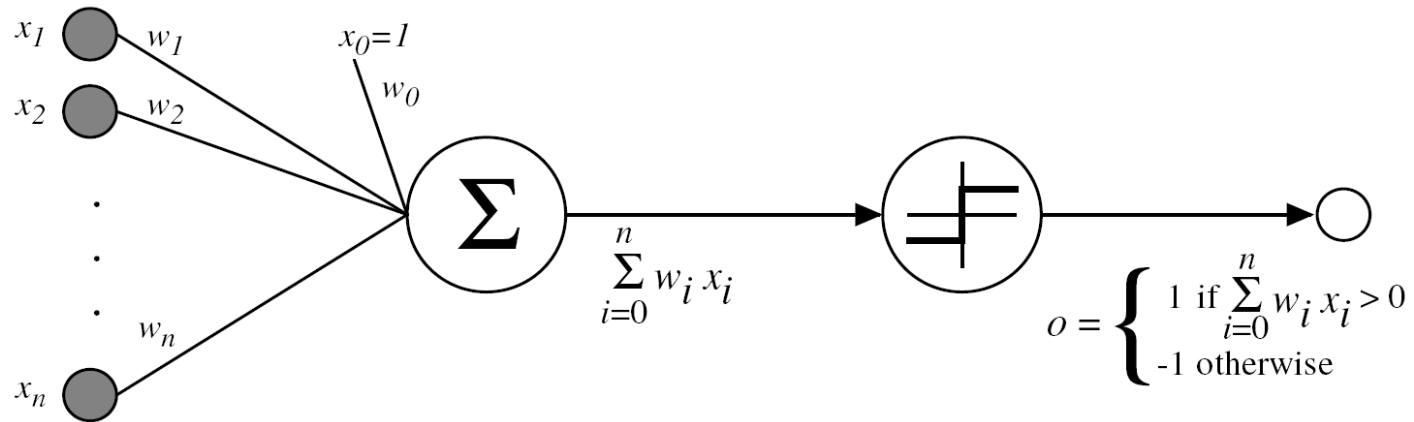
Examples:

- Speech recognition, speech translation
- Image classification, Object and Face Recognition
- Financial prediction

ALVINN drives 70 mph on highways

1989



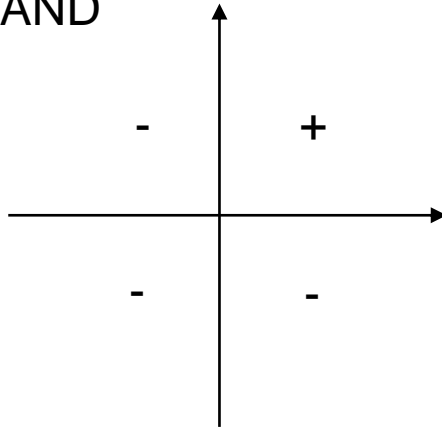


$$o(x_1, \dots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \dots + w_n x_n > 0 \\ -1 & \text{otherwise} \end{cases}$$

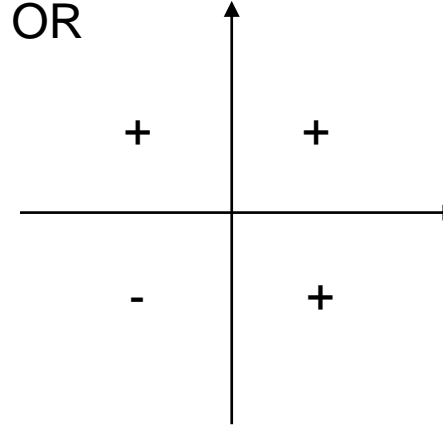
Sometimes we'll use simpler vector notation ($x_0 = 1$):

$$o(\mathbf{x}) = \text{sgn}(\mathbf{w}^T \mathbf{x}) = \begin{cases} +1 & \text{if } \mathbf{w} \cdot \mathbf{x} > 0 \\ -1 & \text{otherwise} \end{cases}$$

AND



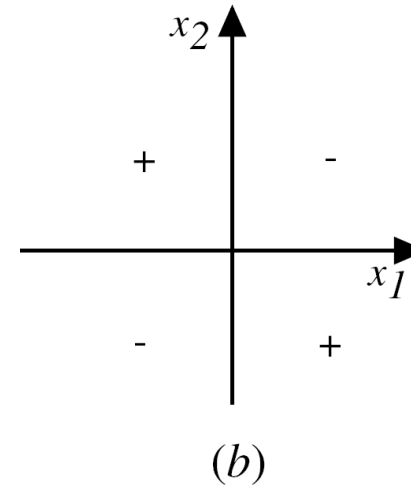
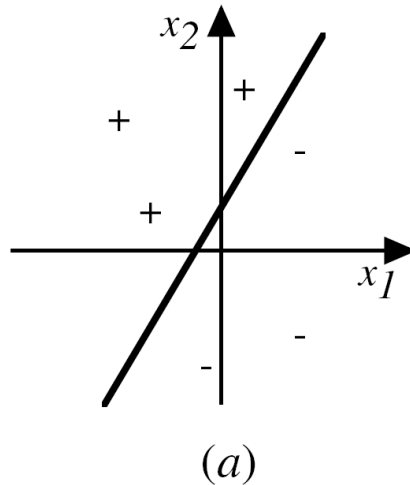
OR



Represents some useful functions

- What weights represent $g(x_1, x_2) = AND(x_1, x_2)$?
- What weights represent $g(x_1, x_2) = OR(x_1, x_2)$?
- NAND? NOR?
- m of n ?

Decision Surface of a Perceptron (2)



But some functions not representable

- e.g., not linearly separable (e.g. XOR)
- Therefore, we'll want networks of these...

Perceptron can represent all primitive boolean functions (AND, OR, NAND, NOR)

- Every boolean function can be represented by some network of perceptrons only two levels deep, in which the inputs are fed to multiple units, and the outputs of these units are then input to a second final stage
- E.g., disjunction (OR) of a set of conjunctions (ANDs)

Algo. 1: Perceptron training rule

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o)x_i$$

with

- $t = c(\vec{x})$ is target value
- o is perceptron output
- η is small constant (e.g., .1) called *learning rate*

Training algorithm:

- Start with random weights
- Iterate through training examples as many times as needed until all examples are classified correctly

Can prove it will converge

- If training data is linearly separable
- And η sufficiently small

BUT: If data is not linearly separable, convergence is not assured!

Algo. 2: Gradient Descent (1)

- Starts with random weights
- Converges also if data is not separable (towards a best fit)
- Basis for Backpropagation

To understand, consider simpler *linear unit*, where

$$o = w_0 + w_1x_1 + \dots + w_nx_n = \vec{w} \cdot \vec{x}$$

Let's learn w_i 's that *minimize the squared error*

$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

where D is the set of training examples and $E[\vec{w}]$ the training error

Algo. 2: Gradient Descent (2)

Direction that produces the steepest increase

Gradient

$$\nabla E[\mathbf{w}] = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \dots, \frac{\partial E}{\partial w_n} \right]$$

Training rule:

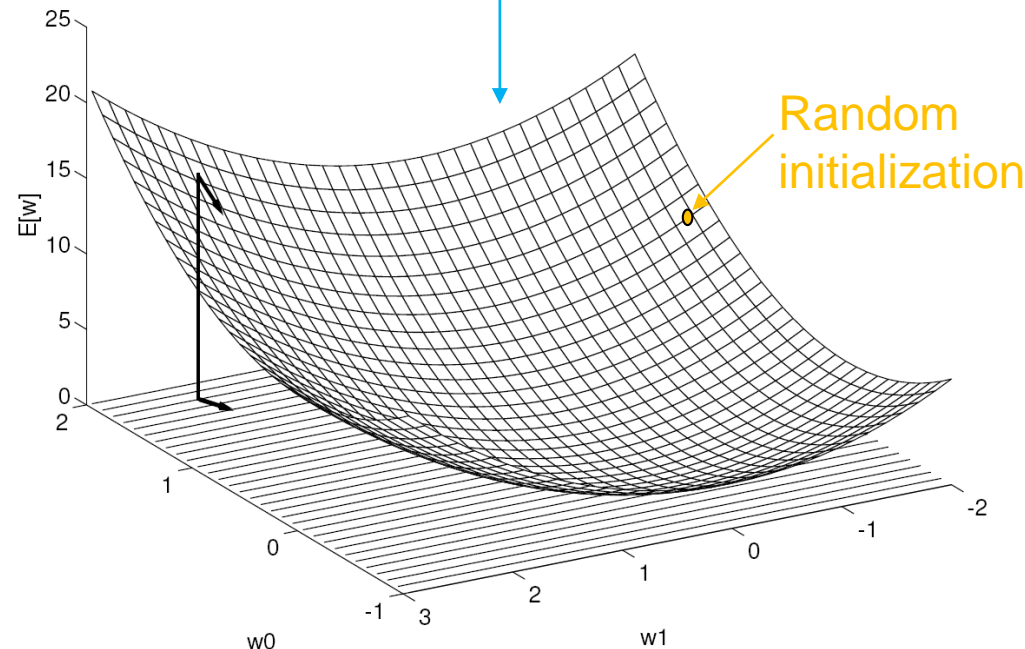
$$\Delta \mathbf{w} = -\eta \cdot \nabla E[\mathbf{w}]$$

i.e.,

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Derivatives of E with respect to w_i

Squared error of linear unit with two weights



Note: Squared error is parabolic with single global minimum

Algo. 2: Gradient Descent (3)

$$\begin{aligned}\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_d (t_d - o_d)^2 = \frac{1}{2} \frac{\partial}{\partial w_i} \sum_d (t_d - o_d)^2 \\&= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\&= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) = \sum_d (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d) \\&= \sum_d (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - w_0 - w_1 x_{1,d} - \dots - w_n x_{n,d}) \\&= \sum_d (t_d - o_d) \frac{\partial}{\partial w_i} (-w_i x_{i,d}) \\&= \sum_d (t_d - o_d) (-x_{i,d})\end{aligned}$$

Algo. 2: Gradient Descent (4)

Substituting in training rule

$$\Delta \mathbf{w} = -\eta \cdot \nabla E[\mathbf{w}]$$

results in

$$\begin{aligned}\Delta w_i &= -\eta \sum_d (t_d - o_d)(-x_{i,d}) \\ &= \eta \sum_d (t_d - o_d)x_{i,d}\end{aligned}$$

= weight update rule

Algo. 2: Gradient Descent (5)

GRADIENT-DESCENT (*training_examples*, η)

Each training example is a pair of the form $\langle \vec{x}, t \rangle$, where \vec{x} is the vector of input values, and t is the target output value. η is the learning rate (e.g., .05).

- Initialize each w_i to some small random value
- Until the termination condition is met, do
 - Initialize each Δw_i to zero.
 - For each $\langle \vec{x}, t \rangle$ in *training_examples*, do
 - Input the instance \vec{x} to the unit and compute the output o
 - For each linear unit weight w_i , do
$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$$
 - For each linear unit weight w_i , do
$$w_i \leftarrow w_i + \Delta w_i$$

- General strategy for searching through a large or infinitely large hypothesis space
- Can be applied if
 - Hypothesis space contains continuously parameterized hypothesis
 - Error can be differentiated with respect to hypothesis parameters
- Problems:
 - Convergence to local minimum can be slow
 - No guarantee to find global minimum (e.g., then multiple local minima exist)

Algo. 1: Perceptron training rule guaranteed to succeed after a finite # of iterations if

- Training examples are linearly separable
- Sufficiently small learning rate η

Algo. 2: Linear unit training rule uses gradient descent

- Guaranteed to converge to hypothesis with minimum squared error
- Given sufficiently small learning rate η
- Even when training data contains noise
- Even when training data not separable by H
but converges only asymptotically towards the minimum error hypothesis, possibly requiring unbounded time.

Incremental (Stochastic) Gradient Descent

Batch mode Gradient Descent:

Do until satisfied

1. Compute the gradient $\nabla E_D[\vec{w}]$
 2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$
-

Incremental mode Gradient Descent:

Do until satisfied

- For each training example d in D
 1. Compute the gradient $\nabla E_d[\vec{w}]$
 2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$
-

$$E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$E_d[\vec{w}] \equiv \frac{1}{2} (t_d - o_d)^2$$

Incremental Gradient Descent can approximate *Batch Gradient Descent* arbitrarily closely if η made small enough.

Meaning in the past

- Epoch
- Batch
- Incremental Mode

Meaning today

- Epoch
- Batch
- Mini-batch

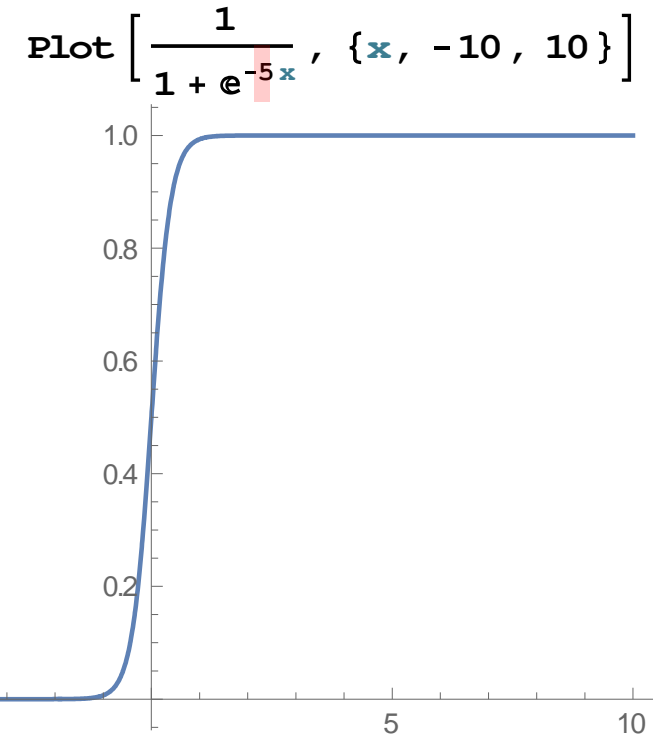
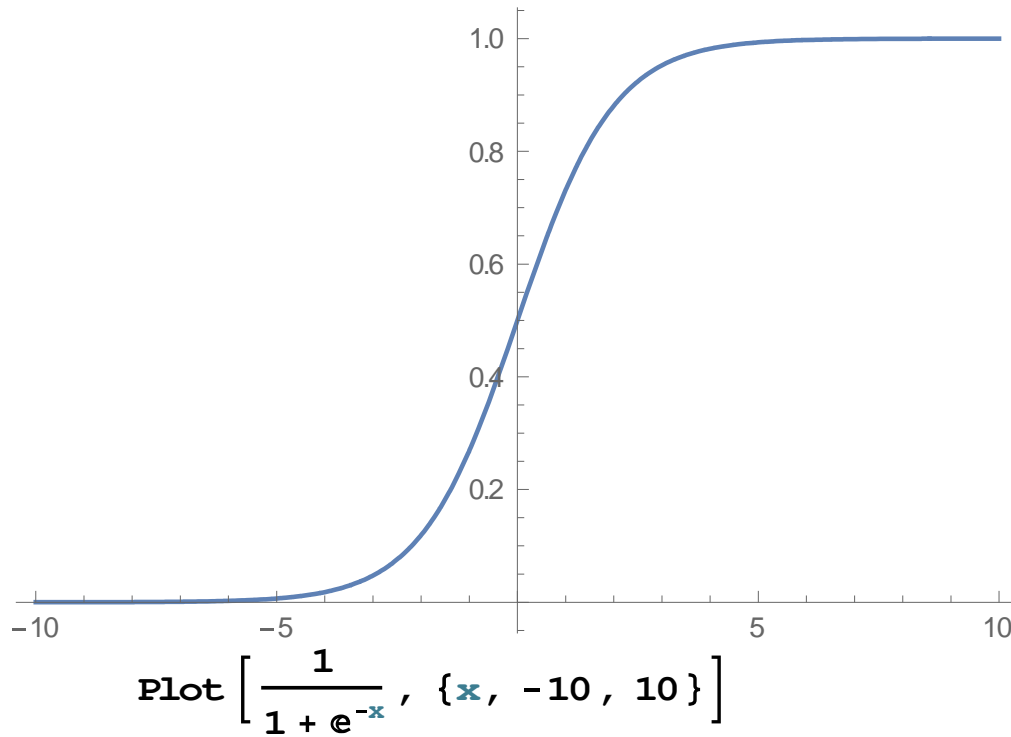
So far

- Gradient descent rule derived only for a linear unit. That was **NOT** our goal
- How do deal with the theshold function?

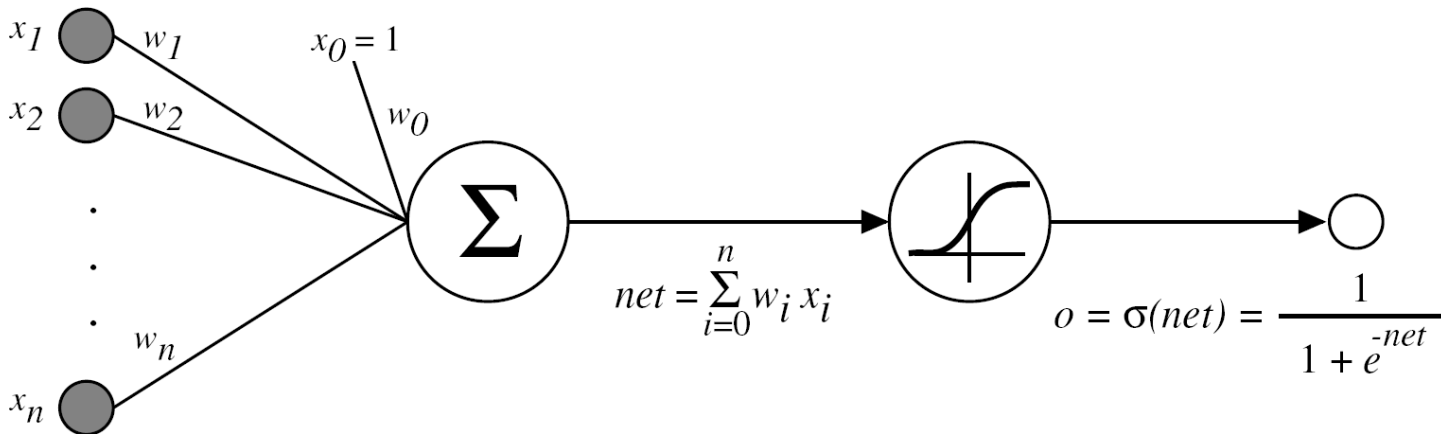
Idea:

- Approximate with something that is differentiable
 - use (logistic) sigmoid function

Logistic Sigmoid Function



Sigmoid Unit



- $\sigma(x) = \frac{1}{1+e^{-x}} = \text{(logistic) sigmoid function} = \text{a squashing func.}$
- Maps large input domain $]-\infty, +\infty[$ to small output range $[0,1]$
- Nice property: $\frac{d}{dx} \sigma(x) = \frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x))$
- We will derive gradient descent rules to train a single sigmoid unit
- Using one individual neuron with sigmoid activation is also called *logistic regression*

Error Gradient for a Sigmoid Unit

$$\begin{aligned}\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_d (t_d - o_d)^2 \\&= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\&= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\&= \sum_d (t_d - o_d) \left(-\frac{\partial o_d}{\partial w_i} \right) \\&= -\sum_d (t_d - o_d) \frac{\partial o_d(\text{net}_d(\vec{x}_d, \vec{w}))}{\partial w_i} \\&= -\sum_d (t_d - o_d) \frac{\partial o_d}{\partial \text{net}_d} \frac{\partial \text{net}_d}{\partial w_i}\end{aligned}$$

But we know:

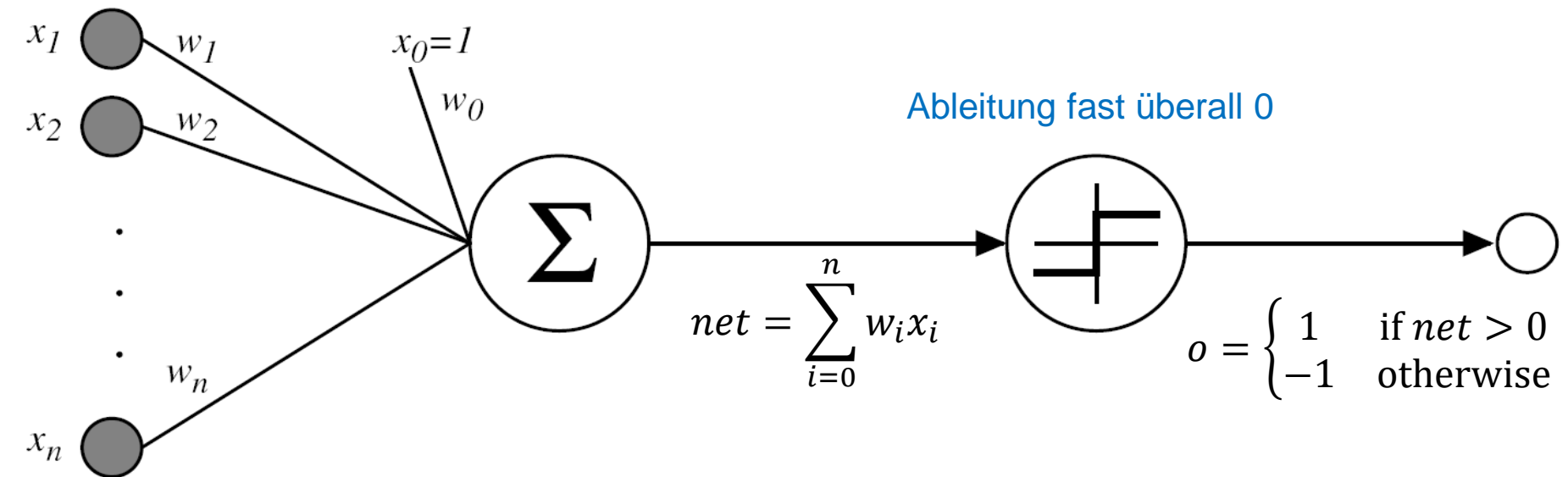
$$\begin{aligned}\frac{\partial o_d}{\partial \text{net}_d} &= \frac{\partial \sigma_d(\text{net}_d)}{\partial \text{net}_d} = o_d(1 - o_d) \\ \frac{\partial \text{net}_d}{\partial w_i} &= \frac{\partial (\vec{w} \cdot \vec{x}_d)}{\partial w_i} = x_{i,d}\end{aligned}$$

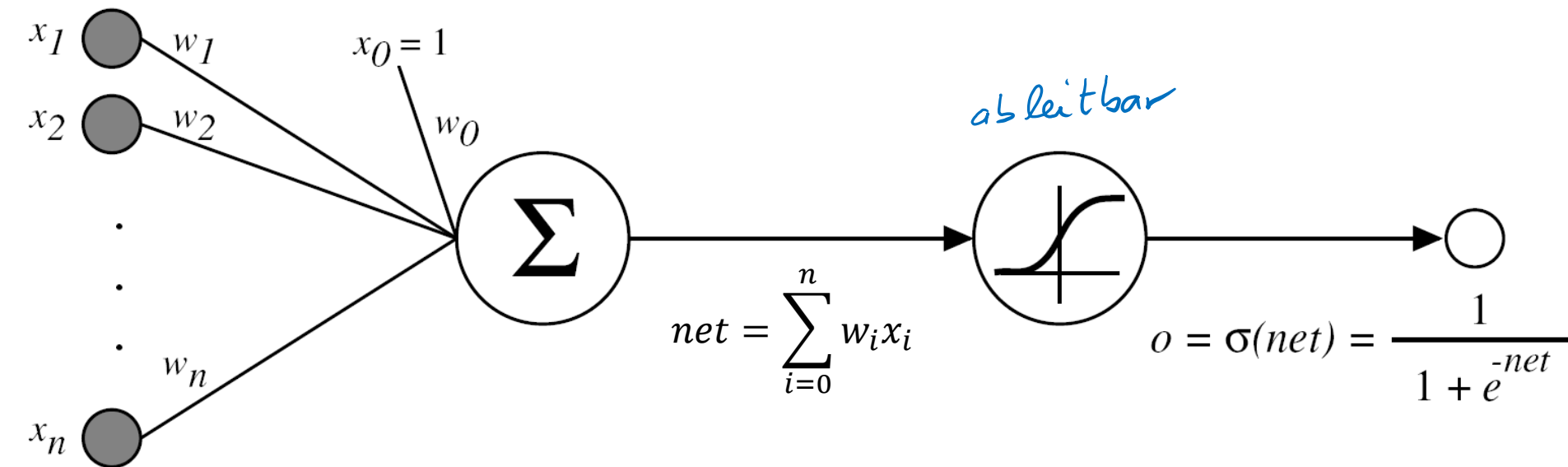
So:

$$\frac{\partial E}{\partial w_i} = -\sum_{d \in D} (t_d - o_d) o_d (1 - o_d) x_{i,d}$$

From a single unit to a multilayer network

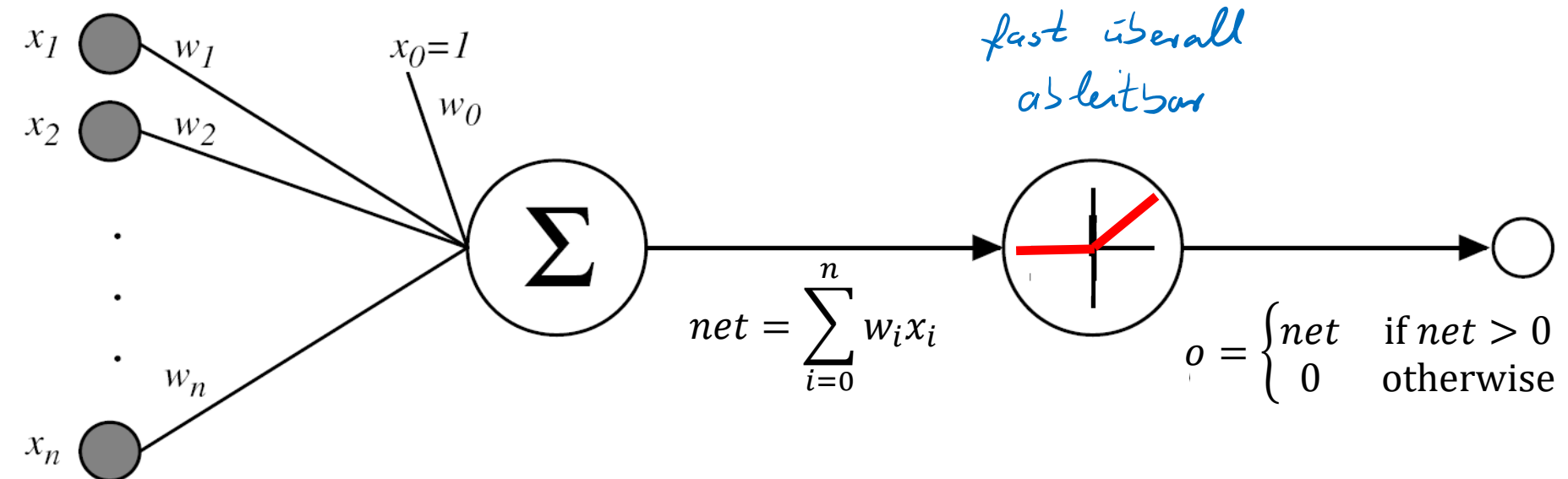
ReCap: Perceptron (Single Neuron)





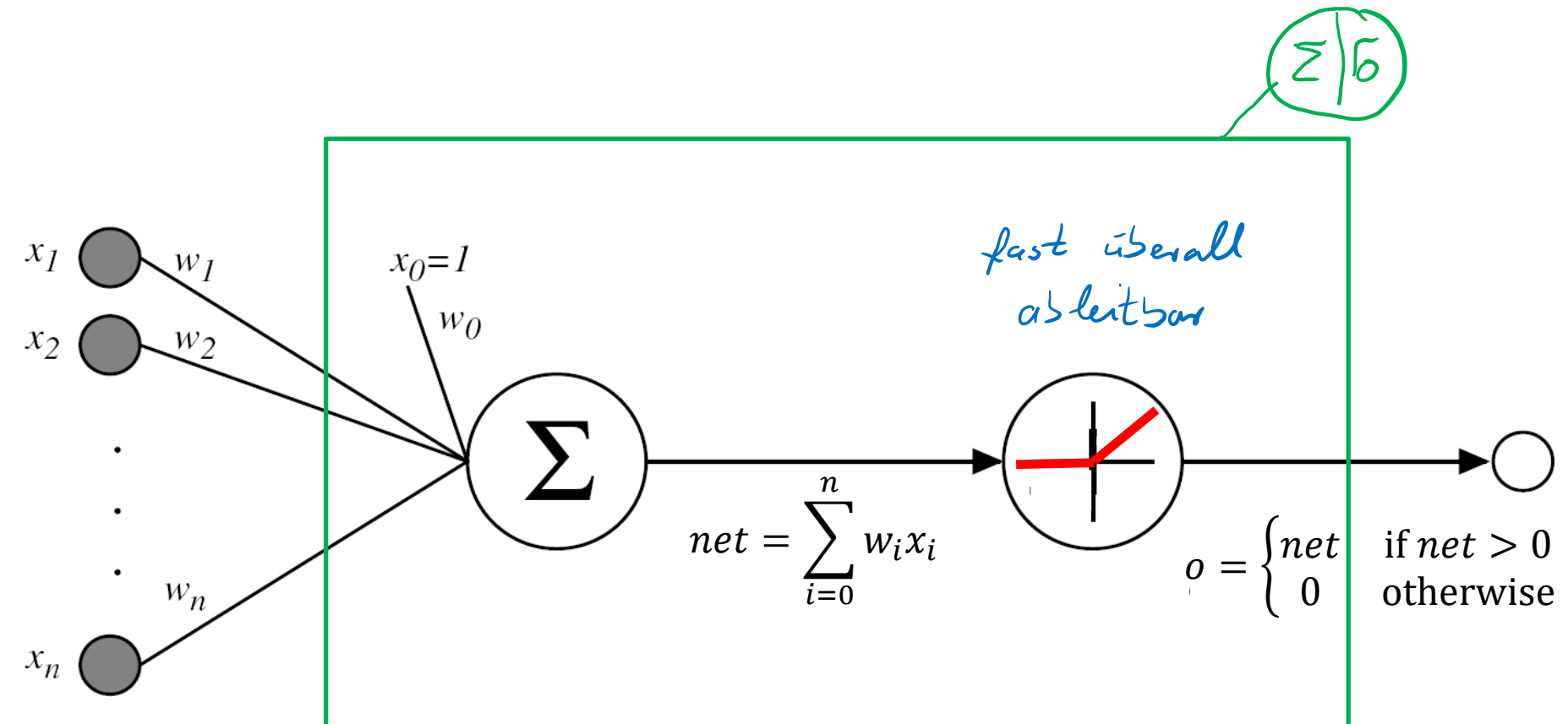
$$\frac{d}{d net} \sigma(net) = \sigma(net)(1 - \sigma(net))$$

Today: Rectify Linear Unit (ReLU) (Single Neuron)



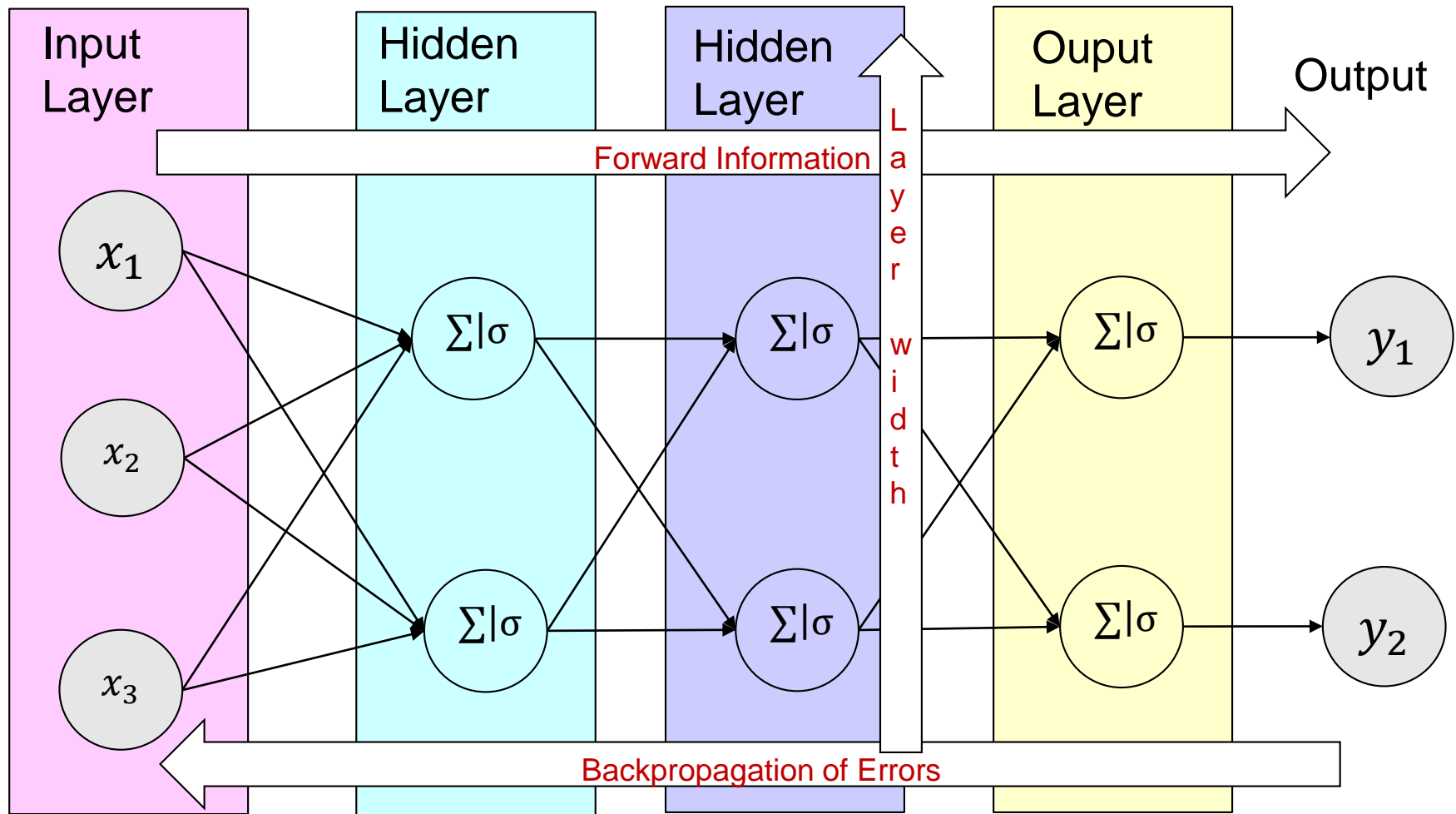
$$\frac{d}{d net} \sigma(net) = \begin{cases} 1 & \text{if } net > 0 \\ 0 & \text{otherwise} \end{cases}$$

Single Neuron with Non-linearity σ



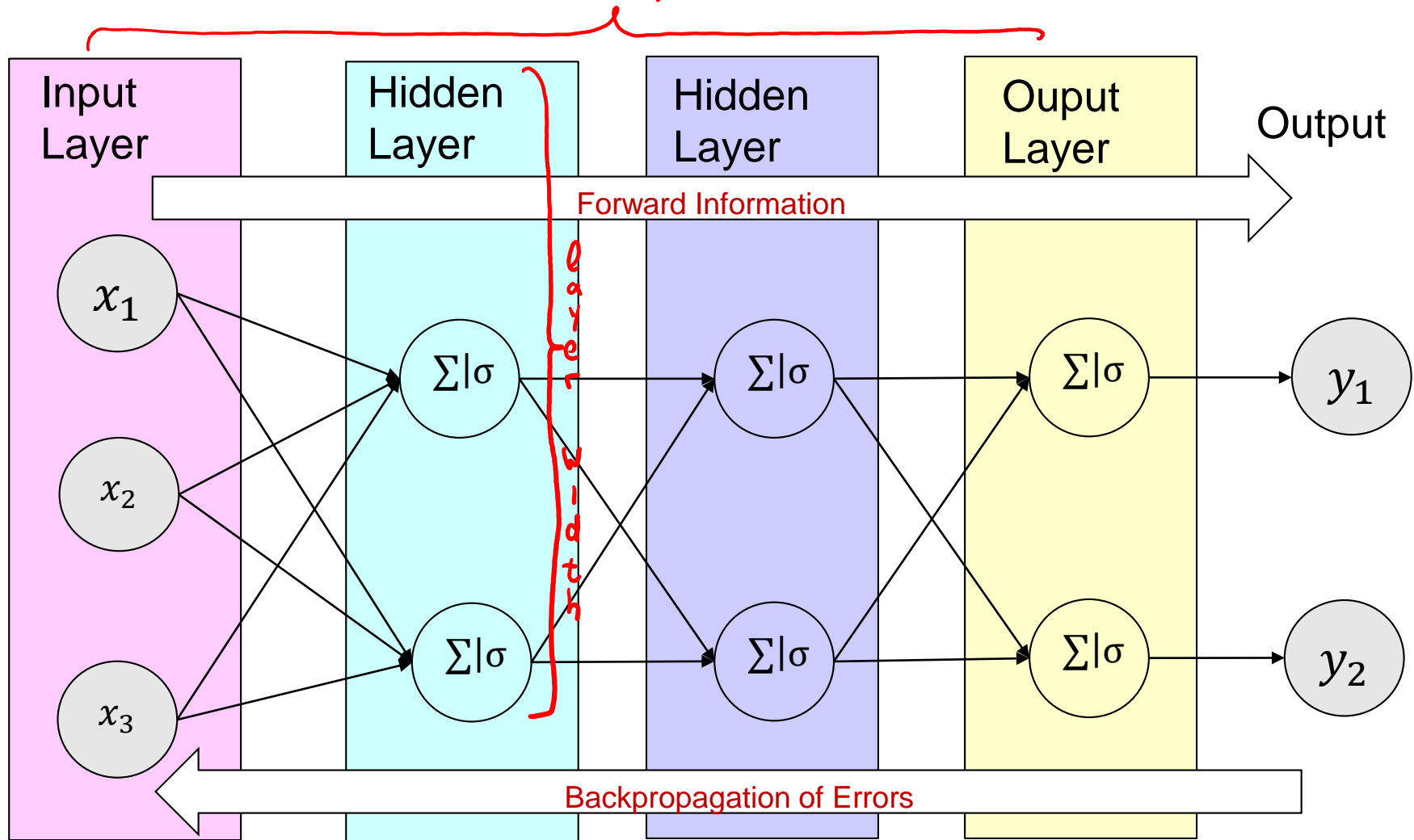
$$\frac{d}{d \text{ net}} \sigma(\text{net}) = \begin{cases} 1 & \text{if } net > 0 \\ 0 & \text{otherwise} \end{cases}$$

Von einem Neuron zu einem mehrschichtigen Netz an Neuronen



From Individual Neurons to Multilayer Neural Networks

network depth



- A neural network can be modeled as a *Directed Acyclic Graph* (*DAG*) describing how the individual functions (= individual neurons) are composed together.
- Two ways to look at a layer:
 - Representing a single vector-to-vector function
 - Many parallel units, each representing a vector-to-scalar function

Learn to compute suitable features $\phi(x)$ of input x such that linear regression or logistic regression can be used to approximate the desired function f^* , i.e.

$$y = f(x; \theta, w) = \phi(x; \theta)^T w$$

Backpropagation Algorithm (1)

- Goal: Minimize

$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{Outputs}} (t_{k,d} - o_{k,d})^2$$

↑
Set of output neurons

- Define:
 - $x_{i,j}$ = input from unit i to unit j (with associated weight $w_{i,j}$)
 - δ_n = error term associated with unit n
- Derivation later

Backpropagation Algorithm (2)

Initialize all weights to small random numbers.

Until satisfied, do

- For each training example, do
 1. Input the training example to the network and compute the network outputs
 2. For each output unit k : $\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$
 3. For each hidden unit: $\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{h,k} \delta_k$

Set of all neurons to which h is connected
 4. Update each network weight $w_{i,j}$:
$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

where $\Delta w_{i,j} = \eta \delta_j x_{i,j}$

Idee der Kettenregel (1D - Fall)

$$x_1 \xrightarrow{f} y = f(x_1)$$

$$x_1 \xrightarrow{f_1} x_2 = f_1(x_1) \xrightarrow{f_2} x_3 = f_2(x_2) \xrightarrow{\dots} \dots \xrightarrow{f_n} y = x_{n+1} = f_n(x_n)$$

$$\hat{=} y = f_n(x_n)$$

$$= f_n(f_{n-1}(x_{n-1}))$$

$$= \dots$$

$$= f_n(f_{n-1}(\dots f_1(x_1) \dots))$$

\Rightarrow

$$\begin{aligned} \frac{\partial}{\partial x_1} f &= \frac{\partial f}{\partial x_1} = \frac{\partial f_n(x_n)}{\partial x_n} \cdot \frac{\partial x_n(x_1)}{\partial x_1} \\ &= \frac{\partial f_n(x_n)}{\partial x_n} \cdot \frac{\partial x_n(x_{n-1})}{\partial x_{n-1}} \cdot \dots \cdot \frac{\partial x_2(x_1)}{\partial x_1} \\ &= \frac{\partial f_n(x_n)}{\partial x_n} \cdot \frac{\partial f_{n-1}(x_{n-1})}{\partial x_{n-1}} \cdot \dots \cdot \frac{\partial f_1(x_1)}{\partial x_1} \end{aligned}$$

Idee der Kettenregel (1D - Fall)

Beispiel:

$$\begin{array}{l} x_1 \xrightarrow{\quad} y = f(x_1) \\ x_1 \xrightarrow{f_1 := x_1^3} x_2 = f_1(x_1) \xrightarrow{f_2 = x_2 + 3} x_3 = f_2(x_2) \xrightarrow{f_3 = 2x_3^2} y = f_3(x_3) \\ \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad = f(x_1) \end{array}$$

\Rightarrow

$$\frac{\partial f}{\partial x_1} = \frac{\partial f_3}{\partial x_3} \cdot \frac{\partial f_2}{\partial x_2} \cdot \frac{\partial f_1}{\partial x_1}$$

Derivation by rule of chain beyond the scalar case

We can generalize this beyond the scalar case. Suppose that $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^n$, g maps from \mathbb{R}^m to \mathbb{R}^n , and f maps from \mathbb{R}^n to \mathbb{R} . If $\mathbf{y} = g(\mathbf{x})$ and $z = f(\mathbf{y})$, then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}. \quad (6.45)$$

In vector notation, this may be equivalently written as

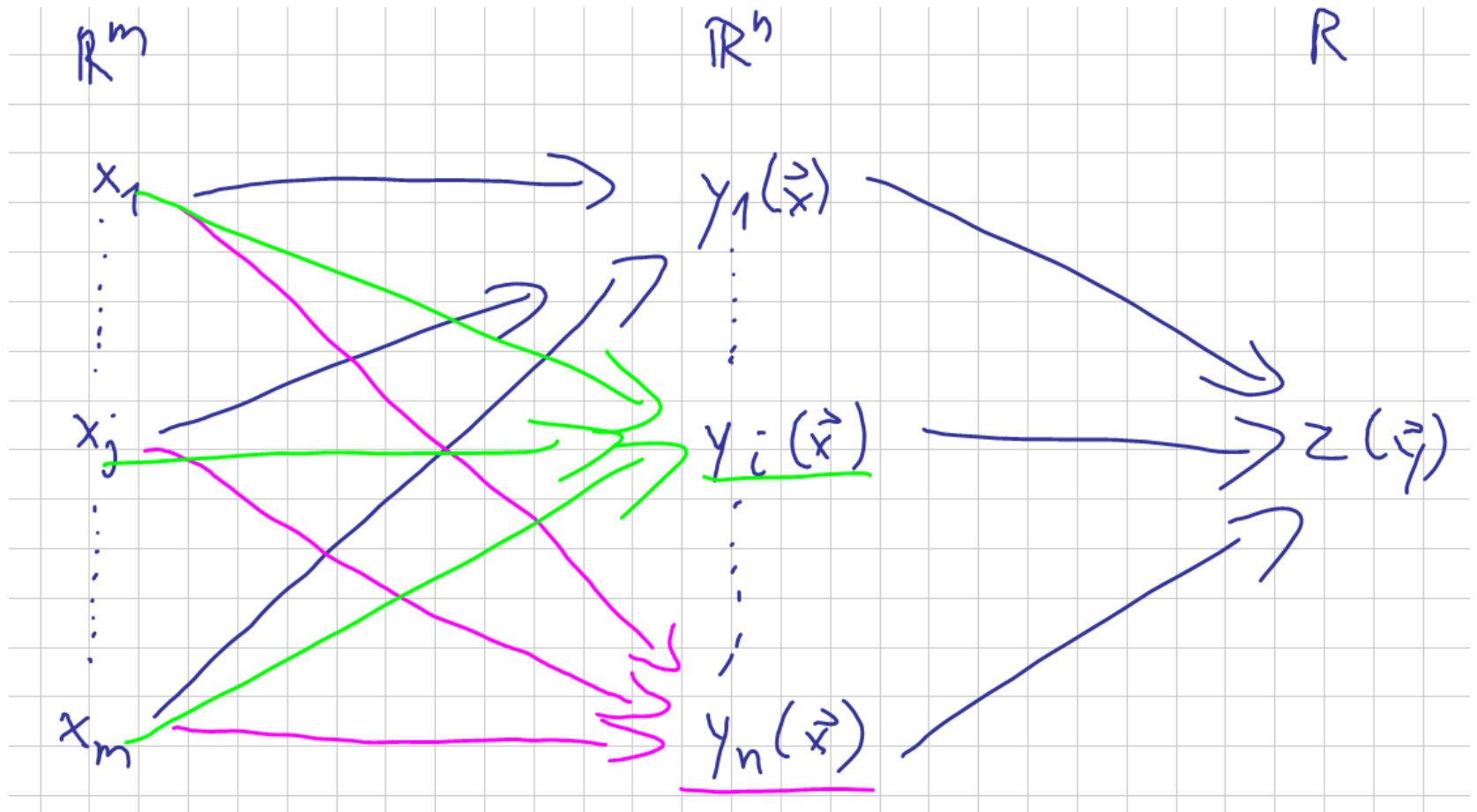
$$\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}} \right)^{\top} \nabla_{\mathbf{y}} z, \quad (6.46)$$

where $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ is the $n \times m$ Jacobian matrix of g .

Handwritten notes: A red bracket on the left groups the partial derivatives $\frac{\partial y_1}{\partial x_1}, \dots, \frac{\partial y_1}{\partial x_m}$ and $\frac{\partial y_n}{\partial x_1}, \dots, \frac{\partial y_n}{\partial x_m}$. A red circle highlights $\nabla_{\mathbf{y}} z$ in the equation, with a red arrow pointing to it from the handwritten $\frac{\partial z}{\partial \mathbf{y}}$ on the right.

Quelle: Ian Goodfellow and Yoshua Bengio and Aaron Courville. *Deep Learning*. MIT Press, <http://www.deeplearningbook.org>, 2016

Derivation by rule of chain beyond the scalar case



Derivation of Backpropagation

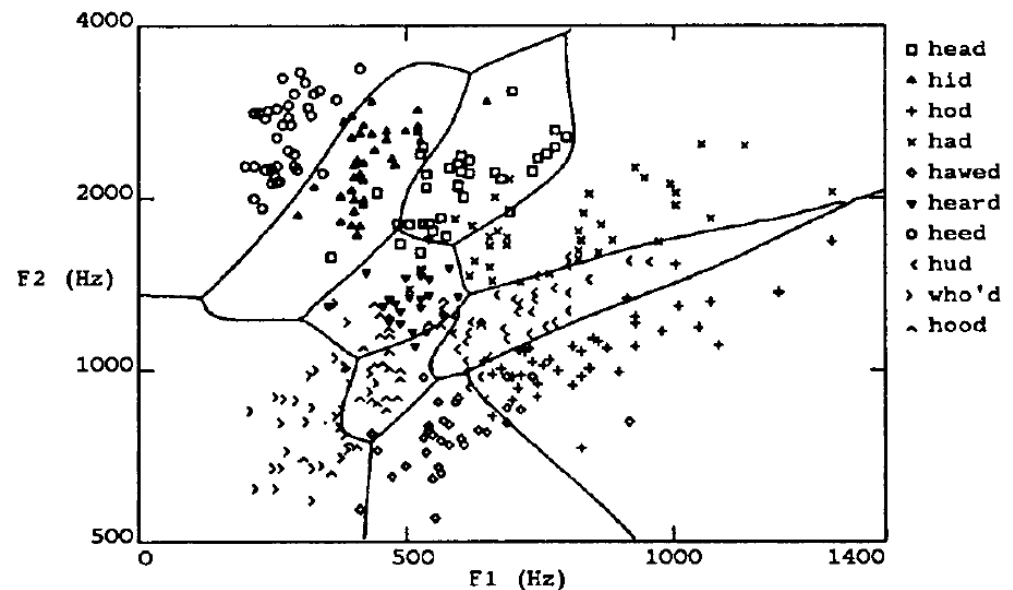
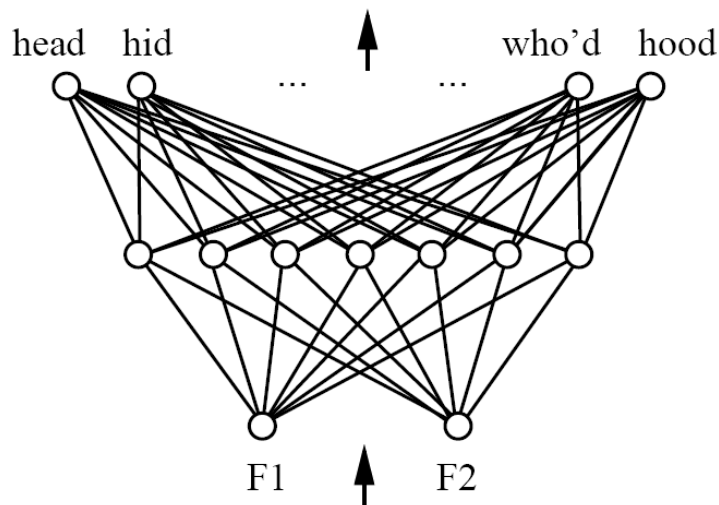
More on Backpropagation

- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
 - In practice, often works well (run multiple times)
- Often include weight *momentum* α

$$\Delta w_{i,j}(n) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(n-1)$$

n -th iteration Momentum term

- Minimizes error over *training* examples
 - Will it generalize well to subsequent examples?
- Training can take thousands of iterations → slow!
- Using network after training is faster



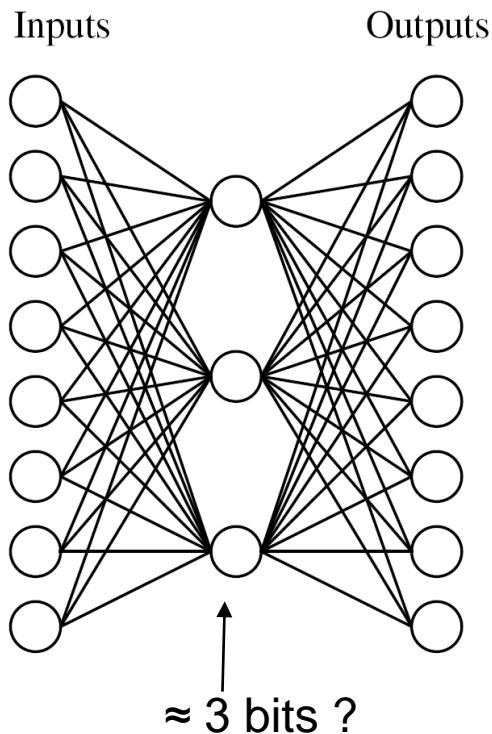
Goal: Given the two features F1 and F2, obtained from the spectral analysis of sound, recognize which 1 of 10 vowel sounds occurred in the context of “h_d”. The network prediction is the output whose values is highest.

- Multiple layers of cascaded linear units still produce only linear functions

➔ Needs

1. non-linear output
2. to be differentiable with respect to input

Example of a compression network:
Learn $f(x) = x$

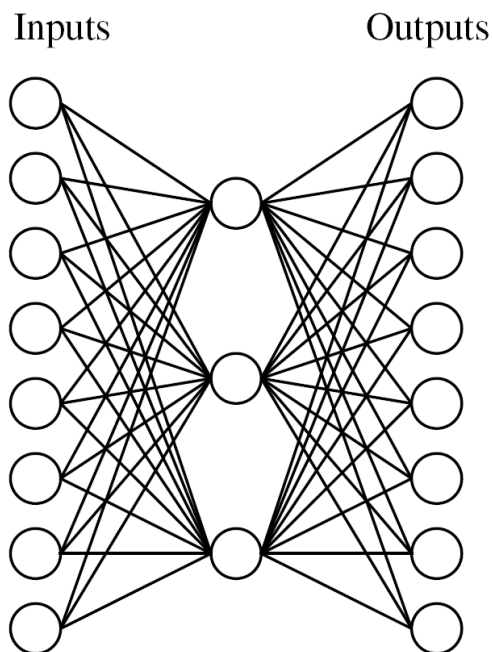


Target Function:

Input	Output
10000000	→ 10000000
01000000	→ 01000000
00100000	→ 00100000
00010000	→ 00010000
00001000	→ 00001000
00000100	→ 00000100
00000010	→ 00000010
00000001	→ 00000001

Can this be learned?

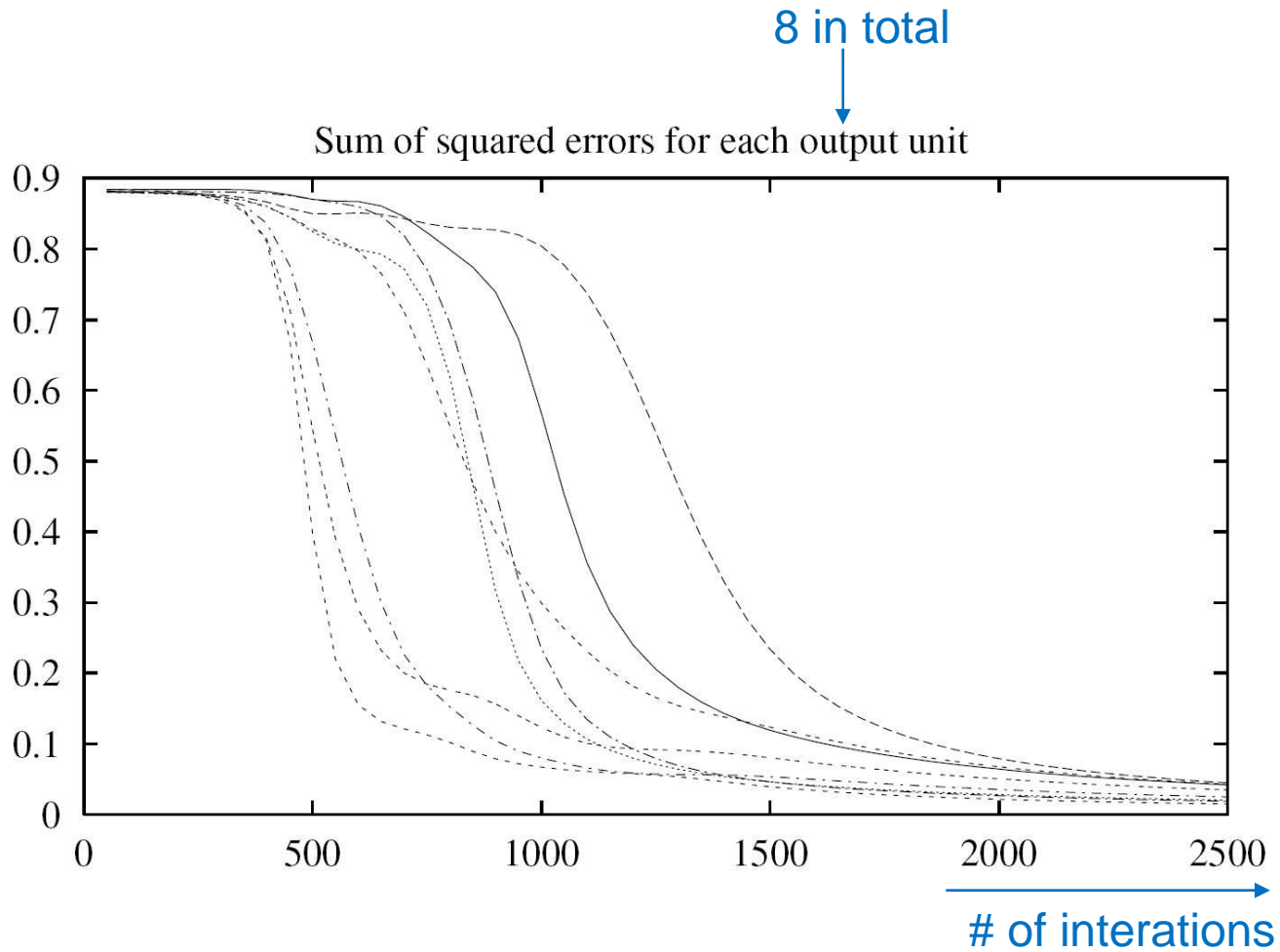
A network:

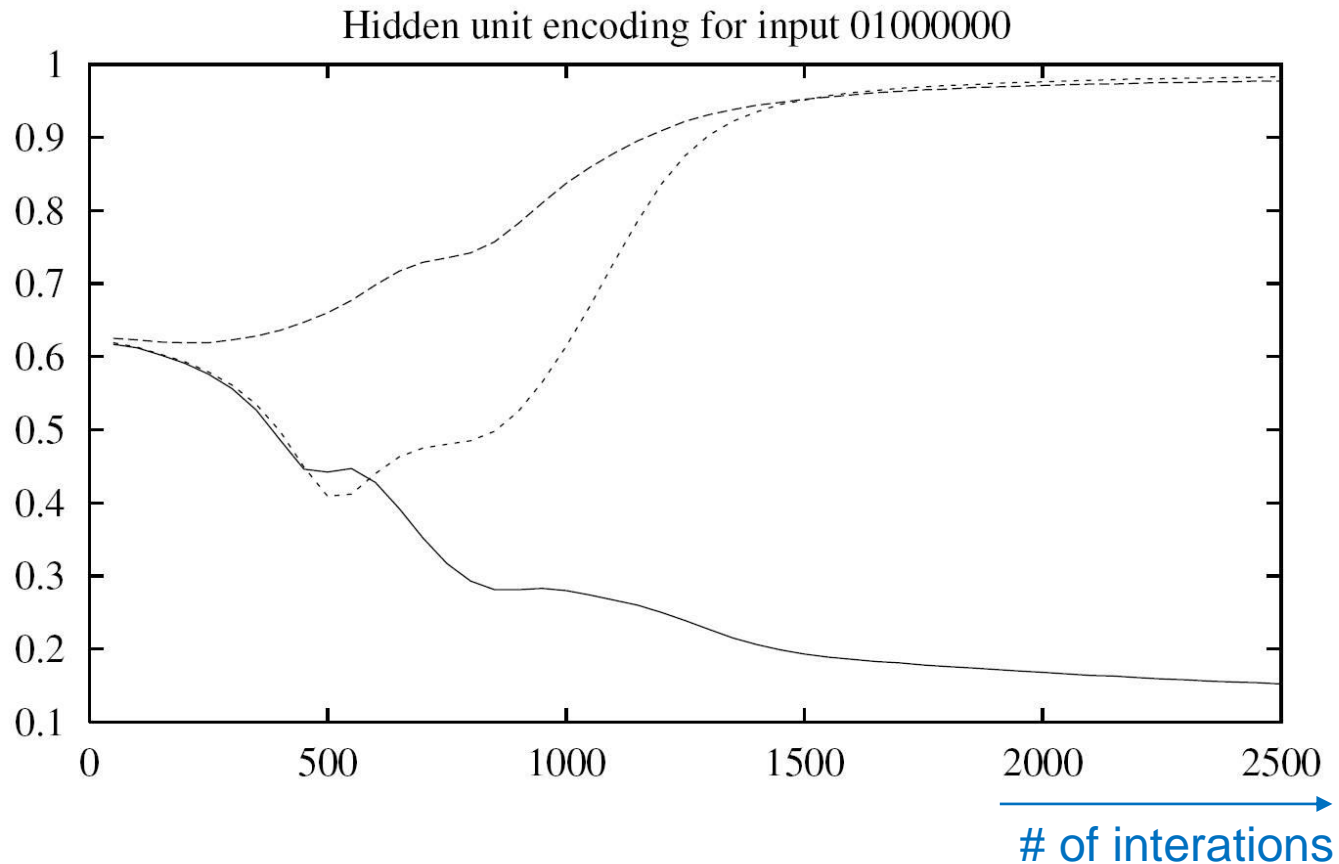


Learned hidden layer representation:

Input		Hidden Values		Output
10000000	→	.89 ₁ .04 ₀ .08 ₀	→	10000000
01000000	→	.01 ₀ .11 ₀ .88 ₁	→	01000000
00100000	→	.01 ₀ .97 ₁ .27 ₀	→	00100000
00010000	→	.99 ₁ .97 ₁ .71 ₁	→	00010000
00001000	→	.03 ₀ .05 ₀ .02 ₀	→	00001000
00000100	→	.22 ₀ .99 ₁ .99 ₁	→	00000100
00000010	→	.80 ₁ .01 ₀ .98 ₁	→	00000010
00000001	→	.60 ₁ .94 ₁ .01 ₀	→	00000001

Initialize weights randomly out of the range $[-.1, +.1]$, $\eta = 0.3$, and 5000 training steps (over all 8 examples).

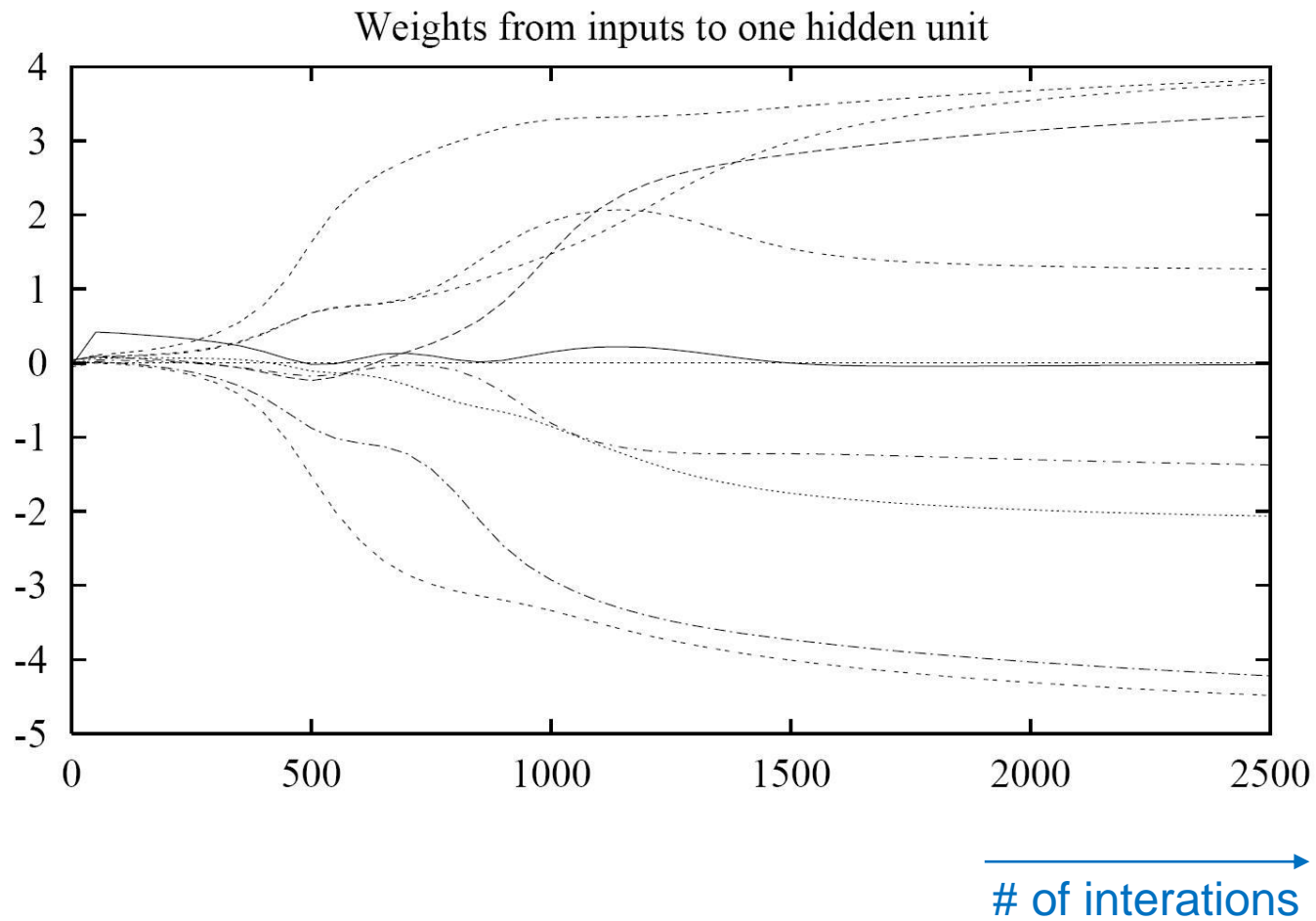




Network passes through a number of different encodings before converging.

Training (3)

8+1 inputs



- Backpropagation has the ability to
 - discover useful intermediate representations at the hidden unit layers
 - Invent features which are not explicitly given by human designer
 - More layers, more complex features

```
#include <ml.h>
#include <highgui.h>
#include <cxcore.h>

using namespace cv;

void main()
{
    // 'Load' training data
    Mat trainingData(8, 8, CV_32FC1);
    for (int n = 0; n < 8; n++) {
        for (int i = 0; i < 8; i++) {
            if (i == n) trainingData.at<float>(n, i) = 0.9f;
            else         trainingData.at<float>(n, i) = -0.9f;
        }
    }

    // create network with 8 input, 3 hidden, and 8 output neurons
    static Ptr< ml::ANN_MLP > nn= ml::ANN_MLP::create();
```

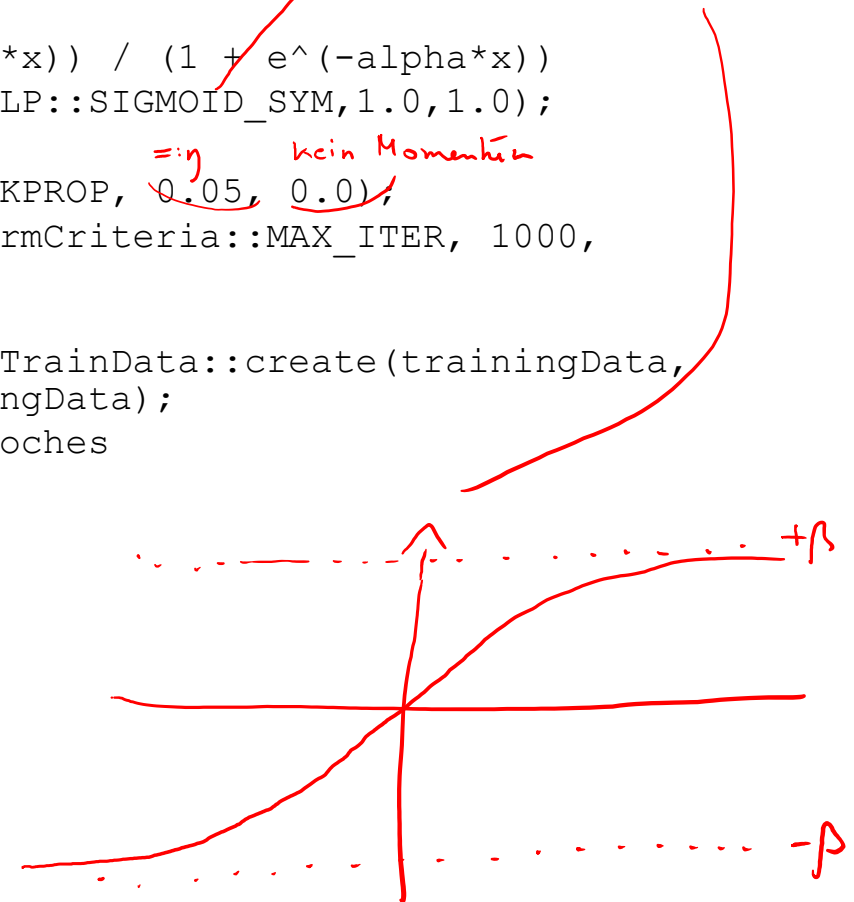
For OpenCV 3.1 (2)

```
int layerCountArray[] = { 8, 3, 8 };
Mat layerCount(3, 1, CV_32S, layerCountArray);
nn->setLayerSizes(layerCount);

// sigma(x) = beta * (1 - e^(-alpha*x)) / (1 + e^(-alpha*x))
nn->setActivationFunction(ml::ANN_MLP::SIGMOID_SYM, 1.0, 1.0);
// alpha=1.0, beta=1.0
nn->setTrainMethod(ml::ANN_MLP::BACKPROP, 0.05, 0.0);
nn->setTermCriteria(TermCriteria(TermCriteria::MAX_ITER, 1000,
DBL_EPSILON) );

Ptr<ml::TrainData> trainData = ml::TrainData::create(trainingData,
ml::SampleTypes::ROW_SAMPLE, trainingData);
// Learning the network for 1000 epoches
nn->train(trainData);
```

$$\sigma(\text{net}) = \beta \cdot \frac{1 - e^{-\alpha x}}{1 + e^{-\alpha x}}$$



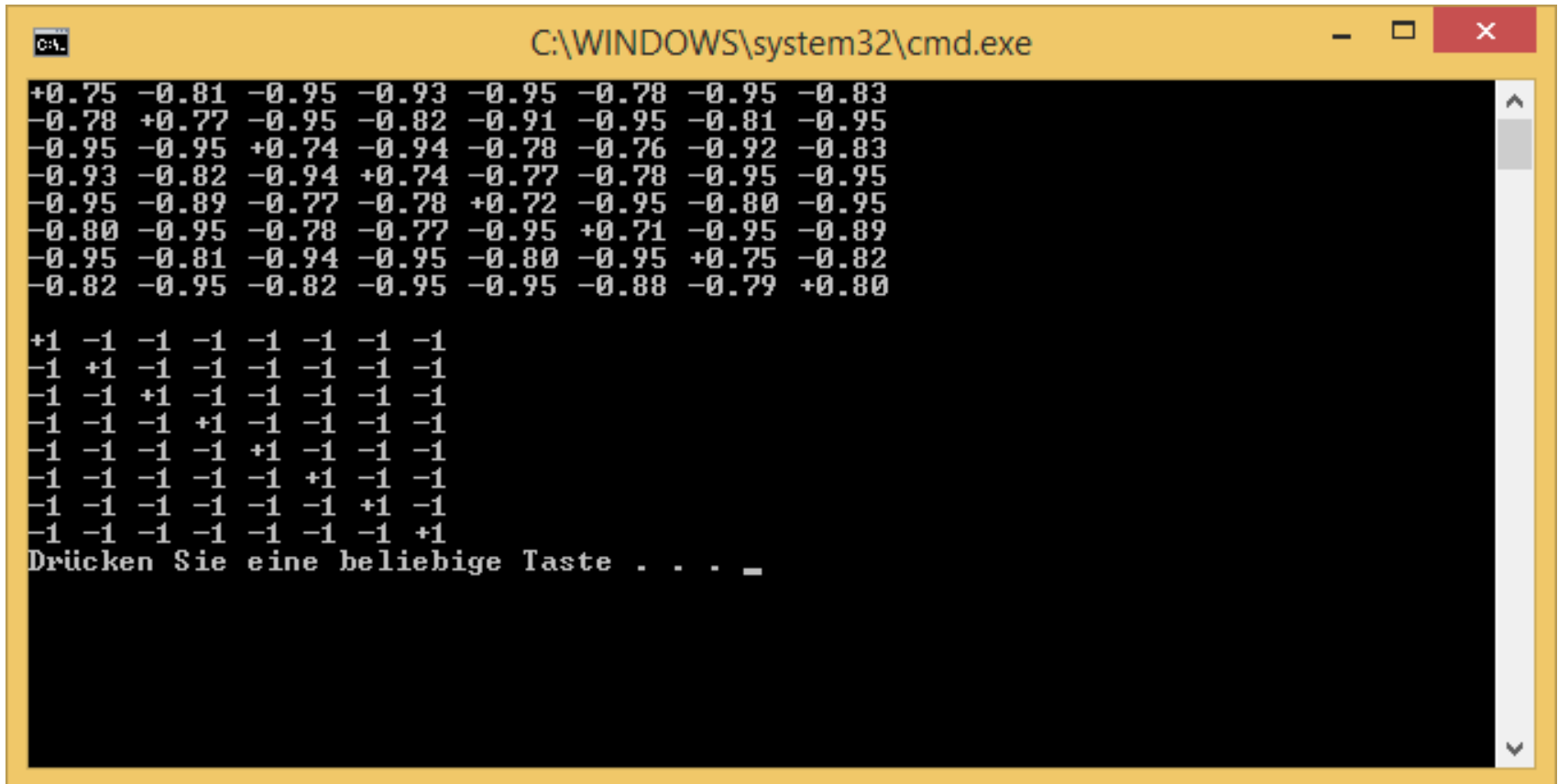
For OpenCV 3.1 (3)

```
// Show the classification results
Mat predictData(8, 8, CV_32FC1);
nn->predict(trainingData, predictData);

for (int n = 0; n < 8; n++) {
    for (int i = 0; i < 8; i++) { printf("%+04.2f ",
predictData.at<float>(n, i)); }
    printf("\n");
}

printf("\n");
for (int n = 0; n < 8; n++) {
    for (int i = 0; i < 8; i++) { printf("%+2d ",
(predictData.at<float>(n, i) >= 0) ? 1 : -1); }
    printf("\n");
}

return;
}
```



```
C:\WINDOWS\system32\cmd.exe

+0.75 -0.81 -0.95 -0.93 -0.95 -0.78 -0.95 -0.83
-0.78 +0.77 -0.95 -0.82 -0.91 -0.95 -0.81 -0.95
-0.95 -0.95 +0.74 -0.94 -0.78 -0.76 -0.92 -0.83
-0.93 -0.82 -0.94 +0.74 -0.77 -0.78 -0.95 -0.95
-0.95 -0.89 -0.77 -0.78 +0.72 -0.95 -0.80 -0.95
-0.80 -0.95 -0.78 -0.77 -0.95 +0.71 -0.95 -0.89
-0.95 -0.81 -0.94 -0.95 -0.80 -0.95 +0.75 -0.82
-0.82 -0.95 -0.82 -0.95 -0.95 -0.88 -0.79 +0.80

+1 -1 -1 -1 -1 -1 -1 -1
-1 +1 -1 -1 -1 -1 -1 -1
-1 -1 +1 -1 -1 -1 -1 -1
-1 -1 -1 +1 -1 -1 -1 -1
-1 -1 -1 -1 +1 -1 -1 -1
-1 -1 -1 -1 -1 +1 -1 -1
-1 -1 -1 -1 -1 -1 +1 -1
-1 -1 -1 -1 -1 -1 -1 +1
Drücken Sie eine beliebige Taste . . . _
```

Convergence of Backpropagation

Gradient descent to some local minimum

- Perhaps not global minimum...
- Add momentum
- Stochastic gradient descent
- Train multiple nets with different initial weights

Nature of convergence

- Initialize weights near zero
 - Therefore, initial networks near-linear
 - Increasingly non-linear functions possible as training progresses
↳ weight decay
- Expressive hypotheses space for Backpropagation
- However, not all possible weights values might be reachable through gradient descent from initial values.

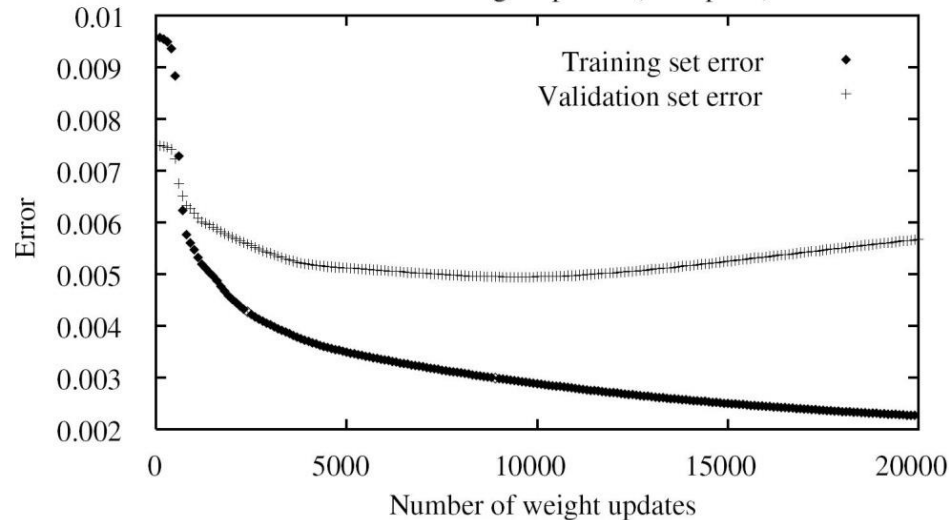
Boolean functions:

- Every boolean function can be represented by network with single hidden layer
- but might require exponential (in number of inputs) hidden units

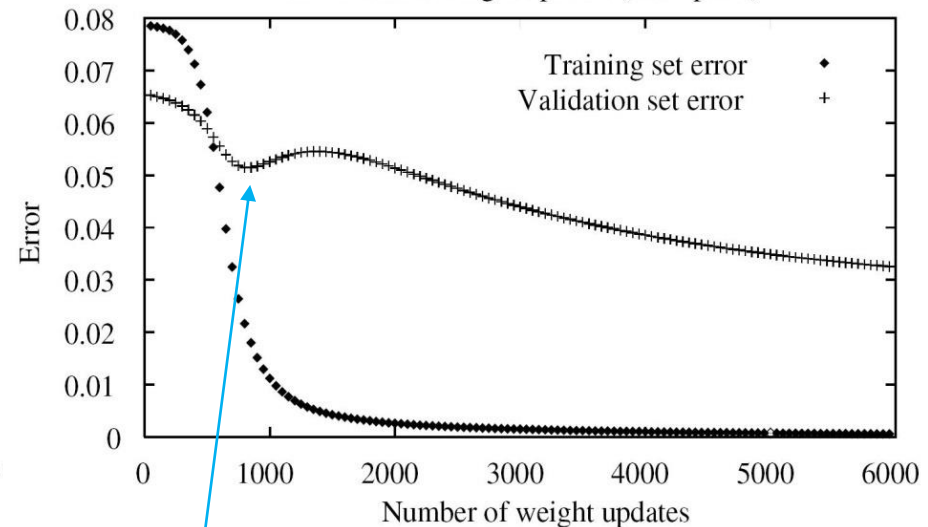
Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989; Hornik et al. 1989]
 - Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].
- Output layer = linear unit
- Hidden layer = sigmoid unit

Error versus weight updates (example 1)



Error versus weight updates (example 2)



Be careful here. Do
not stop too early.

Practical tip:

- The less hidden neurons, the lower the likelihood of overfitting.

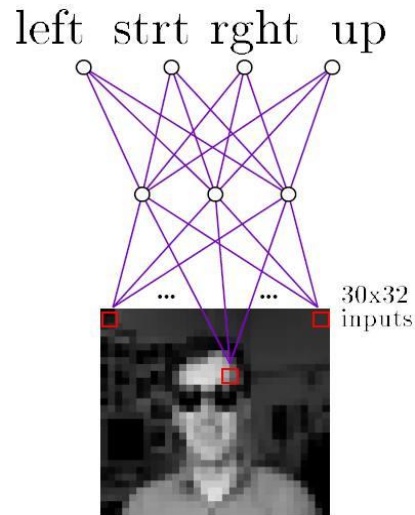
➔ Start with one hidden neuron and slowly increase.

➔ Always keep best weight set on validation set and continue training until stop criterion (max. # of iteration or magnitude of change of error below epsilon threshold)

NNs for Face Pose Classification

Tip:

Learn 0.1 vs 0.9 instead
of 0 vs 1 because sigmoid
function cannot produce
this value with finite
weights



960 x 3 x 4

Dataset:

624 grayscale images,
each 120 x 120

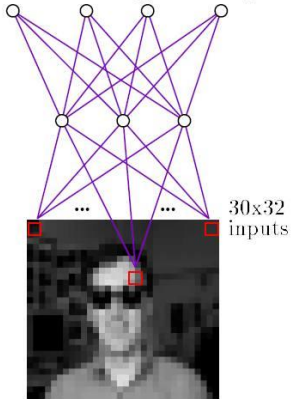


Typical input images

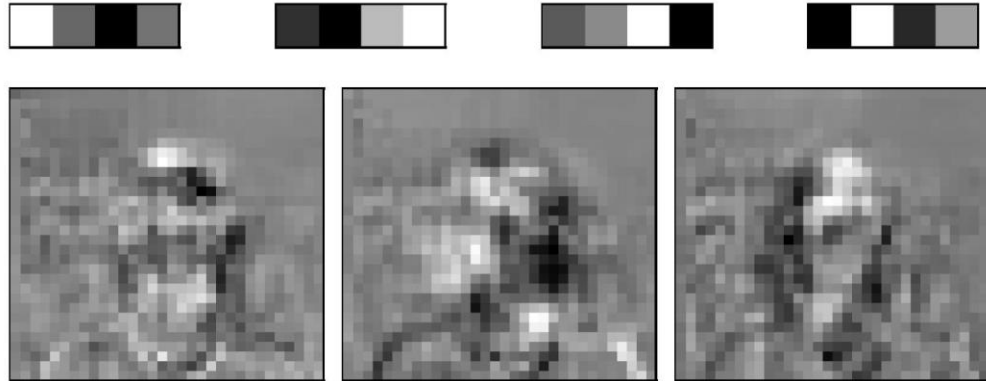
90% accurate in learning **head pose** (20 people, 32 images per person)

Learned Hidden Unit Weights

left strtr right up



Learned Weights

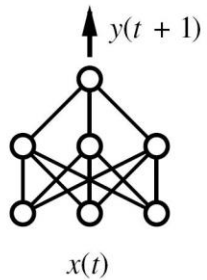


Typical input images

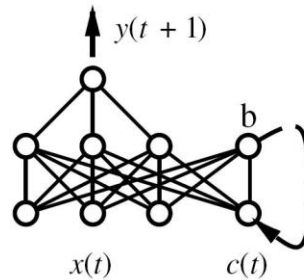
<http://www.cs.cmu.edu/~tom/faces.html>

Penalize large weights:

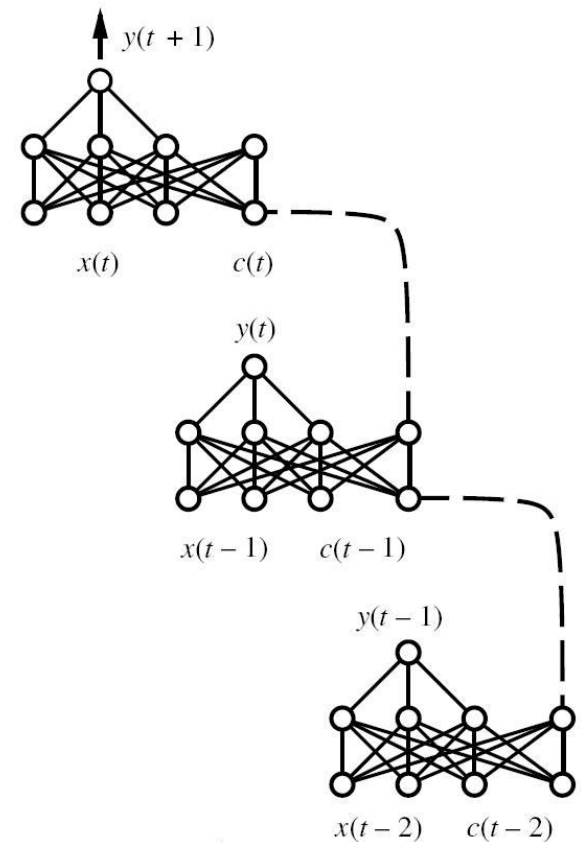
$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{outputs}} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$



(a) Feedforward network



(b) Recurrent network



(c) Recurrent network
unfolded in time

- Linear Units
 - For Gaussian output distributions
- Simoid Units
 - For Bernoulli output distributions
- Softmax Units
 - For Multinoulli output distributions