Machine Learning 10-315

Maria Florina Balcan
Machine Learning Department
Carnegie Mellon University

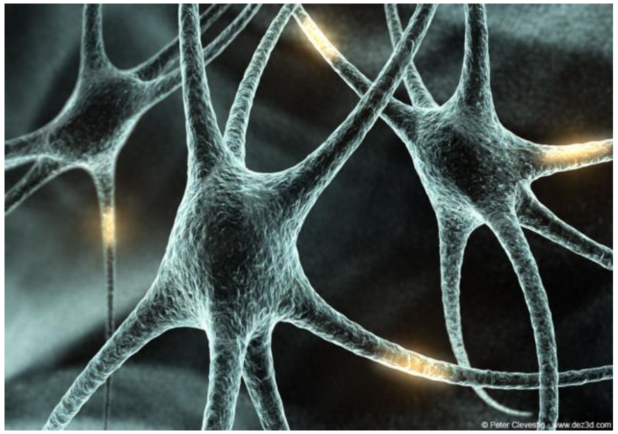
03/29/2019

Today:
- Artificial neural networks
- Backpropagation

Reading:
- Mitchell: Chapter 4
- Bishop: Chapter 5

Artificial Neural Network (ANN)

- Biological systems built of very complex webs of interconnected neurons.
- Highly connected to other neurons, and performs computations by combining signals from other neurons.
- Outputs of these computations may be transmitted to one or more other neurons.



- Artificial Neural Networks built out of a densely interconnected set of simple units (e.g., sigmoid units).
- Each unit takes real-valued inputs (possibly the outputs of other units) and produces a real-valued output (which may become input to many other units).

Connectionist Models

Consider humans:

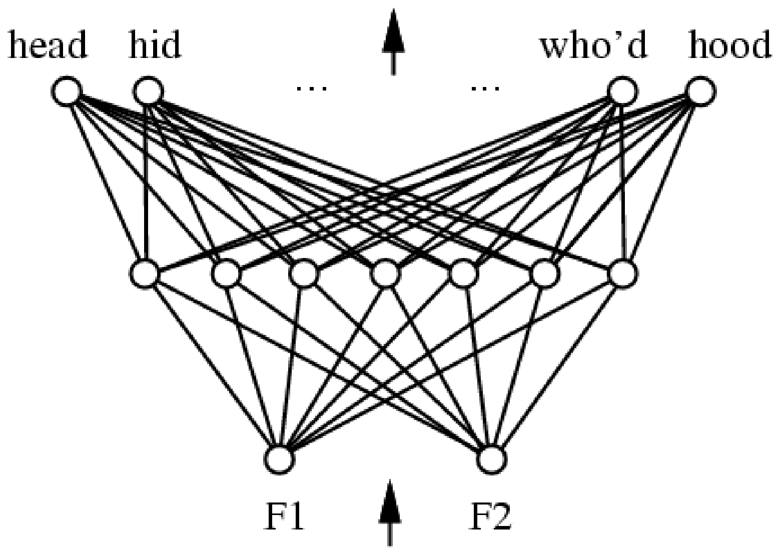
- Neuron switching time ˜ .001 second
- Number of neurons ˜ 10^{10}
- Connections per neuron ˜ 10^{4-5}
- Scene recognition time ˜ .1 second
- 100 inference steps doesn't seem like enough

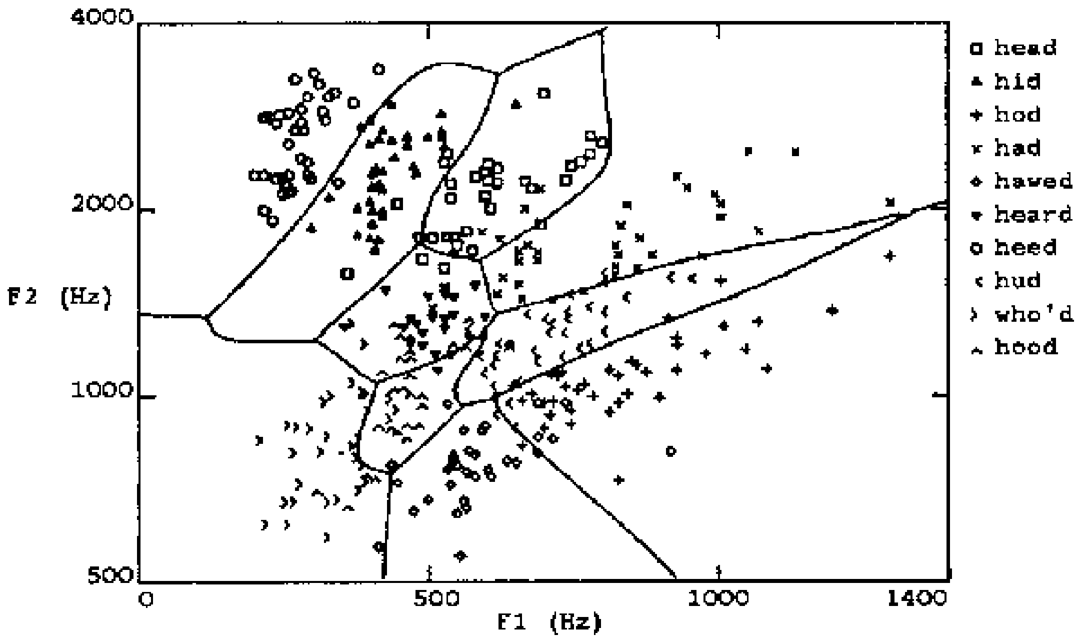
$\boxed{\rightarrow \text{much parallel computation}}$

Properties of artificial neural nets (ANN's):

- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process

Multilayer Networks of Sigmoid Units

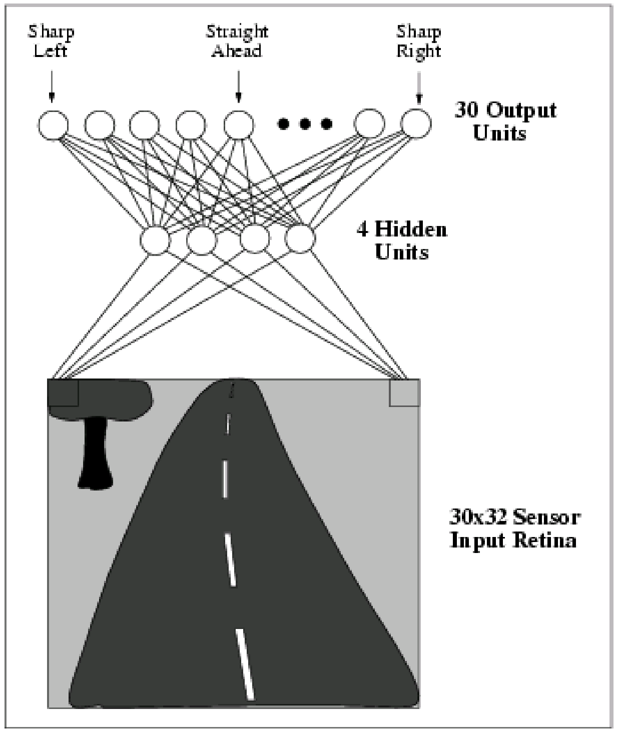


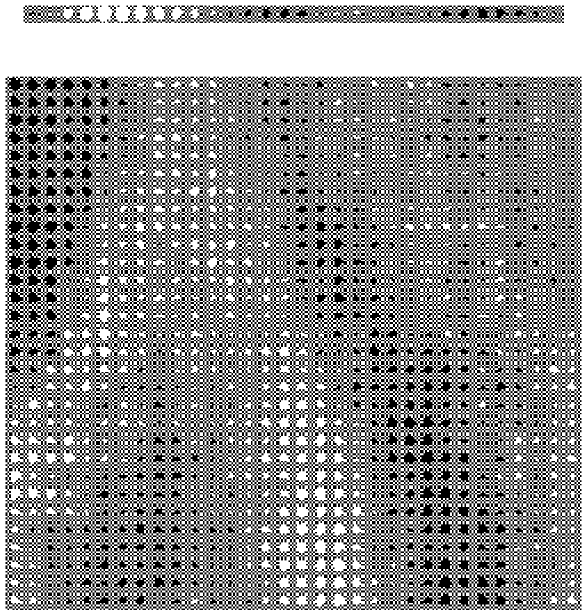


input: two features from spectral analysis of a spoken sound
output: vowel sound occurring in the context "h__d"

ALVINN

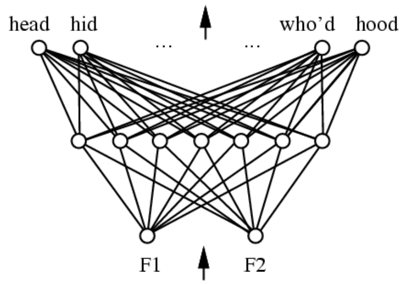
[Pomerleau 1993]





Artificial Neural Networks to learn f: X → Y

- $f_{\mathbf{w}}$ typically a non-linear function, $f_{\mathbf{w}}$: X $\rightarrow$ Y



- X feature space: (vector of) continuous and/or discrete vars
- Y ouput space: (vector of) continuous and/or discrete vars
- $f_{\mathbf{w}}$ *network* of basic units

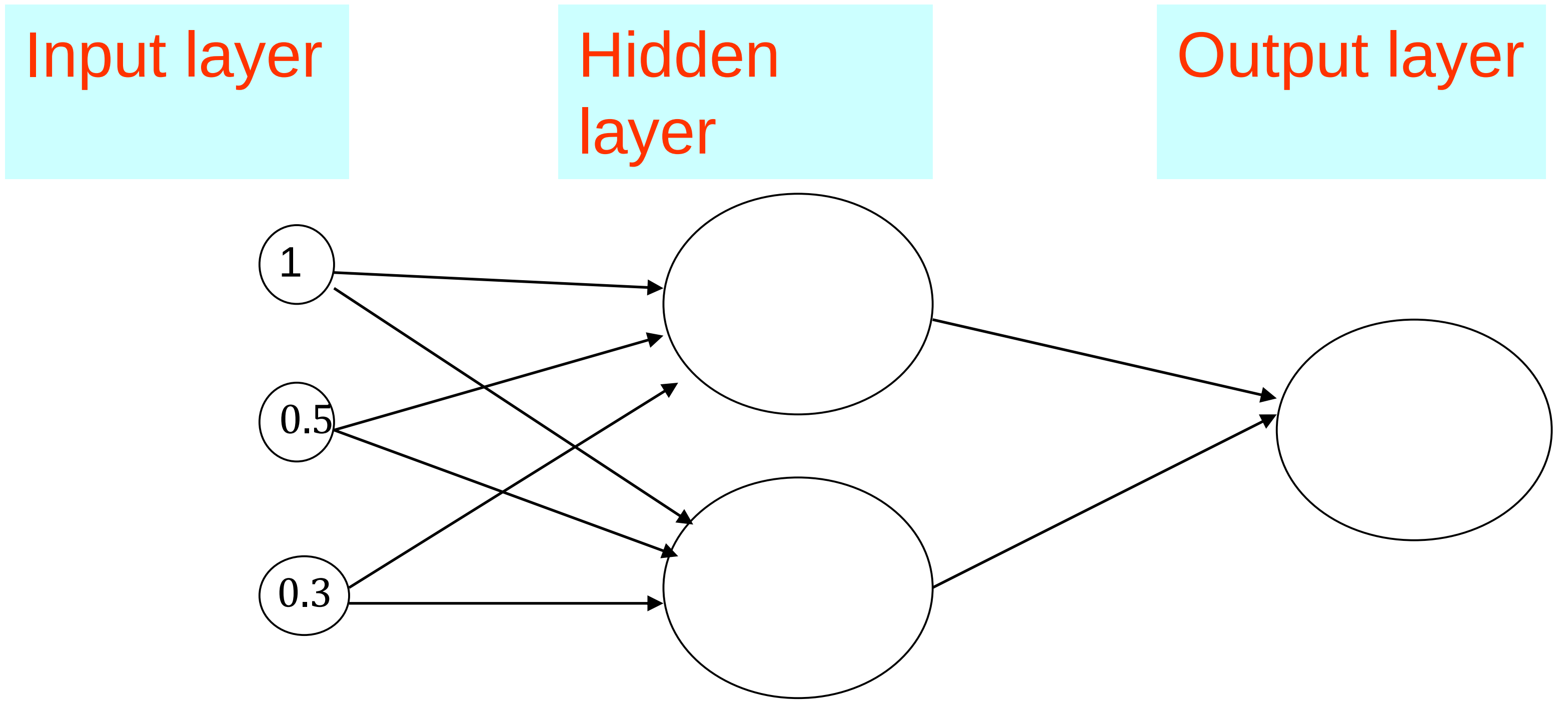
Learning algorithm: given $(x_d, t_d)_{d \in D}$, train weights w of all units to minimize sum of squared errors of predicted network outputs.

Find parameters w to minimize $\sum_{d \in D} (f_w(x_d) - t_d)^2$

Use gradient descent!

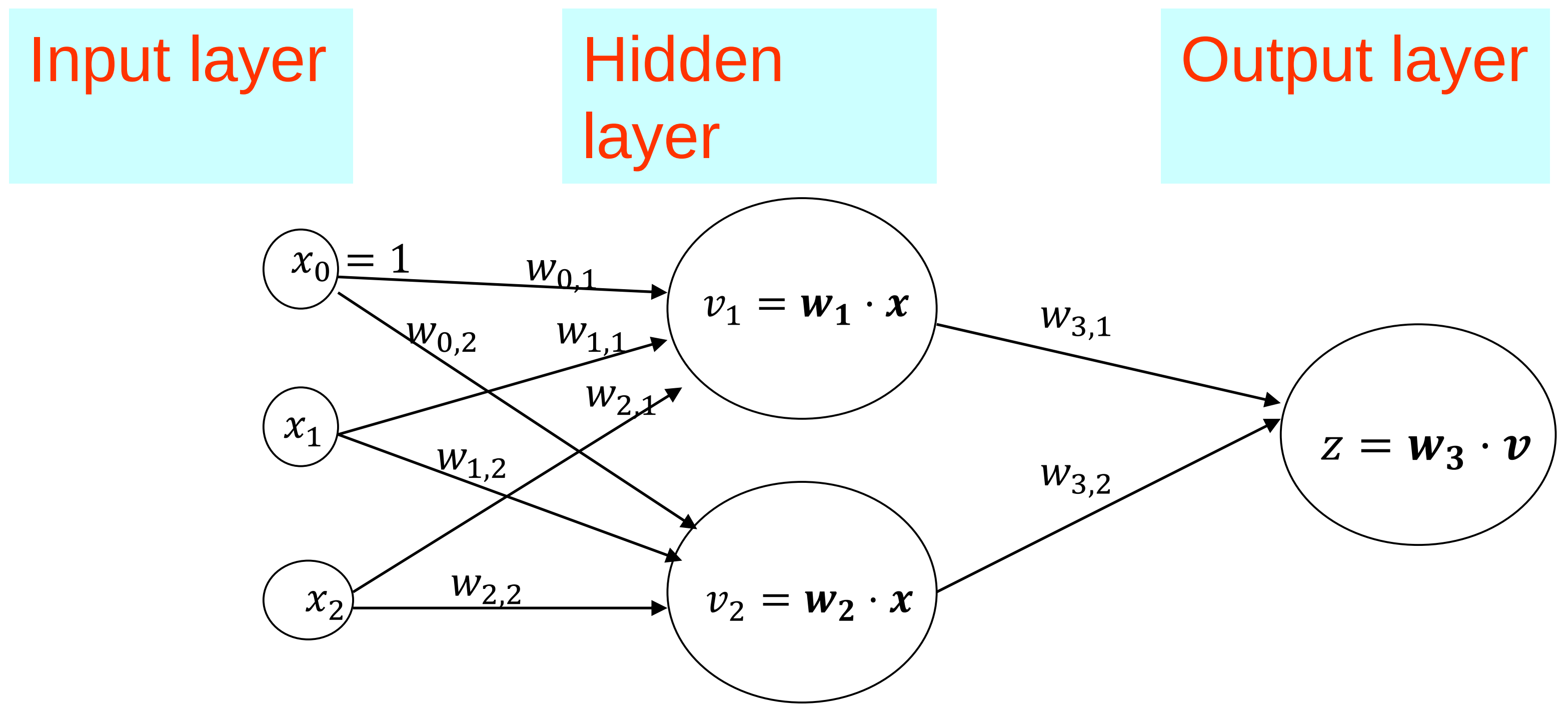
What type of units should we use?

- Classifier is a multilayer *network of units.*
- Each *unit* takes some inputs and produces one output. Output of one unit can be the input of another.



Multilayer network of Linear units?

- Advantage: we know how to do gradient descent on linear units

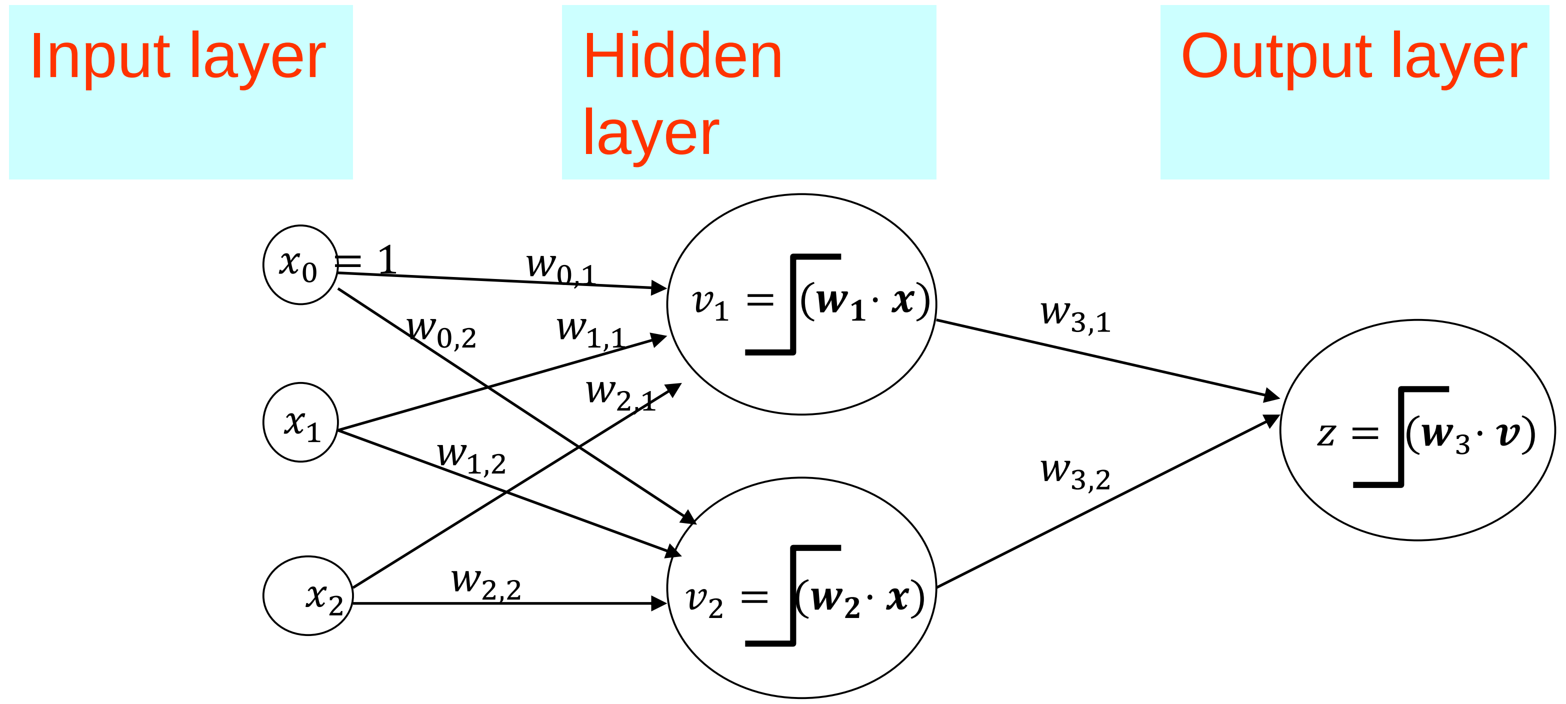


Problem: linear of linear is just linear.

$$z = w_{3,1}(\mathbf{w_1} \cdot \mathbf{x}) + w_{3,2}(\mathbf{w_2} \cdot \mathbf{x}) = \left(w_{3,1}\mathbf{w_1} + w_{3,2}\mathbf{w_2}\right) \cdot \mathbf{x} = \text{linear}$$

Multilayer network of Perceptron units?

- Advantage: Can produce highly non-linear decision boundaries!

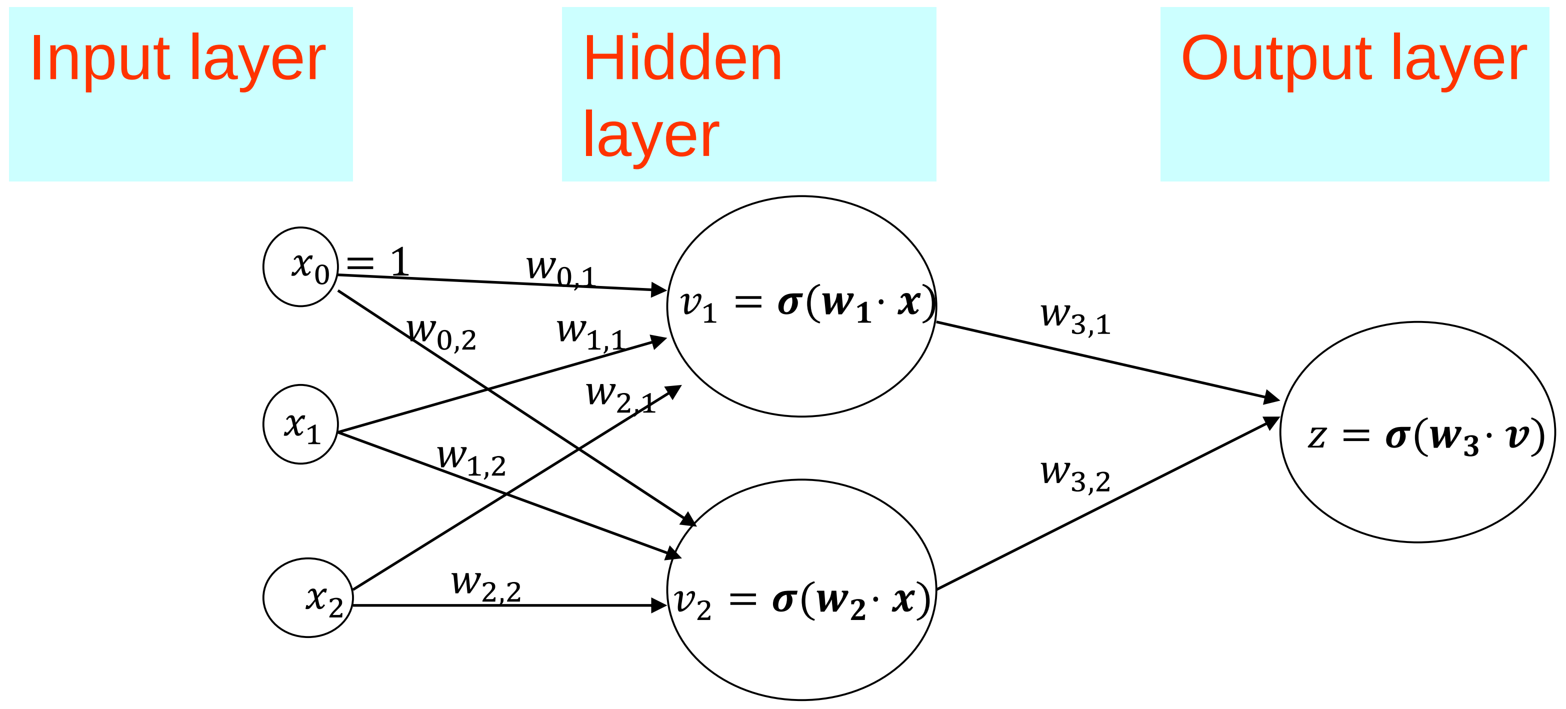

Input layer
Hidden layer
Output layer
$x_0 = 1$
x_1
x_2
$w_{0,1}$
$w_{0,2}$
$w_{1,1}$
$w_{2,1}$
$w_{1,2}$
$w_{2,2}$
$v_1 = \lrcorner(\boldsymbol{w_1} \cdot \boldsymbol{x})$
$v_2 = \lrcorner(\boldsymbol{w_2} \cdot \boldsymbol{x})$
$w_{3,1}$
$w_{3,2}$
$z = \lrcorner(\boldsymbol{w}_3 \cdot \boldsymbol{v})$

Threshold function: $\lrcorner x = 1$ if x is positive, 0 if x is negative.

Problem: discontinuous threshold is not differentiable. Can't do gradient descent.

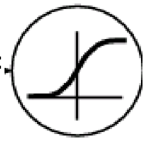
Multilayer network of sigmoid units

- Advantage: Can produce highly non-linear decision boundaries!
- Sigmoid is differentiable, so can use gradient descent



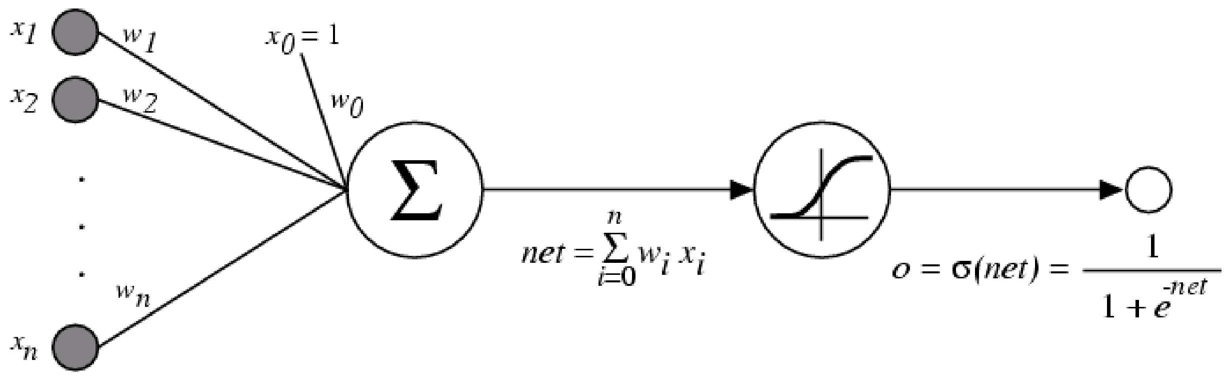
Input layer
Hidden layer
Output layer
$x_0 = 1$
x_1
x_2
$w_{0,1}$
$w_{0,2}$
$w_{1,1}$
$w_{2,1}$
$w_{1,2}$
$w_{2,2}$
$v_1 = \boldsymbol{\sigma}(\boldsymbol{w_1} \cdot \boldsymbol{x})$
$v_2 = \boldsymbol{\sigma}(\boldsymbol{w_2} \cdot \boldsymbol{x})$
$w_{3,1}$
$w_{3,2}$
$z = \boldsymbol{\sigma}(\boldsymbol{w_3} \cdot \boldsymbol{v})$

$$\boldsymbol{\sigma}(x) = \frac{1}{1+e^{-x}} =$$



Very useful in practice!

The Sigmoid Unit



x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1+e^{-net}}$

σ is the sigmoid function; $\sigma(x) = \frac{1}{1+e^{-x}}$

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)\big(1 - \sigma(x)\big)$

We can derive gradient descent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units → Backpropagation

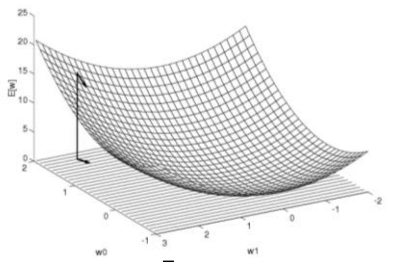
Gradient Descent to Minimize Squared Error

Goal: Given $(x_d, t_d)_{d \in D}$ find w to minimize $E_D[\boldsymbol{w}] = \frac{1}{2}\sum_{d \in D}(f_w(x_d) - t_d)^2$

Batch mode Gradient Descent:

Do until satisfied

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



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

Incremental (stochastic) Gradient Descent:

Do until satisfied

- For each training example d in D

1. Compute the gradient $\nabla E_d[\boldsymbol{w}]$
2. **w**$\leftarrow \boldsymbol{w} - \eta \nabla E_d[\boldsymbol{w}]$

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

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

Gradient descent in weight space

Goal: Given $(x_d, t_d)_{d \in D}$ find w to minimize $E_D[\mathbf{w}] = \frac{1}{2}\sum_{d \in D}(f_w(x_d) - t_d)^2$

This error measure defines a surface over the hypothesis (i.e. weight) space

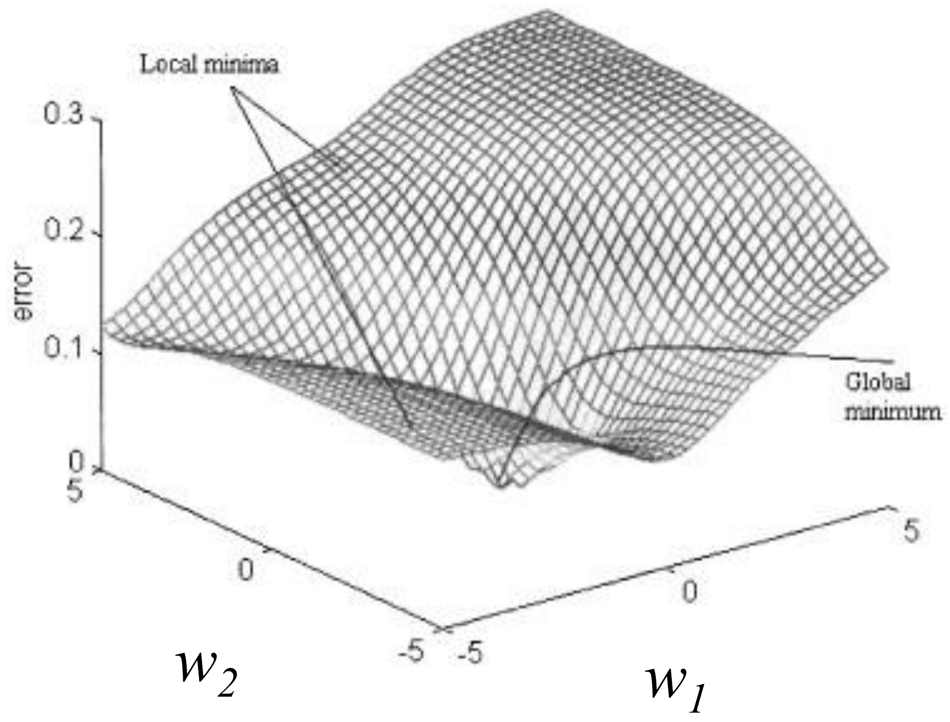


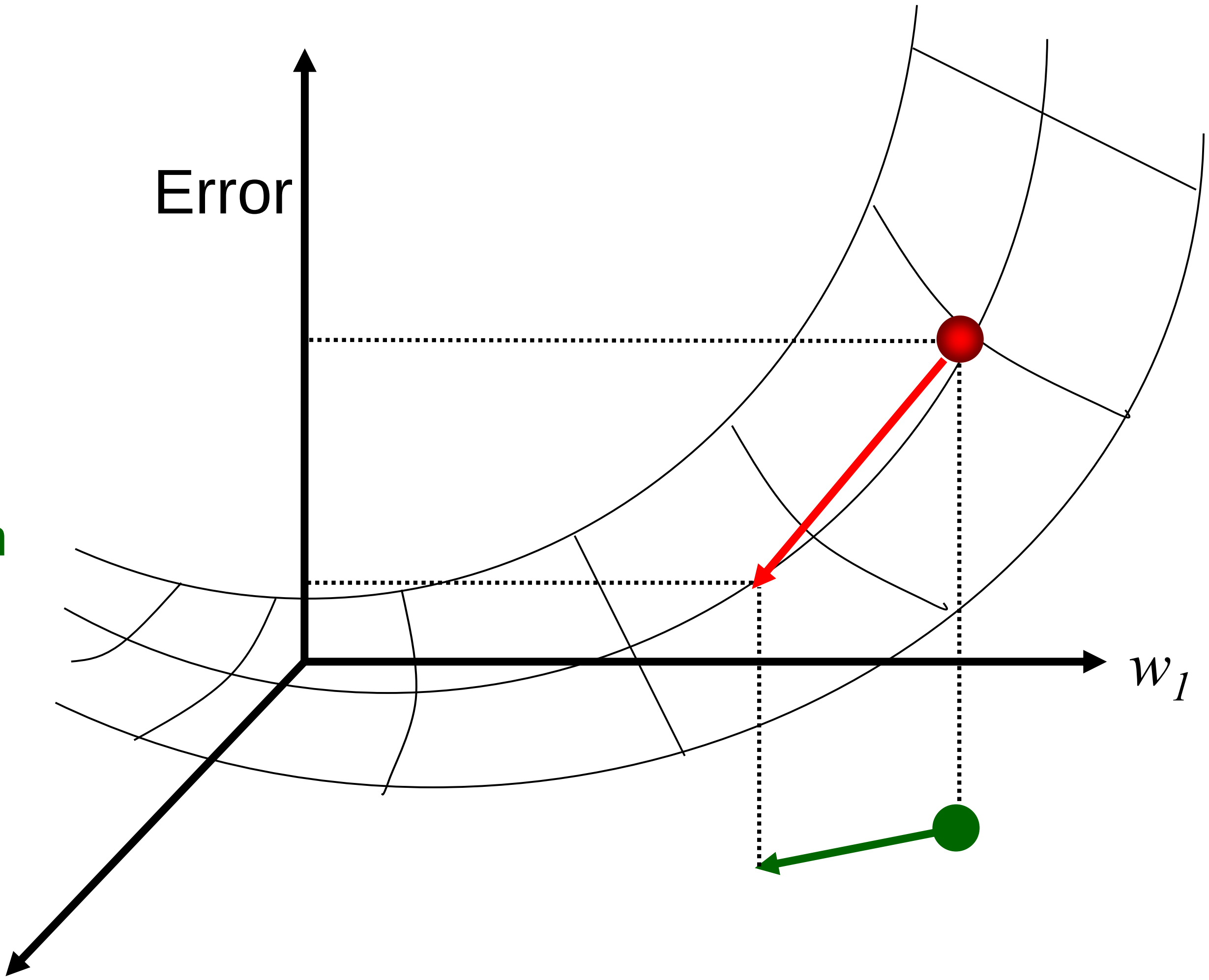
figure from Cho & Chow, *Neurocomputing* 1999

Gradient descent in weight space

Gradient descent is an iterative process aimed at finding a minimum in the error surface.

on each iteration

- current weights define a point in this space
- find direction in which error surface descends most steeply
- take a step (i.e. update weights) in that direction



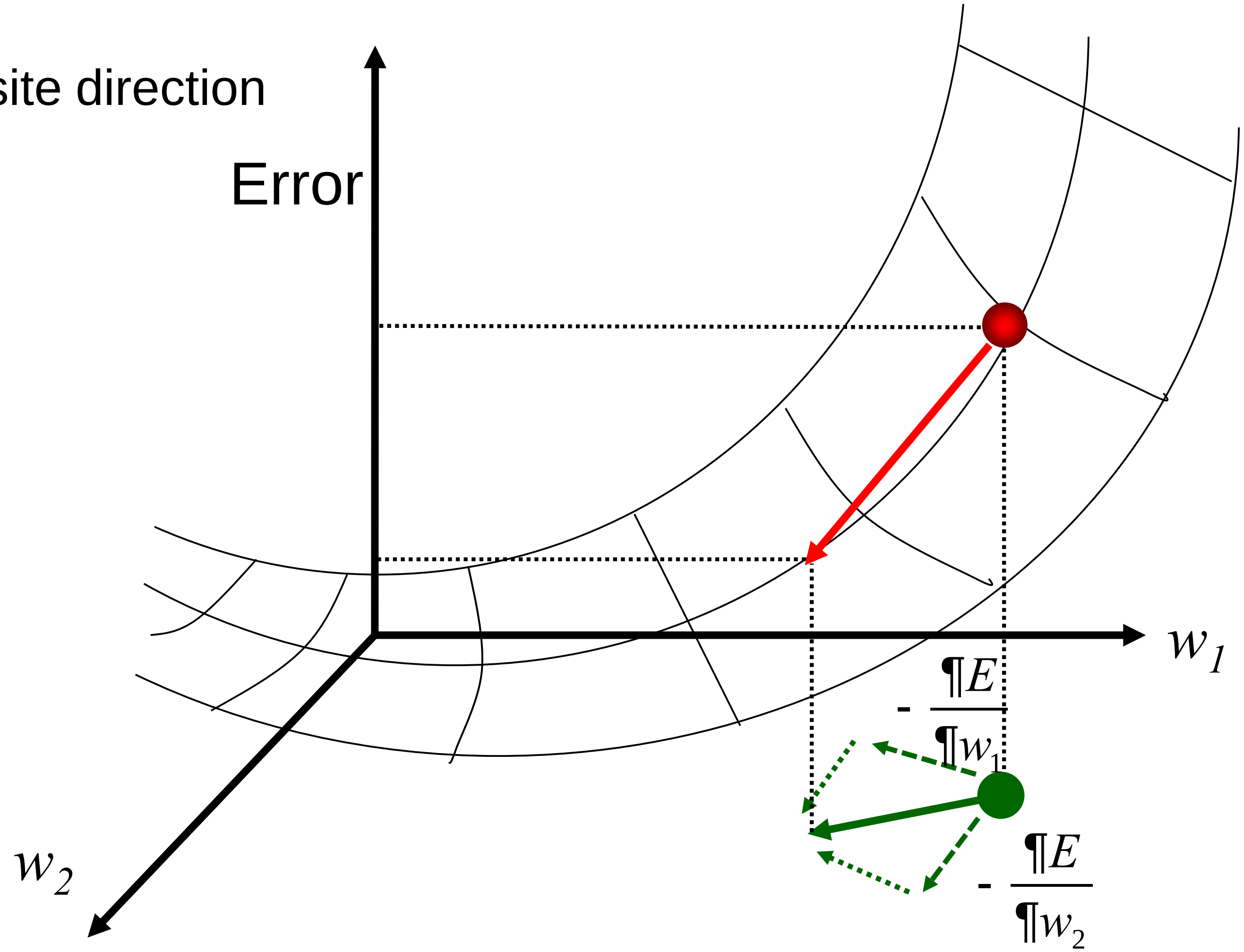
Gradient descent in weight space

Calculate the gradient of E: $\nabla E(\boldsymbol{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n} \right]$

Take a step in the opposite direction

$$\mathrm{D}\boldsymbol{w} = - h\, \nabla E\left(\boldsymbol{w}\right)$$

$$\mathrm{D}w_i = - h\, \frac{\partial E}{\partial w_i}$$



Taking derivative: chain rule

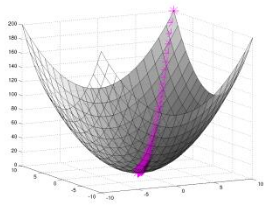
Recall the chain rule from calculus

$$y = f(u)$$

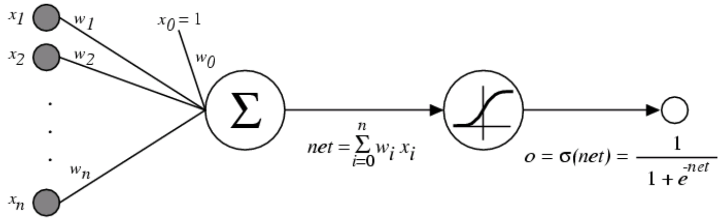
$$u = g(x)$$

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial x}$$

Gradient Descent for the Sigmoid Unit



Given $(x_d, t_d)_{d \in D}$ find **w** to minimize $\sum_{d \in D} (o_d - t_d)^2$



x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

o_d = observed unit output for x_d

$o_d = \sigma(net_d); \quad net_d = \sum_i w_i x_{i,d}$

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 = \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) = \sum_{d \in D} (t_d - o_d) \left(- \frac{\partial o_d}{\partial w_i} \right)$$

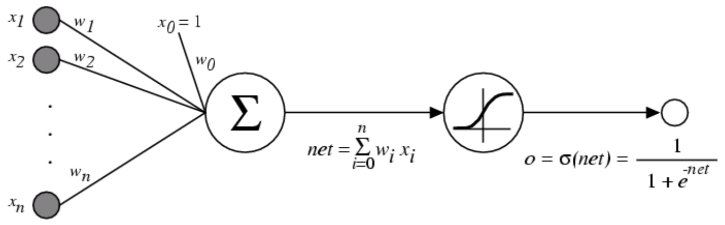
$$= - \sum_{d \in D} (t_d - o_d) \frac{\partial o_d}{\partial net_d} \frac{\partial net_d}{\partial w_i}$$

But we know: $\frac{\partial o_d}{\partial net_d} = \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1 - o_d)$ and $\frac{\partial net_d}{\partial w_i} = \frac{\partial(\mathbf{w} \cdot \mathbf{x_d})}{\partial w_i} = x_{i,d}$

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

Gradient Descent for the Sigmoid Unit

Given $(x_d, t_d)_{d \in D}$ find **w** to minimize $\sum_{d \in D} (o_d - t_d)^2$



x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

o_d = observed unit output for x_d

$$o_d = \sigma(net_d); \quad net_d = \sum_i w_i x_{i,d}$$

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

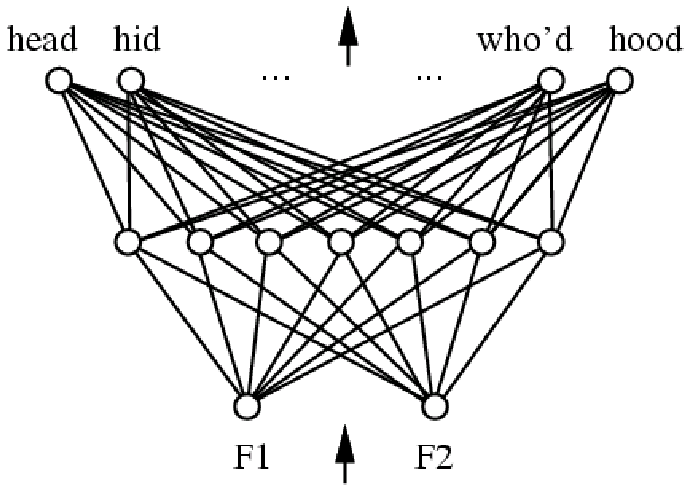
δ_d error term $t_d - o_d$ multiplied by $o_d(1 - o_d)$ that comes from the derivative of the sigmoid function

$$\frac{\partial E}{\partial w_i} = -\sum_{d \in D} \delta_d \, x_{i,d}$$

Update rule: $w \leftarrow w - \eta \nabla E[w]$

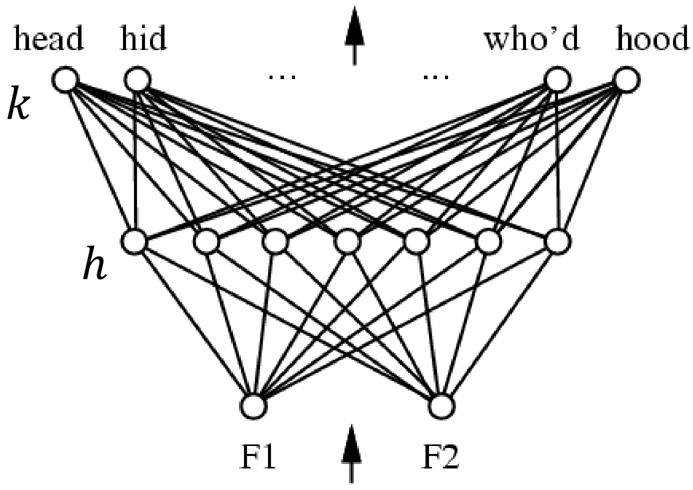
Gradient Descent for Multilayer Networks

Given $(x_d, t_d)_{d \in D}$ find **w** to minimize $\frac{1}{2}\sum_{d \in D}\sum_{k \in Outputs}(o_{k,d} - t_{kd})^2$



Backpropagation Algorithm

Incremental/stochastic gradient descent



o = observed unit output

t = target output

x = input

$x_{i,j}$ = ith input to jth unit

w_{ij} = wt from i to j

Initialize all weights to small random numbers.

Until satisfied, Do:

- **For** each training example (x, t) **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 h:
     $$\delta_h \leftarrow o_h(1 - o_h)\sum_{k \in outputs} w_{h,k}\delta_k$$
  4. Update each network weight $w_{i,j}$
     $$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$
     where $\Delta w_{i,j} = \eta \delta_j x_{i,j}$

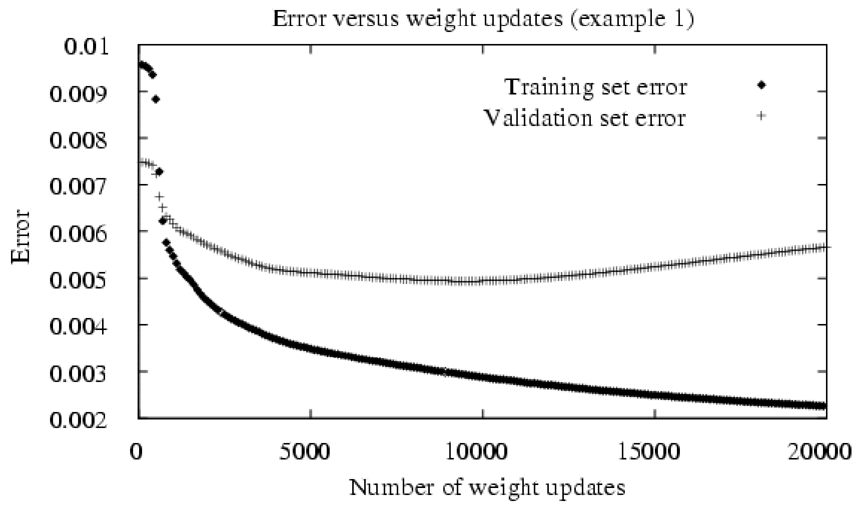
More on Backpropagation

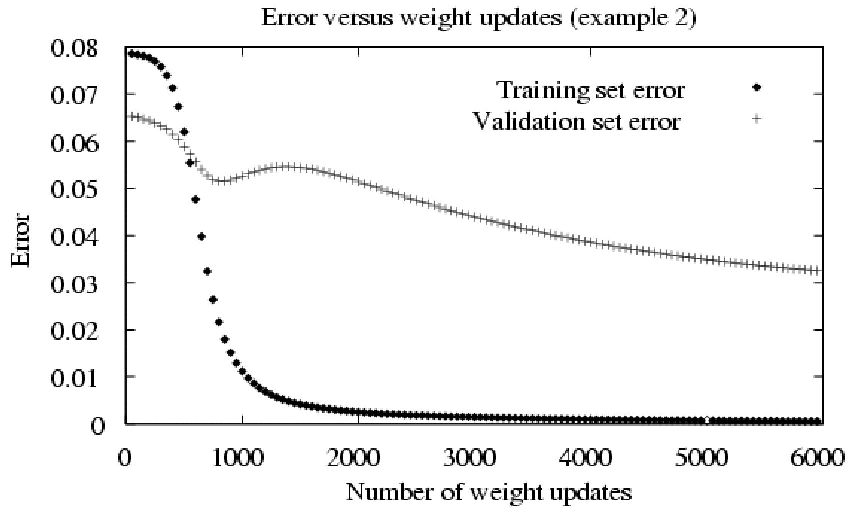
- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
  - In practice, often works well (can run multiple times)
- Often include weight *momentum* α

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

- Minimizes error over *training* examples
  - Will it generalize well to subsequent examples?
- Training can take thousands of iterations $\rightarrow$ slow!
- Using network after training is very fast

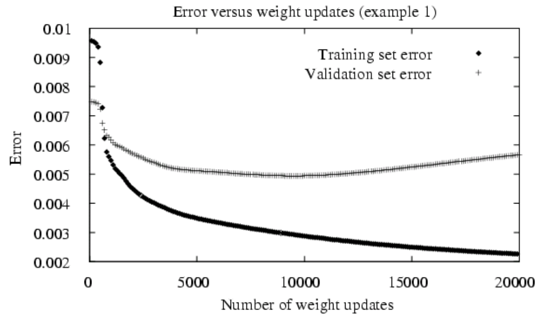
Overfitting in ANNs





- Validation/generalization error first decreases, then increases.
- Weights tuned to fit the idiosyncrasies of the training examples that are not representative of the general distribution.
- Stop when lowest error over validation set.
- Not always obvious when lowest error over validation set has been reached.

Dealing with Overfitting



Our learning algorithm involves a parameter

n=number of gradient descent iterations

How do we choose n to optimize future error?

- Separate available data into <u>training</u> and <u>validation</u> set
- Use <u>training</u> to perform gradient descent
- n ← number of iterations that optimizes <u>validation</u> set error

Dealing with Overfitting

- Regularization techniques
  - norm constraint
  - dropout
  - batch normalization
  - data augmentation
  - early stopping
  - …

Convergence of Backpropagation

Gradient descent to some local minimum

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

Nature of convergence

- Initialize weights near zero
- Therefore, initial networks near-linear
- Increasingly non-linear functions possible as training progresses

Expressive Capabilities of ANNs

Boolean functions:

- Every Boolean function can be represented by a network with a 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]

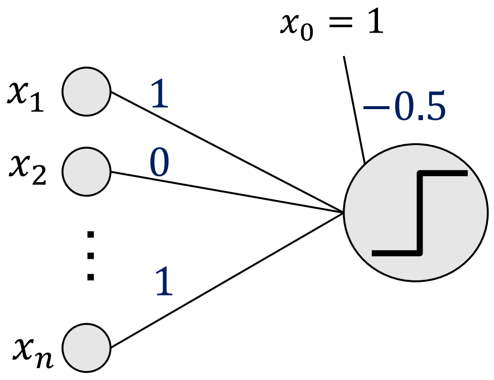
- Any function can be approximated to arbitrarily accuracy by a network with two hidden layers [Cybenko 1988]

Representing Simple Boolean Functions

Inputs $x_i \in \{0,1\}$

Or function

$$x_{i_1} \vee x_{i_2} \vee \cdots \vee x_{i_k}$$

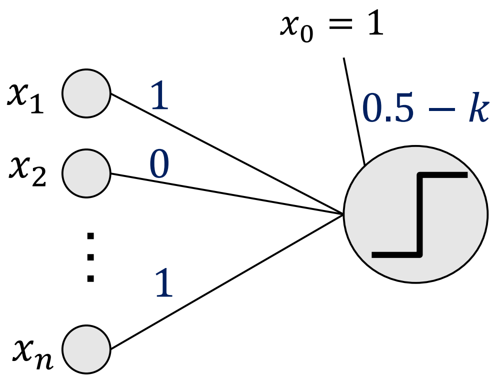


$w_i = 1$ if i is an i_j

$w_i = 0$ otherwise

And function

$$x_{i_1} \wedge x_{i_2} \wedge \cdots \wedge x_{i_k}$$

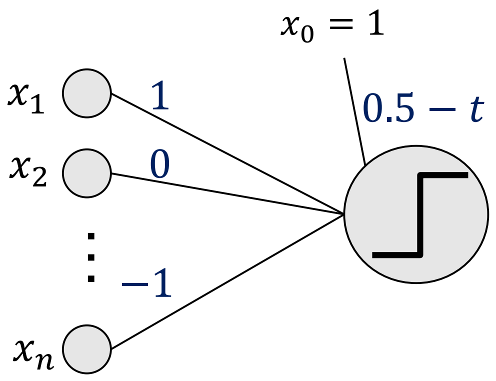


$w_i = 1$ if i is an i_j

$w_i = 0$ otherwise

And with negations

$$x_{i_1} \wedge \bar{x}_{i_2} \wedge \cdots \wedge x_{i_k}$$



$w_i = 1$ if i is i_j not negated

$w_i = -1$ if i is i_j negated

$w_i = 0$ otherwise

$t = \#$ not negated

General Boolean functions

Every Boolean function can be represented by a network with a single hidden layer; might require exponential # of hidden units

Can write any Boolean function as a truth table:

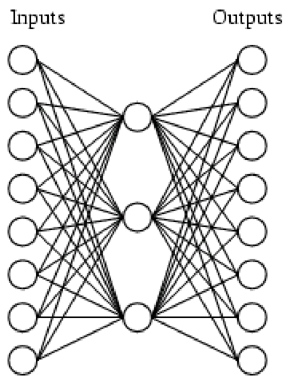
000	+
001	–
010	–
011	+
100	–
101	–
110	+
111	+

View as OR of ANDs, with one AND for each positive entry.

$$\bar{x}_1\bar{x}_2\bar{x}_3 \vee \bar{x}_1 x_2 x_3 \vee x_1 x_2 \bar{x}_3 \vee x_1 x_2 x_3$$

Then combine AND and OR networks into a 2-layer network.

Learning Hidden Layer Representations



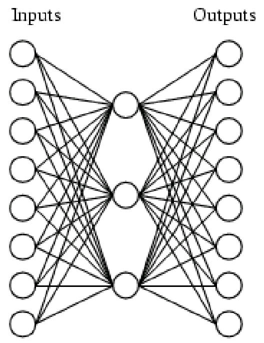
A target function:

Input		Output
10000000	$\rightarrow$	10000000
01000000	$\rightarrow$	01000000
00100000	$\rightarrow$	00100000
00010000	$\rightarrow$	00010000
00001000	$\rightarrow$	00001000
00000100	$\rightarrow$	00000100
00000010	$\rightarrow$	00000010
00000001	$\rightarrow$	00000001

Can this be learned??

Learning Hidden Layer Representations

A network:



Learned hidden layer representation:

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

Artificial Neural Networks: Summary

- Highly non-linear regression/classification
- Vector valued inputs and outputs
- Potentially millions of parameters to estimate
- Actively used to model distributed comptutation in the brain
- Hidden layers learn intermediate representations

- Stochastic gradient descent, local minima problems
- Overfitting and how to deal with it.

Other Activation Functions

- Problem with sigmoid: saturation

Too small gradient

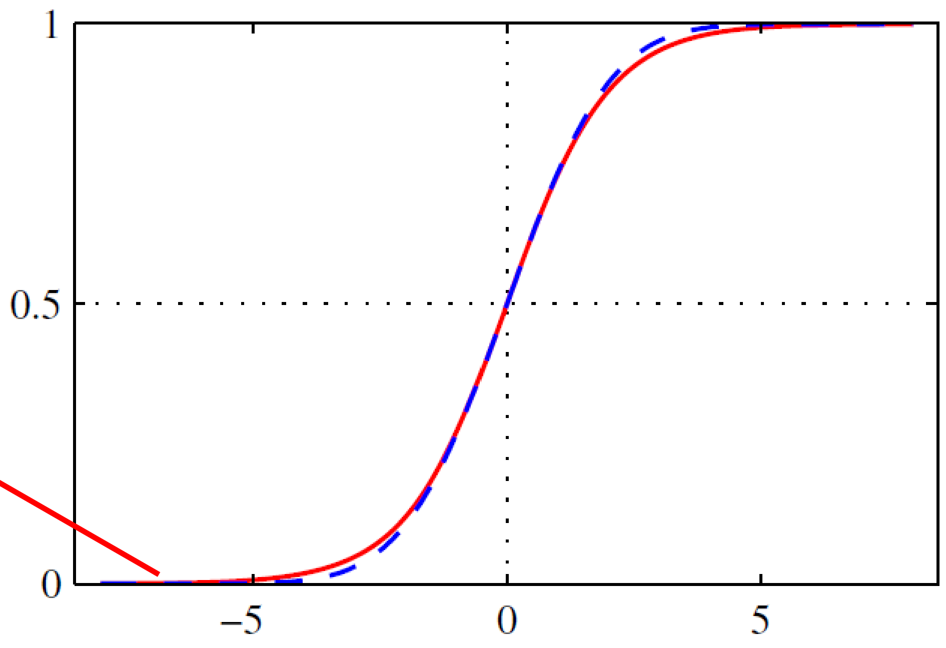


Figure borrowed from *Pattern Recognition and Machine Learning*, Bishop

Other Activation Functions

- Activation function ReLU (rectified linear unit)

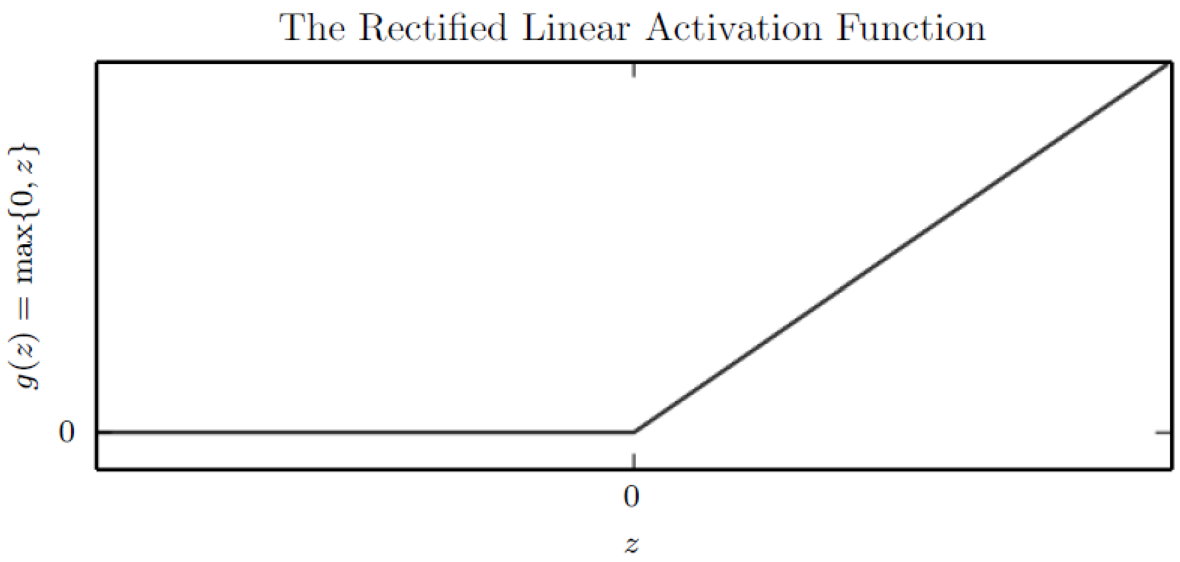
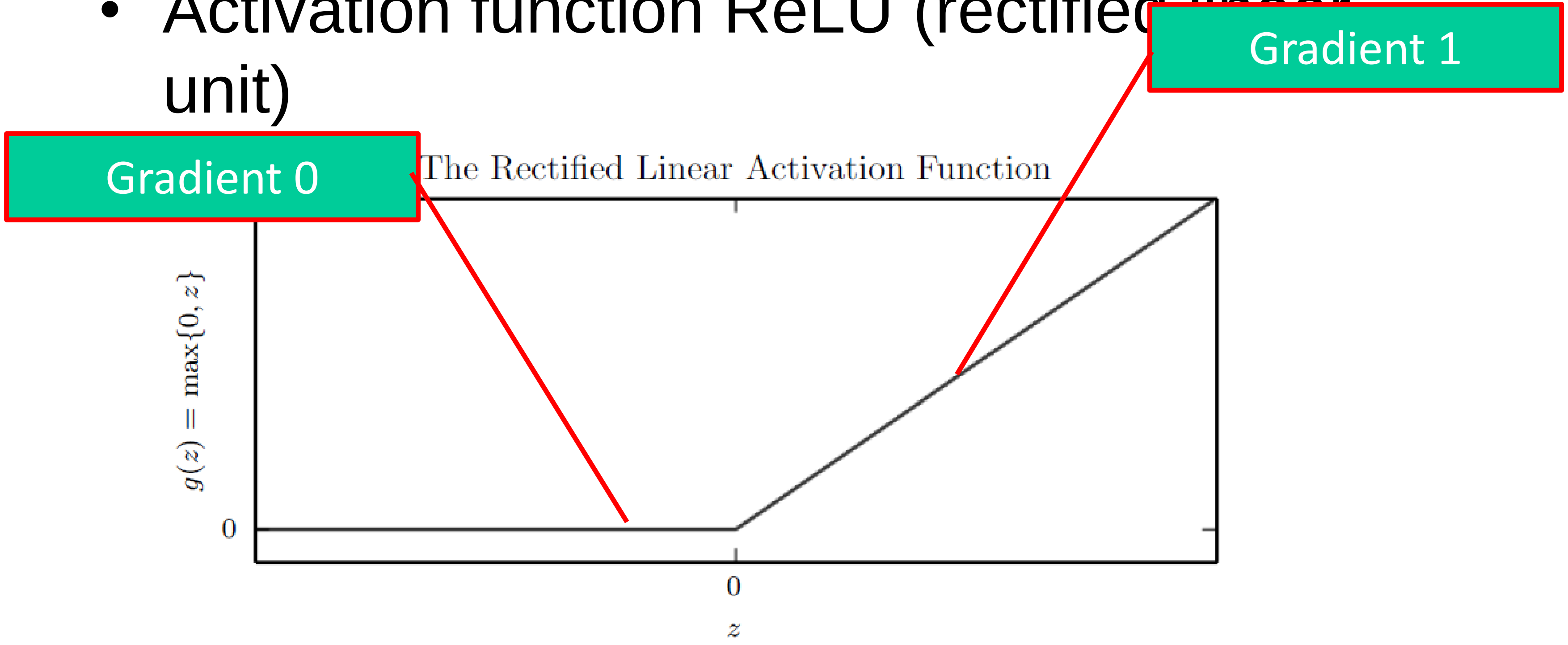


Figure from *Deep learning*, by Goodfellow, Bengio, Courville.

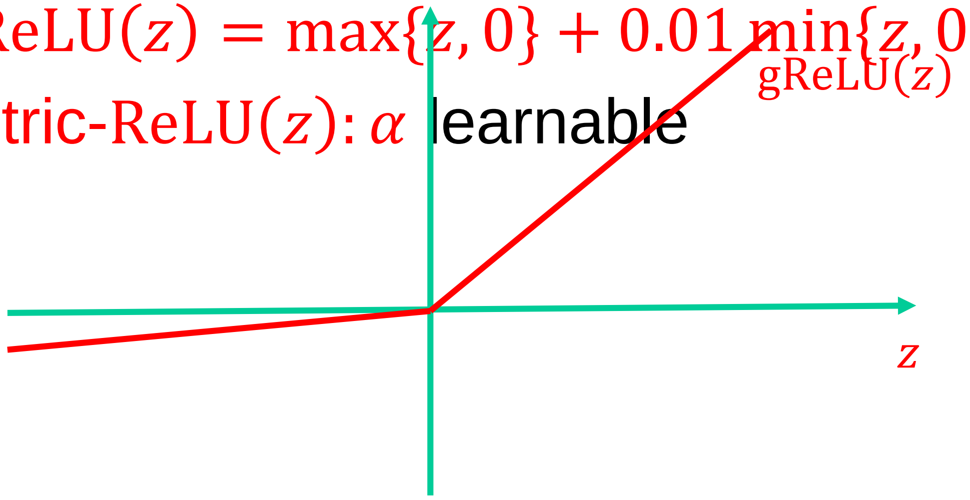
Other Activation Functions

- Activation function ReLU (rectified linear unit)



Other Activation Functions

- Generalizations of ReLU $\mathrm{gReLU}(z) = \max\{z, 0\} + \alpha \min\{z, 0\}$
  - Leaky-$\mathrm{ReLU}(z) = \max\{z, 0\} + 0.01 \min\{z, 0\}$
  - Parametric-$\mathrm{ReLU}(z)$: α learnable


$\mathrm{gReLU}(z)$
z