ECE7252
Statistical Learning for Signal Processing

Lectures 13-14: Artificial Neural Networks

Chin-Hui Lee
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332, USA
chl@ece.gatech.edu

Neural Computation

- Neuroscience
  - The objective is to understand the human brain
  - Biologically realistic models of neurons
  - Biologically realistic connection topologies
- Neural computation
  - The objective is to develop learning, representation and computation methods
  - Novel architectures for data representation and processing

The Goals of Neural Computation

- To understand how the brain actually works
  - Its big and very complicated and made of yukky stuff that dies when you poke it around
- To understand a new style of computation
  - Inspired by neurons and their adaptive connections
  - Very different style from sequential computation
    - should be good for things that brains are good at (e.g. vision)
    - Should be bad for things that brains are bad at (e.g. 23 x 71)
- To solve practical problems by using novel learning algorithms
  - Learning algorithms can be very useful even if they have nothing to do with how the brain works

Human vs. Machine (Hardware)

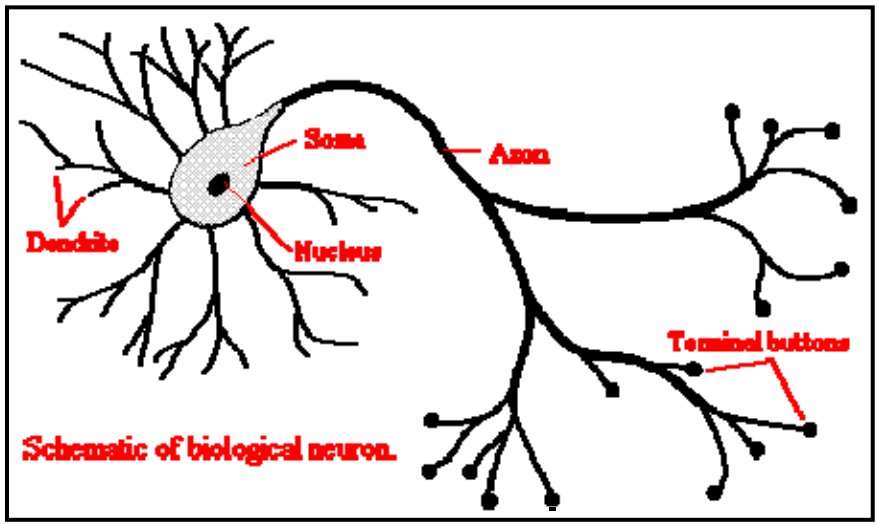
Numbers	Human brain	Von Neumann computer
# elements	10^{10} - 10^{12} neurons	10^{7} - 10^{8} transistors
# connections / element	10^{4} - 10^{3}	10
switching frequency	10^{3} Hz	10^{9} Hz
energy / operation	10^{-16} Joule	10^{-6} Joule
power consumption	10 Watt	100 - 500 Watt
reliability of elements	low	reasonable
reliability of system	high	reasonable

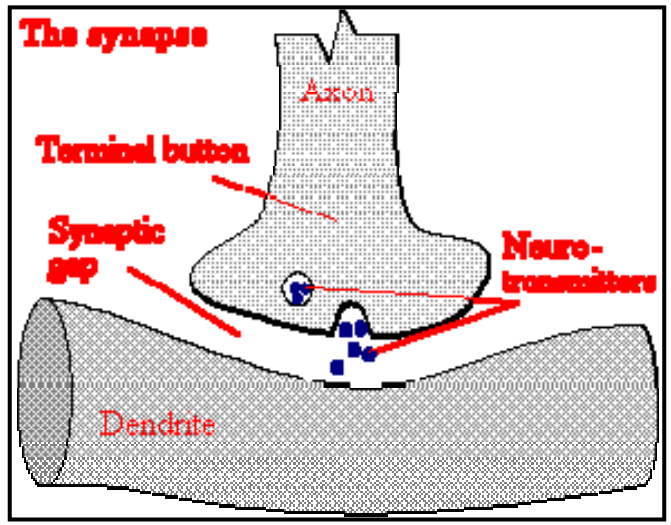
Human vs. Machine (Information Processing)

Features	Human Brain	Von Neumann computer
Data representation	analog	digital
Memory localization	distributed	localized
Control	distributed	localized
Processing	parallel	sequential
Skill acquisition	learning	programming

No memory management, no hardware/software/data distinction

The Biological Neuron





- 10 billion neurons in human brain
- Summation of input stimuli
  - Spatial (signals)
  - Temporal (pulses)
- Threshold over composed inputs
- Constant firing strength

- 10^6 billion synapses in human brain
- Chemical transmission and modulation of signals
- Inhibitory synapses
- Excitatory synapses

Biological Neural Networks

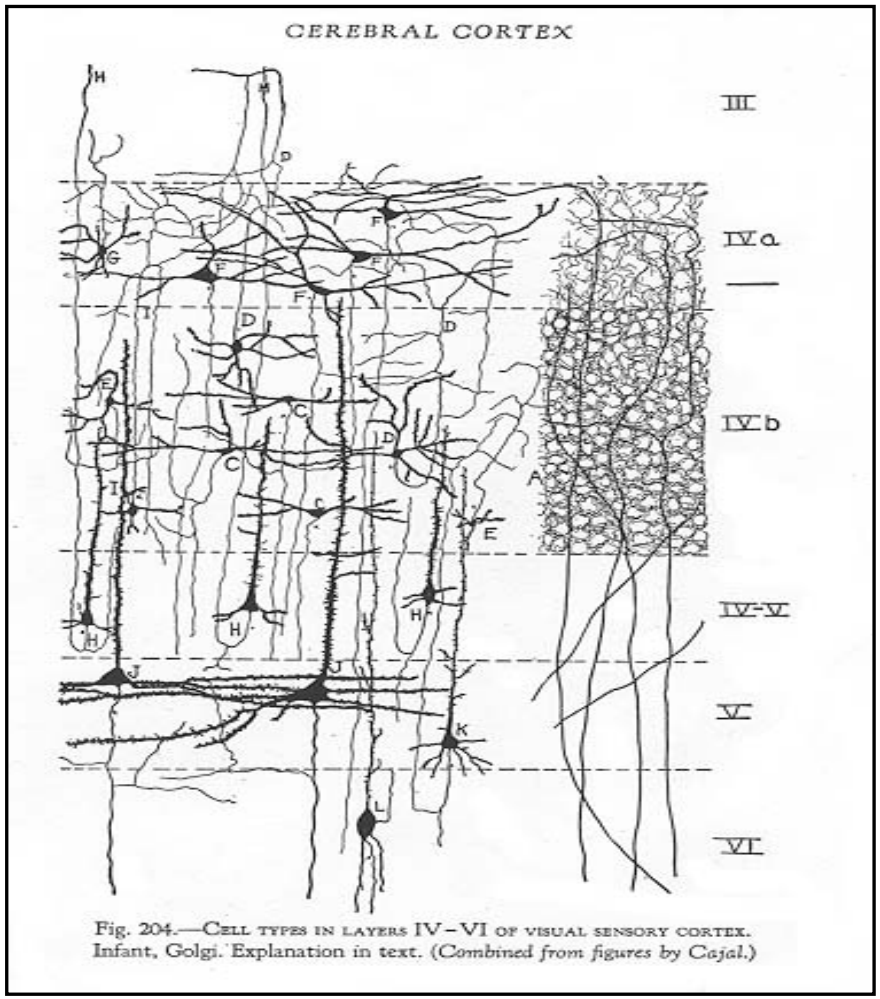


Fig. 204.—Cell types in layers IV–VI of visual sensory cortex. Infant, Golgi. Explanation in text. (*Combined from figures by Cajal.*)

- 10,000 synapses per neuron
- Computational power = connectivity
- Plasticity
  - new connections (?)
  - strength of connections modified

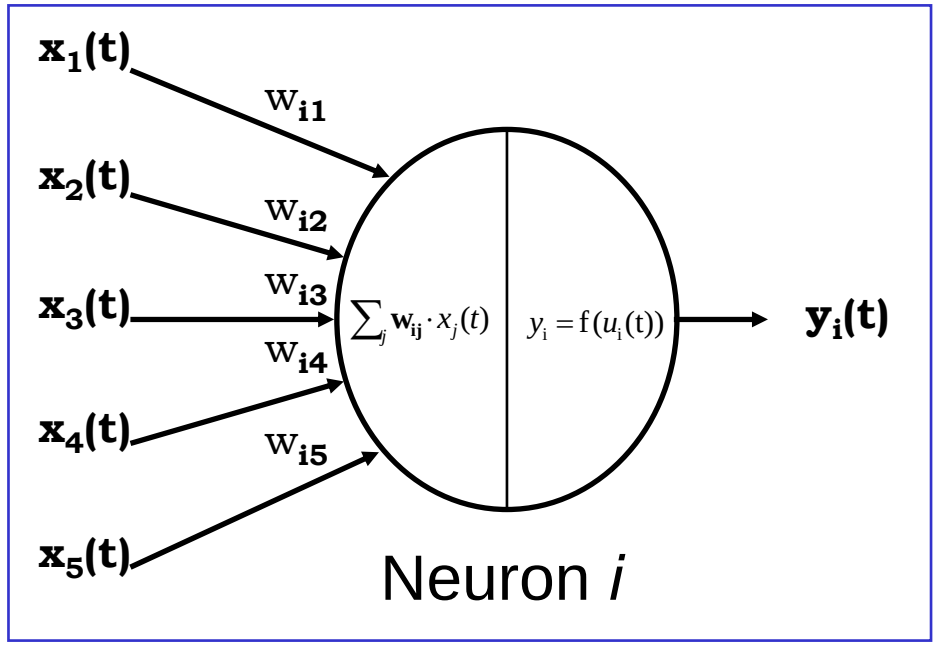
The Artificial Neuron

Stimulus

$$u_i(t) = \sum_j w_{ij} \cdot x_j(t)$$

Response

$$y_i(t) = f\left(u_{rest} + u_i(t)\right)$$

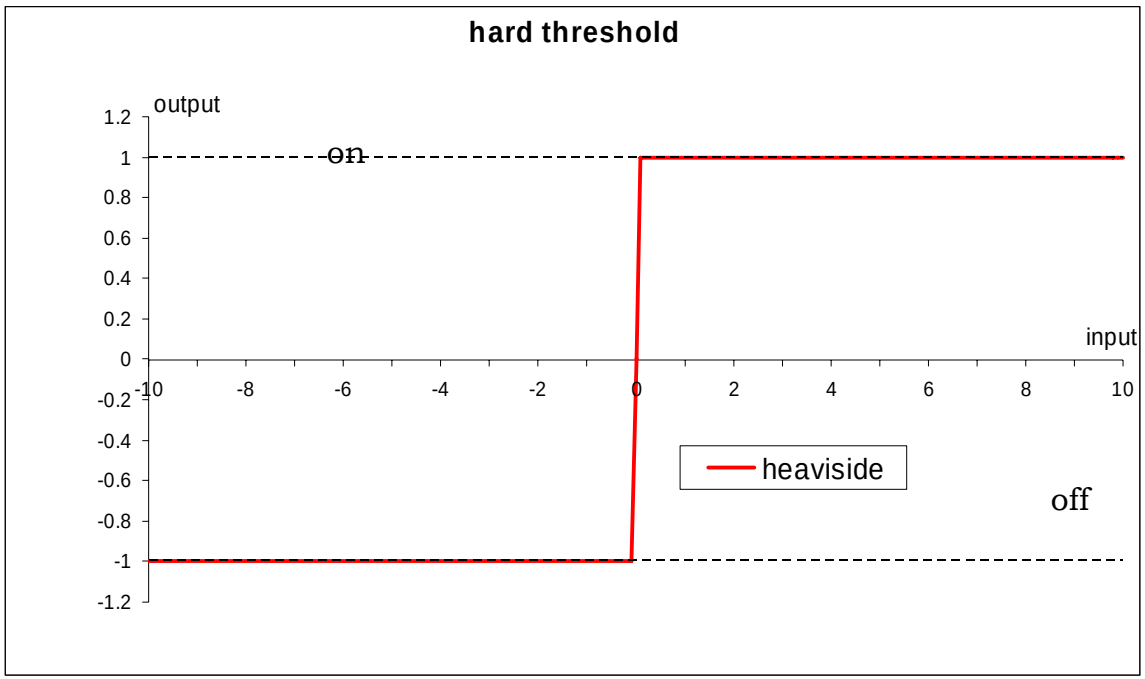


u_{rest} = resting potential
$x_j(t)$ = output of neuron j at time t
w_{ij} = connection strength between neuron i and neuron j
$u(t)$ = total stimulus at time t

Artificial Neural Models

- McCulloch Pitt-type Neurons (static)
  - Digital neurons: activation state interpretation (snapshot of the system each time a unit fires)
  - Analog neurons: firing rate interpretation (activation of units equal to firing rate)
  - Activation of neurons encodes information
- Spiking Neurons (dynamic)
  - Firing pattern interpretation (spike trains of units)
  - Timing of spike trains encodes information (time to first spike, phase of signal, correlation and synchronicity

Binary Neurons



Stimulus

$$u_i = \sum_j w_{ij} \cdot x_j$$

Response

$$y_i = f\left(u_{rest} + u_i\right)$$

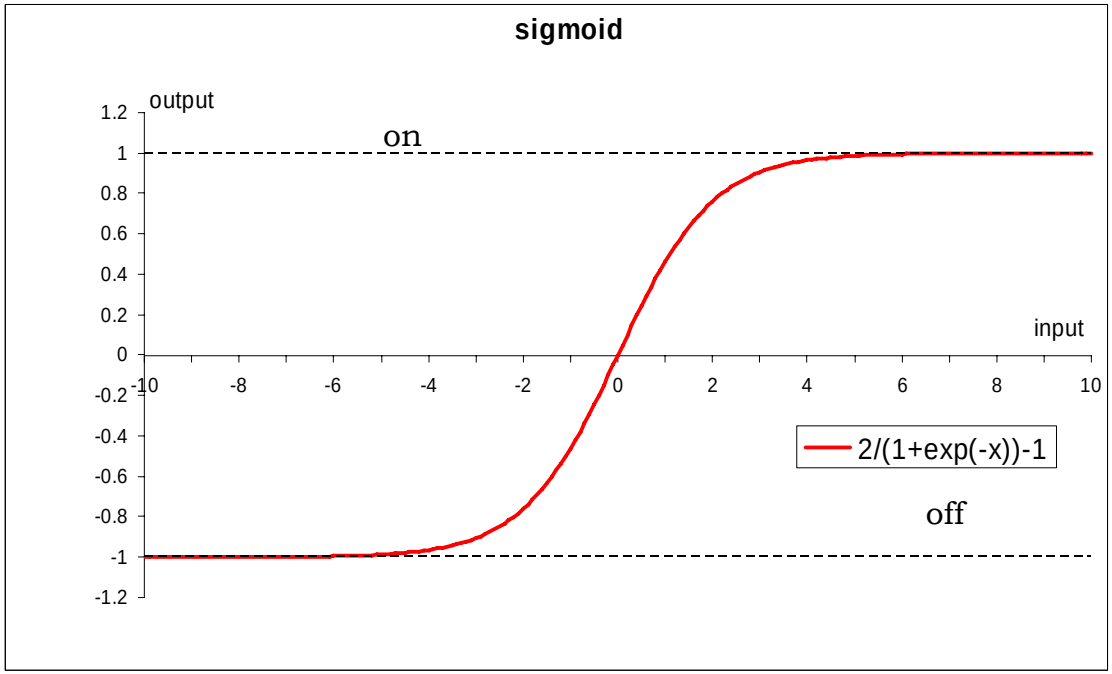
"Hard" threshold

$$f(z) = \begin{cases} z \geq \Theta \Rightarrow ON \\ else \Rightarrow OFF \end{cases}$$

Θ= threshold

- ex: Perceptrons, Hopfield NNs, Boltzmann Machines
- Main drawbacks: can only map binary functions, biologically implausible

Analog Neurons



Stimulus

$$u_i = \sum_j w_{ij} \cdot x_j$$

Response

$$y_i = f\left(u_{rest} + u_i\right)$$

"Soft" threshold

$$f(z) = \frac{2}{1+e^{-z}} - 1$$

- ex: MLPs, Recurrent NNs, RBF NNs...
- Main drawbacks: difficult to process time patterns, biologically implausible

Spiking Neurons

Stimulus

$$u_i(t) = \sum_j w_{ij} \cdot x_j(t)$$

η = spike and afterspike potential
u_{rest} = resting potential
ε(t,u(τ)) = trace at time t of input at time τ
Θ= threshold
$x_j(t)$ = output of neuron j at time t
w_{ij} = efficacy of synapse from neuron i to neuron j
$u(t)$ = input stimulus at time t

Response

$$y_i(t) = f\left(u_{rest} + \eta(t - t_f) + \sum_{\tau=0}^{t} \varepsilon(t, u_i(\tau))\right)$$

$$f(z) = \begin{cases} z \geq \Theta \;\&\; \dfrac{dz}{dt} > 0 \Rightarrow ON \\ else \Rightarrow OFF \end{cases}$$

Idealized Neurons

- To model things we have to idealize them (e.g. atoms)
  - Idealization removes complicated details that are not essential for understanding the main principles
  - Allows us to apply mathematics and to make analogies to other, familiar systems.
  - Once we understand the basic principles, its easy to add complexity to make the model more faithful
- It is often worth understanding models that are known to be wrong (but we mustn't forget that they are wrong!)
  - E.g. neurons that communicate real values rather than discrete spikes of activity.

Linear Neurons

- These are simple but computationally limited
  - If we can make them learn we may get insight into more complicated neurons

bias

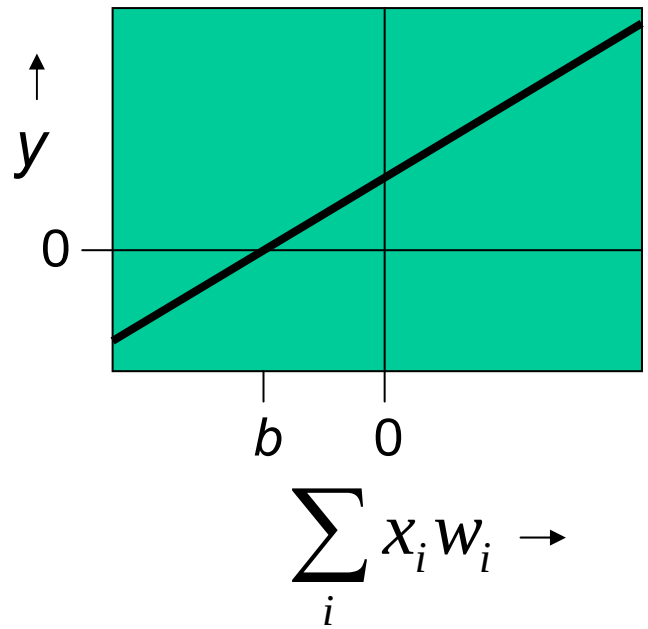
i^{th} input

$$y = b + \sum_i x_i w_i$$

output

index over input connections

weight on i^{th} input

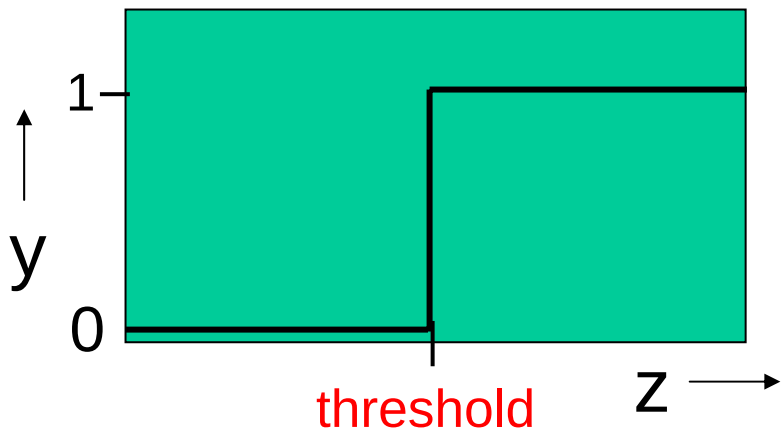


Binary Threshold Neurons

- McCulloch-Pitts (1943): influenced Von Neumann!
  - Compute a weighted sum of the inputs from other neurons
  - Then send out a fixed size spike of activity if the weighted sum exceeds a threshold.
  - Maybe each spike is like the truth value of a proposition and each neuron combines truth values to compute the truth value of another proposition!

$$z = \sum_i x_i w_i$$

$$y = \begin{cases} 1 \text{ if } z \geq \theta \\ 0 \text{ otherwise} \end{cases}$$

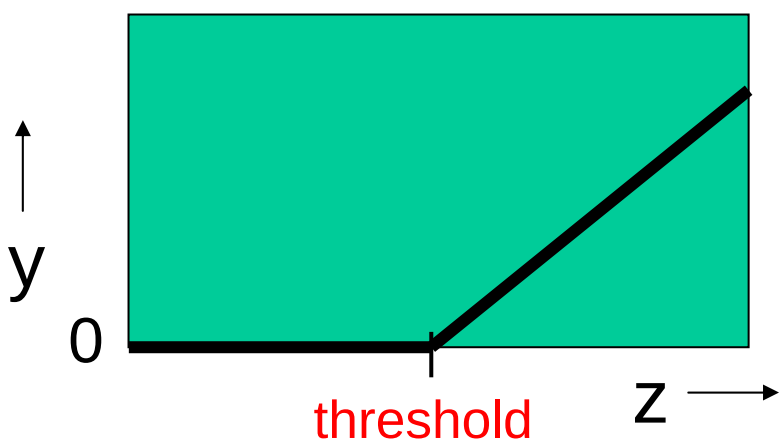


Linear Threshold Neurons

- These have a confusing name.
- They compute a linear weighted sum of their inputs
- The output is a non-linear function of the total input

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

$$y_j = \begin{cases} z_j \text{ if } z_j \geq 0 \\ 0 \text{ otherwise} \end{cases}$$

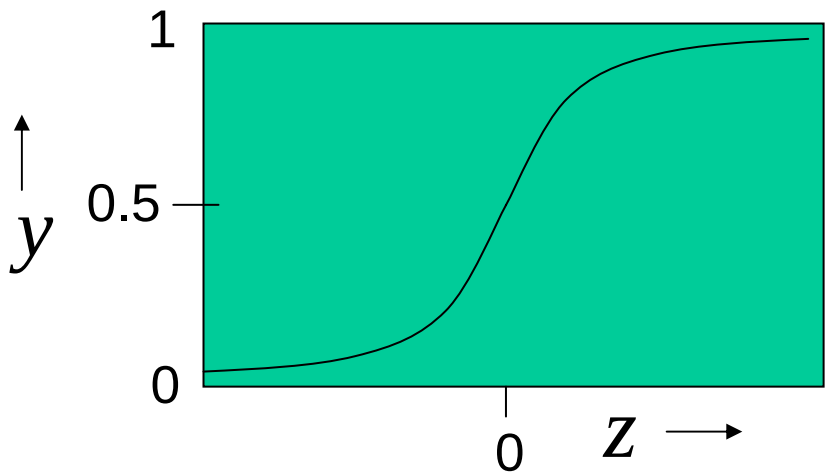


Sigmoid Neurons

- These give a real-valued output that is a smooth and bounded function of their total input
  - –Typically they use the logistic function
  - –They have nice derivatives which make learning easy
- If we treat y as a probability of producing a spike, we get stochastic binary neurons

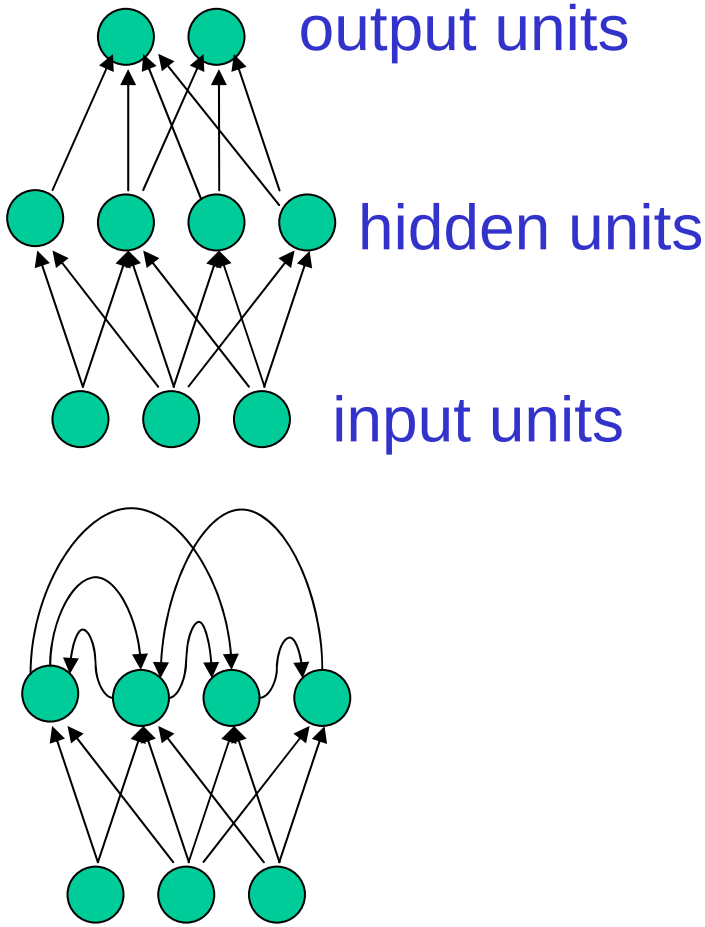
$$z = b + \sum_i x_i w_i$$

$$y = \frac{1}{1 + e^{-z}}$$



Types of Connectivity

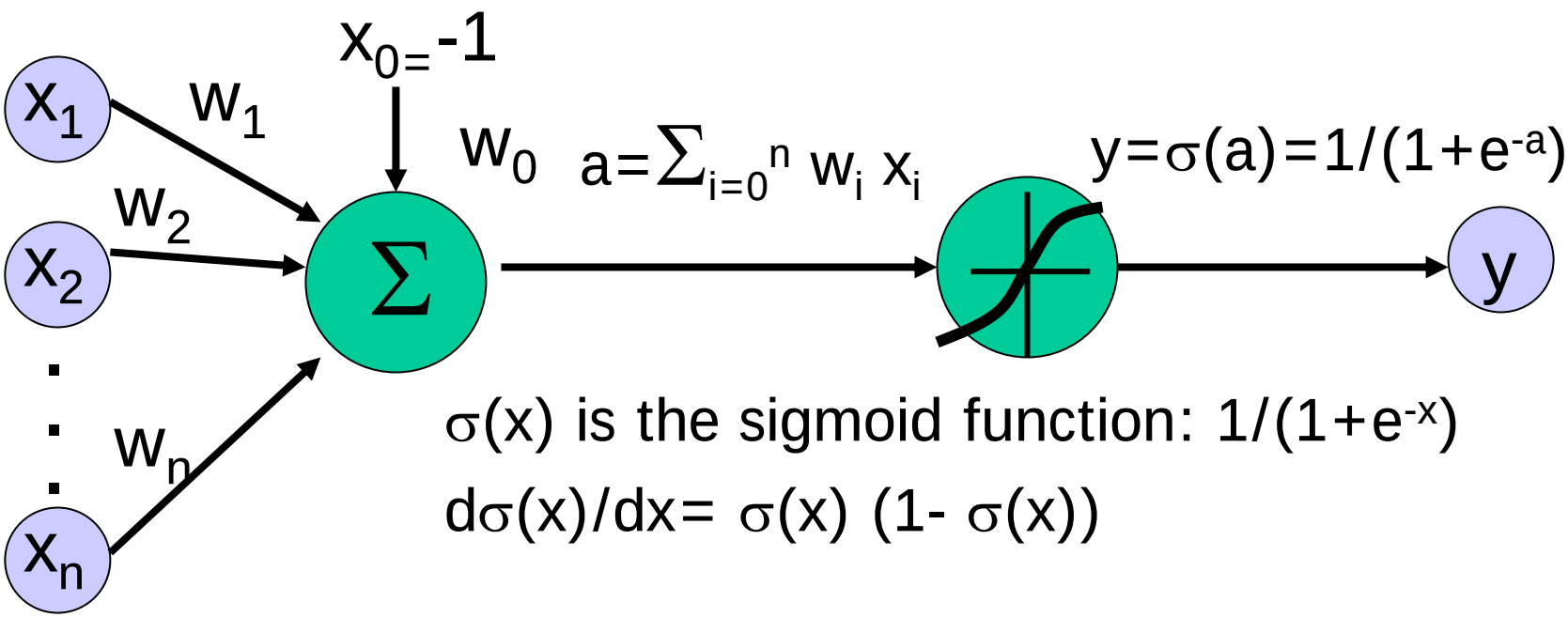
- Feedforward networks
  - –These compute a series of transformations
  - –Typically, the first layer is the input and the last layer is the output
- Recurrent networks
  - – These have directed cycles in their connection graph They can have complicated dynamics
  - – More biologically realistic



Types of Learning Task

- Supervised learning
  - Learn to predict output with given input vector
    - Who provides the correct answer?
- Reinforcement learning
  - Learn action to maximize payoff
    - Not much information in a payoff signal
    - Payoff is often delayed
- Unsupervised learning
  - Create an internal representation of the input e.g. form clusters; extract features
    - How do we know if a representation is good?

Sigmoid Unit



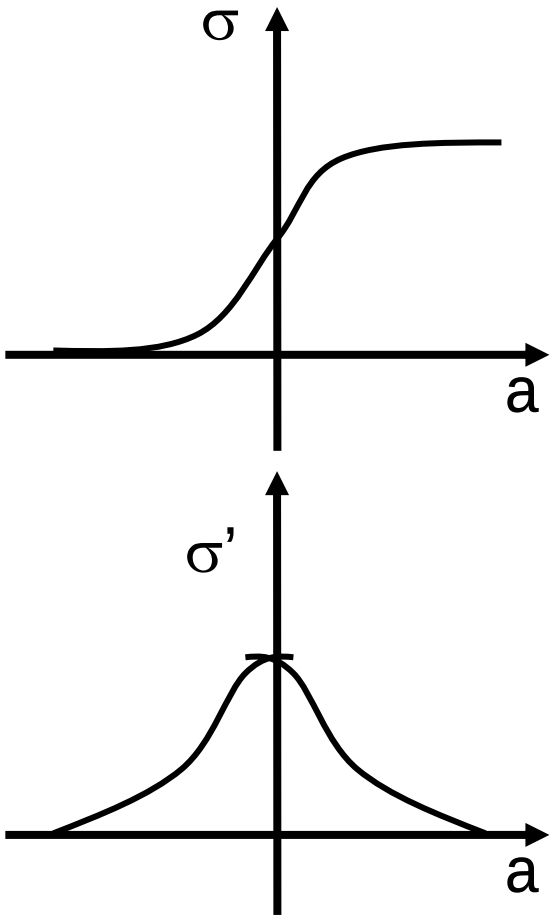
$\sigma(x)$ is the sigmoid function: $1/(1+e^{-x})$

$d\sigma(x)/dx= \sigma(x)\ (1- \sigma(x))$

Derive gradient decent rules to train:

- One sigmoid function: $\partial E/\partial w_i = -\Sigma_p (t^p-y)\ y\ (1-y)\ x_i^p$
- Multilayer networks of sigmoid units: backpropagation

Gradient Descent for Sigmoid Output



Sigmoid: $E^p[w_1,\ldots,w_n] = \frac{1}{2}(t^p-y^p)^2$

t^p: Desired output

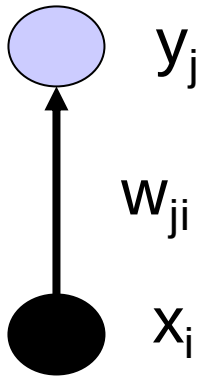
$$\partial E^p/\partial w_i = \partial/\partial w_i \, \frac{1}{2}(t^p-y^p)^2$$
$$= \partial/\partial w_i \, \frac{1}{2}(t^p- \sigma(\Sigma_i w_i x_i^p))^2$$
$$= (t^p-y^p)\,\sigma'(\Sigma_i w_i x_i^p)\,(-x_i^p)$$

for $y=\sigma(a) = 1/(1+e^{-a})$

$\sigma'(a)= e^{-a}/(1+e^{-a})^2=\sigma(a)\,(1-\sigma(a))$

$$w'_i = w_i + \Delta w_i = w_i + \alpha\, y(1-y)(t^p-y^p)\, x_i^p$$

Gradient Descent Learning Rule



$$\Delta w_i = \alpha \; y_j^p(1-y_j^p) \; (t_j^p-y_j^p) \; x_i^p$$

learning rate

derivative of activation function

error δ_j of post-synaptic neuron

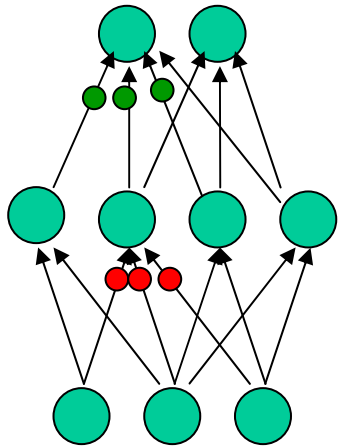
activation of pre-synaptic neuron

Learning with Hidden Units

- Networks without hidden units are very limited in the input-output mappings they can model
  - More layers of linear units do not help. Its still linear.
  - Fixed output non-linearities are not enough
- We need multiple layers of adaptive non-linear hidden units. This gives us a universal approximator. But how can we train such nets?
  - We need an efficient way of adapting all the weights, not just the last layer. This is hard. Learning the weights going into hidden units is equivalent to learning features.
  - Nobody is telling us directly what hidden units should do

Learning by Perturbing Weights

- Randomly perturb one weight and see if it improves performance. If so, save the change
  - Very inefficient. We need to do multiple forward passes on a representative set of training data just to change one weight.
  - Towards the end of learning, large weight perturbations will nearly always make things worse.
- We could randomly perturb all the weights in parallel and correlate the performance gain with the weight changes
  - Not any better because we need lots of trials to "see" the effect of changing one weight through the noise created by all the others.

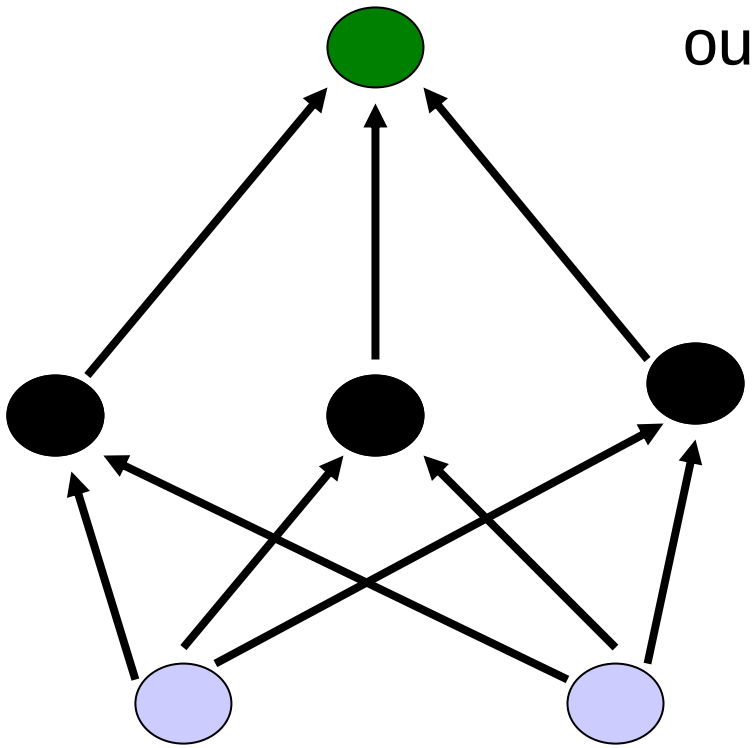


Learning the hidden to output weights is easy. Learning the input to hidden weights is hard.

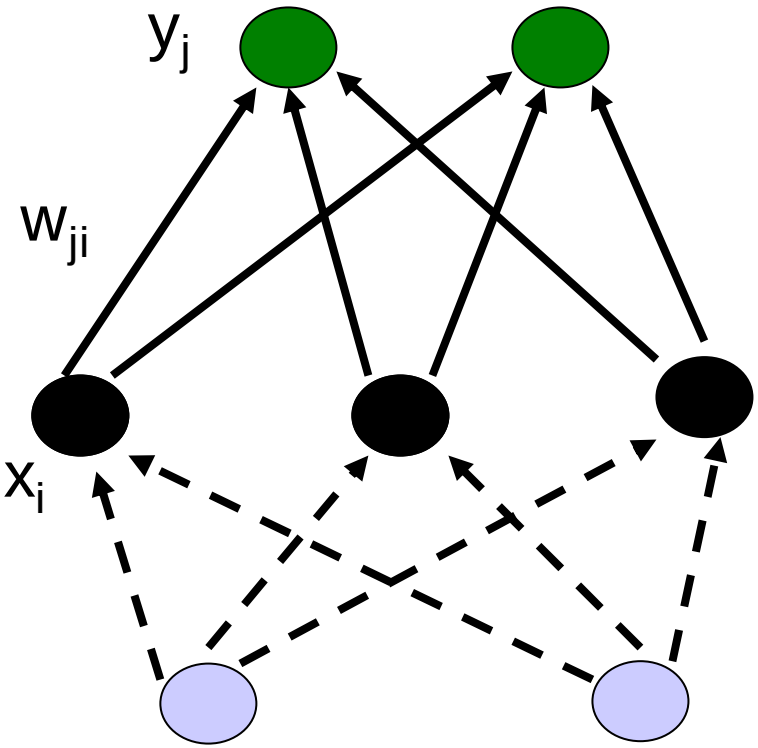
The Idea behind Backpropagation

- We don't know what the hidden units ought to do, but we can compute how fast the error changes as we change a hidden activity
  - Instead of using desired activities to train the hidden units, use error derivatives w.r.t. hidden activities
  - Each hidden activity can affect many output units and can therefore have many separate effects on the error. These effects must be combined
  - We can compute error derivatives for all the hidden units efficiently
  - Once we have the error derivatives for the hidden activities, its easy to get the error derivatives for the weights going into a hidden unit

Multi-Layer Networks



Training-Rule for Weights to Output Layer



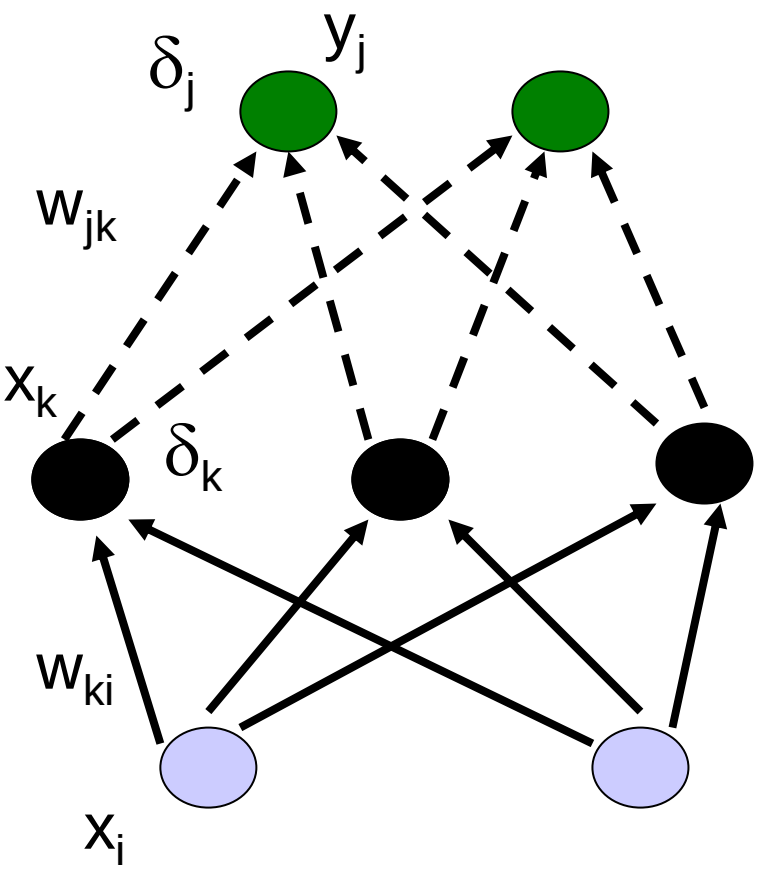
$$E^p[w_{ij}] = \tfrac{1}{2} \sum_j (t_j^p - y_j^p)^2$$

$$\partial E^p/\partial w_{ij} = \partial/\partial w_{ij}\ \tfrac{1}{2} \sum_j (t_j^p - y_j^p)^2$$
$$= \ldots$$
$$= -\, y_j^p(1-y^p_j)(t^p_j - y^p_j)\ \ x_i^p$$

$$\Delta w_{ij} = \alpha\ \ y_j^p(1-y_j^p)\ (t^p_j - y_j^p)\ x_i^p$$
$$= \alpha\ \ \delta_j^p\ x_i^p$$

with $\delta_j^p := y_j^p(1-y_j^p)\ (t^p_j - y_j^p)$

Training-Rule for Weights to Hidden Layer



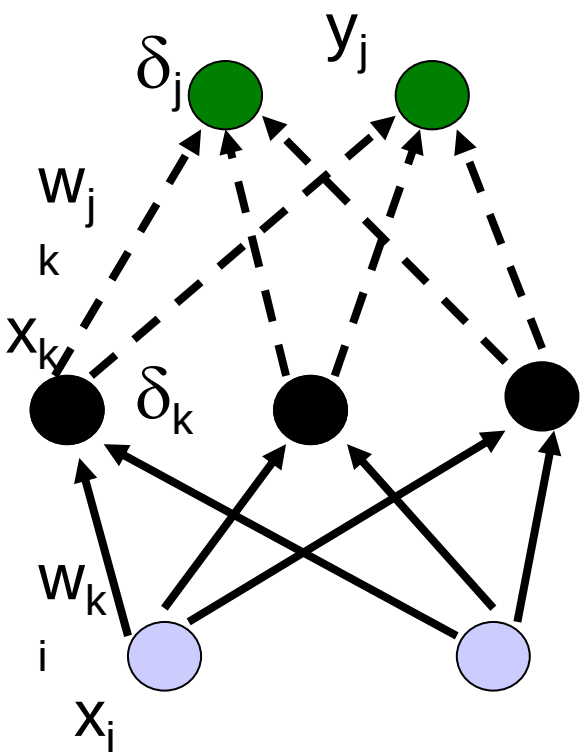
- Credit assignment problem:
No target values for hidden layer units

Error for hidden units?

$$\delta_k = \Sigma_j \, w_{jk} \, \delta_j \, y_j \, (1-y_j)$$

$$\Delta w_{ki} = \alpha \; x_k^p(1-x_k^p) \; \delta_k^p \; x_i^p$$

Training-Rule for Hidden Layer



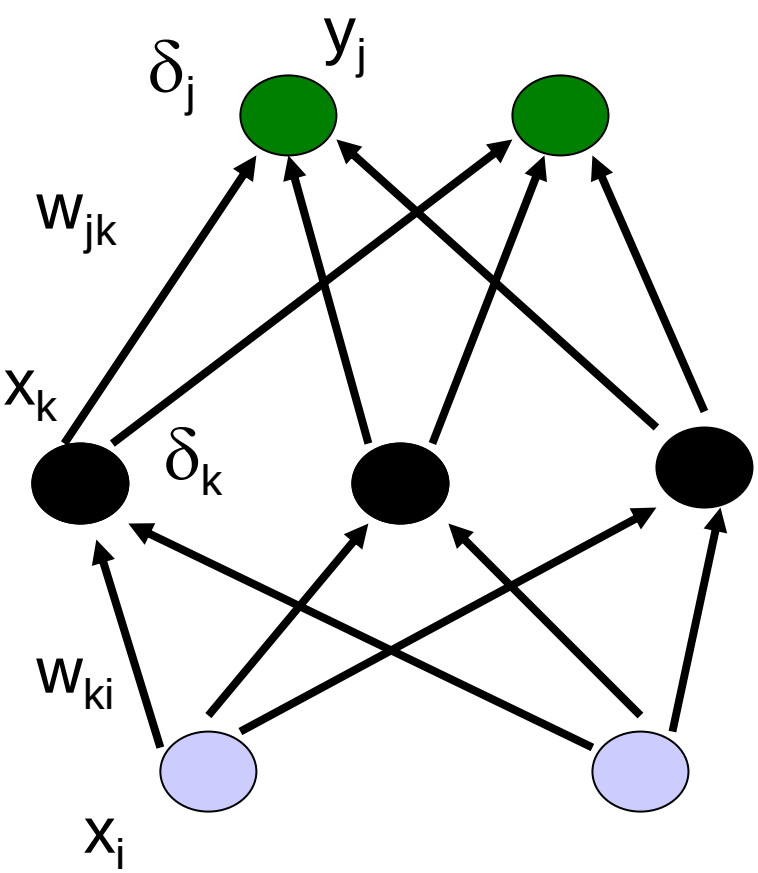
$$E^p[w_{ki}] = \tfrac{1}{2} \Sigma_j (t_j^p - y_j^p)^2$$

$$\partial E^p/\partial w_{ki} = \partial/\partial w_{ki}\ \tfrac{1}{2} \Sigma_j (t_j^p - y_j^p)^2$$
$$= \partial/\partial w_{ki}\ \tfrac{1}{2} \Sigma_j (t_j^p - \sigma(\Sigma_k w_{jk}\ x_k^p))^2$$
$$= \partial/\partial w_{ki}\ \tfrac{1}{2} \Sigma_j (t_j^p - \sigma(\Sigma_k w_{jk}\ \sigma(\Sigma_i w_{ki}\ x_i^p)))^2$$
$$= -\Sigma_j (t_j^p - y_j^p)\ \sigma'_j(a)\ w_{jk}\ \sigma'_k(a)\ x_i^p$$
$$= -\Sigma_j\ \delta_j\ w_{jk}\ \sigma'_k(a)\ x_i^p$$
$$= -\Sigma_j\ \delta_j\ w_{jk}\ x_k\ (1-x_k)\ x_i^p$$

$$\Delta w_{ki} = \alpha\ \delta_k\ x_i^p$$

with $\delta_k = \Sigma_j\ \delta_j\ w_{jk}\ x_k(1-x_k)$

Backpropagation



Backward step:
propagate errors from output to hidden layer

Forward step:
Propagate activation from input to output layer

Backpropagation Algorithm

Initialize each w_i to some small random value

Until the termination condition is met, Do

For each training example $<(x_1,\ldots x_n),t>$ Do

Input the instance $(x_1,\ldots,x_n)$ to the network and compute the network outputs y_k

For each output unit k

$\delta_k=y_k(1-y_k)(t_k-y_k)$

For each hidden unit h

$\delta_h=y_h(1-y_h)\ \Sigma_k\, w_{h,k}\ \delta_k$

For each network weight $w_{i,j}$ Do

$w_{i,j}=w_{i,j}+\Delta w_{i,j}$ where

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

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 be invoked multiple times with different initial weights)
- Often include weight *momentum* term
  - $\Delta w_{i,j}(n) = \eta\, \delta_j\, x_{i,j} + \alpha\, \Delta w_{i,j}(n-1)$
- Minimizes error training examples
  - Will it generalize well to unseen instances (over-fitting)?
- Training can be slow typical 1000-10000 iterations

  (use Levenberg-Marquardt instead of gradient descent)
- Using network after training is fast

Convergence of Backprop

- Gradient descent to some local minimum perhaps not global minimum
- Add momentum term: $\Delta w_{ki}(n)$
  - $\Delta w_{ki}(n) = a\, d_k(n)\, x_i(n) + l\, Dw_{ki}(n-1)$
    with $l \in [0,1]$
- Stochastic gradient descent
- Train multiple nets with different initial weights
- Nature of convergence
- Initialize weights near zero
- Therefore, initial networks near-linear
- Increase non-linear functions possible as training progresses

Optimization Methods

- There are other more efficient (faster convergence) optimization methods than gradient descent
  - Newton's method uses a quadratic approximation (2^{nd} order Taylor expansion)
  - $F(\mathbf{x}+D\mathbf{x}) = F(\mathbf{x}) + \nabla F(x)\, Dx + Dx\, \nabla^2 F(x)\, Dx + \ldots$
  - Conjugate gradients
  - Levenberg-Marquardt algorithm

NN: Universal Approximator?

- Kolmogorov proved that any continuous function *g(x)* defined on the unit hypercube I^n can be represented as

$$g(x) = \sum_{j=1}^{2n+1} \Xi_j \left(\sum_{i=1}^{d} \psi_{ij}(x_i) \right)$$

for *properly chosen* Ξ_j and ψ_{ij}

(A. N. Kolmogorov. *On the representation of continuous functions of several variables by superposition of continuous functions of one variable and addition.* **Doklady Akademiia Nauk SSSR**, 114(5):953-956, 1957)

Universal Approximation Property of ANN

- 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 1989]
  - Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988]

Ways to Use Weight Derivatives

- How often to update
  - after each training case?
  - after a full sweep through the training data?
- How much to update
  - Use a fixed learning rate?
  - Adapt the learning rate?
  - Add momentum?
  - Don’t use steepest descent?

MLP Summary: Terminology

- **Formulas for multilayer perceptron (MLP)**

$$Z_m = \sigma(\underbrace{\alpha_{0m} + \boldsymbol{\alpha}_m^T \boldsymbol{X}}_{C_1}), \quad m = 1, \ldots, M$$

$$f_k = g(\underbrace{\beta_{0k} + \boldsymbol{\beta}_k^T \boldsymbol{Z}}_{C_2}), \quad k = 1, \ldots, K$$

C_1, C_2 combination function

g, ς activation function

α_{0m} β_{0k} bias of hidden unit

α_{im} β_{jk} weight of connection

Fitting Neural Networks

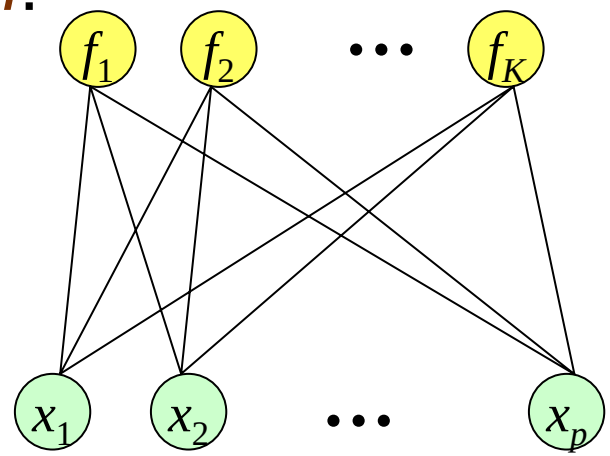
- **Simple perceptron** (no hidden layer)

$$f_k = \sigma(\alpha_{0k} + \boldsymbol{\alpha}_k^T \boldsymbol{X}) = \sigma(\alpha_{0k} + \alpha_{1k}\mathrm{X}_1 + \ldots + \alpha_{pk}\mathrm{X}_\mathrm{p}), \quad \mathrm{k} = 1 \ldots \mathrm{K}$$

- The aim: to find weights and bias
- minimizing the *error function*:

$$R = \sum_{k,s} \left(f_k^s - Y_k^s \right)^2$$

- Note: S- training set size



Algorithm: Delta Rule

- **Simple perceptron-fitting algorithm**

- Initialize weights
- Using inputs (given sample) and current weights , compute output $\delta_k = (Y_k - f_k)\sigma'(\alpha_{0k} + \boldsymbol{\alpha}_k^T \boldsymbol{X})$
- Compute delta $\Delta\alpha_{ik} = \gamma_r \delta_k X_i$
- Weights increment
- Repeat 3-4 for other samples from data set, compute total increment and update weights
- Repeat 2 until convergence

Important Remarks on Delta Rule

- Step 4 is a *learning step*, network attempts to remember given sample
- If step 5 uses all samples from data set one by one -> *batch learning,* some of them (or just one) -> *incremental learning*
- Batch learning: slow, convergence guaranteed, incremental learning: normally much faster, convergence is not guaranteed
- Example: child learning same 20 toys every day 20 days, baby learning new 2 toys 10 days (totally 20)+ 10 days for repetition

Algorithm (Backpropogation)

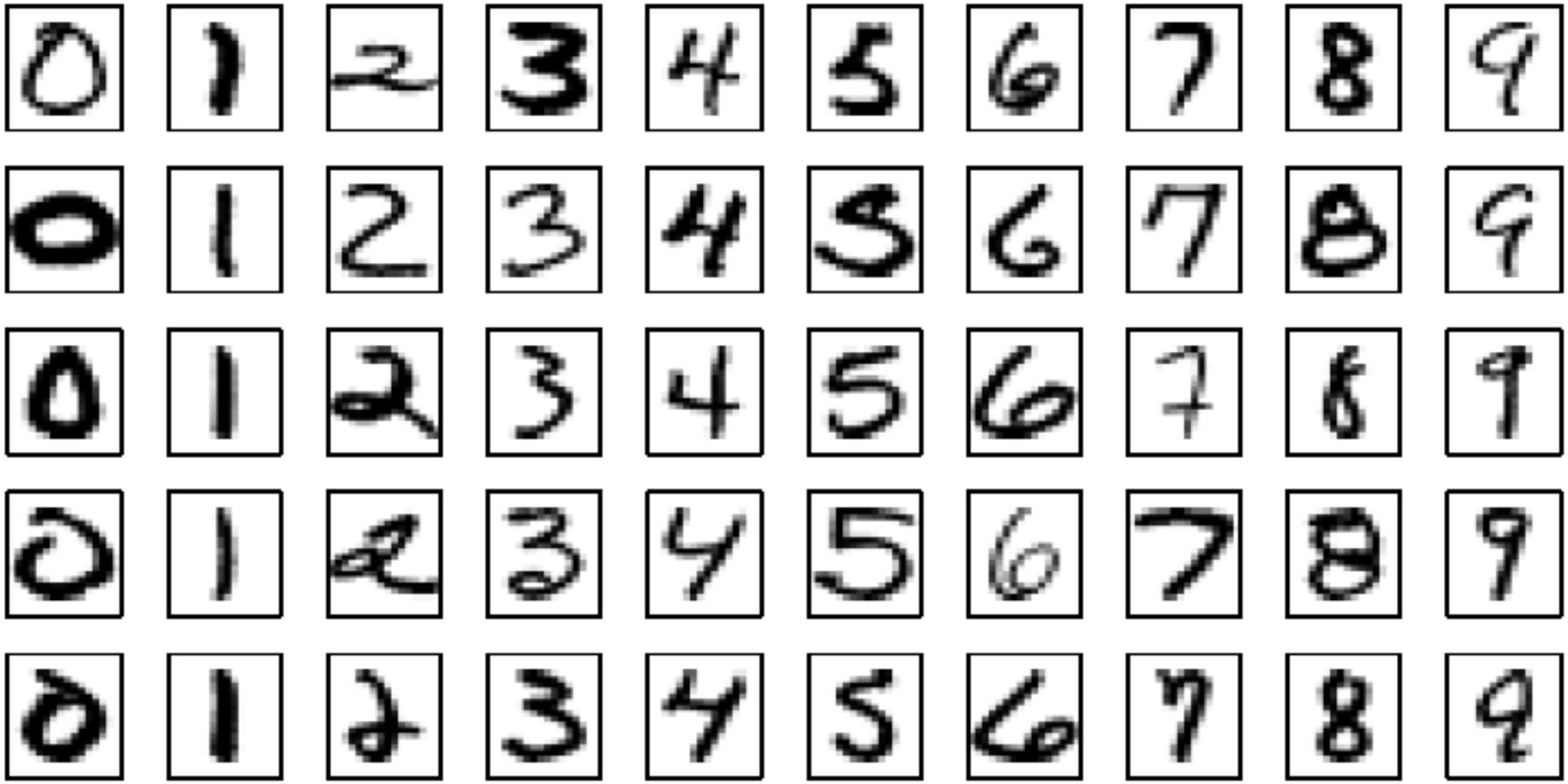
1. Initialize weights
2. Use some sample X from the data set
3. Propagate signals forward and obtain Z_i, Z'_i, f_i, f'_i
4. For each output unit compute $\delta_k = 2(f_k - Y_k) f'_k$
5. For each hidden unit compute $d_m = Z'_m \sum_k \beta_{mk} \delta_k$
6. Weights increment: $\Delta\beta_{mk} \propto -\delta_k Z_m$

   $\Delta\alpha_{jm} \propto -d_m X_j$
7. Repeat steps 2-6 for other samples, compute total increment and update weights (batch or increment)
8. Repeat from step 2 until convergence

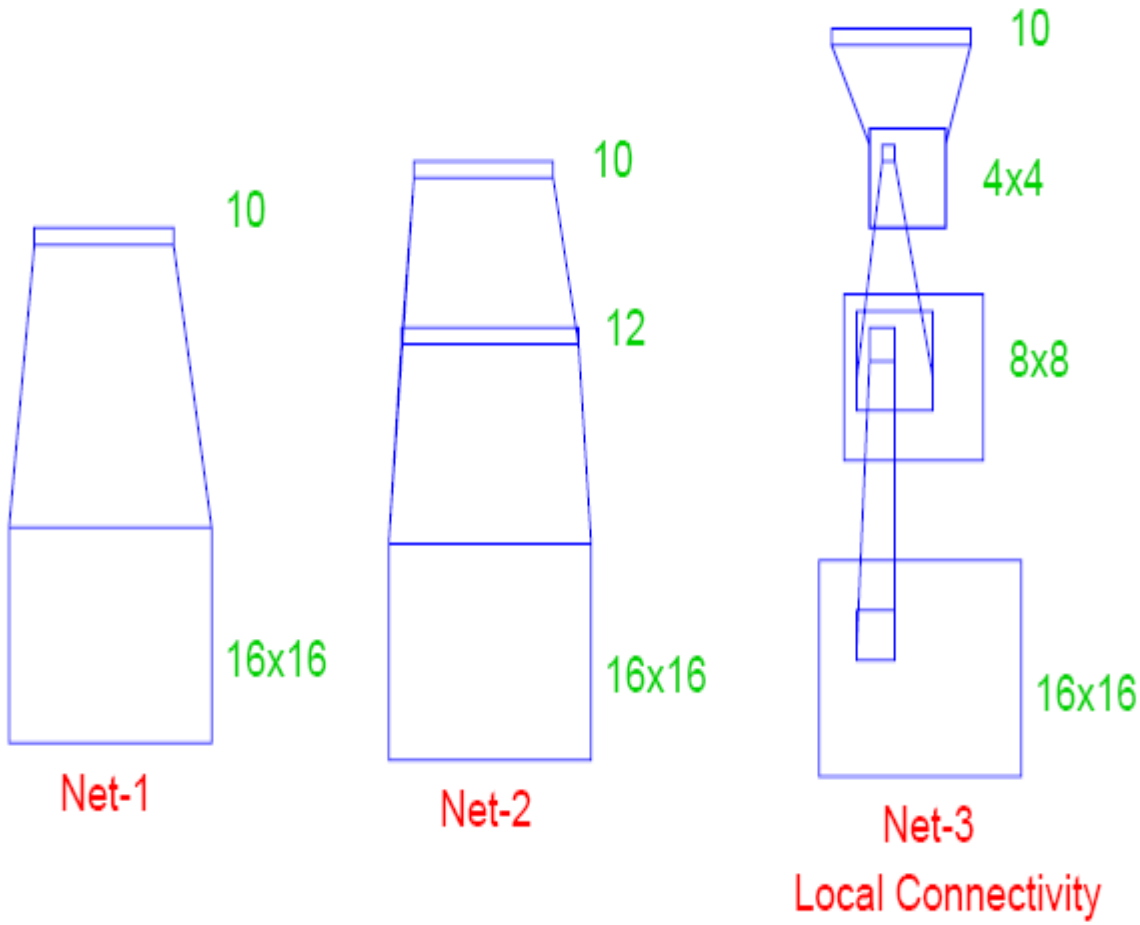
Some Remarks on Backpropagation

- Algorithm can be easily extended to the network with many hidden layers
  - Batch backpropagation (slow)
  - Incremental backpropagation(fast, no convergence guarantee)
  - QuickProp (reliable, faster 1)
  - RPROP (fast, reliable)
  - Levenberg-Marquart(small least-squares network, much memory)
  - Quasi-Newton (good for medium size)
  - Conjugate gradient (good for large size- low memory)

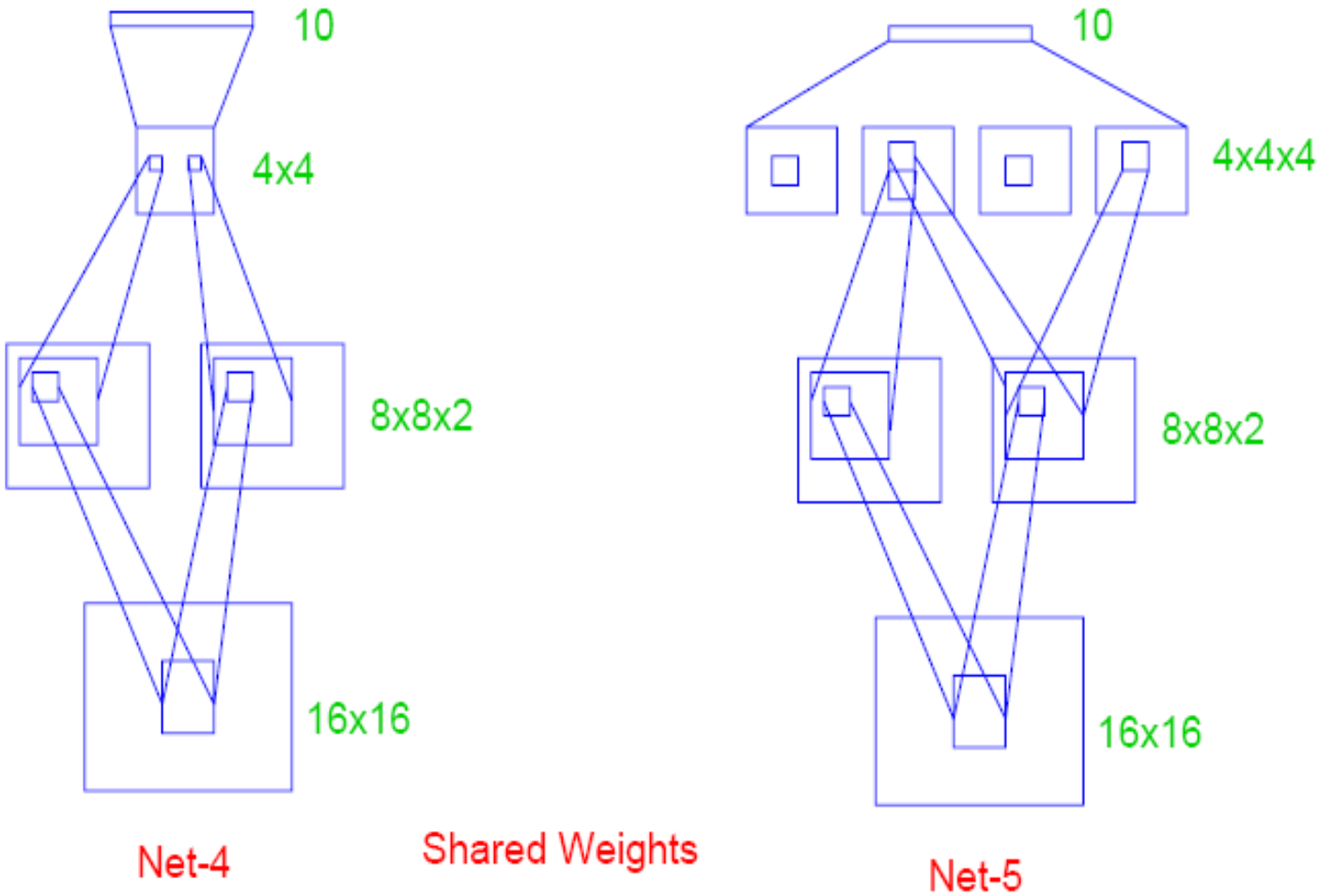
ZIP Code (Digit Image 16x16)



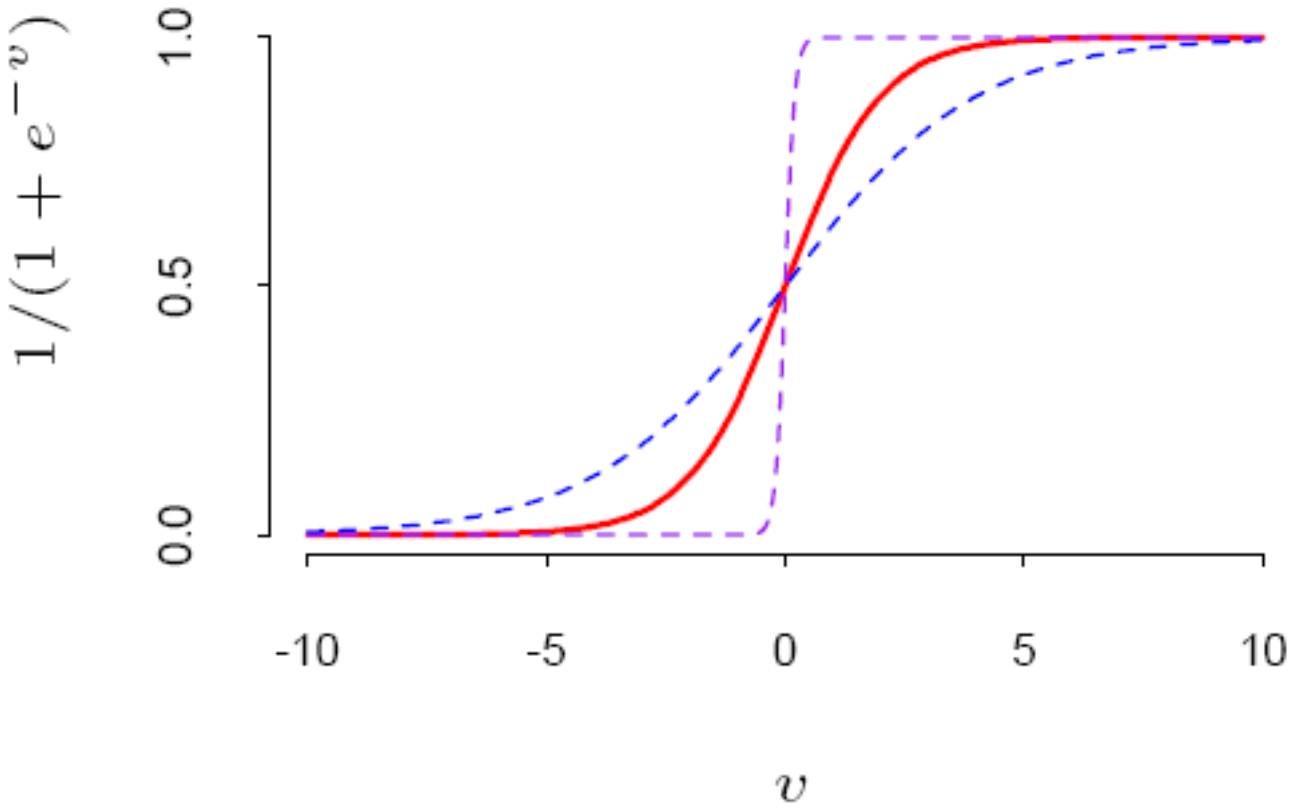
Three ANN Examples



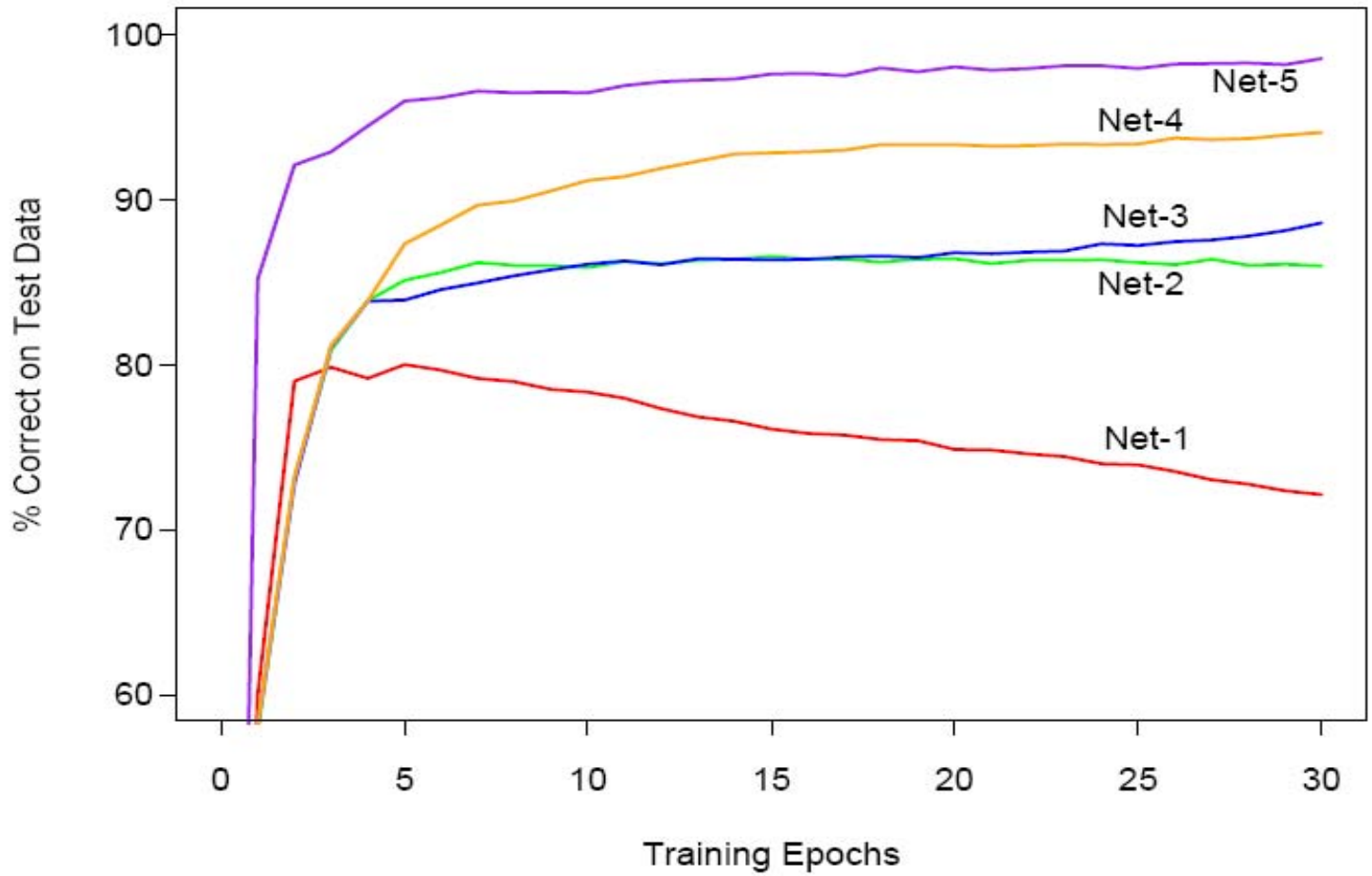
Two More ANN Examples



A Sigmoid Nonlinearity

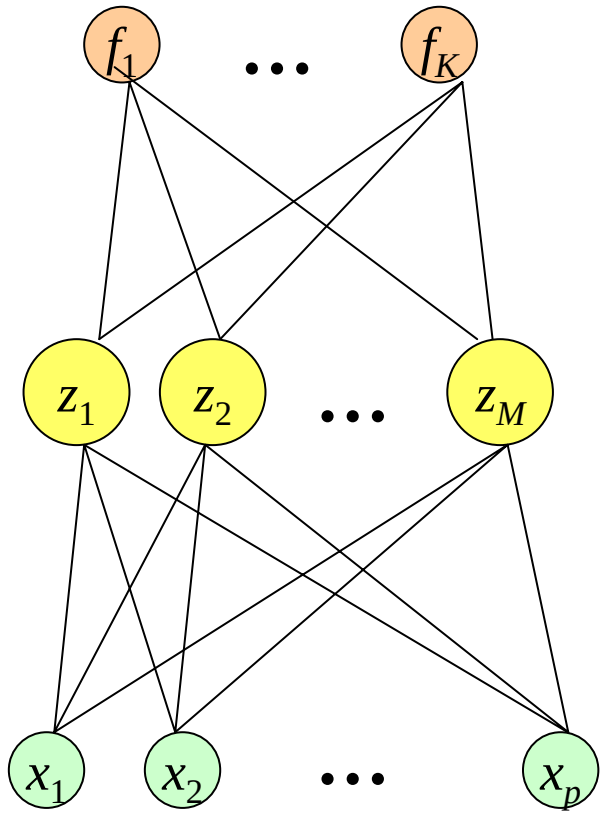


Training Iterations and Convergence



Radial Basis Function Networks

- Input, output, 1 hidden layer
- **In hidden layer**:
  - combination function=radial,
  - activation function = exponential, softmax
- **In output layer**:
  - Combination function = linear
  - Activation function = any, normally identity



Summary

- Today's Class
  - ANN (Chapter 11)
- Next Classes
  - Nonlinear method for classification (Chs 6, 9, 11, 12, 13)
- Exercises: make sure you know the topics discussed and how to do all the exercises suggested in Chapter 4
- Reading Assignments
  - HTF, Chapters 11 & 12