Neural Networks I

perceptrons

Neural Network Models

- a massively parallel distributed processor made up of simple processing units, which has a natural propensity for storing experimental knowledge and making it available for use. It resembles the brain in two respects:
- Knowledge is acquired by the network from its environment through a learning process
- Interneuron connection strengths, known as synaptic weights, are used to store the acquired knowledge

Neural Network Models

- procedure used to perform the learning process is called a *learning algorithm*
- function is to modify the synaptic weights of the network in such a way as the resulting network attains a specific objective

Benefits of Neural Networks as models of nervous systems

- **Generalization**: ability to produce reasonable outputs for inputs not encountered during training (learning)
- **Nonlinearity**: NNs are generally nonlinear in many respects thus they are able to tackle real-world problems that are often highly nonlinear in nature
- **Parallelism**: information-processing functionality is distributed in parallel across a large number of individual processing elements

Benefits of Neural Networks as models of nervous systems

- **Gradedness**: information-processing is not categorical but can represent the extent or degree to which something is true
- **Contextual processing**: highly interconnected architecture enables context to be taken into account at many levels
- **Adaptivity**: NNs have a built-in capability to adapt to changes in their environment
- **Graceful degradation**: damage to or noise in part of a network results in gradual decreases in performance rather than catastrophic failure

Perceptrons

- Rosenblatt (1961)
- single-layer feed-forward network
- used for pattern classification problems
- mapping many inputs to few outputs
- supervised learning paradigm - learning with a teaching signal

Historical Notes

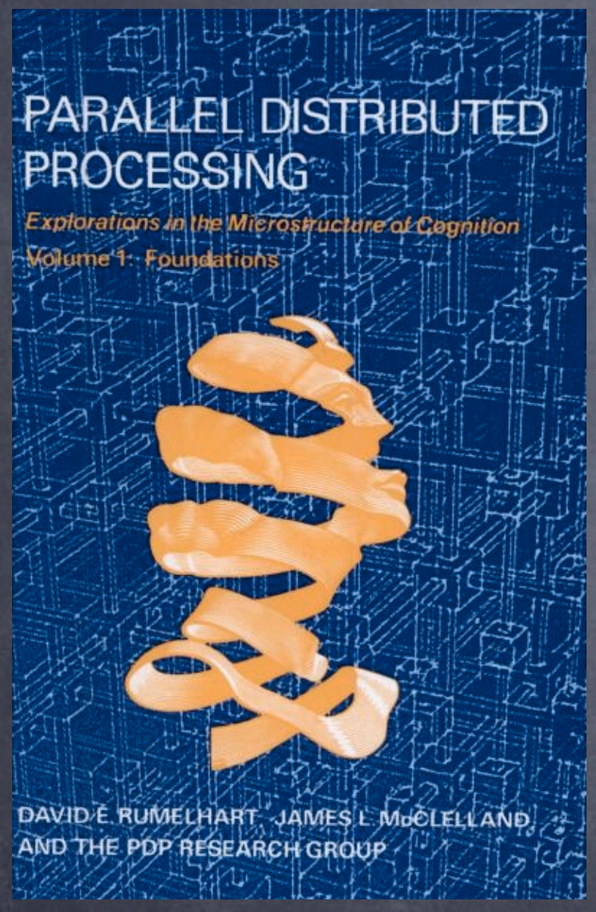
- 1943: McCulloch & Pitts (Univ. Chicago) describe a logical calculus of neural networks combining neurophysiological knowledge with mathematical logic
  - Proved: with a sufficient number of simple all-or-none units, and synaptic connections set to proper values, a network would compute *any computable function*
  - groundbreaking result... but how could a real nervous system **determine the proper synaptic connections**?

Historical Notes

- 1949: Donald Hebb (McGill Psychology) publishes *The Organization of Behavior*
- for the first time, gives us an explicit statement of a physiological learning rule for synaptic modification
- *postulate of learning*: effectiveness of a variable synapse between two neurons is increased by the repeated activation of one neuron by the other across that synapse

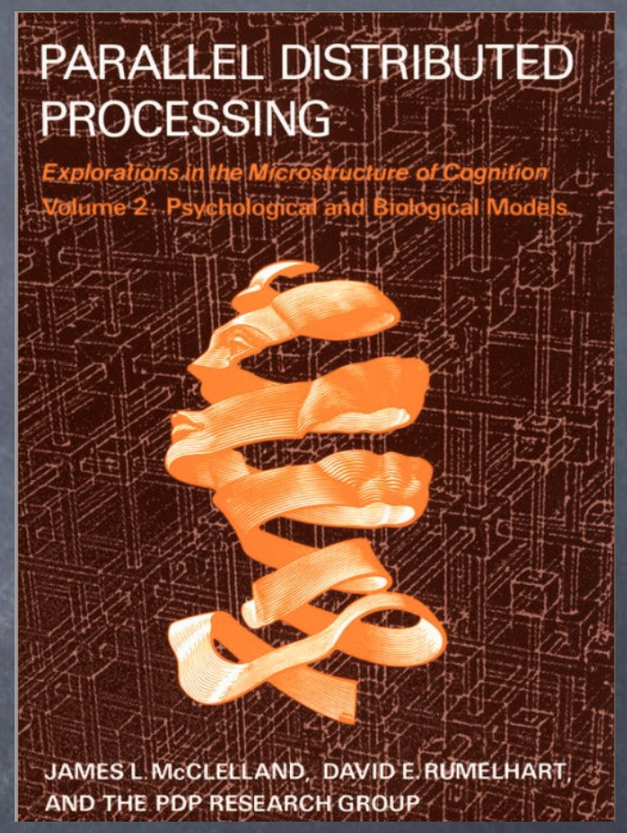
Historical Notes

- 1954: Marvin Minsky (MIT) doctoral thesis; 1961: “Steps Toward Artificial Intelligence” contains rigorous exploration of abilities of neural networks to compute through learning
- 1960: Rosenblatt’s perceptron: novel way of achieving supervised learning
- 1986: Rumelhart, Williams, & Hinton (now @ U of T): back-propagation algorithm



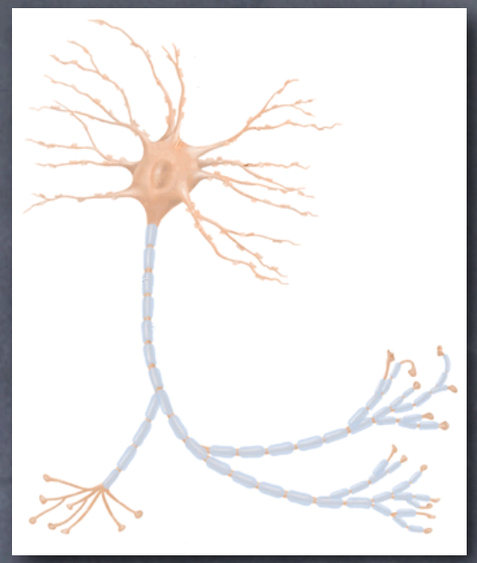
- 1986: Rumelhart & McClelland publish 2-volume “Parallel Distributed Processing - Explorations in the Microstructure of Cognition”

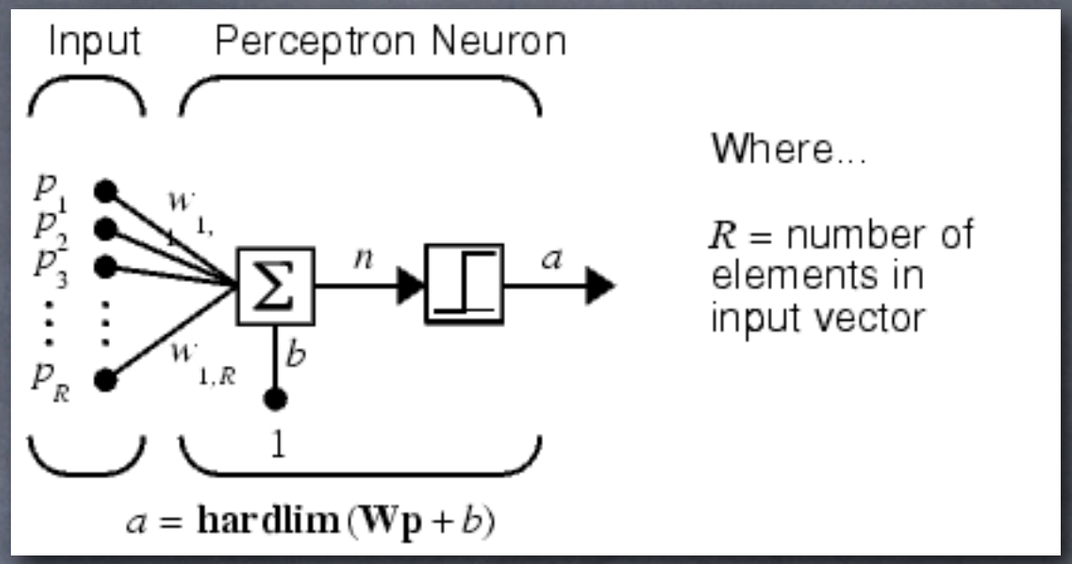
- landmark book showing power of backprop as learning algorithm for training multi-layer networks



Perceptrons

a very simple model of a neuron



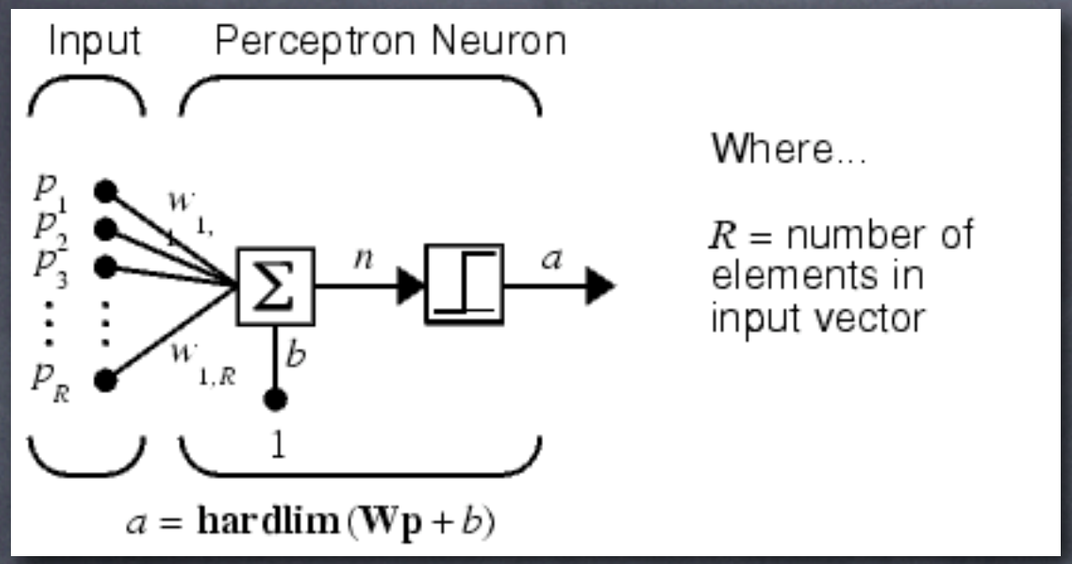


- presynaptic inputs **p** = *p1, p2, ..., pi*
- weighted by connections **W** = *w1, w2, ..., wi*
- summed at postsynaptic neuron (+ bias *b*)
- transfer function (hard limiter)
- neuron output *a*
- matrix equation: *a* = hardlim(**Wp** + *b*)
- *hardlim: input<0: output=0; input>=0: output=1*

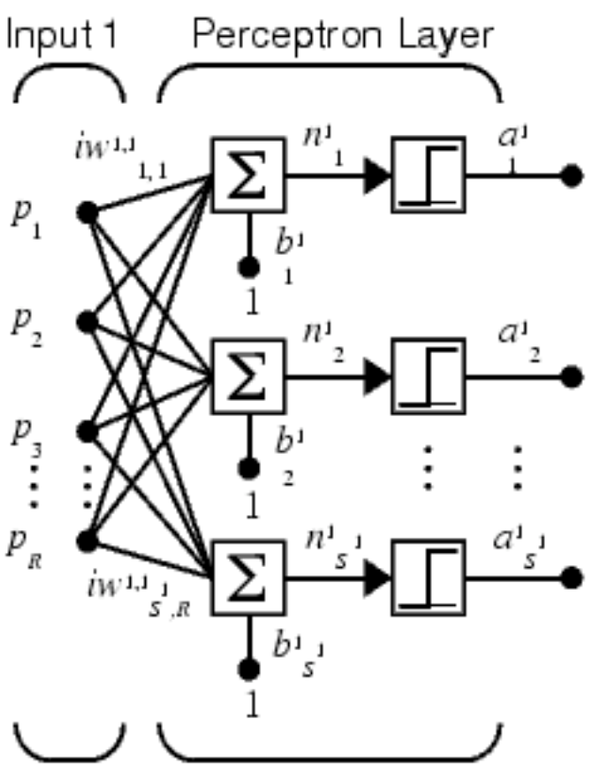
Perceptrons

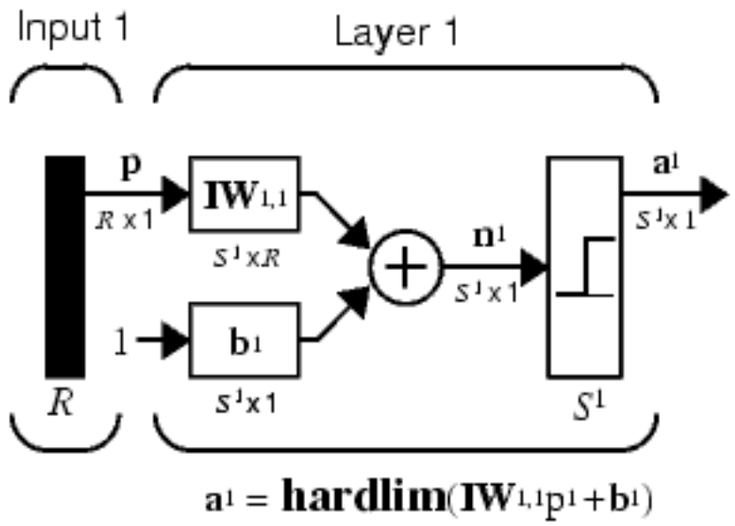
- perform mapping from one space to another
- many inputs => few outputs
- e.g. sensory -> memory
- e.g. sensory -> motor
- especially suited for pattern classification problems

Perceptrons



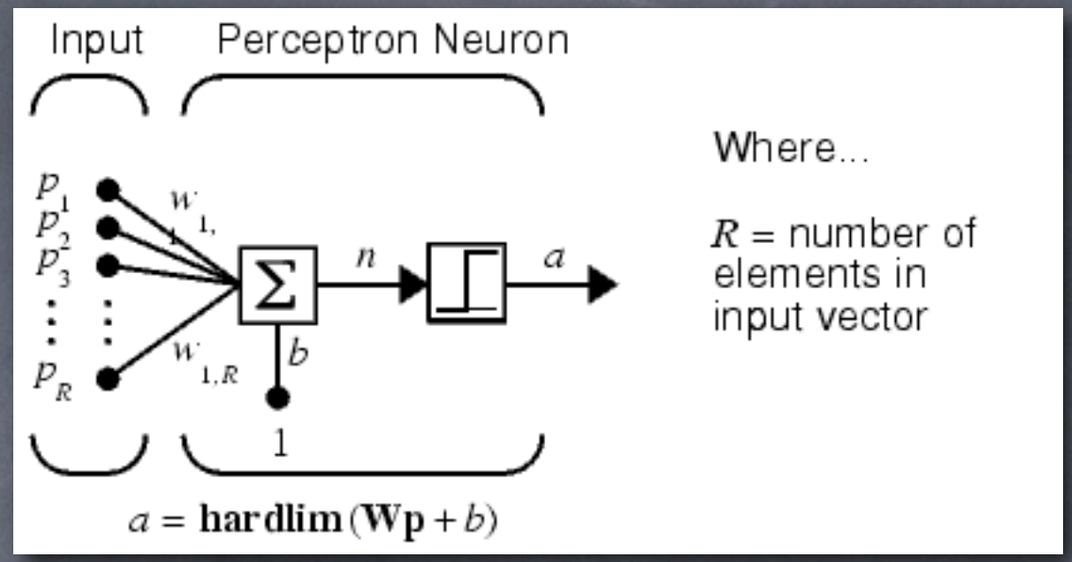
many continuous-valued inputs,
single hardlim output:
yes/no classification





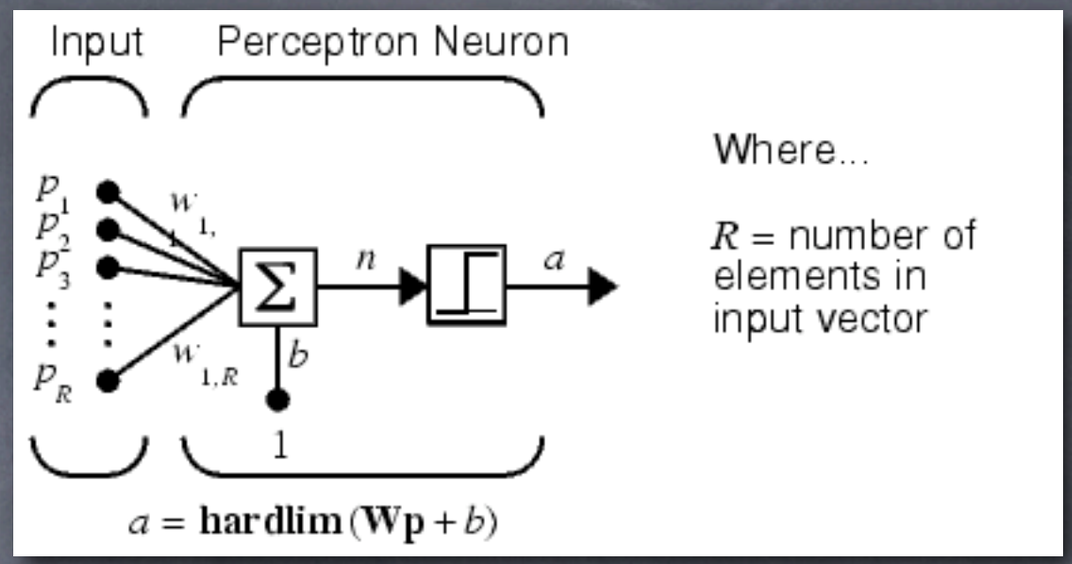
many inputs,
n outputs:
n possible
classifications

Perceptrons



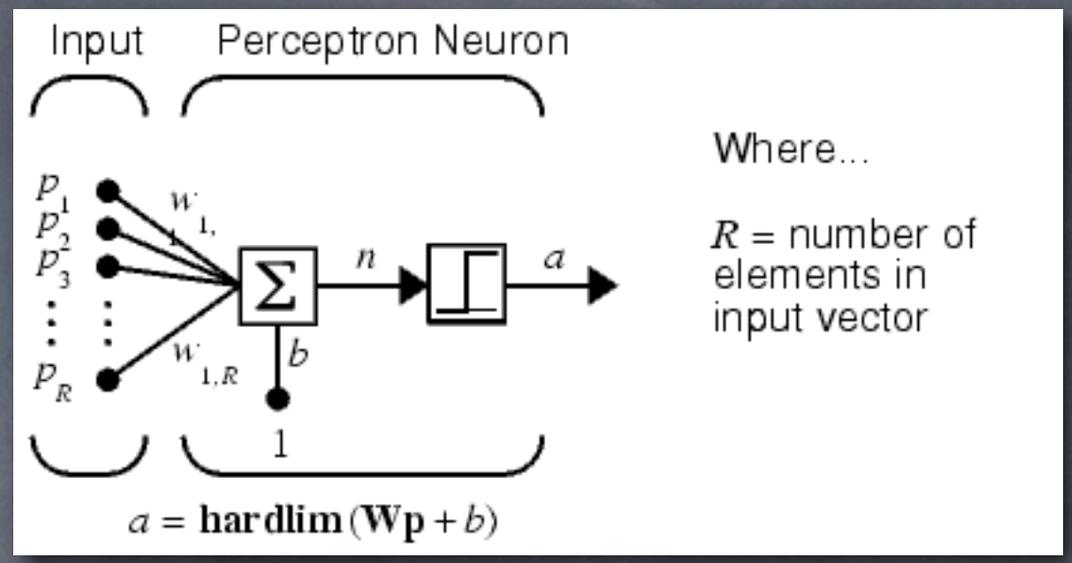
- simple example: the boolean “AND” mapping
- [0,0] -> [0]
- [0,1] -> [0]
- [1,0] -> [0]
- [1,1] -> [1]
- 4 input sets **p** = [0, 0; 0, 1; 1, 0; 1, 1]’
- 4 outputs T = [0, 0, 0, 1]
- 2 weights **W** = [w1, w2]; 1 bias b
- 1 network output *a* = hardlim(**Wp** + b)
- BUT: how to determine the correct weights **W** and bias b, so that *a* = T?

Perceptrons



- perceptron learning rule:
- a given weight should be changed in proportion to the product of the error and the input
- dW = (T-a)*$\mathbf{p}$'; dBias = (T-a)
- e.g.: $\mathbf{W}$=[0.5, 0.2]; b=[0];
- run net on input #3: p3=[1, 0]';
- a = hardlim($\mathbf{Wp}$ + b) = [0.5, 0.2]*[1, 0]' + 0 = hardlim(0.5) = 1.0
- error = (T-a) = (0 - 1.0) = -1.0;

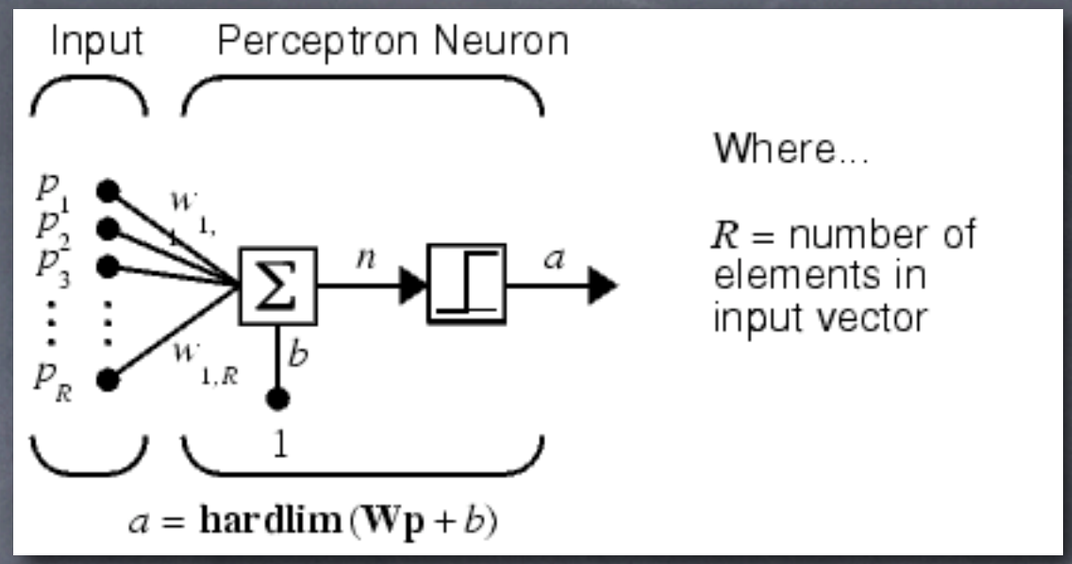
Perceptrons



- error = (T-a) = (0 - 1.0) = -1.0;
- d**W** = (T-a)*p' = [-1.0]*[1, 0] = [-1.0, 0]
- update weights: **W** = **W** + d**W**

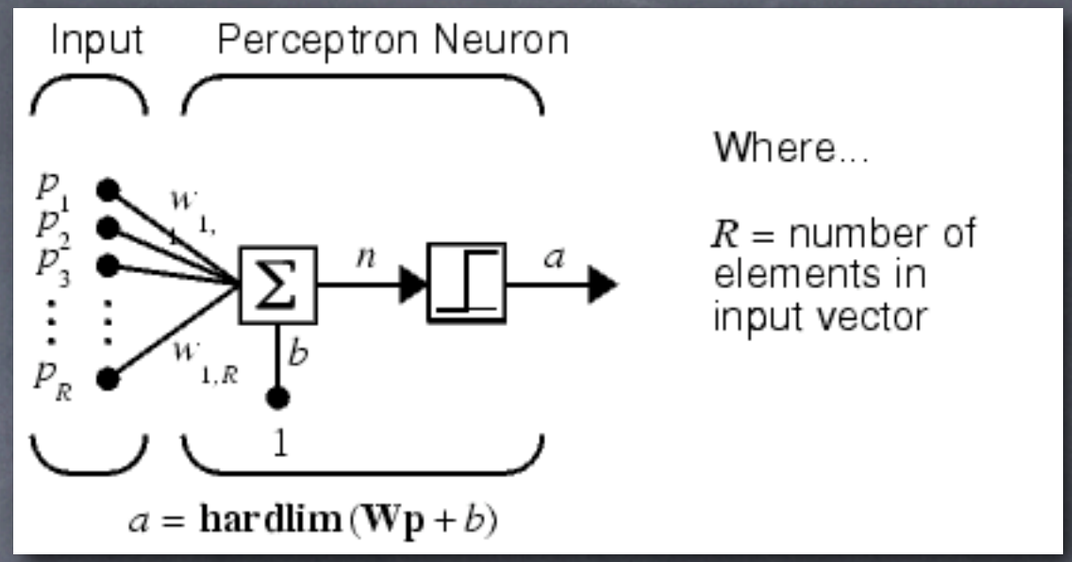
- W = [0.5, 0.2] + [-1.0, 0] = [-0.5, 0.2]
- update bias: b = b + dB
- b = [0] + (T-a) = [0] + [-1.0] = -1.0
- now re-run network: a = hardlim(**Wp** + b)
- a = hardlim([-0.5, 0.2]*[1,0]' + [-1.0]) = hardlim(-1.5) = 0.0

Perceptrons



- a = hardlim([-0.5, 0.2]*[1,0]' + [-1.0]) = hardlim(-1.5) = 0.0
- output a = 0.0; target T = 0.0; we're done!
- oops = but what about the other 3 input pairs?
- usual procedure: iterate through each training pair, each time updating weights.
- e.g. run net on {p1,T1}, compute err, update W
- run net on {p2, T2}, compute err, update W
- repeat for {p3, T3}, {p4, T4}, and then repeat whole thing until error is zero for all 4 pairs

Perceptrons

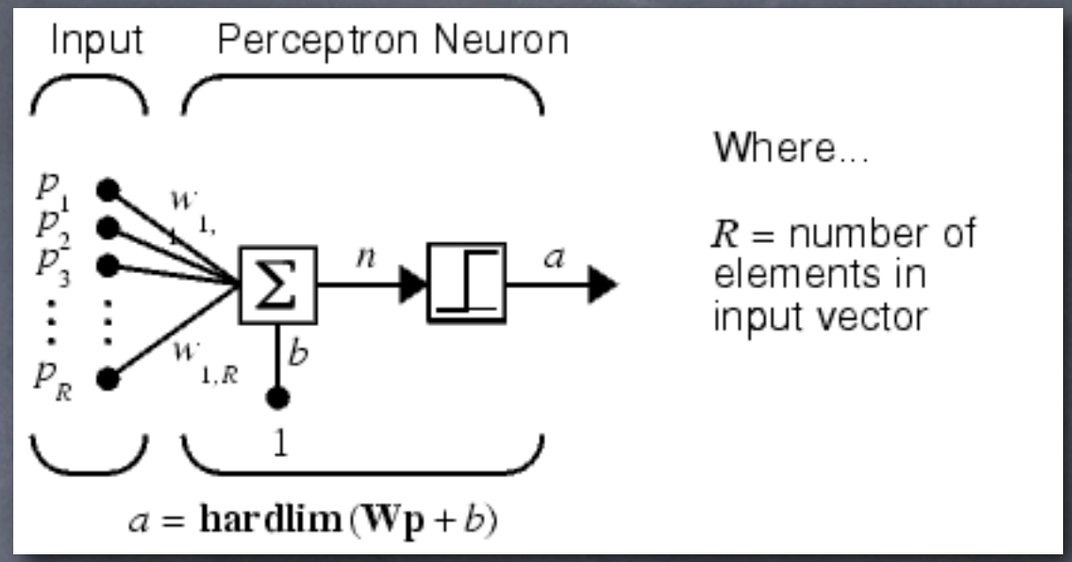


- perceptron convergence theorem:
- perceptron learning rule is **guaranteed** to converge on a solution!
- (eventually)
- *caveats:*
  - *the underlying problem has to be "linearly separable"*
  - *"eventually" can be a long time*

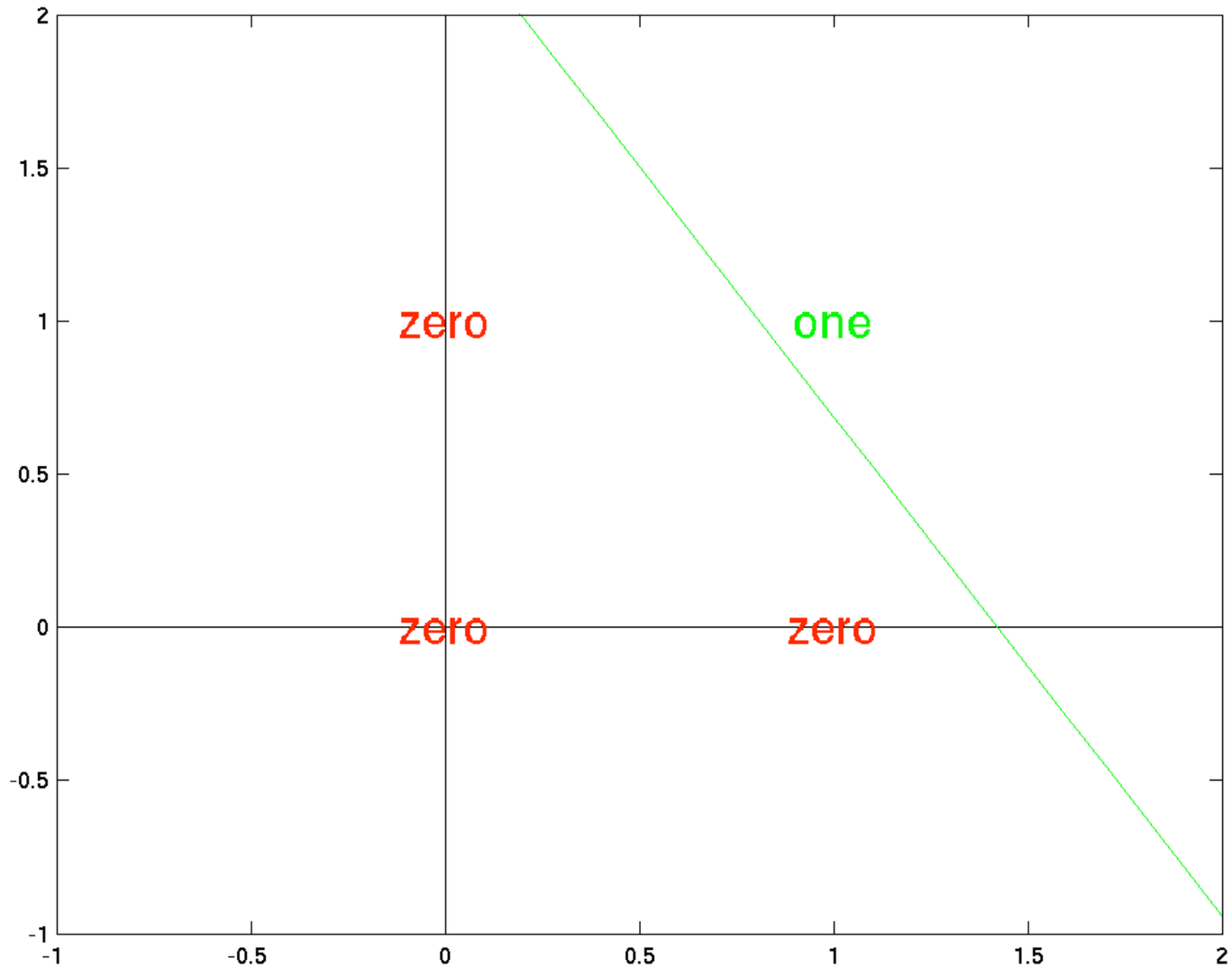
Perceptrons

- matlab demo perceptron1.m using boolean "AND" training set

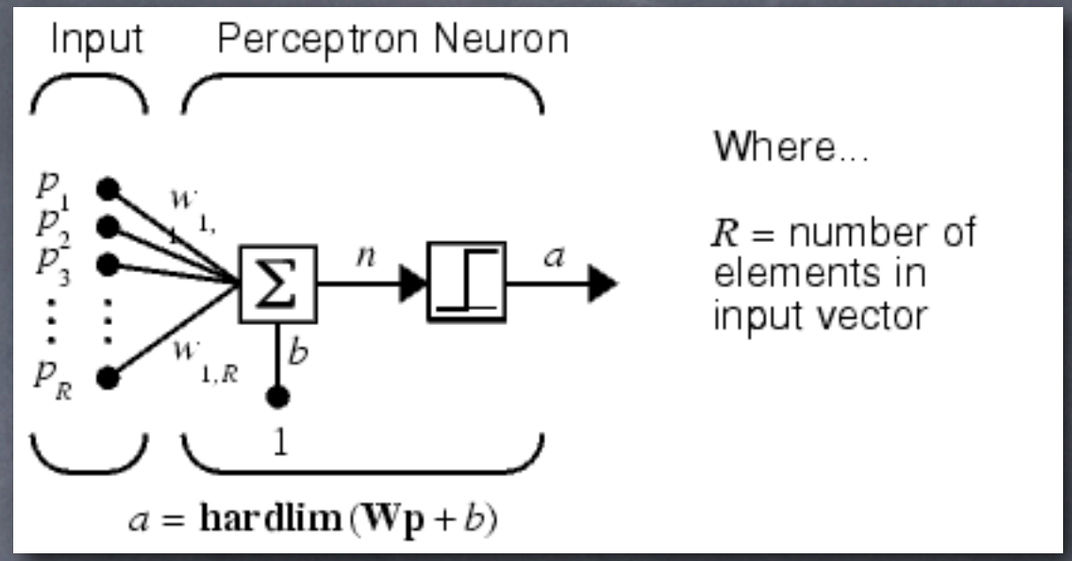
Perceptrons



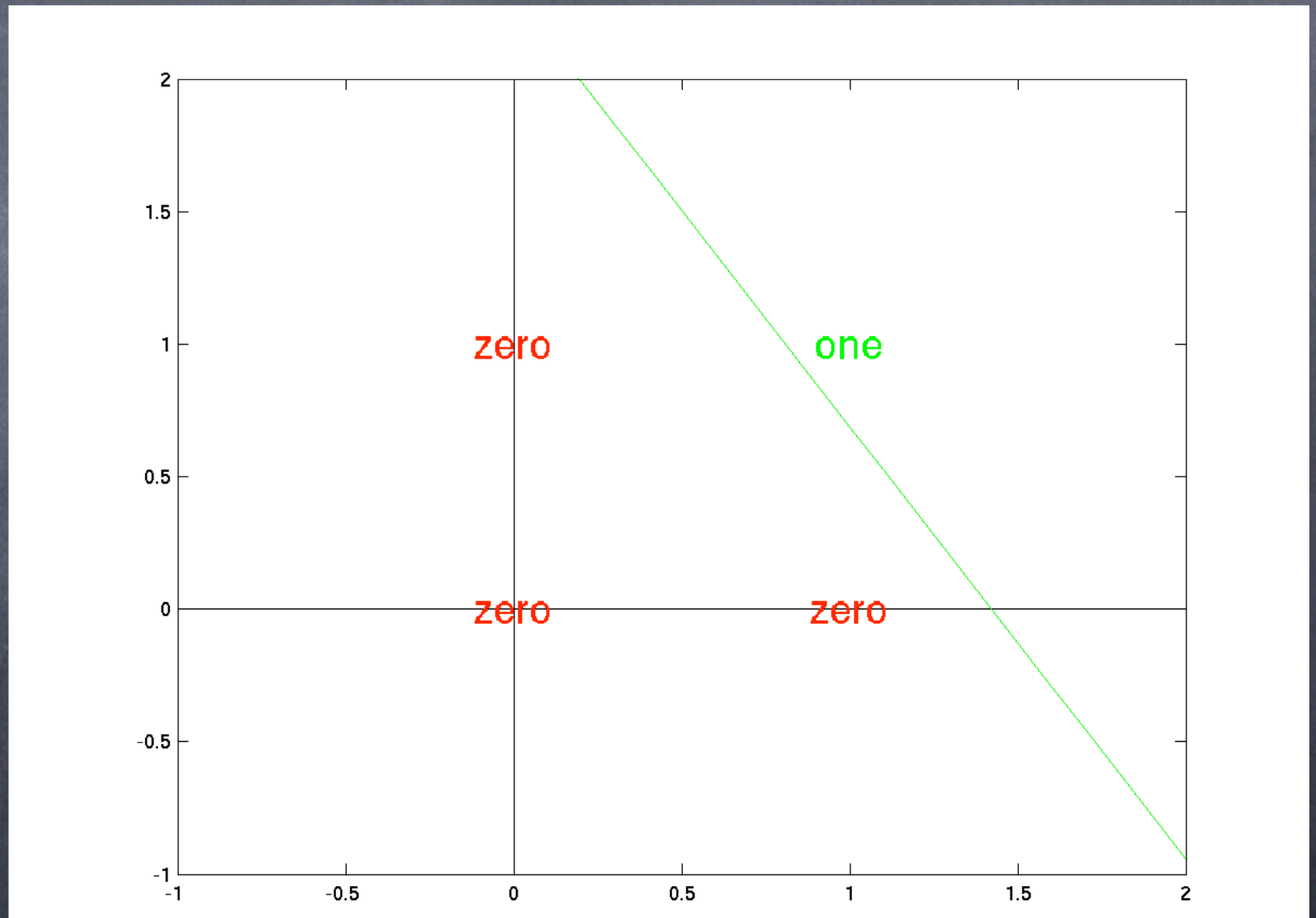
- weights **W** and bias b can be regarded as a "**decision boundary**" in input space
- **W** = [1.9, 1.2]; b=[-2.7];



Perceptrons

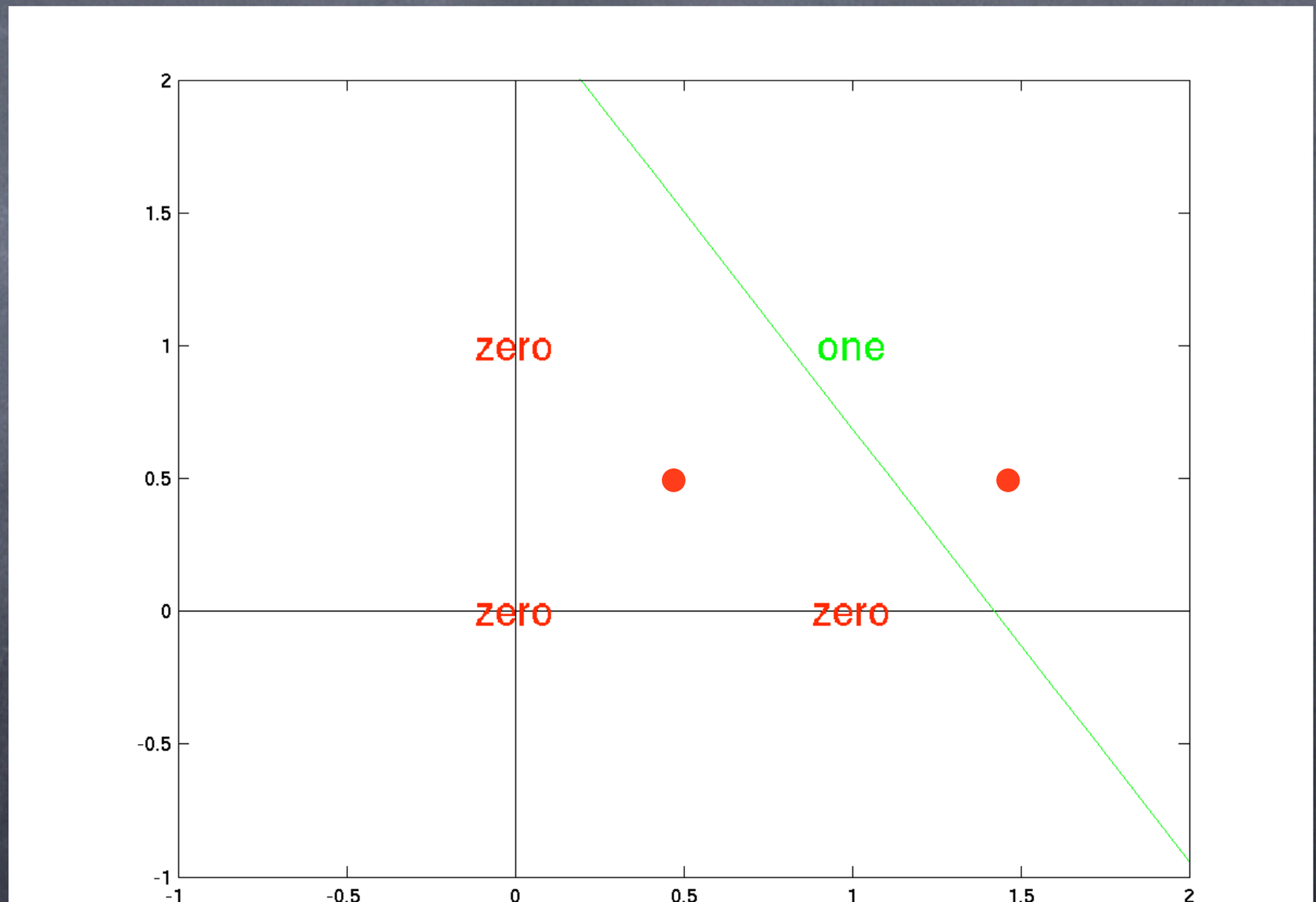


- boolean “AND” problem IS *linearly separable*
- i.e. we can draw a straight line that discriminates between the two output classes {0; 1}



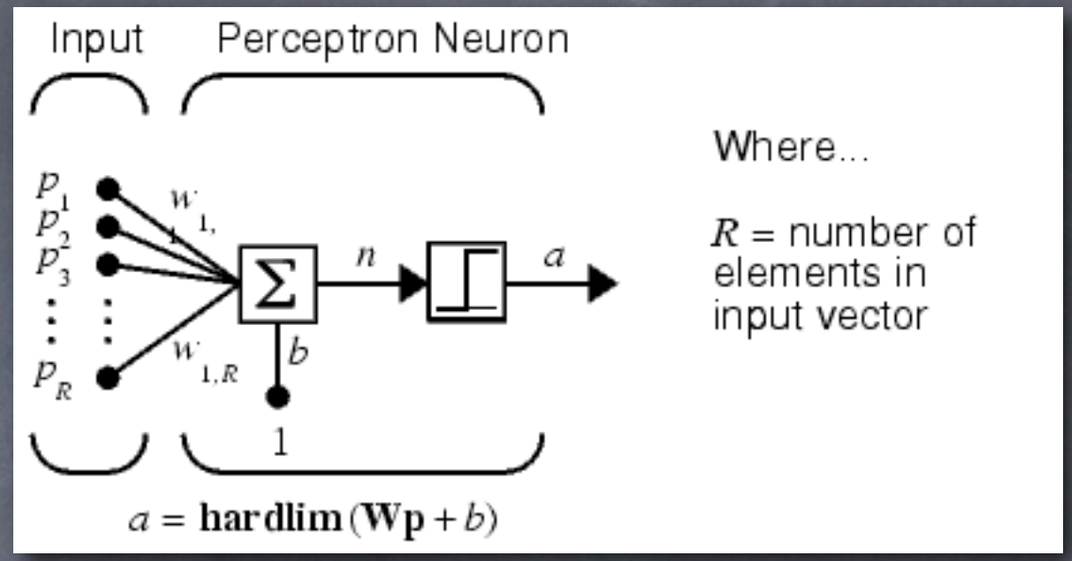
Perceptrons

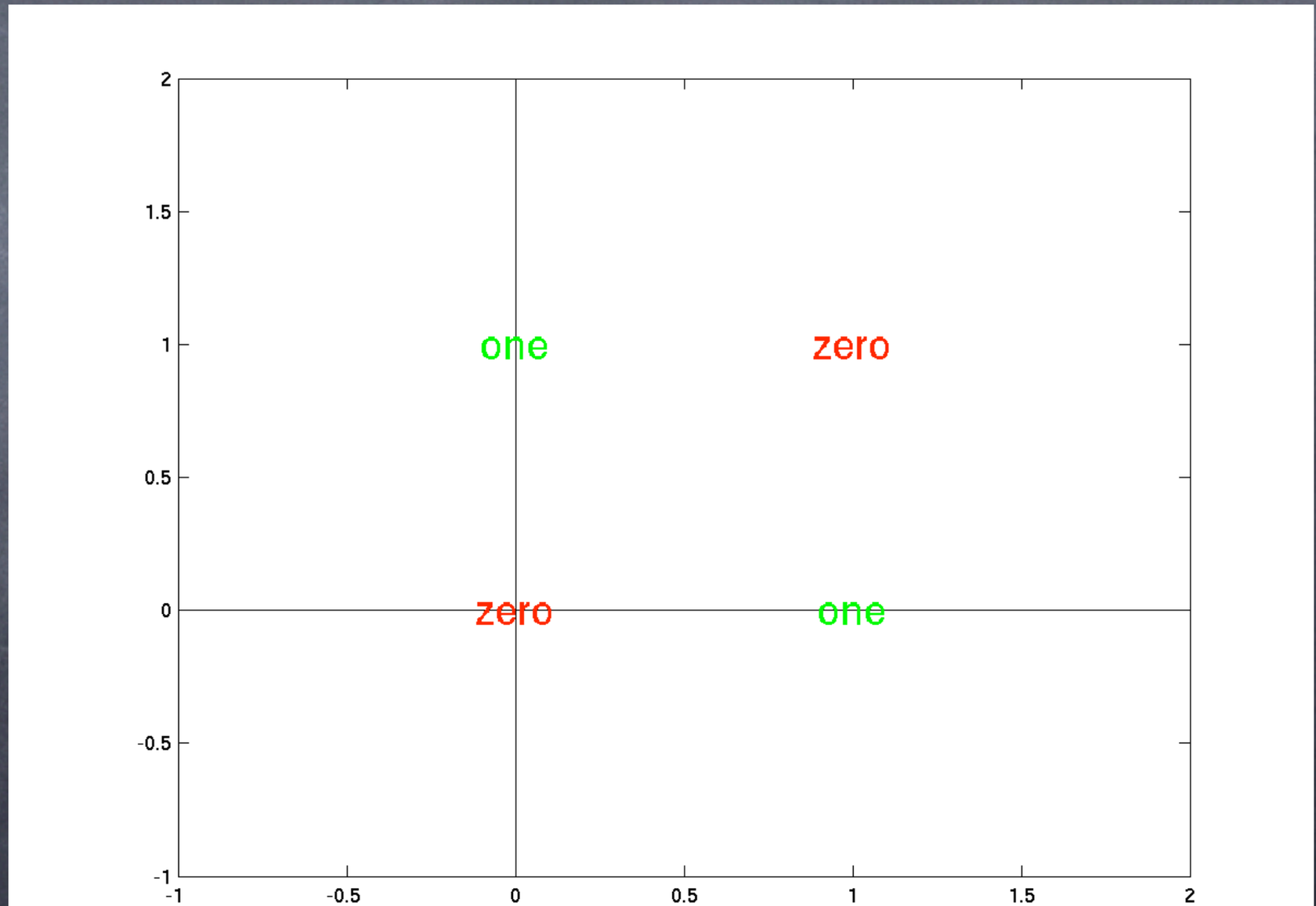
- matlab demo perceptron1.m
- try inputs not in the training set
- e.g. p=[.5, .5]
- or p=[1.5, .5]
- illustrates property of *generalization*



Perceptrons

- what about boolean “XOR” (exclusive-or)?
- [0,0] -> [0]
- [0,1] -> [1]
- [1,0] -> [1]
- [1,1] -> [0]
- is XOR linearly separable?
- (no)



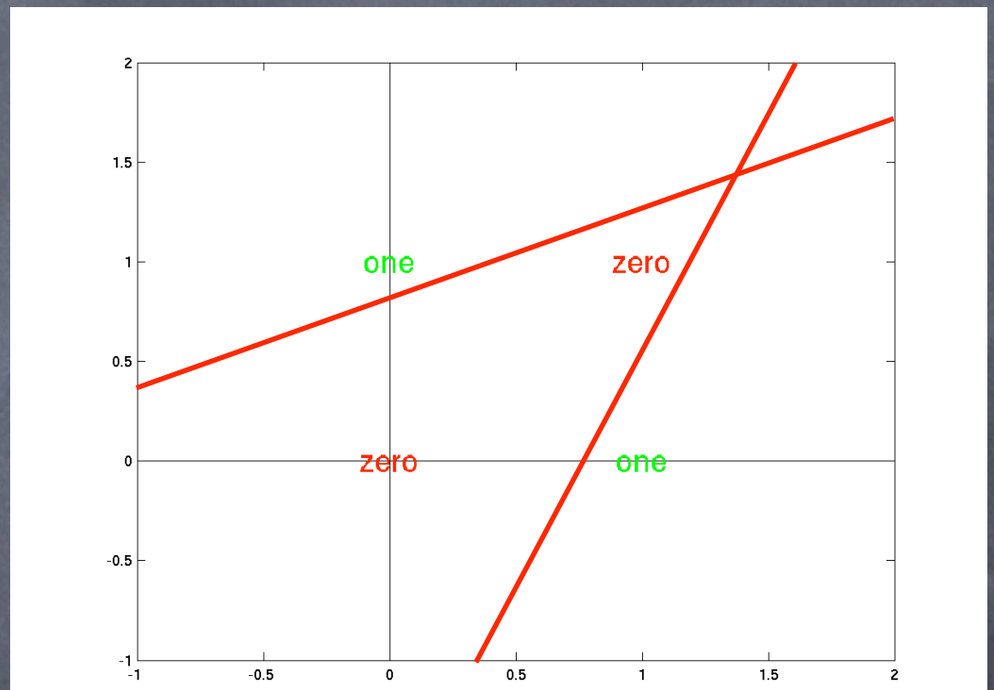


Perceptrons

- matlab demo perceptron1.m using boolean "XOR" training set

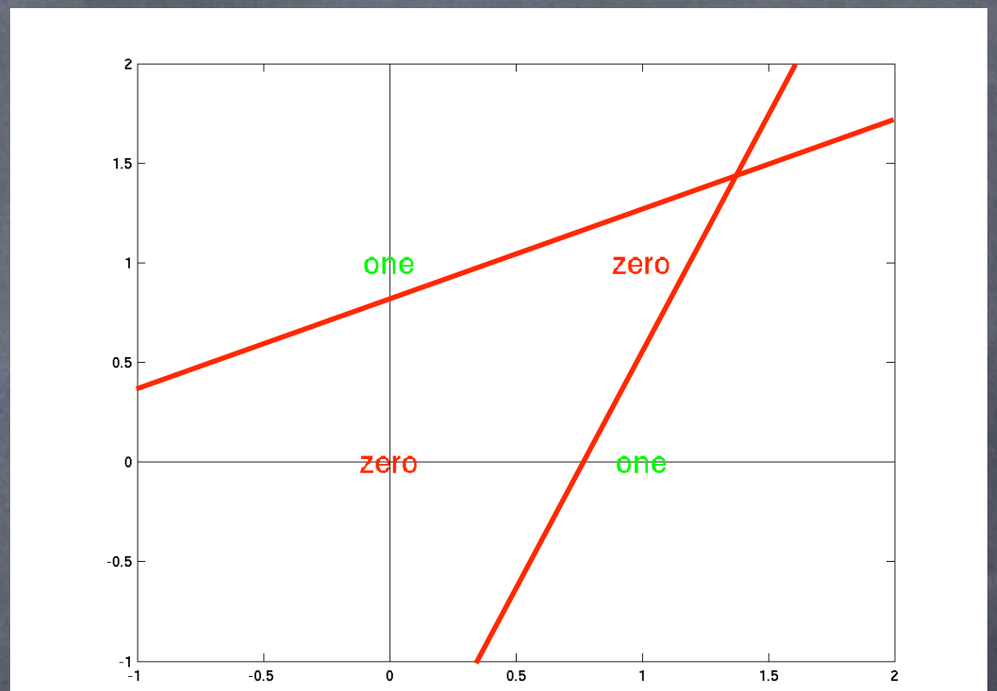
Perceptrons

- only way to learn a problem like XOR that is not linearly separable is to use a multi-layer network
- multiple layers provide the ability for **piecewise linear** decision boundaries



Perceptrons

- we will look at multi-layer nets later
- now: let's look at single-layer perceptrons with more than one neuron

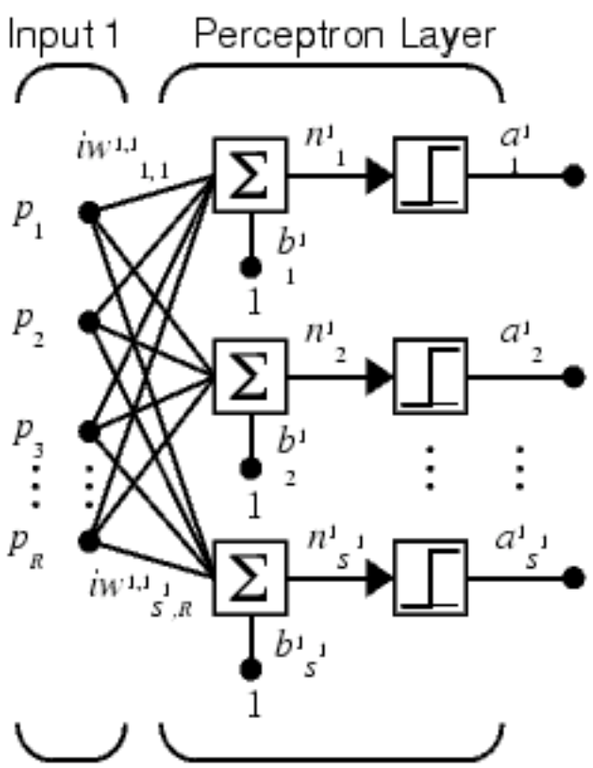


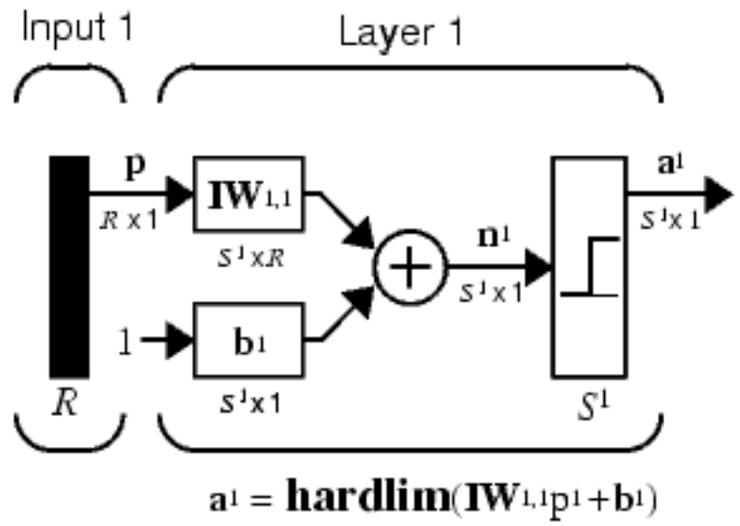
Perceptrons

- matlab demo perceptron2.m
- single-layer perceptron with 26 neurons
- we will train it to do optical character recognition

Perceptrons

- matlab demo perceptron2.m
- single-layer perceptron with 26 neurons
- we will train it to do optical character recognition

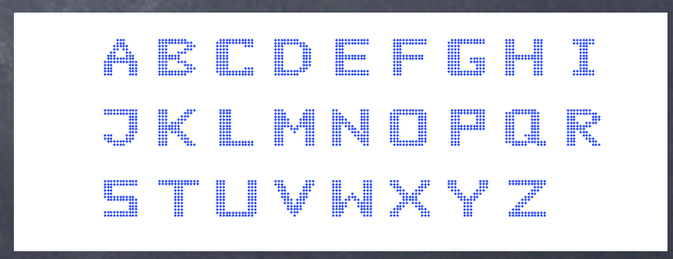




Where...

R = number of elements in Input

S^1 = number of neurons in layer 1



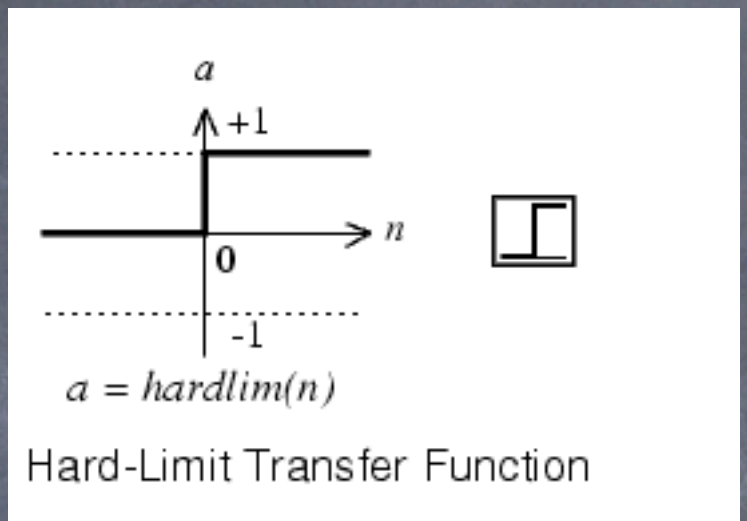
Neural Networks II

multi-layer feedforward nets
backprop

Multilayer feedforward networks

- single-layer perceptrons can only succeed for linearly separable problem spaces
- multi-layer nets can do non-linear classification
- more generally, multi-layer NNs can perform any arbitrary mapping, given enough neurons and layers (more on this later)

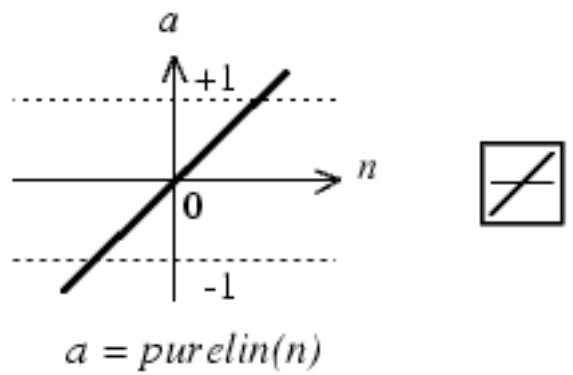
Transfer Functions



Hard-Limit Transfer Function

- we looked at perceptrons with hardlim (boolean) transfer functions; output of NNet can only be boolean {0,1}

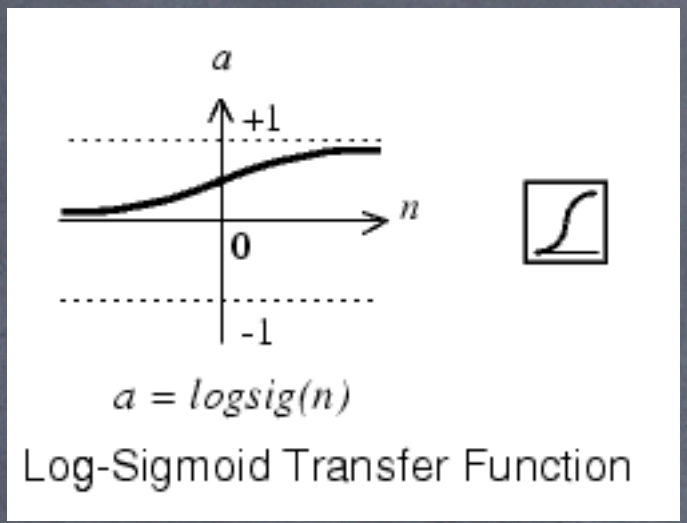
Transfer Functions



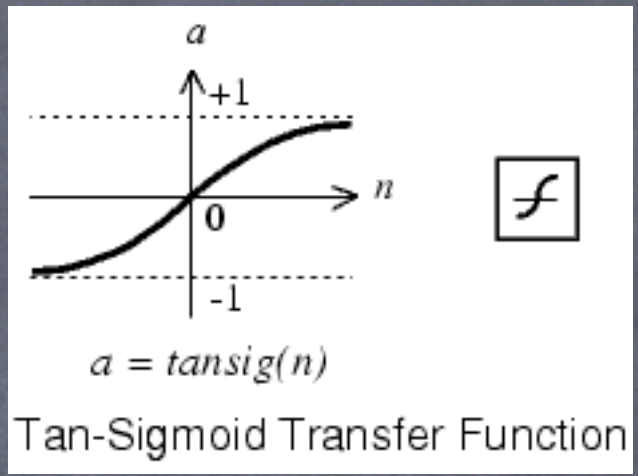
Linear Transfer Function

- purelin (linear): output of neuron can take on any value

Transfer Functions



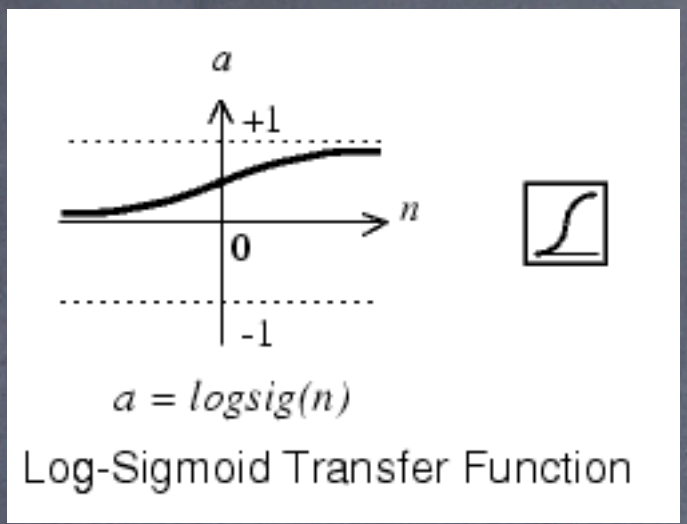
Log-Sigmoid Transfer Function



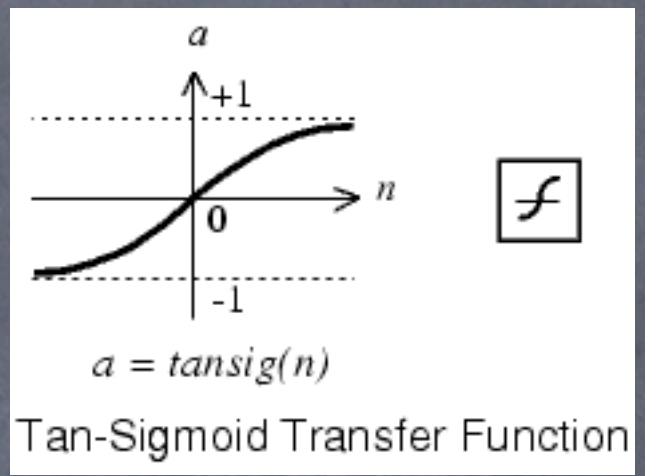
Tan-Sigmoid Transfer Function

- logsig: output is linear near mid-range, but asymptotic in minimum and maximum values [0,1]
- tansig: similar to logsig but constrained to [-1,1]

Transfer Functions



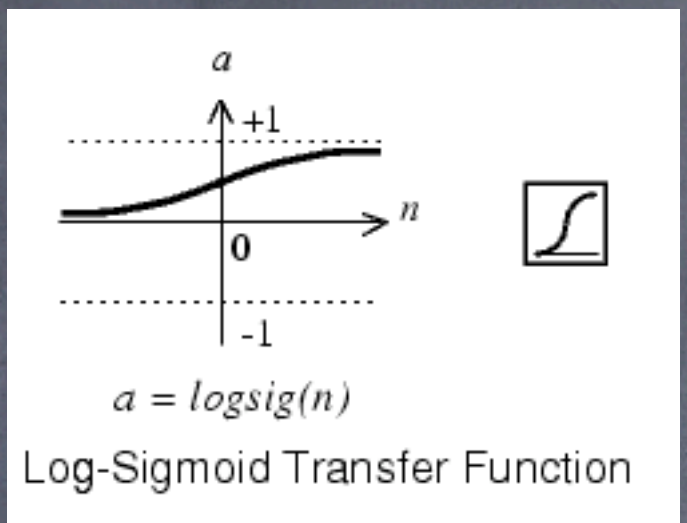
Log-Sigmoid Transfer Function



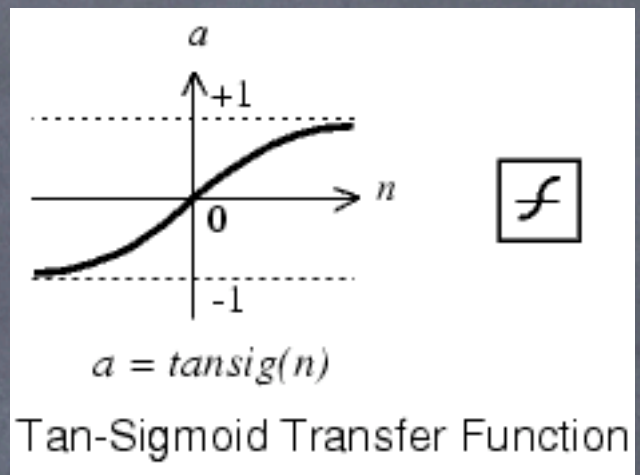
Tan-Sigmoid Transfer Function

- advantages of logsig / tansig:
- nonlinearity helps NNets to approximate any non-linear function / mapping
- nonlinear TFs “resemble” saturating properties of biological neurons

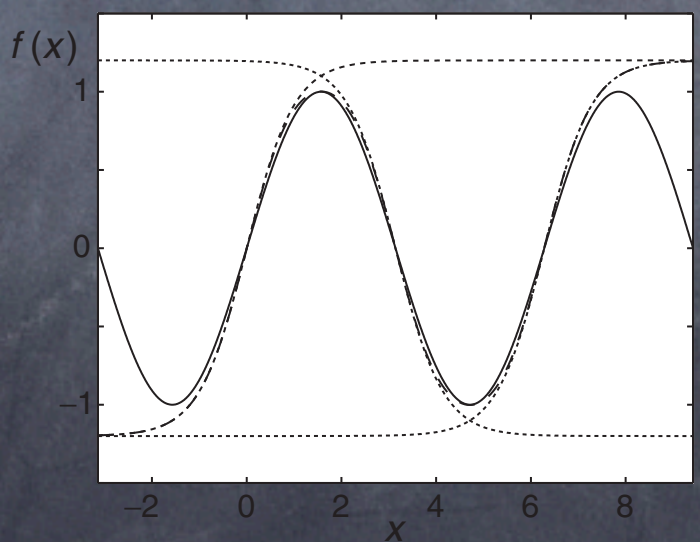
Transfer Functions



Log-Sigmoid Transfer Function

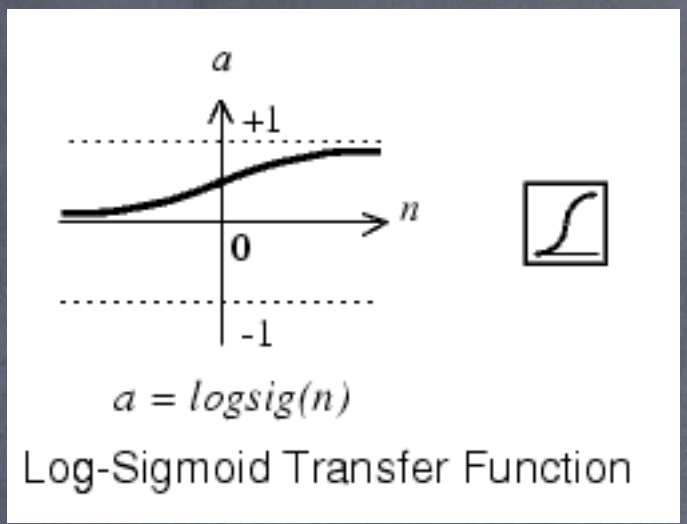


Tan-Sigmoid Transfer Function

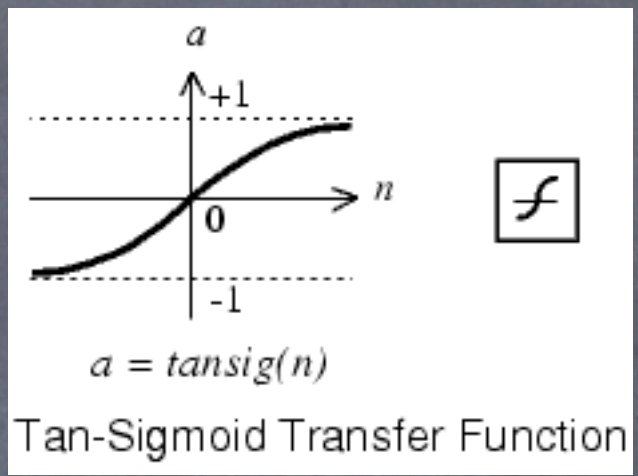


- e.g. approx. of sine function using sum of 3 sigmoids

Transfer Functions



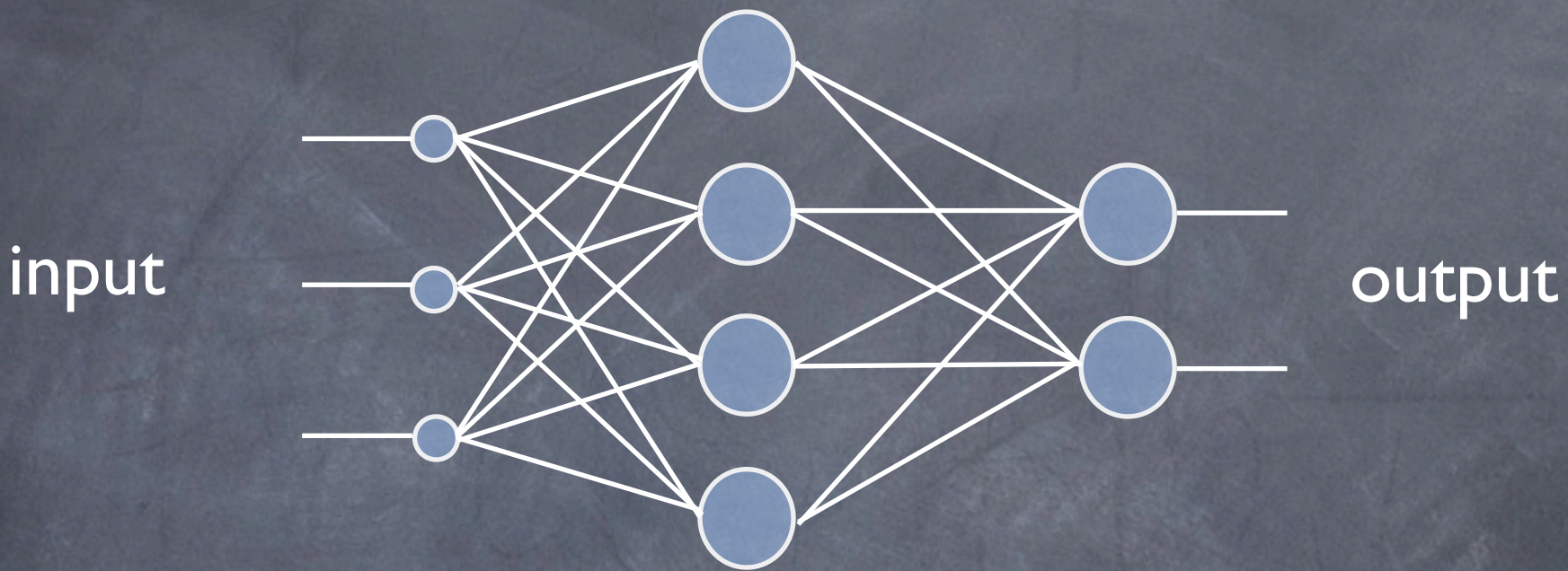
Log-Sigmoid Transfer Function



Tan-Sigmoid Transfer Function

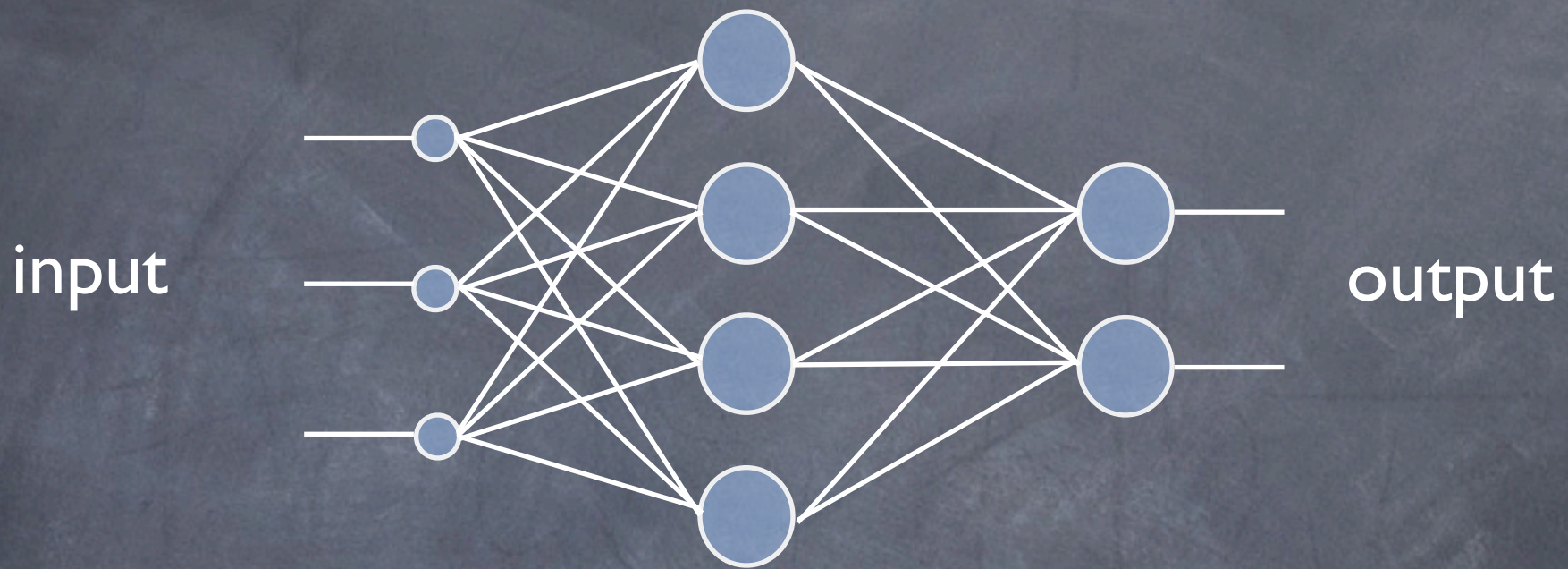
- choice of transfer function is important
- will determine which values your NN is capable of producing
- “typical” (if there is such a thing): use logsig or tansig for hidden layers and purelin for output layer

Multi-layer feedforward NNets



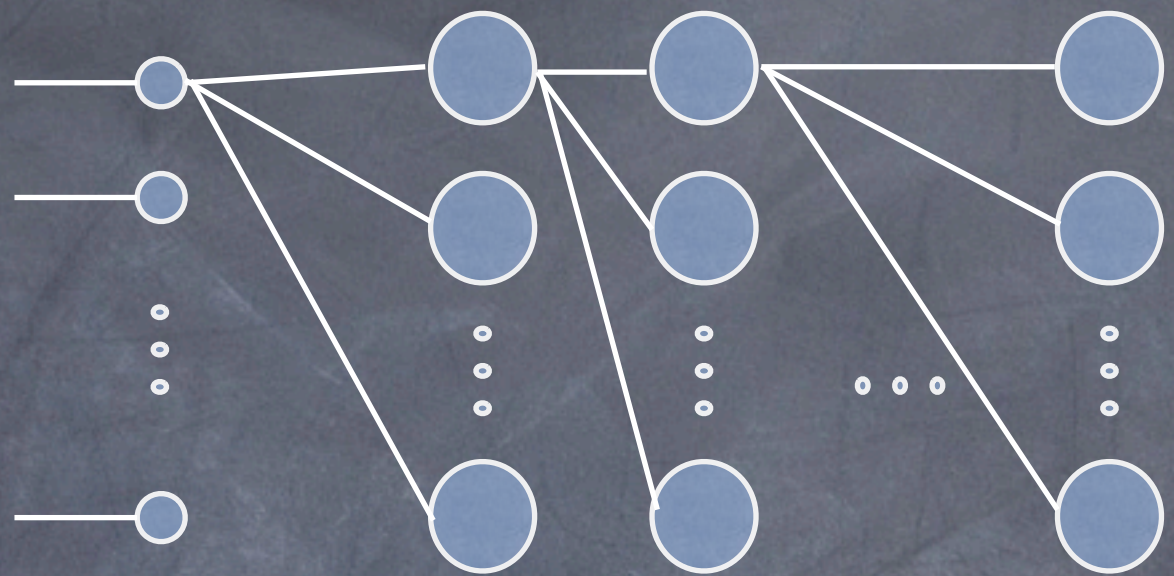
- multi-layer nets have 1 input layer, 1 output layer and at least one hidden layer
- any pattern of connectivity is possible (e.g. fully connected)
- between each level the usual computations happen
- out = Tf(Wp + b); where Tf is transfer function, W weight matrix, b bias
- e.g. Tf=tansig for hidden layer(s), Tf=purelin for output layer

Multi-layer feedforward NNets



- # weights = in*hid + hid*out; # bias = hid+out
- # weights can get very large very easily!

Multi-layer feedforward NNets

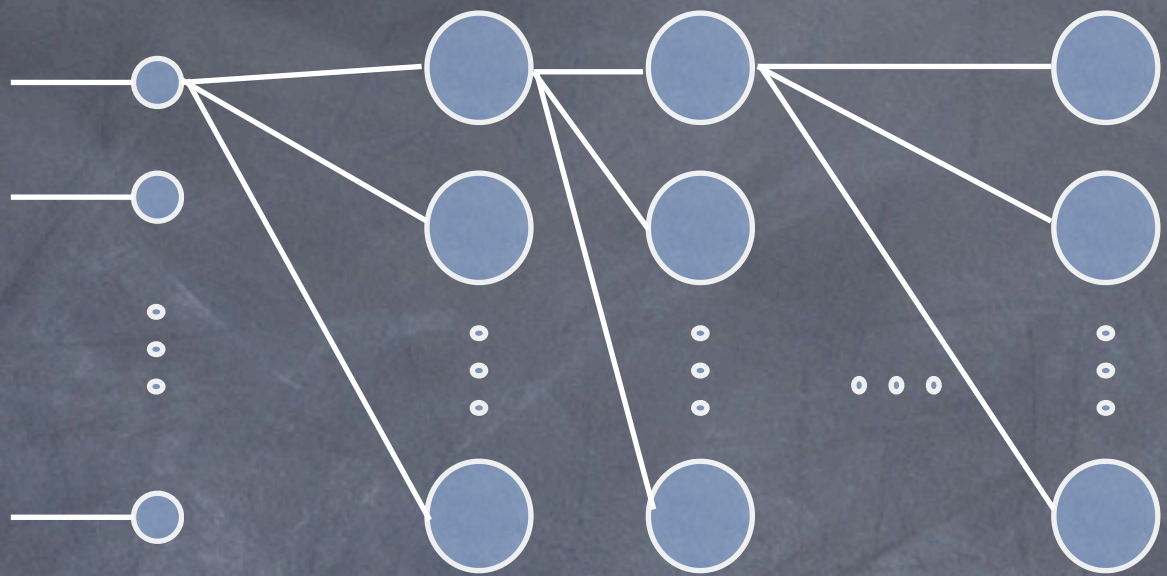


- in general, can have any number of neurons in each layer, and any number of hidden layers
- as # neurons and # layers increase, computational power increases
- can regard NN as a system of nonlinear equations in which outputs are constructed from a series of weighted sums of inputs (passed through Tf)
- how to come up with appropriate weights?
- # weights >> # inputs & # outputs ; mathematically ill-posed problem i.e. no unique solution

Multi-layer feedforward NNets

- curse of dimensionality
- error surface (manifold): n-dimensional space where n=# free parameters in network
- if error surface is complex, we need a dense sample of data to span it, to enable the network to learn it well
- difficult to get dense enough sampling with high dimensional space!
- exponential growth in complexity with increases in dimensionality => in turn leads to deterioration of space-filling properties of uniformly randomly distributed points (e.g. data)
- some solutions: “regularization” - mathematical procedure to introduce prior knowledge about error manifold (e.g. smoothness) - see ch. 5, Haykin

Multi-layer feedforward NNets



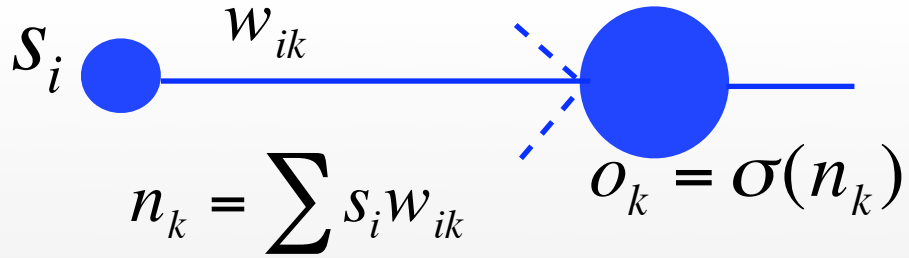
- Rumelhart & McClelland (1986) book popularized an algorithm for training multi-layer NNs by adjusting weights in an iterative fashion: "back-propagation algorithm"
- back-prop is similar in spirit to the perceptron training rule
- d**W** is proportional to error and activation
- back-prop was landmark achievement
- first time a computationally efficient method of training large NNs
- not necessarily optimal for every problem however generally able to converge on a solution for most problems

Back-Prop

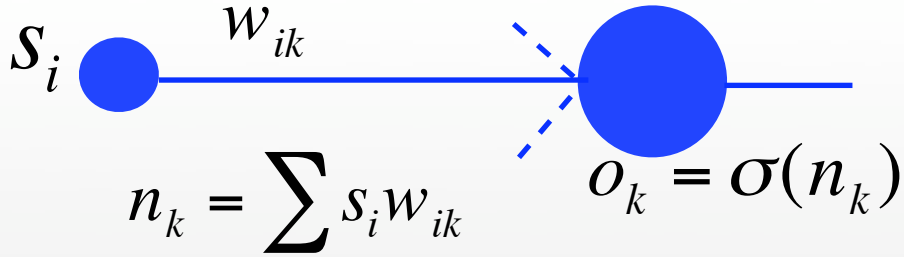
- back-prop involves two distinct stages:
- (1) a forward-pass in which outputs are computed given inputs & weights; error is computed as target minus output
- (2) a backwards-pass in which error signals are propagated backwards in order to adjust weights

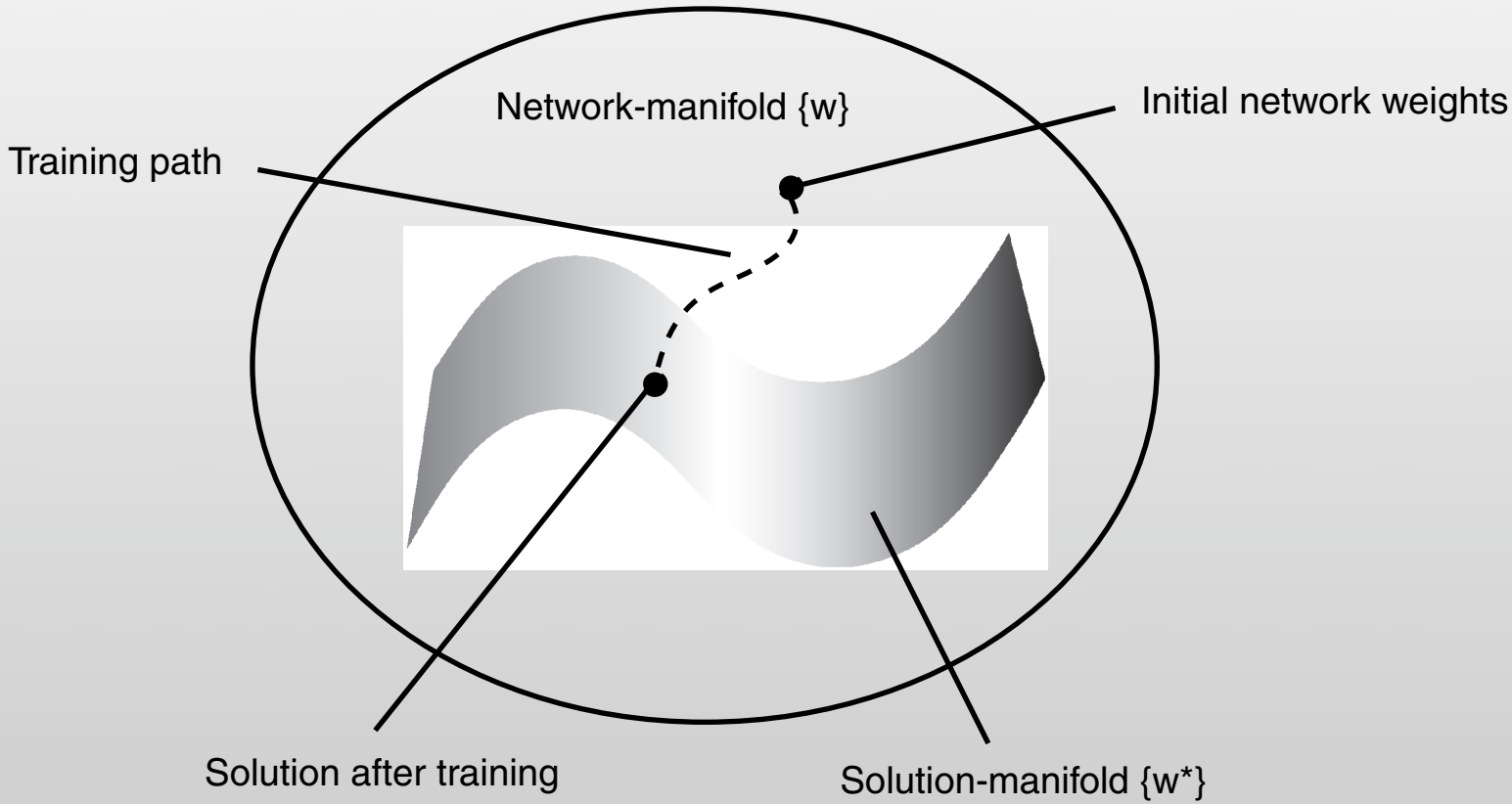
the Delta Rule

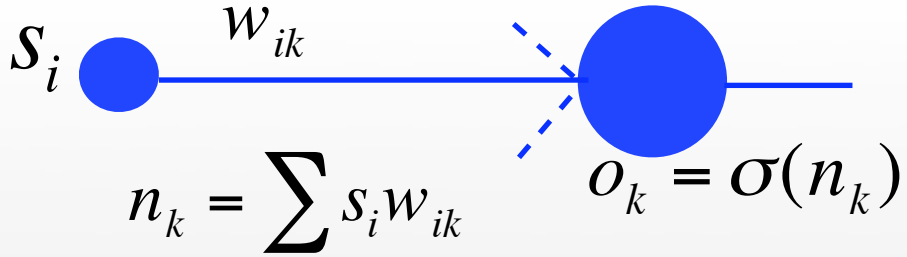
- first : let's mathematically derive the "delta rule"
- delta rule gives an expression for how to change the weights, given a certain output error of the network
- based on simple calculus
- we will derive the delta rule for adapting weights connecting to the output layer first
- then we will derive the rule for adapting hidden layer weights



- start with simple one-layer net
- given an output o_k, how to change weight w_{ik}?
- what we want to define is the gradient of the error manifold
- i.e. we want an expression relating change in w_{ik} to change in error.
- Then what we will do to adjust weights during learning, is adjust w_{ik} in a direction in which error is reduced (down the gradient)


s_i
w_{ik}
$n_k = \sum s_i w_{ik}$
$o_k = \sigma(n_k)$


Network-manifold {w}
Initial network weights
Training path
Solution after training
Solution-manifold {w*}



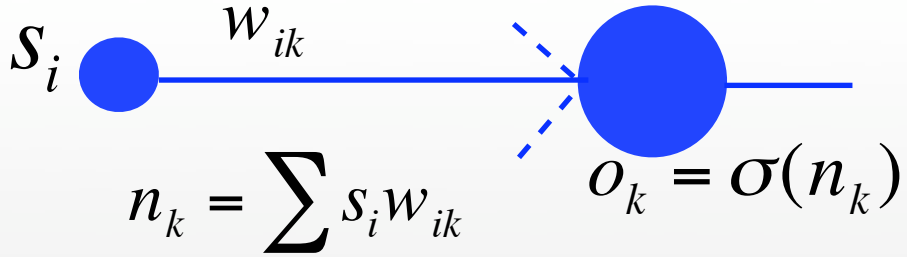
gradient of error manifold: $\frac{\partial E}{\partial w_{ik}}$

remember the chain rule: $\frac{\partial A}{\partial C} = \frac{\partial A}{\partial B}\frac{\partial B}{\partial C}$

split up derivative: $\frac{\partial E}{\partial w_{ik}} = \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial w_{ik}}$

let's find expressions for each of the sub-terms:

$$\frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial w_{ik}}$$



subterm: $\frac{\partial E}{\partial o_k}$ expand: $E = (t_k - o_k)^2$

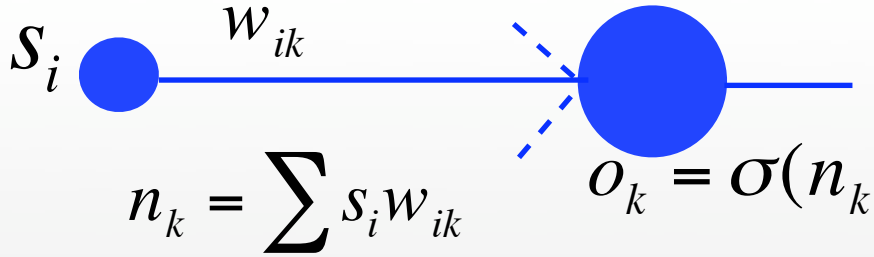
to derive the partial derivative, remember rule:

if $h(x) = f(g(x))$ then $h'(x) = f'(g(x))g'(x)$

in our case: $f(x) = x^2$ and: $g(x) = t_k - o_k$

so: $\frac{\partial E}{\partial o_k} = 2(t_k - o_k)(-1)$

$$\boxed{\frac{\partial E}{\partial o_k} = -2(t_k - o_k)}$$



subterm: $\dfrac{\partial o_k}{\partial w_{ik}}$

expand: $o_k = \sigma\left(\sum_i s_i w_{ik}\right)$

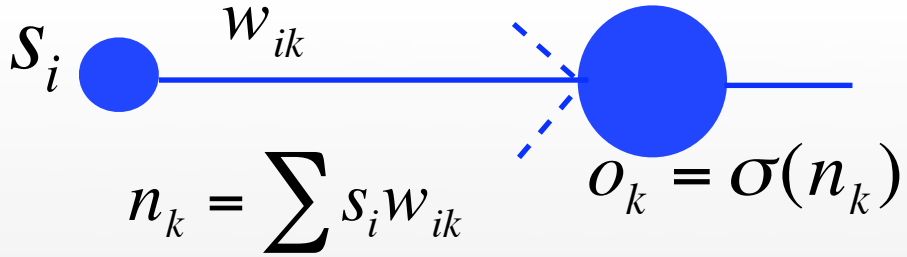
where $\sigma(x)$ is transfer function

remember, if

$$h(x) = f(g(x)) \quad \text{then} \quad h'(x) = f'(g(x))g'(x)$$

$$f(x) = \sigma(x) \quad g(x) = \sum_i s_i w_{ik}$$

$$\boxed{\frac{\partial o_k}{\partial w_{ik}} = \sigma'(n_k)s_i}$$

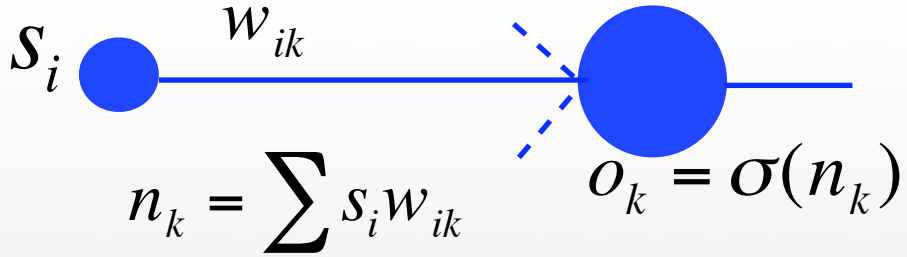


$$\frac{\partial E}{\partial w_{ik}} = \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial w_{ik}}$$

$$\frac{\partial E}{\partial o_k} = -2(t_k - o_k)$$

$$\frac{\partial o_k}{\partial w_{ik}} = \sigma'(n_k)s_i$$

$$\boxed{\frac{\partial E}{\partial w_{ik}} = -2(t_k - o_k)\sigma' s_i}$$ local gradient



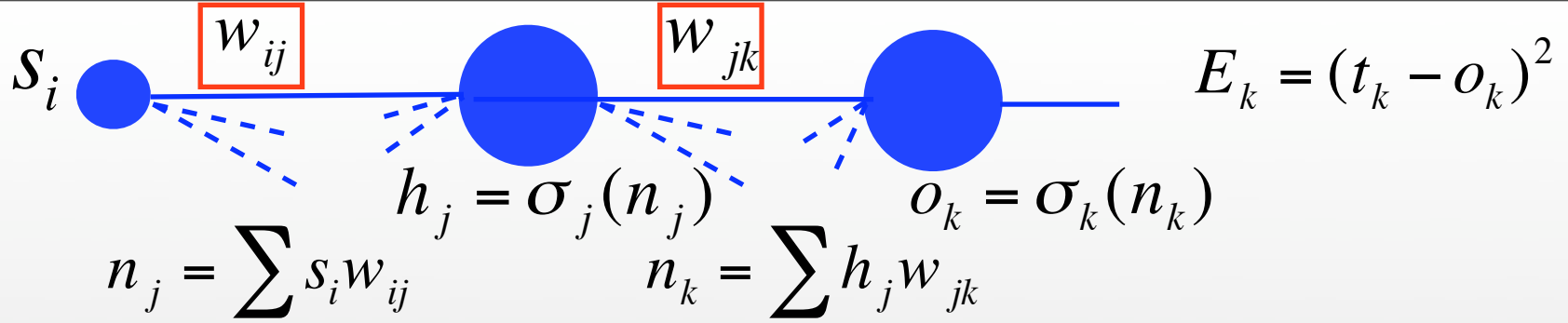
$$\frac{\partial SSE}{\partial w_{ik}} = -2e_k \sigma' s_i$$ local gradient

delta rule: $\Delta w_{ik} = \eta e_k \sigma' s_i$

η is a learning rate parameter

this works for output neurons where we know what the error e_k is.

What about hidden layer neurons??
What is the "desired output" of a hidden neuron??



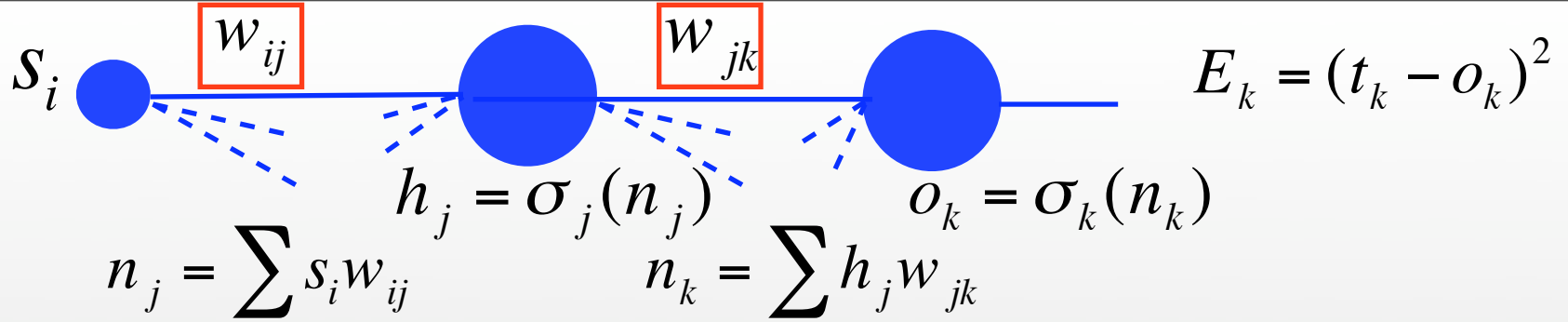
generalized delta rule: $\Delta w_{ij} = -\eta \delta_j s_i$

s_i is activation of sending unit associated with w_{ij}

η is learning rate parameter

δ_j is local gradient (contribution of receiving unit j toward overall error at layer k above)

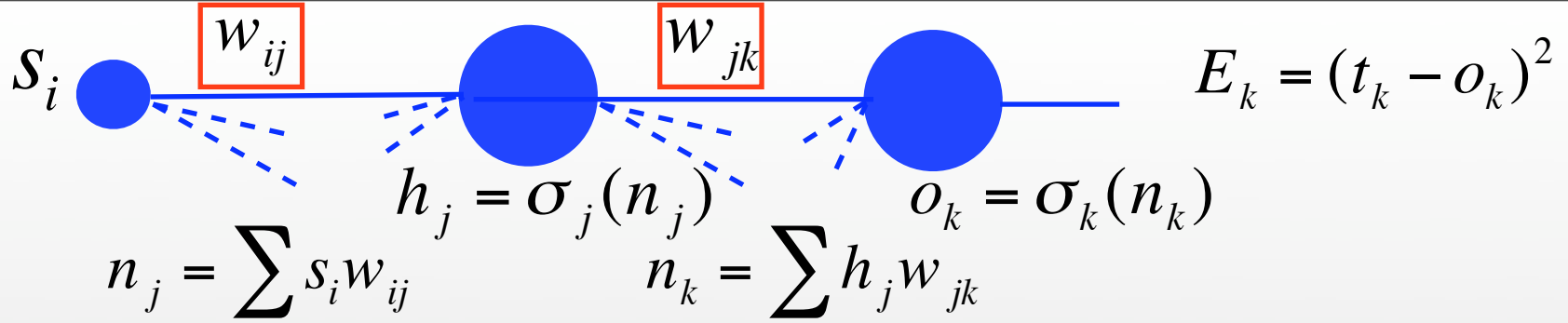
how to find δ_j ?



Main idea behind backprop: We can train weights from input to hidden layer by applying Chain Rule all the way down through the network

Chain of derivatives:

$$\frac{\partial E}{\partial w_{ij}} = \sum_k \frac{\partial E}{\partial n_k} \frac{\partial n_k}{\partial h_j} \frac{\partial h_j}{\partial n_j} \frac{\partial n_j}{\partial w_{ij}}$$



after doing the calculus (see Haykin ch. 4, Trappenberg ch. 10), we get for a hidden neuron j,

$$\delta_j = \sigma'_j \sum_k e_k \sigma'_k w_{jk} \qquad \delta_k = e_k \sigma'_k \quad \Rightarrow \quad \boxed{\delta_j = \sigma'_j \sum_k \delta_k w_{jk}}$$

δ_j contribution of a neuron to overall error at next layer up equals sum (over all neurons that it projects to), of the contributions of those upper neurons to overall error, multiplied by the weights of the connections

so generalized recursive update rule for weights at any layer:

$$\boxed{\Delta w_{ij} = -\eta \delta_j s_i} \qquad \text{where} \qquad \boxed{\delta_j = \sigma'_j \sum_k \delta_k w_{jk}}$$

Backprop

$$\Delta w_{ij} = -\eta \delta_j s_i$$

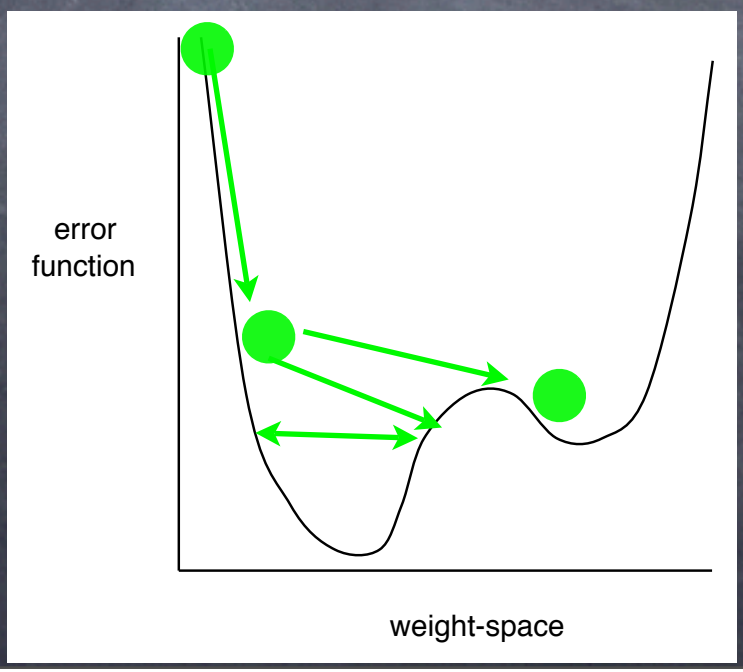
for hidden layer: $\delta_j = \sigma'_j \sum_k \delta_k w_{jk}$

for output layer: $\delta_k = e_k \sigma'_k$

1. propagate activations forwards to output layer, and compute activations and then errors e_k
2. compute δ_k for output neurons
3. use update rule to update output weights w_{jk}
4. propagate δ_k backwards to compute δ_j for hidden layer neurons
5. use update rule to update hidden weights w_{ij}
6. This recursive method can work for many layers

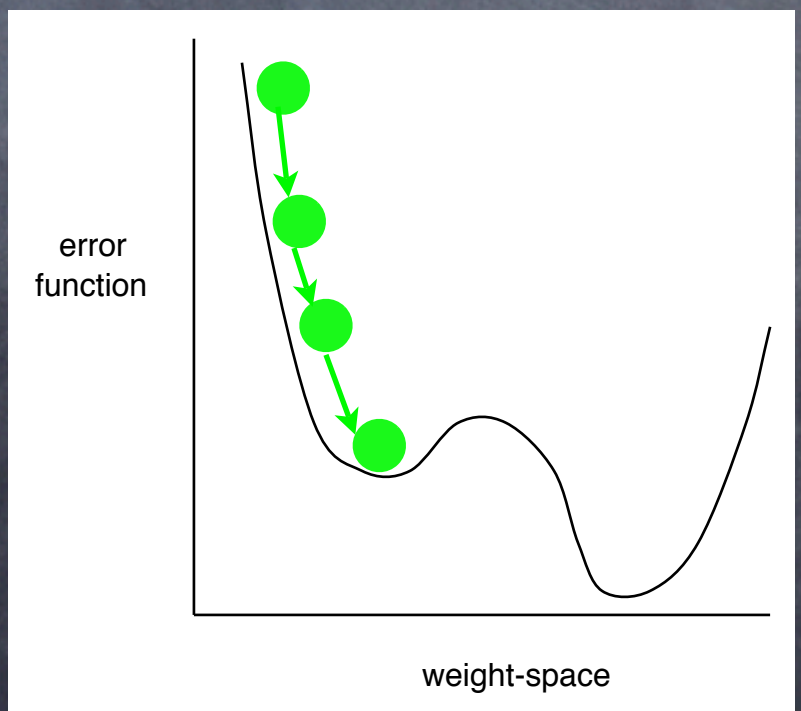
Learning Rate

- learning rate parameter is like a gain on the gradient
- controls how large the steps down the gradient are
- tradeoff
  - small steps guard against missing solution, but result in very long training times
  - large steps speed up training but introduce possibility of “stepping over” solution or oscillating around solution



Momentum

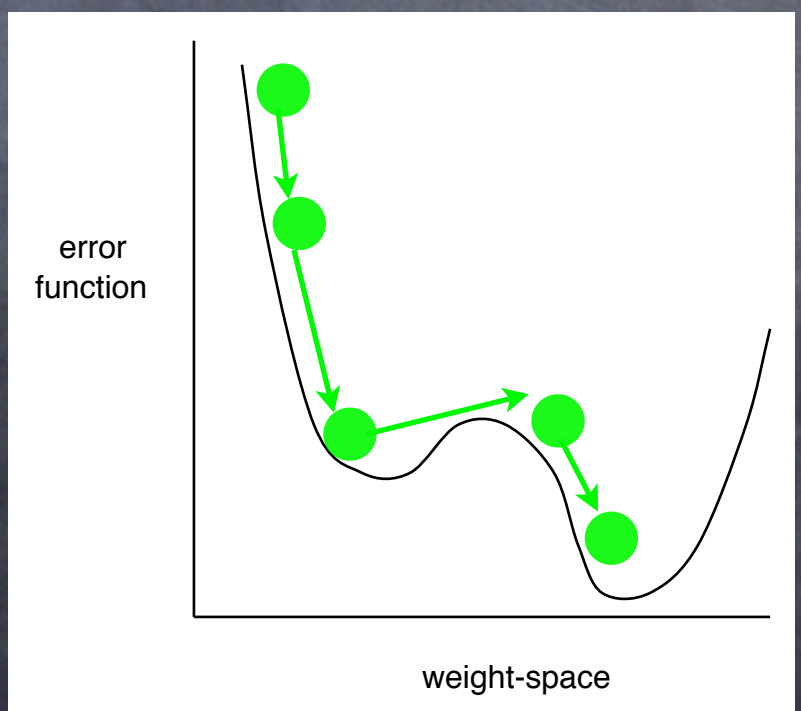
- if there are local minima in error manifold, training algorithm might get “stuck”
- to move in either direction would increase error
- thus stay where we are!
- miss better (lower error) solution



Momentum

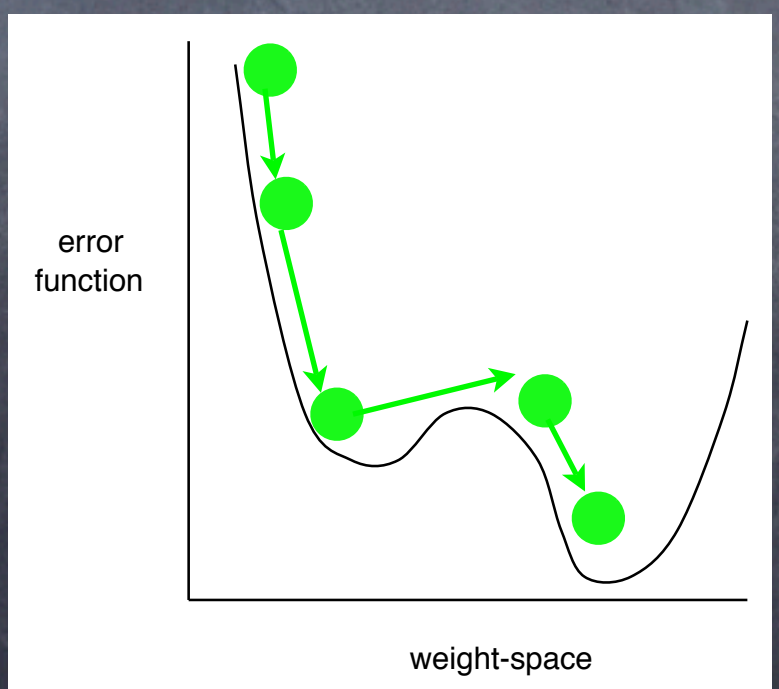
- concept of “momentum” from mechanics
- ignores small features in error surface
- add a momentum term to the delta rule that “remembers” the change of weights in the previous time step:

$$\Delta w_{ij}(t+1) = -\eta \frac{\partial E}{\partial w_{ij}} + \alpha \Delta w_{ij}(t)$$



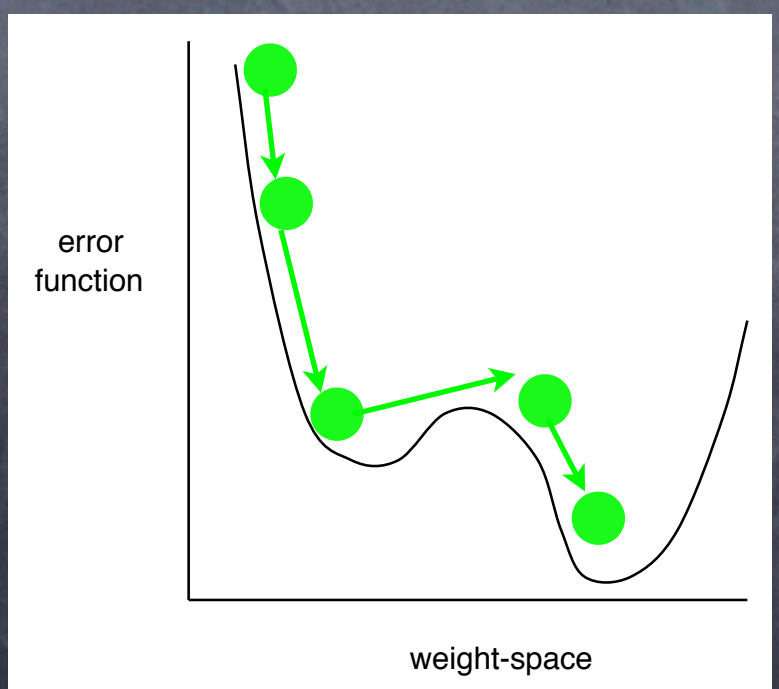
Momentum

- depending on which way one is approaching a local / global minimum,
- if momentum term is too big, you might skip over a global minimum and end up in a local minimum
- if momentum term is too small, you might stay in local minimum



Adaptive Schemes

- algorithms exist that implement adaptive learning rates, and adaptive momentum terms
- adjust learning rate and/or momentum terms on a trial-by-trial basis to take into account changes in gradient of error manifold
- matlab: implements all sorts of schemes



Matlab

- let's try to train a 2-layer network on the XOR problem using backprop
- matlab script nnxor.m
- try:
  - adjusting learning rate parameter
  - adaptive learning rate algorithm
- also see nntool command in matlab

Backprop notes

- transfer function must be differentiable in order to compute error contributions for a neuron $\delta_j = \sigma'_j \sum_k \delta_k w_{jk}$
- maximize information content: training examples should be chosen that fully span the input/output spaces you are trying to model
- tansig Tfs usually produce faster convergence than logsig Tfs
- appropriate Tf needs to be chosen for output layer so that target values can in principle be attained!!

Backprop notes

- input normalization: input variables can be pre-processed so that mean value over entire training set is close to zero
  - also useful: input variables should be uncorrelated (e.g. using PCA)
  - decorrelated input variables should be scaled so that their covariances are about equal (ensures different weights in network learn at about same speed)

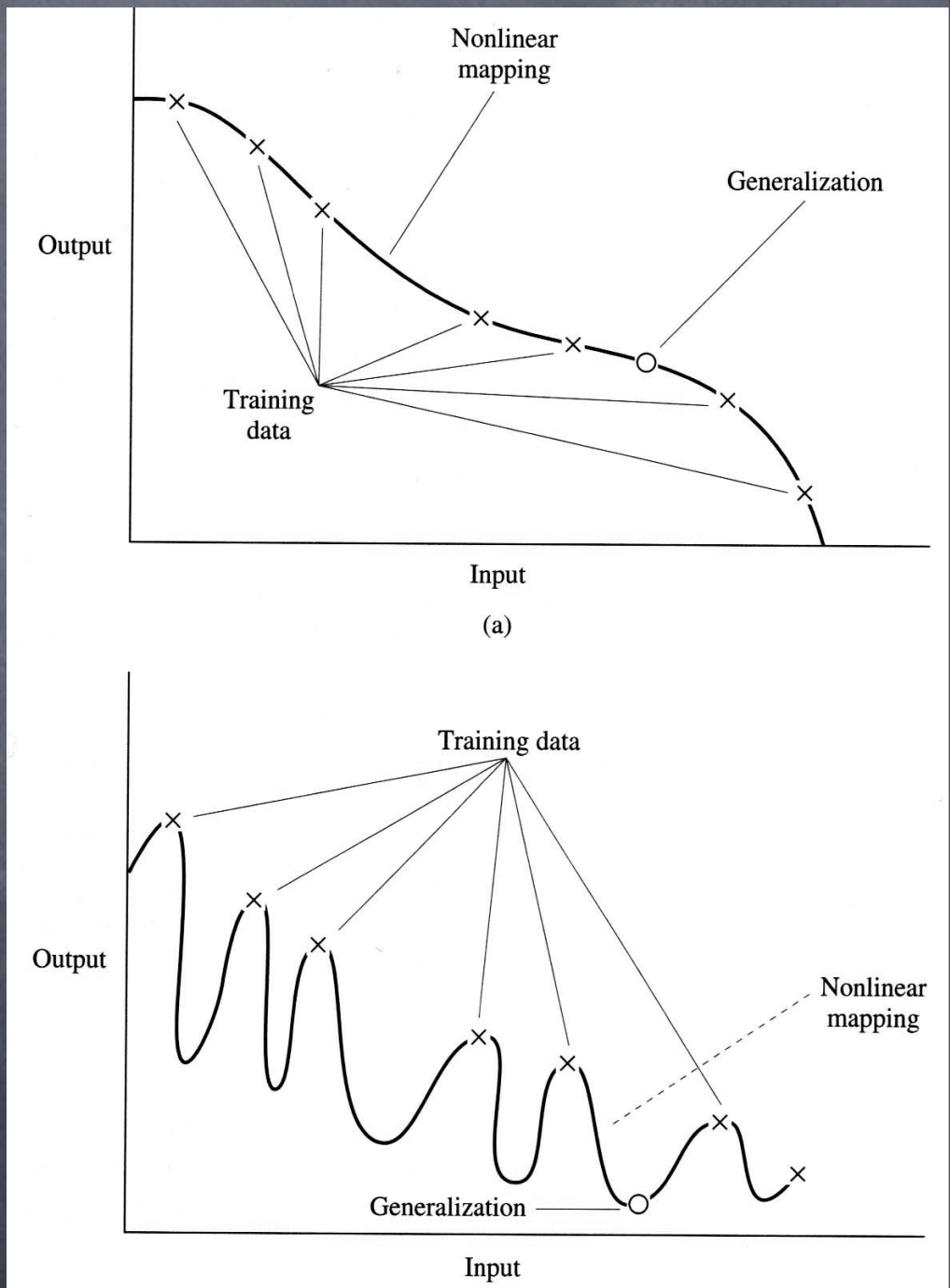
Backprop notes

- weight initialization: a good choice of initial weights can greatly affect speed (and success) of training
  - e.g. if initial weights are very large, then highly likely that neurons with sigmoidal Tfs will be driven into saturation... then local gradients will be very small, and learning will be slow
  - strategy: set initial weights so that s.d. of induced local field of neurons lies in linear portion of Tf
  - heuristic: choose initial weights from a uniform distribution with mean zero and variance equal to the reciprocal of the # connections of a neuron:

$$\sigma_w = m^{-1/2}$$

Generalization

- how many hidden units???
- small networks will do 2 things:
  1. represent the true function correctly
  2. generalize to pts not present in training
- large networks can only do #1 and will tend to overfit the dataset
- generalization will suffer!



Network size

so how to decide on # hidden units??
4 general strategies:

1. make a guess
2. make an educated guess (e.g. use heuristic like # hidden units should be about 10% of the # of training examples
3. network-building: start with small # of hidden units; train until no further improvement; add a hidden unit; re-train; keep adding until (a) overall goal error minimum is reached; or (b) no further improvement is gained by adding hidden units
4. network-pruning: start with large # of hidden units; train until network learns mapping; remove a hidden unit; retrain; repeat until further pruning decreases performance below desired level

Stopping rules

- maximum # of epochs reached
- minimum error reached
- minimum gradient reached
- no further improvement in cross-validation set

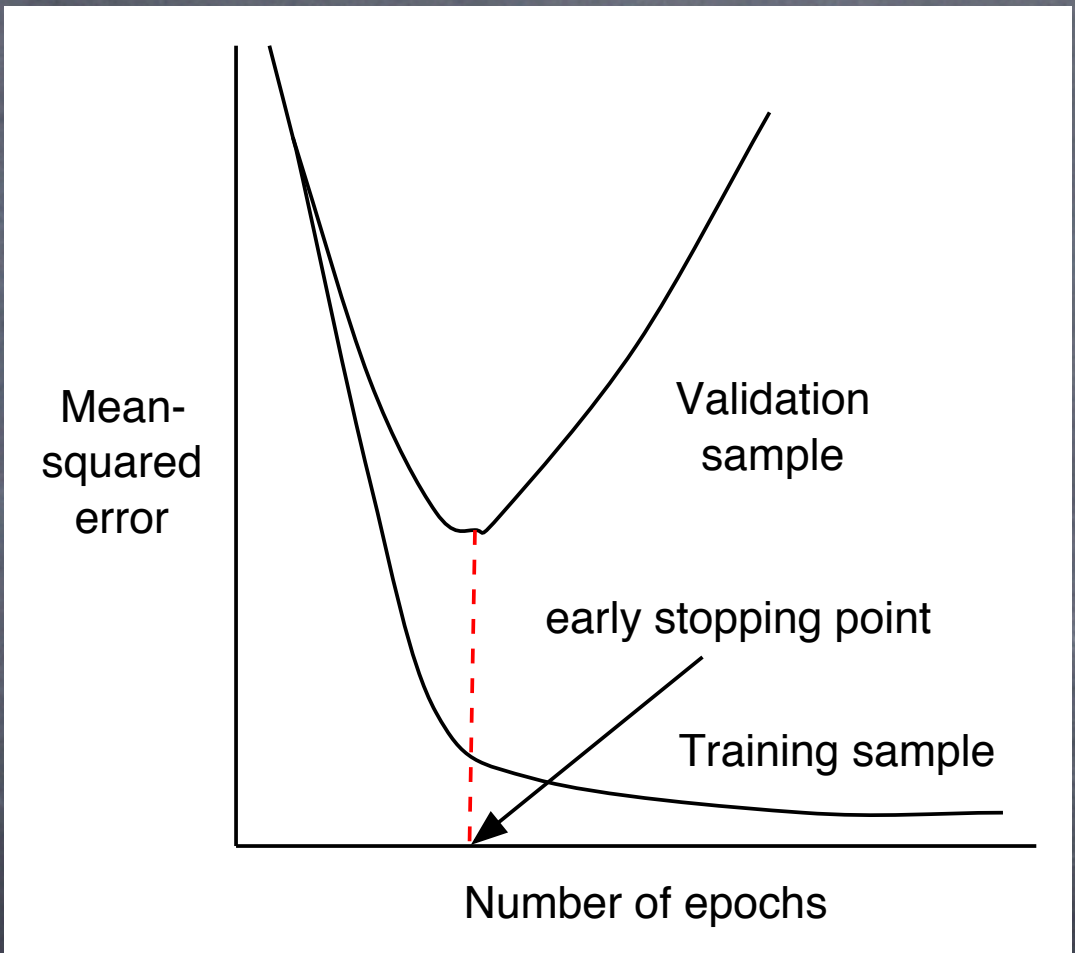
Cross-Validation

- appealing statistical tool for deciding on best solution, guarding against overfitting (or lack of generalization)
- available data is randomly partitioned into a training set and a test set
- training set further partitioned into:
  - estimation subset - used to select the model
  - validation subset - used to test/validate the model
- idea: validate model on a dataset different from the one used for parameter estimation

Cross-Validation

- procedure:
- train network using estimation subset
- after each epoch, test network using validation subset
- if performance on the validation subset stops improving or gets worse, STOP training
- the idea is that although further training may improve performance on estimation subset, that improvement will come at the expense of generalization to non-trained examples (e.g. those in the validation subset)
- finally, test network on test set

Cross-Validation



Biological Plausibility of Backprop

1. Sign of weights: NNet: pos and neg; brain: excitatory OR inhibitory (not both, not changing)
2. Global communication (e.g. hormones) ignored
3. Backwards propagation of errors: unlikely that there is rapid transmission of errors backwards along a network
4. Implies existence of “teacher” - not straightforward concept

Applications

- Aerospace: Aircraft autopilot systems
- Banking: document reading (e.g. cheques); credit application evaluation; credit card activity monitoring
- Defense: sonar object classification; image identification; weapon steering
- Financial: real estate appraisal; mortgage screening; currency / equity price prediction
- Industrial: large process control; process/equipment failure
- Manufacturing: beer testing; quality inspection; machine diagnosis
- Medical: breast cancer cell analysis; EEG/ECG analysis
- Computational Neuroscience!!