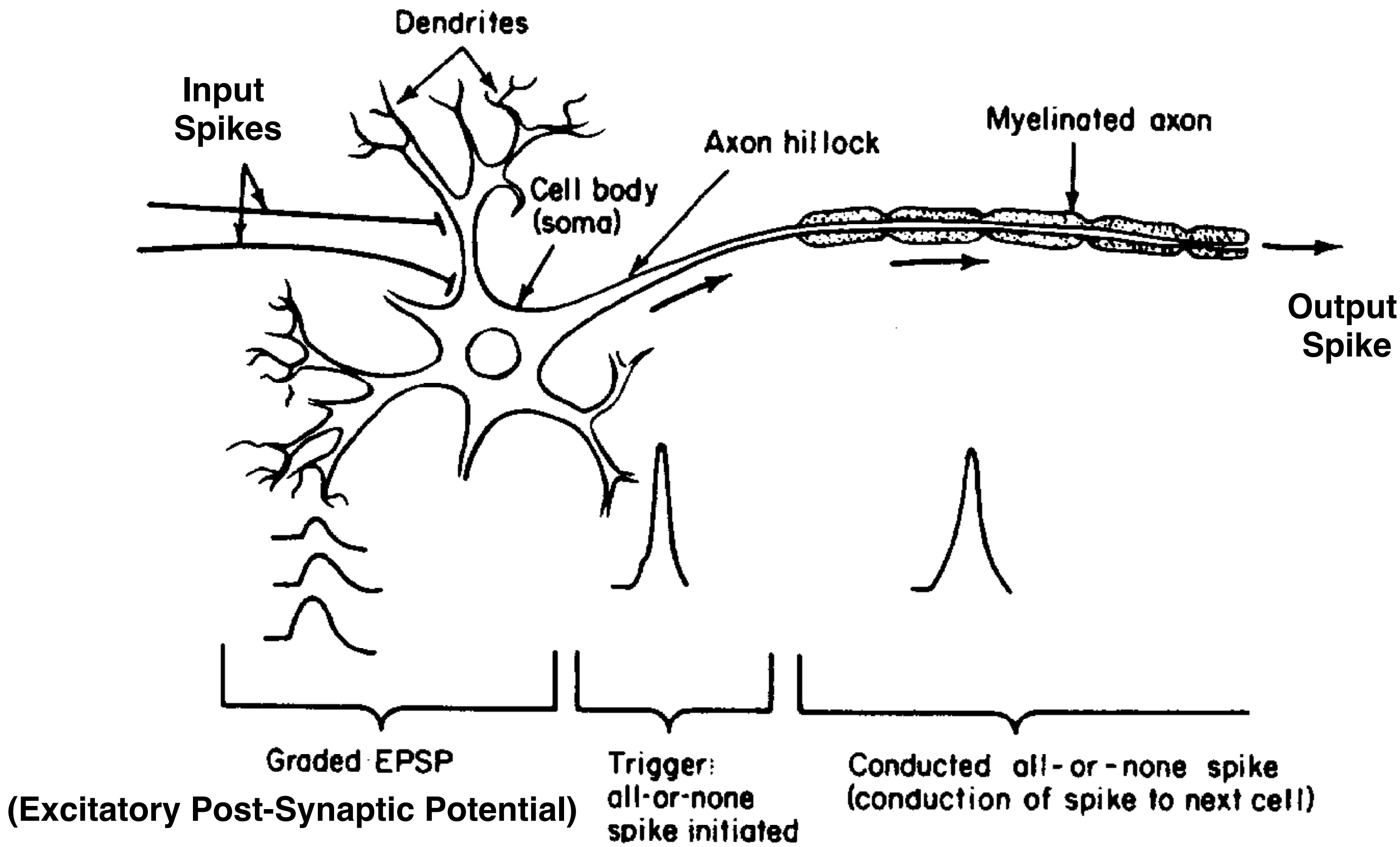
Introduction to Neural Networks

What are (Artificial) Neural Networks?

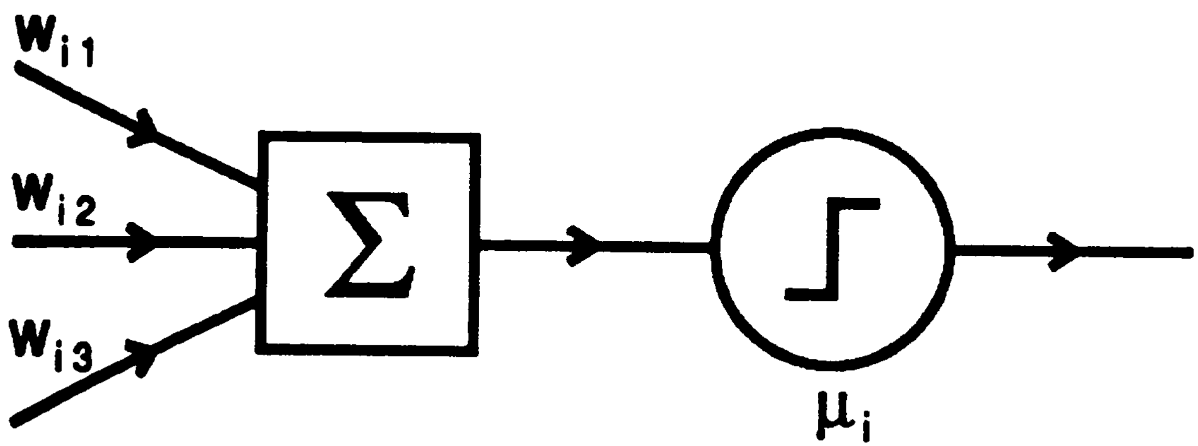
- Models of the brain and nervous system
- Highly parallel
  - Process information much more like the brain than a serial computer
- Learning
- Very simple principles
- Very complex behaviours
- Applications
  - As powerful problem solvers
  - As biological models

Basic Input-Output Transformation



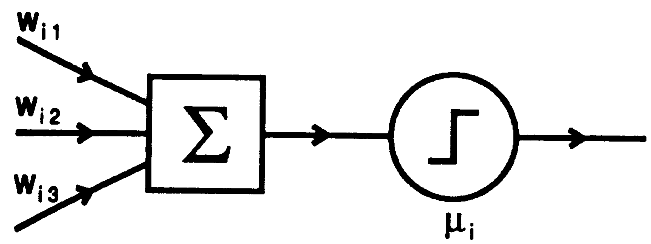
McCulloch–Pitts “neuron” (1943)

- Attributes of neuron
  - m binary inputs and 1 binary output (simplified model)
  - Synaptic weights w_{ij}
  - Threshold μ_i



McCulloch–Pitts Neural Networks

- Synchronous discrete time operation
  - Time quantized in units of synaptic delay



$$n_i(t+1) = \Theta\left[\sum_j w_{ij} n_j(t) - \mu_i\right]$$

$n_i \equiv$ output of unit i
$\Theta \equiv$ step function
w_{ij} = weight from unit j to i
μ_i = threshold

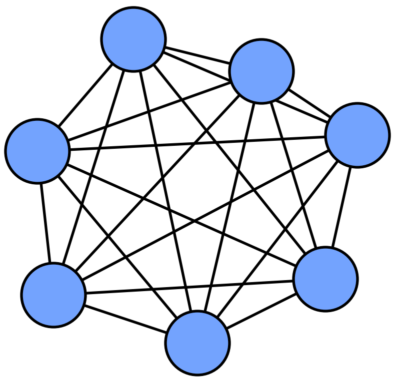
- Output is 1 if and only if weighted sum of inputs is greater than threshold
  $\Theta(x) = 1$ if $x \geq 0$ and 0 if $x < 0$
  (Θ, the output function, is called *activation function*)

- Remarks:
  - Behavior of network can be simulated by a finite automaton (FA)
  - Any FA can be simulated by a McCulloch-Pitts Network

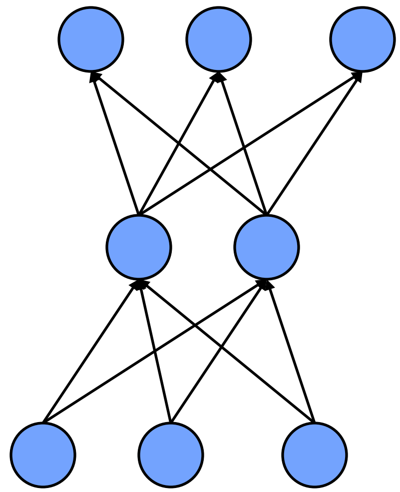
Properties of Artificial Neural Networks

- High level abstraction of neural input-output transformation
  - Inputs → weighted sum of inputs → nonlinear function → output
- Often used where data or functions are uncertain
  - Goal is to learn from a set of training data
  - And to generalize from learned instances to new unseen data
- Key attributes
  - Parallel computation
  - Distributed representation and storage of data
  - Learning (networks adapt themselves to solve a problem)
  - Fault tolerance (insensitive to component failures)

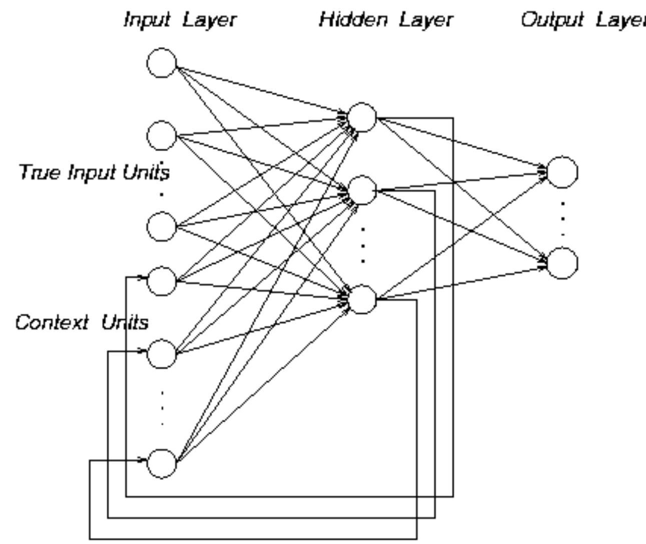
Topologies of Neural Networks



completely connected



feedforward (directed, a-cyclic)



recurrent (feedback connections)

Networks Types

- Feedforward versus recurrent networks
  - Feedforward: No loops, input → hidden layers → output
  - Recurrent: Use feedback (positive or negative)
- Continuous versus spiking
  - Continuous networks model mean spike rate (firing rate)
    - Assume spikes are integrated over time
  - Consistent with rate-code model of neural coding
- Supervised versus unsupervised learning
  - Supervised networks use a "teacher"
    - The desired output for each input is provided by user
  - Unsupervised networks find hidden statistical patterns in input data
    - Clustering, principal component analysis

History

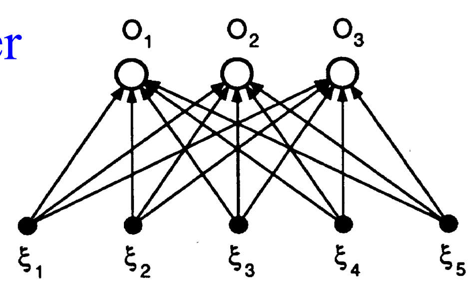
- 1943: McCulloch–Pitts "neuron"
  - Started the field
- 1962: Rosenblatt's perceptron
  - Learned its own weight values; convergence proof
- 1969: Minsky & Papert book on perceptrons
  - Proved limitations of single-layer perceptron networks
- 1982: Hopfield and convergence in symmetric networks
  - Introduced energy-function concept
- 1986: Backpropagation of errors
  - Method for training multilayer networks
- Present: Probabilistic interpretations, Bayesian and spiking networks

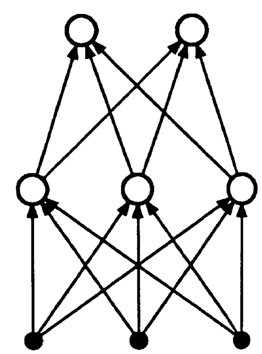
Perceptrons

- In machine learning, the *perceptron* is an algorithm for supervised learning of binary classifiers: functions that can decide whether an input (represented by a vector of numbers) belongs to one class or another.
- Attributes
  - Layered feedforward networks
  - Supervised learning
    - Hebbian: Adjust weights to enforce correlations
  - Parameters: weights w_{ij}
  - Binary output = Θ(weighted sum of inputs)
    - Take w_o to be the threshold with fixed input –1.

$$Output_i = \Theta\left[\sum_j w_{ij}\xi_j\right]$$

Single-layer





Multilayer

Training Perceptrons to Compute a Function

- Given inputs ξ_j to neuron i and desired output Y_i, find its weight values by iterative improvement:
  1. Feed an input pattern
  2. Is the binary output correct?
     - ⇒Yes: Go to the next pattern
     - ⇒No: Modify the connection weights using error signal $(Y_i - O_i)$
     - ⇒Increase weight if neuron didn't fire when it should have and vice versa

$$\Delta w_{ij} = \eta \left(Y_i - O_i \right) \xi_j$$

$\eta \equiv$ learning rate

$\xi_j \equiv$ input

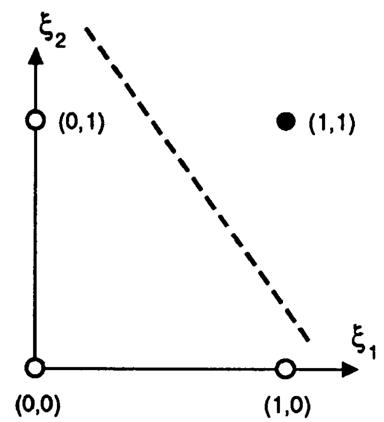
$Y_i \equiv$ desired output

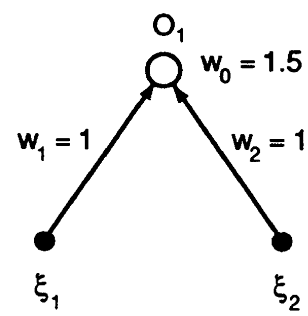
$O_i \equiv$ actual output

- Learning rule is Hebbian (based on input/output correlation)
  - This update rule is in fact the stochastic gradient descent update for linear regression, converging to least square error.
  - converges in a finite number of steps if a solution exists
  - Used in ADALINE (adaptive linear neuron) networks

Computational Power of Perceptrons

- Consider a single-layer perceptron
  - Assume threshold units
  - Assume binary inputs and outputs
  - Weighted sum forms a linear hyperplane $\sum_j w_{ij}\xi_j = 0$
- Consider a single output network with two inputs
  - Only functions that are linearly separable can be computed
  - Example: AND is linearly separable

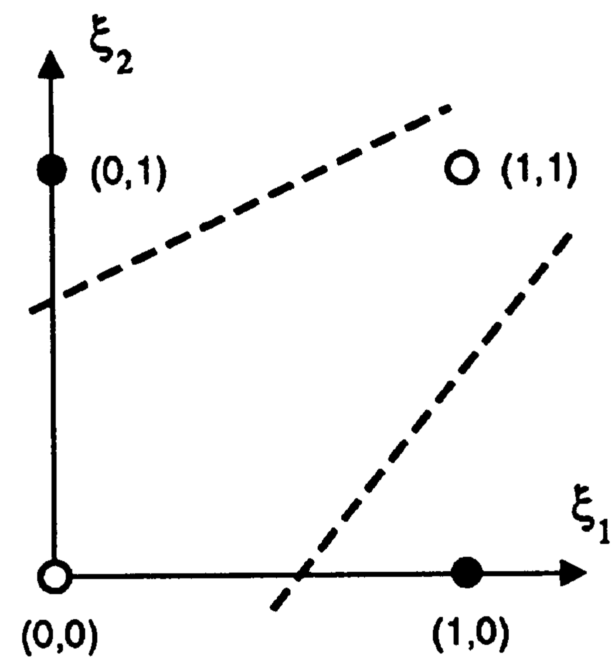




$\xi_o = -1$

Linear inseparability

- Single-layer perceptron with threshold units fails if problem is not linearly separable
  - Example: XOR



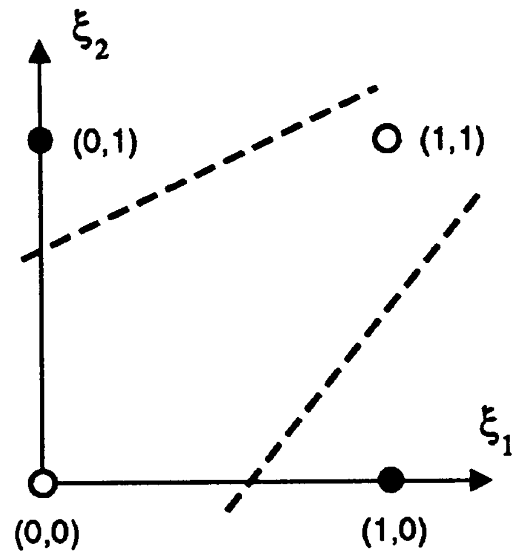
- Can use other tricks (e.g. complicated threshold functions) but complexity blows up

- Minsky and Papert's book showing these negative results was very influential

Solution in 1980s: Multilayer perceptrons

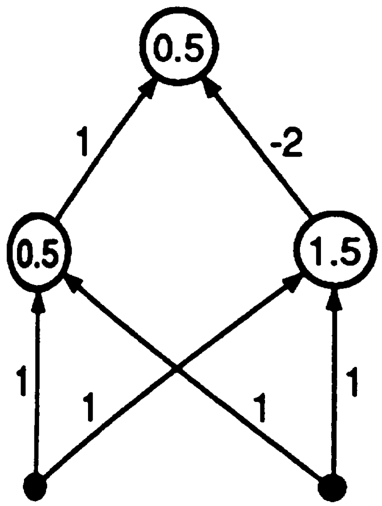
- Removes many limitations of single-layer networks
  - Can solve XOR

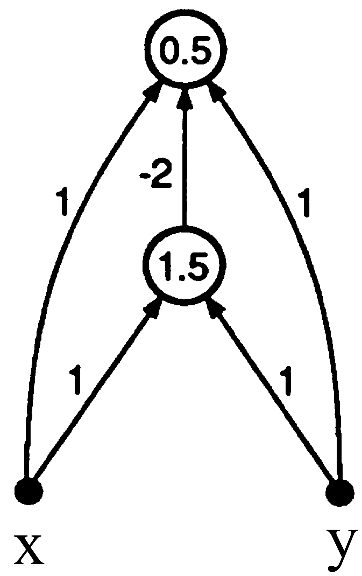
- Exercise: Draw a two-layer perceptron that computes the XOR function
  - 2 binary inputs ξ_1 and ξ_2
  - 1 binary output
  - One "hidden" layer
  - Find the appropriate weights and threshold



Solution in 1980s: Multilayer perceptrons

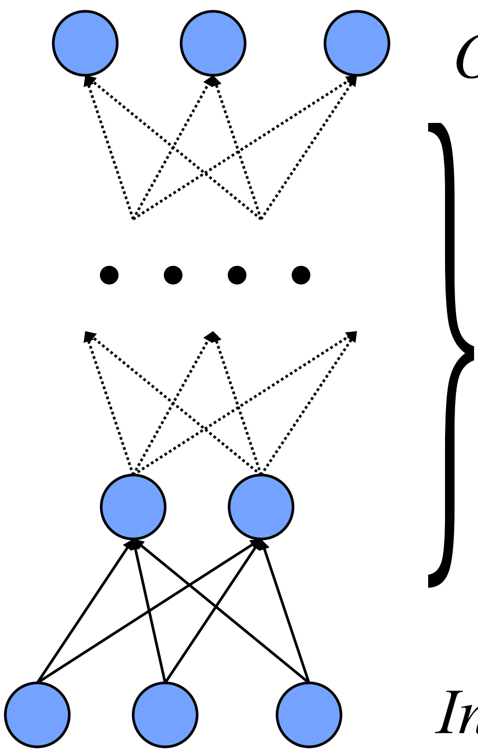
- Examples of two-layer perceptrons that compute XOR





- E.g. Right side network
  - Output is 1 if and only if x + y – 2(x + y – 1.5 > 0) – 0.5 > 0

Multilayer Perceptron



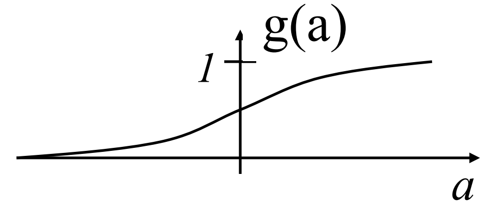
Output neurons

One or more layers of hidden units (hidden layers)

Input nodes

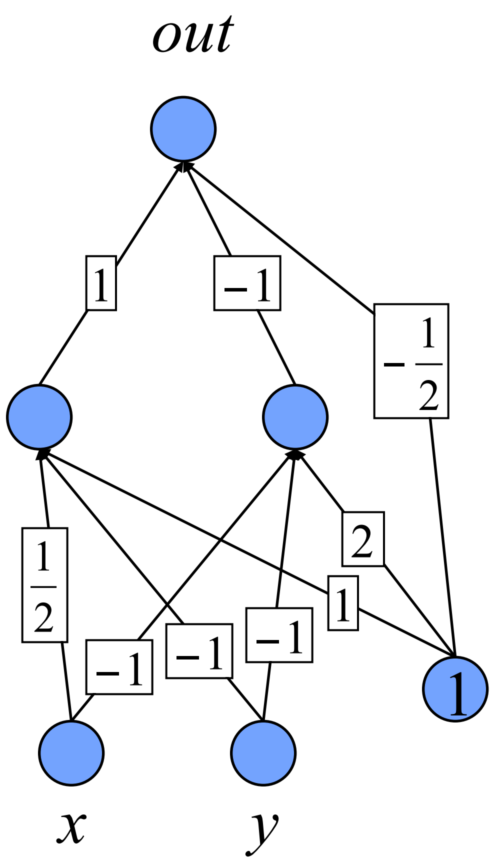
The most common output function (Sigmoid):

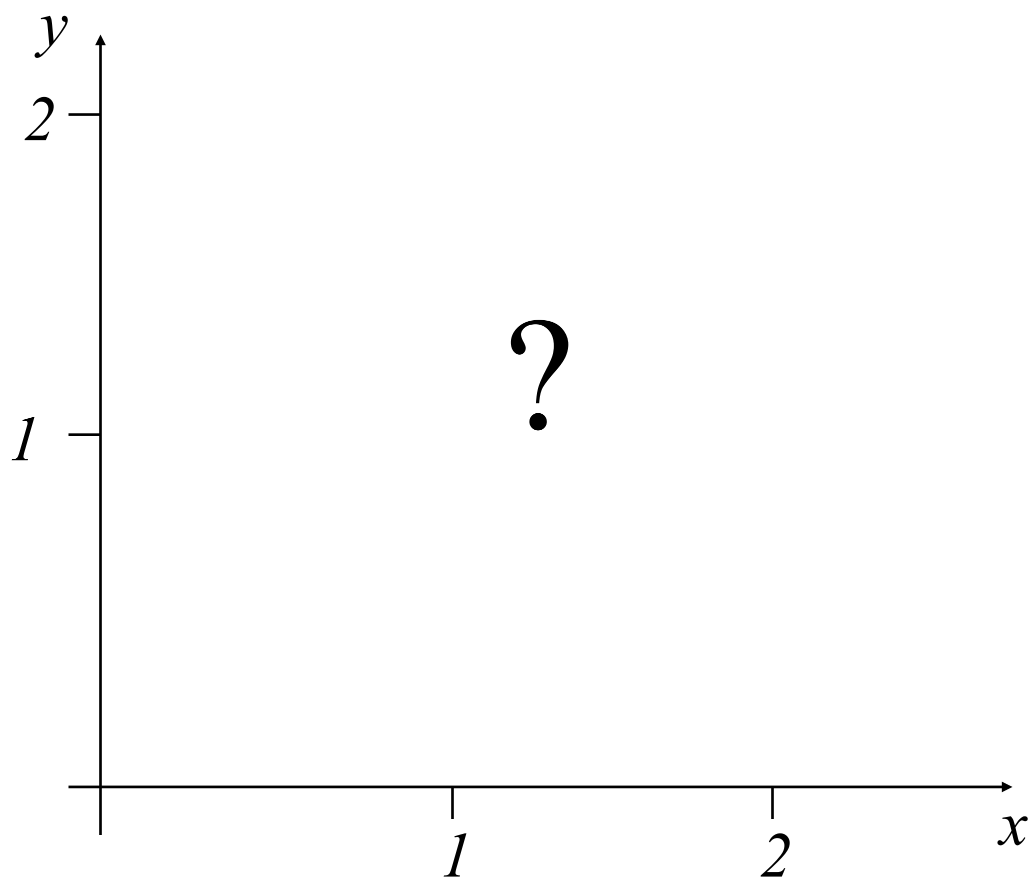
$$g(a) = \frac{1}{1 + e^{-\beta a}}$$



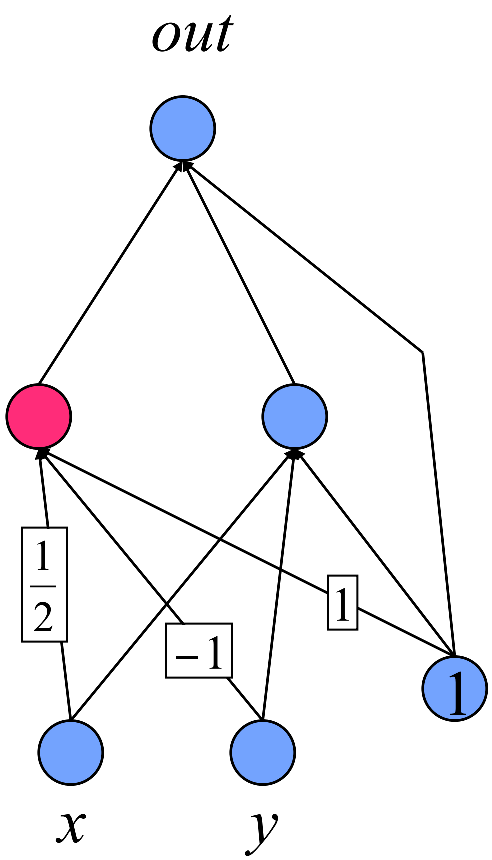
(non-linear squashing function)

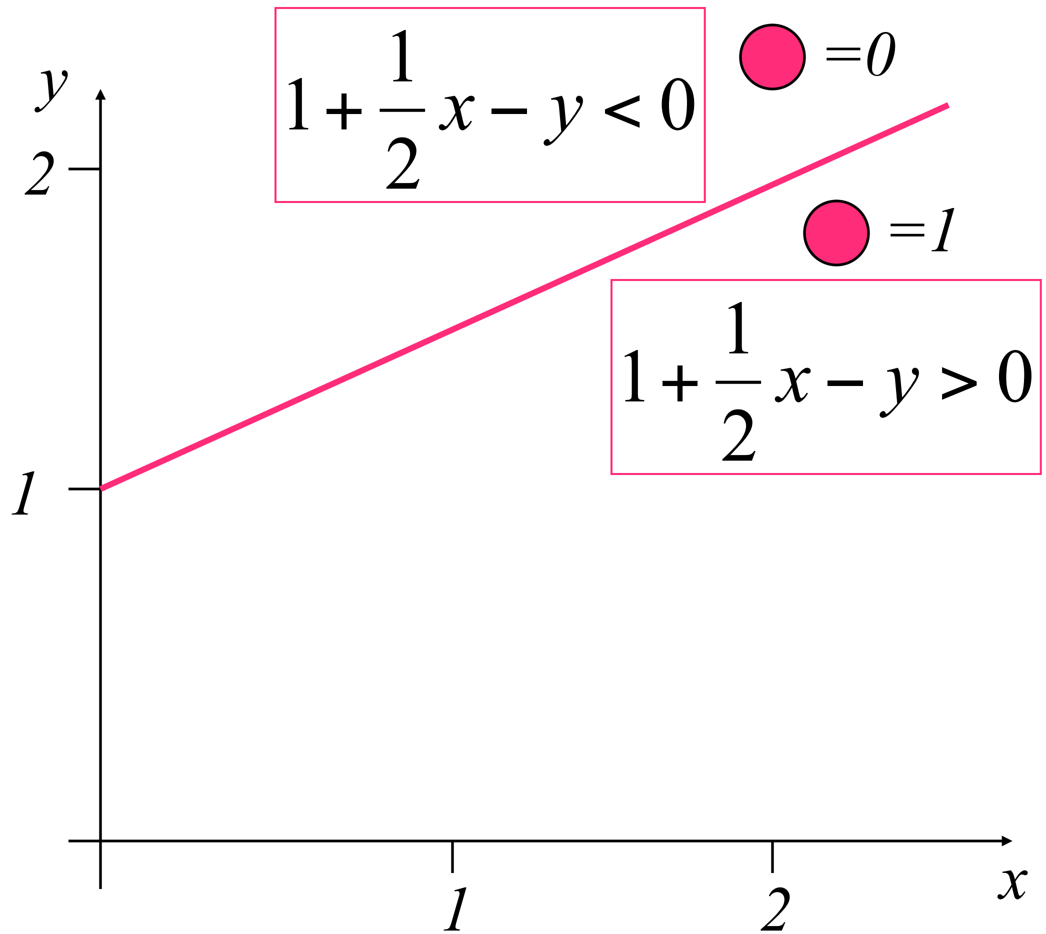
Example: Perceptrons as Constraint Satisfaction Networks



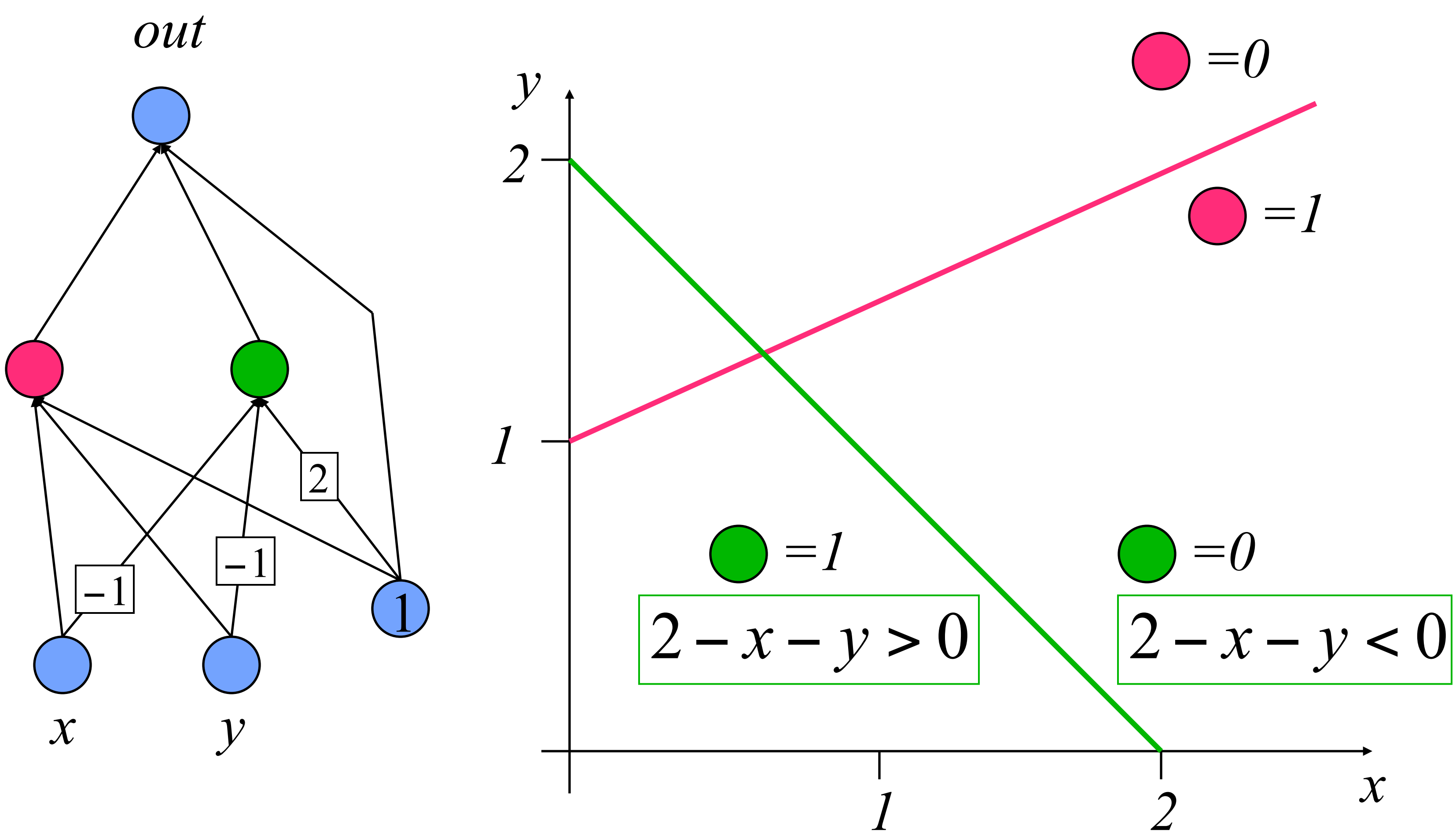


Example: Perceptrons as Constraint Satisfaction Networks

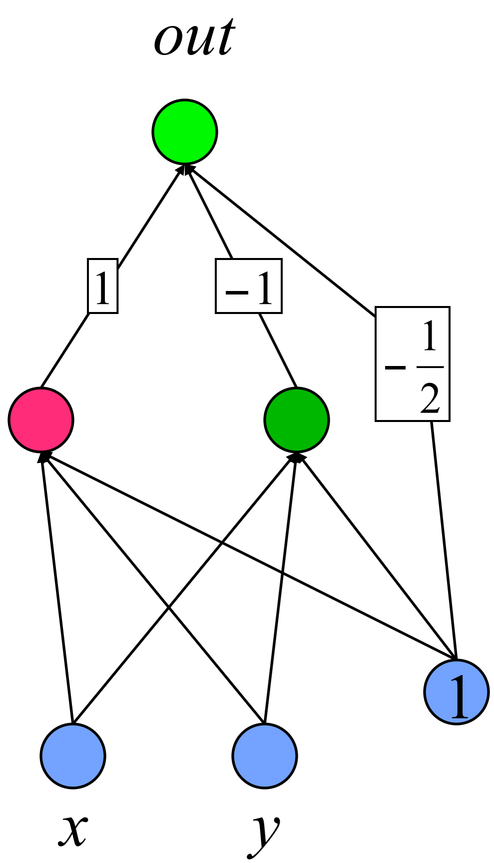


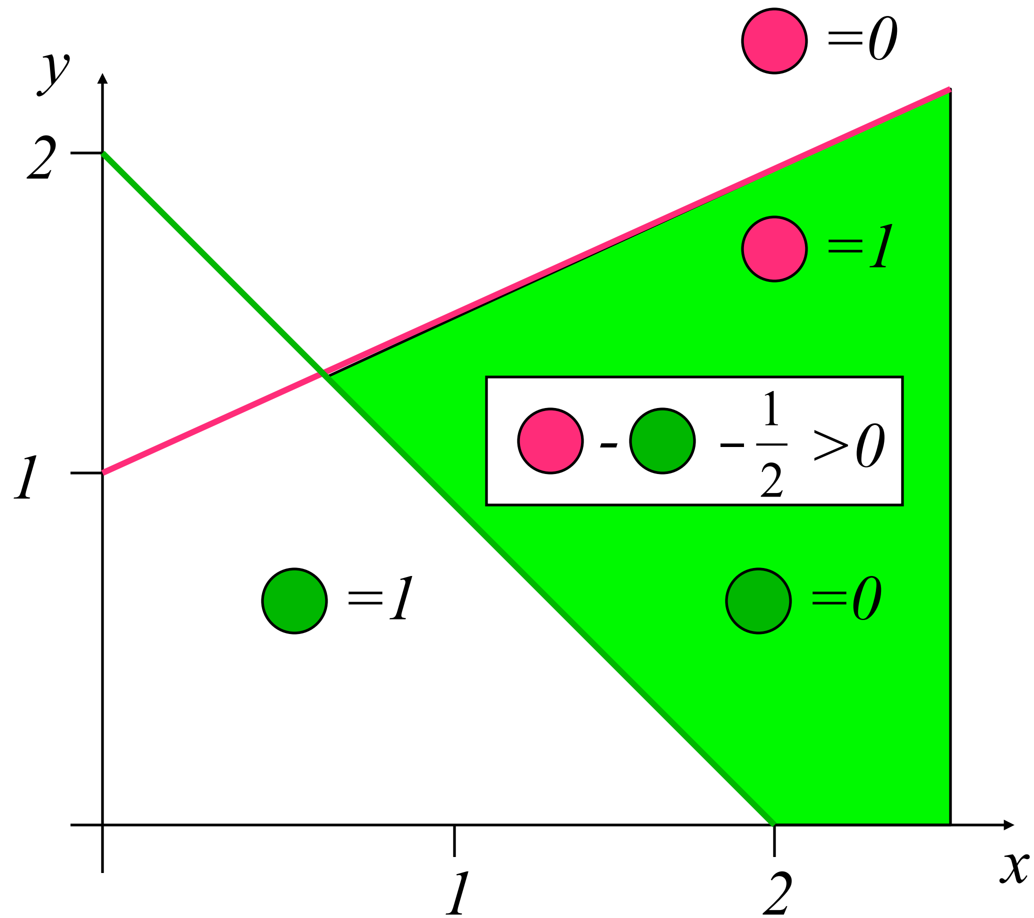


Example: Perceptrons as Constraint Satisfaction Networks

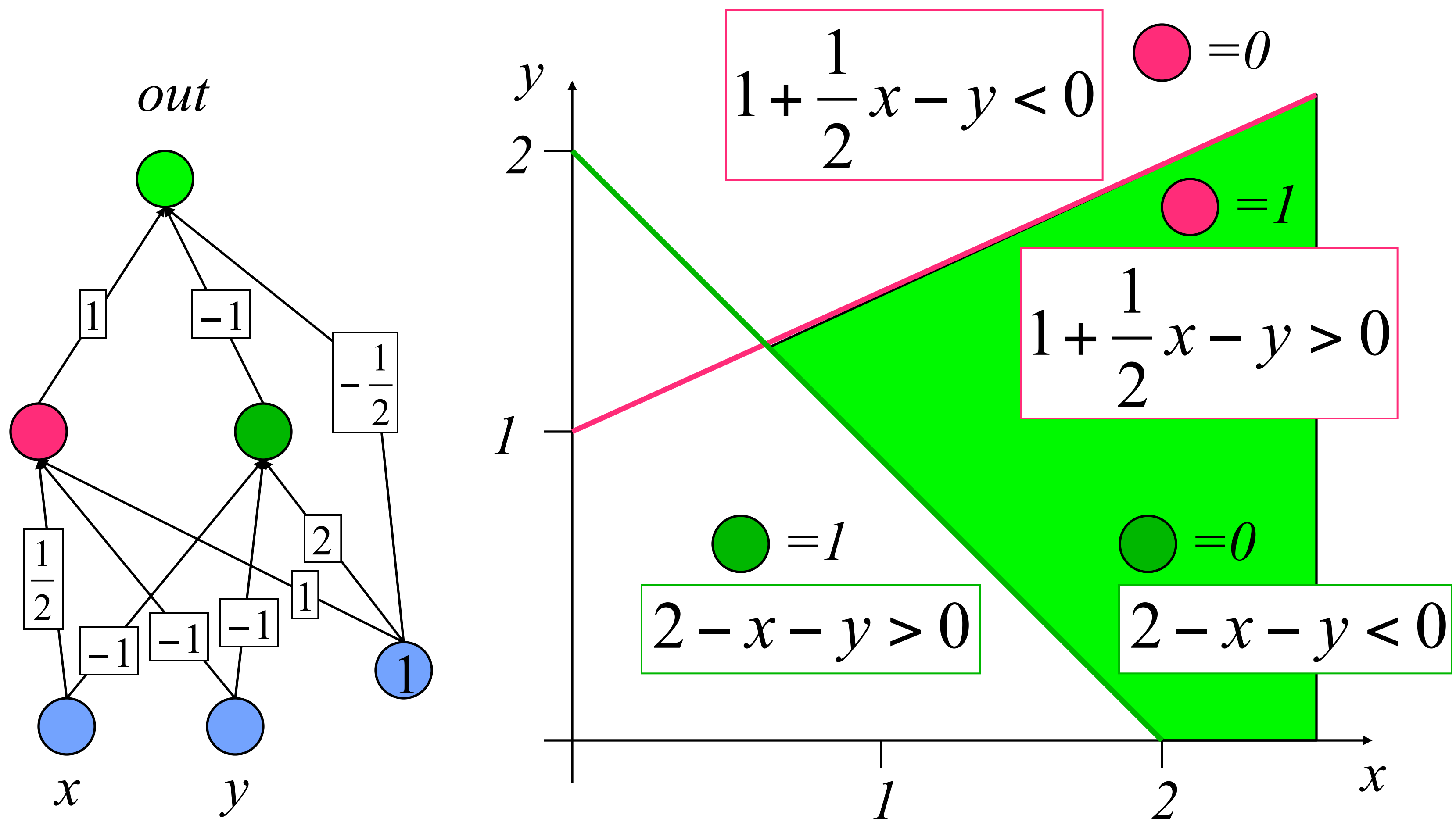


Example: Perceptrons as Constraint Satisfaction Networks



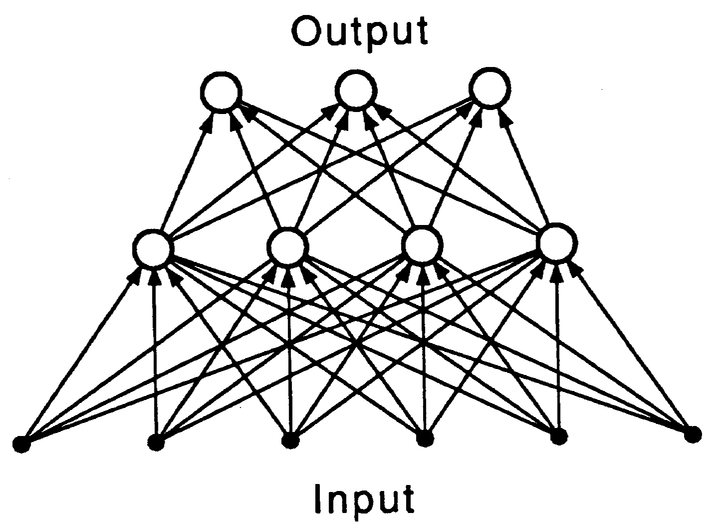


Perceptrons as Constraint Satisfaction Networks



Learning networks

- We want networks that configure themselves
  - Learn from the input data or from training examples
  - Generalize from learned data



Can this network configure itself to solve a problem?

How do we train it?

Gradient-descent learning

- Use a differentiable activation function
  - Try a continuous function $f(\,)$ instead of $\Theta(\,)$
    - First guess: Use a linear unit (without activation function $f(\,)$)
  - Define an error function (cost function or "energy" function)

$$E = \frac{1}{2}\sum_i \sum_u \left[Y_i^u - \sum_j w_{ij}\xi_j^{\mu} \right]^2$$

The idea is to make the change of the weight proportional to the negative derivative of the error.

Then $$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} = \eta \sum_u \left[Y_i^u - \sum_j w_{ij}\xi_j^{\mu} \right] \xi_j^{\mu} \,, \quad w_{ij} = w_{ij} + \Delta w_{ij}$$

- Changes weights in the direction of smaller errors
  - Minimizes the mean-squared error over input patterns μ
  - Called Delta rule = adaline rule = Widrow-Hoff rule = LMS rule

Gradient-descent learning

Then $\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} = \eta \sum_u \left[Y_i^u - \sum_j w_{ij} \xi_j^{\mu} \right] \xi_j^{\mu}$

About learning rate η :

In order for Gradient Descent to work we must set η to an appropriate value. This parameter determines how fast or slow we will move towards the optimal weights. If the η is very large we will skip the optimal solution. If it is too small we will need too many iterations to converge to the best values. So using a good η is crucial.

Backpropagation of errors

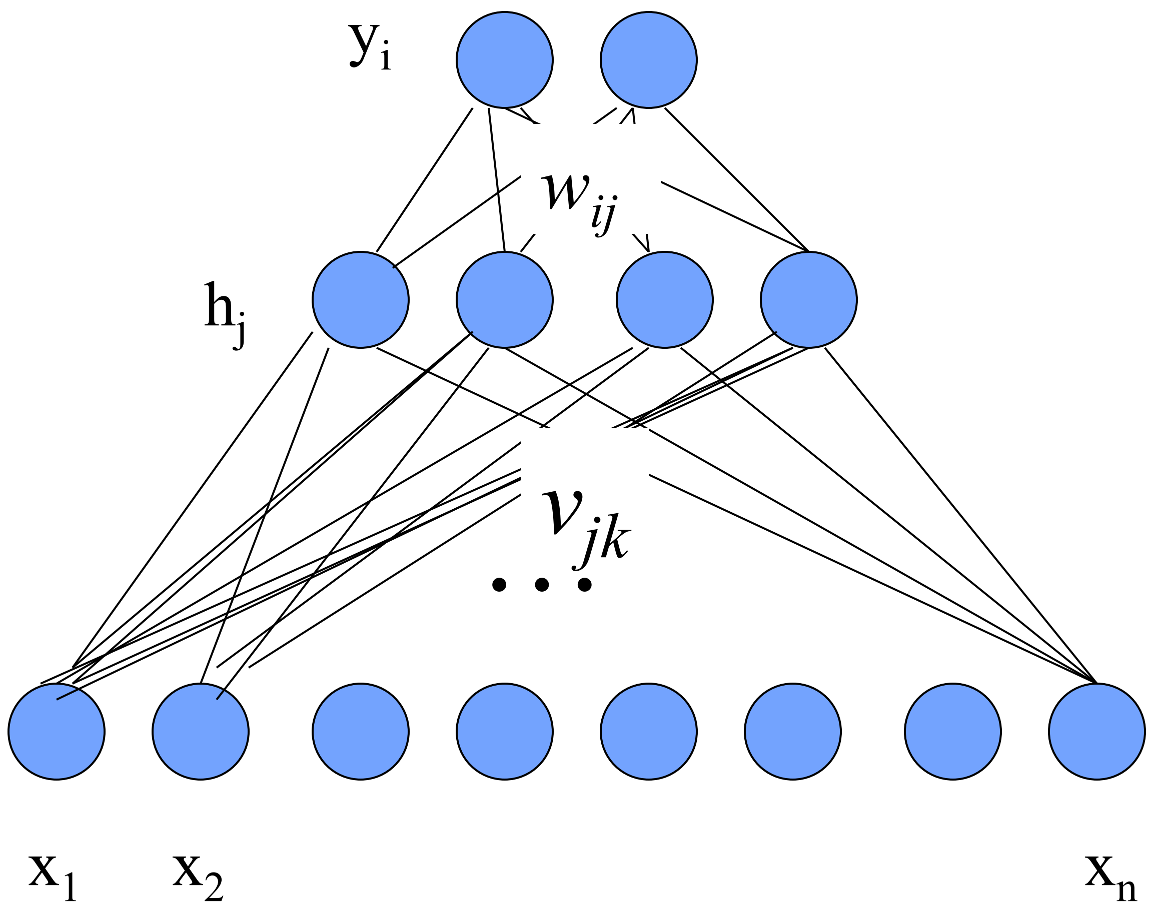
- Use a *nonlinear*, differentiable activation function
  - Such as a sigmoid $$f \equiv \frac{1}{1+\exp(-\rho h)} \quad \text{where} \quad h \equiv \sum_j w_{ij}\xi_j$$

    $$[\, f' = \rho f(1-f) \,]$$

- Use a multilayer feedforward network
  - Outputs are differentiable functions of the inputs

- Result: Can propagate credit/blame back to internal nodes
  - Chain rule (calculus) gives Δw_{ij} for internal "hidden" nodes
  - Based on gradient-descent learning

Backpropagation



Backpropagation

- When a learning pattern is clamped, the activation values are propagated to the output units, and the actual network output is compared with the desired output values, we usually end up with an error in each of the output units. Let's call this error e_o for a particular output unit o. We have to bring e_o to zero.

Backpropagation

- Remark:

  Generally, there are two modes of learning/training to choose from: on-line and batch.

  In on-line training, each propagation is followed immediately by a weight update.

  In batch training, many propagations occur before updating the weights.

Backpropagation

- The simplest method to do this is the greedy method: we strive to change the connections in the neural network in such a way that, next time around, the error e_o will be zero for this particular pattern. We know from the delta rule that, in order to reduce an error, we have to adapt its incoming weights according to the equation:

$$\Delta w_{\mathrm{ij}} = -\eta \; \partial E \,/\, \partial w_{\mathrm{ij}}$$

Backpropagation

- In order to adapt the weights from input to hidden units, we again want to apply the delta rule. In this case, however, we do not have a value for the hidden units.

Backpropagation

- Calculate the activation of the hidden units

$$h_j = f\left(\sum_{k=0}^{n} v_{jk} x_k\right)$$

Backpropagation

- And the activation of the output units

$$y_i = f\left(\sum_{j=0} w_{ij} h_j\right)$$

Backpropagation

- If we have μ pattern to learn (μ is from 1 or more training patterns – batch training), the error is

$$E = \tfrac{1}{2}\sum_{\mu}\sum_{i}\left(t_i^{\mu} - y_i^{\mu}\right)^2 =$$

(t_i is target output for output unit i)

$$= \tfrac{1}{2}\sum_{\mu}\sum_{i}\left[t_i^{\mu} - f\left(\sum_{j} w_{ij} h_j^{\mu}\right)\right]^2$$

$$= \tfrac{1}{2}\sum_{\mu}\sum_{i}\left[t_i^{\mu} - f\left(\sum_{j} w_{ij} f\left(\sum_{k=0}^{n} v_{jk} x_k^{\mu}\right)\right)\right]^2$$

Backpropagation

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} =$$

$$= \eta \sum_{\mu} \left(t_i^{\mu} - y_i^{\mu} \right) \dot{f}'\left(A_i^{\mu} \right) h_j^{\mu} =$$

$$= \eta \sum_{\mu} \delta_i^{\mu} h_j^{\mu}$$

where A_i is the activation (weighted sum of inputs) of output unit i, and

$$\delta_i^{\mu} = \left(t_i^{\mu} - y_i^{\mu} \right) \dot{f}'\left(A_i^{\mu} \right)$$

Backpropagation

$$\Delta v_{jk} = -\eta \frac{\partial E}{\partial v_{jk}} = -\eta \sum_{\mu} \frac{\partial E}{\partial h_j^{\mu}} \frac{\partial h_j^{\mu}}{\partial v_{jk}} =$$

$$= \eta \sum_{\mu} \sum_{i} \left(t_i^{\mu} - y_i^{\mu}\right) f'\left(A_i^{\mu}\right) w_{ij} f'\left(A_j^{\mu}\right) x_k^{\mu} =$$

$$= \eta \sum_{\mu} \sum_{i} \delta_i^{\mu} w_{ij} f'\left(A_j^{\mu}\right) x_k^{\mu}$$

where A_j is the activation (weighted sum of inputs) of hidden unit j.

Backpropagation

- The weight correction is given by :

$$\Delta w_{mn} = \eta \sum_{\mu} \delta_m^{\mu} x_n^{\mu}$$

where

$$\delta_m^{\mu} = \left(t_m^{\mu} - y_m^{\mu}\right) f'\left(A_m^{\mu}\right) \quad \text{if m is the output layer}$$

or

$$\delta_m^{\mu} = f'\left(A_m^{\mu}\right) \sum_{s} w_{sm} \delta_s^{\mu} \quad \text{if m is a hidden layer}$$

(where *s* runs through all output units)

Backpropagation

- For $f(x) = \dfrac{1}{1+\exp(-\rho x)}$, we have $f'(x) = \rho f(x)\,(1-f(x))$

Therefore, if m is the output layer

$$\delta_m^\mu = \left(t_m^\mu - y_m^\mu\right) f'\left(A_m^\mu\right) = \left(t_m^\mu - y_m^\mu\right)\rho y_m^\mu (1 - y_m^\mu)$$

and if m is a hidden layer

$$\delta_m^\mu = f'\left(A_m^\mu\right)\sum_s w_{sm}\delta_s^\mu = \rho h_m^\mu (1 - h_m^\mu)\sum_s w_{sm}\delta_s^\mu$$

(where *s* runs through all output units)

Backpropagation

- For example, if (1) $f(x) = \dfrac{1}{1+\exp(-x)}$ (that is, when $\rho = 1$)

  and (2) μ is from a training batch containing only one training pattern (i.e. now like <u>online training</u>)

Then, if m is the output layer

$$\delta_m = \left(t_m - y_m\right) y_m (1 - y_m)$$

and if m is an hidden layer

$$\delta_m = h_m (1 - h_m) \sum_s w_{sm} \delta_s$$

So,

$$\Delta w_{mn} = \eta \sum_\mu \delta_m^\mu x_n^\mu = \eta \delta_m x_n$$

and the new weight $w_{mn} = w_{mn} + \Delta w_{mn}$

Backpropagation Algorithm

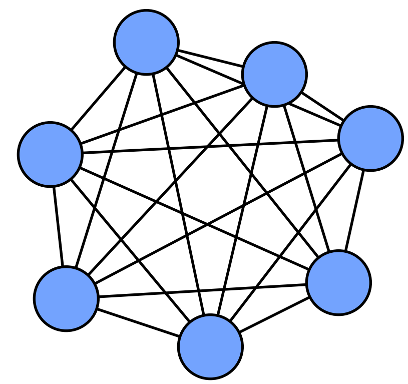
initialize network weights (often small random values)
  do
    for each batch of training patterns //*on-line if only 1 pattern/batch*
      compute error E at the output units
      compute Δw_{ij} for all weights from hidden layer to output layer
        // backward pass
      compute Δv_{jk} for all weights from input layer to hidden layer
        // backward pass continued
      $w_{ij} = w_{ij} + \Delta w_{ij}$ and $v_{jk} = v_{jk} + \Delta v_{jk}$
        //update network weights
  until E is less than the target error
return the network

Backpropagation

- Can be extended to arbitrary number of layers but three is most commonly used
- Can approximate arbitrary functions: crucial issues are
  - generalization to examples not in test data set
  - number of hidden units
  - number of samples
  - speed of convergence to a stable set of weights (sometimes a momentum term $\alpha \, \Delta w_{pq}$ is added to the learning rule to speed up learning)

Hopfield networks

- Act as "autoassociative" memories to store patterns
  - McCulloch-Pitts neurons with outputs -1 or 1, and threshold Θ
  - All neurons connected to each other
    - Symmetric weights ($w_{ij} = w_{ji}$) and $w_{ii} = 0$
  - Asynchronous updating of outputs
    - Let s_i be the state of unit i
    - At each time step, pick a random unit
    - Set s_i to 1 if $\Sigma_j w_{ij} s_j \geq \Theta_i$; otherwise, set s_i to -1



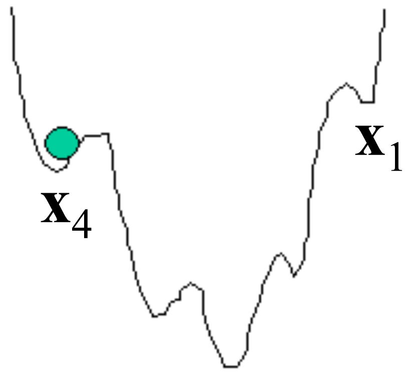
completely connected

Hopfield networks

- Hopfield showed that asynchronous updating in symmetric networks minimizes an "energy" function and leads to a stable final state for a given initial state
- Define an energy function (analogous to the gradient descent error function)
  - $E = -1/2 \, \Sigma_{i,j} \, w_{ij} \, s_i \, s_j + \Sigma_i \, s_i \, \Theta_i$
- Suppose a random unit i was updated: E always decreases!
  - If s_i is initially –1 and $\Sigma_j \, w_{ij} \, s_j > \Theta_i$, then s_i becomes +1
    - Change in $E = -1/2 \, \Sigma_j \, (w_{ij} \, s_j + w_{ji} \, s_j) + \Theta_i = - \Sigma_j \, w_{ij} \, s_j + \Theta_i < 0$!!
  - If s_i is initially +1 and $\Sigma_j \, w_{ij} \, s_j < \Theta_i$, then s_i becomes -1
    - Change in $E = 1/2 \, \Sigma_j \, (w_{ij} \, s_j + w_{ji} \, s_j) - \Theta_i = \Sigma_j \, w_{ij} \, s_j - \Theta_i < 0$!!

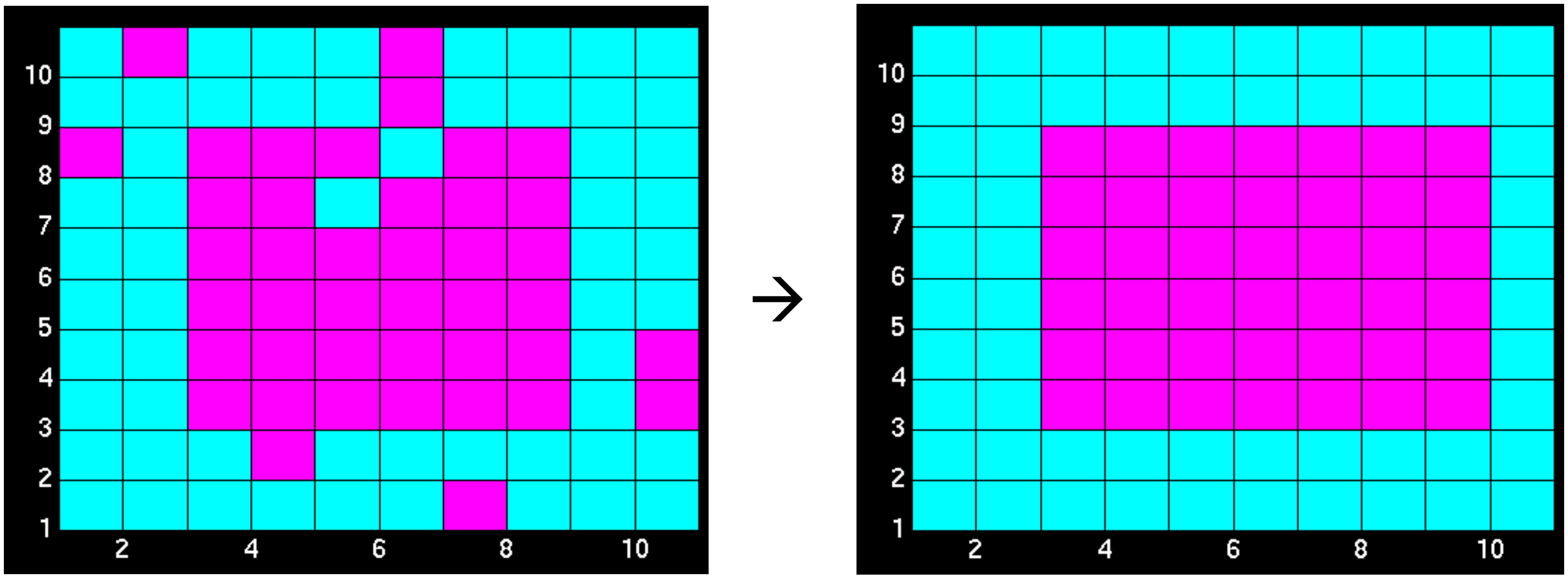
Hopfield networks

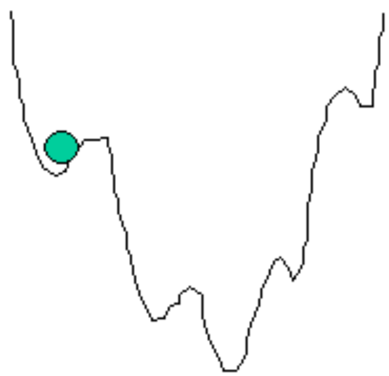
- Note: Network converges to local minima which store different patterns.



- Store p N-dimensional pattern vectors $\mathbf{x}_1, \ldots, \mathbf{x}_p$ using Hebbian learning rule:
  - $w_{ji} = 1/N \sum_{m=1,..,p} x_{m,j} \, x_{m,i}$ for all $j \neq i$; 0 for $j = i$
  - $W = 1/N \sum_{m=1,..,p} \mathbf{x}_m \, \mathbf{x}_m^T$ (outer product of vectors; diagonal zero)
    - T denotes vector transpose

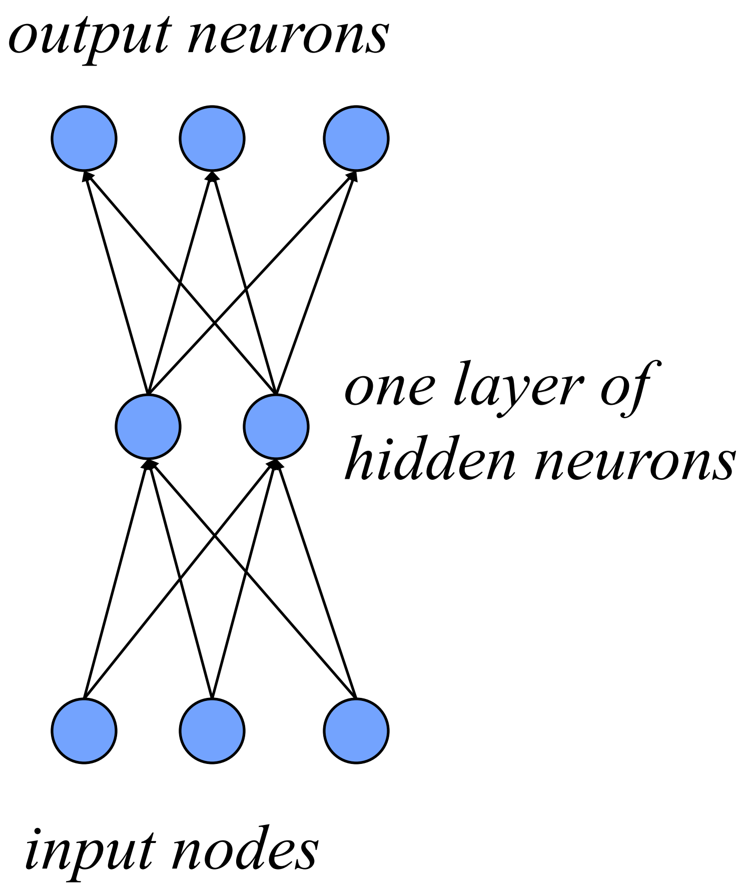
Pattern Completion in a Hopfield Network





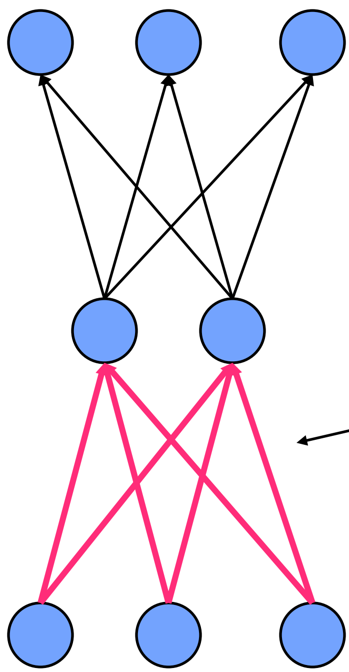
Local minimum
("attractor")
of energy function
stores pattern

Radial Basis Function Networks



Radial Basis Function Networks

output neurons



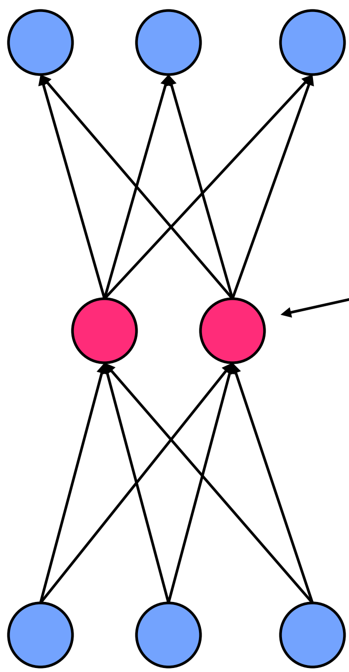
input nodes

propagation function:

$$a_j = \sqrt{\sum_{i=1}^{n} (x_i - \mu_{i,j})^2}$$

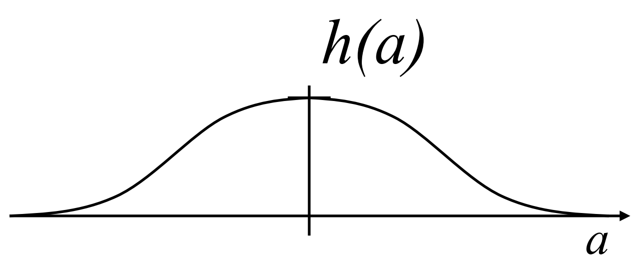
Radial Basis Function Networks

output neurons



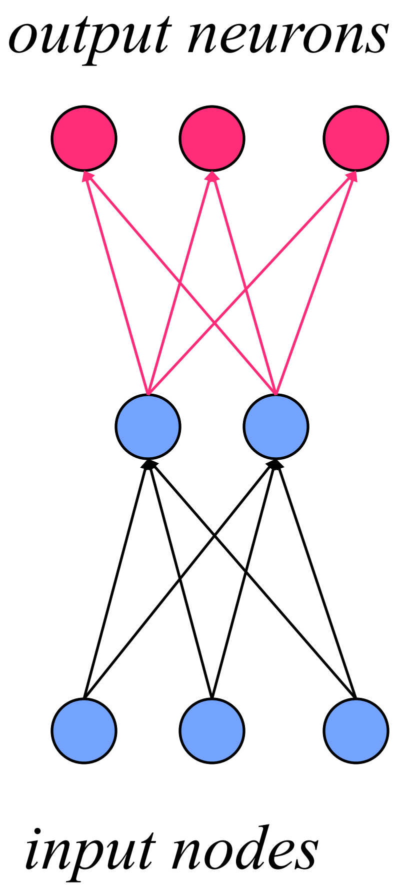
input nodes

output function:
(Gauss' bell-shaped function)



$$h(a) = e^{-\frac{a^2}{2\sigma^2}}$$

Radial Basis Function Networks



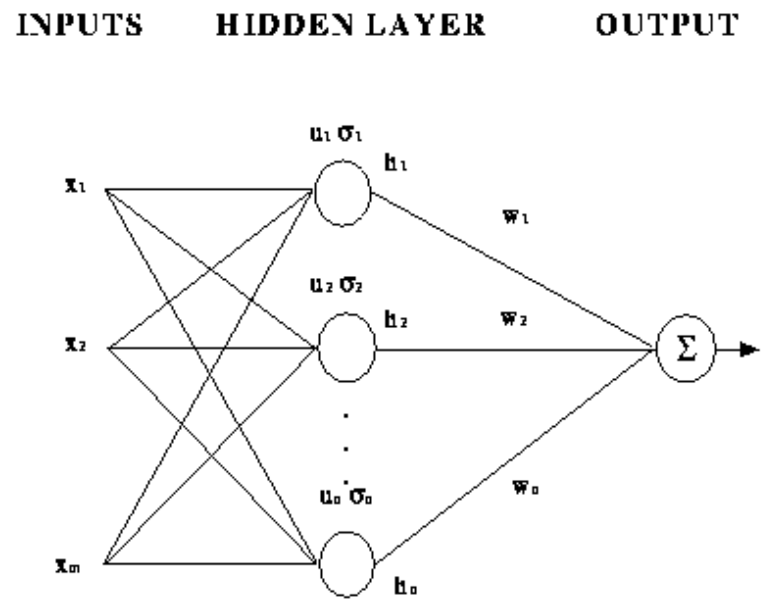
output of network:

$$\text{out}_j = \sum_i w_{i,j} h_i$$

RBF networks

- Radial basis functions
  - Hidden units store means and variances
  - Hidden units compute a Gaussian function of inputs $x_1, \ldots x_n$ that constitute the input vector $\mathbf{x}$
- Learn weights w_i, means μ_i, and variances σ_i by minimizing squared error function (gradient descent learning)

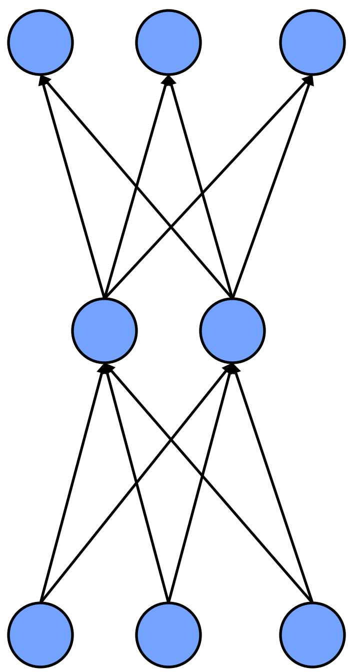
Radial Basis Function Network



$$h_i = exp[-\frac{(\mathbf{x} - \mathbf{u}_i)^{\mathbf{T}}(\mathbf{x} - \mathbf{u}_i)}{2\sigma^2}], \; y = \sum_i h_i w_i$$

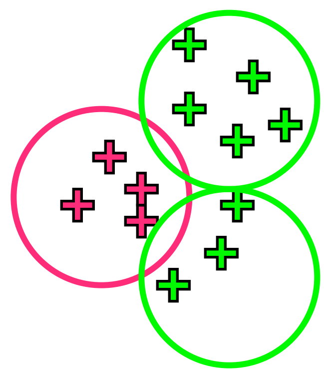
RBF Networks and Multilayer Perceptrons

output neurons

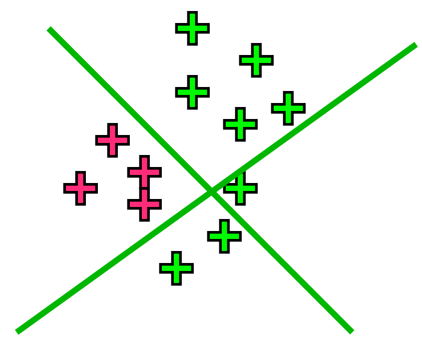


input nodes

RBF:



MLP:



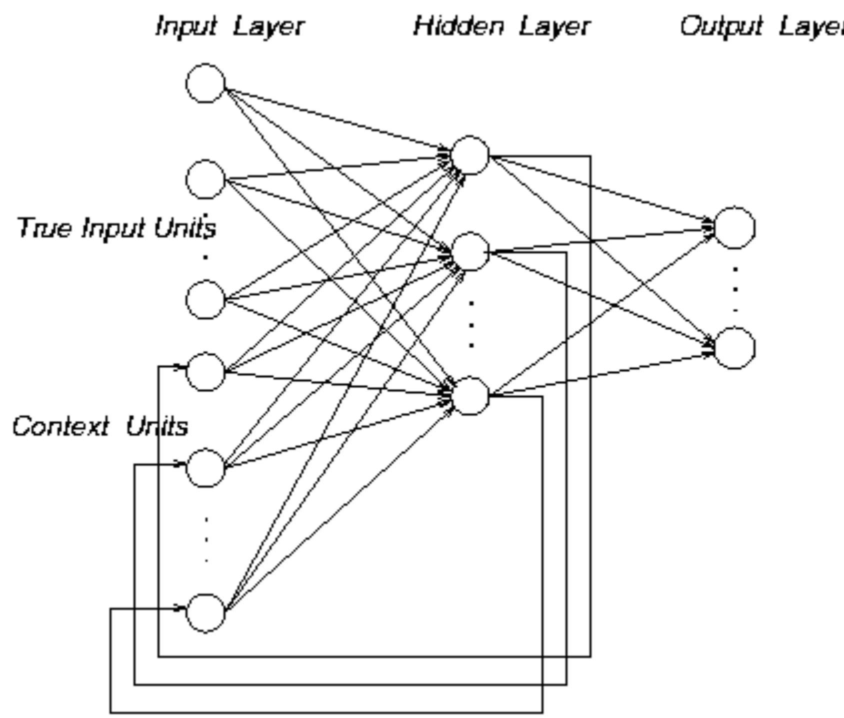
Recurrent networks

- Employ feedback (positive, negative, or both)
  - Not necessarily stable
    - Symmetric connections can ensure stability
- Why use recurrent networks?
  - Can learn temporal patterns (time series or oscillations)
  - Biologically realistic
    - Majority of connections to neurons in cerebral cortex are feedback connections from local or distant neurons
- Examples
  - Hopfield network
  - Boltzmann machine (Hopfield-like net with input & output units)
  - Recurrent backpropagation networks: for small sequences, unfold network in time dimension and use backpropagation learning

Recurrent networks (con’t)

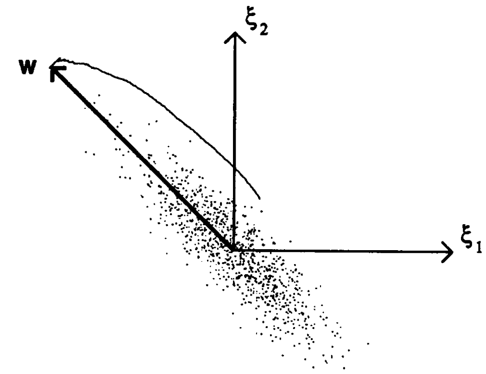
- Example
  - Elman networks
    - Partially recurrent
    - Context units keep internal memory of part inputs
    - Fixed context weights
    - Backpropagation for learning
    - E.g. Can disambiguate A→B→C and C→B→A

Elman network



Unsupervised Networks

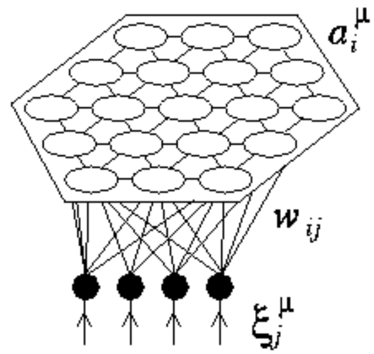
- No feedback to say how output differs from desired output (no error signal) or even whether output was right or wrong
- Network must discover patterns in the input data by itself
  - Only works if there are redundancies in the input data
  - Network self-organizes to find these redundancies
    - Clustering: Decide which group an input belongs to
      - Synaptic weights of one neuron represents one group
    - Principal Component Analysis: Finds the principal eigenvector of data covariance matrix
    - Hebb rule performs PCA! (Oja, 1982)
      - $\Delta w_i = \eta\, \xi_i y$
      - Output $y = \Sigma_i\, w_i\, \xi_i$



Self-Organizing Maps (Kohonen Maps)

- Feature maps
  - Competitive networks
  - Neurons have locations
  - For each input, winner is the unit with largest output
  - Weights of winner and nearby units modified to resemble input pattern
  - Nearby inputs are thus mapped topographically
- Biological relevance
  - Retinotopic map
  - Somatosensory map
  - Tonotopic map

Self Organized Map (SOM)



$$a_i^\mu = \begin{cases} 1 & \text{if } \; i = i^* \\ 0 & \text{if } \; i \neq i^* \end{cases}$$

$$h_i^\mu = \sum_j w_{ij} \xi_j \qquad h_{i^*}^\mu \geq h_i^\mu \;\text{ for all } i$$

Kohonen learning : $\Delta w_{ij} = \eta \Lambda(i,i^*)(\xi_j^\mu - w_{ij})$

Neighborhood function : $\Lambda(i,i^*) = e^{-\frac{|\mathbf{r}_i - \mathbf{r}_{i^*}|^2}{2\sigma^2}}$

Summary: Biology and Neural Networks

- So many similarities
  - Information is contained in synaptic connections
  - Network learns to perform specific functions
  - Network generalizes to new inputs
- But NNs are woefully inadequate compared with biology
  - Simplistic model of neuron and synapse, implausible learning rules
  - Hard to train large networks
  - Network construction (structure, learning rate etc.) is a heuristic art
- One obvious difference: Spike representation
  - Recent models explore spikes and spike-timing dependent plasticity
- Other Recent Trends: Probabilistic approach
  - NNs as Bayesian networks (allows principled derivation of dynamics, learning rules, and even structure of network)
  - Not clear how neurons encode probabilities in spikes

References on ANN and Stock Prediction

http://www.cs.berkeley.edu/~akar/IITK_website/EE671/report_stock.pdf

http://www.cs.ucsb.edu/~nanli/publications/stock_pattern.pdf

and the references in the papers above