

CS 4602

Introduction to Machine Learning

Neural Networks

Instructor: Po-Chih Kuo

Roadmap

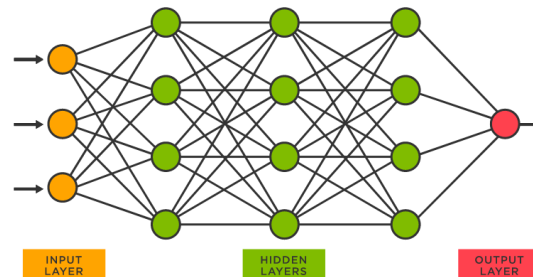
- Introduction and Basic Concepts
- Regression
- Bayesian Classifiers
- Decision Trees
- Linear Classifier
- Neural Networks
- Deep learning
- Convolutional Neural Networks
- Reinforcement Learning
- KNN
- Model Selection and Evaluation
- Clustering
- Data Exploration & Dimensionality reduction

Outline

- Motivation
- Multilayer perceptrons (MLP)
- Backpropagation
- Extension

Connectionism

- A computer modeling approach based upon the architecture of the brain
- Multiple, individual “**nodes**” or “**units**” that operate at the same time (in parallel)
- A **network** that connects the nodes together
- Information is stored in a distributed fashion among the **links** that connect the nodes
- **Learning** can occur with gradual changes in connection strength



Brains vs Computers



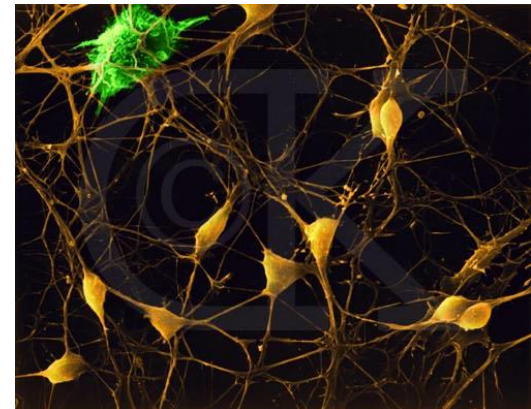
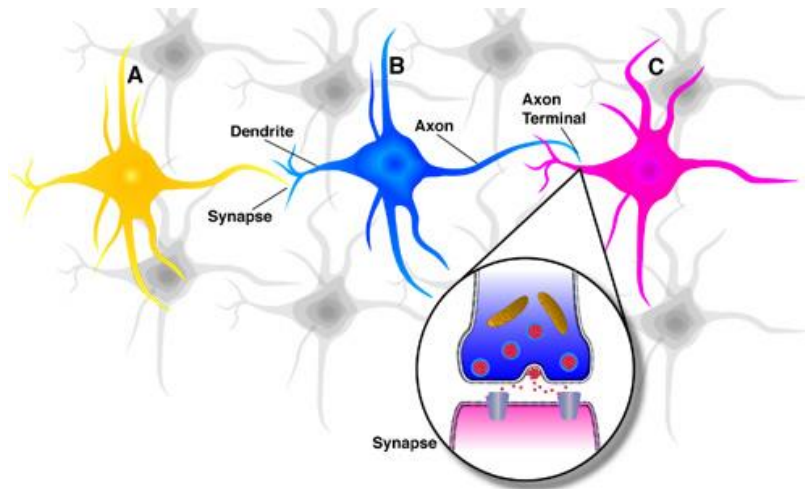
- 200 billion neurons, 32 trillion synapses
- Element size: 10^{-6} m
- Energy use: 25W
- Processing speed: 100 Hz
- Parallel, Distributed
- Fault Tolerant
- Learns: Yes
- Conscious: Usually



- 1 billion bytes RAM but trillions of bytes on disk
- Element size: 10^{-9} m
- Energy watt: 30-90W (CPU)
- Processing speed: 10^9 Hz
- Serial, Centralized
- Generally not Fault Tolerant
- Learns: Some
- Conscious: Generally No

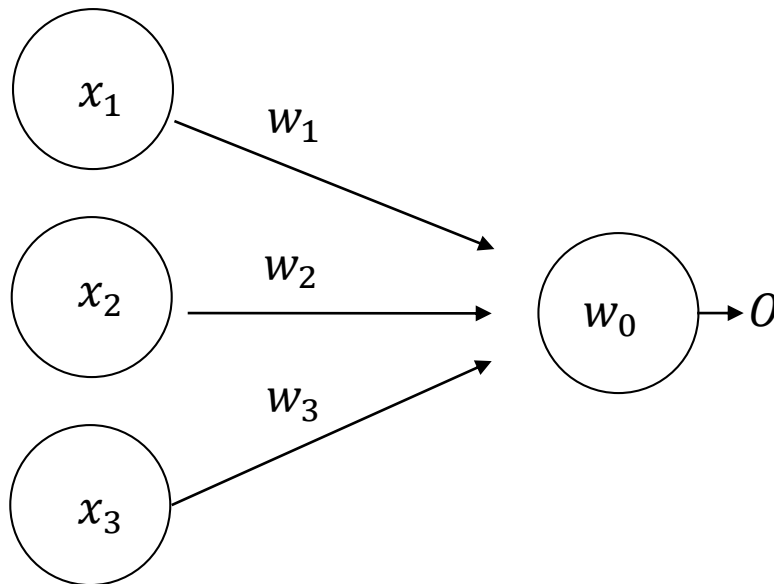
Neurons in the Brain

- A neuron receives input from other neurons (generally thousands) from its synapses
- Inputs are approximately summed
- When the input exceeds a threshold the neuron sends an electrical spike that travels that travels from the body, down the axon, to the next neuron(s)



Perceptron

- Initial proposal of connectionist networks
- Essentially a linear discriminant composed of nodes, weights

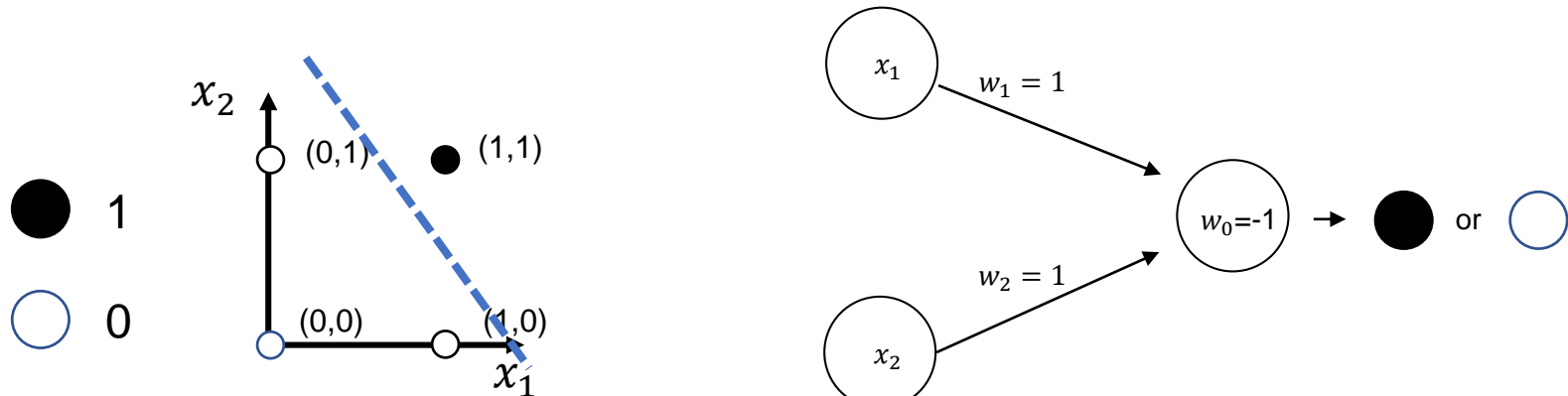


Activation Function

$$O = \begin{cases} 1: \left(\sum_i w_i x_i \right) + w_0 > 0 \\ 0: \text{otherwise} \end{cases}$$

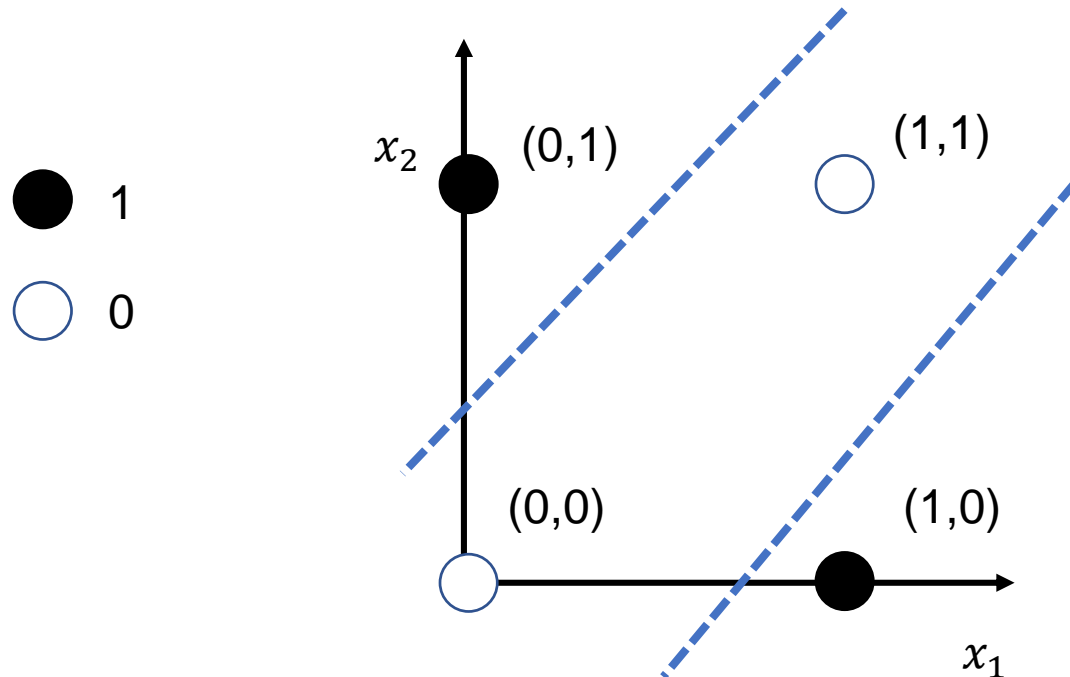
Linear separability

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



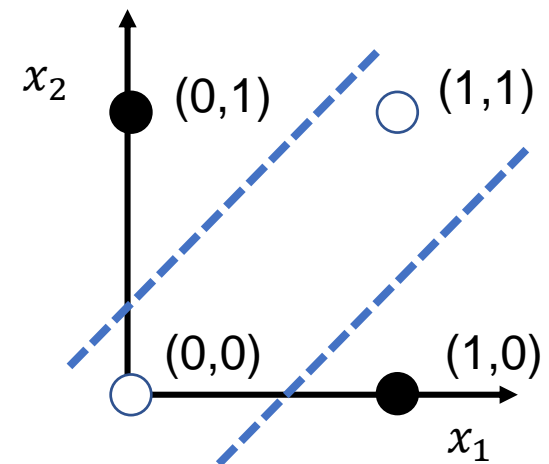
Linear inseparability

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



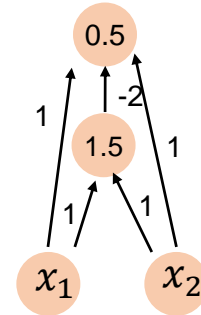
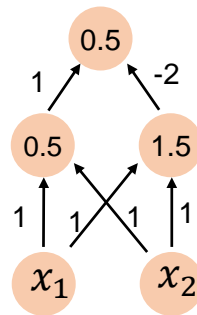
Solution in 1980s: Multilayer perceptron

- Removes many limitations of single-layer networks
 - Can solve XOR
- How to Draw a two-layer perceptron that computes the XOR function?
 - 2 binary inputs x_1 and x_2
 - 1 binary output
 - One “hidden” layer
 - Find the appropriate weights and threshold



Multilayer perceptron

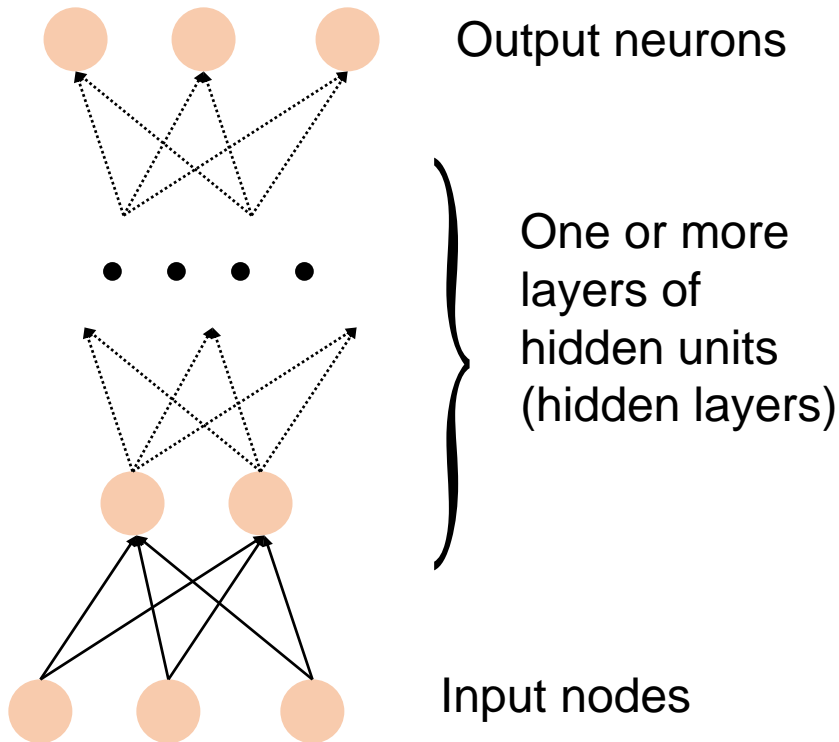
- Examples of two-layer perceptrons that compute XOR



- E.g. Right side network
 - Output is 1
if and only if $x_1 + x_2 - 2(x_1 + x_2 - 1.5 > 0) - 0.5 > 0$

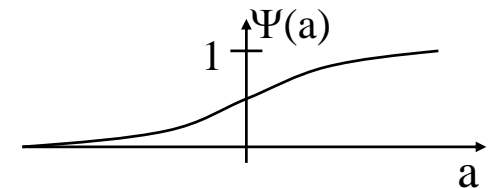
Multilayer perceptron

You'll see why later

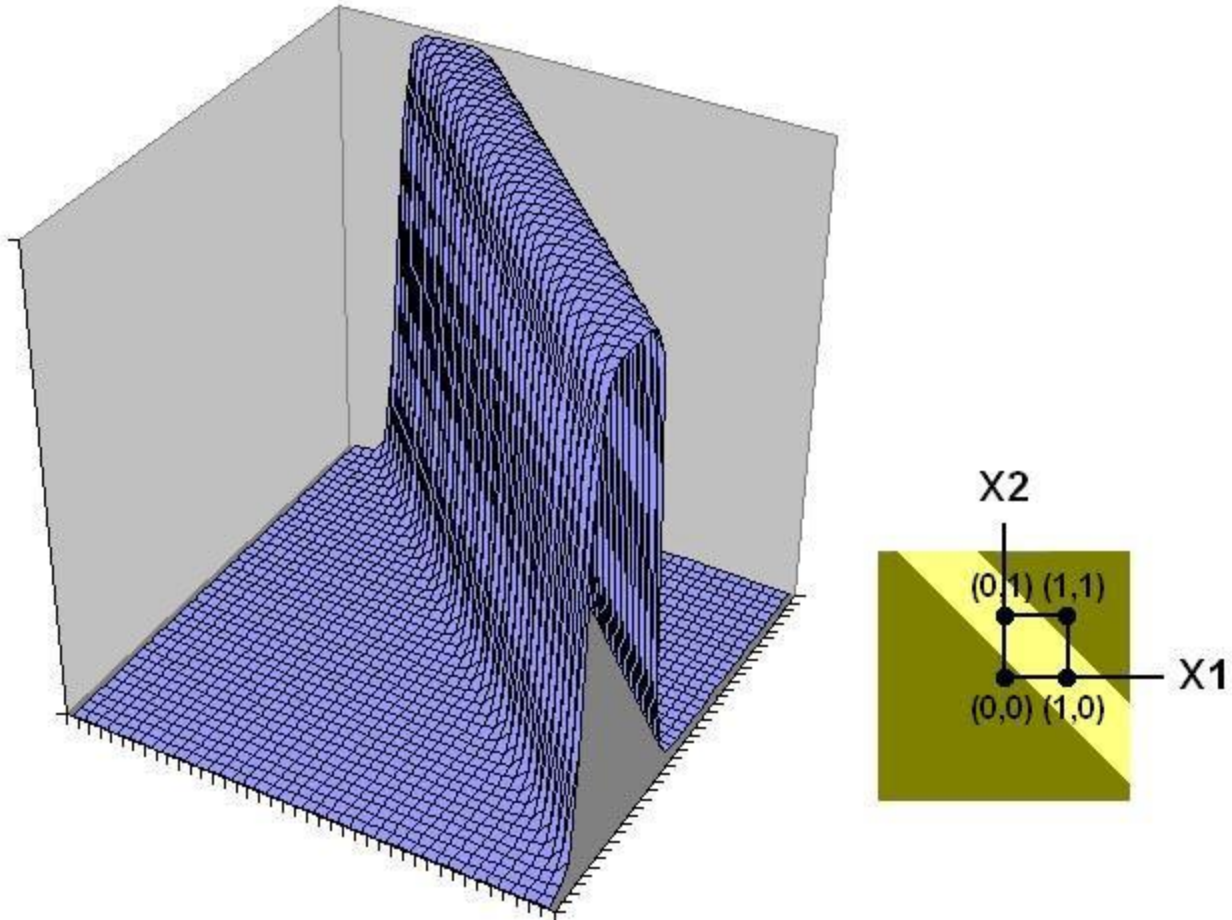


The most common output function (Sigmoid):

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

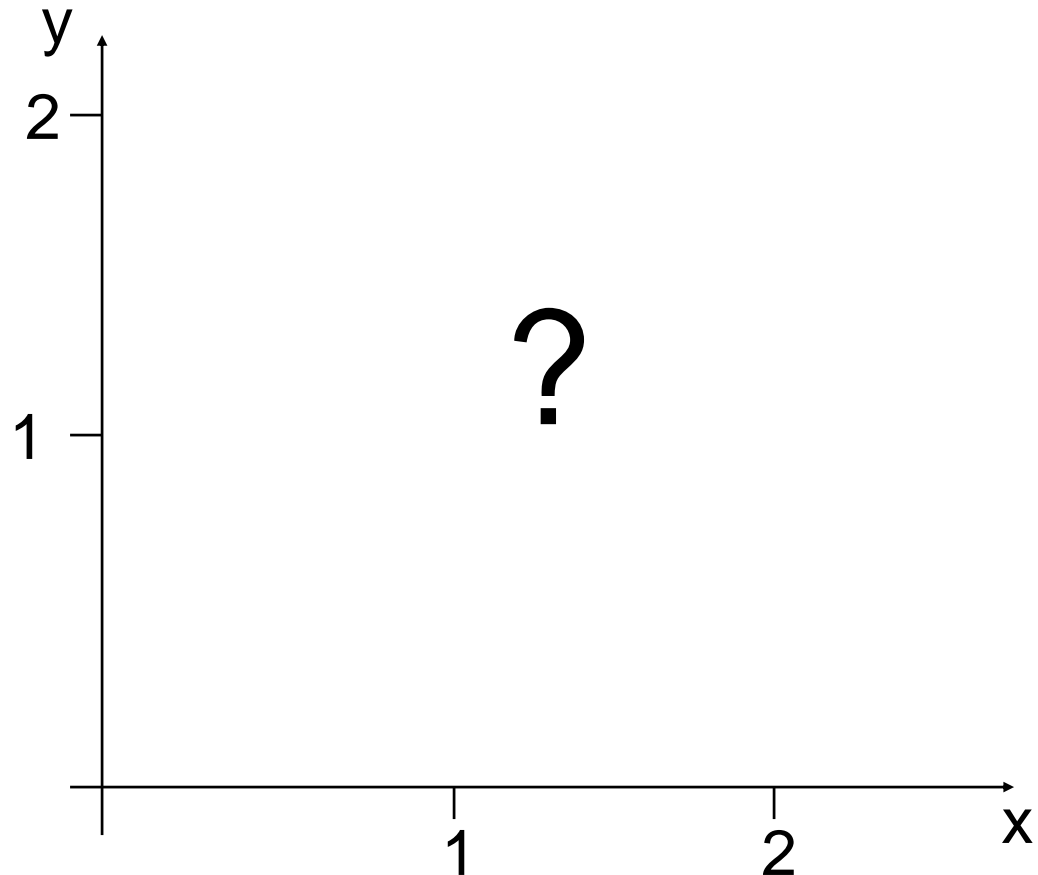
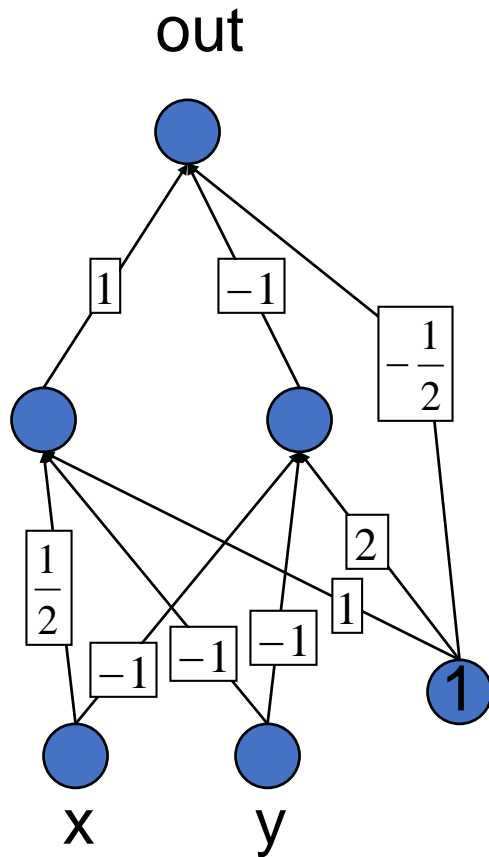


(non-linear function)

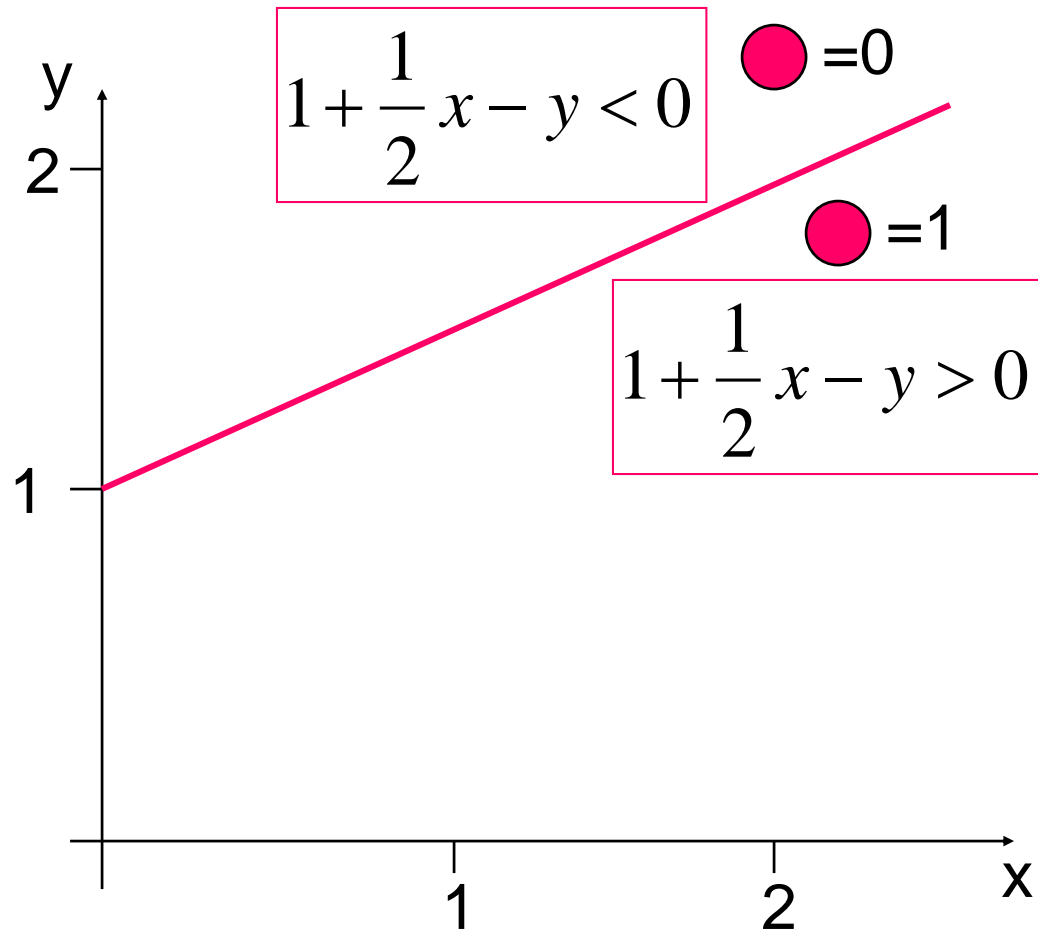
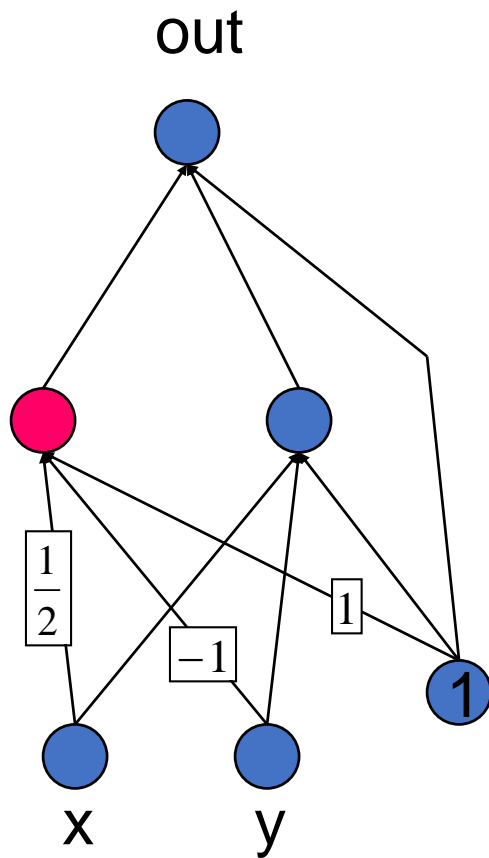


Source: <http://colinfahey.com/>

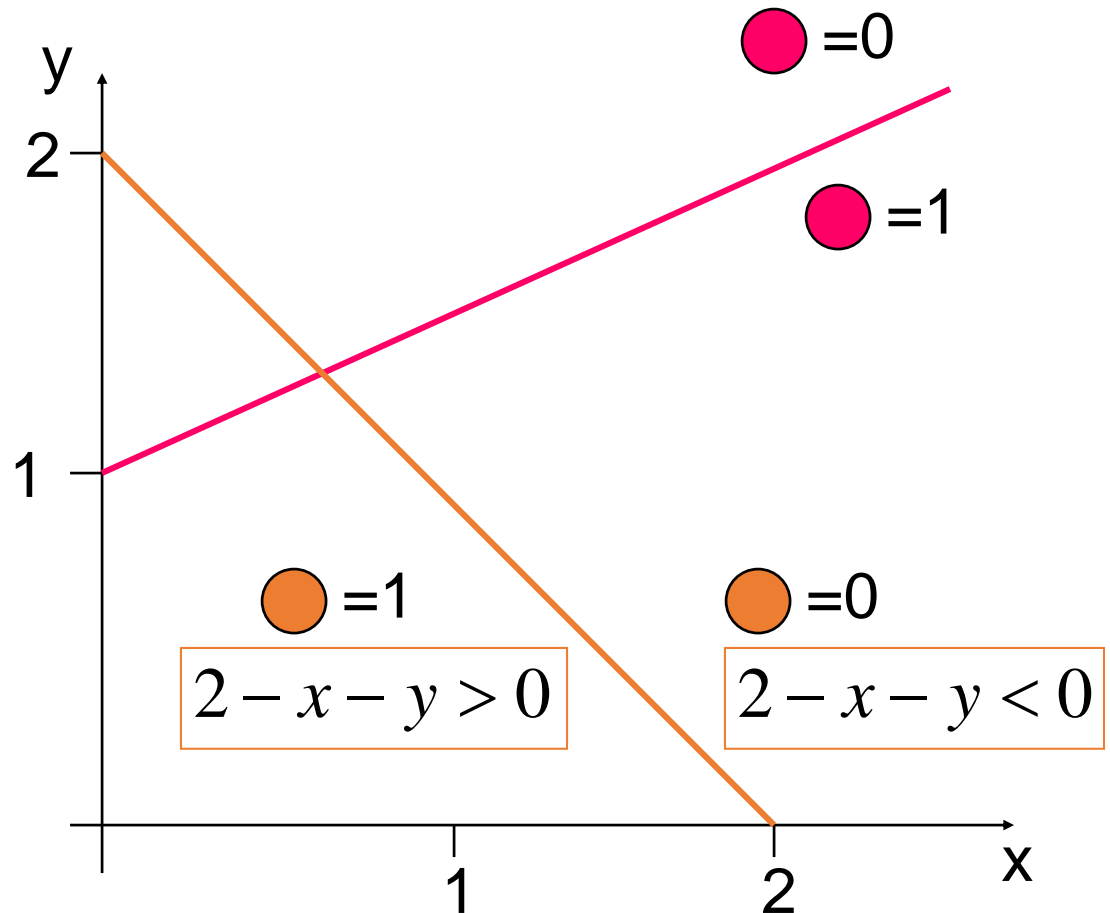
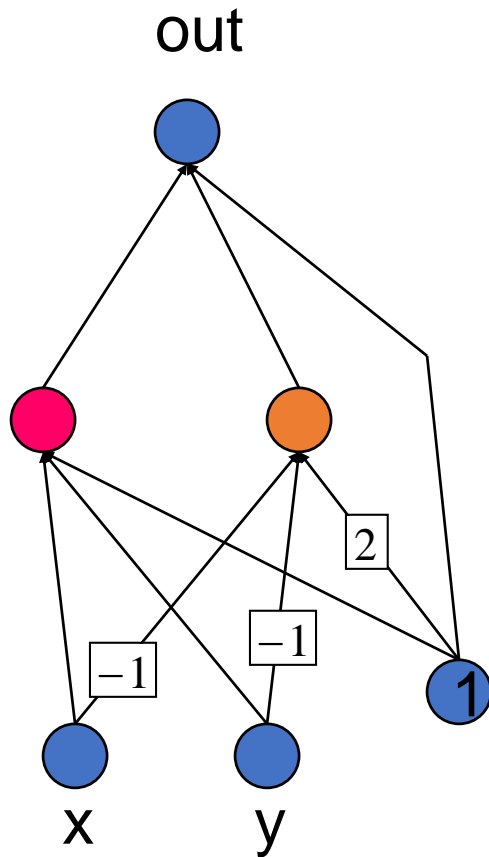
Perceptrons as Constraint Satisfaction Networks



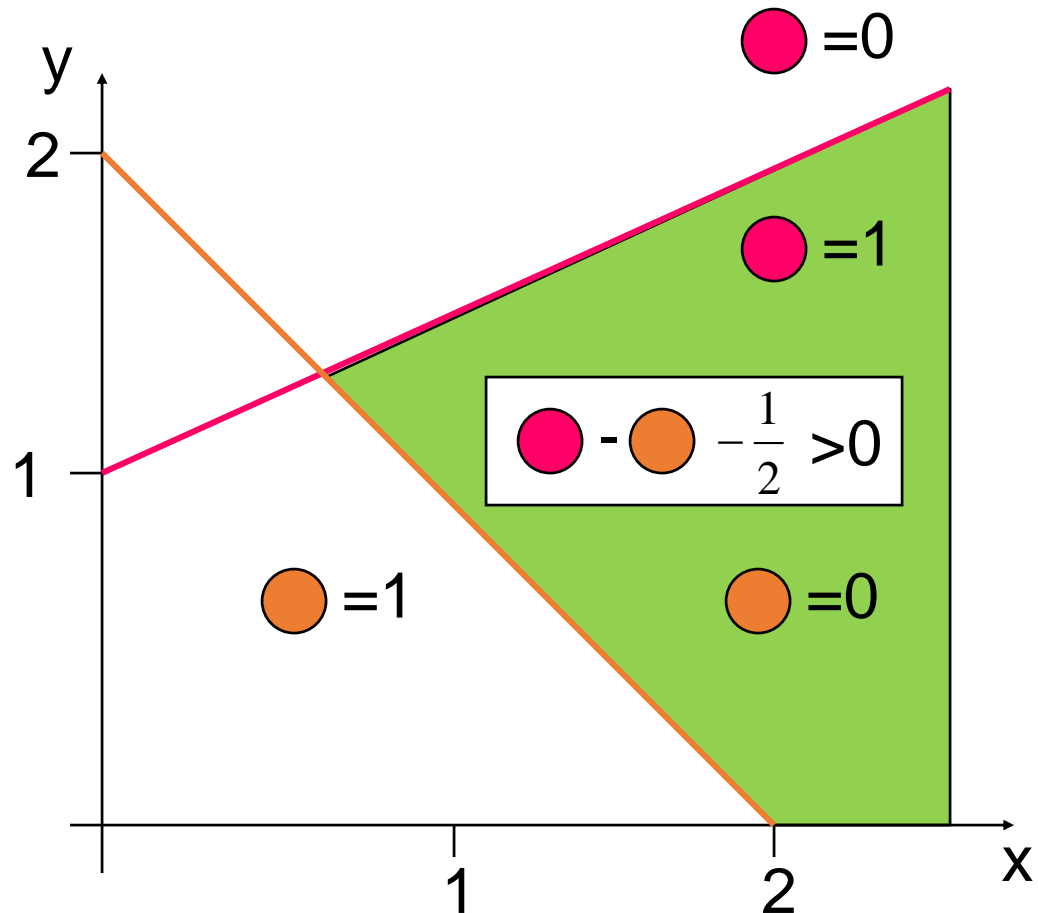
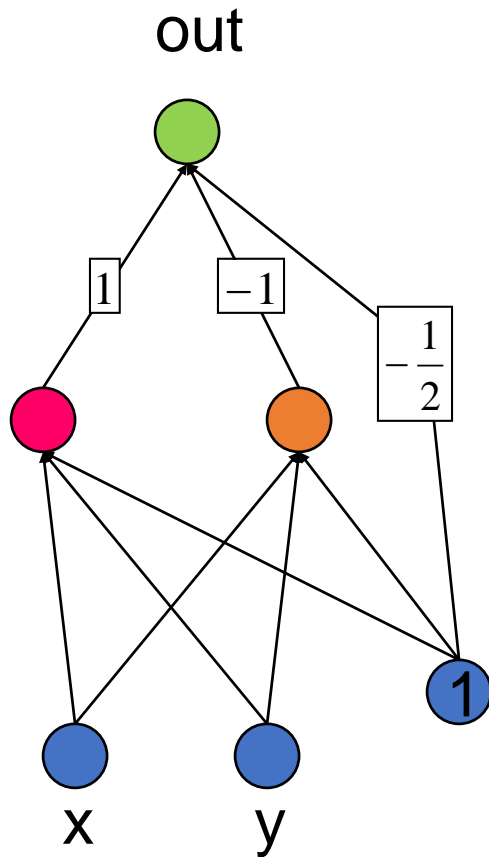
Perceptrons as Constraint Satisfaction Networks



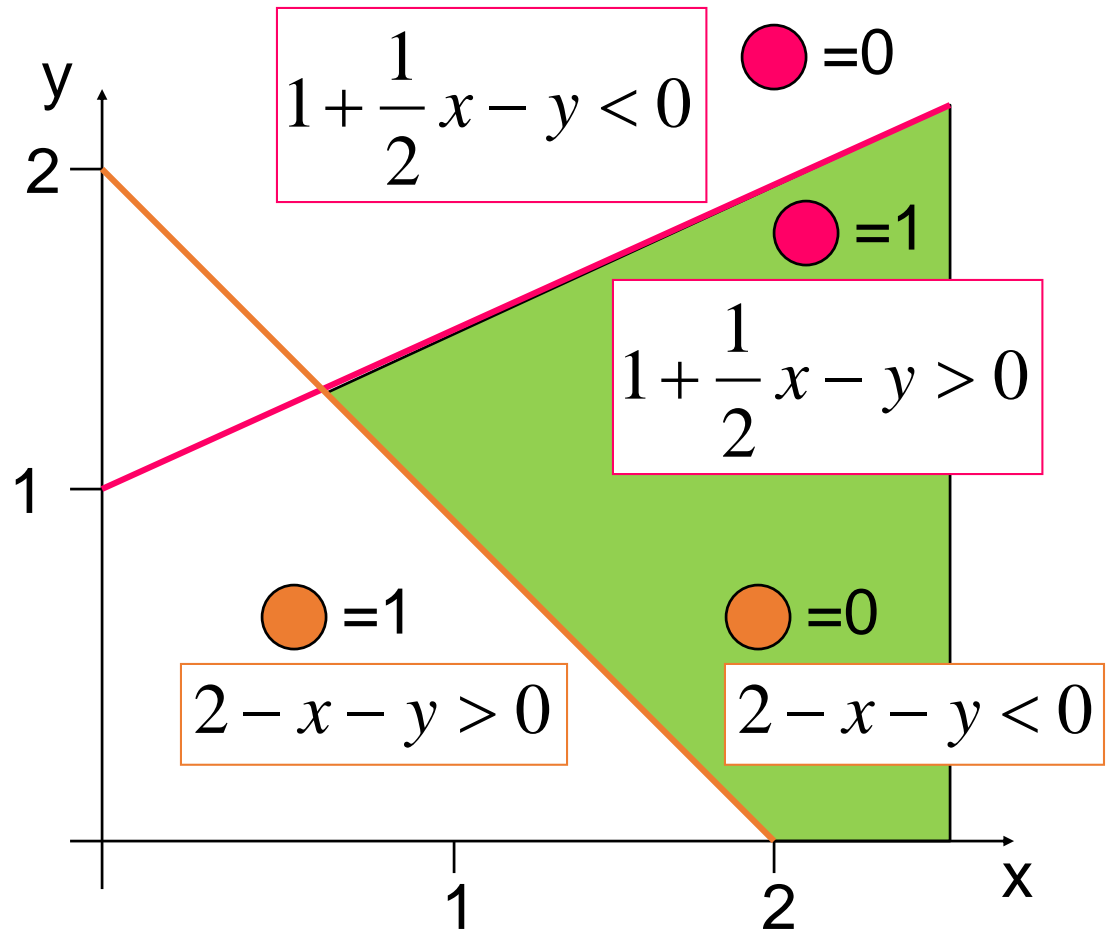
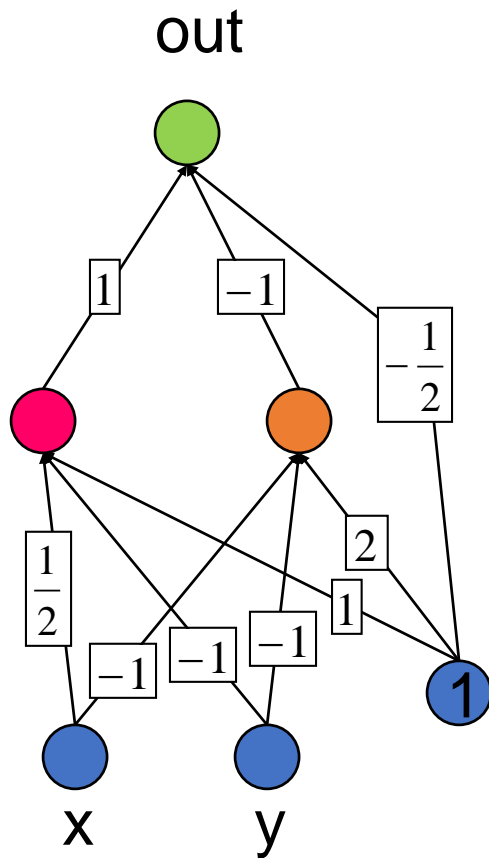
Perceptrons as Constraint Satisfaction Networks



Perceptrons as Constraint Satisfaction Networks

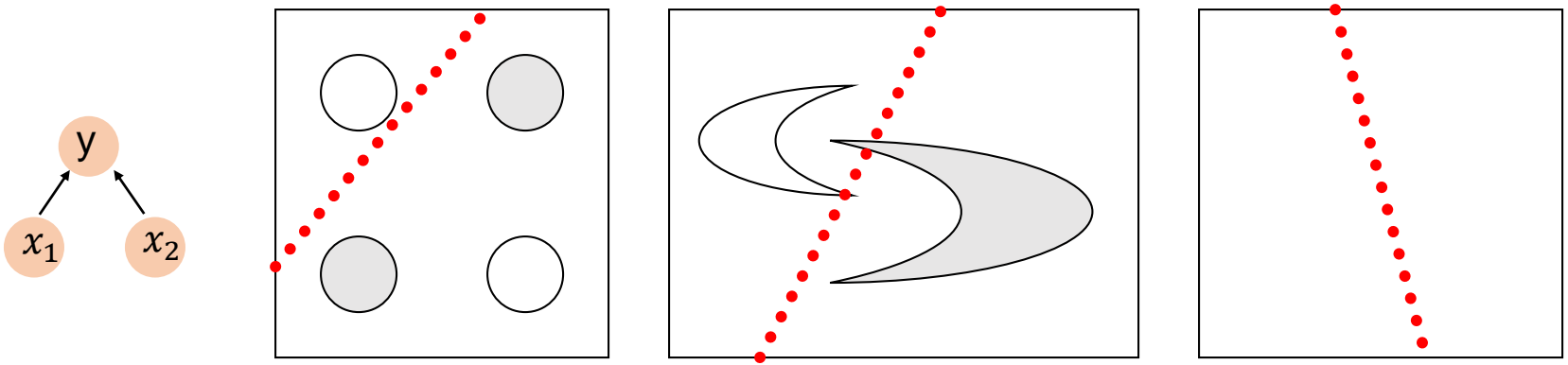


Perceptrons as Constraint Satisfaction Networks



Decision Boundary

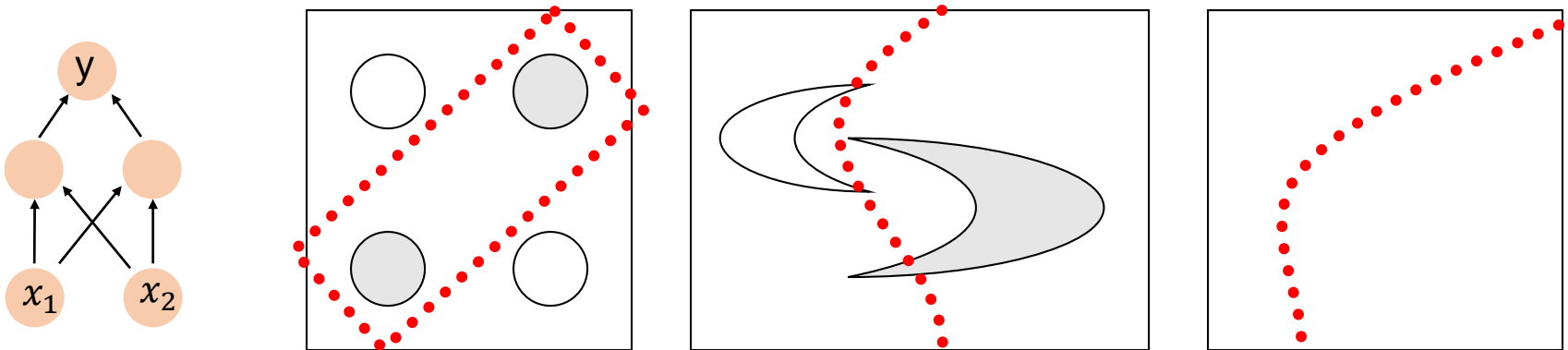
- 0 hidden layers: linear classifier
 - Hyperplanes



Example from to Eric Postma via Jason Eisner

Decision Boundary

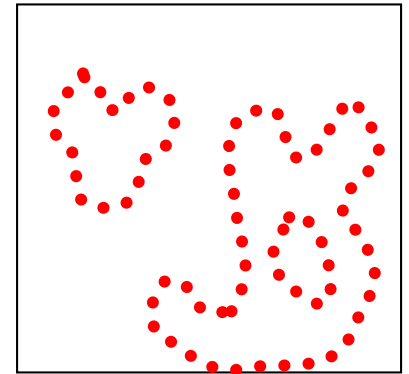
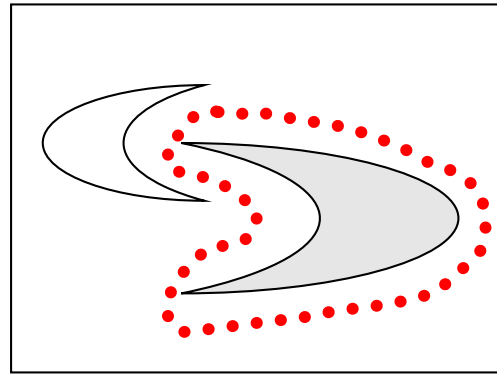
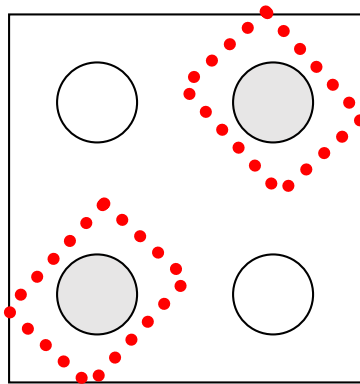
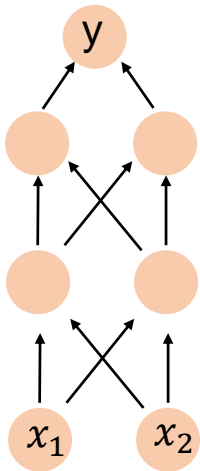
- 1 hidden layer
 - Boundary of convex region (open or closed)



Example from to Eric Postma via Jason Eisner

Decision Boundary

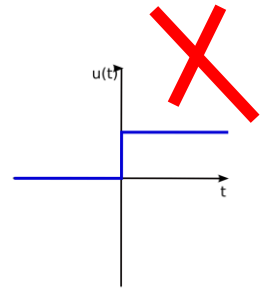
- 2 hidden layers
 - Combinations of convex regions



Example from to Eric Postma via Jason Eisner

Outline

- Motivation
- Multilayer perceptrons (MLP)
- **Backpropagation (BP)**
- RBF



Perceptron Alg. vs. Gradient Descent Rule

Perceptron learning rule guaranteed to succeed if

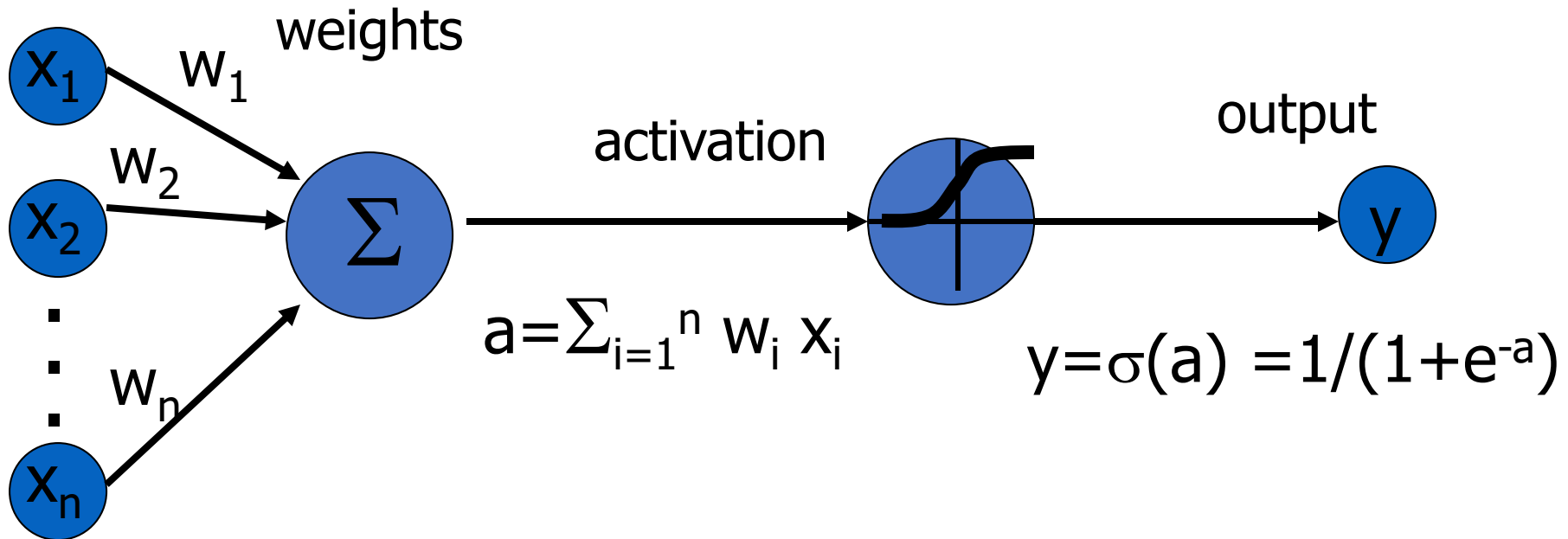
- Training examples are linearly separable
- Sufficiently small learning rate η

Linear unit training rules uses gradient descent

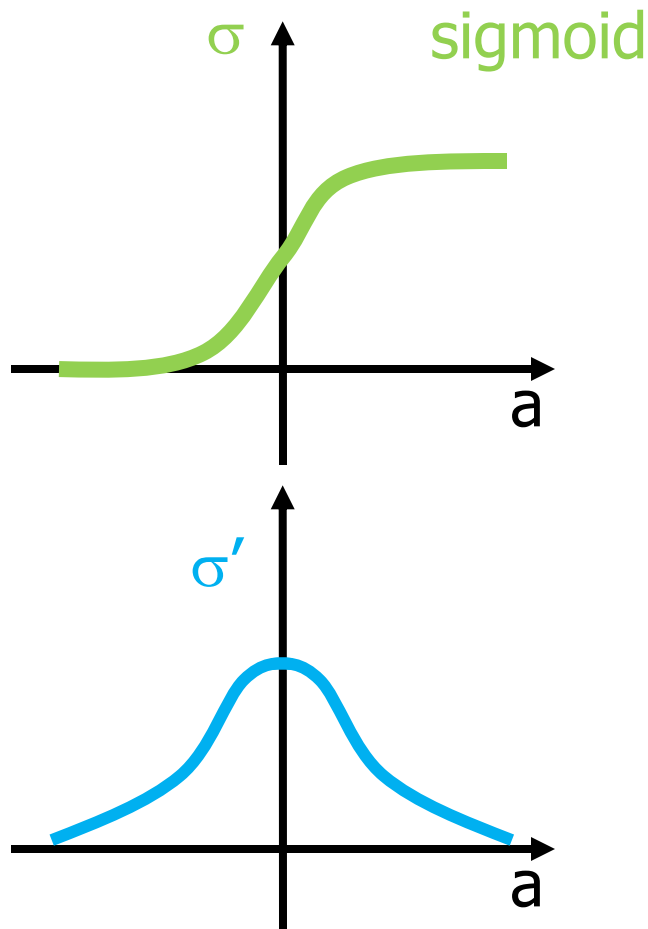
- Use a differentiable activation function
- Guaranteed to converge to hypothesis with minimum squared error
- Given sufficiently small learning rate η
- Even when training data contains noise
- Even when training data not separable by H

Neuron with Sigmoid Function

inputs



Gradient Descent for Sigmoid Function



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

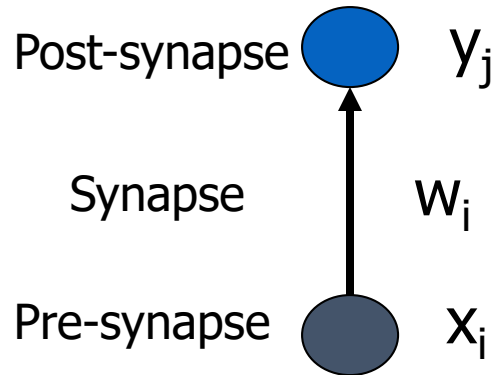
$$\begin{aligned} \frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} (t-y)^2 \\ &= \frac{\partial}{\partial w_i} \frac{1}{2} (t - \sigma(\sum_i w_i x_i))^2 \\ &= (t-y) \sigma'(\sum_i w_i x_i) (-x_i) \end{aligned}$$

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

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

$$\Delta w_i = \alpha y (1-y) (t-y) x_i$$

Gradient Descent Learning Rule



$$\Delta w_i = \alpha \, y_j(1-y_j) \, (t_j - y_j) \, x_i$$

learning rate

derivative of activation function

error of post-synaptic neuron

activation of pre-synaptic neuron

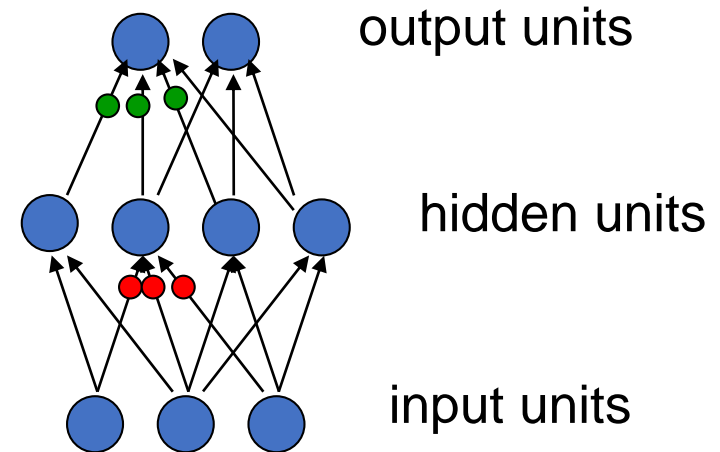
Learning with hidden units

- Networks without hidden units are very limited.
(More layers of linear units do not help. It's still linear.)
- We need multiple layers of adaptive non-linear hidden units. But how can we train such nets?
 - We need an efficient way of adapting **all** the weights, not just the last layer.
 - Learning the weights going into hidden units is equivalent to learning **features**.
 - It's hard to tell 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 to change one weight.
- Randomly perturb **all** the weights in parallel and correlate the performance gain with the weight changes.
 - 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**.

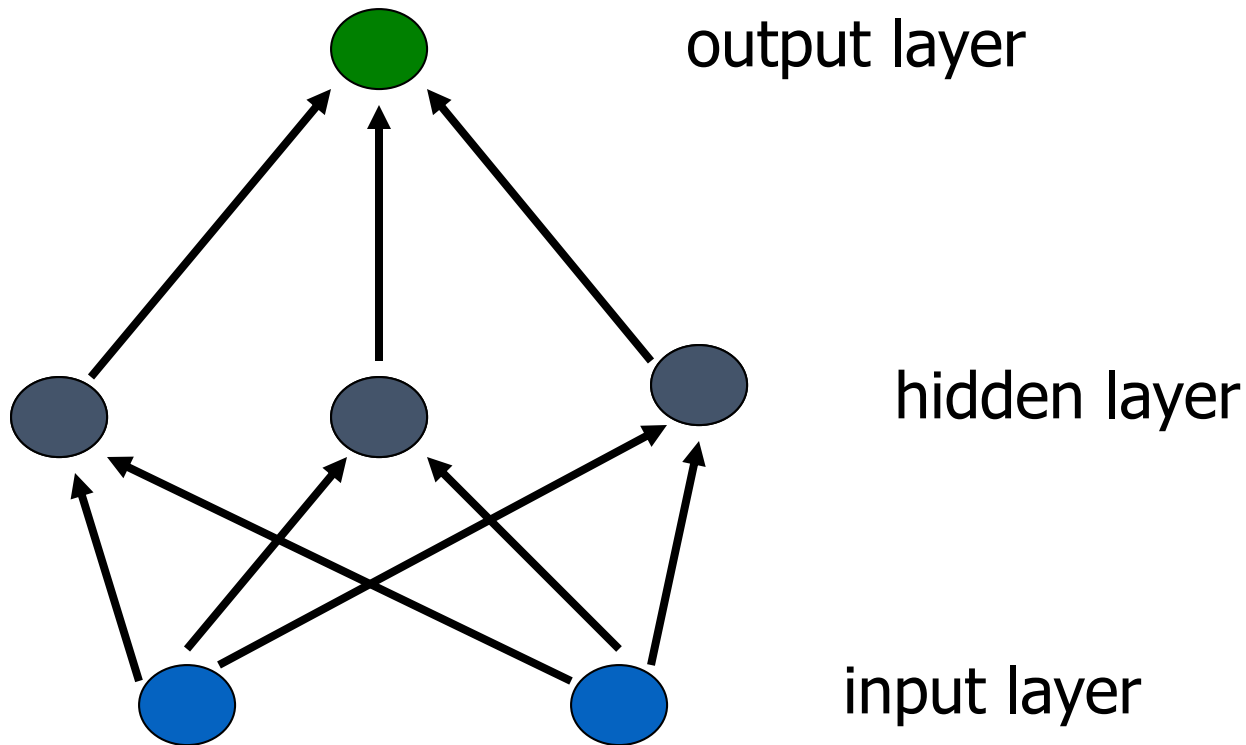


Learning the input to hidden weights is **hard**.

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.

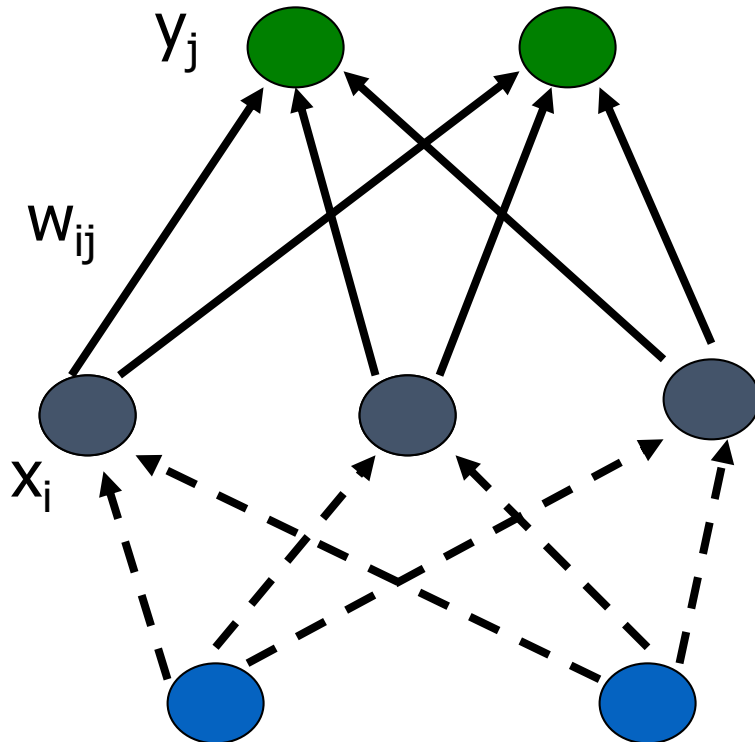
Multi-Layer Networks



Training Rule for Weights to the Output Layer

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

$$\begin{aligned} \partial E / \partial w_{ij} &= \partial / \partial w_{ij} \frac{1}{2} \sum_j (t_j - y_j)^2 \\ &= \dots \\ &= - y_j(1 - y_j)(t_j - y_j) x_i \end{aligned}$$

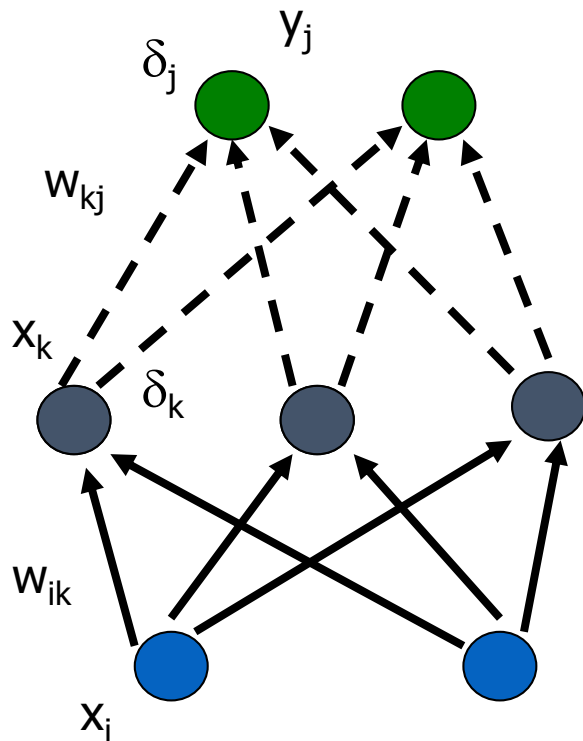


$$\begin{aligned} \Delta w_{ij} &= \alpha y_j(1 - y_j) (t_j - y_j) x_i \\ &= \alpha \delta_j x_i \end{aligned}$$

Derivative of activation function
Learning rate
Error of post-synaptic neuron
Activation of pre-synaptic neuron

$$\text{with } \delta_j := y_j(1 - y_j) (t_j - y_j)$$

Training Rule for Weights to the Hidden Layer



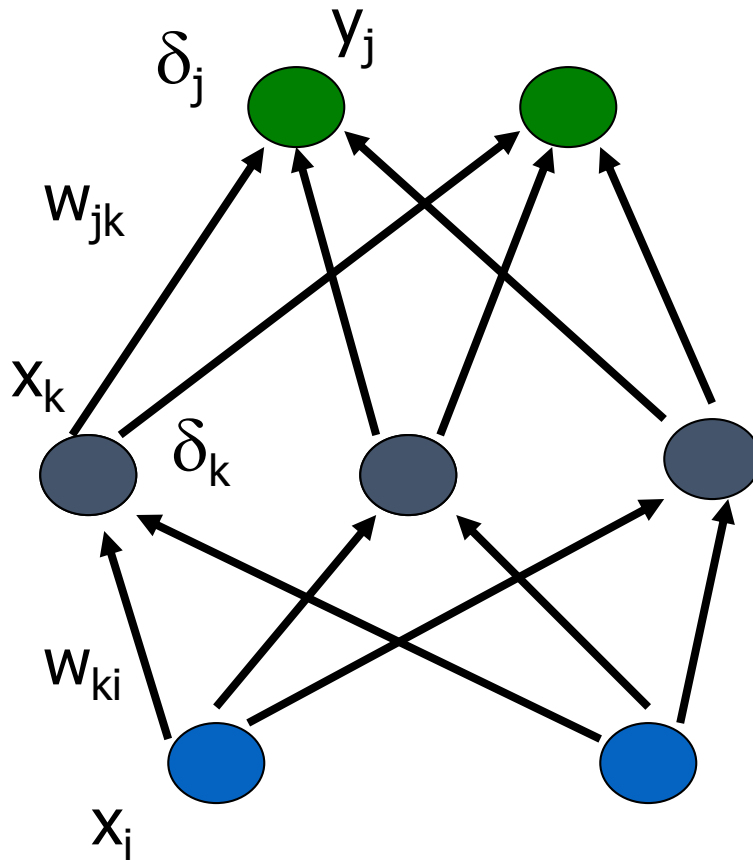
$$E[w_{ik}] = \frac{1}{2} \sum_j (t_j - y_j)^2$$

$$\begin{aligned} \frac{\partial E}{\partial w_{ik}} &= \frac{\partial}{\partial w_{ik}} \frac{1}{2} \sum_j (t_j - y_j)^2 \\ &= \frac{\partial}{\partial w_{ik}} \frac{1}{2} \sum_j (t_j - \sigma(\sum_k w_{kj} x_k))^2 \\ &= \frac{\partial}{\partial w_{ik}} \frac{1}{2} \sum_j (t_j - \sigma(\sum_k w_{kj} \sigma(\sum_i w_{ik} x_i)))^2 \\ &= -\sum_j (t_j - y_j) \sigma'_j(a) w_{jk} \sigma'_k(a) x_i \\ &= -\sum_j \delta_j w_{kj} \sigma'_k(a) x_i \\ &= -\sum_j \delta_j w_{kj} x_k (1 - x_k) x_i \end{aligned}$$

$$\Delta w_{ik} = \alpha \delta_k x_i$$

$$\text{with } \delta_k = \sum_j \delta_j w_{kj} x_k (1 - x_k)$$

Backpropagation



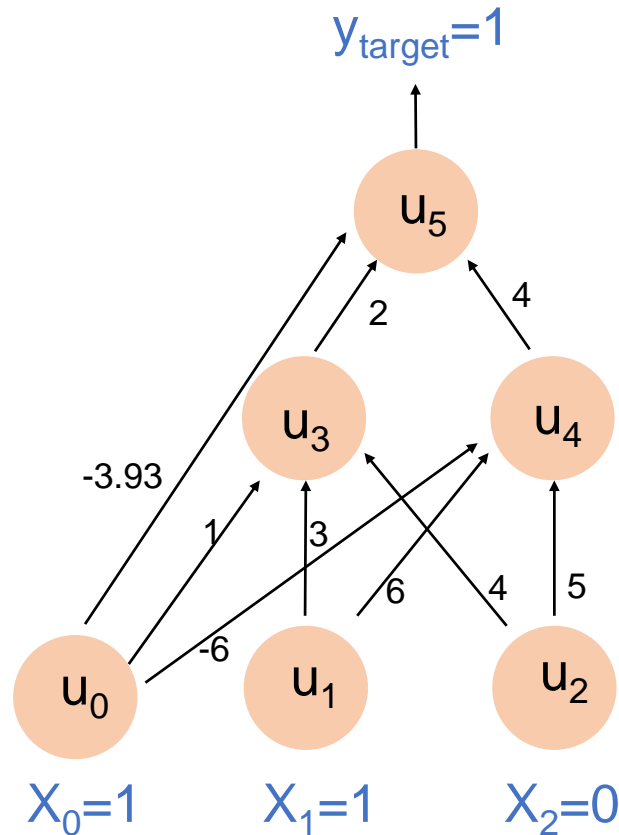
Backward pass:
propagate errors from
output to hidden layer

Forward pass:
Propagate activation
from input to output layer

Backpropagation Algorithm

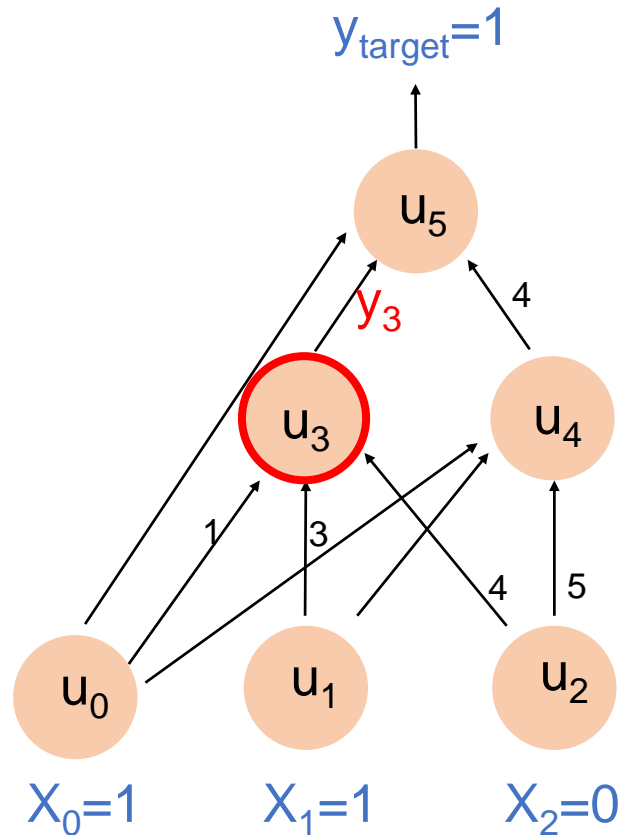
- Initialize each w_i to some small random value
- Until the termination condition is met, Do
 - For each training example $\langle (x_1, \dots, x_n), t \rangle$ Do
 - Input the instance (x_1, \dots, x_n) to the network and compute the network outputs y_k **Forward Pass**
 - For each output unit k
$$\delta_k = y_k(1 - y_k)(t_k - y_k)$$
 Backward Pass
 - For each hidden unit h
$$\delta_h = y_h(1 - y_h) \sum_k w_{h,k} \delta_k$$
 - For each network weight $w_{i,j}$ Do **Update**
$$w_{i,j} = w_{i,j} + \Delta w_{i,j} \quad \text{where } \Delta w_{i,j} = \alpha \delta_j x_{i,j}$$

BP: A worked example



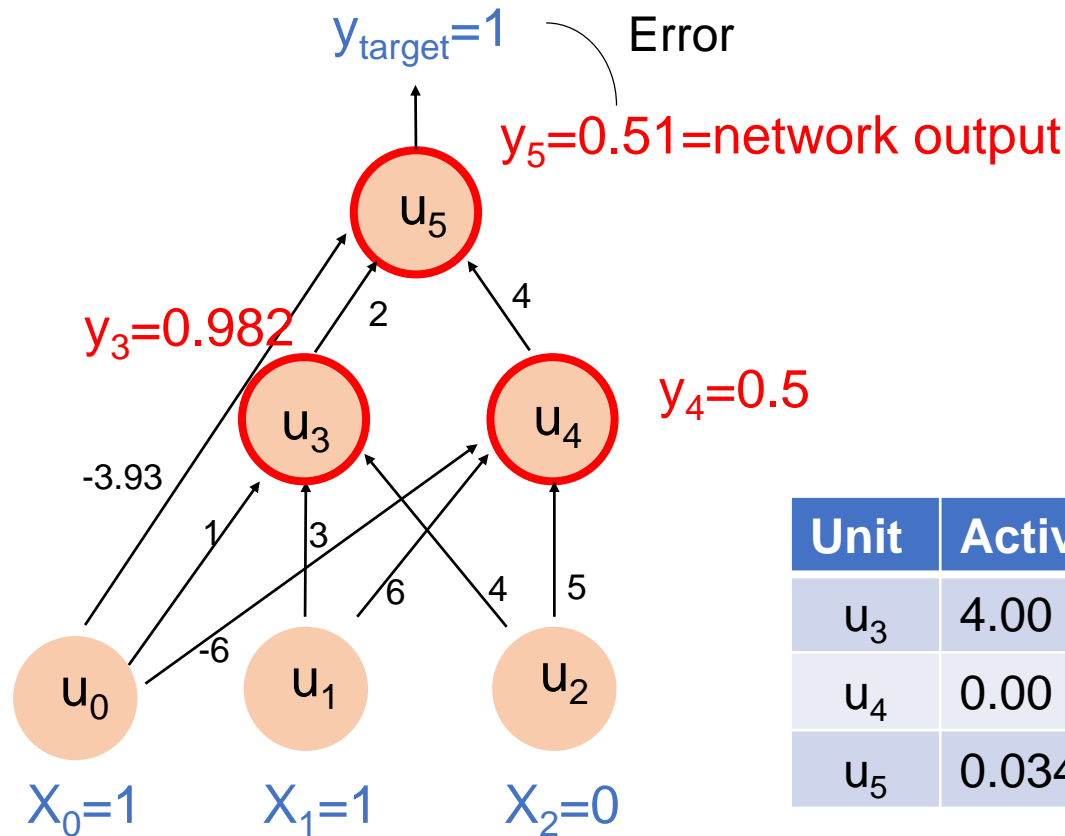
- Current state:
 - Weights: $w_{13}=3, w_{35}=3, w_{24}=5...$
 - Bias of u_4 is $w_{04}=-6...$
- Training data
 - $X_1=1, X_2=0$
 - $Y_{\text{target}}=1$

Worked example: Forward Pass



- Output for unit j is:
 - $a_j = \sum_i w_{ij}x_i$
 - $y_j = \sigma(a_j) = \frac{1}{1+e^{-a_j}}$
- E.g. Output for unit 3
 - $a_3 = 1*1 + 3*1 + 4*0 = 4$
 - $y_3 = \sigma(4) = 0.982$

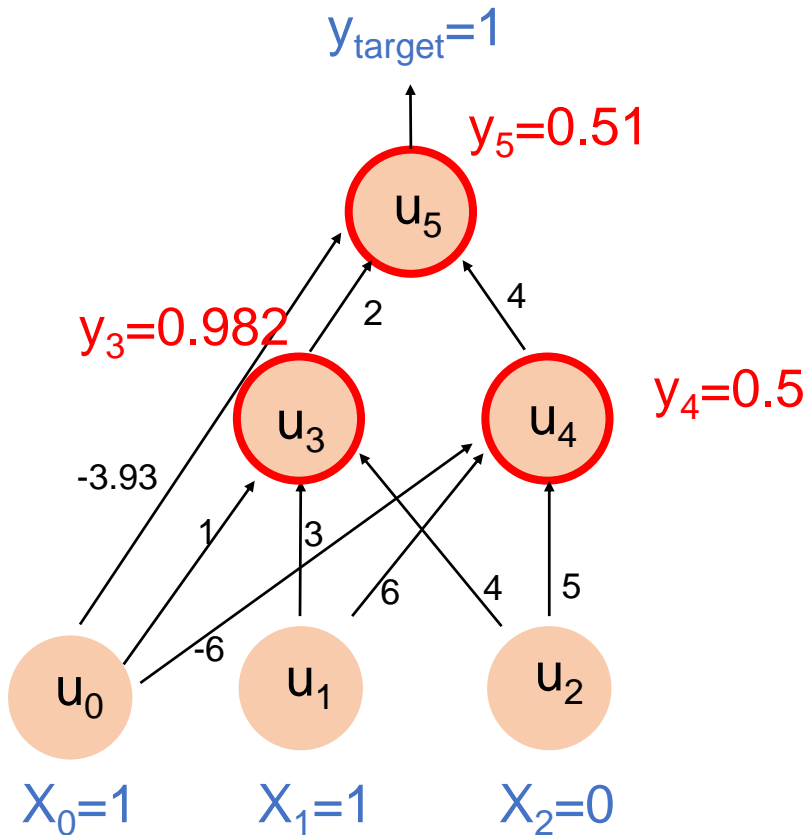
Worked example: Forward Pass



Unit	Activation a_i	Output y_i
u_3	4.00	0.982
u_4	0.00	0.50
u_5	0.034	0.51

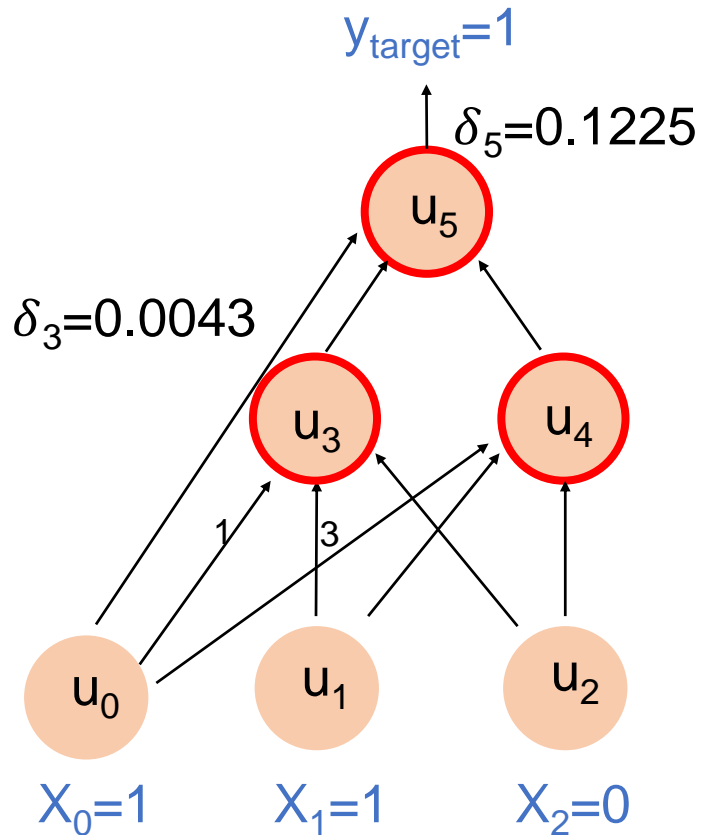
$$\text{Error} = Y_{\text{target}} - y_5 = 1 - 0.51 = 0.49$$

Worked example: Backward Pass



- Compute δ starting at output:
- $$\delta_5 = y_5(1 - y_5)(y_{target} - y_5)$$
$$= 0.51(1 - 0.51) \times 0.49$$
$$= 0.1225$$
- $$\delta_4 = y_4(1 - y_4) w_{45} \delta_5$$
$$= 0.5(1 - 0.5) \times 4 \times 0.122$$
$$= 0.1225$$
- $$\delta_3 = y_3(1 - y_3) w_{35} \delta_5$$
$$= 0.982(1 - 0.982) \times 2 \times 0.122$$
$$= 0.0043$$

Worked example: Update Weights

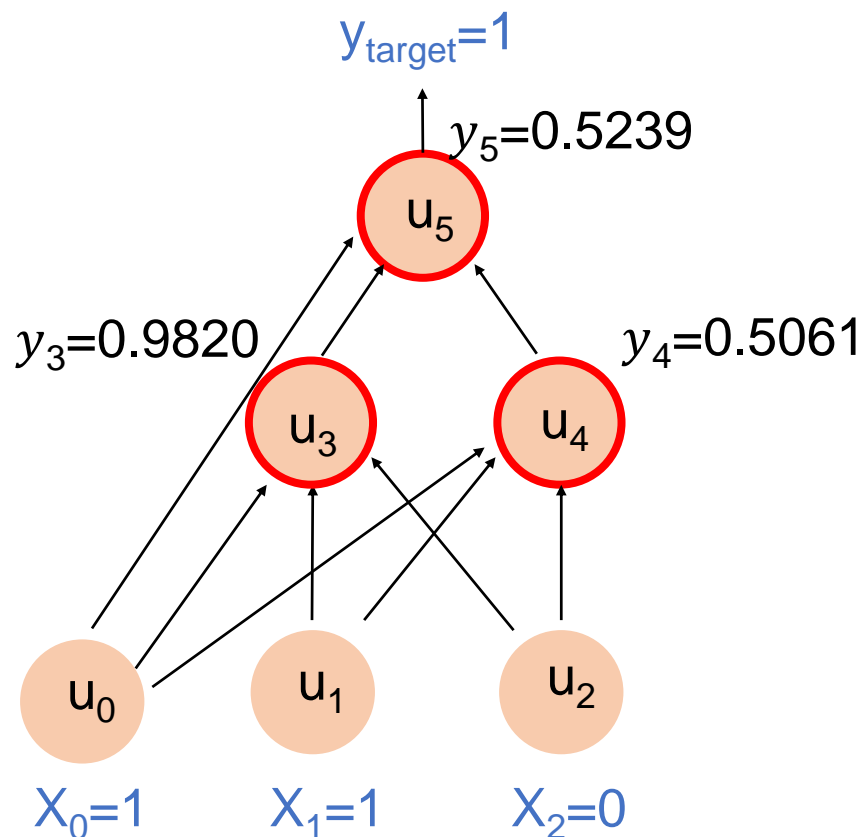


- Set learning rate $\alpha = 0.1$
- Change weights by:
 - $\Delta w_{ij} = \alpha \delta_j y_i$
- E.g. Bias weight on u_3 :
 - $\Delta w_{03} = \alpha \delta_3 x_0 = 0.1 * 0.0043 * 1 = 0.0004$
- New $w_{03} = w_{03}(\text{old}) + \Delta w_{03} = 1 + 0.0004 = 1.0004$
- New $w_{13} = 3 + 0.0004 = 3.0004$

For the all weights w_{ij} :

i	j	w_{ij}	δ_j	y_i	Updated w_{ij}
0	3	1	0.0043	1.0	1.0004
1	3	3	0.0043	1.0	3.0004
2	3	4	0.0043	0.0	4.0000
0	4	-6	0.1225	1.0	-5.9878
1	4	6	0.1225	1.0	6.0123
2	4	5	0.1225	0.0	5.0000
0	5	-3.92	0.1225	1.0	-3.9078
3	5	2	0.1225	0.9820	2.0120
4	5	4	0.1225	0.5	4.0061

Verification that it works



- On the next forward pass, the new activation are:
 - $y_3 = \sigma(4.0008) = 0.9820$
 - $y_4 = \sigma(0.0245) = 0.5061$
 - $y_5 = \sigma(0.0955) = \mathbf{0.5239}$
- The new error = $1 - 0.5239 = 0.476$
- Error has been reduced from 0.49 to 0.476

Backpropagation

- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
- Often include weight *momentum* term

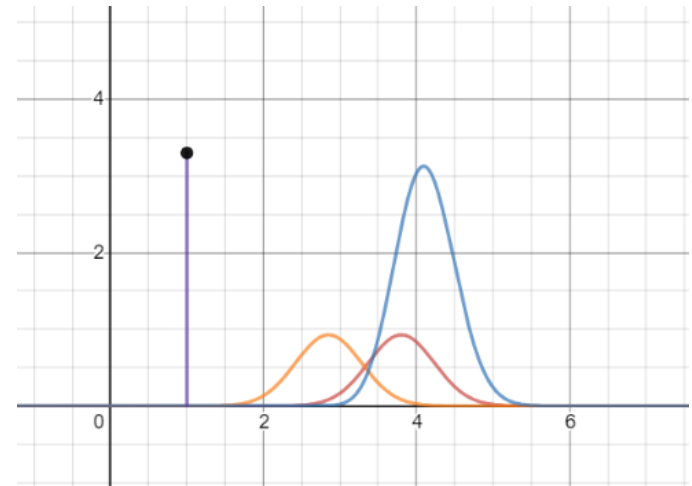
$$\Delta w_{i,j}(n) = \alpha \delta_j x_{i,j} + \eta \Delta w_{i,j}(n-1)$$

- Minimizes error training examples
 - Will it generalize well to unseen instances (overfitting)?
- Training can be slow typical 1000-10000 iterations

Cross entropy

- Cross-entropy is a measure of the difference between two probability distributions.
- The first distribution consists of true values. The second distribution consists of estimated values.

$$-\sum_i^N p(i) \log q(i)$$



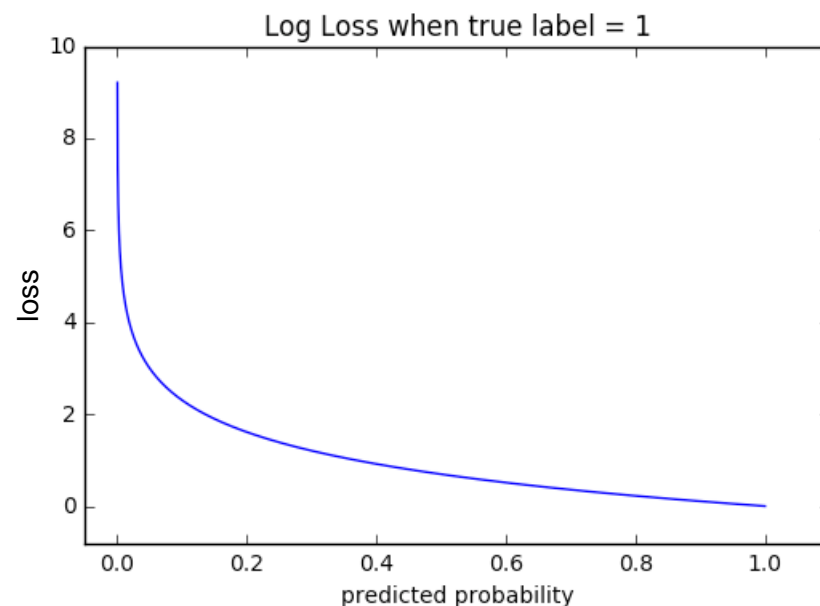
<https://www.desmos.com/calculator/zytm2sf56e>

Cross entropy (Cont.)

- Cross-entropy loss, or log loss, measures the performance of a classification model whose output is a probability value between 0 and 1

Y_{predict}	Y_{target}
0.8	1.0
0.2	0.0
0.1	0.0

$L(Y_{\text{predict}}, Y_{\text{target}}) = - \sum_i^N Y_{\text{target}}^i \log(Y_{\text{predict}}^i)$

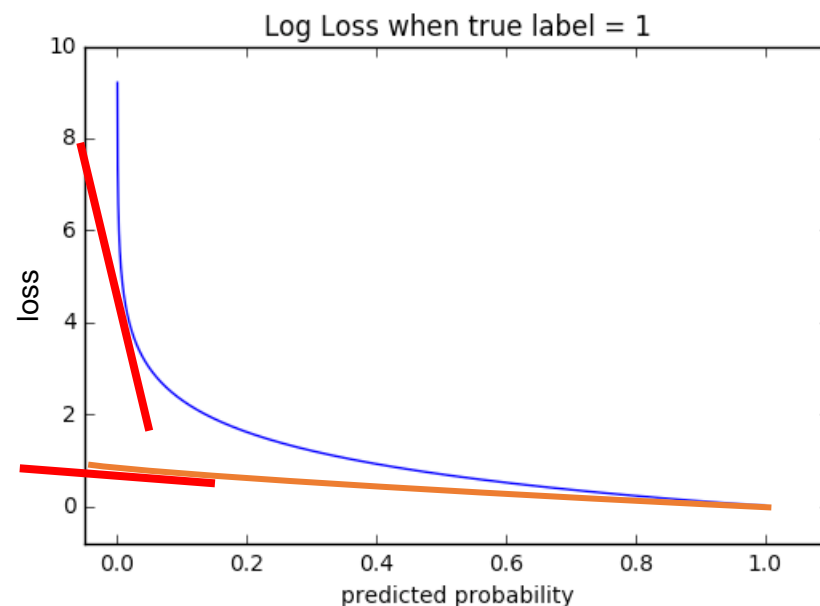


Why use cross entropy?

$$L(Y_{\text{predict}}, Y_{\text{target}}) = \text{Residual}^2 = \sum_i (Y_{\text{target}}^i - Y_{\text{predict}}^i)^2$$

Y_{predict}	Y_{target}
0.8	1.0
0.2	0.0
0.1	0.0

$L(Y_{\text{predict}}, Y_{\text{target}}) = -\sum_i Y_{\text{target}}^i \log(Y_{\text{predict}}^i)$



Cross entropy (Cont.)

- Binary Cross-Entropy Loss
- Categorical Cross-Entropy Loss
 - More than 2 classes and outcomes are one hot encoded
- Sparse Categorical Cross-Entropy Loss
 - More than 2 classes and outcomes are **not** one hot encoded

$$-\sum_i^N (Y_{\text{target}}^i \log(Y_{\text{predict}}^i) + (1 - Y_{\text{target}}^i) \log(1 - Y_{\text{predict}}^i))$$

$$-\sum_c^M \sum_i^N (Y_{\text{target}}^{i,c} \log(Y_{\text{predict}}^{i,c}))$$

[1,0,0]
[0,1,0]
[0,0,1]

1
2
3

TRUE	CLASS - 1	CLASS - 2
RECORD - 1	1	0
RECORD - 2	1	0
RECORD - 3	0	1

	CROSS-ENTROPY	CALCULATION
RECORD - 1	$-(1 * \log(0.9))$	0.1053605157
RECORD - 2	$-(1 * \log(0.7))$	0.3566749439
RECORD - 3	$-(1 * \log(0.6))$	0.5108256238

PREDICTED	CLASS - 1	CLASS - 2
RECORD - 1	0.9	0.1
RECORD - 2	0.7	0.3
RECORD - 3	0.4	0.6

0.3242870277875165

Outline

- Motivation
- Multilayer perceptrons (MLP)
- Backpropagation
- **RBF**

Radial Basis Function Networks

output neurons



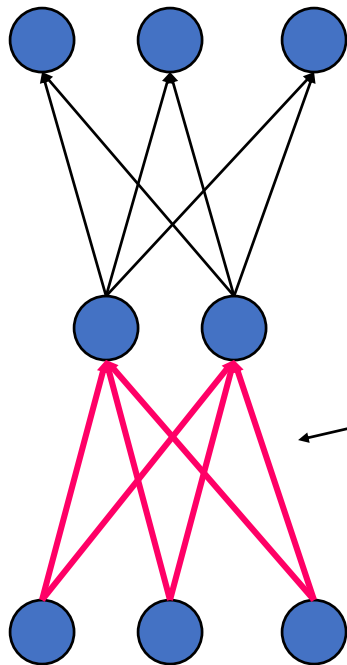
one layer of
hidden neurons



input nodes

Radial Basis Function Networks

output neurons



propagation function:

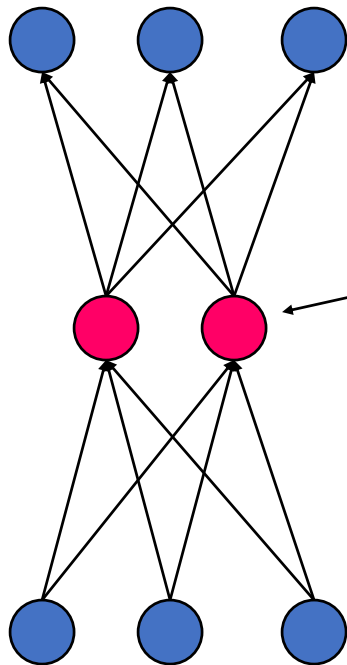
$$a_j = \sqrt{\sum_{i=1}^n (x_i - \mu_{i,j})^2}$$

input nodes

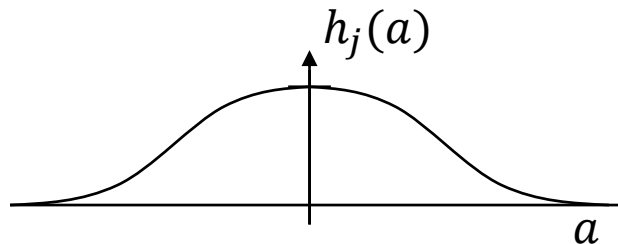
Radial Basis Function Networks

- Hidden units store means and variances
- Hidden units compute a Gaussian function of inputs vector \mathbf{x}

output neurons



output function:
(Gauss' bell-shaped function)

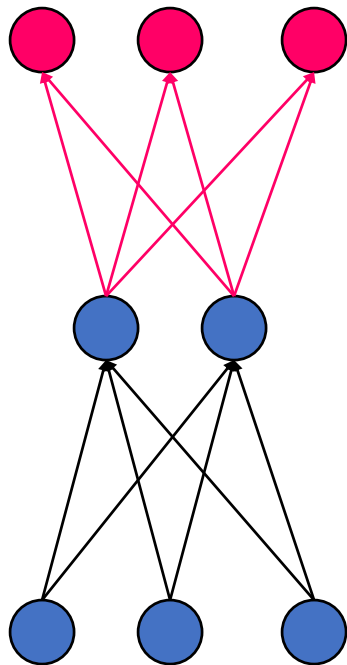


$$h_j(a_j) = e^{-\frac{a_j^2}{2\sigma_j^2}}$$

input nodes

RBF Networks

output neurons



output of network:

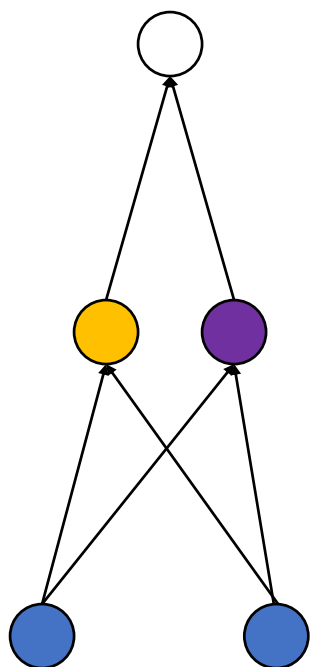
$$\text{out}_k = \sum_j w_{j,k} h_j$$

- Learn weights w , means μ , and variances σ by minimizing squared error function

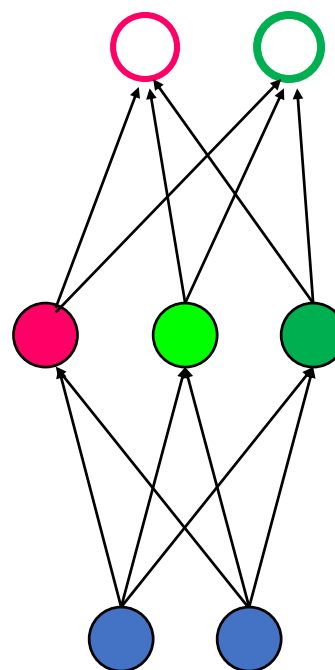
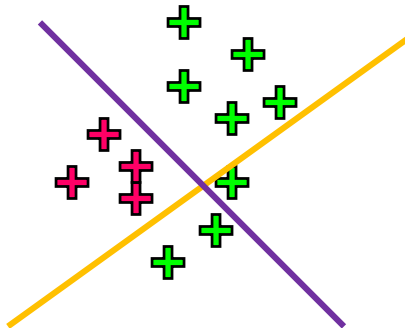
input nodes

RBF Networks and MLP

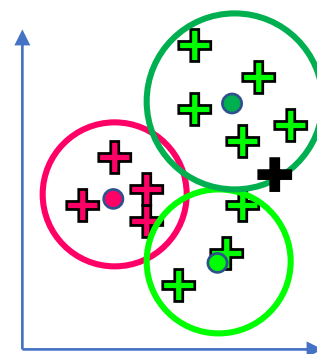
output neurons



MLP:



RBF:



input nodes

Summary

- Many similarities between biology and neural networks
 - 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
 - Network construction (structure, learning rate etc.) is a heuristic art
 - Hard to train large networks => deep learning

Questions?

"I work with **models**."

Others:



Me:

