CS 4602

Introduction to Machine Learning

Neural Networks

Instructor: Po-Chih Kuo

Roadmap

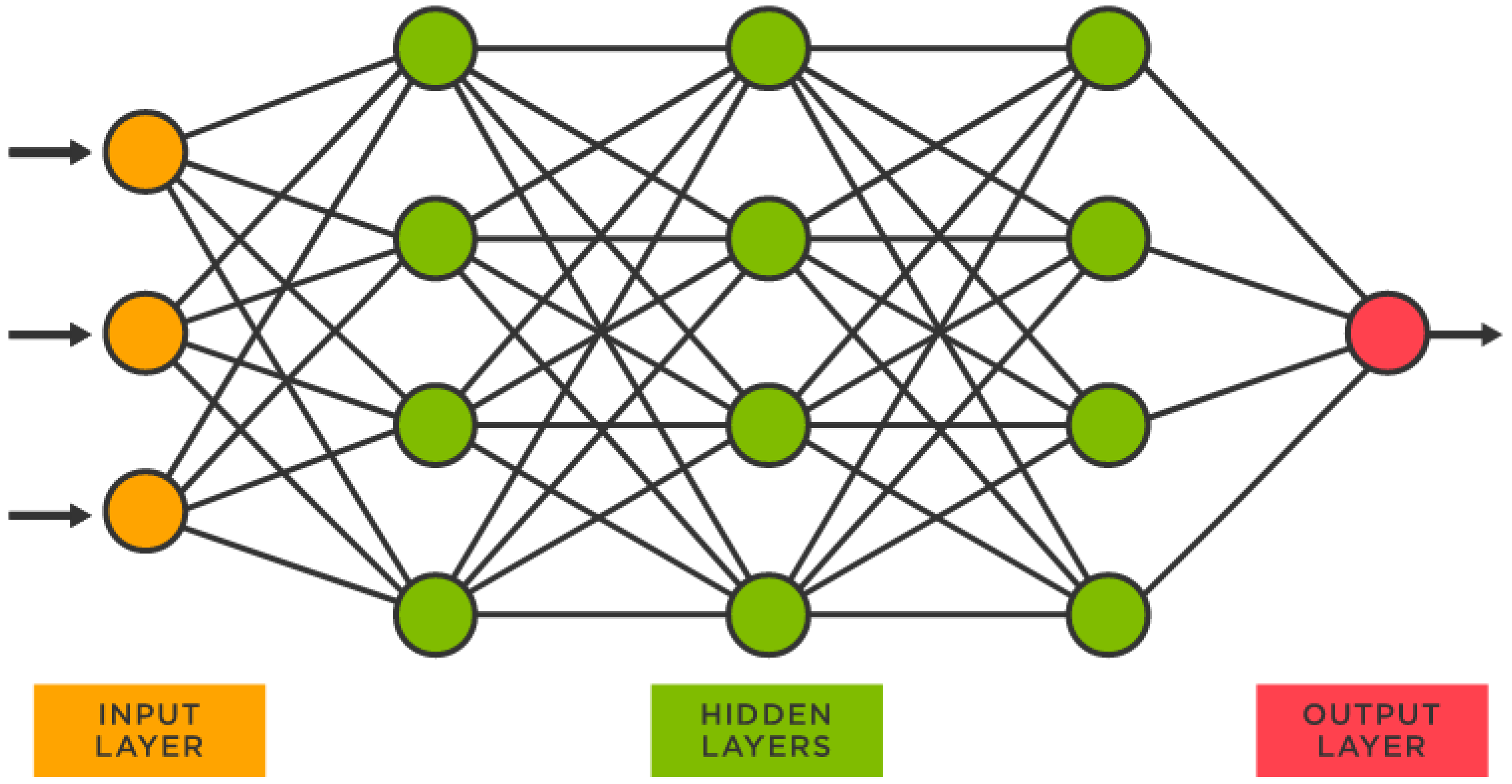
- Introduction and Basic Concepts
- Regression
- Bayesian Classifiers
- Decision Trees
- KNN
- Linear Classifier
- Neural Networks
- Deep learning
- Convolutional Neural Networks
- RNN/Transformer
- Reinforcement Learning
- Model Selection and Evaluation
- Clustering
- Data Exploration & Dimensionality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

Outline

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

What are connectionist neural networks?

- **Connectionism** refers to a computer modeling approach to computation that is loosely based upon the architecture of the brain.
- Many different models, but all include:
  - 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


INPUT LAYER
HIDDEN LAYERS
OUTPUT LAYER

Neural Network History

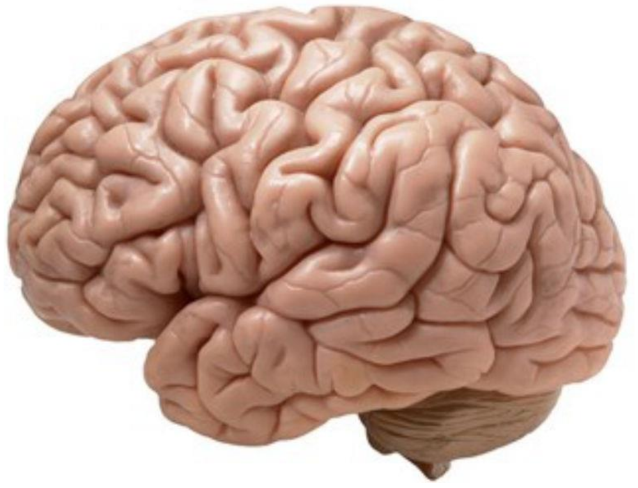
- History traces back to the 50's
  - became popular in the 80's with work by Rumelhart, Hinton, and Mclelland
  - A General Framework for Parallel Distributed Processing
- Peaked in the 90's. Today:
  - Hundreds of variants
  - Less a model of the actual brain than a useful tool, but still some debate
- Numerous applications
  - Handwriting, face, speech recognition
  - Self-driving Vehicles
  - Models of reading, sentence production, dreaming
- Debate for philosophers and cognitive scientists
  - Can human consciousness or cognitive abilities be explained by a connectionist model?

Comparison of Brains and Traditional Computers

• 200 billion neurons, 32 trillion synapses	• 1 billion bytes RAM but trillions of bytes on disk
• Element size: 10^{-6} m	• Element size: 10^{-9} m
• Energy use: 25W	• Energy watt: 30-90W (CPU)
• Processing speed: 100 Hz	• Processing speed: 10^{9} Hz
• Parallel, Distributed	• Serial, Centralized
• Fault Tolerant	• Generally not Fault Tolerant
• Learns: Yes	• Learns: Some
• Conscious: Usually	• Conscious: Generally No

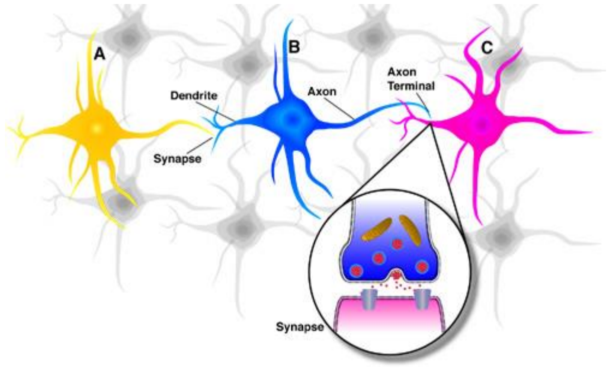
Biological Inspiration

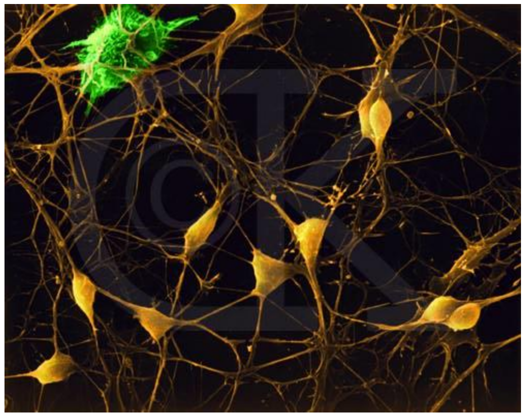
Idea : To make the computer more robust, intelligent, and learn, …
Let’s model our computer software (and/or hardware) after the brain



Neurons in the Brain

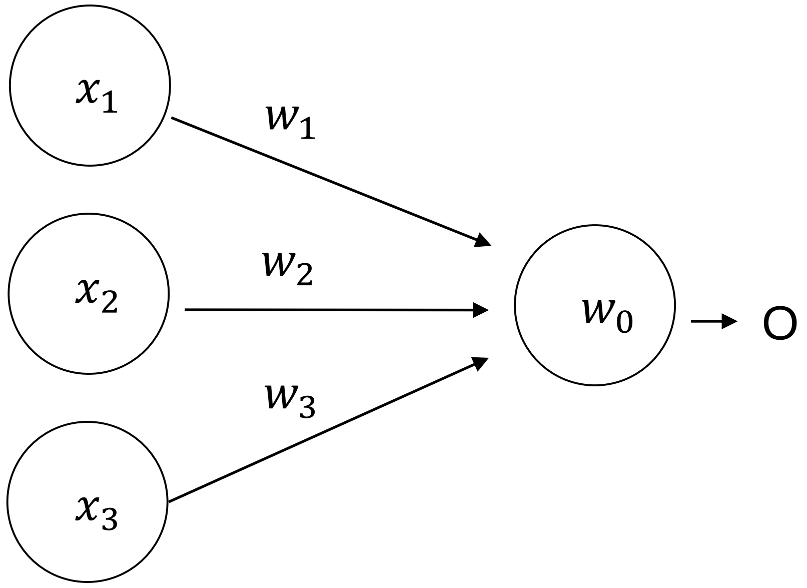
- Although heterogeneous, at a low level the brain is composed of neurons
  - A neuron receives input from other neurons (generally thousands) from its synapses
  - Inputs are approximately summed
  - When the input exceeds a threshold the neuron sends an electrical spike that travels that travels from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





Perceptrons

- Initial proposal of connectionist networks
- Rosenblatt, 50's and 60's
- Essentially a linear discriminant composed of nodes, weights

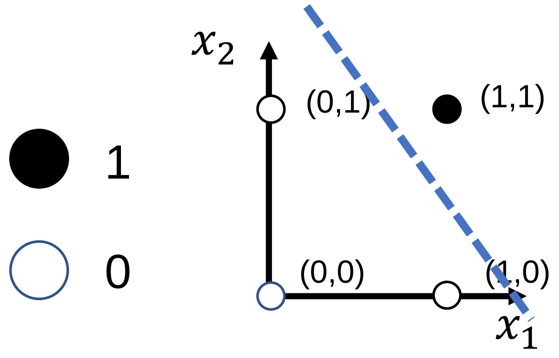


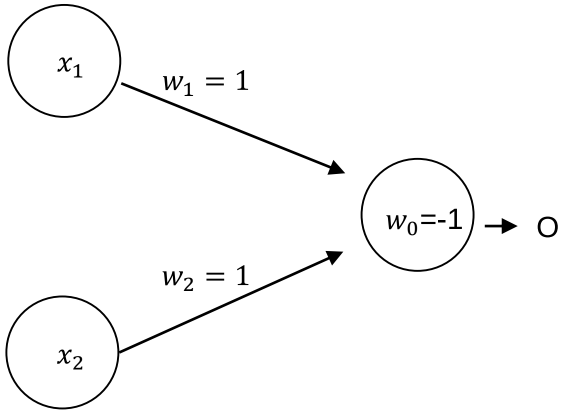
Activation Function

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

Linear separability

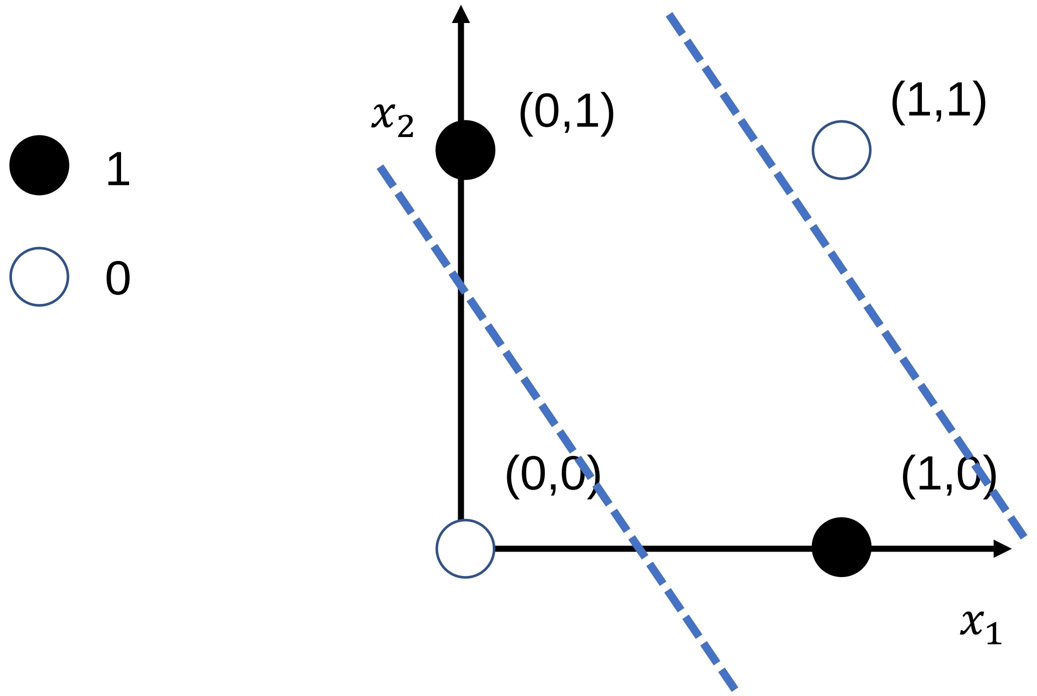
- Consider a single-layer perceptron
  - Assume threshold units
  - Assume binary inputs and outputs
  - Weighted sum forms a linear hyperplane $\sum_i w_i x_i = 0$
- Consider a single output network with two inputs
  - Only functions that are linearly separable can be computed
  - Example: AND is linearly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





Linear inseparability

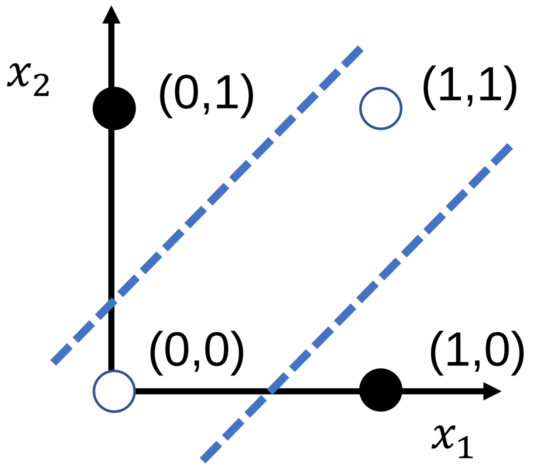
- Single-layer perceptron with threshold units fails if problem is not linearly separable
    - Example: XOR



Solution in 1980s: Multilayer perceptrons

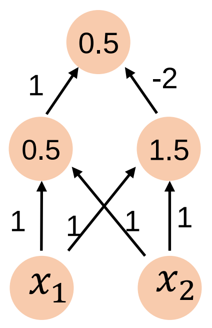
- Removes many limitations of single-layer networks
  - Can solve XOR

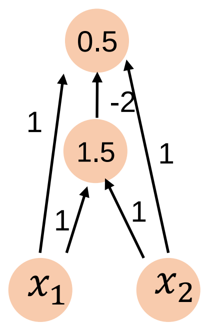
- How to Draw a two-layer perceptron that computes the XOR function?
  - 2 binary inputs x_1 and x_2
  - 1 binary output
  - One “hidden” layer
  - Find the appropriate weights and threshold



Multilayer perceptrons

- Examples of two-layer perceptrons that compute XOR

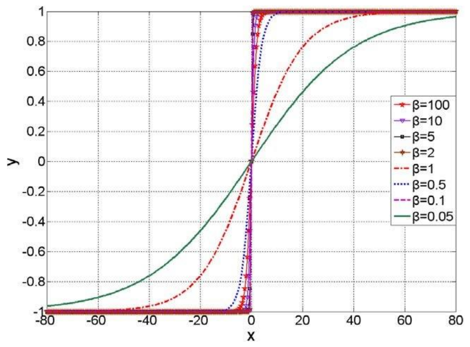


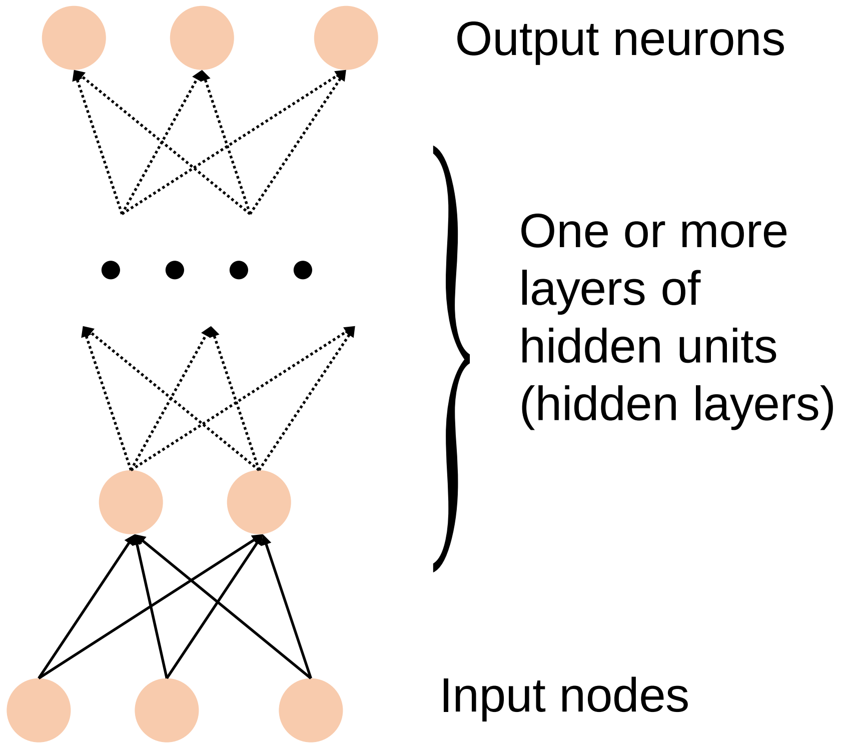


- E.g. Right side network
  - Output is 1

  if and only if $x_1 + x_2 – 2(x_1 + x_2 – 1.5 > 0) – 0.5 > 0$

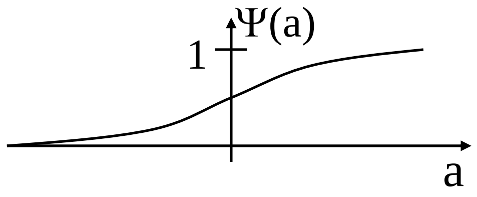
Multilayer perceptron



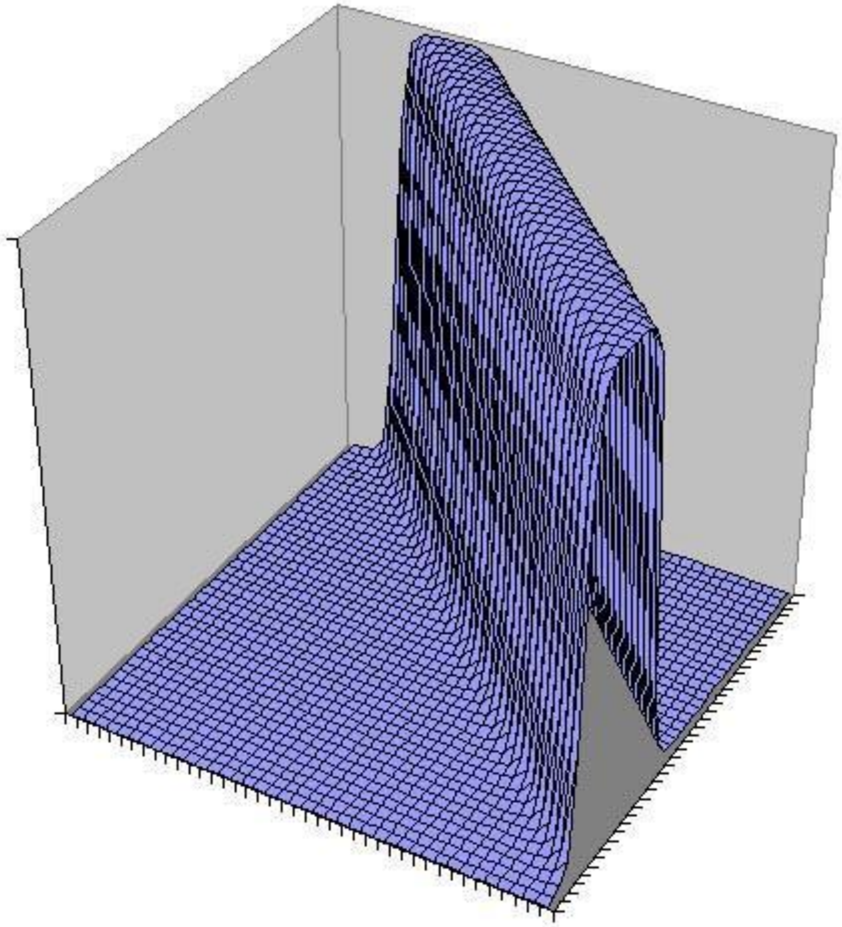


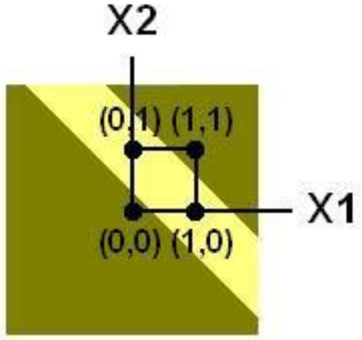
The most common output function (Sigmoid):

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



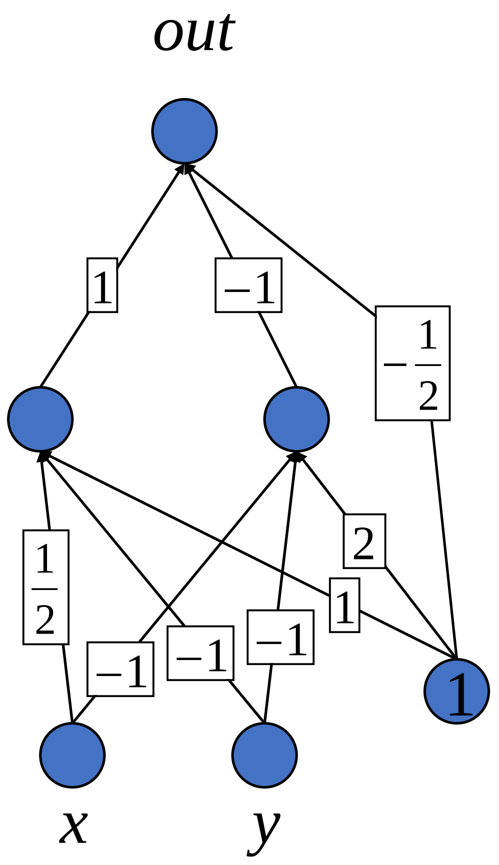
(non-linear function)

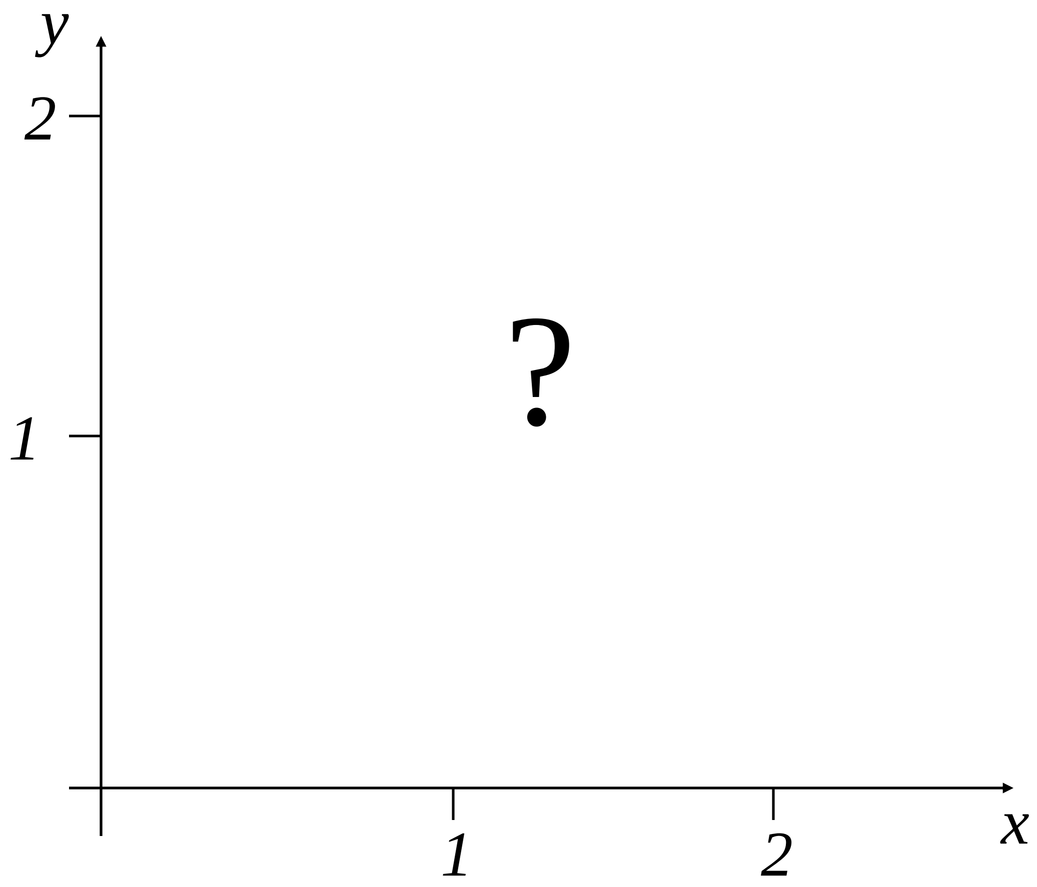




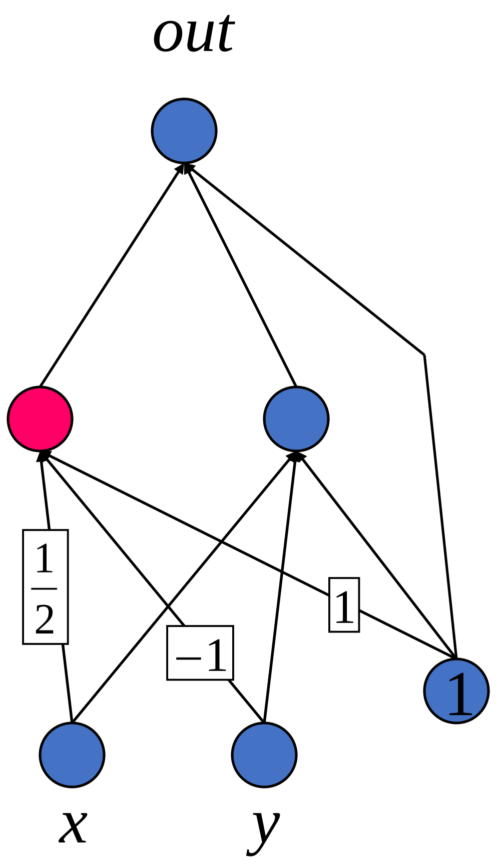
Source: http://colinfahey.com/

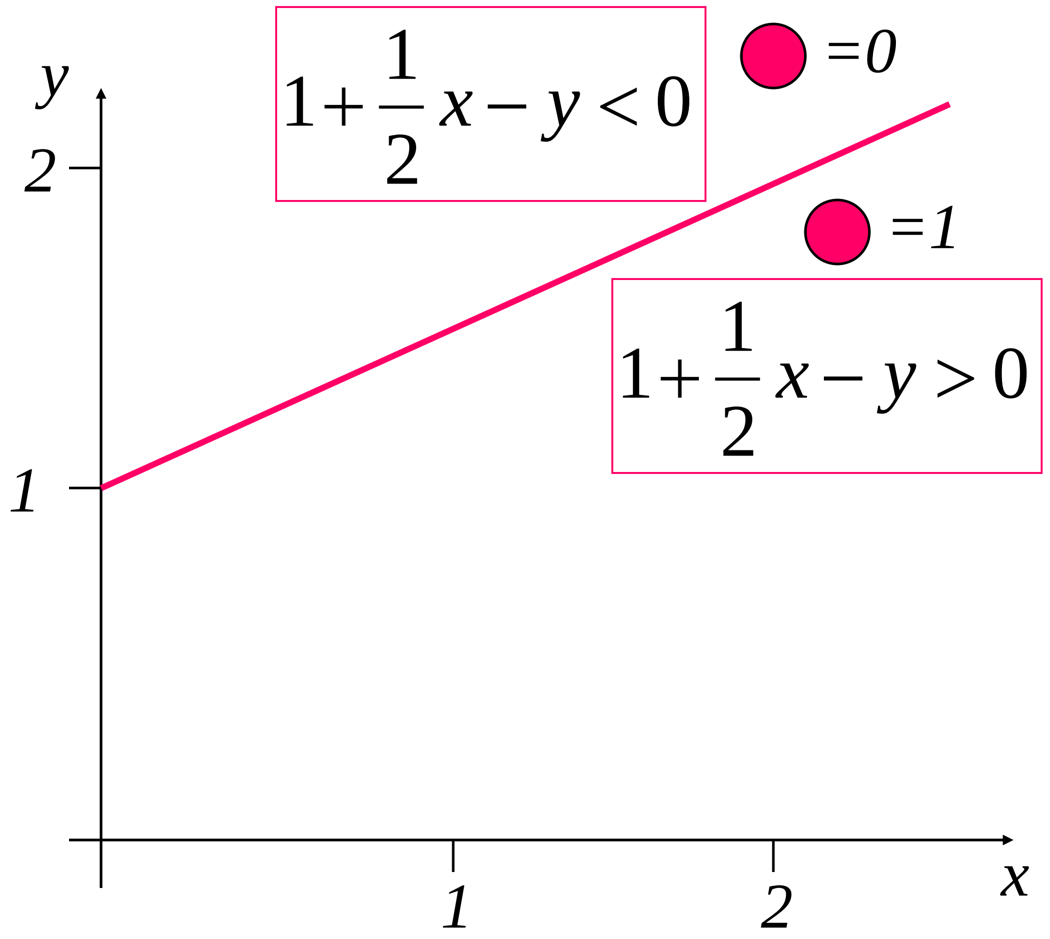
Perceptrons as Constraint Satisfaction Networks



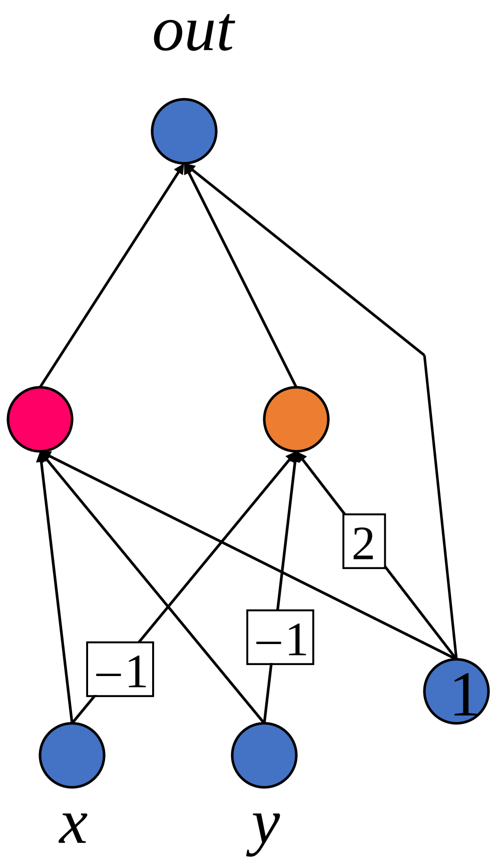


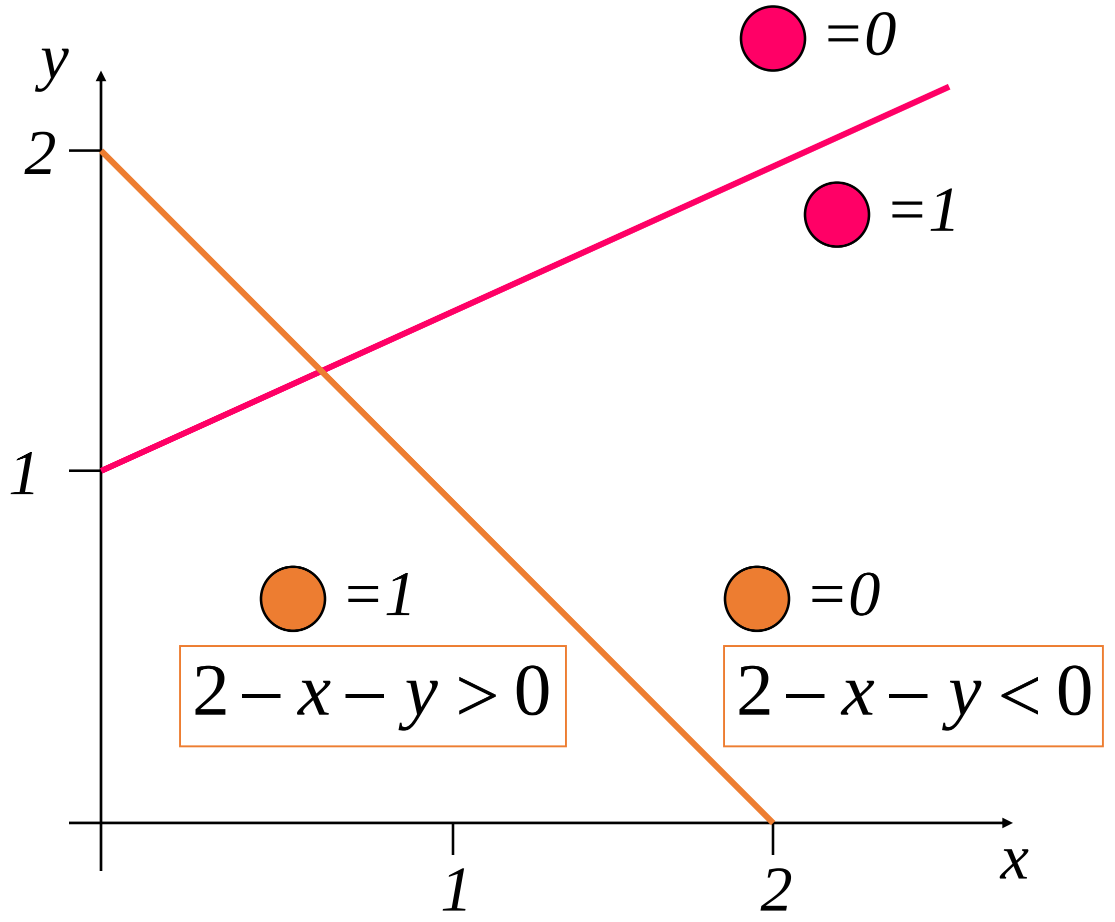
Perceptrons as Constraint Satisfaction Networks



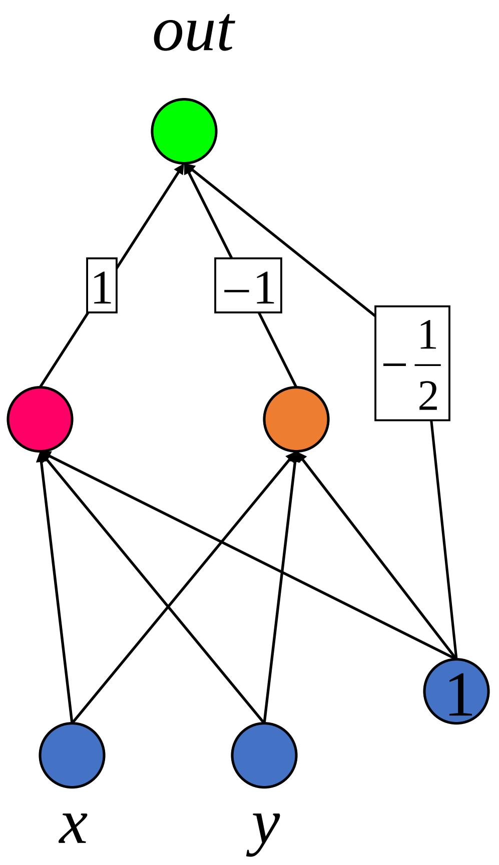


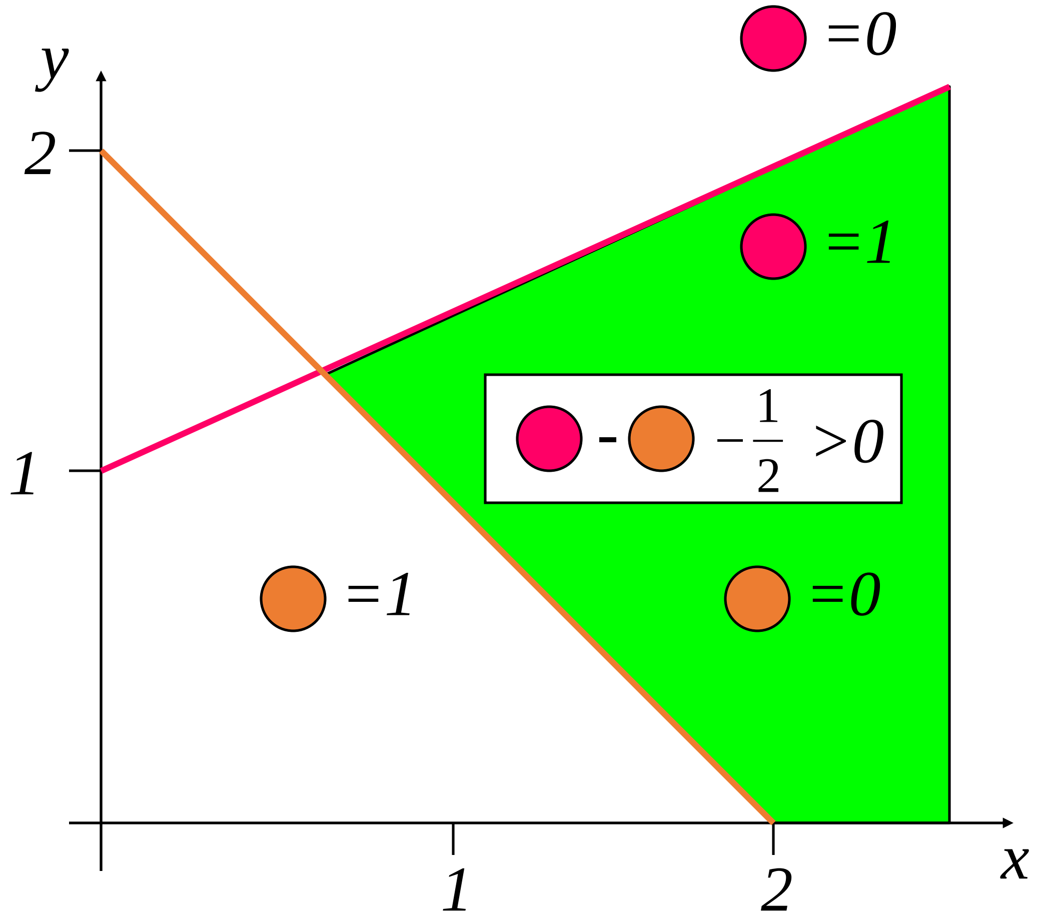
Perceptrons as Constraint Satisfaction Networks



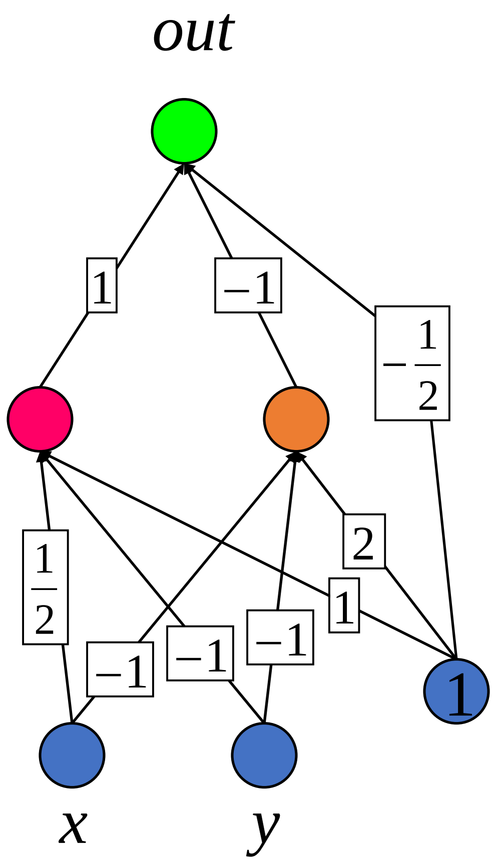


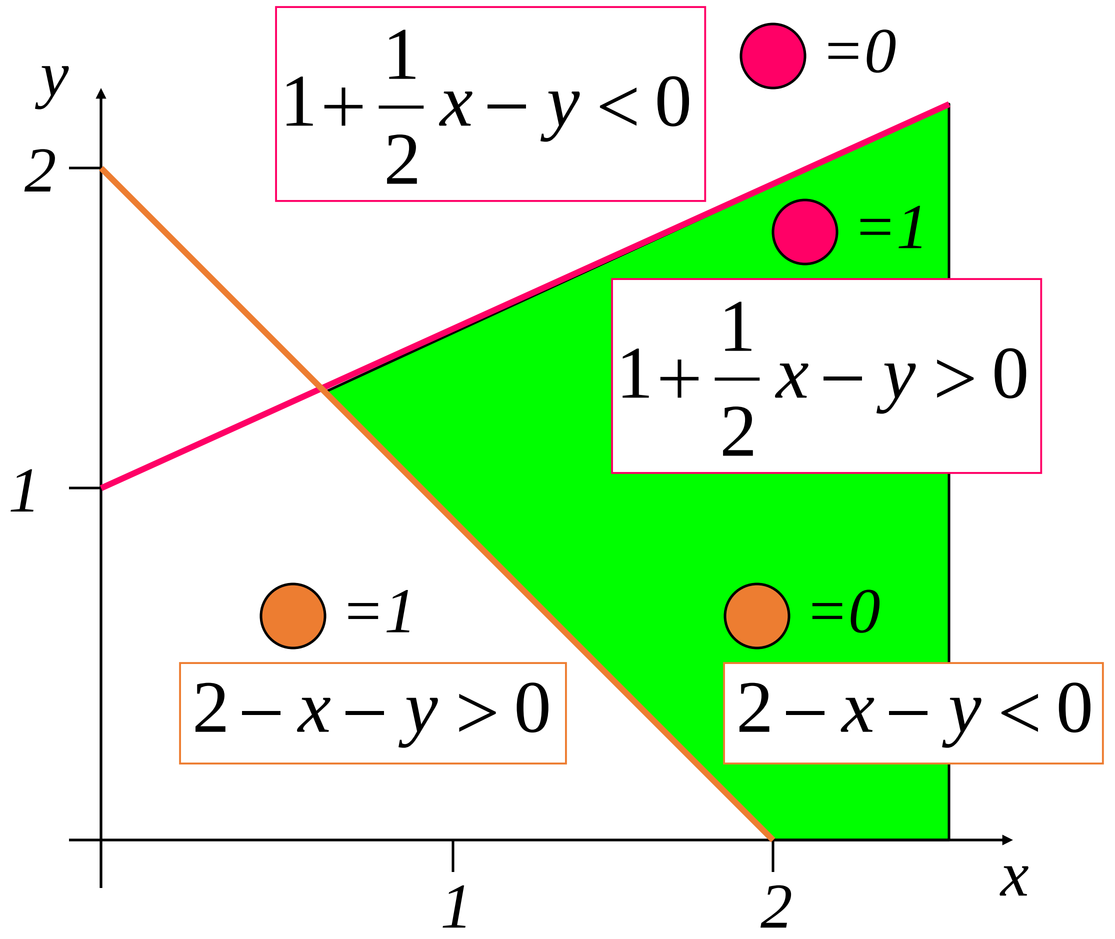
Perceptrons as Constraint Satisfaction Networks





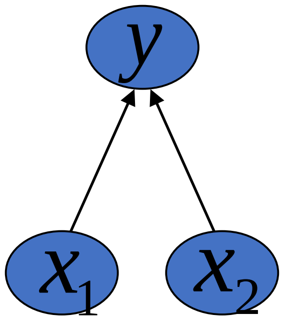
Perceptrons as Constraint Satisfaction Networks

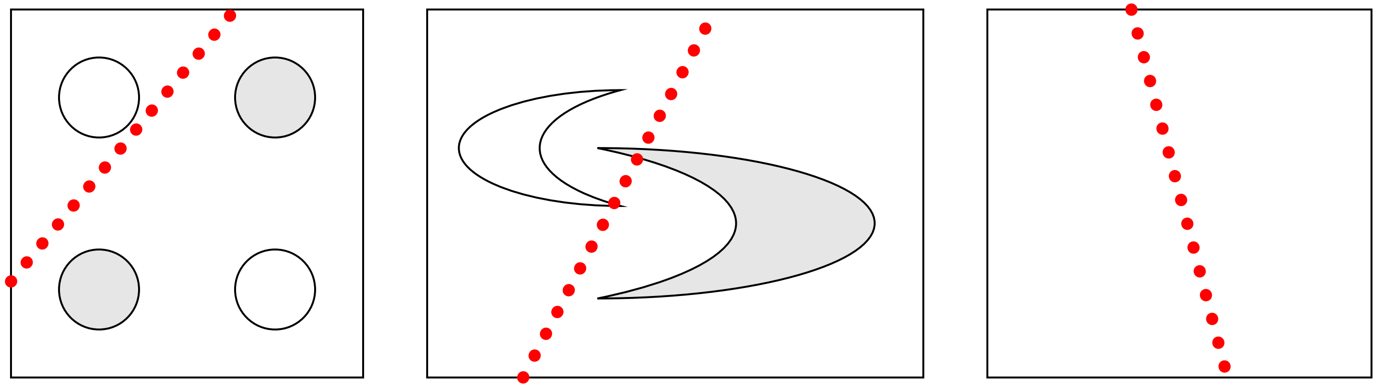




Decision Boundary

- 0 hidden layers: linear classifier
  - Hyperplanes

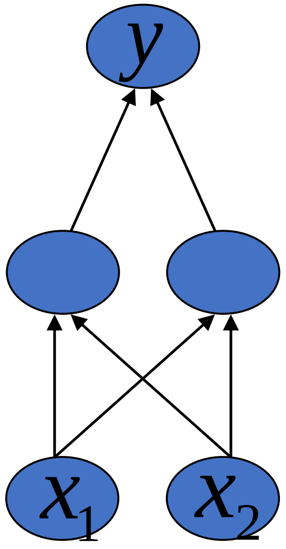


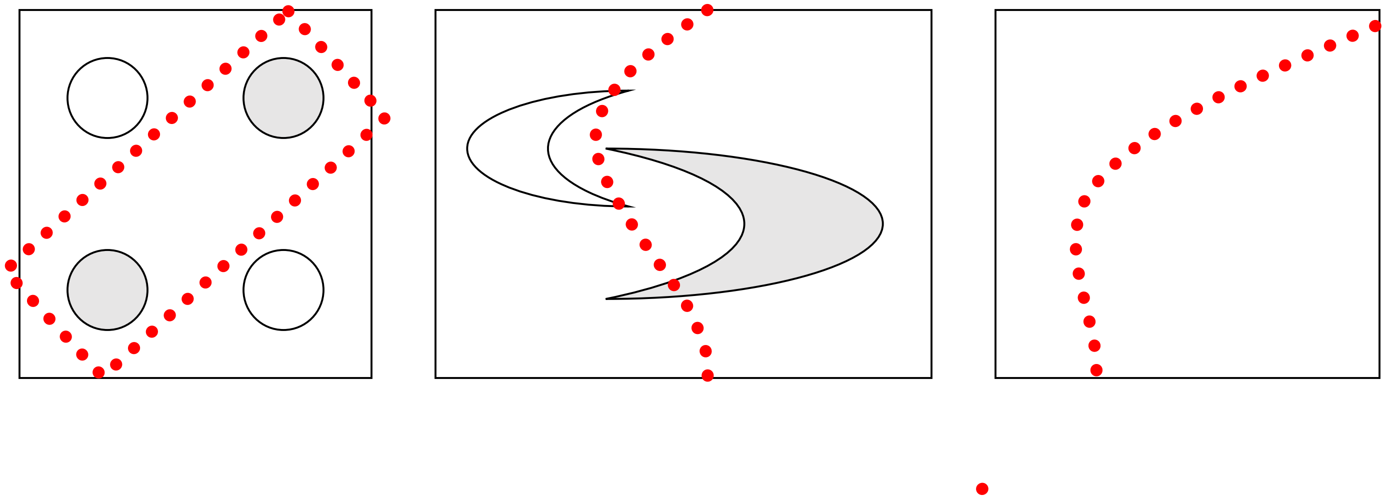


Example from to Eric Postma via Jason Eisner

Decision Boundary

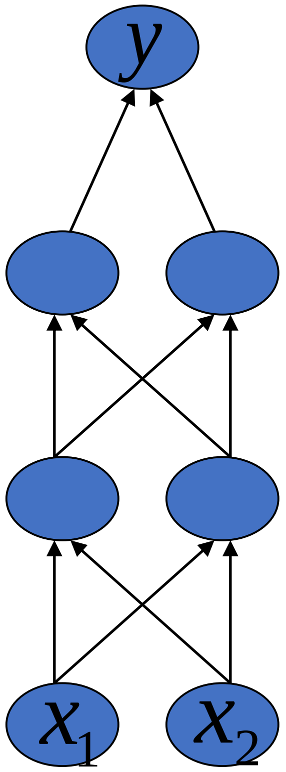
- 1 hidden layer
  - Boundary of convex region (open or closed)



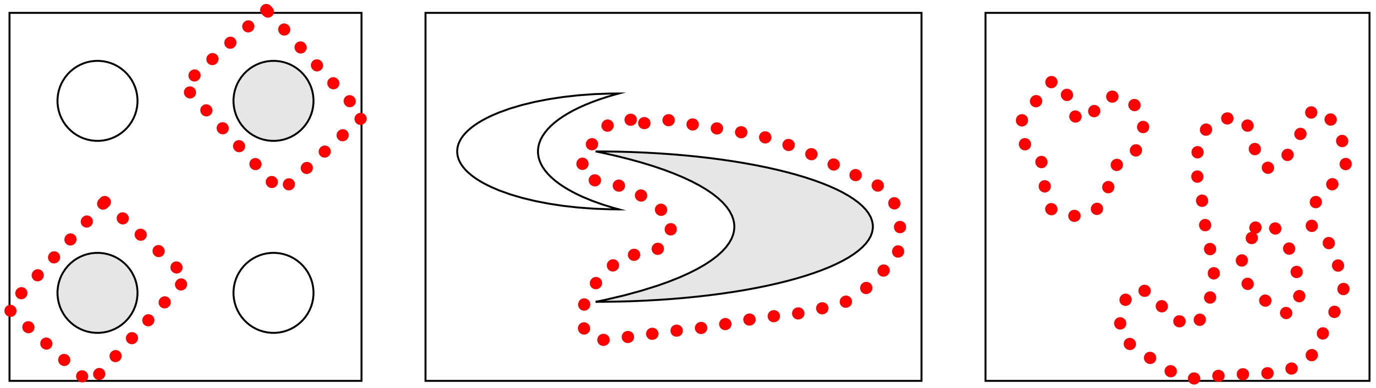


Example from to Eric Postma via Jason Eisner

Decision Boundary



- 2 hidden layers
  - Combinations of convex regions



Example from to Eric Postma via Jason Eisner

Outline

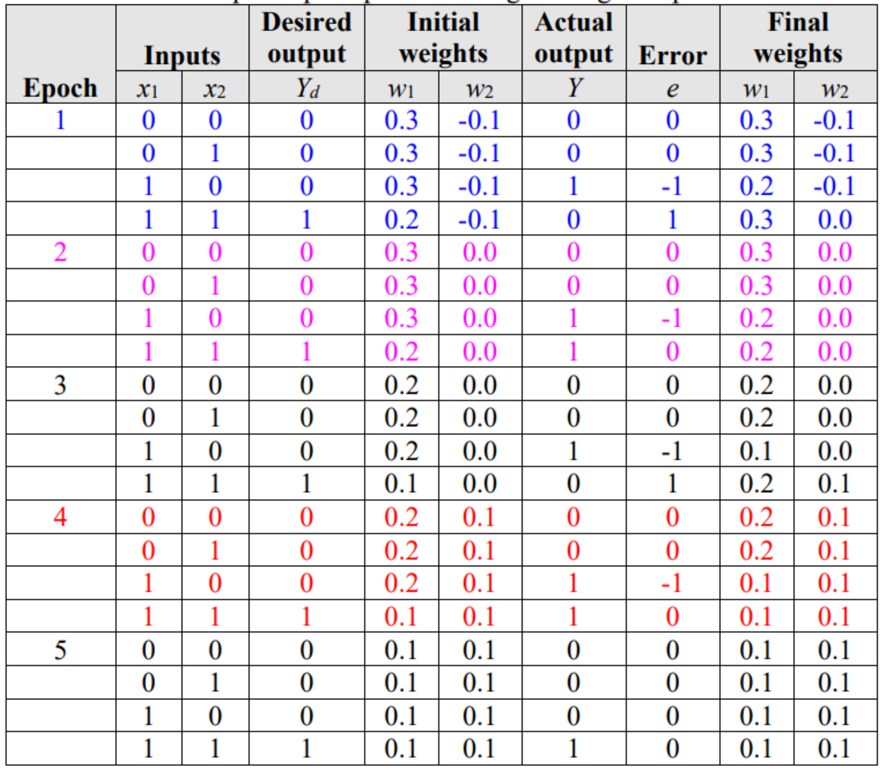
- Motivation
- Multilayer perceptrons (MLP)
- Backpropogation (BP)
- Extension

What about learning?

- Perceptron learning rule

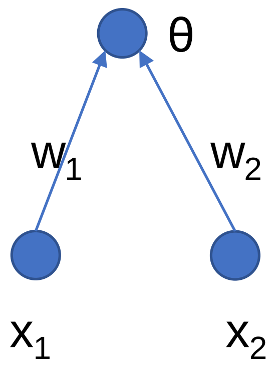
$$\boldsymbol{w}' = \boldsymbol{w} + \alpha \sum_{n \in M} \boldsymbol{x}^n (Y_d - Y)$$

Table 6.3 Example of perceptron learning: the logical operation AND



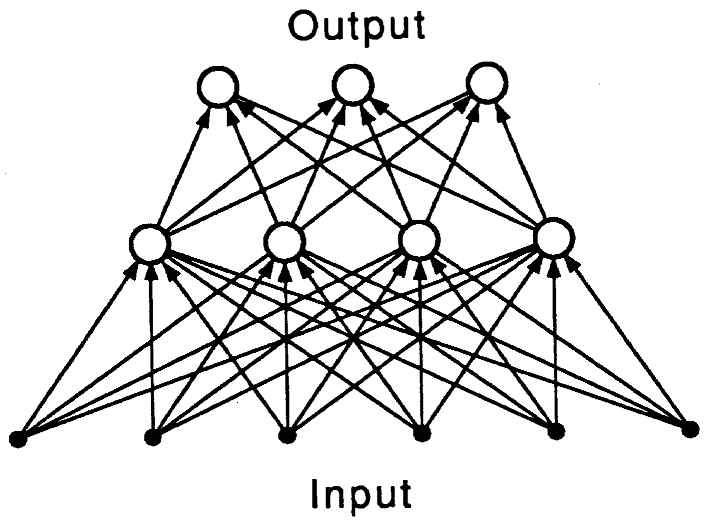
	Inputs		**Desired output**	**Initial weights**		**Actual output**	**Error**	**Final weights**	
Epoch	x_1	x_2	Y_d	w_1	w_2	Y	e	w_1	w_2
1	0	0	0	0.3	-0.1	0	0	0.3	-0.1
	0	1	0	0.3	-0.1	0	0	0.3	-0.1
	1	0	0	0.3	-0.1	1	-1	0.2	-0.1
	1	1	1	0.2	-0.1	0	1	0.3	0.0
2	0	0	0	0.3	0.0	0	0	0.3	0.0
	0	1	0	0.3	0.0	0	0	0.3	0.0
	1	0	0	0.3	0.0	1	-1	0.2	0.0
	1	1	1	0.2	0.0	1	0	0.2	0.0
3	0	0	0	0.2	0.0	0	0	0.2	0.0
	0	1	0	0.2	0.0	0	0	0.2	0.0
	1	0	0	0.2	0.0	1	-1	0.1	0.0
	1	1	1	0.1	0.0	0	1	0.2	0.1
4	0	0	0	0.2	0.1	0	0	0.2	0.1
	0	1	0	0.2	0.1	0	0	0.2	0.1
	1	0	0	0.2	0.1	1	-1	0.1	0.1
	1	1	1	0.1	0.1	1	0	0.1	0.1
5	0	0	0	0.1	0.1	0	0	0.1	0.1
	0	1	0	0.1	0.1	0	0	0.1	0.1
	1	0	0	0.1	0.1	0	0	0.1	0.1
	1	1	1	0.1	0.1	1	0	0.1	0.1

Threshold $\theta = 0.2$, learning rate $\alpha = 0.1$



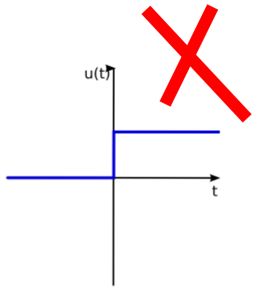
Learning networks

- 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 (Refer to previous lecture slides)

Perceptron vs. Gradient Descent Rule

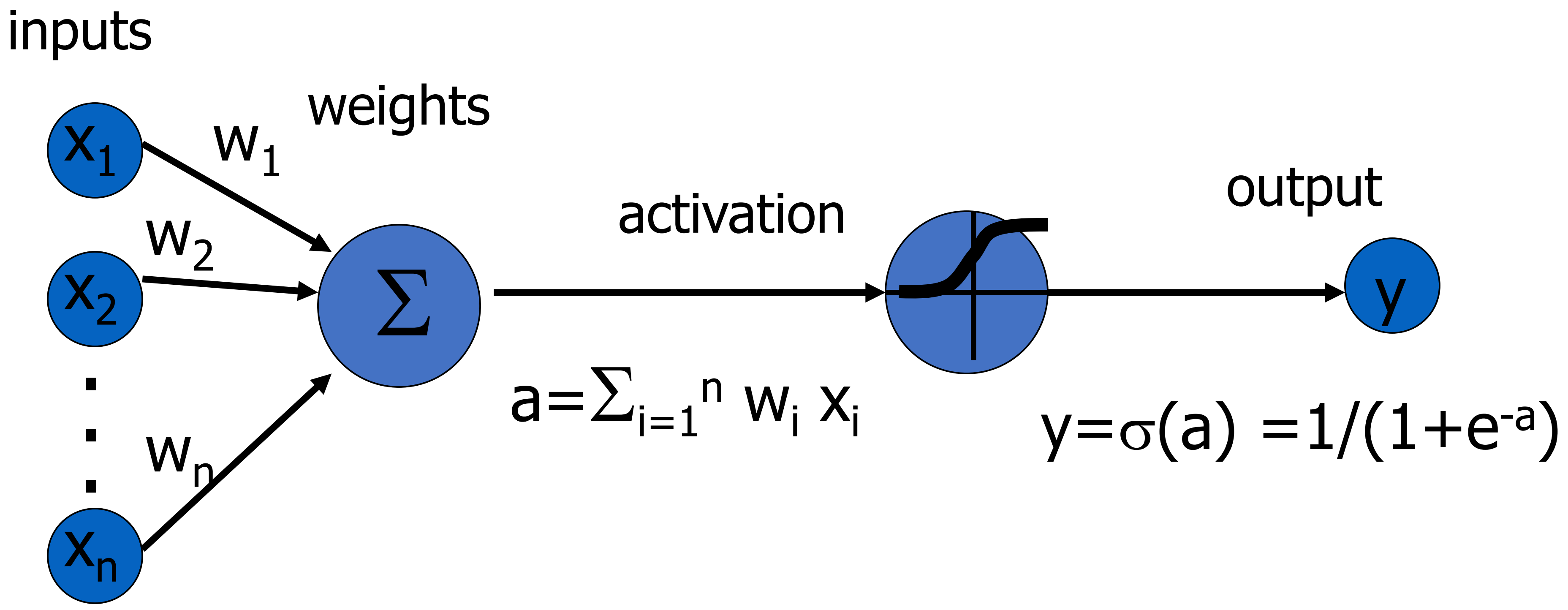
Perceptron learning rule guaranteed to succeed if

- 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

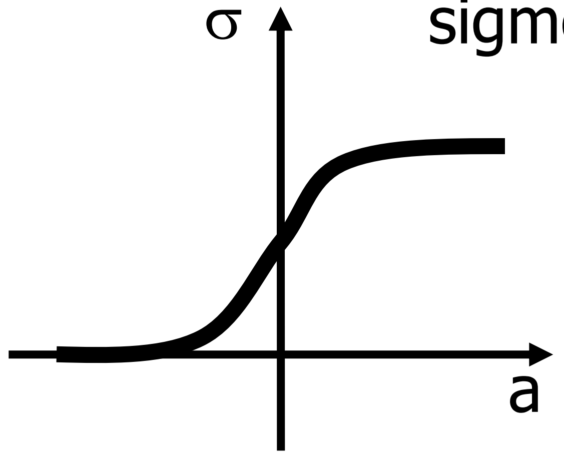
Neuron with Sigmoid Function

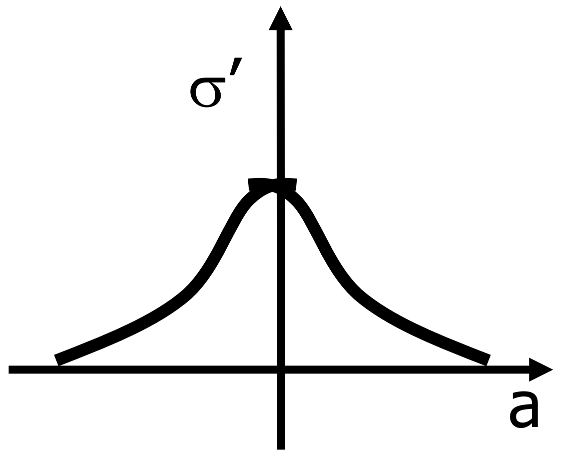


$$a=\sum_{i=1}^{n} w_i\, x_i$$

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

Gradient Descent Rule for Sigmoid Output Function





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

$$\partial E^p/\partial w_i = \partial/\partial w_i\ \tfrac{1}{2}\,(t^p-y^p)^2$$

$$= \partial/\partial w_i\ \tfrac{1}{2}\,(t^p-\ \sigma(\textstyle\sum_i w_i\, x_i^p))^2$$

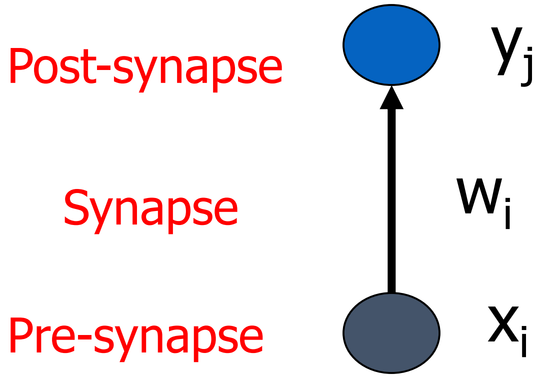
$$= (t^p-y^p)\ \sigma'(\textstyle\sum_i w_i\, x_i^p)\ (-x_i^p)$$

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

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

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

Gradient Descent Learning Rule



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

learning rate

derivative of
activation function

error δ_j of
post-synaptic neuron

activation of
pre-synaptic neuron

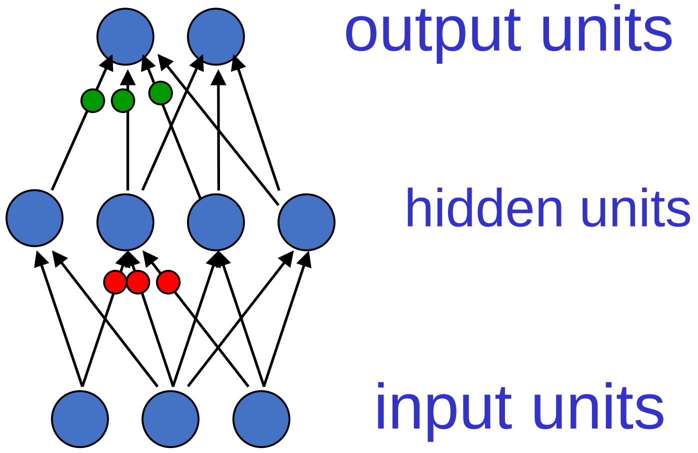
Learning with hidden units

- Networks without hidden units are very limited in the input-output mappings they can model.
- More layers of linear units do not help. It’s still linear.
- We need multiple layers of adaptive non-linear hidden units. This gives us a universal approximator. But how can we train such nets?
  - We need an efficient way of adapting all the weights, not just the last layer.
  - 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.

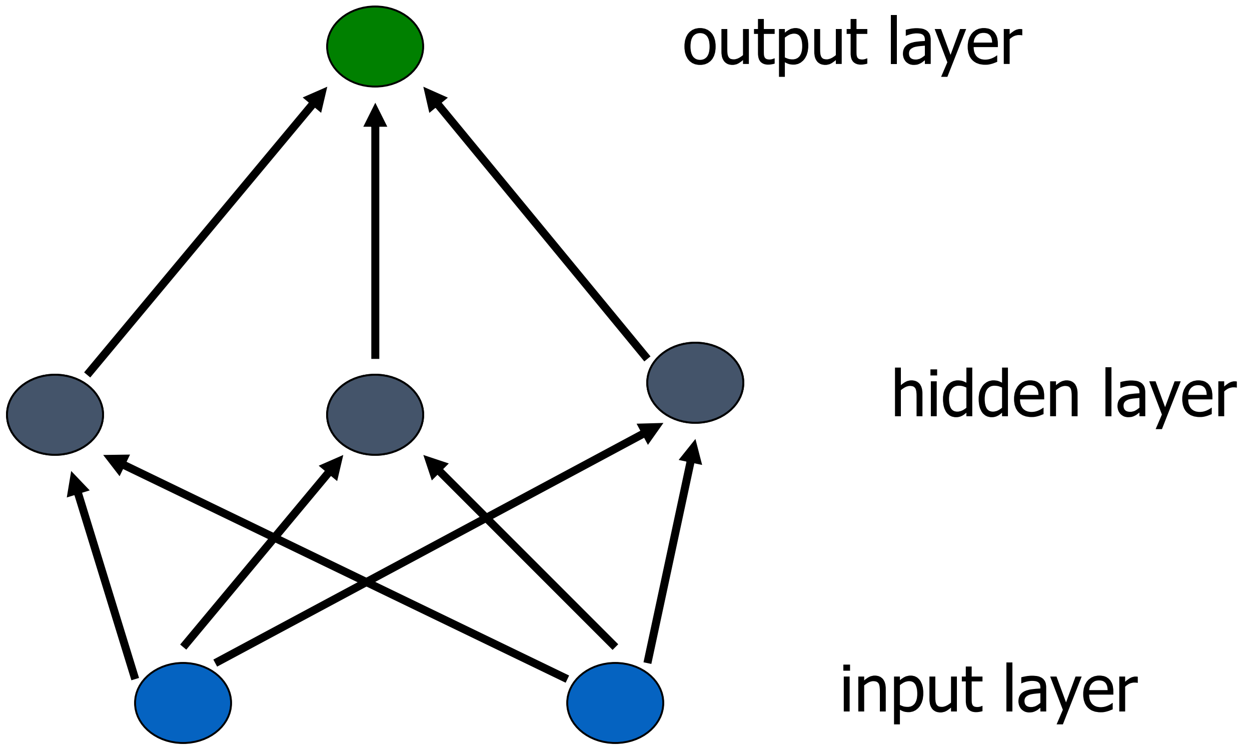


Learning the input to hidden weights is hard.

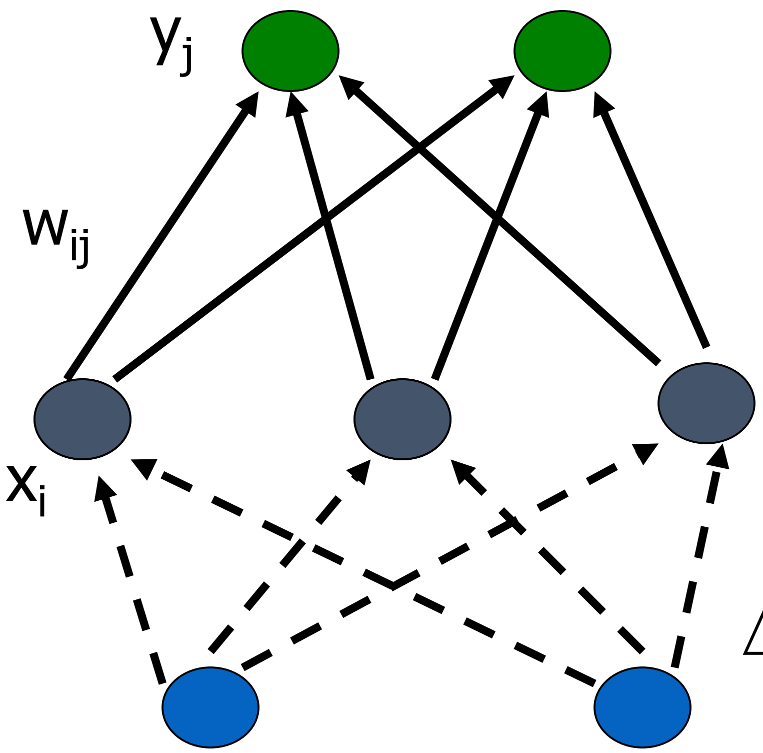
The idea behind backpropagation

- 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



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

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

Derivative of activation function

Learning rate

Error of post-synaptic neuron

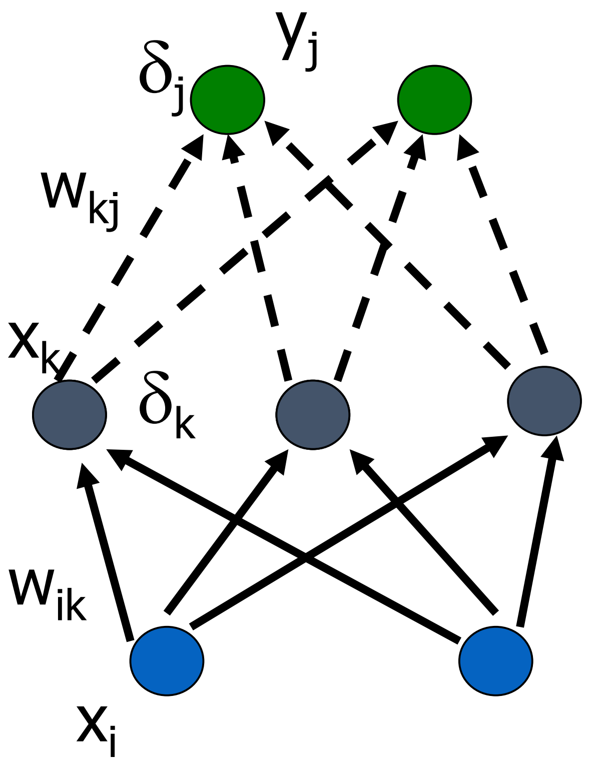
$$\Delta w_{ij} = \alpha \ \ y_j^p(1-y_j^p)\ (t^p_j - y_j^p)\ x_i^p$$

Activation of pre-synaptic neuron

$$= \alpha \ \ \delta_j^p\, x_i^p$$

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

Training Rule for Weights to the Hidden Layer



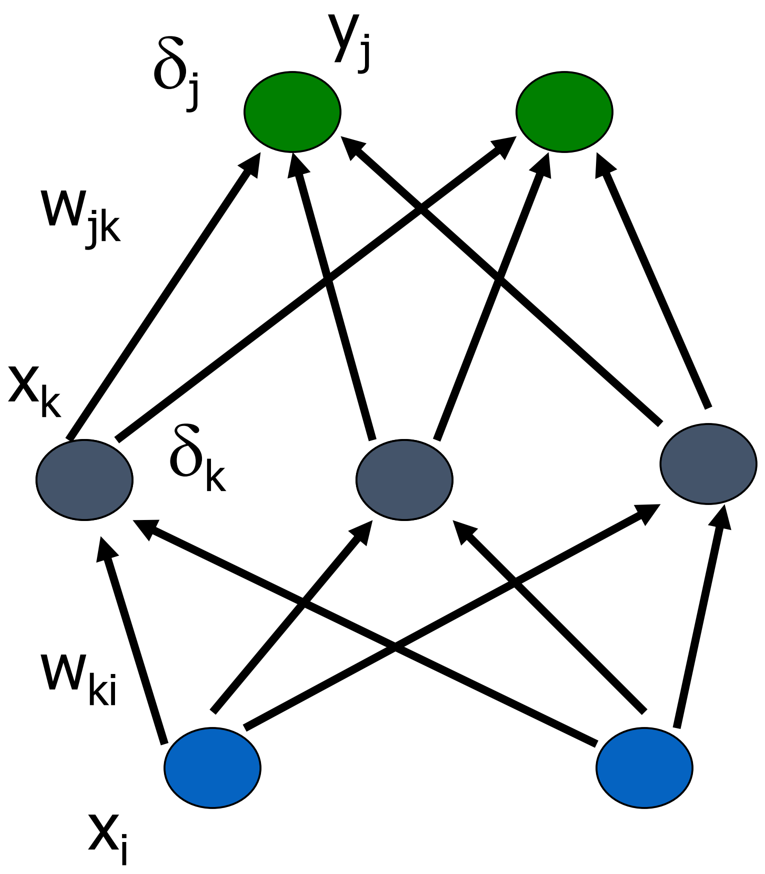
$$E^p[w_{ik}] = ½ \Sigma_j (t_j^p - y_j^p)^2$$

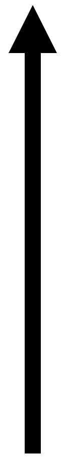
$$\partial E^p/\partial w_{ik} = \partial/\partial w_{ik}\ ½ \Sigma_j (t_j^p - y_j^p)^2$$
$$= \partial/\partial w_{ik}\ ½\Sigma_j (t_j^p - \sigma(\Sigma_k w_{kj}\ x_k^p))^2$$
$$= \partial/\partial w_{ik}\ ½\Sigma_j (t_j^p - \sigma(\Sigma_k w_{kj}\ \sigma(\Sigma_i w_{ik}\ x_i^p)))^2$$
$$= -\Sigma_j (t_j^p - y_j^p)\ \sigma'_j(a)\ w_{jk}\ \sigma'_k(a)\ x_i^p$$
$$= -\Sigma_j \delta_j\ w_{kj}\ \sigma'_k(a)\ x_i^p$$
$$= -\Sigma_j \delta_j\ w_{kj}\ x_k\ (1-x_k)\ x_i^p$$

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

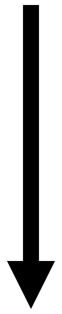
with $\delta_k = \Sigma_j \delta_j\ w_{kj}\ x_k(1-x_k)$

Backpropagation





Backward step:
propagate errors from
output to hidden layer



Forward step:
Propagate activation
from input to output layer

Backpropagation Algorithm

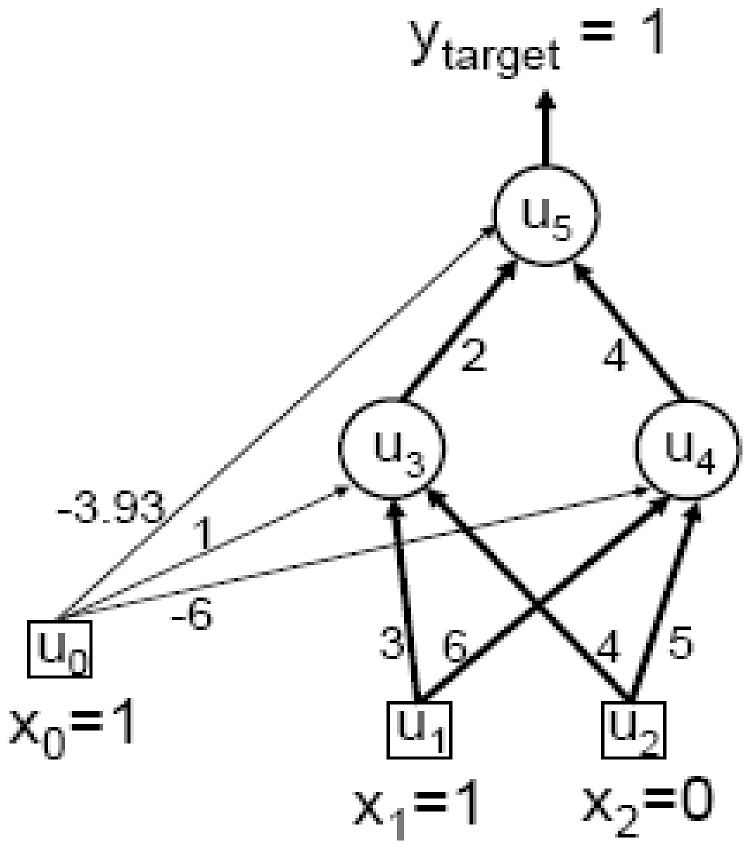
- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - For each training example $<(x_1,\ldots x_n),t>$ Do
    - Input the instance $(x_1,\ldots,x_n)$ to the network and compute the network outputs y_k Forward Pass
    - For each output unit k

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

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

BP: A worked example



Current state:

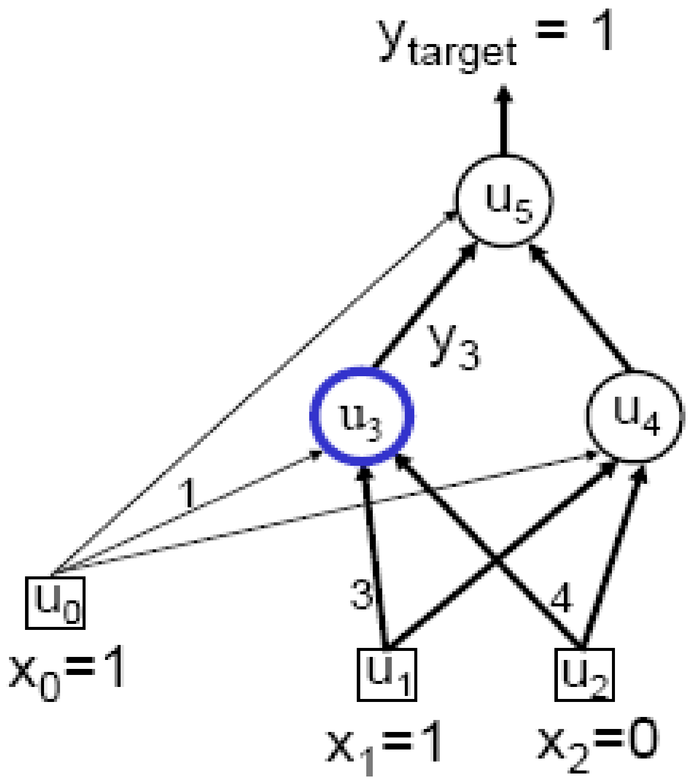
- Weights on arrows e.g. $w_{13} = 3$, $w_{35} = 2$, $w_{24} = 5$
- Bias weights, e.g.

bias for unit 4 (u_4) is w_{04}= -6

Training example (e.g. for logical OR problem):

- Input pattern is x_1=1, x_2=0
- Target output is y_{target}=1

Worked example: Forward Pass



Output for any neuron/unit j can be calculated from:

$$a_j = \sum_i w_{ij} x_i$$

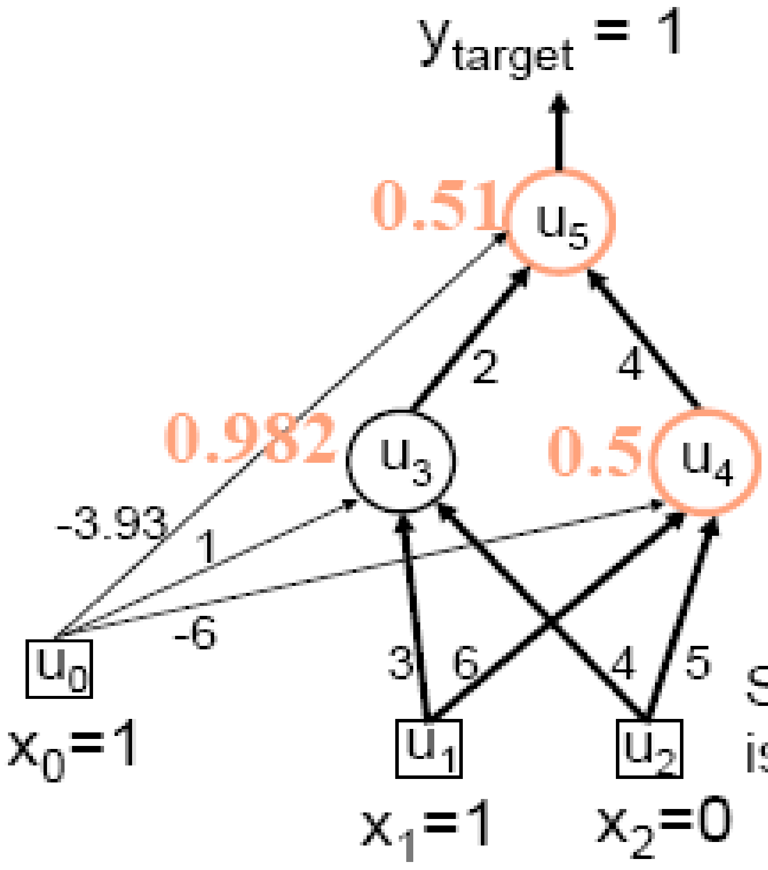
$$y_j = f(a_j) = \frac{1}{1+e^{-a_j}}$$

e.g Calculating output for Neuron/unit 3 in hidden layer:

$$a_3 = 1*1 + 3*1 + 4*0 = 4$$

$$y_3 = f(4) = \frac{1}{1+e^{-4}} = 0.982$$

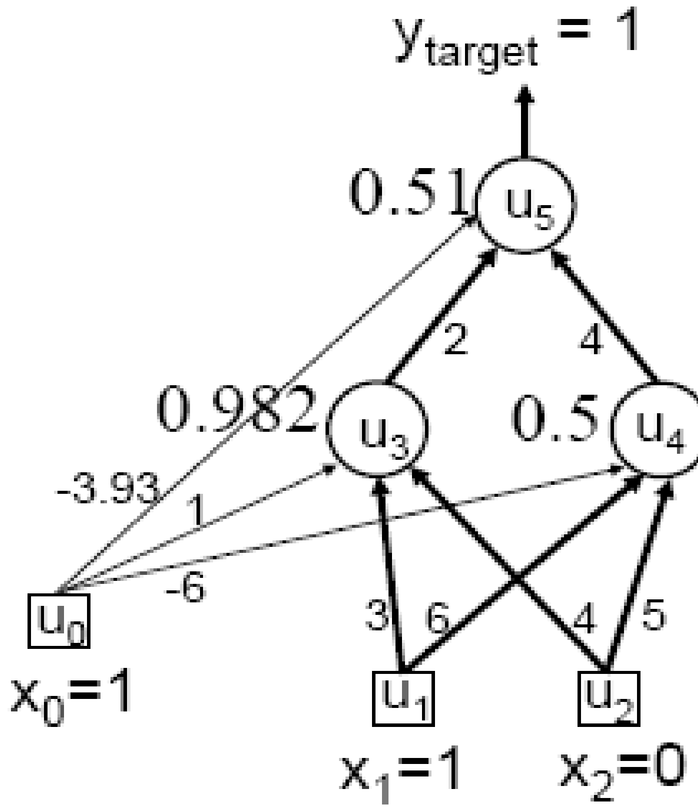
Worked example: Forward Pass



Unit	activation a_j	output y_j
u_3	4.00	0.982
u_4	0.00	0.500
u_5	0.04	**0.510** (network output)

So the error for this training example is: $(y_{target} - y_5) = (1 - 0.510) = 0.490$

Worked example: Backward Pass



Now compute delta values starting at the output:

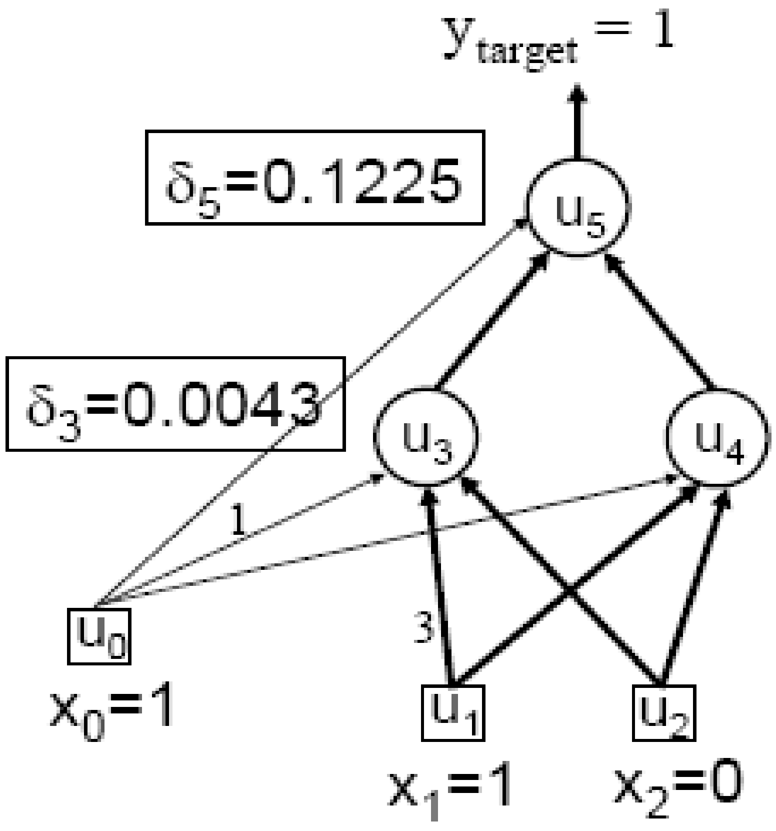
$\delta_5 = y_5(1 - y_5)(y_{target} - y_5)$
$= 0.51(1 - 0.51) \times 0.49$
$= \mathbf{0.1225}$

Then for hidden units:

$\delta_4 = y_4(1 - y_4)\, w_{45}\, \delta_5$
$= 0.5(1 - 0.5) \times 4 \times 0.1225$
$= \mathbf{0.1225}$

$\delta_3 = y_3(1 - y_3)\, w_{35}\, \delta_5$
$= 0.982(1 - 0.982) \times 2 \times 0.1225$
$= \mathbf{0.0043}$

Worked example: Update Weights



- Set learning rate η = 0.1
  Change weights by:
  $$\Delta w_{ij} = \eta \delta_j y_i$$
- e.g.bias weight on u_3:

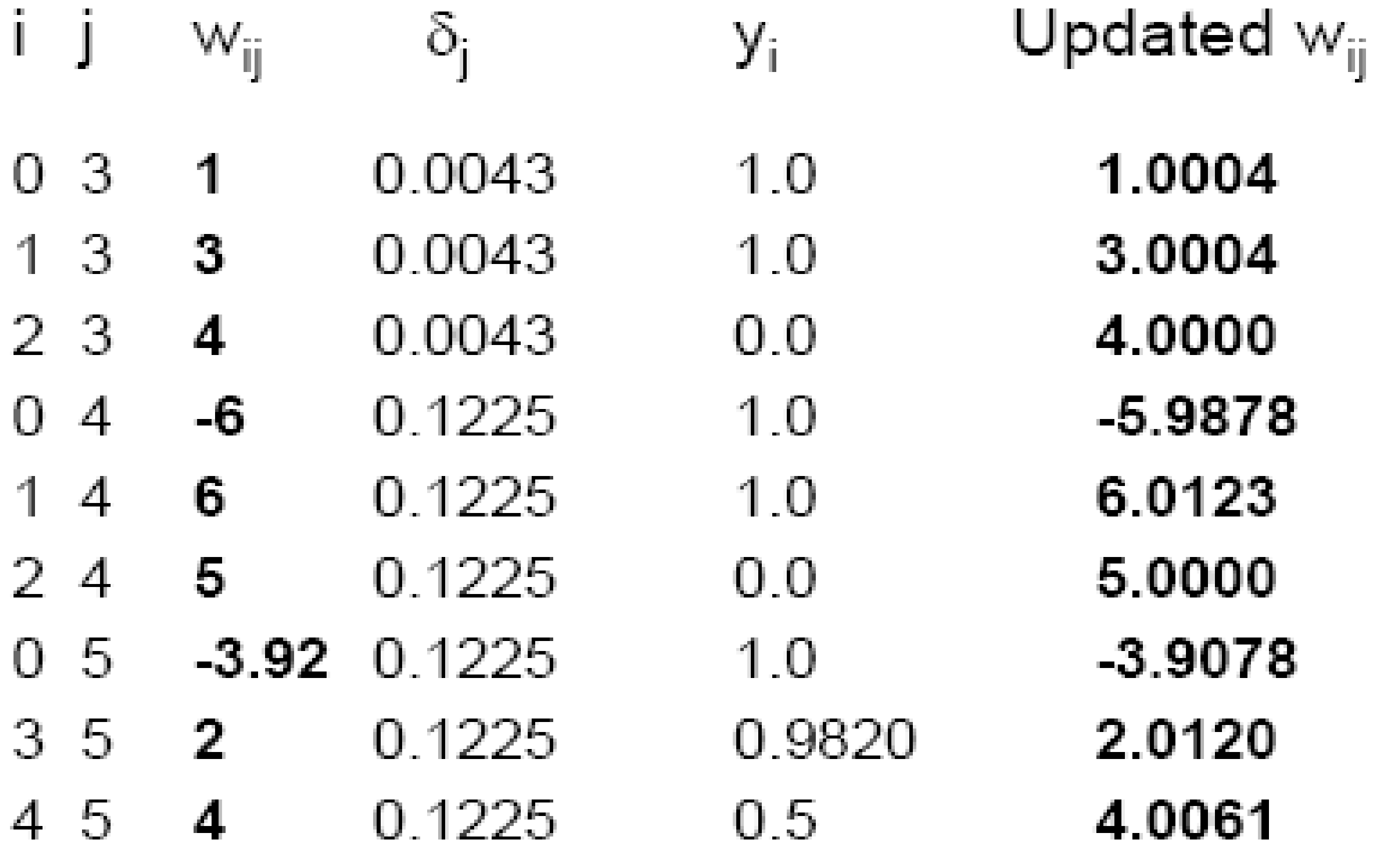
Δw_{03} = $\eta \delta_3 x_0$
=0.1*0.0043*1
= 0.0004

So, new w_{03}' =
w_{03}(old)+Δw_{03}
=1+0.0004=1.0004

- and likewise:

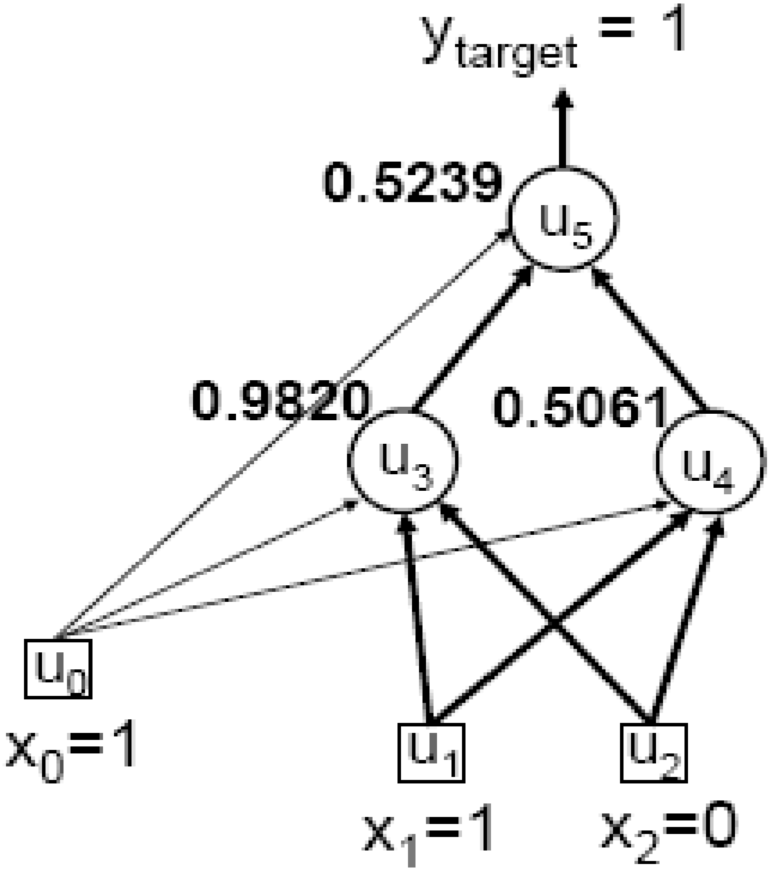
w_{13} ' = 3 + 0.0004
= 3.0004

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 <u>next forward pass</u>:

The new activations are:

$y_3 = f\,(4.0008) = 0.9820$

$y_4 = f\,(0.0245) = 0.5061$

$y_5 = f\,(0.0955) = \mathbf{0.5239}$

Thus the <u>new error</u>

$(y_{target} - y_5)=(1 - 0.5239)=0.476$

<u>has been reduced</u> by **0.014**

(from **0.490** to **0.476**)

Ref: "Neural Network Learning & Expert Systems" by Stephen Gallant

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

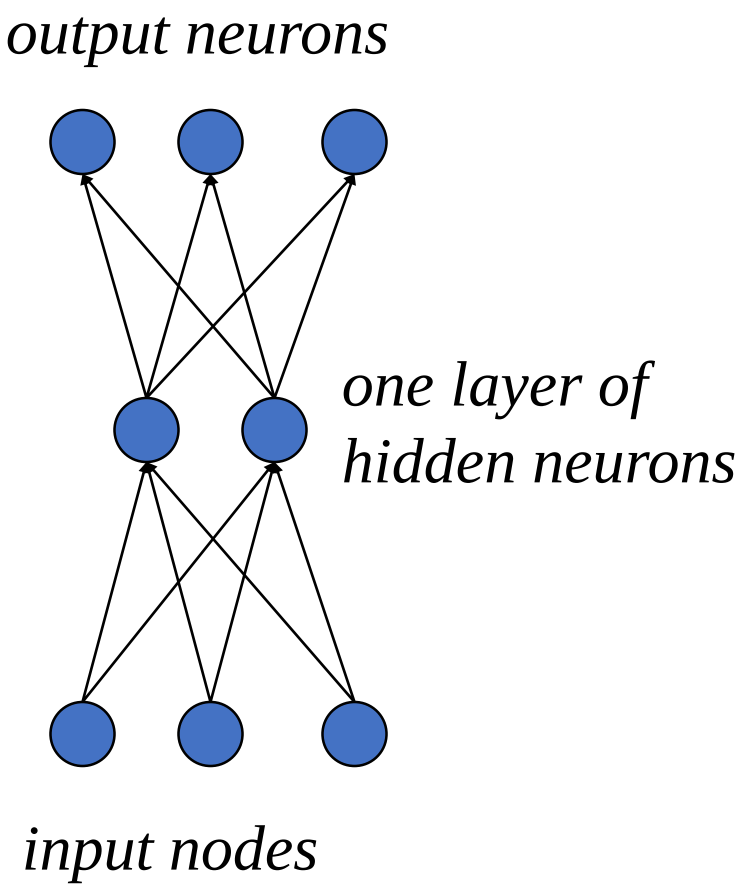
$$\Delta w_{i,j}(n) = \eta\, \delta_j\, x_{i,j} + \alpha\, \Delta w_{i,j}(n-1)$$

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

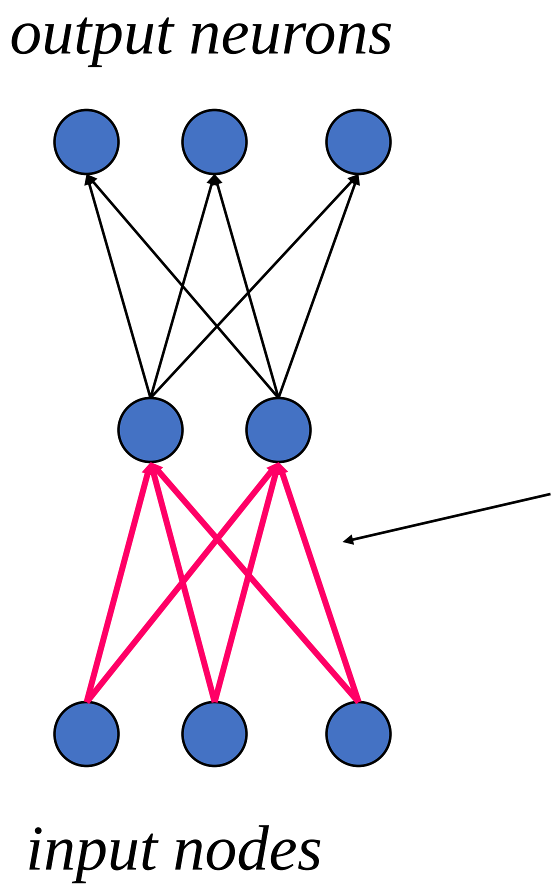
Outline

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

Radial Basis Function Networks



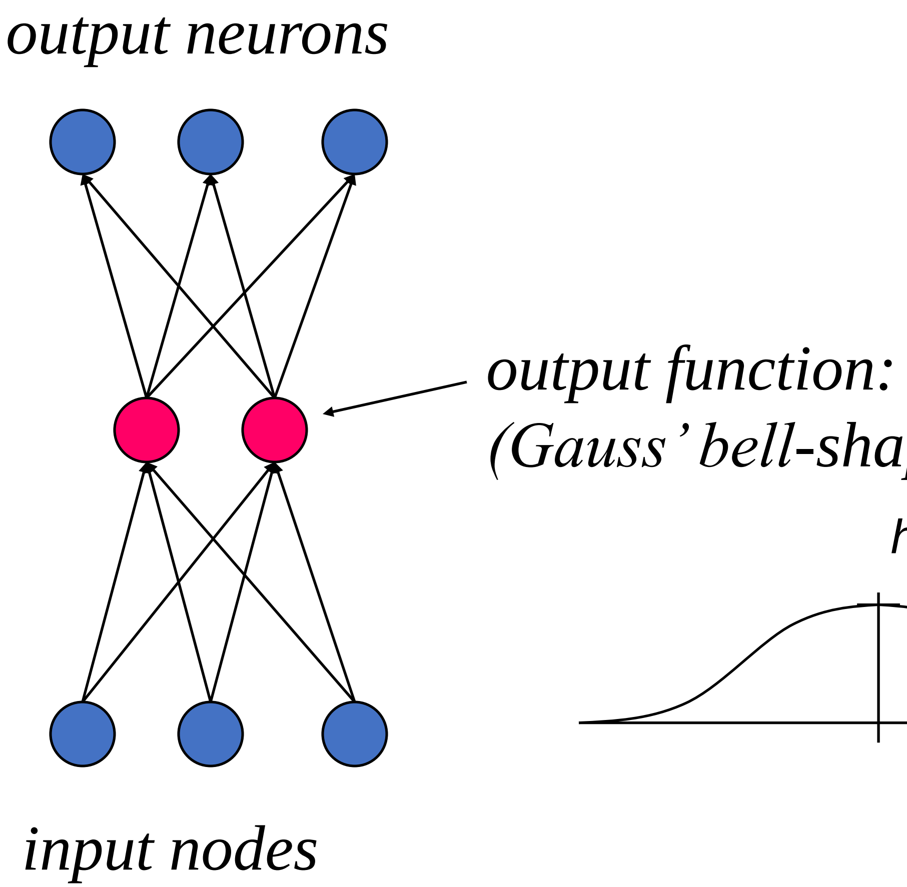
Radial Basis Function Networks

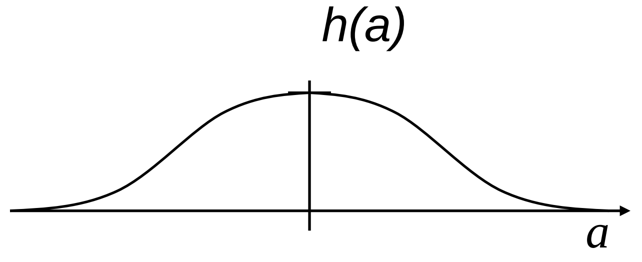


propagation function:

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

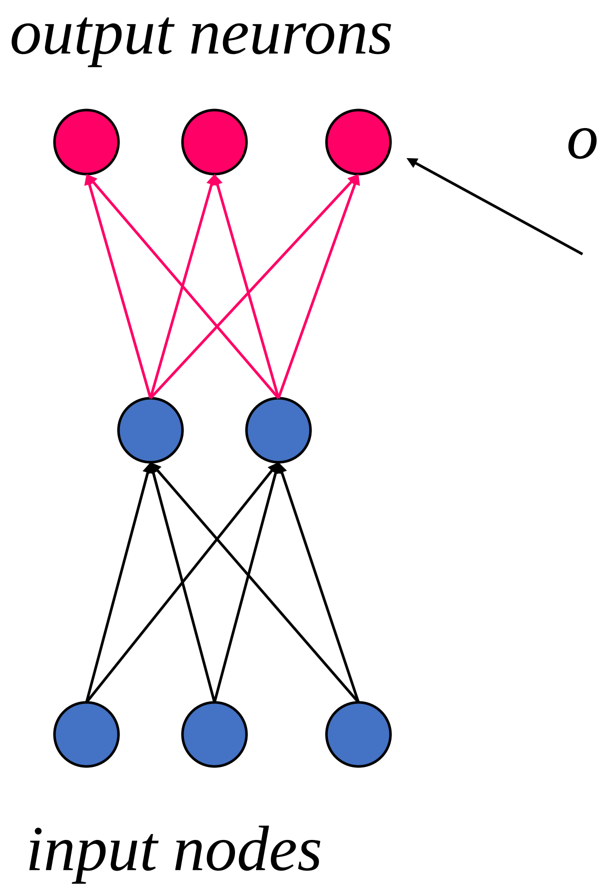
Radial Basis Function Networks





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

Radial Basis Function Networks

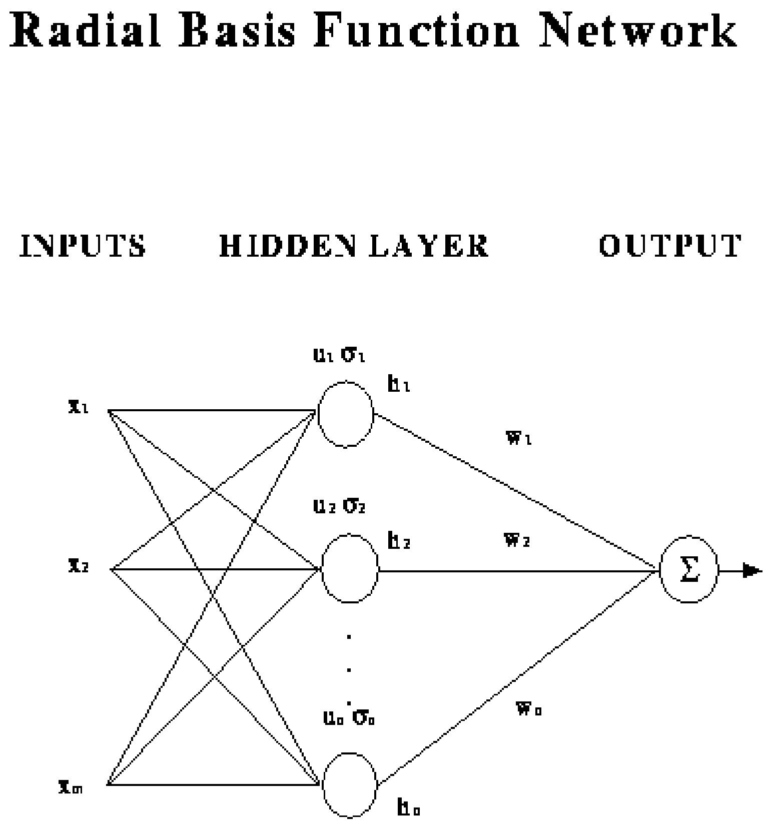


output of network:

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

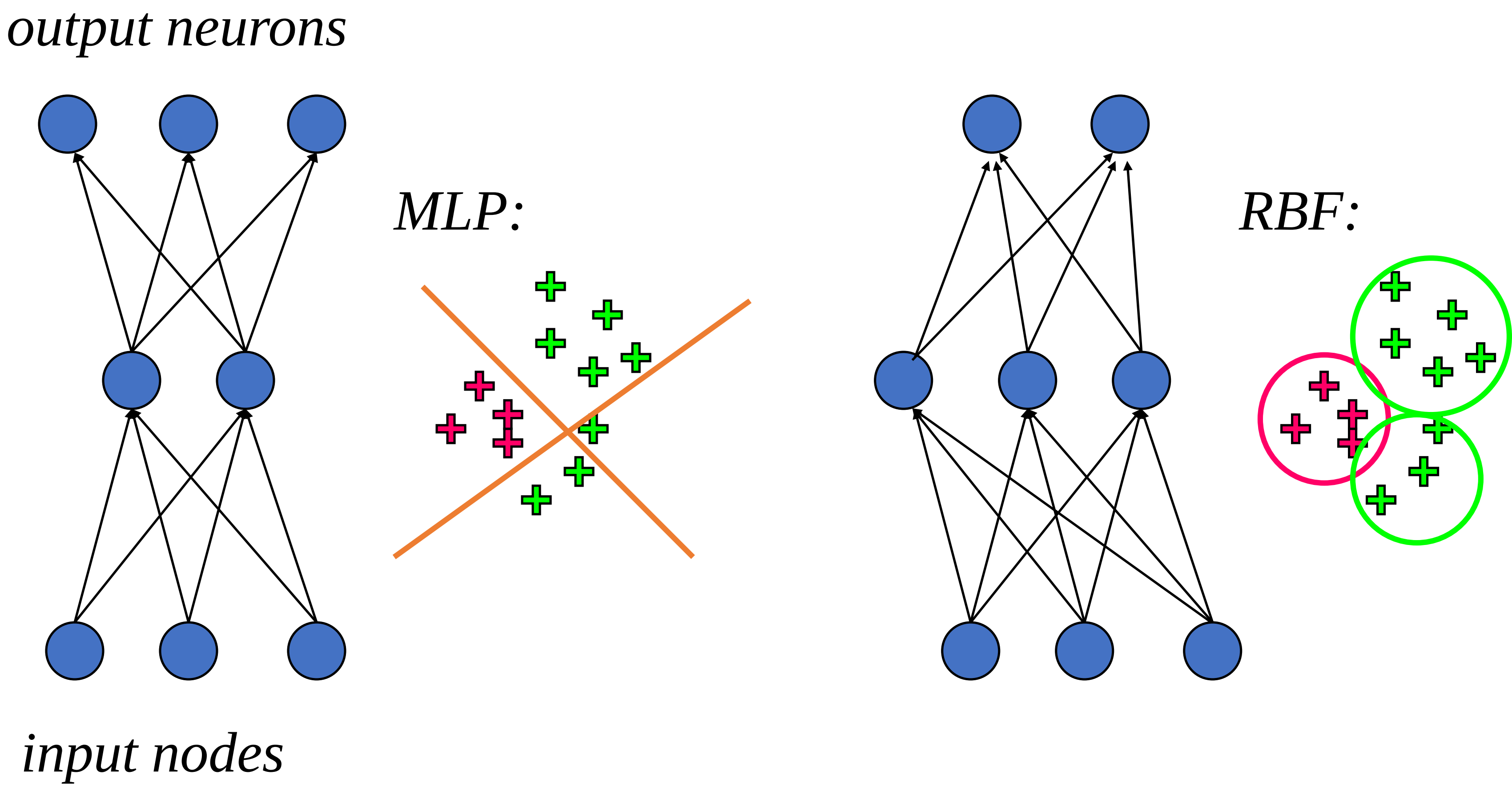
RBF networks

- 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 **x**
- Learn weights w_i, means μ_i, and variances σ_i by minimizing squared error function (gradient descent learning)

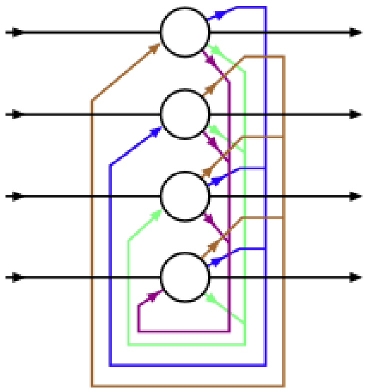


$$h_i = exp[-\frac{(\mathbf{x}-\mathbf{u}_i)^{\mathbf{T}}(\mathbf{x}-\mathbf{u}_i)}{2\sigma^2}],\ y = \sum_i h_i w_i$$

RBF Networks and Multilayer Perceptrons



Recurrent networks

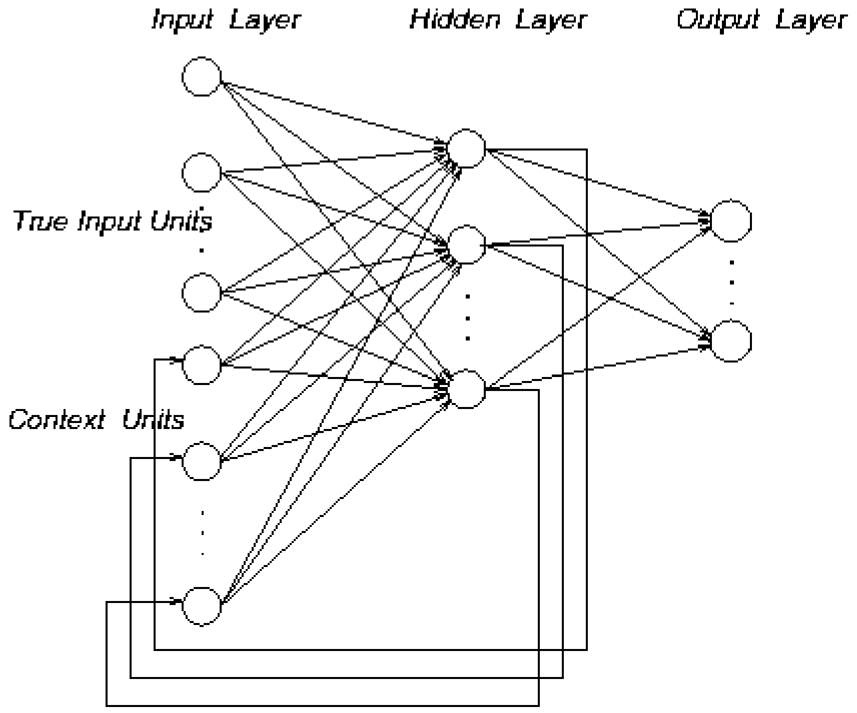


- Employ feedback (positive, negative, or both)
  - Not necessarily stable
    - Symmetric connections can ensure stability
- Why use recurrent networks?
  - Can learn temporal patterns (time series or oscillations)
  - Biologically realistic
    - Majority of connections to neurons in cerebral cortex 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

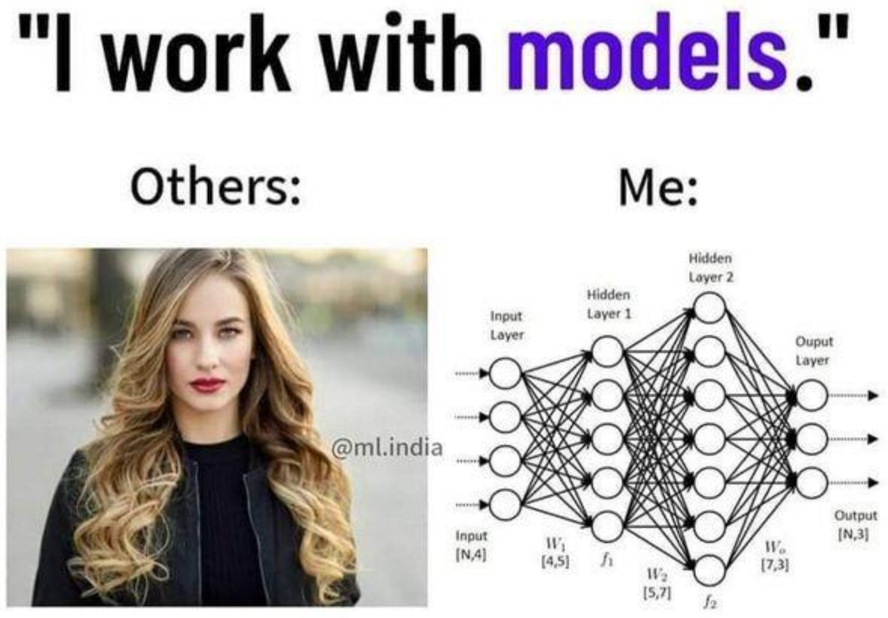
Elman network (1990)



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

Questions?


"I work with models."
Others:
Me:
@ml.india
Input Layer
Hidden Layer 1
Hidden Layer 2
Ouput Layer
Input [N,4]
W_1 [4,5]
f_1
W_2 [5,7]
f_2
W_o [7,3]
Output [N,3]