Chapter 4 Machine Learning

COMP 6721 Introduction of AI

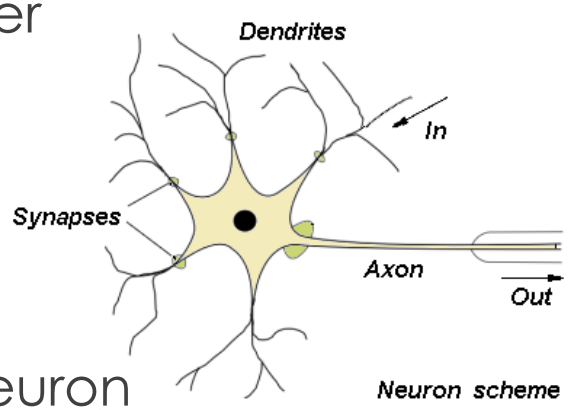
Russell & Norvig – Section 18.1 & 18.2

Neural Networks

- Radically different approach to reasoning and learning
- Inspired by biology
  - the neurons in the human brain
- Set of many simple processing units (neurons) connected together
- Behavior of each neuron is very simple
  - but a collection of neurons can have sophisticated behavior and can be used for complex tasks
- In a neural network, the behavior depends on weights on the connection between the neurons
- The weights will be learned given training data

Biological Neurons

- Human brain =
  - 100 billion neurons
  - each neuron may be connected to 10,000 other neurons
  - passing signals to each other via 1,000 trillion **synapses**
- A neuron is made of:
  - Dendrites: filaments that provide input to the neuron
  - Axon: sends an output signal
  - Synapses: connection with other neurons – releases neurotransmitters to other neurons



树突状体：为神经元提供输入的细丝
轴突：发送输出信号

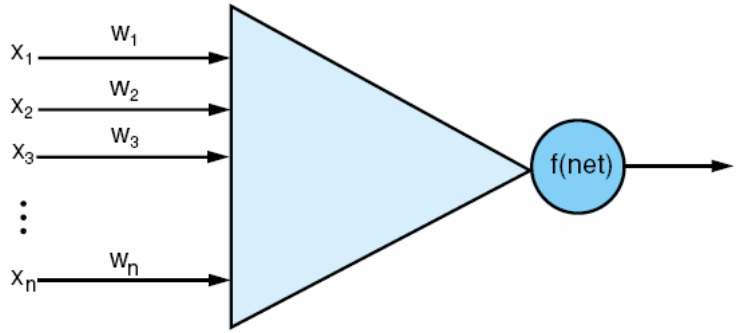
突触：与其他神经元的连接-将神经递质释放到其他神经元

Behavior of a Neuron

- A neuron receives inputs from its neighbors
- If enough inputs are received at the same time:
  - the neuron is activated
  - and fires an output to its neighbors
- Repeated firings across a synapse increases its sensitivity and the future likelihood of its firing 在突触中反复发射会增加其敏感度和未来发射可能性
- If a particular stimulus repeatedly causes activity in a group of neurons, they become strongly associated

  如果特定刺激反复引起一组神经元的活动，它们将紧密相关

A Perceptron

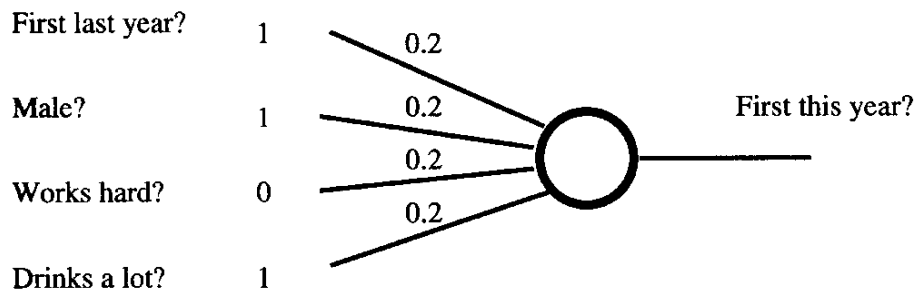


- A single computational neuron (no network yet…)
- Input:
  - input signals x_i
  - weights w_i for each feature x_i
    - represents the strength of the connection with the neighboring neurons
- Output:
  - if sum of input weights >= some threshold, neuron fires (output=1)
  - otherwise output = 0
    - If $(w_1 x_1 + \ldots + w_n x_n) >= \mathbf{t}$
    - Then output = 1
    - Else output = 0
- Learning :
  - use the training data to adjust the weights in the percetron

source: Luger (2005)

The idea

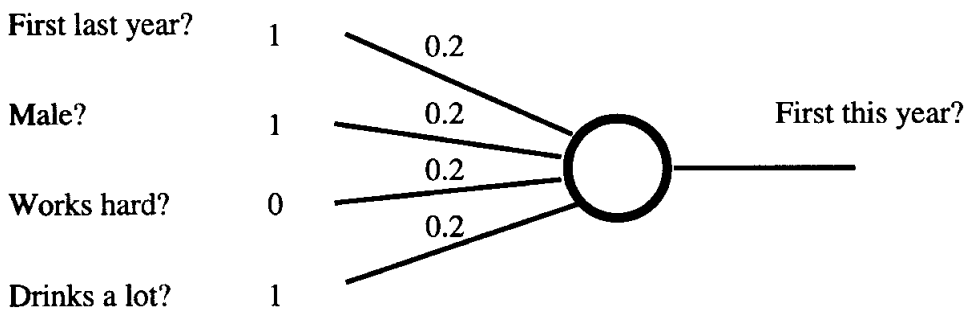
- Step 1: Set weights to random values
- Step 2: Feed perceptron with a set of inputs
- Step 3: Compute the network outputs
- Step 4: Adjust the weights
  - if output correct → weights stay the same
  - if output = 0 but it should be 1 →
    increase weights on *active* connections
  - if output = 1 but should be 0 →
    decrease weights on *active* connections
- Step 5: Repeat steps 2 to 4 a large number of times until the network converges to the right results for the given training examples



	Features (x_i)				Output
Student	First last year?	Male?	Works hard?	Drinks?	**First this year?**
Richard	Yes	Yes	No	Yes	**No**
Alan	Yes	Yes	Yes	No	**Yes**
...					

A Simple Example

- Each feature (works hard, male, …) is an x_i
  - if $x_1 = 1$, then student got an A last year,
  - if $x_1 = 0$, then student did not get an A last year,
  - …
- Initially, set all weights to random values (all 0.2 here)
- Assume:
  - threshold = 0.55
  - constant learning rate = 0.05



A Simple Example (2)

	Features (x_i)				Output
Student	'A' last year?	Male?	Works hard?	Drinks?	**'A' this year?**
Richard	Yes	Yes	No	Yes	**No**
Alan	Yes	Yes	Yes	No	**Yes**
Alison	No	No	Yes	No	**No**
Jeff	No	Yes	No	Yes	**No**
Gail	Yes	No	Yes	Yes	**Yes**
Simon	No	Yes	Yes	Yes	**No**

- Richard:
  - $(1 \times 0.2) + (1 \times 0.2) + (0 \times 0.2) + (1 \times 0.2) = 0.6 >= 0.55$-->output is 1
  - …but he did not get an A this year
  - So reduce weights of all active connections (with input 1) by 0.05.

    Do not change the weight on the inactive connections.
  - So we get $w_1 = 0.15$, $w_2 = 0.15$, $w_3 = 0.2$, $w_4 = 0.15$

A Simple Example (3)

	Features (x_i)				Output
Student	'A' last year?	Male?	Works hard?	Drinks?	**'A' this year?**
Richard	Yes	Yes	No	Yes	**No**
Alan	Yes	Yes	Yes	No	**Yes**
Alison	No	No	Yes	No	**No**
Jeff	No	Yes	No	Yes	**No**
Gail	Yes	No	Yes	Yes	**Yes**
Simon	No	Yes	Yes	Yes	**No**

- Alan:
  - $(1 \times 0.15) + (1 \times 0.15) + (1 \times 0.2) + (0 \times 0.15) = 0.5 < 0.55$ → output is 0
  - … but he got an A this year
  - So increase all weights of active connections by 0.05
  - So we get $w_1 = 0.2$, $w_2 = 0.2$, $w_3 = 0.25$, $w_4 = 0.15$
- Alison…Jeff …Gail … Simon…Richard…Alan…
- After 2 iterations over the training set, we get: $w_1 = 0.2$ $w_2 = 0.1$ $w_3 = 0.25$ $w_4 = 0.1$

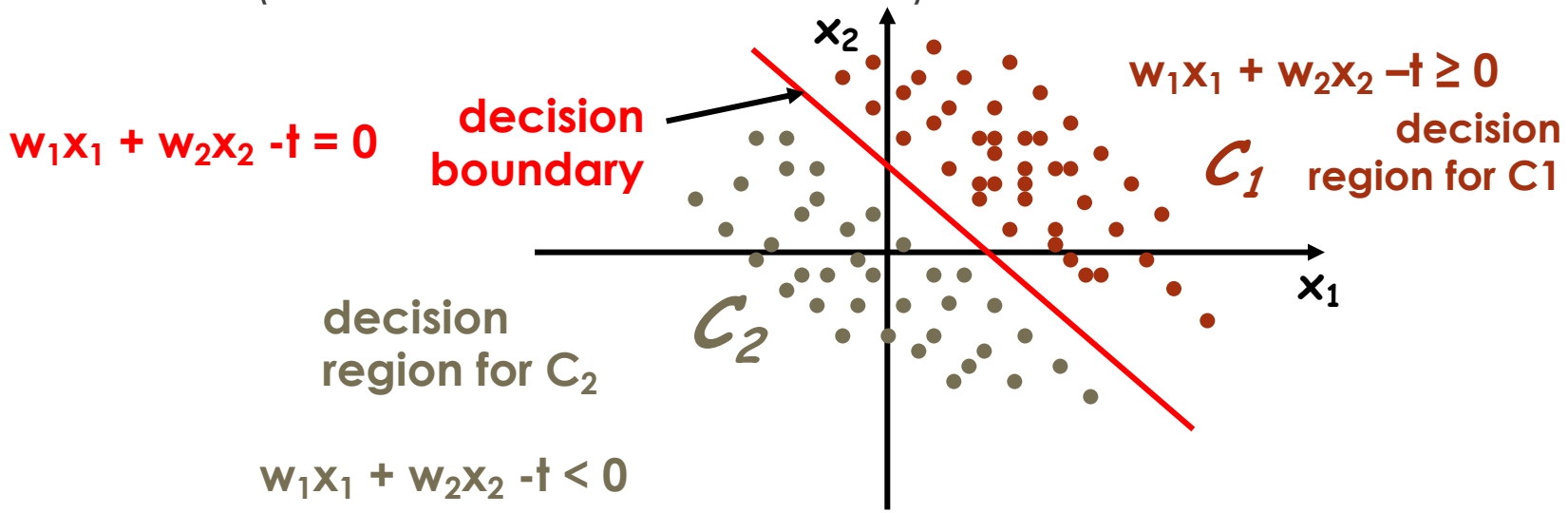
A Simple Example (3)

	Features (x_i)				Output
Student	'A' last year?	Male?	Works hard?	Drinks?	**'A' this year?**
Richard	Yes	Yes	No	Yes	**No**
Alan	Yes	Yes	Yes	No	**Yes**
Alison	No	No	Yes	No	**No**
Jeff	No	Yes	No	Yes	**No**
Gail	Yes	No	Yes	Yes	**Yes**
Simon	No	Yes	Yes	Yes	**No**

- Let's check… (w_1= 0.2 w_2= 0.1 w_3= 0.25 w_4= 0.1)
  - Richard: (1×0.2) + (1×0.1) + (0×0.25) + (1×0.1) = 0.4 < 0.55 -> output is 0 ✓
  - Alan: (1×0.2) + (1×0.1) + (1×0.25) + (1×0.1) = 0.55 ≥ 0.55 -> output is 1 ✓
  - Alison: (0×0.2) + (0×0.1) + (1×0.25) + (0×0.1) = 0.35 < 0.55 -> output is 0 ✓
  - Jeff: (0×0.2) + (1×0.1) + (0×0.25) + (1×0.1) = 0.2 < 0.55 -> output is 0 ✓
  - Gail: (1×0.2) + (0×0.1) + (1×0.25) + (1×0.1) = 0.55 ≥ 0.55 -> output is 1 ✓
  - Simon: (0×0.2) + (1×0.1) + (1×0.25) + (1×0.1) = 0.45 < 0.55 -> output is 0 ✓

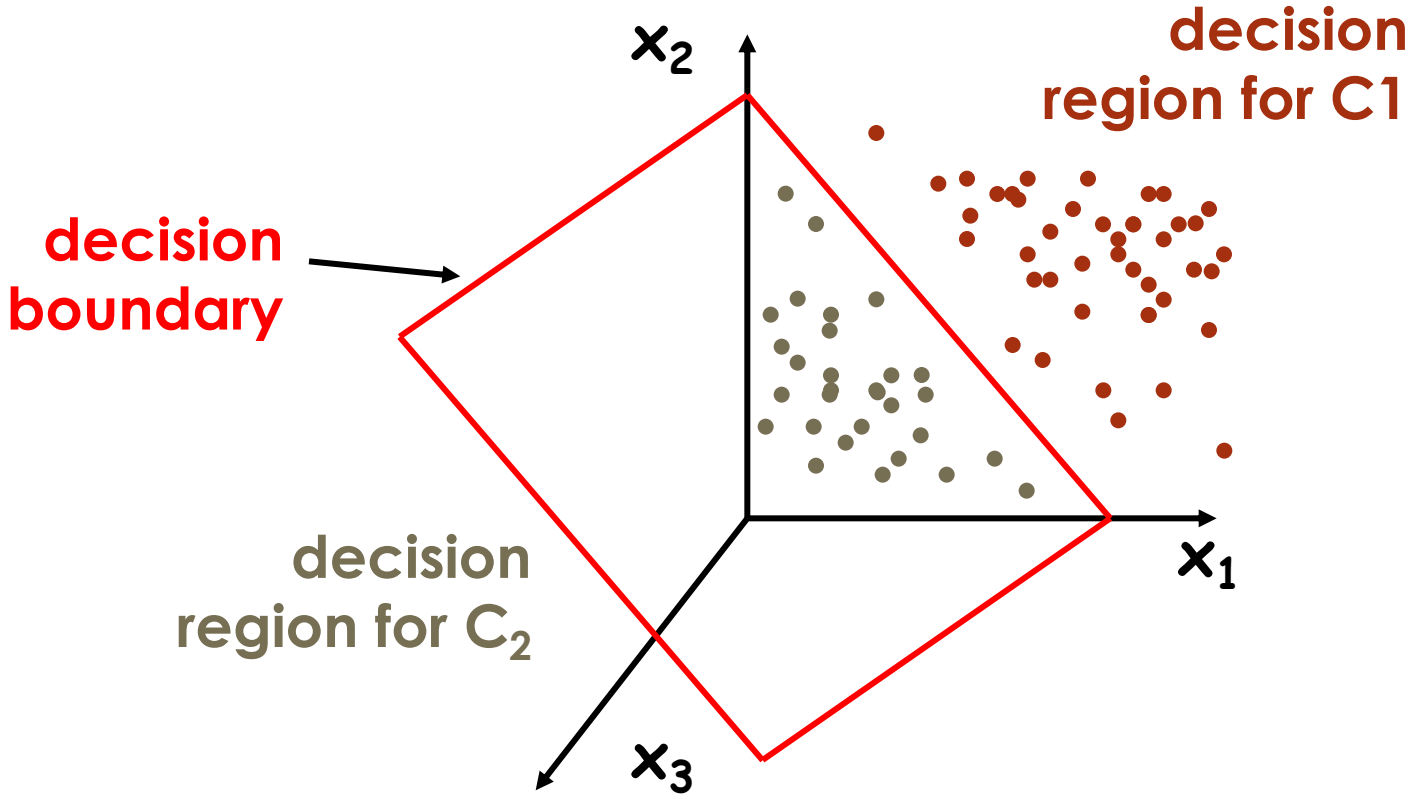
Decision Boundaries of Perceptrons

- So we have just learned the function:
  - If ($0.2x_1 + 0.1x_2 + 0.25x_3 + 0.1x_4 \geq 0.55$) then 1 otherwise 0
  - If ($0.2x_1 + 0.1x_2 + 0.25x_3 + 0.1x_4 - 0.55 \geq 0$) then 1 otherwise 0
- Assume we only had 2 features:
  - If ($w_1x_1 + w_2x_2 - t >= 0$) then 1 otherwise 0
  - The learned function describes a line in the input space
  - This line is used to separate the two classes C1 and C2
  - t (the threshold, later called 'b') is used to shift the line on the horizontal axis



Decision Boundaries of Perceptrons

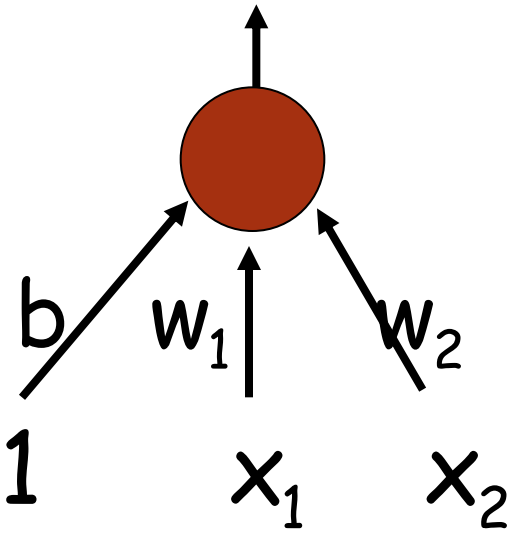
- More generally, with n features, the learned function describes a hyperplane in the input space.



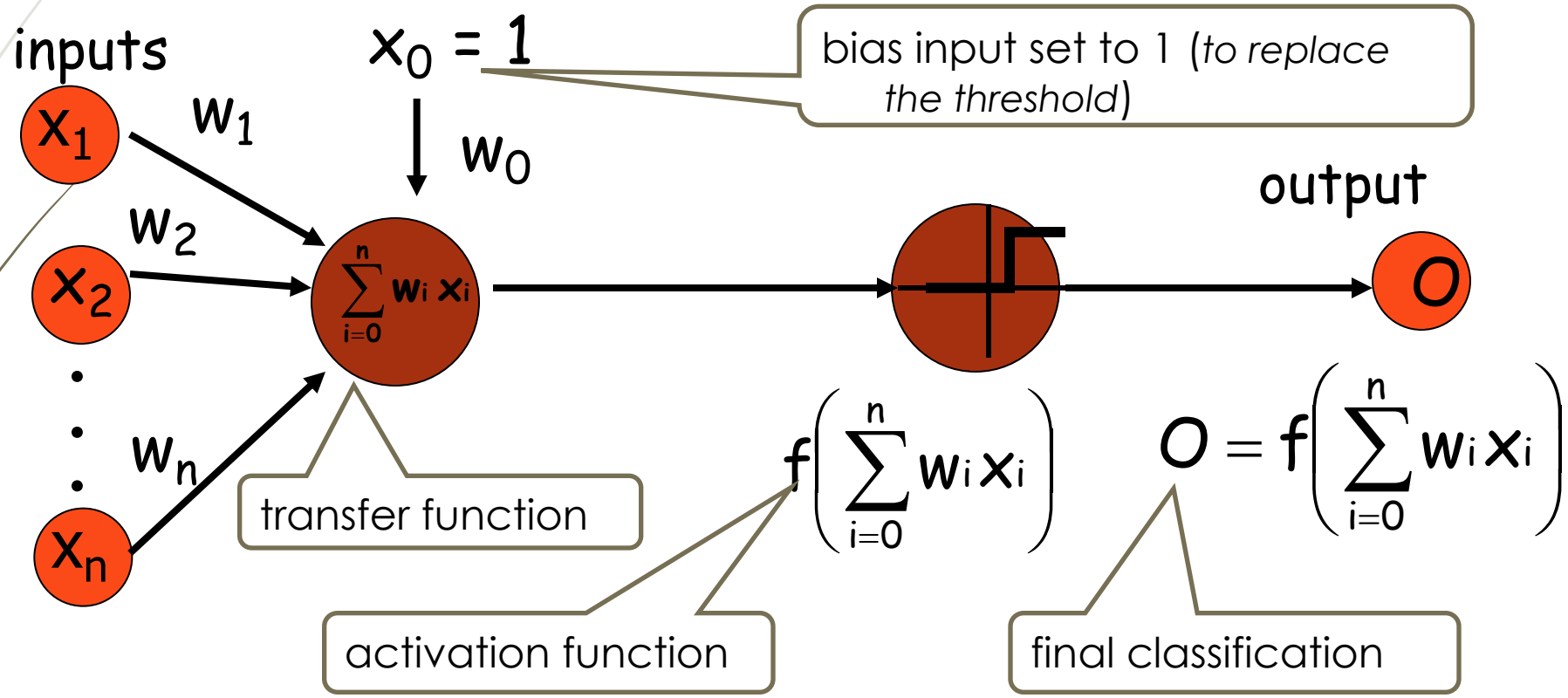
Adding a Bias

- We can avoid having to figure out the threshold by using a "bias"
- A bias is equivalent to a weight on an extra input feature that always has a value of 1.

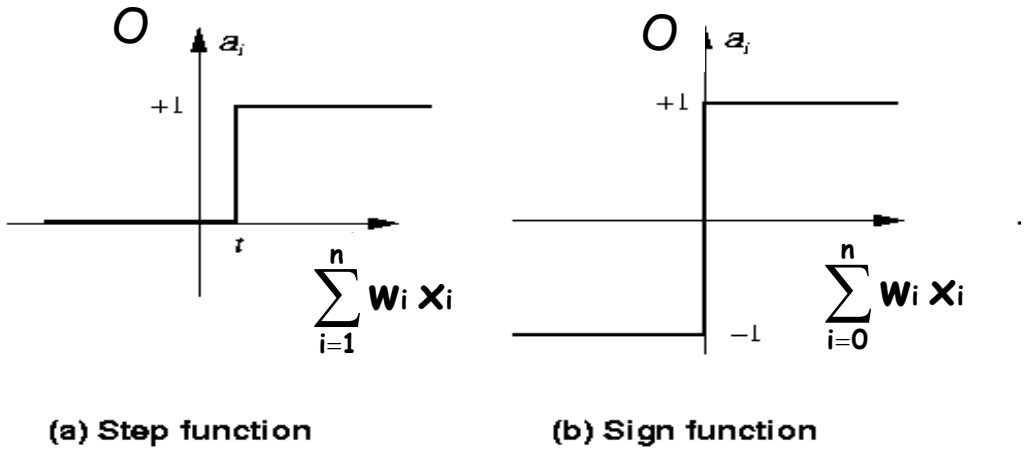
$$b + \sum_i x_i w_i$$



Perceptron - More Generally



Common Activation Functions



(a) Step function (b) Sign function

- Hard Limit activation functions:
  - step $O = \begin{cases} 1 & \text{if } \left(\sum_{i=1}^{n} w_i x_i\right) \geq t \\ 0 & \text{otherwise} \end{cases}$
  - sign $O = \begin{cases} +1 & \text{if } \left(\sum_{i=0}^{n} w_i x_i\right) \geq 0 \\ -1 & \text{otherwise} \end{cases}$

[Russell & Norvig, 1995]

Learning Rate

1. Learning rate can be a constant value (as in the previous example)

learning rate

$$\Delta w = \eta(T - O)$$

Error = target output – actual output

   - So:
     - if T=zero and O=1 (i.e. a false positive) -> decrease w by η
     - if T=1 and O=zero (i.e. a false negative) -> increase w by η
     - if T=O (i.e. no error) -> don't change w
2. Or, a fraction of the input feature x_i

$$\Delta w_i = \eta(T - O)\, x_i$$

value of input feature x_i

   - So the update is proportional to the value of x
     - if T=zero and O=1 (i.e. a false positive) -> decrease w_i by ηx_i
     - if T=1 and O=zero (i.e. a false negative) -> increase w_i by ηx_i
     - if T=O (i.e. no error) -> don't change w_i
   - This is called the ***delta rule*** or ***perceptron learning rule***

Perceptron Convergence Theorem

- Cycle through the set of training examples.
- If a solution with zero error exists,
- The delta rule will find a solution in finite time.

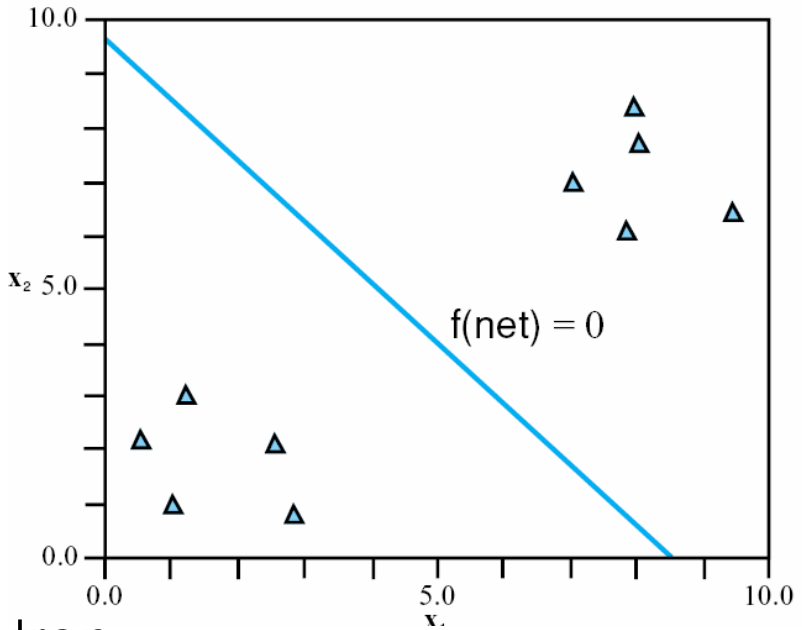
Example of the Delta Rule

training data:

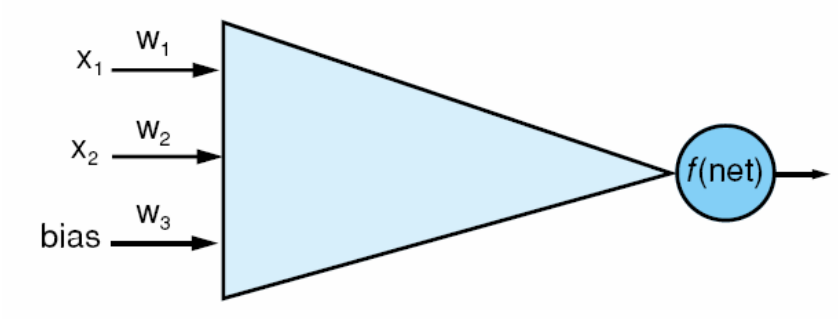
x_1	x_2	Output
1.0	1.0	1
9.4	6.4	−1
2.5	2.1	1
8.0	7.7	−1
0.5	2.2	1
7.9	8.4	−1
7.0	7.0	−1
2.8	0.8	1
1.2	3.0	1
7.8	6.1	−1

source: Luger (2005)

plot of the training data:

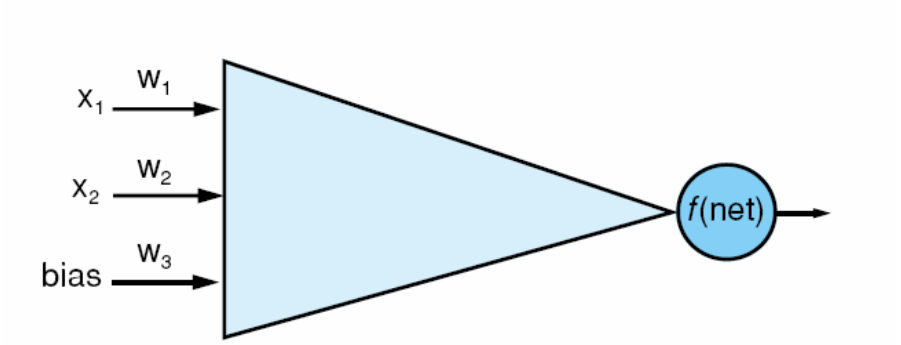


perceptron



Training the Perceptron

- Assume random initialization
  - w1 = 0.75
  - w2 = 0.5
  - w3 = -0.6
- Assume:
  - sign function (threshold = 0)
  - learning rate η = 0.2



source: Luger (2005)

Training

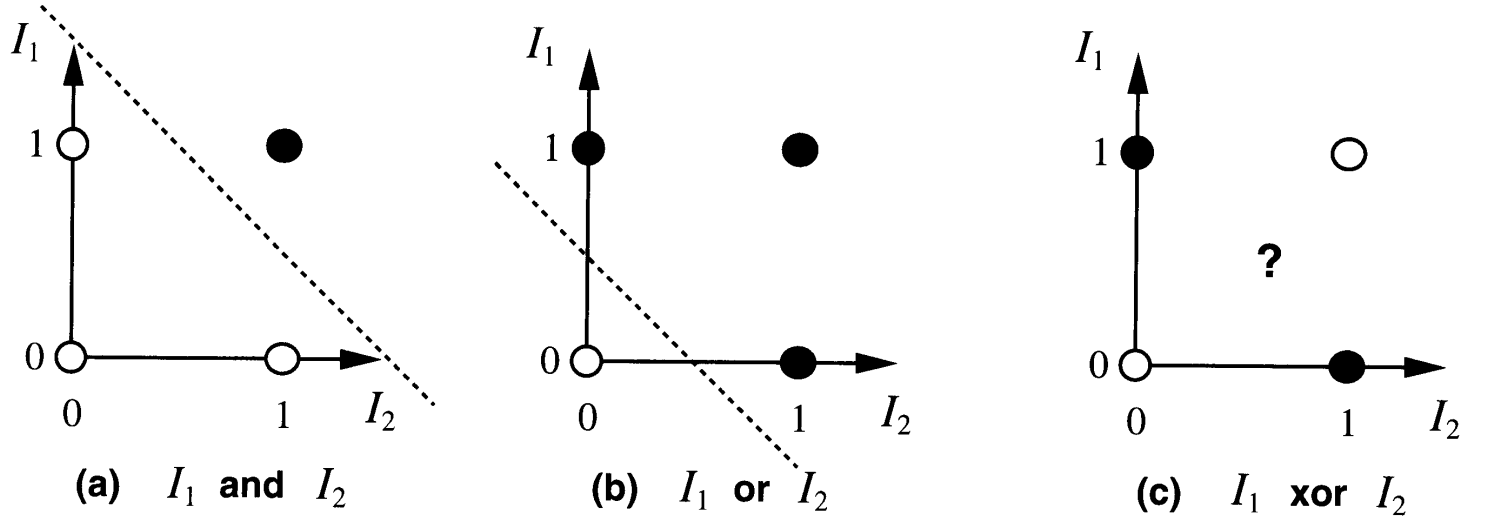
- data #1: f(0.75x1 + 0.5x1 - 0.6x1) = f(0.65) -> 1 ✓
- data #2: f(0.75x9.4 + 0.5x6.4 - 0.6x1) = f(9.65) -> 1 ✗
  - --> error = (-1 - 1) = -2
    - --> $w_1 = w_1$ -2x0.2x9.4 = 0.75 - 3.76 = -3.01
    - --> $w_2 = w_2$ -2x0.2x6.4 = -2.06
    - --> $w_3 = w_3$ -2x0.2x1 = -1.00
- data #3: f(-3.01x2.5 -2.06x2.1 -1x1) = f(-12.84) -> -1 ✗
  - --> error = (1 - -1) = 2
    - --> w_1 = -3.01 + 2x0.2x2.5 = -2.01
    - --> w_2 = -2.06 + 2x0.2x2.1 = -1.22
    - --> w_3 = -1.00 + 2x0.2x1 = -0.60
- repeat… over 500 iterations, we converge to: w_1 = -1.3 w_2 = -1.1 w_3 = 10.9

x_1	x_2	Output
1.0	1.0	1
9.4	6.4	−1
2.5	2.1	1
8.0	7.7	−1
0.5	2.2	1

source: Luger (2005)

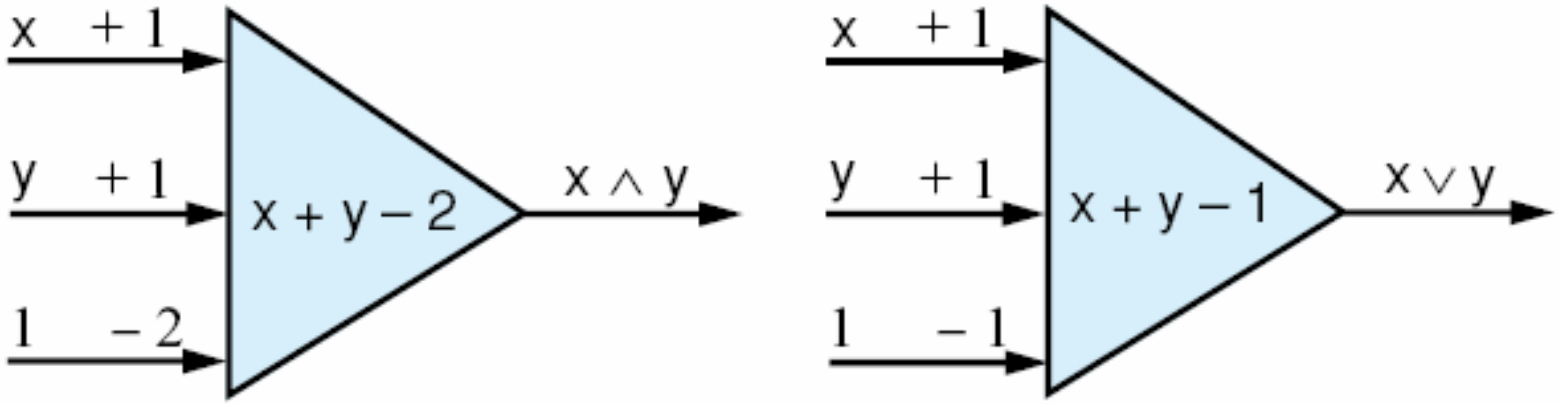
Limits of the Perceptron

- In 1969, Minsky and Papert showed formally what functions could and could not be represented by perceptrons
- Only linearly separable functions can be represented by a perceptron



source: Luger (2005)

AND and OR Perceptrons



x	y	x + y − 2	Output
1	1	0	1
1	0	−1	−1
0	1	−1	−1
0	0	−2	−1

source: Luger (2005)

Example: the XOR Function

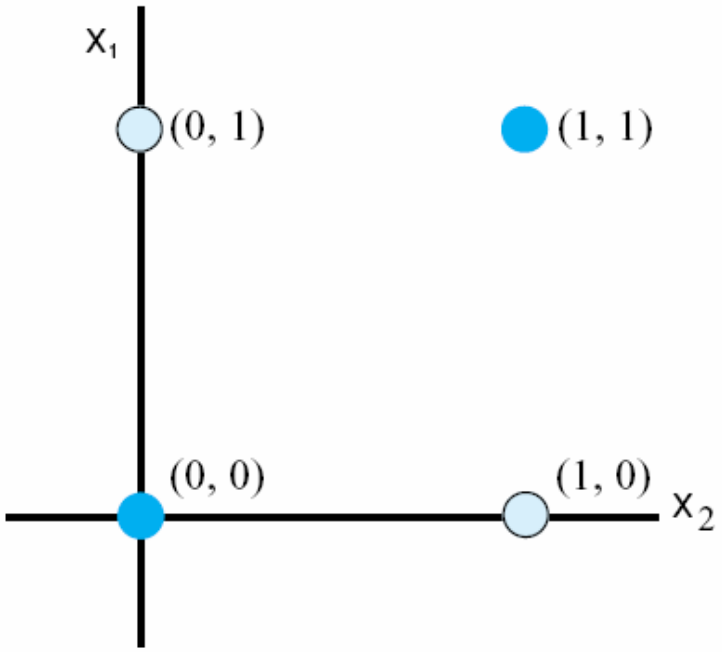
- We cannot build a perceptron to learn the exclusive-or function
- To learn the XOR function, with:
  - two inputs x_1 and x_2
  - two weights w_1 and w_2
  - A threshold t
- i.e. must have:
  - $(1 \times w_1) + (1 \times w_2) < t$ (for the first line of truth table)
  - $(1 \times w_1) + 0 >= t$
  - $0 + (1 \times w_2) >= t$
  - $0 + 0 < t$
- Which has no solution… so a perceptron cannot learn the XOR function

x1	x2	Output
1	1	0
1	0	1
0	1	1
0	0	0

source: Luger (2005)

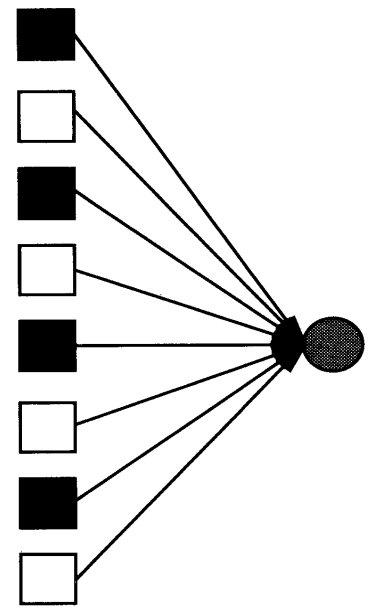
The XOR Function - Visually

- In a 2-dimentional space (2 features for the X)
- No straight line in two-dimensions can separate
  - (0, 1) and (1, 0) from
  - (0, 0) and (1, 1).



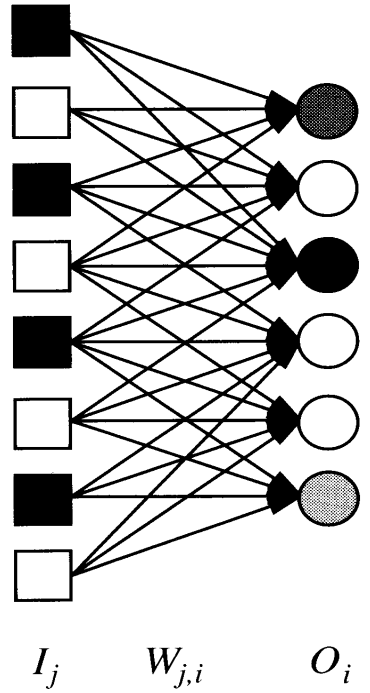
source: Luger (2005)

A Perceptron Network



Single Perceptron

- So far, we looked at a single perceptron
- But if the output needs to learn more than a binary (yes/no) decision
- Ex: learning to recognize digit --> 10 possible outputs --> need a perceptron network

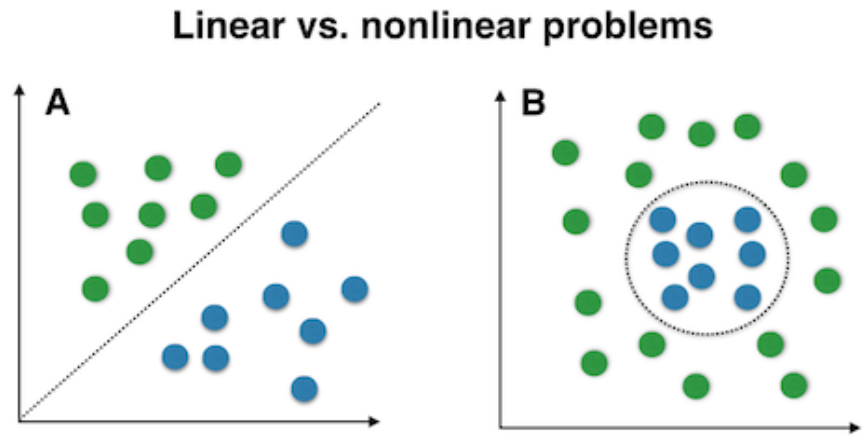


Perceptron Network

source: Luger (2005)

Non-linearly Separable Functions

- Real-world problems cannot always be represented by linearly-separable functions…



- This caused a decrease in interest in neural networks in the 1970's.

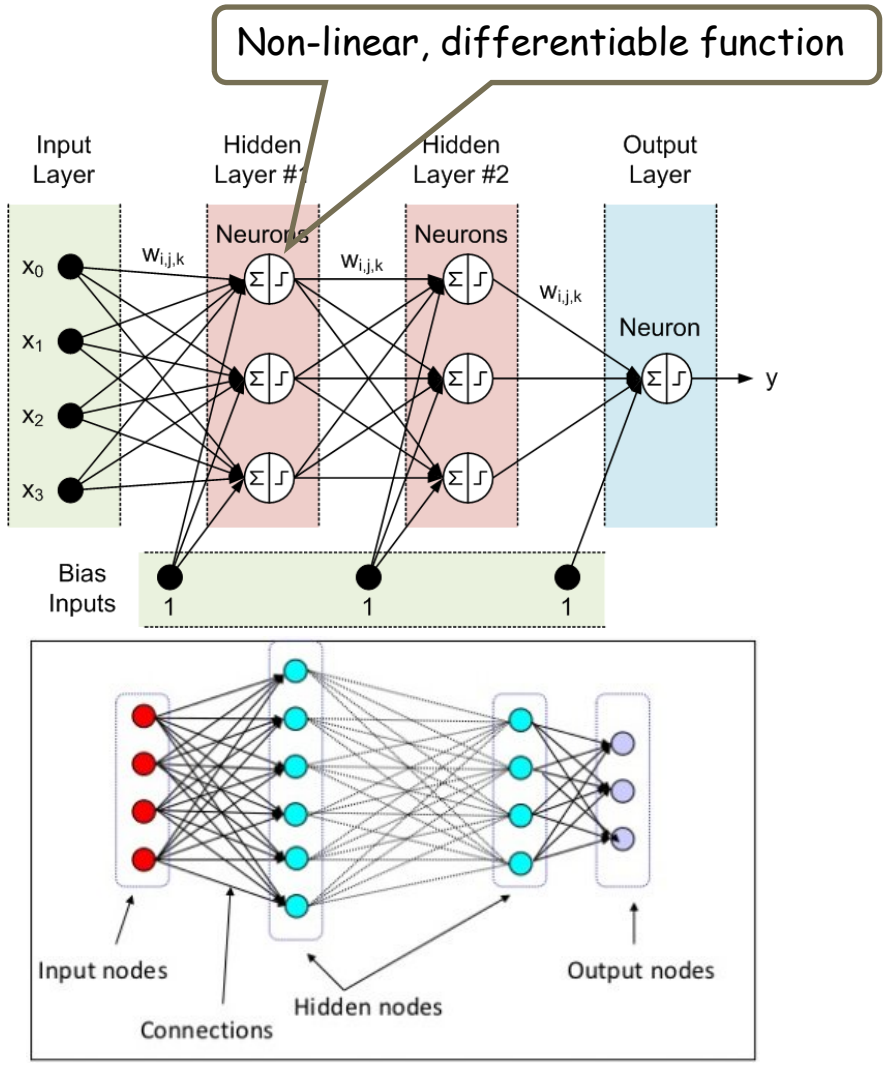
http://sebastianraschka.com/Articles/2014_kernel_pca.html

source: Luger (2005)

Multilayer Neural Networks

- Solution:

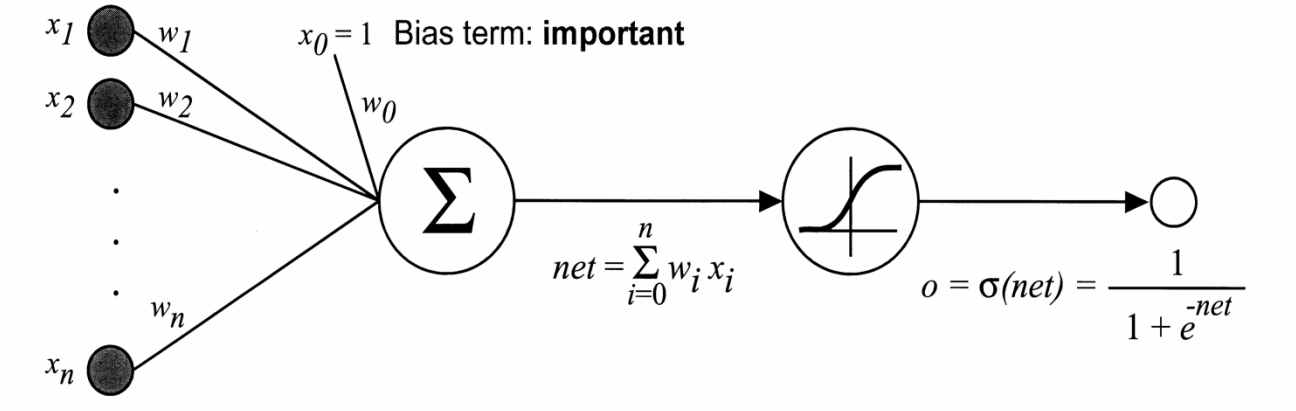
1. use a non-linear activation function
2. to learn more complex functions (more complex decision boundaries), have hidden nodes
3. and for non-binary decisions, have multiple output nodes



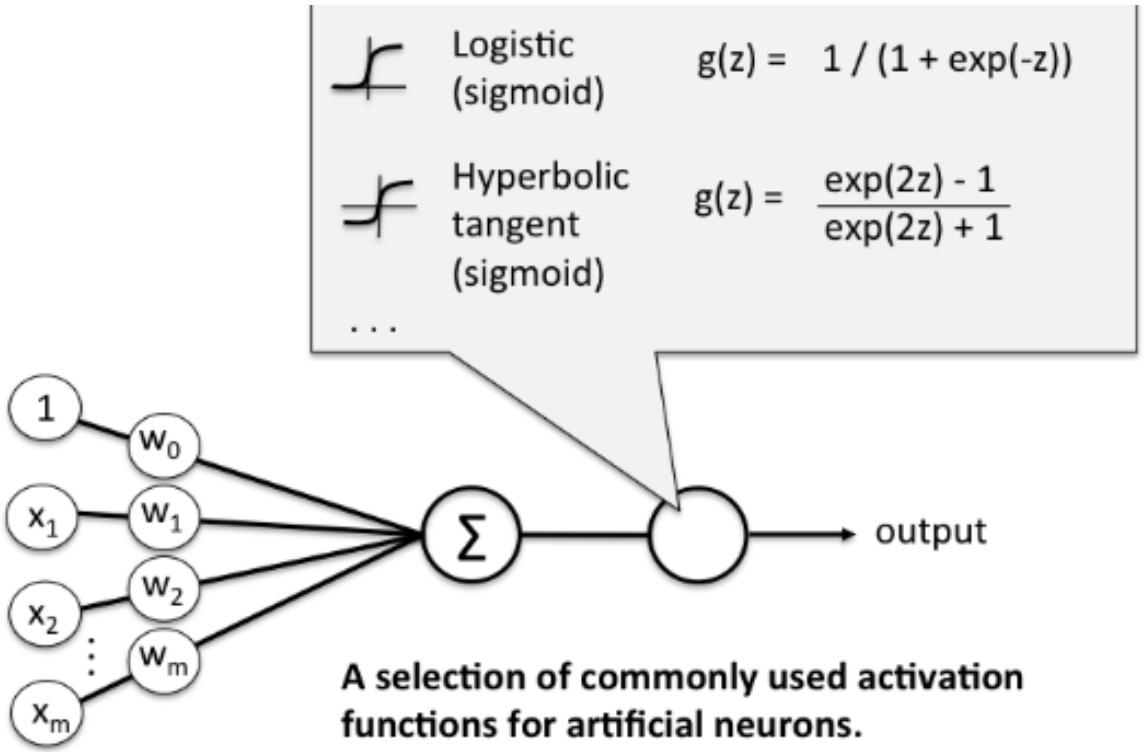
http://www.kdnuggets.com/2016/10/deep-learning-key-terms-explained.html

Activation Functions

- instead of a hard activation function (sign or step)
- We will use one that returns a value *between* 0 and 1 (instead of 0 or 1)
- To indicate how close/how far the output of the network is compared to the right answer (the *error term*)
- Often denoted g or σ (*sigma*)



Typical Activation Functions



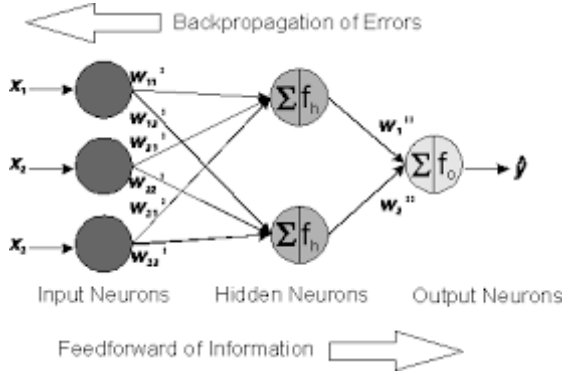
A selection of commonly used activation functions for artificial neurons.

https://medium.com/towards-data-science/activation-functions-and-its-types-which-is-better-a9a5310cc8f

Learning in a Neural Network

- Learning is the same as in a perceptron:
  - feed network with training data
  - if there is an error (a difference between the output and the target), adjust the weights
- So we must assess the blame for an error to the contributing weights

Feed-forward + Backpropagation



- Feed-forward:
  - Input from the features is fed forward in the network from input layer towards the output layer
- Backpropagation:
  - Error rate flows backwards from the output layer to the input layer (to adjust the weights in order to minimize the output error)
- Iterate until error rate is minimized
  - repeat the forward pass and back pass for the next data points until all data points are examined (1 epoch)
  - repeat this entire exercise (several epochs) until the overall error is minimised
  - Eg: MSR = mean squared errors = $\frac{1}{2}\sum_{i=1}^{n}(T_i - O_i) < \varepsilon$

where $\varepsilon \sim 0.001$ and n= nb of training examples

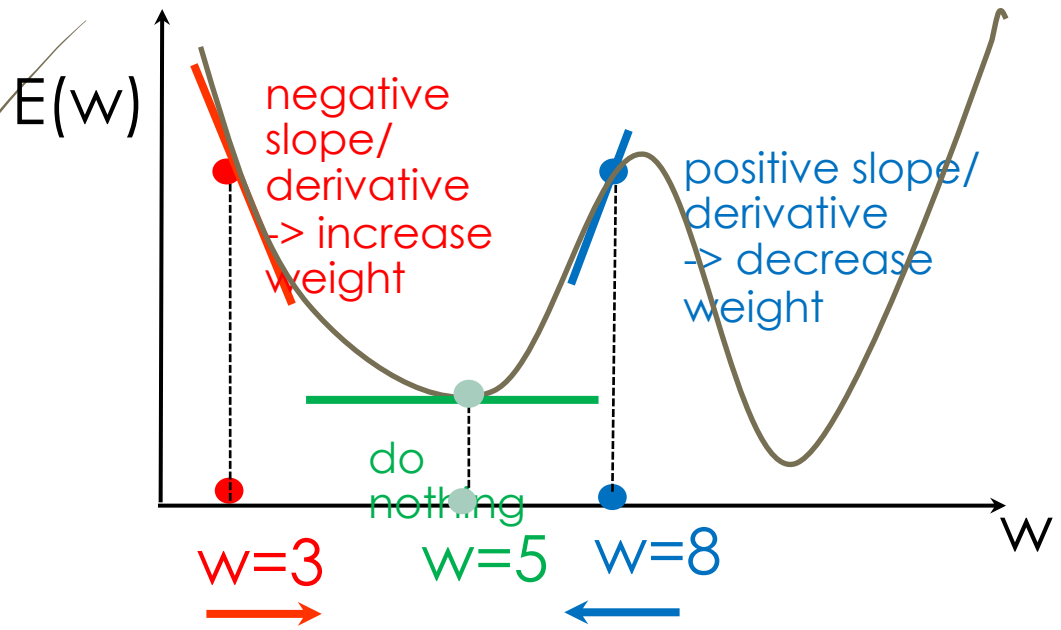
Backpropagation

- In a multilayer network…
  - Computing the error in the output layer is clear.
  - Computing the error in the hidden layer is not clear, because we don't know what output it should be
- Intuitively:
  - A hidden node h is "responsible" for some fraction of the error in each of the output node to which it connects.
  - So the error values (δ):
    - are divided according to the weight of their connection between the hidden node and the output node
    - and are propagated back to provide the error values (δ) for the hidden layer.

Gradients

Gradient is just derivative in 1D

Ex: $E(w) = (w - 5)^2$ derivative is: $\frac{\partial}{\partial w} E = 2(w - 5)$



If **w**=3 $\frac{\partial}{\partial w} E(3) = 2(3 - 5) = -4$

derivative says increase w (go in opposite direction of derivative)

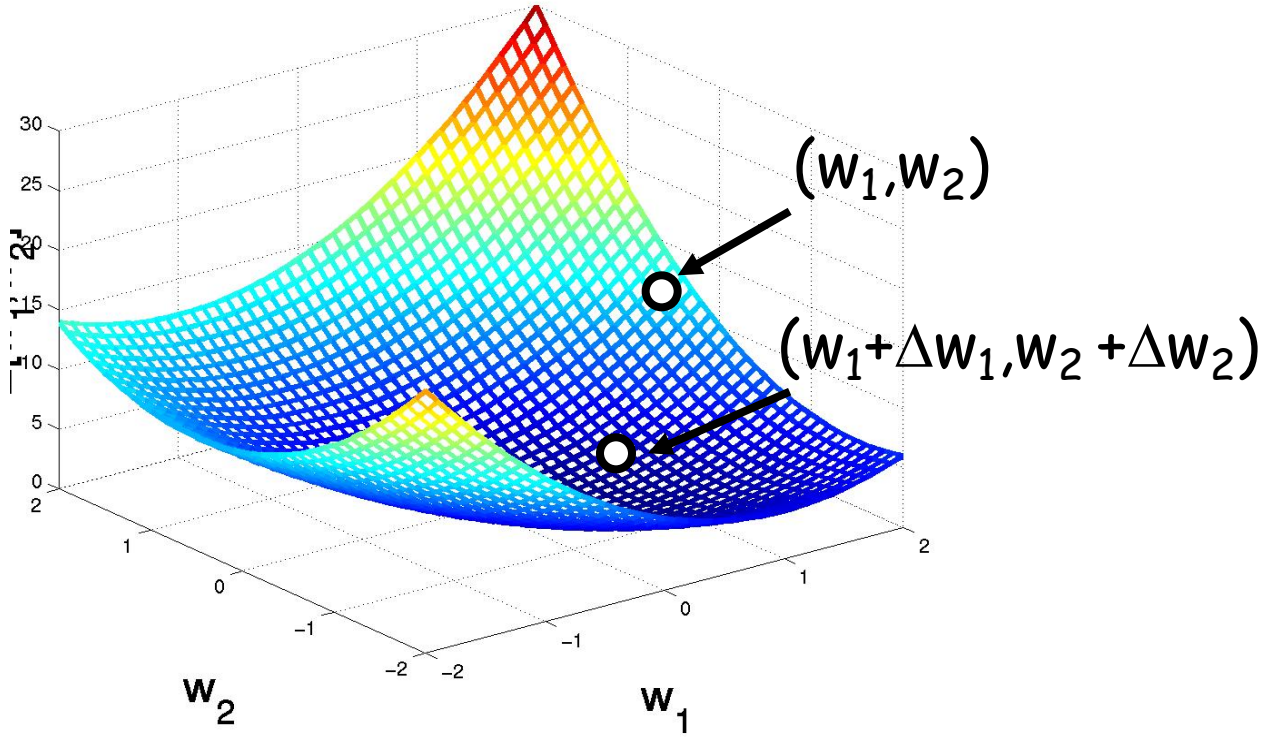
If **w**=8 $\frac{\partial}{\partial w} E(8) = 2(8 - 5) = 6$

derivative says decrease w (go in opposite direction of derivative)

Gradient Descent Visually

E(w1,w2)=

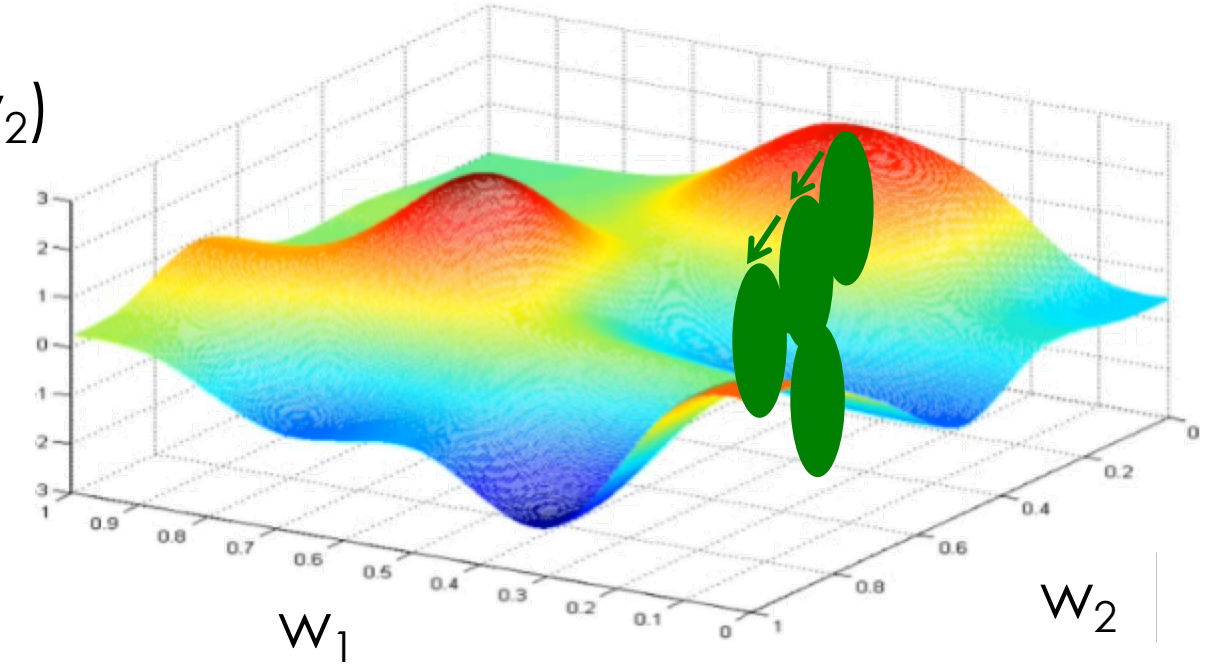
$$\frac{1}{2}\sum_{i=1}^{n}(T_i - O_i)$$



- Goal: minimize E(w1,w2) by changing w1 and w2
- But what is the best combination of change in w1 and w2 to minimize E faster?
- The delta rule is a gradient descent technique for updating the weights in a single-layer perceptron.

Gradient Descent Visually

$E(w_1,w_2)$



Partial derivative (or gradient) of E with respect to w1

- need to know how much a change in w1 will affect E(w1,w2) i.e $\frac{\partial E}{\partial w1}$
- need to know how much a change in w2 will affect E(w1,w2) i.e $\frac{\partial E}{\partial w2}$
- **Gradient ∇E** points in the opposite direction of steepest decrease of E(w1,w2)
- i.e. hill-climbing approach…

Source: Andrew Ng

Training the Network

After a little bit of calculus (see: https://en.wikipedia.org/wiki/Backpropagation we get…

- Step 0: Initialise the weights of the network randomly
- Step 1: Do a forward pass through the network (use sigmoid) // feedforward

$$O_i = g\left(\sum_j w_{ji}\, x_j\right) = \text{sigmoid}\left(\sum_j w_{ji}\, x_j\right) = \frac{1}{1+e^{-\left(\sum_j w_{ji}\, x_j\right)}}$$

// propagate the errors backwards

- Step 2: For each output unit k, calculate its error term δ_k

$$\delta_k \leftarrow g'(x_k) \times Err_k = O_k(1 - O_k) \times (O_k - T_k)$$

Derivative of sigmoid

note, if we use $g = \text{sigmoid}$:
$g'(x) = g(x)\,(1 - g(x))$

- Step 3: For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow g'(x_h) \times Err_h = O_h(1 - O_h) \times \sum_{k \in \text{outputs}} w_{hk}\,\delta_k$$

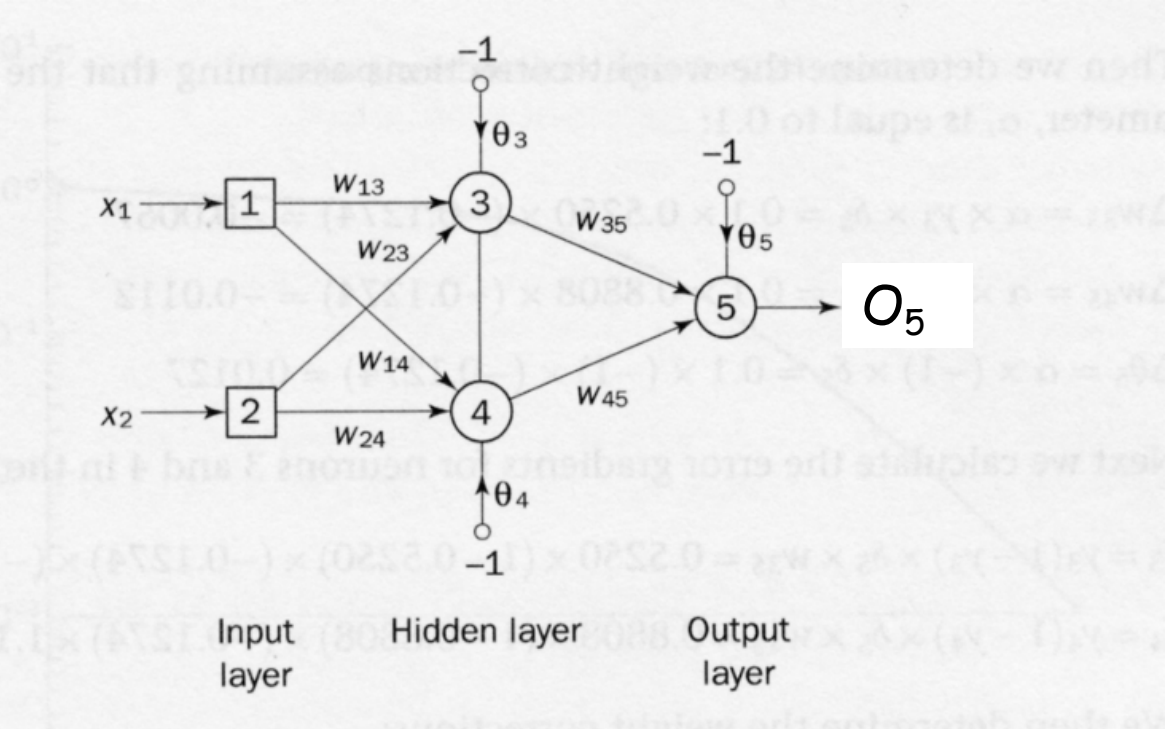
Sum of the weighted error term of the output nodes that h is connected to (ie. h contributed to the errors δ_k)

- Step 4: Update each network weight w_{ij}:

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij} \quad \text{where } \Delta w_{ij} = -\eta\,\delta_j\, O_i$$

- Repeat steps 1 to 4 until the error is minimised to a given level

Example: XOR

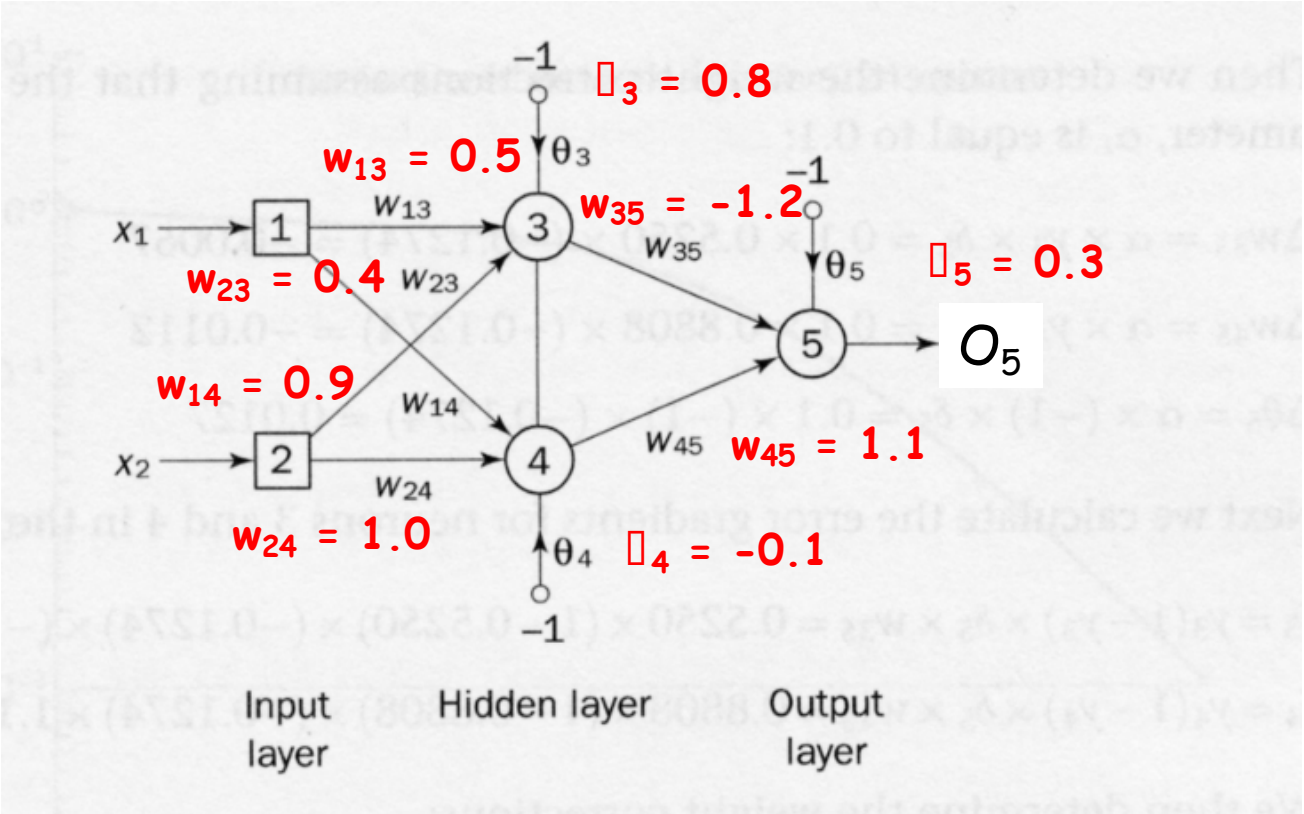


- 2 input nodes + 2 hidden nodes + 1 output node + 3 biases

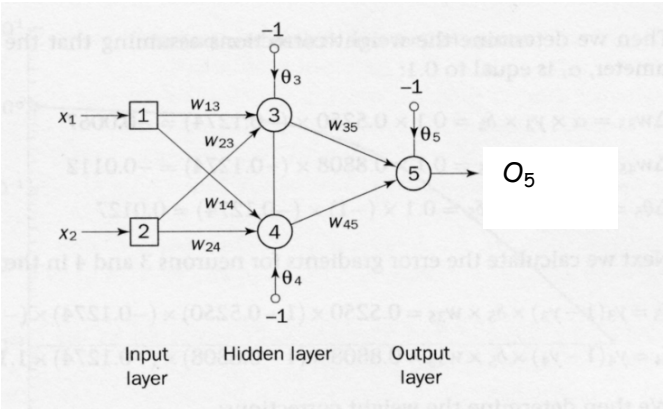
source: Negnevitsky, Artificial Intelligence, p. 181

Example: Step 0 (initialization)

- Step 0: Initialize the network at random



Step 1: Feed Forward



- Step 1: Feed the inputs and calculate the output

$$O_i = \text{sigmoid}\left(\sum_j w_{ji} x_j\right) = \frac{1}{1+e^{-\left(\sum_j w_{ji} x_j\right)}}$$

x_1	x_2	Target output T
1	1	0
0	0	0
1	0	1
0	1	1

- With (x_1=1, x_2=1) as input:
  - Output of the hidden node 3:
    - O_3= sigmoid($x_1 w_{13} + x_2 w_{23} - \theta_3$) = 1 / (1 + $e^{-(1x.5 + 1x.4 - 1x.8)}$)= 0.5250
  - Output of the hidden node 4:
    - O_4= sigmoid($x_1 w_{14} + x_2 w_{24} - \theta_4$) = 1 / (1 + $e^{-(1x.9 + 1x1.0 + 1x0.1)}$)= 0.8808
  - Output of neuron 5:
    - O_5 = sigmoid($O_3 w_{35} + O_4 w_{45} - \theta_5$) = 1 / (1 + $e^{-(0.5250x-1.2 + 0.8808x1.1 - 1x0.3)}$)= 0.5097

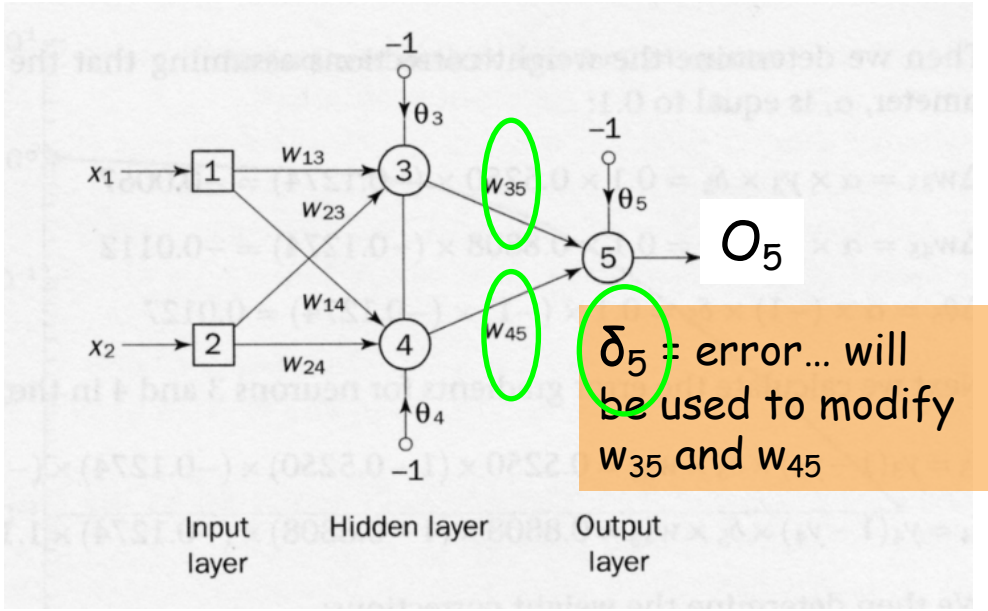
Step 2: Calculate error term of output layer

$$\delta_k \leftarrow g'(x_k) \times Err_k = O_k(1 - O_k) \times (O_k - T_k)$$

- Error term of neuron 5 in the output layer:
  - $\delta_5 = O_5\ (1-O_5)\ (O_5 - T_5)$

    $= (0.5097) \times (1-0.5097) \times (0.5097-0)$

    $= 0.1274$



Step 3: Calculate error term of hidden layer

$$\delta_h \leftarrow g'(x_h) \times Err_h = O_k(1 - O_k) \times \sum_{k \in outputs} w_{kh}\delta_k$$

- Error term of neurons 3 & 4 in the hidden layer:
  - $\delta_3 = O_3 (1-O_3)\ \delta_5 w_{35}$

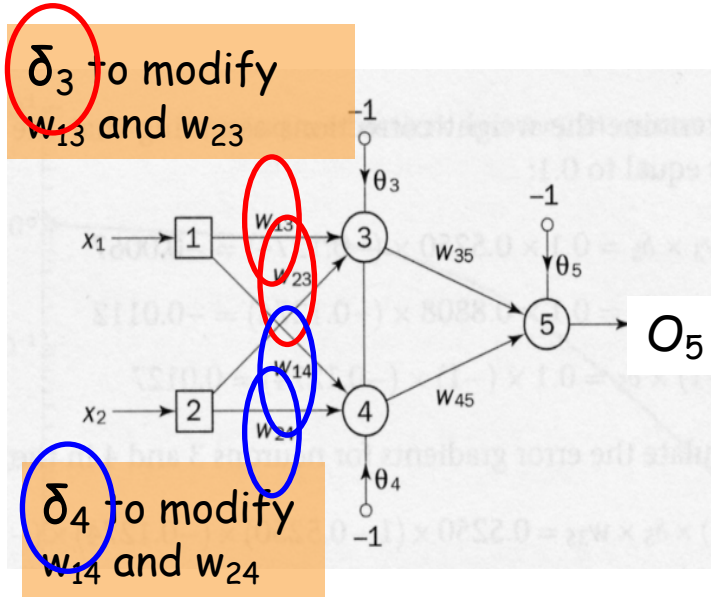
    $= (0.5250) \times (1-0.5250) \times (0.1274) \times (-1.2)$

    $= -0.0381$

  - $\delta_4 = O_4 (1-O_4)\ \delta_5 w_{45}$

    $= (0.8808) \times (1-0.8808) \times (0.1274) \times (1.1)$

    $= 0.0147$



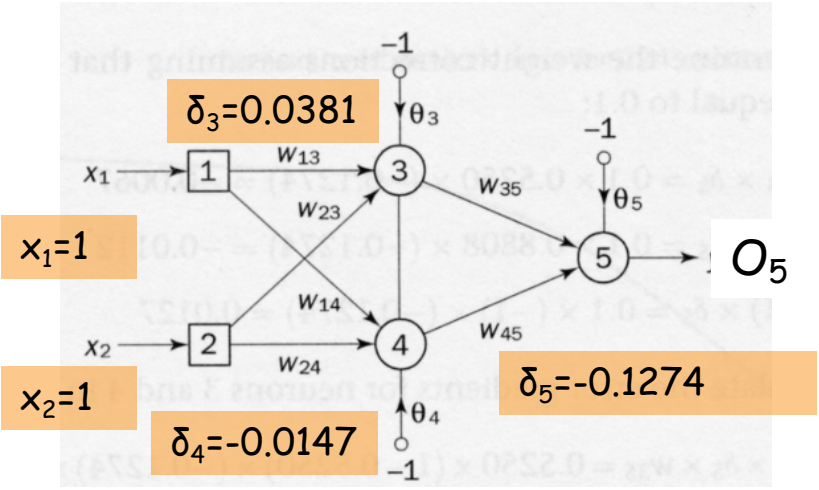
Step 4: Update Weights

- Update all weights (assume a constant learning rate η = 0.1)
  - $\Delta w_{13} = -a\ \delta_3\ x_1 = -0.1 \times -0.0381 \times 1 = +0.0038$
  - $\Delta w_{14} = -a\ \delta_4\ x_1 = -0.1 \times 0.0147 \times 1 = -0.0015$
  - $\Delta w_{23} = -a\ \delta_3\ x_2 = -0.1 \times -0.0381 \times 1 = +0.0038$
  - $\Delta w_{24} = -a\ \delta_4\ x_2 = -0.1 \times 0.0147 \times 1 = -0.0015$
  - $\Delta w_{35} = -a\ \delta_5\ O_3 = -0.1 \times 0.1274 \times 0.5250 = -0.00669$

  // O_3 is seen as x_5 (output of 3 is input to 5)
  - $\Delta w_{45} = -a\ \delta_5\ O_4 = -0.1 \times 0.1274 \times 0.8808 = -0.01122$

  // O_4 is seen as x_5 (output of 4 is input to 5)
  - $\Delta\theta_3 = -a\ \delta_3\ (-1) = -0.1 \times -0.0381 \times -1 = -0.0038$
  - $\Delta\theta_4 = -a\ \delta_4\ (-1) = -0.1 \times 0.0147 \times -1 = +0.0015$
  - $\Delta\theta_5 = -a\ \delta_5\ (-1) = -0.1 \times 0.1274 \times -1 = +0.0127$

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij} \quad \text{where } \Delta w_{ij} = -\eta\ \delta_j\ x_i$$

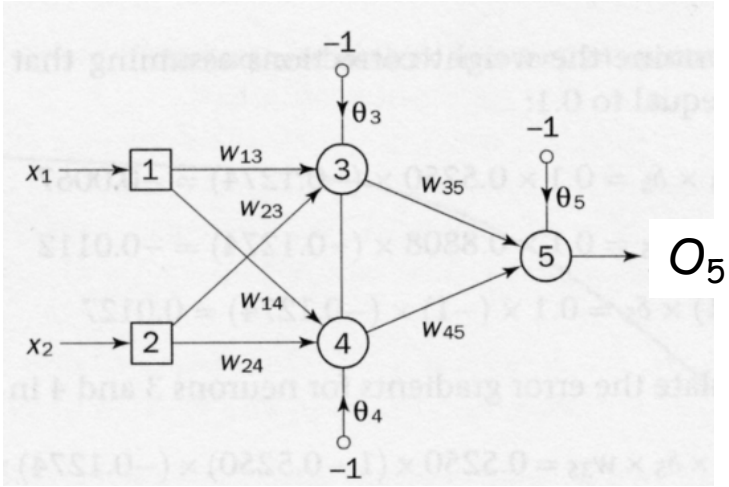


Step 4: Update Weights

- Update all weights (assume a constant learning rate η = 0.1)

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij} \quad \text{where } \Delta w_{ij} = -\eta\, \delta_j\, x_i$$

- $w_{13} = w_{13} + \Delta w_{13} = 0.5 + 0.0038 = 0.5038$
- $w_{14} = w_{14} + \Delta w_{14} = 0.9 - 0.0015 = 0.8985$
- $w_{23} = w_{23} + \Delta w_{23} = 0.4 + 0.0038 = 0.4038$
- $w_{24} = w_{24} + \Delta w_{24} = 1.0 - 0.0015 = 0.9985$
- $w_{35} = w_{35} + \Delta w_{35} = -1.2 - 0.00669 = -1.20669$
- $w_{45} = w_{45} + \Delta w_{45} = 1.1 - 0.01122 = 1.08878$
- $\theta_3 = \theta_3 + \Delta\theta_3 = 0.8 - 0.0038 = 0.7962$
- $\theta_4 = \theta_4 + \Delta\theta_4 = -0.1 + 0.0015 = -0.0985$
- $\theta_5 = \theta_5 + \Delta\theta_5 = 0.3 + 0.0127 = 0.3127$



Step 4: Iterate through data

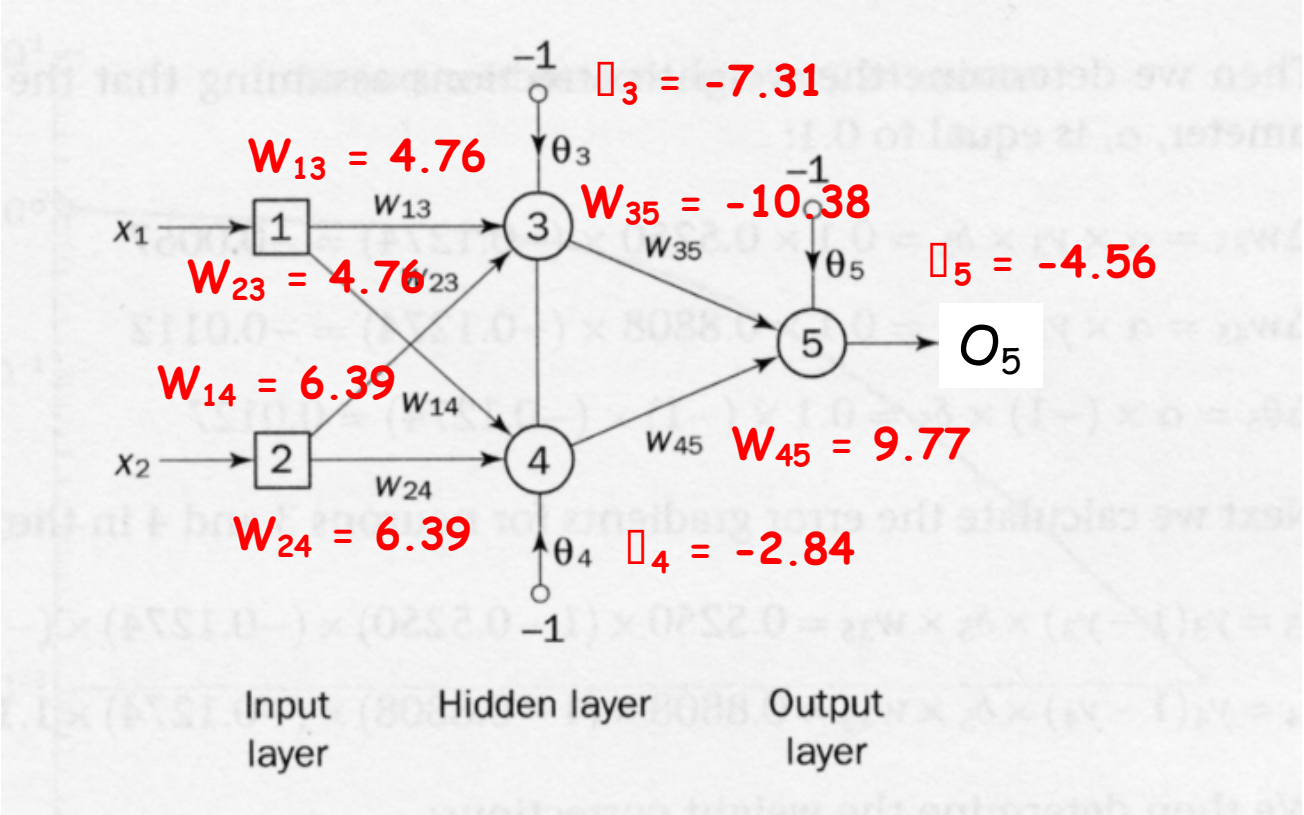
- after adjusting all the weights, repeat the forward pass and back pass for the next data point until all data points are examined
- repeat this entire exercise until the overall error is minimised
  - Ex: $Mean\ Squared\ Error\ (MSE)$=

$$\frac{1}{2}\sum_{i=1}^{n}(T_i - O_i) < \varepsilon$$

where $\varepsilon \sim 0.001$ and n= number of training examples

The Result…

- After 224 epochs, we get:
  - (1 epoch = going through all data once)



Error is minimized

Inputs		Target Output	Actual Output	Error
x_1	x_2	T_5	O_5	e
1	1	0	0.0155	-0.0155
0	1	1	0.9849	0.0151
1	0	1	0.9849	0.0151
0	0	0	0.0175	-0.0175

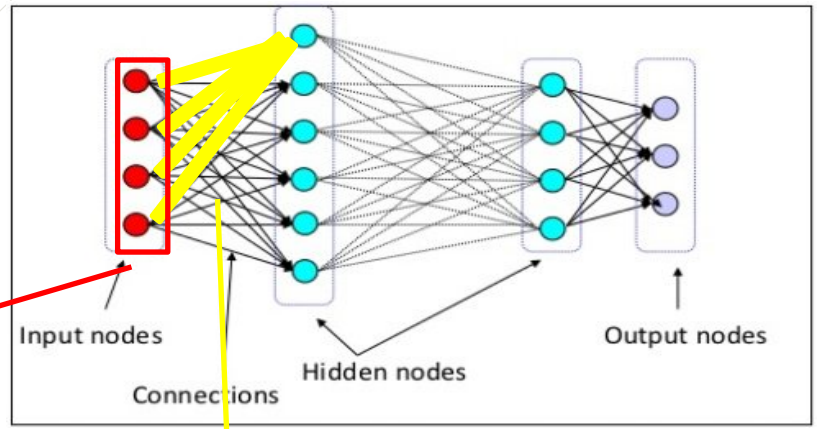
$$Mean\ Squared\ Error = \frac{1}{2}.\frac{(-0.0155^2 + 0.0151^2 + 0.0151^2 + 0.0175^2)}{4}$$
$$= 0.000125 < 0.001\ (some\ threshold)\ \ldots\ stop!$$

May be a local minimum…

Stochastic Gradient Descent

- Batch Gradient Descent (GD)
  - updates the weights after 1 epoch
  - can be costly (time & memory) since we need to evaluate the whole training dataset before we take one step towards the minimum.
- Stochastic Gradient Descent (SGD)
  - updates the weights after each training example
  - often converges faster compared to GD
  - but the error function is not as well minimized as in the case of GD
  - to obtain better results, shuffle the training set for every epoch
- MiniBatch Gradient Descent:
  - compromise between GD and SGD
  - cut your dataset into sections, and update the weights after training on each section

Matrix Notation



$$O_i = \text{sigmoid}\left(\sum_{j=1}^{n} w_{ji} x_j\right)$$

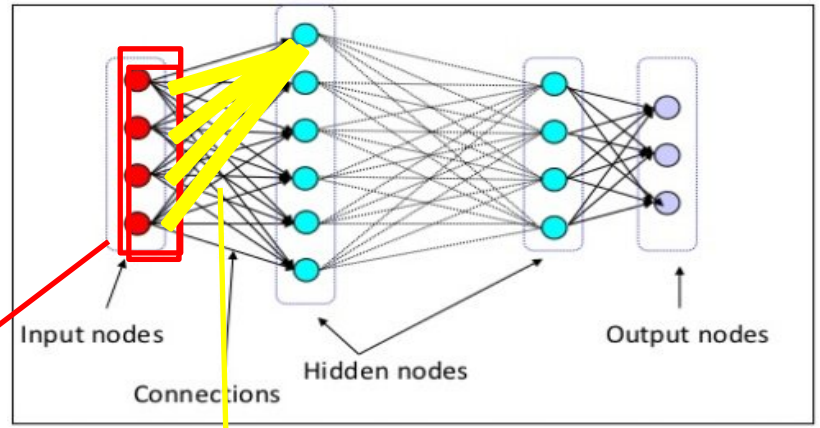
Assume:
n=4 input nodes
6 hidden nodes at layer 2

$$\begin{bmatrix} x1 \\ x2 \\ x3 \\ x4 \end{bmatrix} \cdot \begin{bmatrix} W_{x1,01} & W_{x2,01} & W_{x3,01} & W_{x4,01} \\ W_{x1,02} & W_{x2,02} & W_{x3,02} & W_{x4,02} \\ W_{x1,03} & W_{x2,03} & W_{x3,03} & W_{x4,03} \\ W_{x1,04} & W_{x2,04} & W_{x3,04} & W_{x4,04} \\ W_{x1,05} & W_{x2,05} & W_{x3,05} & W_{x4,05} \\ W_{x1,06} & W_{x2,06} & W_{x3,06} & W_{x4,06} \end{bmatrix}$$

Value of all incoming inputs to hidden nodes

All incoming weights to hidden node O1

Take dot product



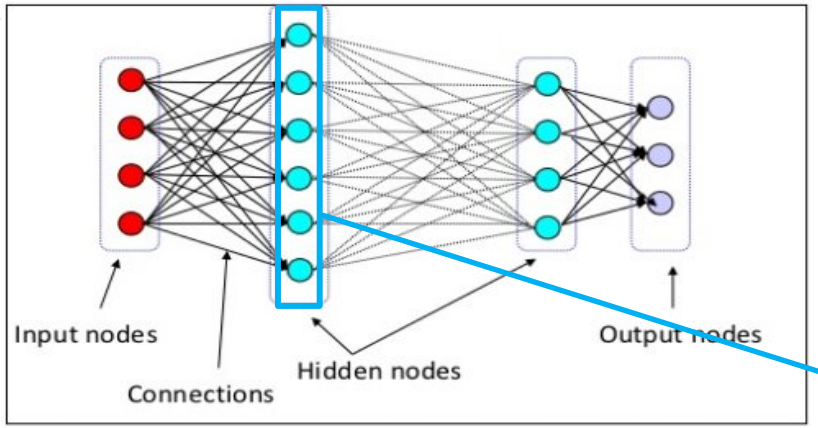
$$O_i = \text{sigmoid}\left(\sum_{j=1}^{n} w_{ji} x_j\right)$$

Assume:
n=4 input nodes
6 hidden nodes at layer 2

$$\begin{bmatrix} x1 \\ x2 \\ x3 \\ x4 \end{bmatrix} \cdot \begin{bmatrix} W_{x1,O1} & W_{x2,O1} & W_{x3,O1} & W_{x4,O1} \\ W_{x1,O2} & W_{x2,O2} & W_{x3,O2} & W_{x4,O2} \\ W_{x1,O3} & W_{x2,O3} & W_{x3,O3} & W_{x4,O3} \\ W_{x1,O4} & W_{x2,O4} & W_{x3,O4} & W_{x4,O4} \\ W_{x1,O5} & W_{x2,O5} & W_{x3,O5} & W_{x4,O5} \\ W_{x1,O6} & W_{x2,O6} & W_{x3,O6} & W_{x4,O6} \end{bmatrix} = \begin{bmatrix} (w_{x1,O1})x_1 + (w_{x2,O1})x_2 + (w_{x3,O1})x_3 + (w_{x4,O1})x_4 \\ (w_{x1,O2})x_1 + (w_{x2,O2})x_2 + (w_{x3,O2})x_3 + (w_{x4,O2})x_4 \\ (w_{x1,O3})x_1 + (w_{x2,O3})x_2 + (w_{x3,O3})x_3 + (w_{x4,O3})x_4 \\ (w_{x1,O4})x_1 + (w_{x2,O4})x_2 + (w_{x3,O4})x_3 + (w_{x4,O4})x_4 \\ (w_{x1,O5})x_1 + (w_{x2,O5})x_2 + (w_{x3,O5})x_3 + (w_{x4,O5})x_4 \\ (w_{x1,O6})x_1 + (w_{x2,O6})x_2 + (w_{x3,O6})x_3 + (w_{x4,O6})x_4 \end{bmatrix}$$

All incoming weights to hidden node O1

Apply activation function

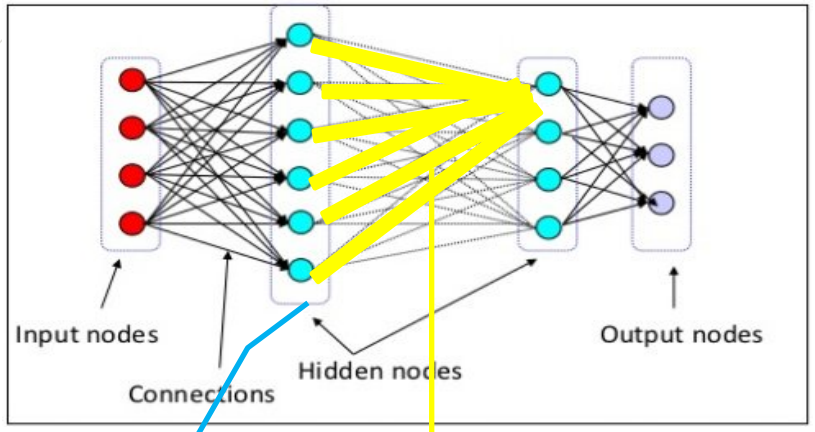


$$O_i = \text{sigmoid}\left(\sum_{j=1}^{n} w_{ji} x_j\right)$$

We have:
n=4 input nodes
6 hidden nodes at layer 2

$$\text{sigmoid}\begin{pmatrix}
(w_{x1,O1})x_1 + (w_{x2,O1})x_2 + (w_{x3,O1})x_3 + (w_{x4,O1})x_4 \\
(w_{x1,O2})x_1 + (w_{x2,O2})x_2 + (w_{x3,O2})x_3 + (w_{x4,O2})x_4 \\
(w_{x1,O3})x_1 + (w_{x2,O3})x_2 + (w_{x3,O3})x_3 + (w_{x4,O3})x_4 \\
(w_{x1,O4})x_1 + (w_{x2,O4})x_2 + (w_{x3,O4})x_3 + (w_{x4,O4})x_4 \\
(w_{x1,O5})x_1 + (w_{x2,O5})x_2 + (w_{x3,O5})x_3 + (w_{x4,O5})x_4 \\
(w_{x1,O6})x_1 + (w_{x2,O6})x_2 + (w_{x3,O6})x_3 + (w_{x4,O6})x_4
\end{pmatrix} = \begin{bmatrix} O_1 \\ O_2 \\ O_3 \\ O_4 \\ O_5 \\ O_6 \end{bmatrix}$$

Repeat through next level



$$O_i = \text{sigmoid}\left(\sum_{j=1}^{n} w_{ji} x_j\right)$$

We have :
n=6 input nodes (from layer 2)
4 hidden nodes at layer 3

$$\begin{bmatrix} O1 \\ O2 \\ O3 \\ O4 \\ O5 \\ O6 \end{bmatrix}$$

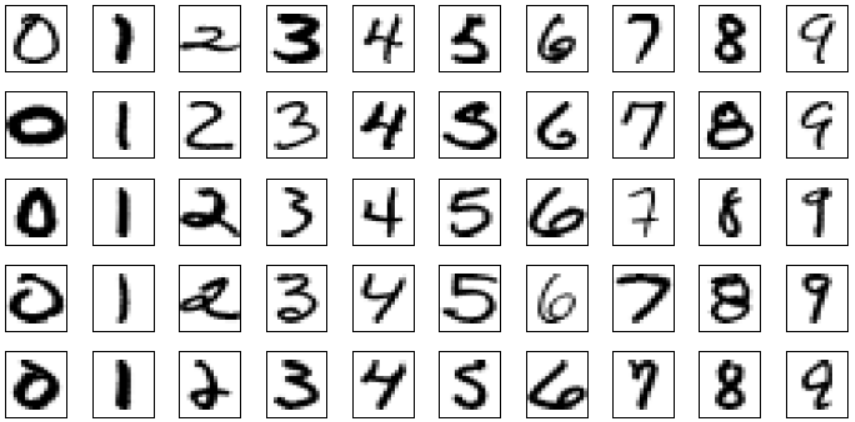
$$\begin{bmatrix} w_{O1,P1} & w_{O2,P1} & w_{O3,P1} & w_{O4,P1} & w_{O5,P1} & w_{O6,P1} \\ w_{O1,P2} & w_{O2,P2} & w_{O3,P2} & w_{O4,P2} & w_{O5,P2} & w_{O6,P2} \\ w_{O1,P3} & w_{O2,P3} & w_{O3,P3} & w_{O4,P3} & w_{O5,P3} & w_{O6,P3} \\ w_{O1,P4} & w_{O2,P4} & w_{O3,P4} & w_{O4,P4} & w_{O5,P4} & w_{O6,P4} \end{bmatrix} \cdot \begin{bmatrix} O_1 \\ O_2 \\ O_3 \\ O_4 \\ O_5 \\ O_6 \end{bmatrix} = \text{...and so on}$$

All incoming weights to hidden node P1

Value of all incoming inputs to the 4 hidden nodes of layer 3 (what we computed in the previous slide)

Applications of Neural Networks

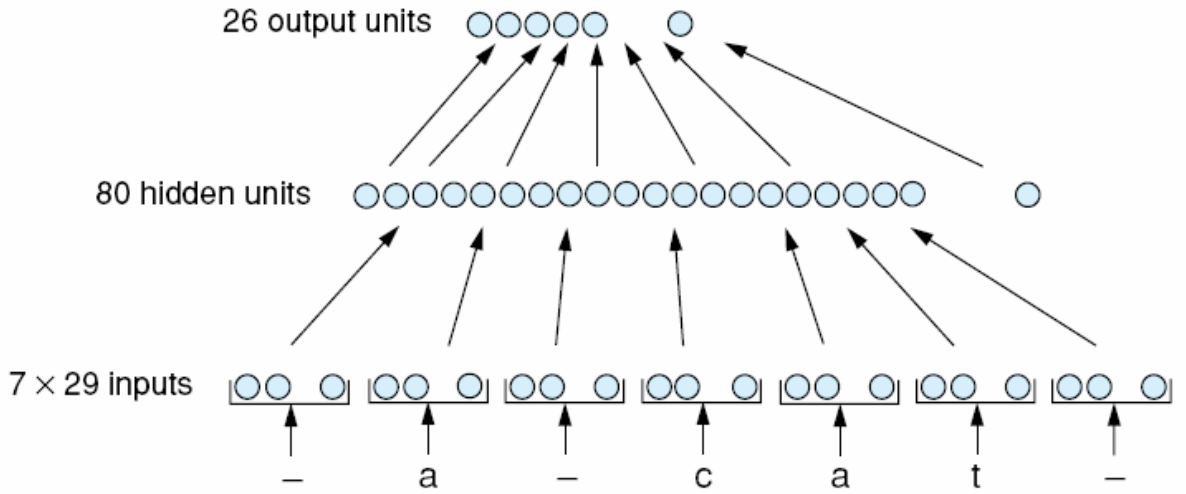
- Handwritten digit recognition
  - Training set = set of handwritten digits (0…9)
  - Task: given a bitmap, determine what digit it represents
  - Input: 1 feature for each pixel of the bitmap
  - Output: 1 output unit for each possible character (only 1 should be activated)
  - After training, network should work for fonts (handwriting) never encountered
- Related pattern recognition applications:
  - recognize postal codes
  - recognize signatures
  - …



Applications of Neural Networks

- Speech synthesis
  - Learning to pronounce English words
  - Difficult task for a rule-based system because English pronunciation is highly irregular
  - Examples:
    - letter "c" can be pronounced [k] (*cat*) or [s] (*cents*)
    - *Woman* vs *Women*
  - NETtalk:
    - uses the context and the letters around a letter to learn how to pronounce a letter
    - Input: letter and its surrounding letters
    - Output: phoneme

NETtalk Architecture



Ex: *a cat → c is pronounced K*

- Network is made of 3 layers of units
- input unit corresponds to a 7 character window in the text
- each position in the window is represented by 29 input units (26 letters + 3 for punctuation and spaces)
- 26 output units – one for each possible phoneme

Listen to the output through iterations:
https://www.youtube.com/watch?v=gakJlr3GecE

source: Luger (2005)

Neural Networks

- Disadvantage:
  - result is not easy to understand by humans (set of weights compared to decision tree)… it is a black box
- Advantage:
  - robust to noise in the input (small changes in input do not normally cause a change in output) and graceful degradation

The End