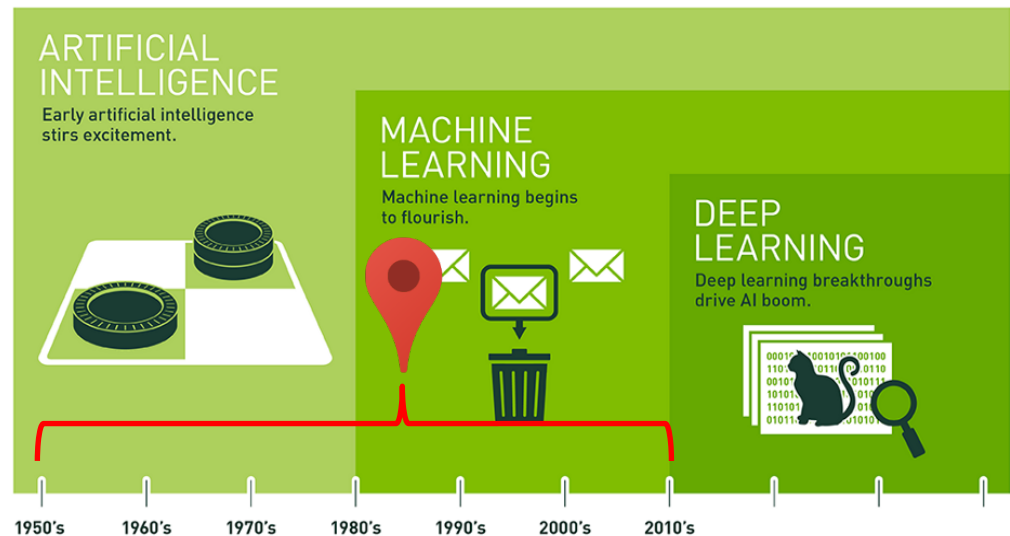

Artificial Intelligence: Introduction to Neural Networks

Perceptron, Backpropagation

Today

■ Neural Networks

- Perceptrons
- Backpropagation



Neural Networks

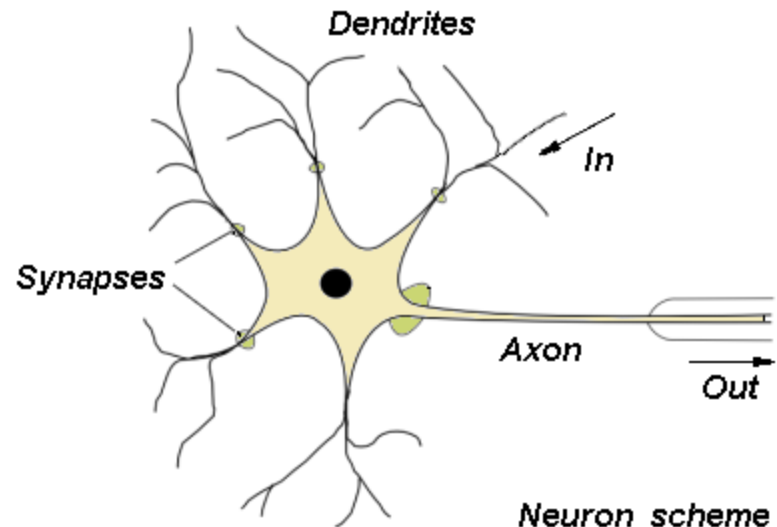
- Radically different approach to reasoning and learning
- Inspired by biology
 - the neurons in the human brain
- Set of many simple processing units (neurons) connected together
- Behavior of each neuron is very simple
 - but a collection of neurons can have sophisticated behavior and can be used for 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

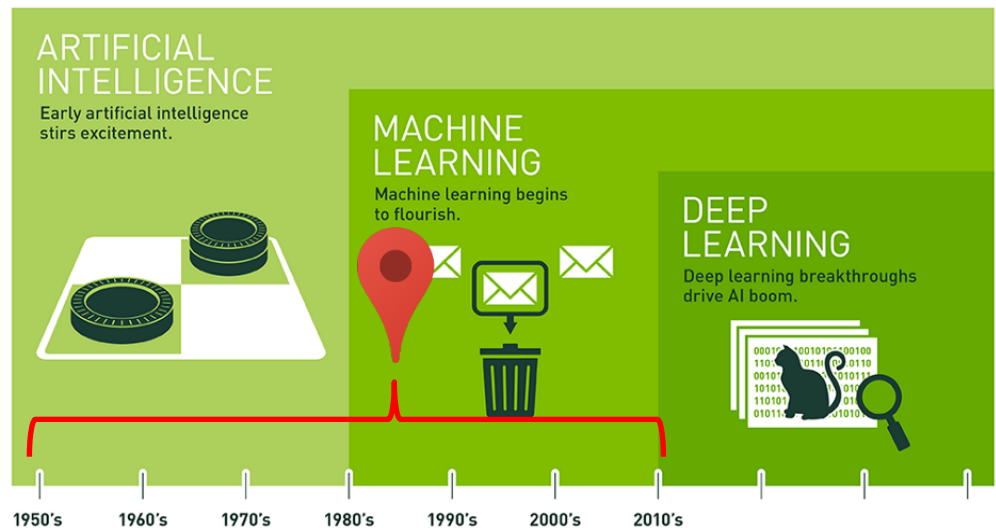


Behavior of a Neuron

- 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
- If a particular stimulus repeatedly causes activity in a group of neurons, they become strongly associated

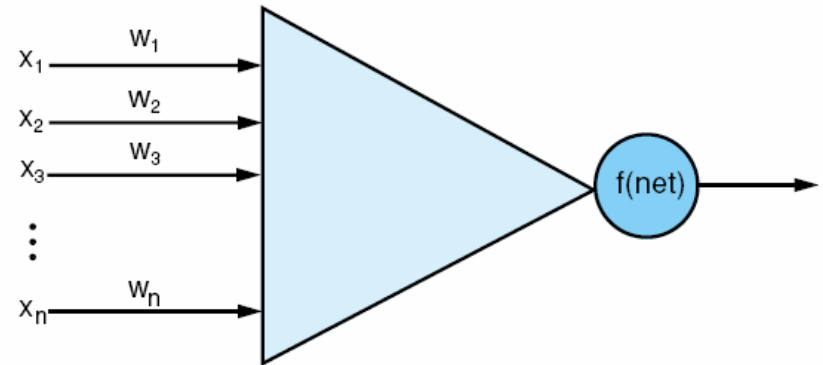
Today

- Neural Netwo
 - Perceptrons
 - Backpropagation



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 \geq some threshold, neuron fires (output=1)
 - otherwise output = 0
 - If $(w_1 x_1 + \dots + w_n x_n) \geq t$
 - Then output = 1
 - Else output = 0
- Learning :
 - use the training data to adjust the weights in the perceptron



A Simple Example

Student	Features (X)				Output f(X)
	'A' last year?	Black hair?	Works hard?	Drinks?	'A' this year?
X1: Richard	Yes	Yes	No	Yes	No
X2: Alan	Yes	Yes	Yes	No	Yes
X3: Alison	No	No	Yes	No	No
X4: Jeff	No	Yes	No	Yes	No
X5: Gail	Yes	No	Yes	Yes	Yes
X6: Simon	No	Yes	Yes	Yes	No

First last year?

1

Male?

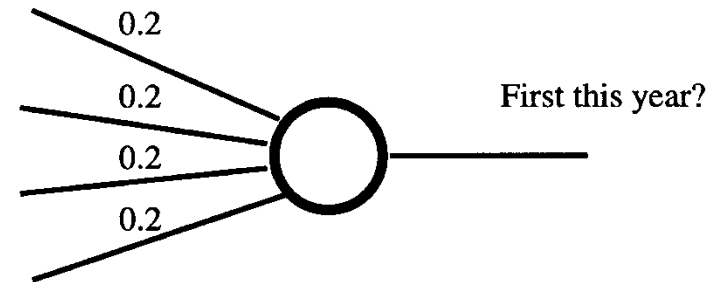
1

Works hard?

0

Drinks a lot?

1



- 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

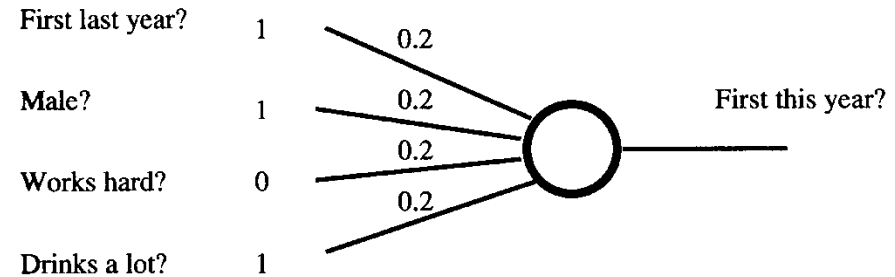
A Simple Example (2)

	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:
 - → **Worksheet #5 ("Perceptron")**

Perceptron Learning: The Idea

	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
...					



1. Step 1: Set weights to random values
2. Step 2: Feed perceptron with a set of inputs
3. Step 3: Compute the network outputs
4. Step 4: Adjust the weights
 1. if output correct \rightarrow weights stay the same
 2. if output = 0 but it should be 1 \rightarrow
 1. increase weights on *active* connections (i.e. input $x_i=1$)
 3. if output = 1 but should be 0 \rightarrow
 1. decrease weights on *active* connections (i.e. input $x_i=1$)
5. 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

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

- Weight adjustment for weights with *activated* input:
 - $\Delta w = \eta(T-O)$ η : Learning Rate T : Expected Output O : Neuron Output
 - $w' = w + \Delta w$

→ **Worksheet #5 ("Perceptron Learning")**

- After 2 iterations over the training set (2 epochs), we get:
 - $w_1 = 0.25$ $w_2 = 0.1$ $w_3 = 0.2$ $w_4 = 0.1$

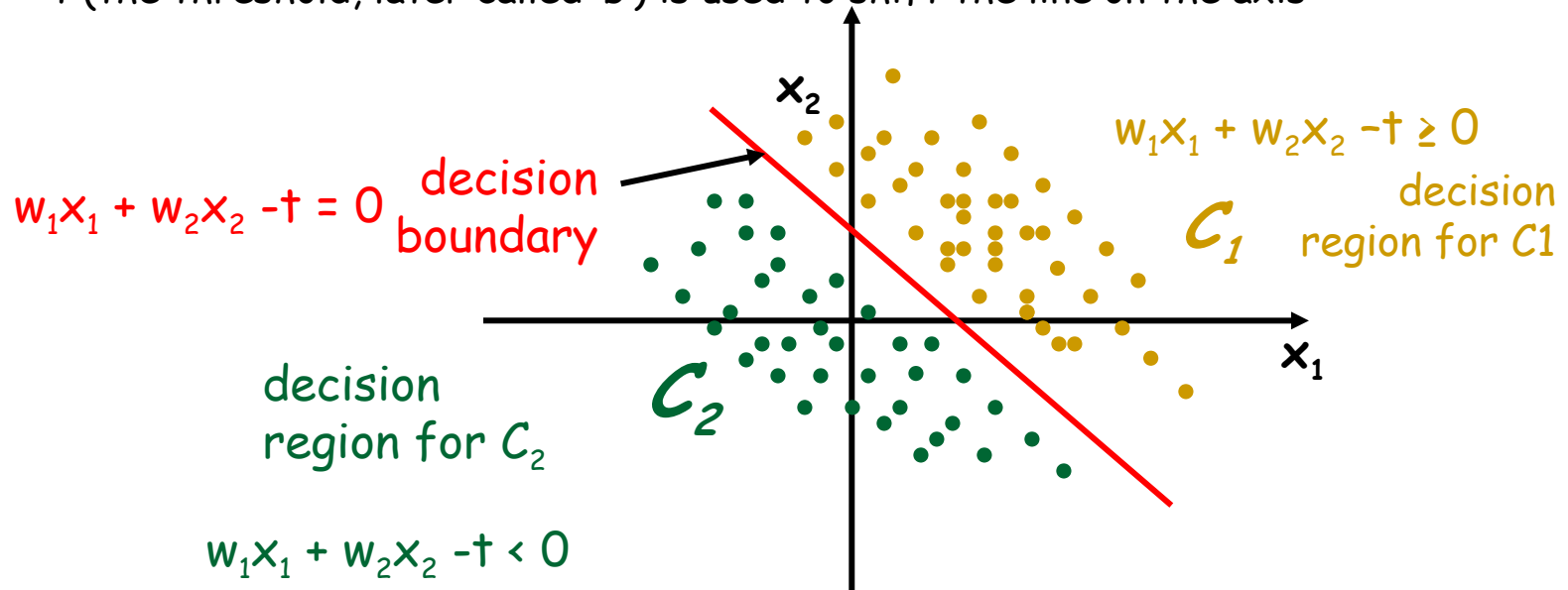
A Simple Example (3)

	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: $(1 \times 0.2) + (1 \times 0.1) + (0 \times 0.25) + (1 \times 0.1) = 0.4 < 0.55 \rightarrow$ output is 0 ✓
 - Alan: $(1 \times 0.2) + (1 \times 0.1) + (1 \times 0.25) + (0 \times 0.1) = 0.55 \geq 0.55 \rightarrow$ output is 1 ✓
 - Alison: $(0 \times 0.2) + (0 \times 0.1) + (1 \times 0.25) + (0 \times 0.1) = 0.25 < 0.55 \rightarrow$ output is 0 ✓
 - Jeff: $(0 \times 0.2) + (1 \times 0.1) + (0 \times 0.25) + (1 \times 0.1) = 0.2 < 0.55 \rightarrow$ output is 0 ✓
 - Gail: $(1 \times 0.2) + (0 \times 0.1) + (1 \times 0.25) + (1 \times 0.1) = 0.55 \geq 0.55 \rightarrow$ output is 1 ✓
 - Simon: $(0 \times 0.2) + (1 \times 0.1) + (1 \times 0.25) + (1 \times 0.1) = 0.45 < 0.55 \rightarrow$ output is 0 ✓

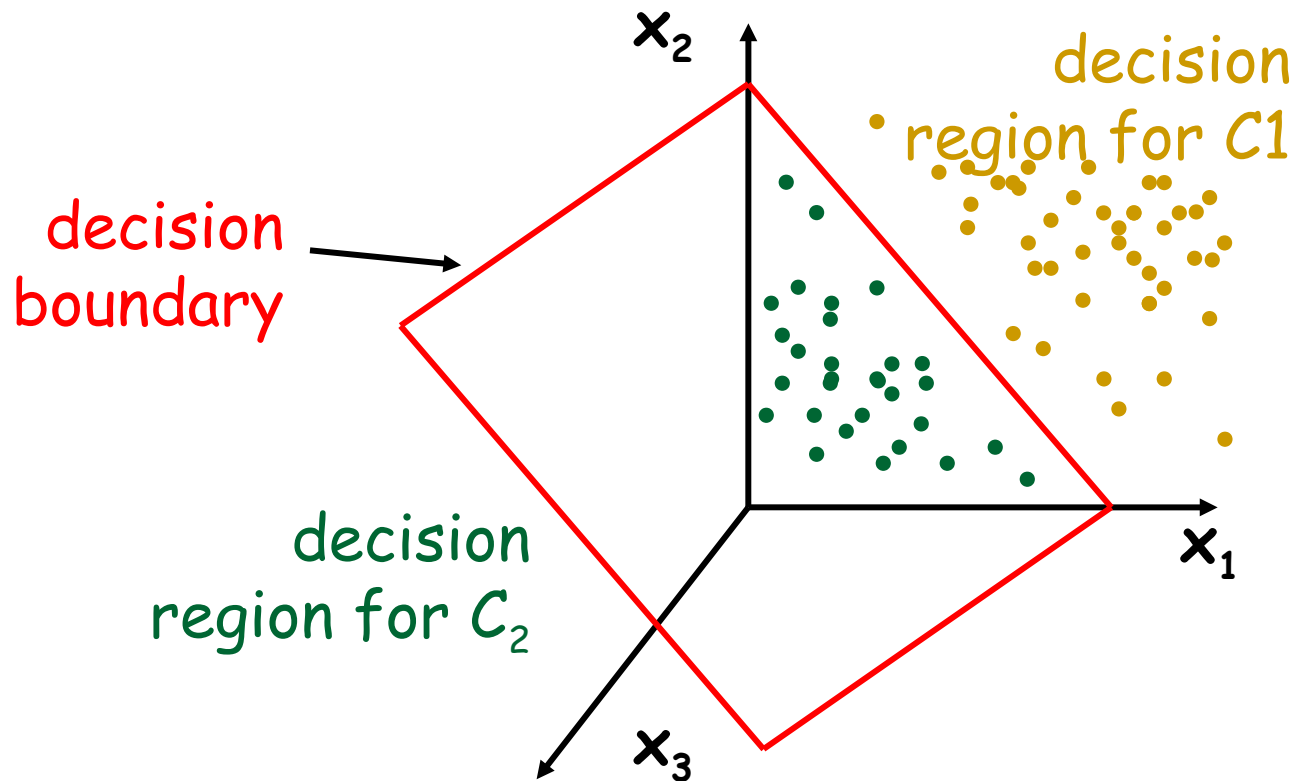
Decision Boundaries of Perceptrons

- 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 $(w_1x_1 + w_2x_2 - t \geq 0)$ then 1 otherwise 0
 - The learned function describes a line in the input space
 - This line is used to separate the two classes C_1 and C_2
 - t (the threshold, later called 'b') is used to shift the line on the axis



Decision Boundaries of Perceptrons

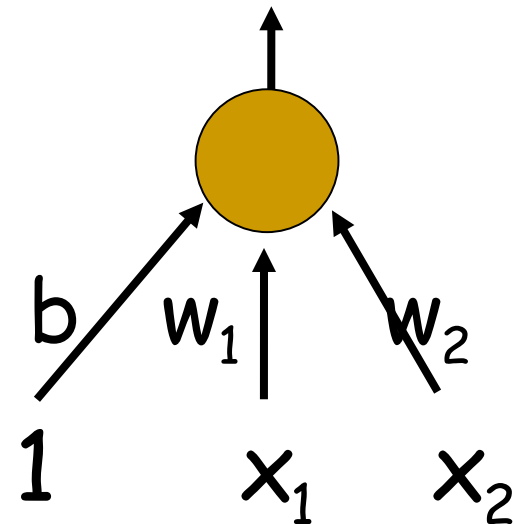
- More generally, with n features, the learned function describes a hyperplane in the input space.



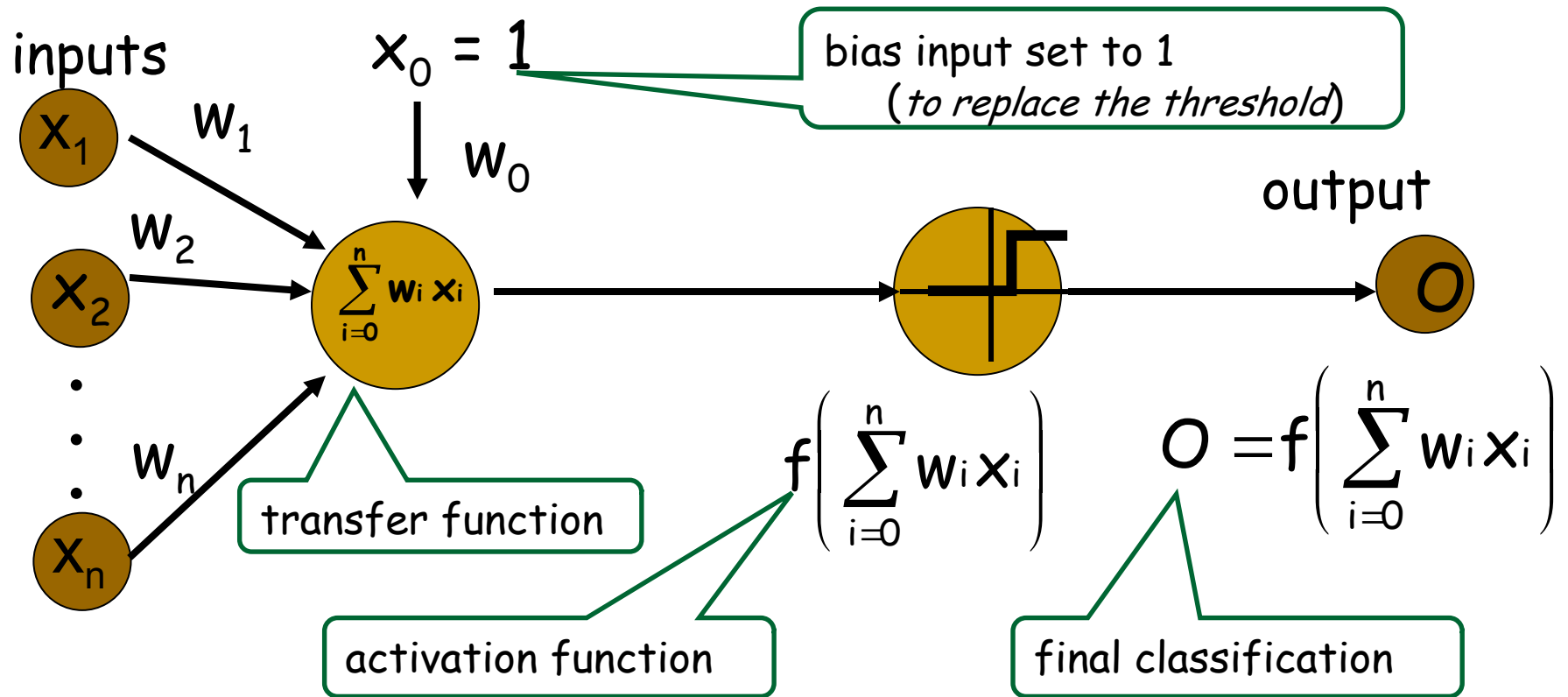
Adding a Bias

- 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.

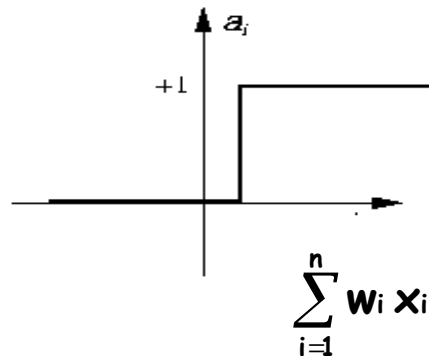
$$b + \sum_i x_i w_i$$



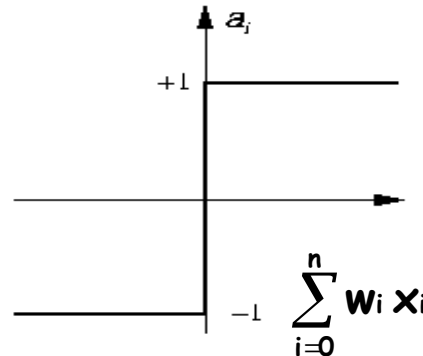
Perceptron - More Generally



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 $O = \begin{cases} +1 & \text{if } \left(\sum_{i=0}^n w_i x_i \right) \geq 0 \\ -1 & \text{otherwise} \end{cases}$

Learning Rate

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

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

learning rate

Error = target output - actual output

□ So:

- if $T=0$ and $O=1$ (i.e. a false positive) → decrease w by η
- if $T=1$ and $O=0$ (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 $T=0$ and $O=1$ (i.e. a false positive) → decrease w_i by ηx_i
- if $T=1$ and $O=0$ (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

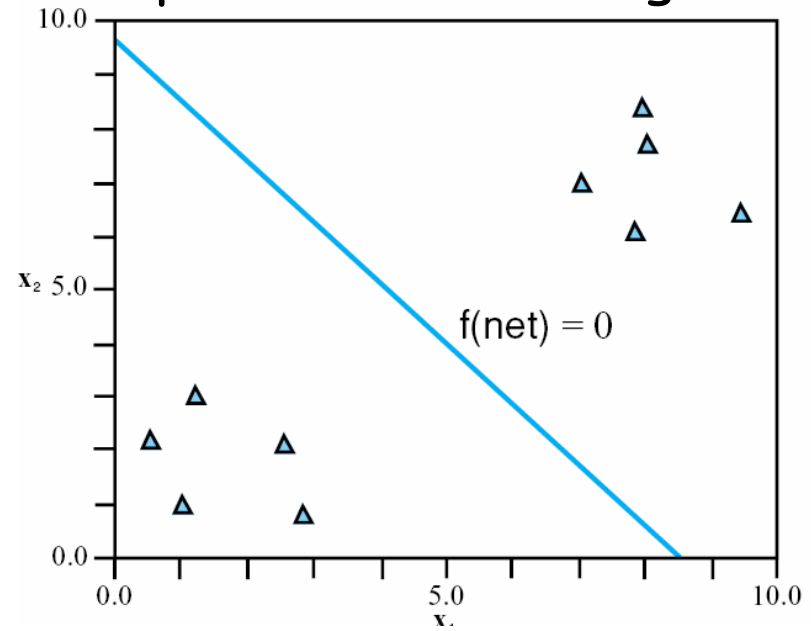
- Cycle through the set of training examples.
- Suppose a solution with zero error exists.
- The delta rule will find a solution in finite time.

Example of the Delta Rule

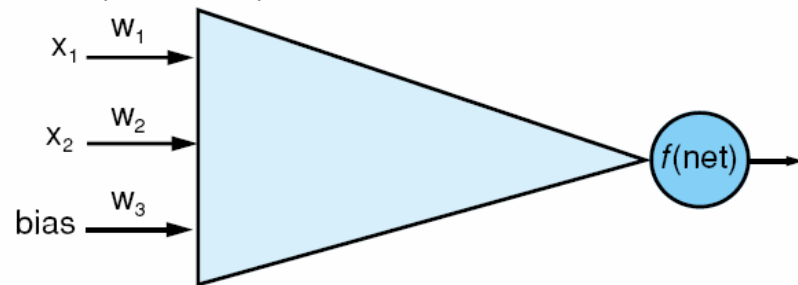
■ training data:

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

■ plot of the training data:



■ perceptron



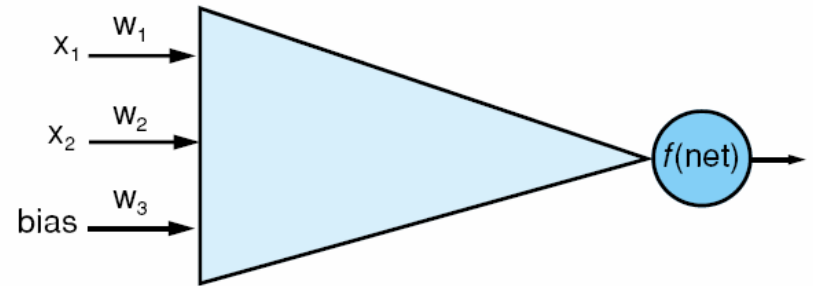
Let's Train the Perceptron

- assume random initialization

- $w_1 = 0.75$

- $w_2 = 0.5$

- $w_3 = -0.6$



- Assume:

- sign function (threshold = 0)

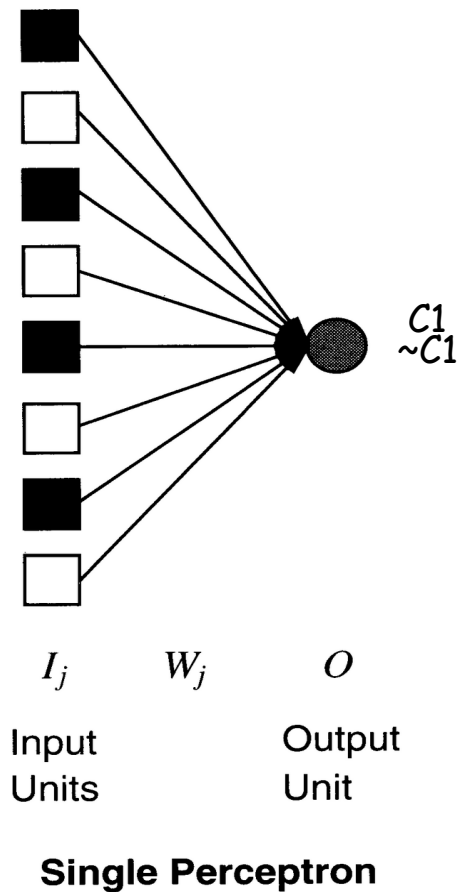
- learning rate $\eta = 0.2$

Training

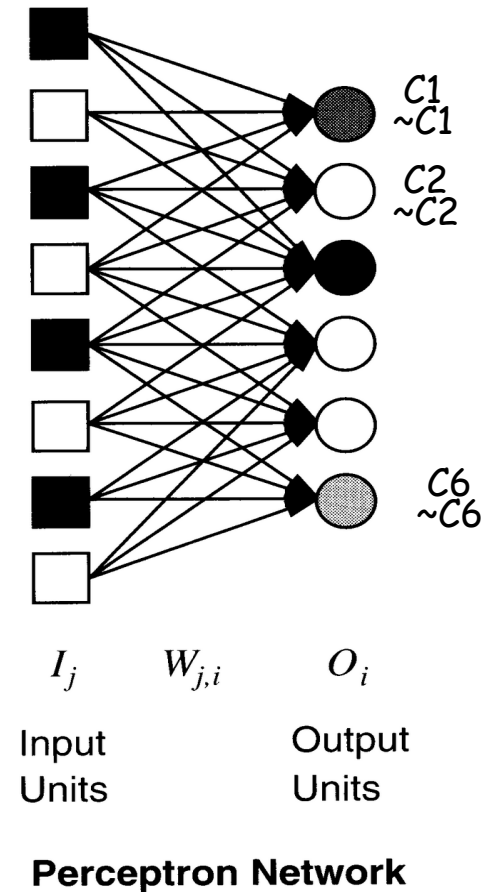
- data #1:
- data #2:
- data #3:
- ...
- → **Worksheet #5 ("Delta Rule")**
- **repeat... over 500 iterations, we converge to:**
 $w_1 = -1.3$ $w_2 = -1.1$ $w_3 = 10.9$

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	0.2	1

A Perceptron Network



- 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



Remember this slide?

History of AI



- Reality hits (late 60s - early 70s)
 - 1966: the ALPAC report kills work in machine translation (and NLP in general)
 - People realized that scaling up from micro-worlds (toy-worlds) to reality is not just a matter of faster machines and larger memories...
 - Minsky & Papert's paper on the limits of perceptrons (cannot learn just any function...) kills work in neural networks
 - in 1971, the British government stops funding research in AI due to no significant results
 - it's the first major *AI Winter*...



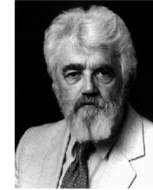
<https://www.vectorstock.com/royalty-free-vector/freezing-snowman-vector-689086>

32

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

Dartmouth Conference: The Founding Fathers of AI



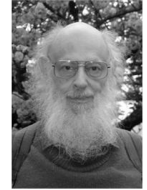
John McCarthy



Marvin Minsky



Claude Shannon



Ray Solomonoff

Alan Newell



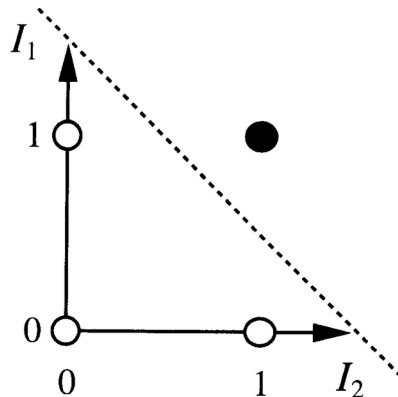
Herbert Simon



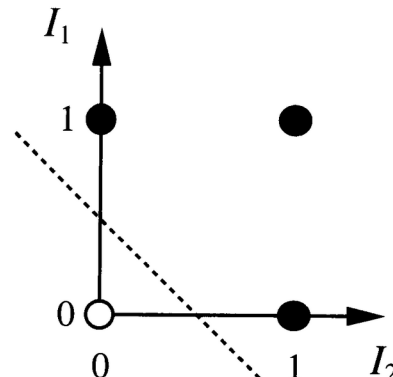
Arthur Samuel



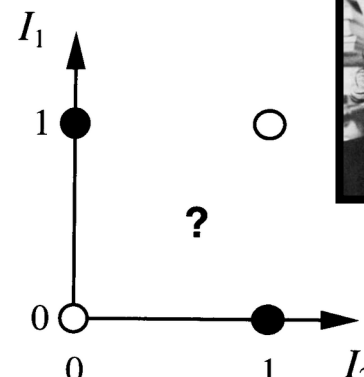
And three others...
Oliver Selfridge
(Pandemonium theory)
Nathaniel Rochester
(IBM, designed 701)
Trenchard More
(Natural Deduction)



(a) I_1 and I_2



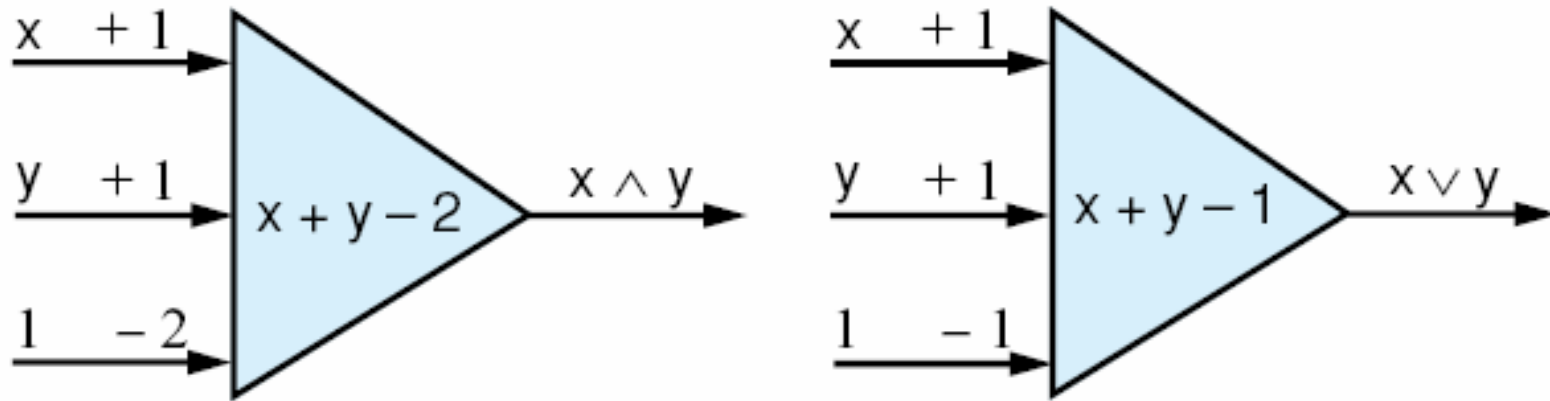
(b) I_1 or I_2



(c) I_1 xor I_2



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

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 \dagger

x_1	x_2	Output
1	1	0
1	0	1
0	1	1
0	0	0

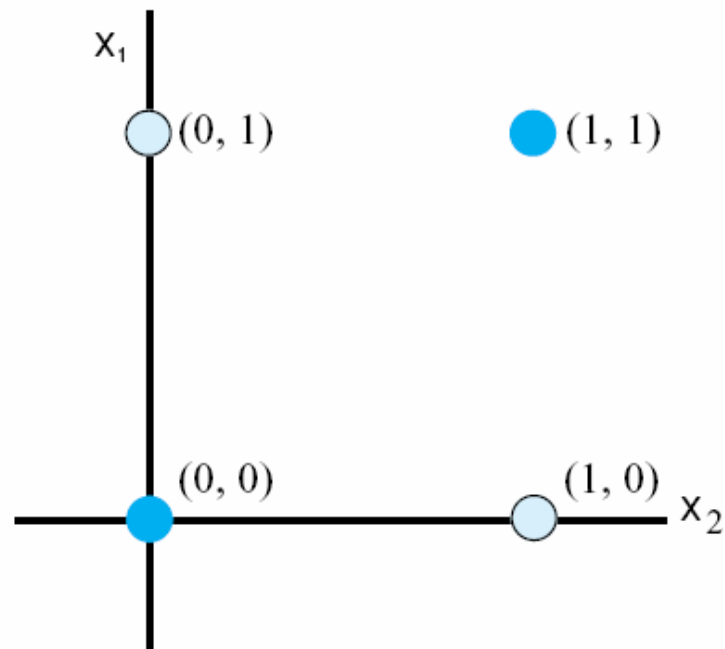
- i.e. must have:

- $(1 \times w_1) + (1 \times w_2) < \dagger$ (for the first line of truth table)
- $(1 \times w_1) + 0 \geq \dagger$
- $0 + (1 \times w_2) \geq \dagger$
- $0 + 0 < \dagger$

- Which has no solution... so a perceptron cannot learn the XOR function

The XOR Function - Visually

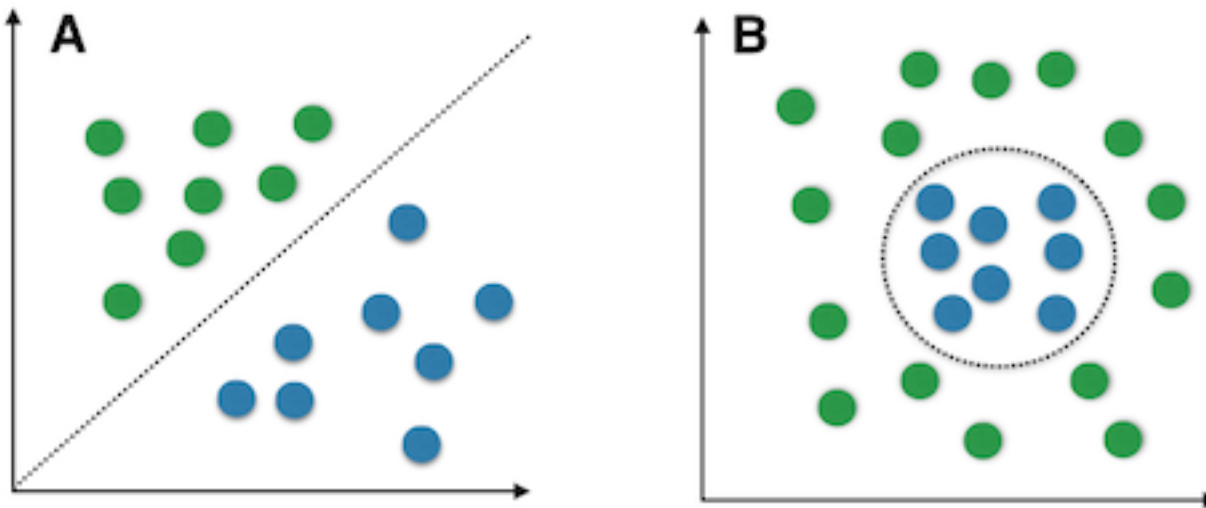
- In a 2-dimensional space (2 features for the X)
- No straight line in two-dimensions can separate
 - (0, 1) and (1, 0) from
 - (0, 0) and (1, 1).



Non-Linearly Separable Functions

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

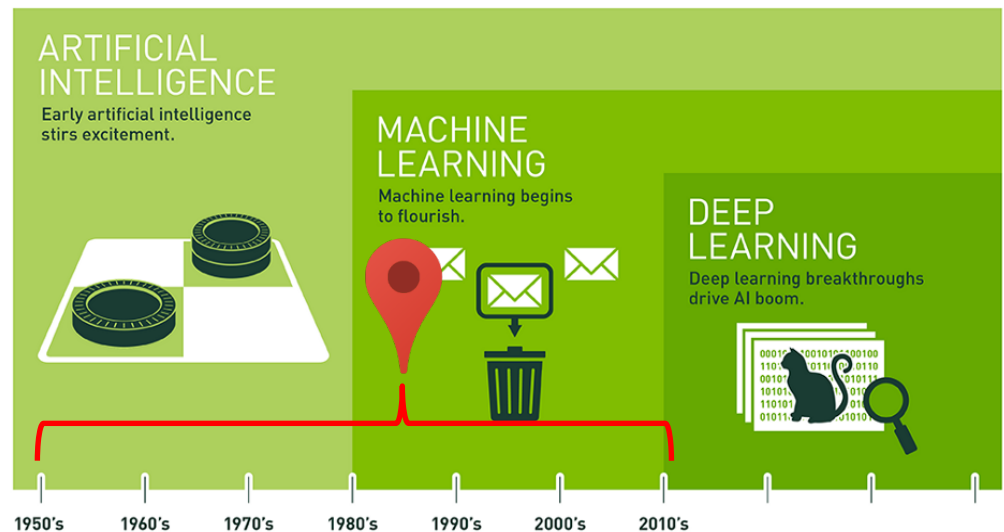
Linear vs. nonlinear problems



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

Today

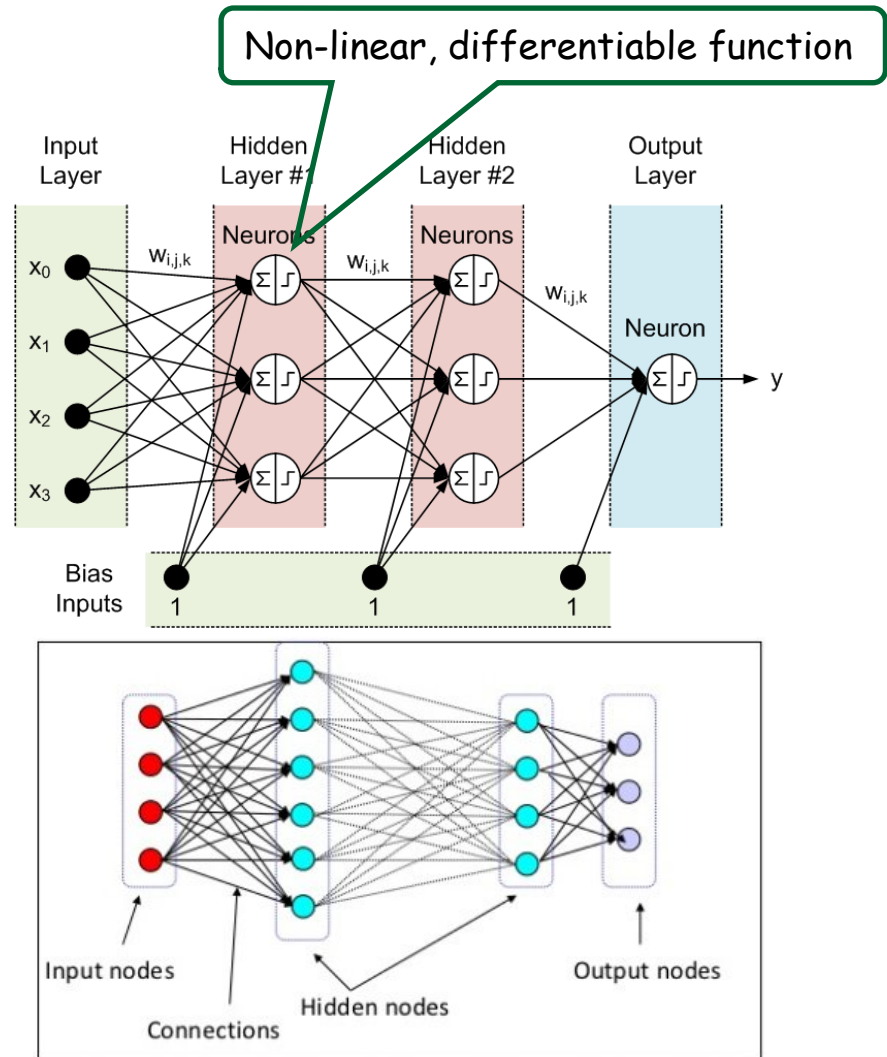
- Neural Networks
 - Perceptrons
 - Backpropagation



Multilayer Neural Networks

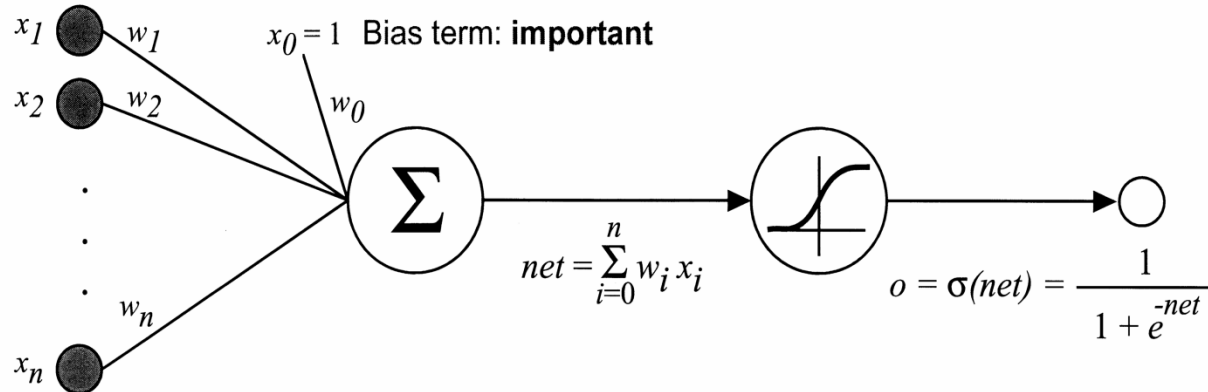
■ Solution:

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

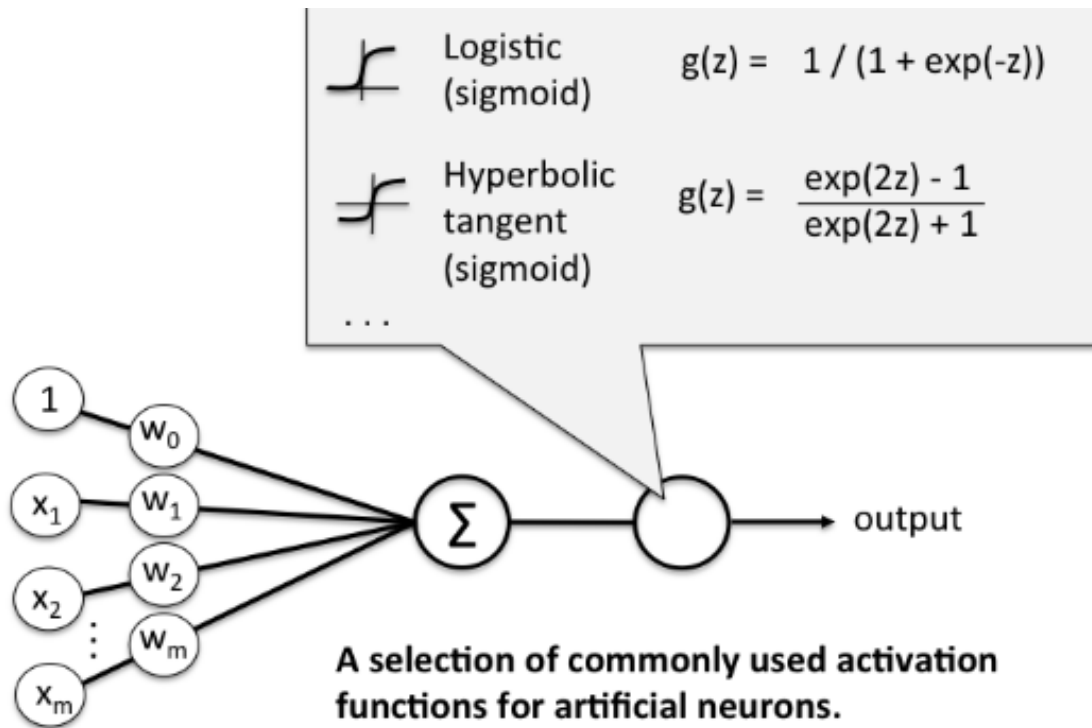


The Sigmoid Function

- Backpropagation requires a differentiable activation function
- sigmoidal (or squashed or logistic) function
- f returns a value *between* 0 and 1 (instead of 0 or 1)
- f indicates how close/how far the output of the network is compared to the right answer (the *error term*)



Typical Activation Functions



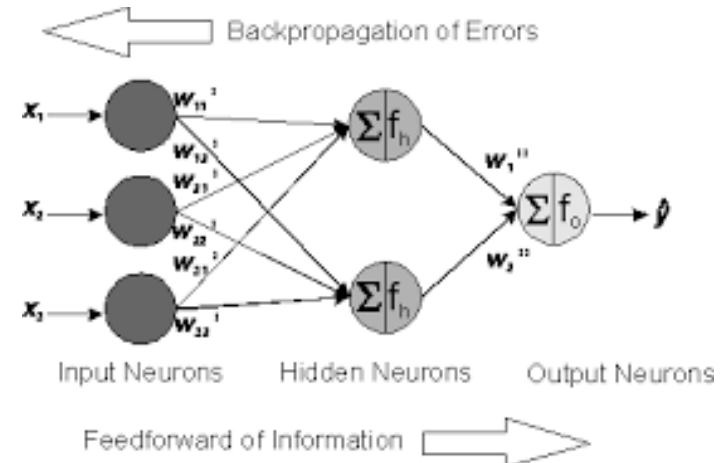
Learning in a Neural Network

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

where $\varepsilon \sim 0.0001$ and
 n = nb of training examples

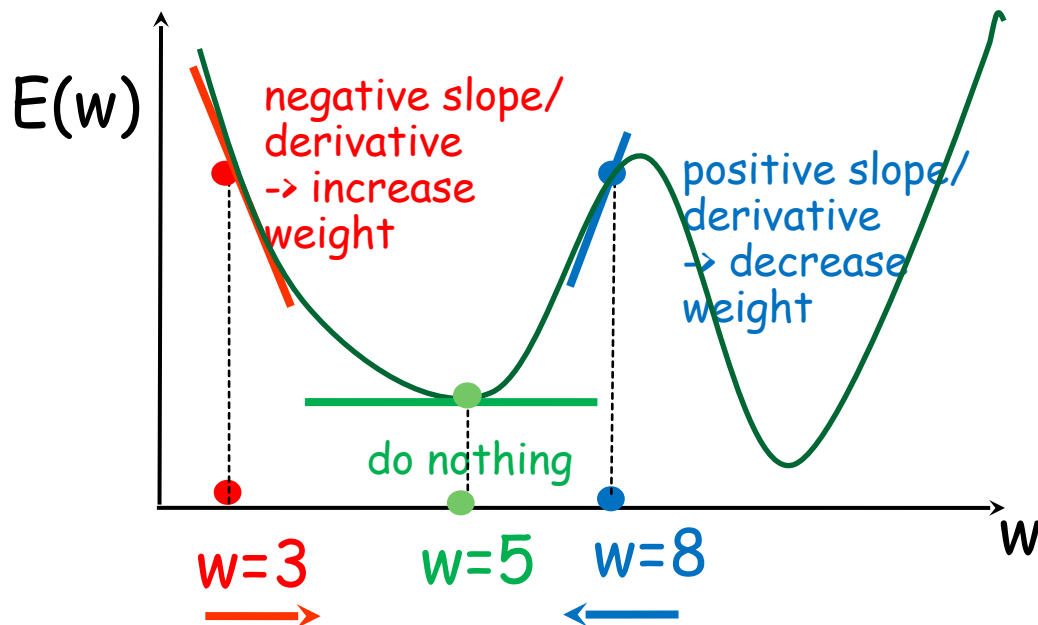
Backpropagation

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

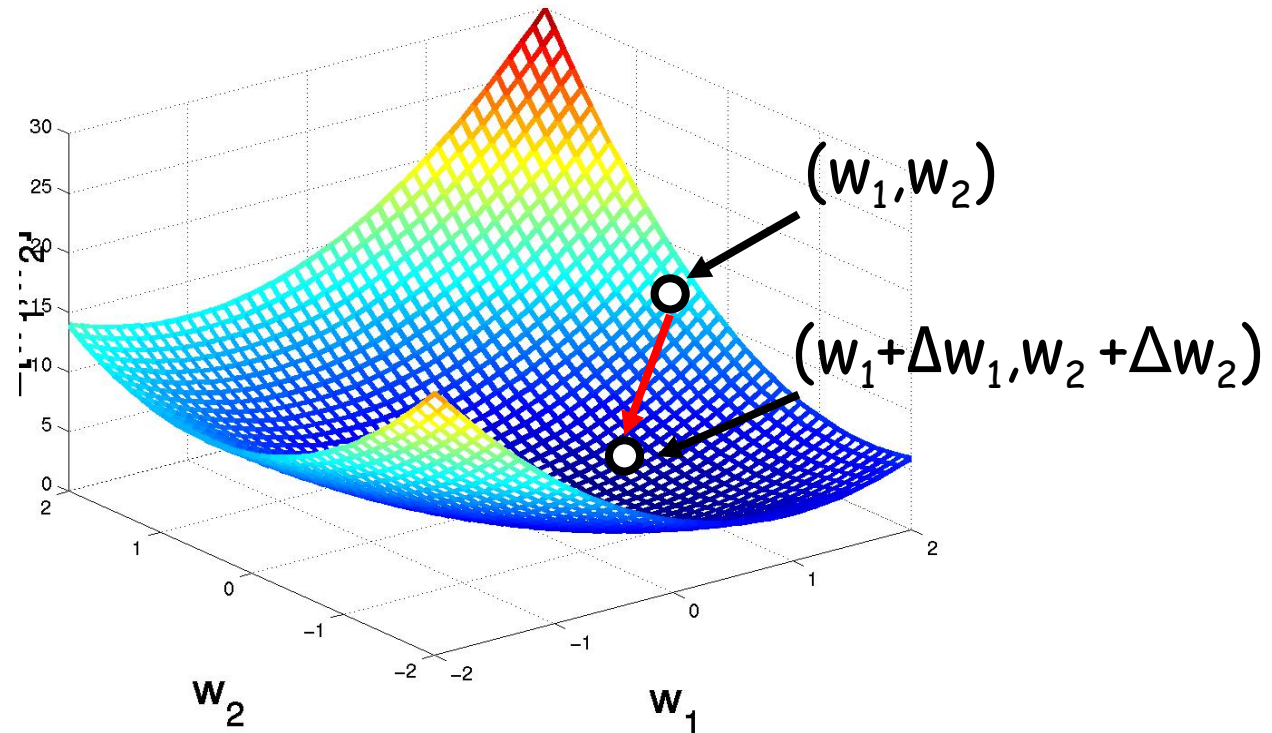
derivative says increase w
(go in opposite direction
of derivative)

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

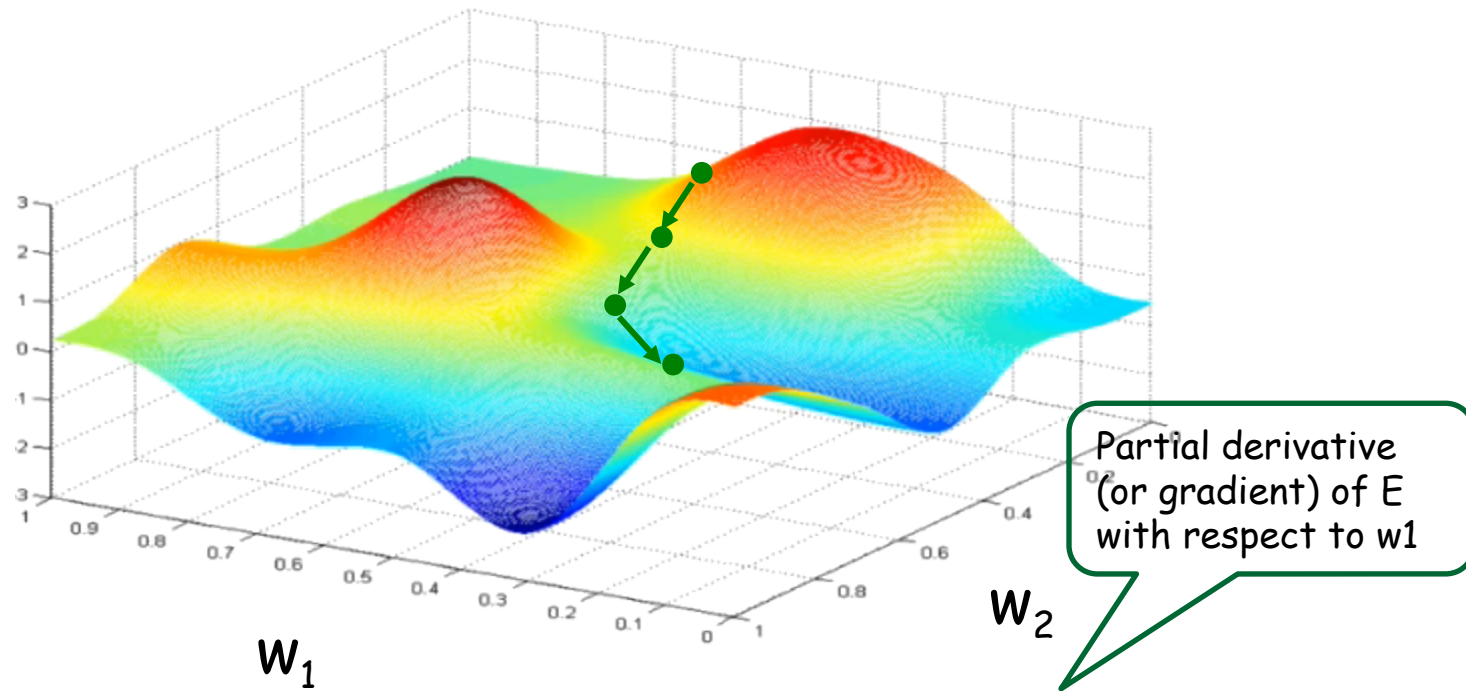
$$E(w_1, w_2) = \frac{1}{2} \frac{\sum_{i=1}^n (T_i - O_i)^2}{n}$$



- Goal: minimize $E(w_1, w_2)$ by changing w_1 and w_2
- But what is the best combination of change in w_1 and w_2 to minimize E faster?
- The delta rule is a gradient descent technique for updating the weights in a single-layer perceptron.

Gradient Descent Visually

$$E(w_1, w_2) = \frac{1}{2} \frac{\sum_{i=1}^n (T_i - O_i)^2}{n}$$



- need to know how much a change in w_1 will affect $E(w_1, w_2)$ i.e. $\frac{\partial E}{\partial w_1}$
- need to know how much a change in w_2 will affect $E(w_1, w_2)$ i.e. $\frac{\partial E}{\partial w_2}$
- **Gradient** ∇E points in the opposite direction of steepest decrease of $E(w_1, w_2)$
- i.e. hill-climbing approach...

Training the Network

After some calculus (see: <https://en.wikipedia.org/wiki/Backpropagation>) we get...

- Step 0: Initialise the weights of the network randomly

// feedforward

- Step 1: Do a forward pass through the network (use sigmoid)

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

Note: To be consistent with Wikipedia, we'll use O-T instead of T-O, but we will subtract the error in the weight update

// propagate the errors backwards

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

$$\delta_k \leftarrow g'(x_k) \times \text{Err}_k = O_k(1 - O_k) \times (O_k - T_k)$$

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

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

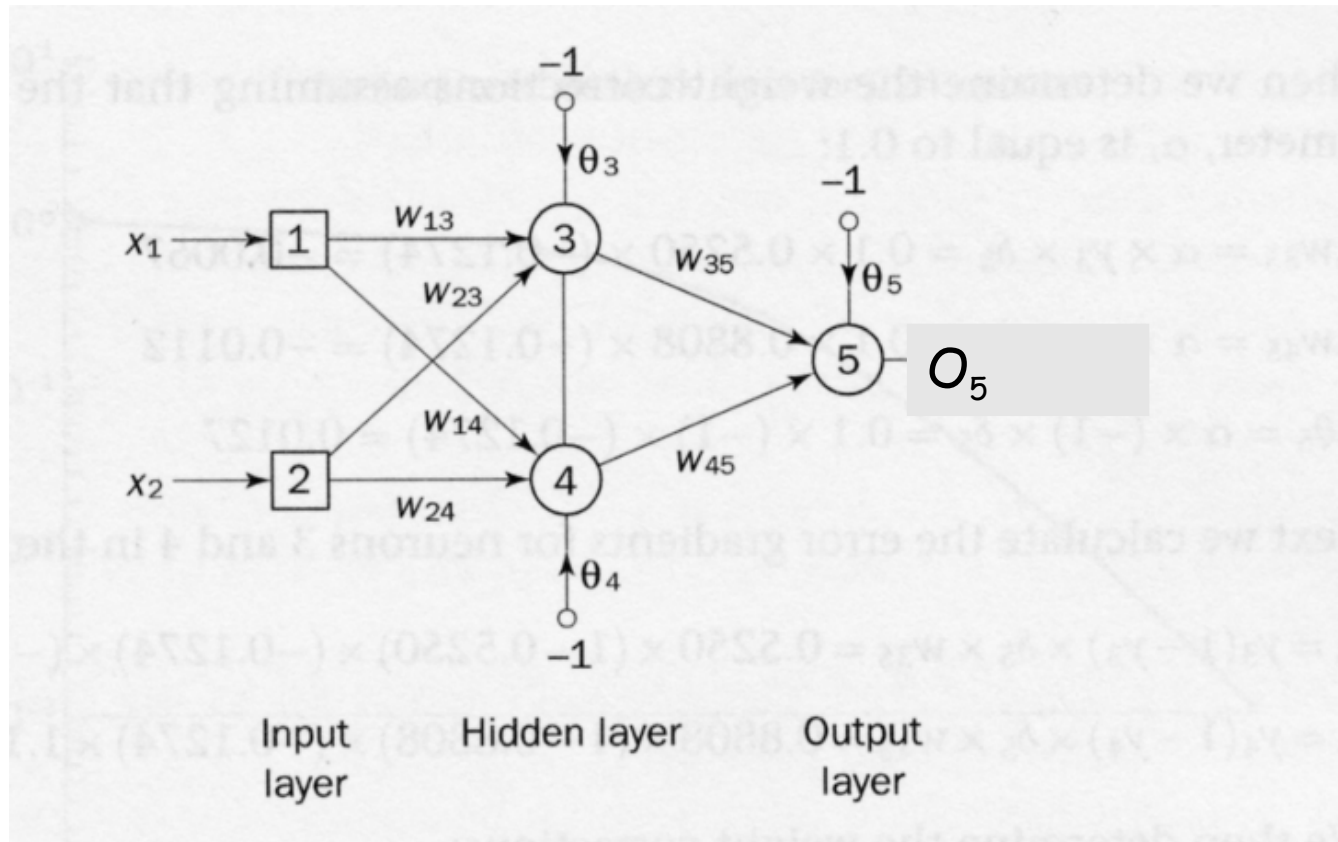
Sum of the weighted error term of the output nodes that h is connected to (ie. h contributed to the errors δ_k)

- 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

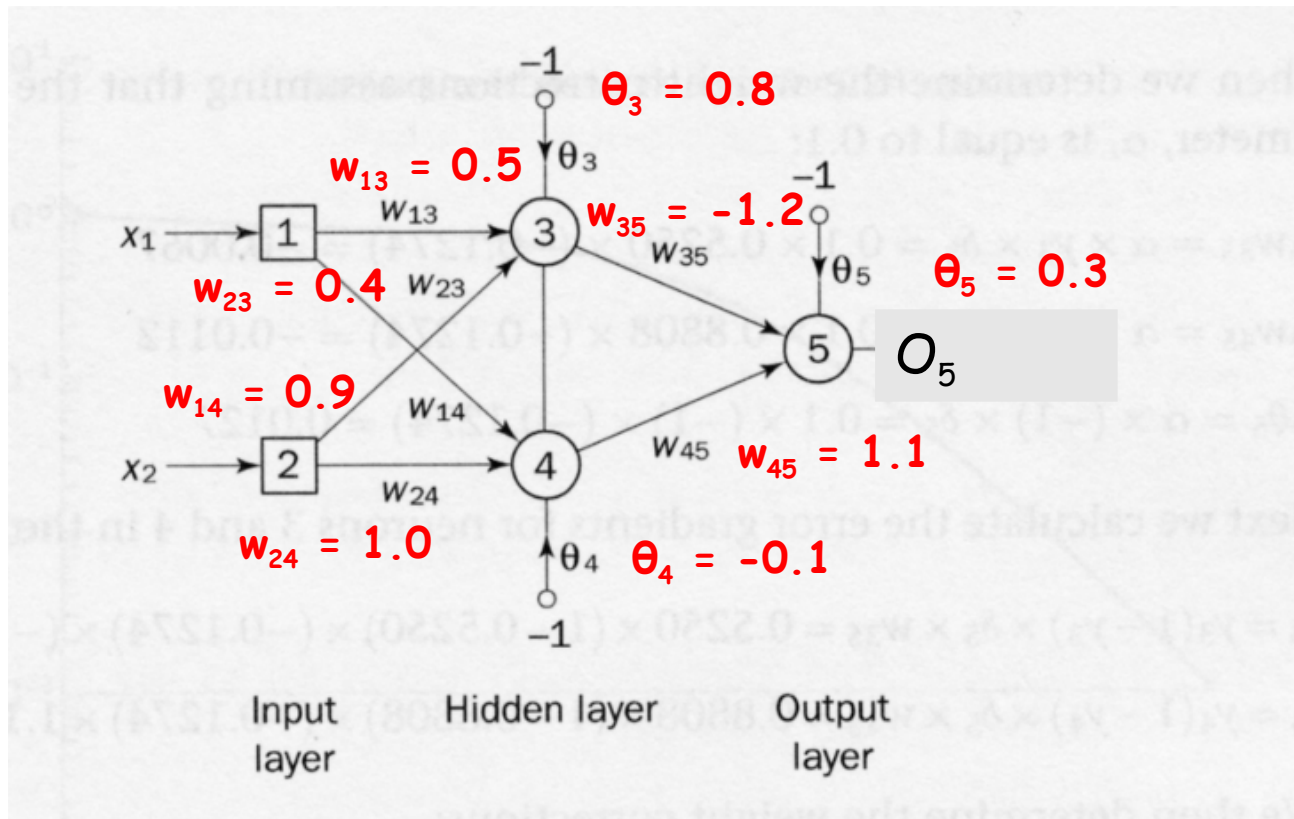
Example: XOR



- 2 inputs + 2 hidden neurons + 1 output neuron + 3 biases (-1)

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

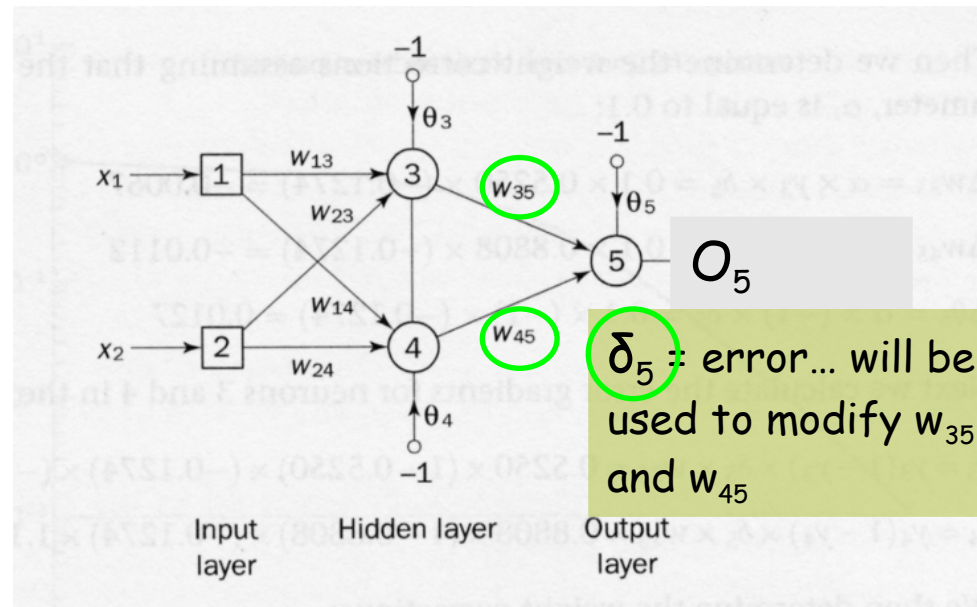
→ **Worksheet #5 (“Neural Network for XOR”: Step 1)**

Step 2: Calculate error term of output layer

$$\delta_k \leftarrow g'(x_k) \times \text{Err}_k = O_k(1 - O_k) \times (O_k - T_k)$$

- Error term of neuron 5 in the output layer:

→ **Worksheet #5** (“Neural Network for XOR”: Step 2)



Step 3: Calculate error term of hidden layer

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

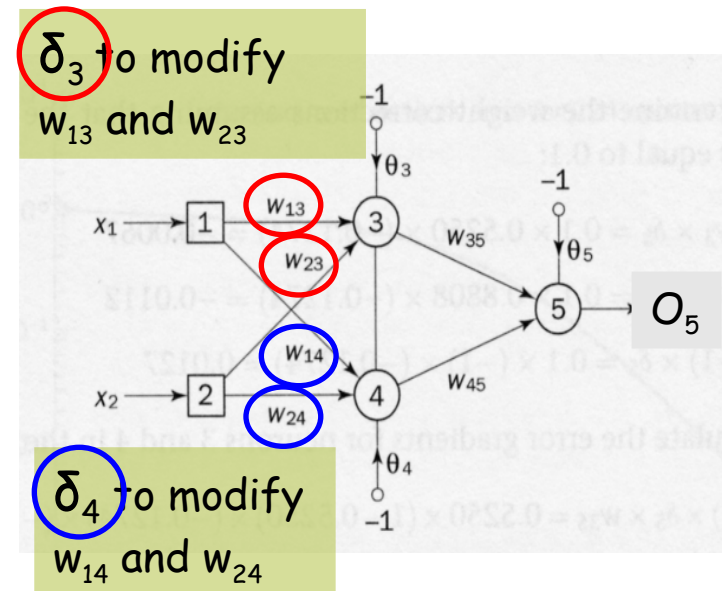
- Error term of neurons 3 & 4 in the hidden layer:

- $\delta_3 = O_3(1 - O_3) \delta_5 w_{35}$
= ...

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

→ **Worksheet #5**

("Neural Network for XOR": Step 3)



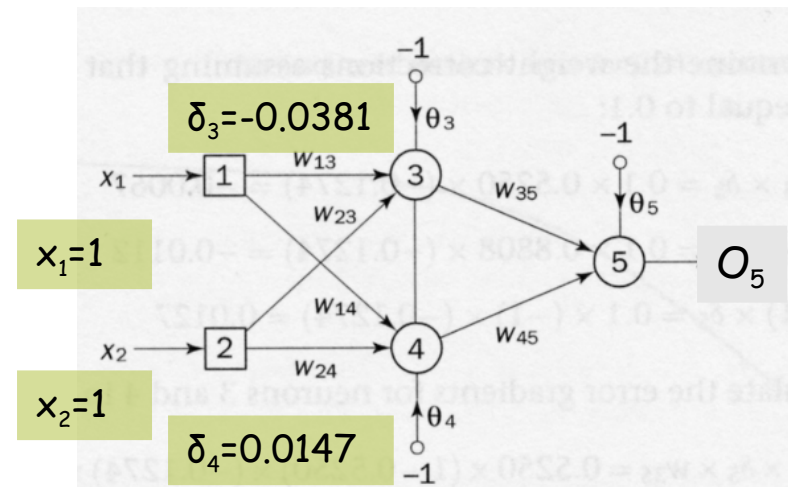
Step 4: Weight Changes

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

- Compute weight changes (assume a constant learning rate $\eta = 0.1$)
 - $\Delta w_{13} = -\eta \delta_3 x_1 = -0.1 \times -0.0381 \times 1 = 0.0038$
 - $\Delta w_{14} = -\eta \delta_4 x_1 =$
 - $\Delta w_{23} = -\eta \delta_3 x_2 = -0.1 \times -0.0381 \times 1 = 0.0038$
 - $\Delta w_{24} = -\eta \delta_4 x_2 =$
 - $\Delta w_{35} = -\eta \delta_5 O_3 = -0.1 \times 0.1274 \times 0.5250 = -0.00669$ // O_3 is seen as x_5 (output of 3 is input to 5)
 - $\Delta w_{45} = -\eta \delta_5 O_4 =$
 - $\Delta \theta_3 = -\eta \delta_3 (-1) = -0.1 \times -0.0381 \times -1 = -0.0038$
 - $\Delta \theta_4 = -\eta \delta_4 (-1) = -0.1 \times 0.0147 \times -1 = 0.0015$
 - $\Delta \theta_5 = -\eta \delta_5 (-1) =$

→ **Worksheet #5**

("Neural Network for XOR": Step 4)

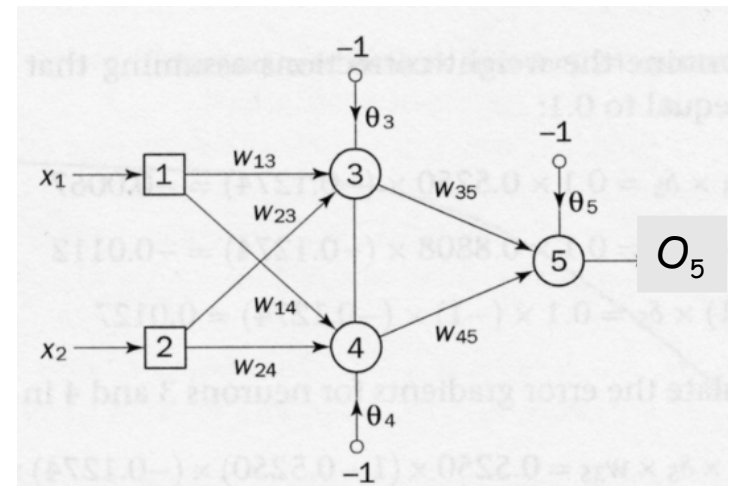


Step 5: Update Weights

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

- Update all weights (assume a constant learning rate $\eta = 0.1$)

- $w_{13} = w_{13} + \Delta w_{13} = 0.5 + 0.0038 = 0.5038$
- $w_{14} = w_{14} + \Delta w_{14} =$
- $w_{23} = w_{23} + \Delta w_{23} = 0.4 + 0.0038 = 0.4038$
- $w_{24} = w_{24} + \Delta w_{24} =$
- $w_{35} = w_{35} + \Delta w_{35} = -1.2 - 0.00669 = -1.20669$
- $w_{45} = w_{45} + \Delta w_{45} =$
- $\theta_3 = \theta_3 + \Delta \theta_3 = 0.8 - 0.0038 = 0.7962$
- $\theta_4 = \theta_4 + \Delta \theta_4 = -0.1 + 0.0015 = -0.0985$
- $\theta_5 = \theta_5 + \Delta \theta_5 =$



→ **Worksheet #5 (“Neural Network for XOR”: Step 5)**

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

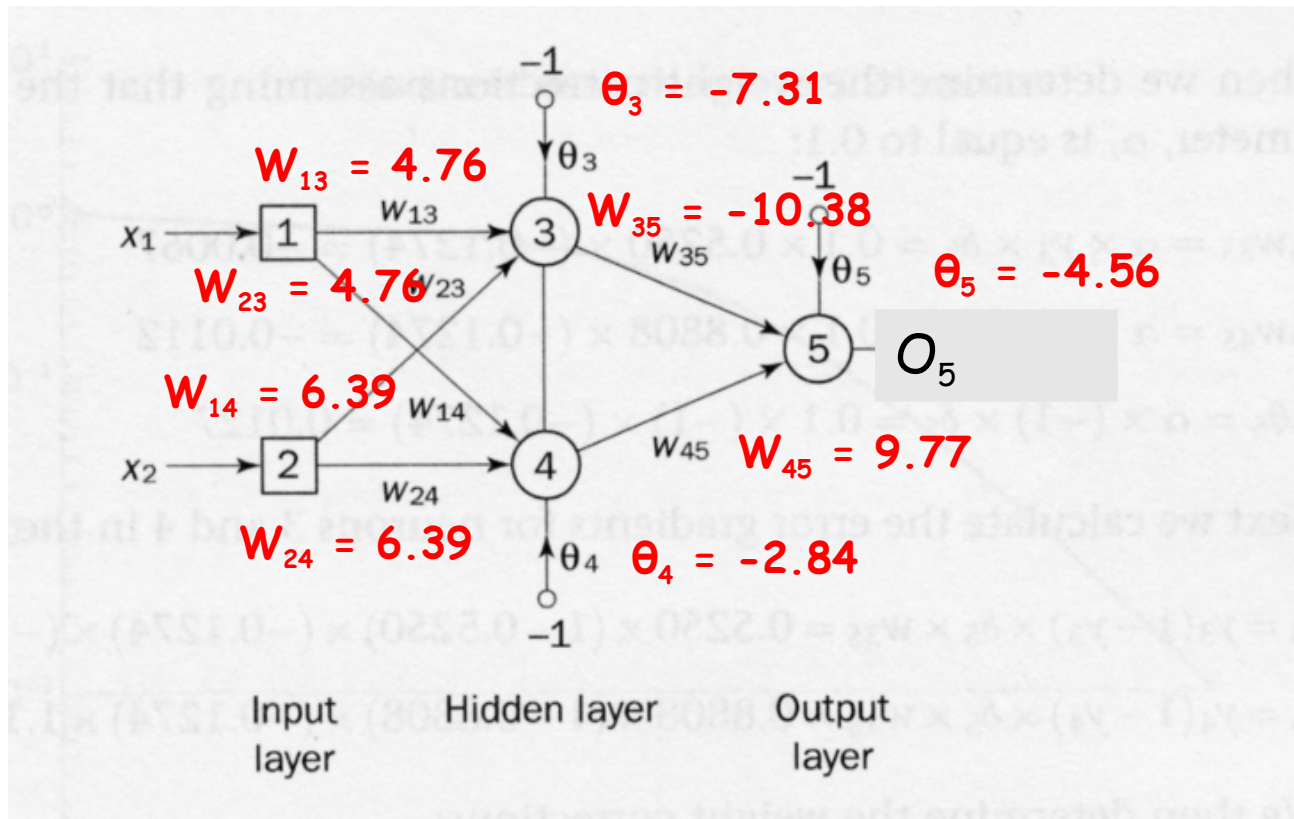
□ Ex: *Mean Squared Error (MSE)*=

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

where $\varepsilon \sim 0.0001$ and n = nb of training examples

The Result...

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



Error is minimized

Inputs		Target Output T	Actual Output O	Error T-O
x_1	x_2			
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

$$\begin{aligned} \text{Mean Squared Error} &= \frac{1}{2} \cdot \frac{(-0.0155)^2 + 0.0151^2 + 0.0151^2 + 0.0175^2}{4} \\ &= 0.000125 < 0.001 \text{ (some threshold } \varepsilon) \text{ ... stop!} \end{aligned}$$



May be a local minimum...

Stochastic Gradient Descent

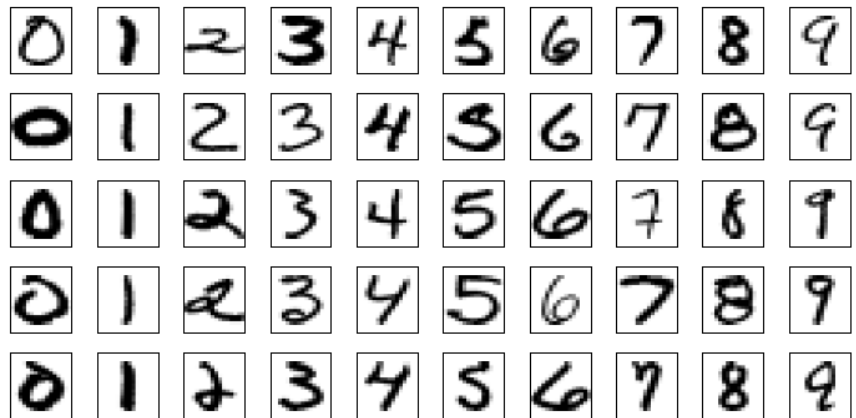
- Batch Gradient Descent (GD)
 - updates the weights after 1 epoch
 - can be costly (time & memory) since we need to evaluate the whole training dataset before we take one step towards the minimum.
- Stochastic Gradient Descent (SGD)
 - updates the weights after each training example
 - often converges 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

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:

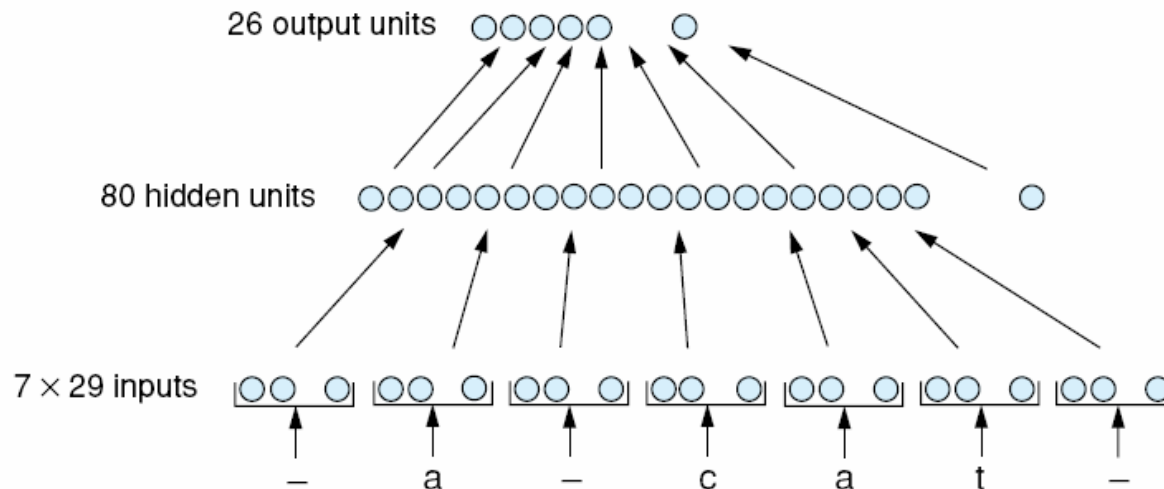
- recognize postal codes
- recognize signatures
- ...



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

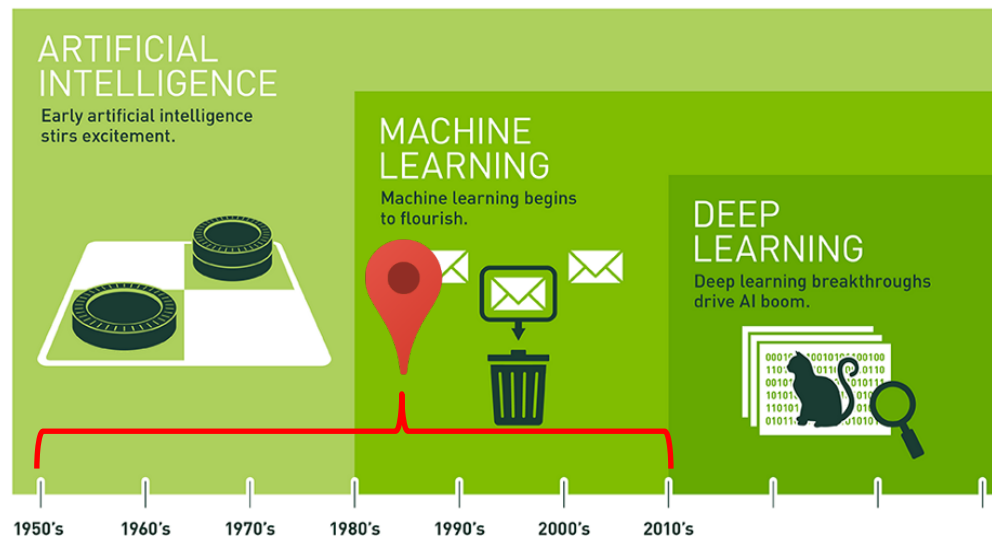
Listen to the output through iterations: <https://www.youtube.com/watch?v=gakJlr3GecE>

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

Today

- Introduction to Neural Networks
 - Perceptrons
 - Backpropagation



YOU ARE HERE!

THE END!