Lecture 1

❖ Introduction to Artificial Neural Networks

❖ Threshold Logic Units

CENG 632- Computational Intelligence, 2024-2025, Spring
Assist. Prof. Dr. Osman GÖKALP

Adapted from lecture notes of Christian Borgelt (https://computational-intelligence.eu/Lecture+Material.html)

Motivation: Why (Artificial) Neural Networks?

- (Neuro-)Biology / (Neuro-)Physiology / Psychology:
  - Exploit similarity to real (biological) neural networks.
  - Build models to understand nerve and brain operation by simulation.

(Artificial) Neural Networks is a highly interdisciplinary research area!

- Computer Science / Engineering / Economics
  - Mimic certain cognitive capabilities of human beings.
  - Solve learning/adaptation, prediction, and optimization problems.

- Physics / Chemistry
  - Use neural network models to describe physical phenomena.
  - Special case: spin glasses (alloys of magnetic and non-magnetic metals).

•**Spin glasses** are disordered magnetic systems where interactions between spins (magnetic moments) are random and frustrated, meaning there is no clear ground state configuration that minimizes energy.
[the energy landscape is rugged, with many metastable states. Finding the ground state (lowest energy configuration) is computationally hard.]

•**Neural networks** are computational models inspired by the brain, consisting of interconnected nodes (neurons) that process information. Training a neural network involves optimizing its parameters (weights) to minimize a loss function.
[the loss function landscape is similarly complex, with many local minima. Training involves navigating this landscape to find a good (not necessarily global) minimum.]

Motivation: Why Neural Networks in AI?

Symbol means a token that refers to an object or a situation.

Physical-Symbol System Hypothesis [Newell and Simon 1976]

A physical-symbol system has the necessary and sufficient means for general intelligent action.

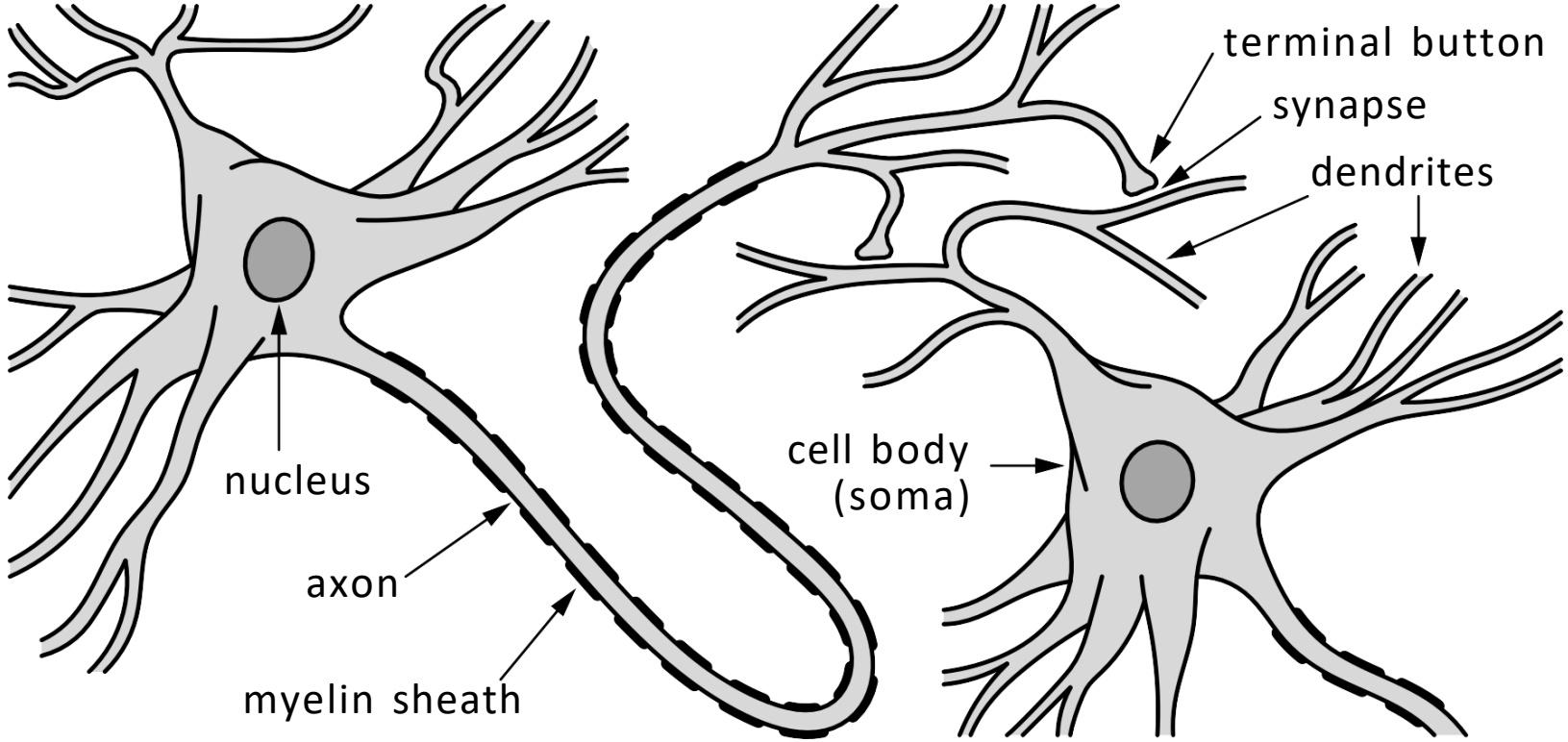
•it claims that any system capable of manipulating symbols (e.g., a computer) can, in principle, exhibit general intelligence. This hypothesis suggests that intelligence arises from the ability to process and manipulate symbolic representations of the world.

Neural networks process simple signals, not symbols (sub-symbolic reasoning).

So **why study neural networks** in Artificial Intelligence?

- Symbol-based representations work well for inference tasks, but are fairly bad for perception tasks.
- Symbol-based expert systems tend to get slower with growing knowledge, human experts tend to get faster.
- Neural networks allow for highly parallel information processing.
- There are several successful applications in industry and finance.

Structure of a prototypical biological neuron (simplified)



Biological Background

(Very) simplified description of neural information processing

- Axon terminal releases chemicals, called neurotransmitters.
- These act on the membrane of the receptor dendrite to change its polarization.
  (The inside is usually 70mV more negative than the outside.)
- Decrease in potential difference: excitatory synapse
  Increase in potential difference: inhibitory synapse
- If there is enough net excitatory input, the axon is depolarized.
- The resulting action potential travels along the axon.
  (Speed depends on the degree to which the axon is covered with myelin.)
- When the action potential reaches the terminal buttons,
  it triggers the release of neurotransmitters.

(Personal) Computers versus the Human Brain

	Personal Computer	Human Brain
processing units	1 CPU, 2–10 cores 10^{10} transistors 1–2 graphics cards/GPUs, 10^{3} cores/shaders 10^{10} transistors	10^{11} neurons
storage capacity	10^{10} bytes main memory (RAM) 10^{12} bytes external memory	10^{11} neurons 10^{14} synapses
processing speed	10^{-9} seconds 10^{9} operations per second	$> 10^{-3}$ seconds < 1000 per second
bandwidth	10^{12} bits/second	10^{14} bits/second
neural updates	10^{6} per second	10^{14} per second

(Personal) Computers versus the Human Brain

- The processing/switching time of a neuron is relatively large ($> 10^{-3}$ seconds), but updates are computed in parallel.
- A serial simulation on a computer takes several hundred clock cycles per update.

Advantages of Neural Networks:

- High processing speed due to massive parallelism.
- Fault Tolerance:
  Remain functional even if (larger) parts of a network get damaged.
- "Graceful Degradation":
  gradual degradation of performance if an increasing number of neurons fail.
- Well suited for inductive learning
  (learning from examples, generalization from instances).

It appears to be reasonable to try to mimic or to recreate these advantages by constructing artificial neural networks.

Threshold Logic Units

Threshold Logic Units

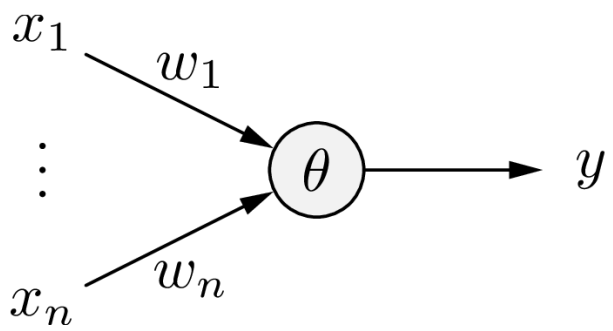
A **Threshold Logic Unit (TLU)** is a processing unit for numbers with n inputs $x_1, \ldots, x_n$ and one output y. The unit has a **threshold** θ and each input x_i is associated with a **weight** w_i. A threshold logic unit computes the function

$$y = \begin{cases} 1, & \text{if } \sum_{i=1}^{n} w_i x_i \geq \theta, \\ 0, & \text{otherwise.} \end{cases}$$

input vector: $x = (x_1, \ldots, x_n)^\top$

weight vector: $w = (w_1, \ldots, w_n)^\top$

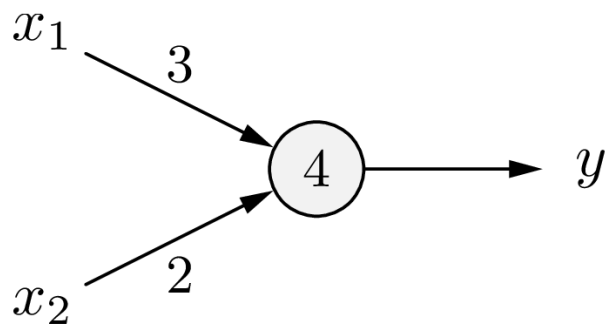
Threshold calc.: $w^\top x \geq \theta$



TLUs mimic the thresholding behavior of biological neurons in a (very) simple fashion.

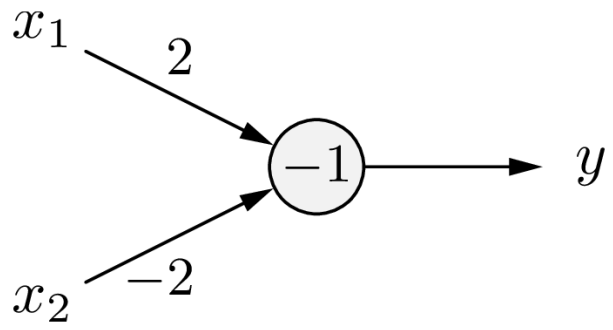
Threshold Logic Units: Examples

Threshold logic unit for the conjunction $x_1 \wedge x_2$.



x_1	x_2	$3x_1 + 2x_2$	y
0	0	0	0
1	0	3	0
0	1	2	0
1	1	5	1

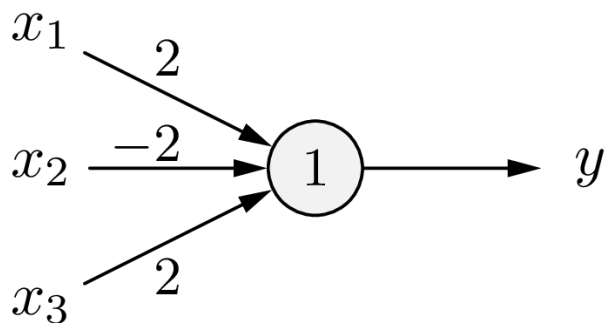
Threshold logic unit for the implication $x_2 \rightarrow x_1$.



x_1	x_2	$2x_1 - 2x_2$	y
0	0	0	1
1	0	2	1
0	1	-2	0
1	1	0	1

Threshold Logic Units: Examples

Threshold logic unit for $(x_1 \wedge \overline{x_2}) \vee (x_1 \wedge x_3) \vee (\overline{x_2} \wedge x_3)$**.**



x_1	x_2	x_3	$\sum_i w_i x_i$	y
0	0	0	0	0
1	0	0	2	1
0	1	0	−2	0
1	1	0	0	0
0	0	1	2	1
1	0	1	4	1
0	1	1	0	0
1	1	1	2	1

Rough Intuition:

- Positive weights are analogous to excitatory synapses.
- Negative weights are analogous to inhibitory synapses.

Threshold Logic Units: Geometric Interpretation

Review of line representations

Straight lines are usually represented in one of the following forms:

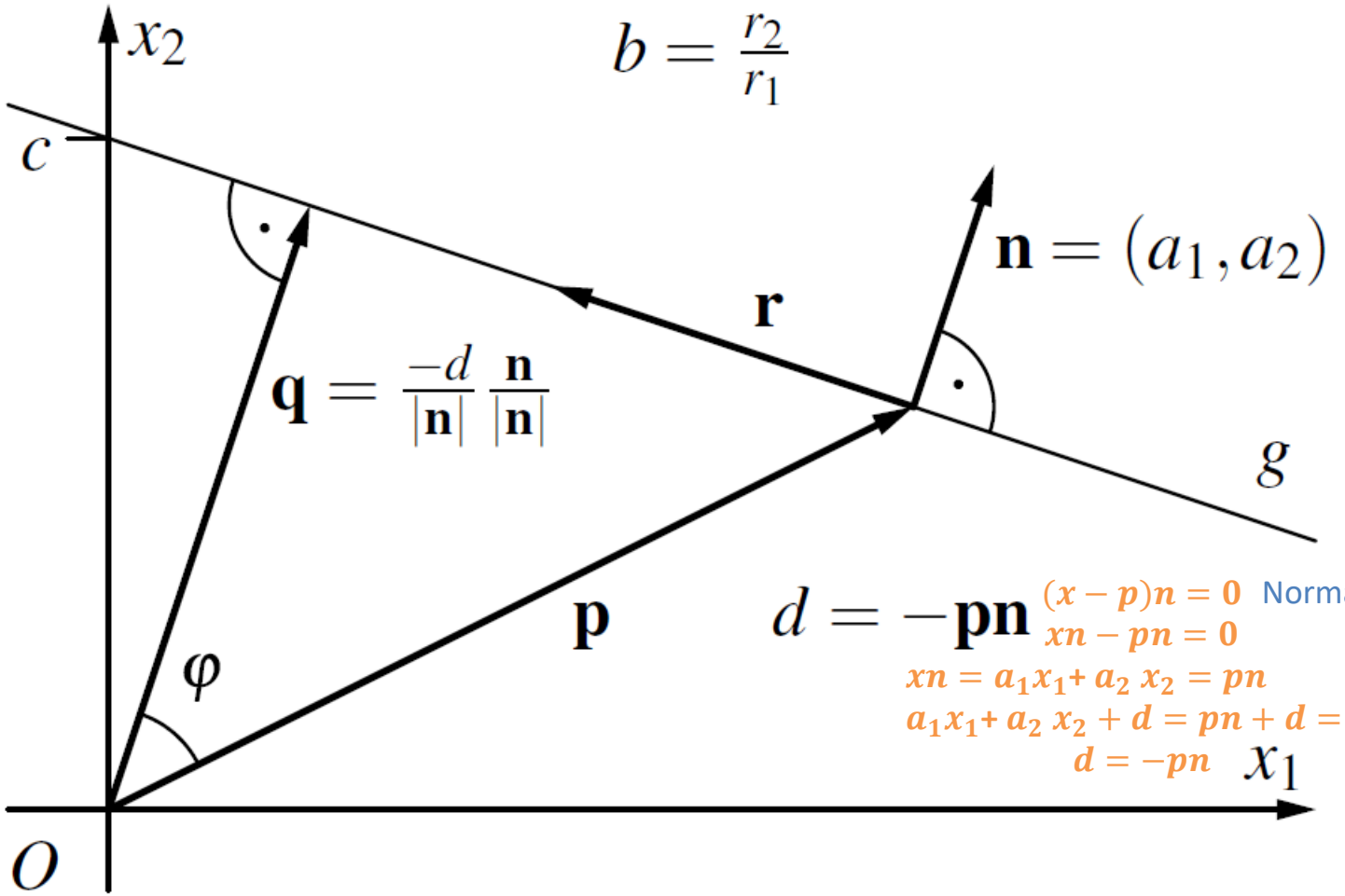
explicit form:	g	$\equiv \quad x_2 = bx_1 + c$
implicit form:	g	$\equiv \quad a_1 x_1 + a_2 x_2 + d = 0$
point-direction form:	g	$\equiv \quad \boldsymbol{x} = \boldsymbol{p} + k\boldsymbol{r}$
normal form:	g	$\equiv \quad (\boldsymbol{x} - \boldsymbol{p})\boldsymbol{n} = 0$

with the parameters:

b : slope of the line
c : intercept (x_2-value at $x_1 = 0$)
$\boldsymbol{p}$: position vector of a point of the line (support vector)
$\boldsymbol{r}$: direction vector of the line
$\boldsymbol{n}$: normal vector of the line. $\boldsymbol{n} = (\boldsymbol{a_1}, \boldsymbol{a_2})$

Threshold Logic Units: Geometric Interpretation

A straight line and its defining parameters:



Vector **q** is the projection of the **p** onto **n**

$(x - p)n = 0$ Normal form

$xn - pn = 0$

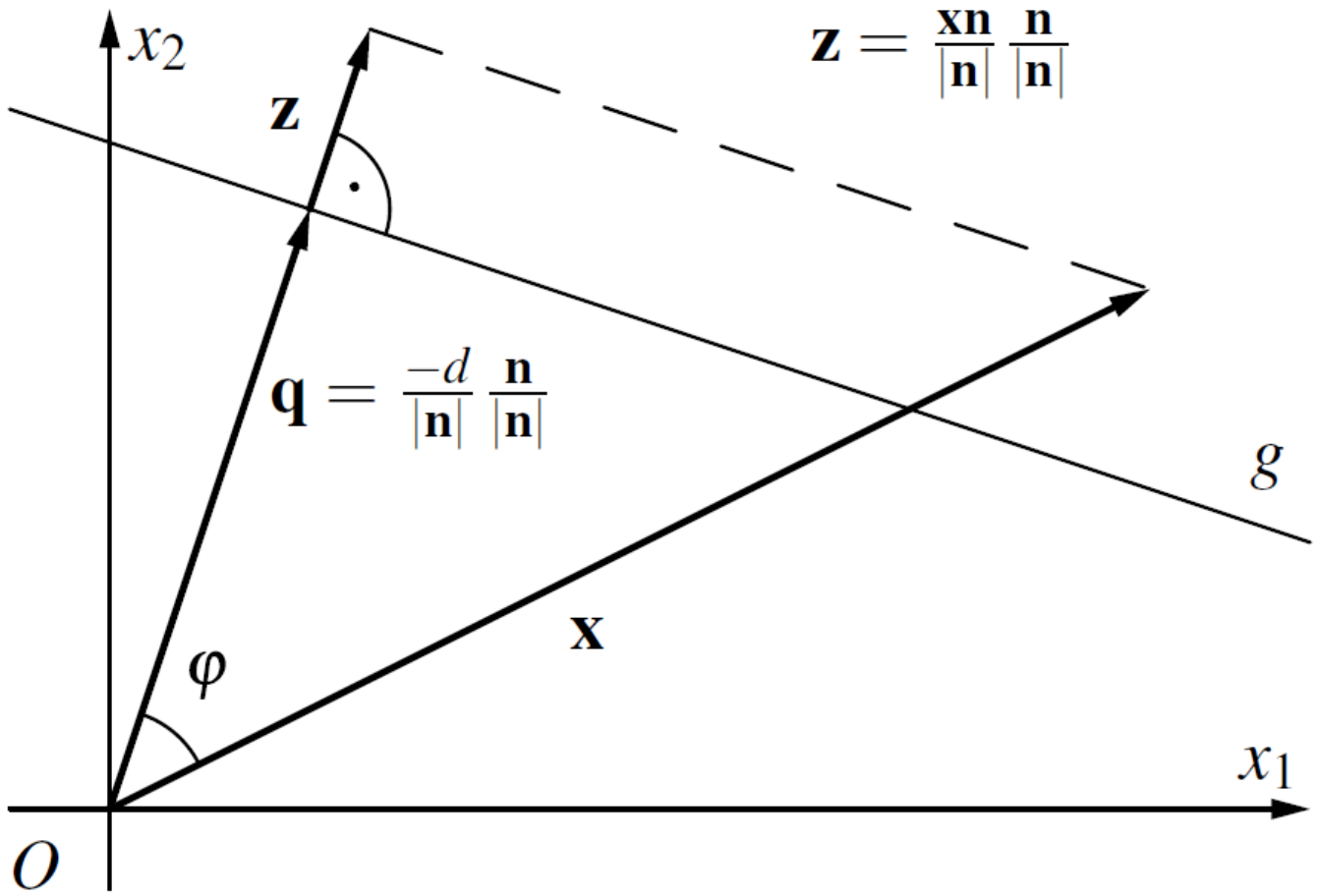
$xn = a_1x_1 + a_2x_2 = pn$

$a_1x_1 + a_2x_2 + d = pn + d = 0$ Implicit form

$d = -pn$

Threshold Logic Units: Geometric Interpretation

How to determine the side on which a point x lies:



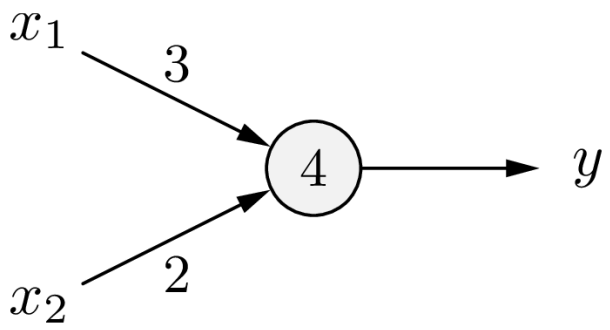
if xn>-d
point x lies on the side
that normal vector points

if xn<-d
point x lies on the opposite side
that normal vector points

if xn=-d
point x lies on the
straight line

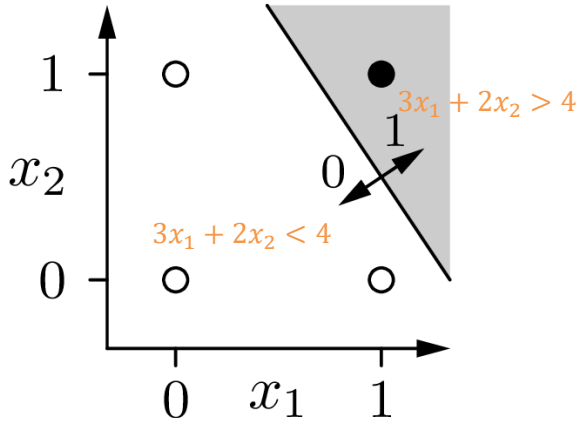
Threshold Logic Units: Geometric Interpretation

Threshold logic unit for $x_1 \wedge x_2$.

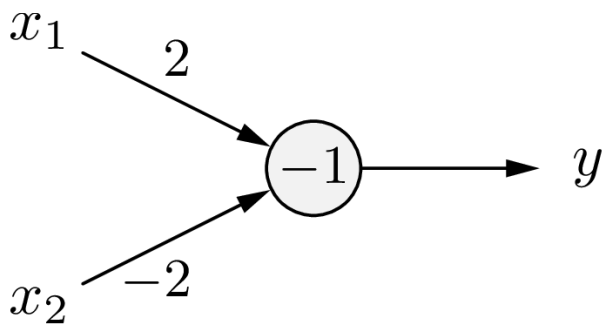


separating line: $3x_1 + 2x_2 = 4$

$n = (3,2)$ *points to the top right*

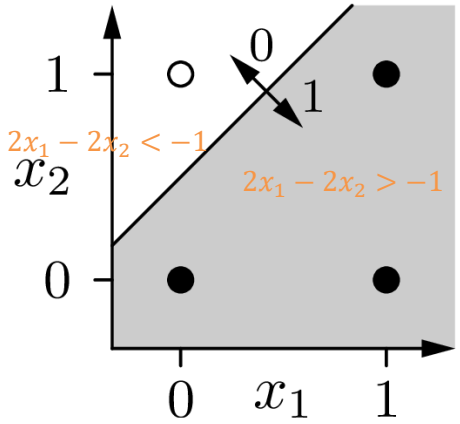


Threshold logic unit for $x_2 \rightarrow x_1$.



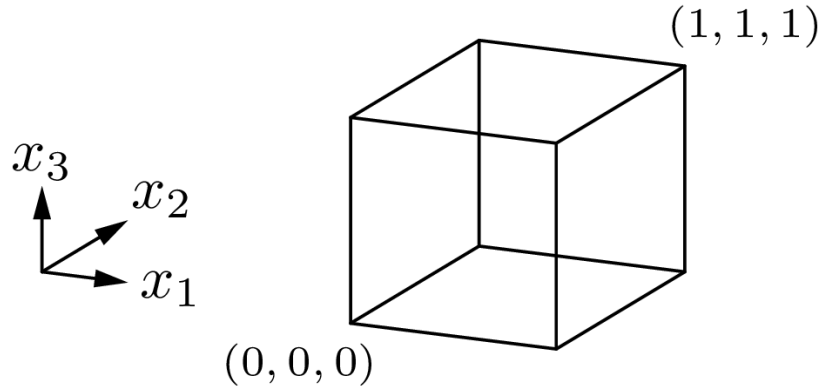
separating line: $2x_1 - 2x_2 = -1$

$n = (2,-2)$ *points to the bottom right*

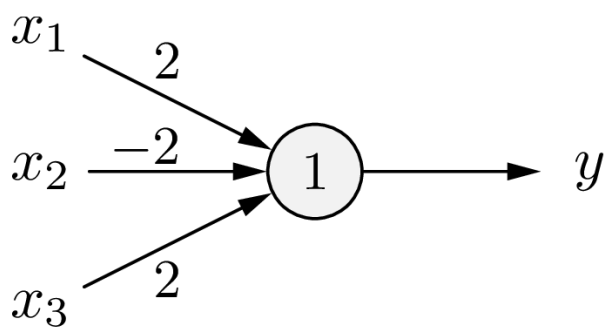


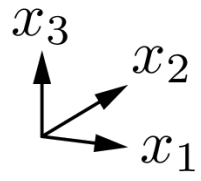
Threshold Logic Units: Geometric Interpretation

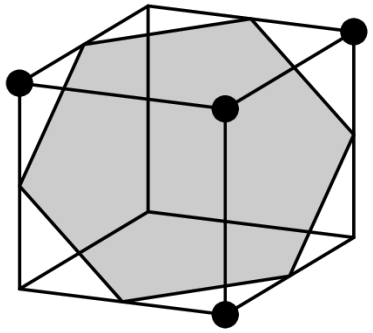
Visualization of 3-dimensional Boolean functions:



Threshold logic unit for $(x_1 \wedge \overline{x_2}) \vee (x_1 \wedge x_3) \vee (\overline{x_2} \wedge x_3)$**.**



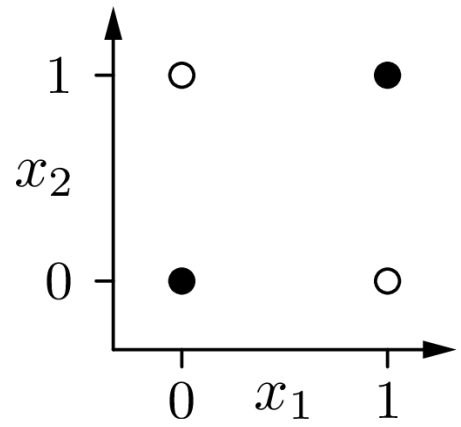




Threshold Logic Units: Limitations

The biimplication problem $x_1 \leftrightarrow x_2$: There is no separating line.

x_1	x_2	y
0	0	1
1	0	0
0	1	0
1	1	1



Formal proof by *reductio ad absurdum*:

$$\begin{aligned}
&\text{since } (0,0) \mapsto 1: && 0 && \geq \theta, && (1)\\
&\text{since } (1,0) \mapsto 0: && w_1 && < \theta, && (2)\\
&\text{since } (0,1) \mapsto 0: && w_2 && < \theta, && (3)\\
&\text{since } (1,1) \mapsto 1: && w_1 + w_2 && \geq \theta. && (4)
\end{aligned}$$

(2) and (3): $w_1 + w_2 < 2\theta$. With (4): $2\theta > \theta$, or $\theta > 0$. Contradiction to (1).

Because Θ is negative!

Linear Separability

Definition: Two sets of points in a Euclidean space are called **linearly separable**, iff there exists at least one point, line, plane or hyperplane (depending on the dimension of the Euclidean space), such that all points of the one set lie on one side and all points of the other set lie on the other side of this point, line, plane or hyperplane (or on it). That is, the point sets can be separated by a **linear decision function**. Formally: Two sets $X, Y \subset \mathbb{R}^m$ are linearly separable iff $\vec{w} \in \mathbb{R}^m$ and $\theta \in \mathbb{R}$ exist such that

$$\forall \vec{x} \in X: \quad \vec{w}^\top \vec{x} < \theta \qquad \text{and} \qquad \forall \vec{y} \in Y: \quad \vec{w}^\top \vec{y} \geq \theta.$$

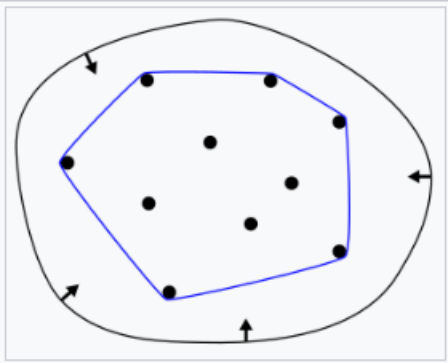
- **Boolean functions** define two points sets, namely the set of points that are mapped to the function value 0 and the set of points that are mapped to 1.
  $\Rightarrow$ The term "linearly separable" can be transferred to Boolean functions.
- As we have seen, **conjunction** and **implication** are **linearly separable** (as are **disjunction**, NAND, NOR etc.).
- The **biimplication** is **not linearly separable** (and neither is the **exclusive or** (XOR)).

Linear Separability

Definition: A set of points in a Euclidean space is called **convex** if it is non-empty and connected (that is, if it is a *region*) and for every pair of points in it every point on the straight line segment connecting the points of the pair is also in the set.

Definition: The **convex hull** of a set of points X in a Euclidean space is the smallest convex set of points that contains X. Alternatively, the **convex hull** of a set of points X is the intersection of all convex sets that contain X.

Theorem: Two sets of points in a Euclidean space are **linearly separable** if and only if their convex hulls are disjoint (that is, have no point in common).



Convex hull of a bounded planar set: rubber band analogy

https://en.wikipedia.org/wiki/Convex_hull

- For the biimplication problem, the convex hulls are the diagonal line segments.
- They share their intersection point and are thus not disjoint.
- Therefore the biimplication is not linearly separable.

Threshold Logic Units: Limitations

Total number and number of linearly separable Boolean functions (On-Line Encyclopedia of Integer Sequences, `oeis.org`, A001146 and A000609):

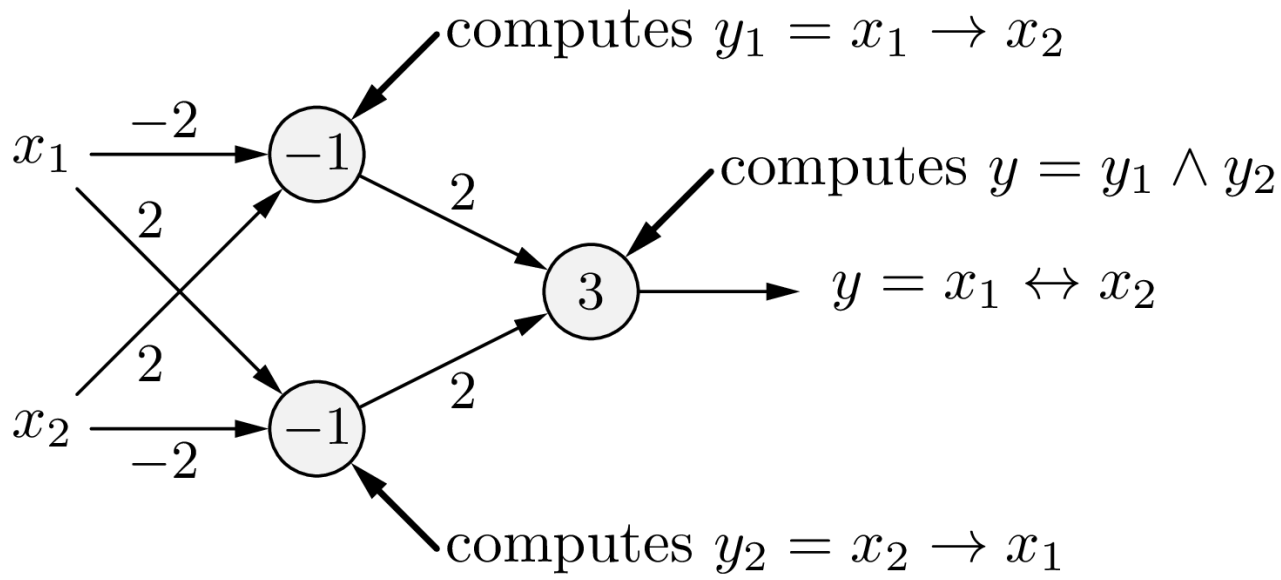
inputs	Boolean functions	linearly separable functions
1	4	4
2	16	14
3	256	104
4	65,536	1,882
5	4,294,967,296	94,572
6	18,446,744,073,709,551,616	15,028,134
n	$2^{(2^n)}$	no general formula known

- For many inputs a threshold logic unit can compute almost no functions.
- Networks of threshold logic units are needed to overcome the limitations.

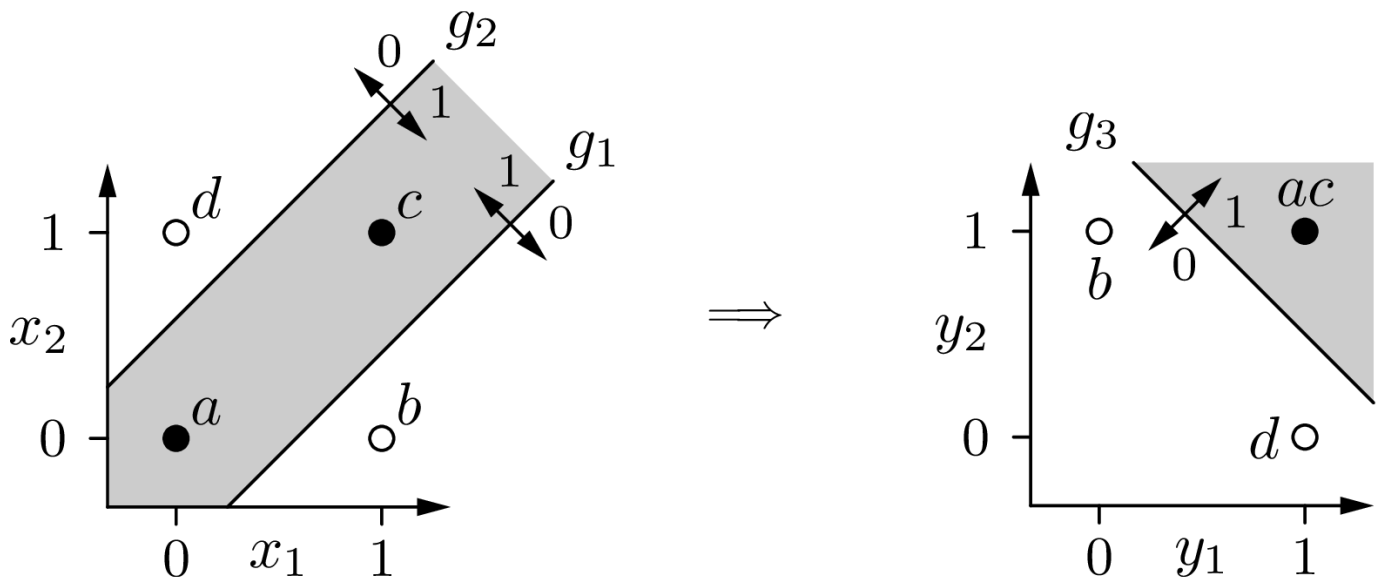
Networks of Threshold Logic Units

Solving the biimplication problem with a network.

Idea: logical decomposition $x_1 \leftrightarrow x_2 \equiv (x_1 \rightarrow x_2) \wedge (x_2 \rightarrow x_1)$



Solving the biimplication problem: Geometric interpretation



x1	x2	y1	y2	
0	0	1	1	*a*
1	0	0	1	*b*
0	1	1	0	*d*
1	1	1	1	*c*

- The first layer computes new Boolean coordinates for the points.
- After the coordinate transformation the problem is linearly separable.

Representing Arbitrary Boolean Functions

The networks only need to have two layers, regardless of the Boolean function to represent:

Algorithm: Let $y = f(x_1, \ldots, x_n)$ be a Boolean function of n variables.

(i) Represent the given function $f(x_1, \ldots, x_n)$ in disjunctive normal form. That is, determine $D_f = C_1 \vee \ldots \vee C_m$, where all C_j are conjunctions of n literals, that is, $C_j = l_{j1} \wedge \ldots \wedge l_{jn}$ with $l_{ji} = x_i$ (positive literal) or $l_{ji} = \neg x_i$ (negative literal).

(ii) Create a neuron for each conjunction C_j of the disjunctive normal form (having n inputs — one input for each variable), where

$$w_{ji} = \begin{cases} 2, & \text{if } l_{ji} = x_i, \\ -2, & \text{if } l_{ji} = \neg x_i, \end{cases} \qquad \text{and} \qquad \theta_j = n - 1 + \frac{1}{2} \sum_{i=1}^{n} w_{ji}.$$

(iii) Create an output neuron (having m inputs — one input for each neuron that was created in step (ii)), where

$$w_{(n+1)k} = 2, \quad k = 1, \ldots, m, \qquad \text{and} \qquad \theta_{n+1} = 1.$$

Remark: weights are set to ± 2 instead of ± 1 in order to ensure integer thresholds.

Representing Arbitrary Boolean Functions

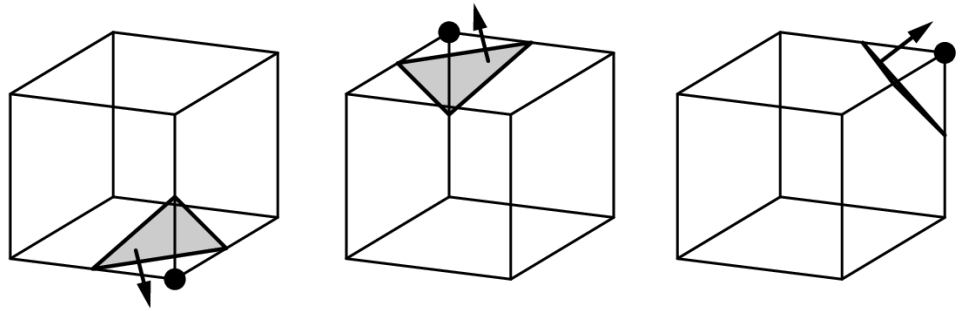
Example:
ternary Boolean function:

x_1	x_2	x_3	y	C_j
0	0	0	0	
1	0	0	1	$x_1 \wedge \overline{x_2} \wedge \overline{x_3}$
0	1	0	0	
1	1	0	0	
0	0	1	0	
1	0	1	0	
0	1	1	1	$\overline{x_1} \wedge x_2 \wedge x_3$
1	1	1	1	$x_1 \wedge x_2 \wedge x_3$

$$D_f = C_1 \vee C_2 \vee C_3$$

One conjunction for each row where the output y is 1 with literals according to input values.

First layer (conjunctions):

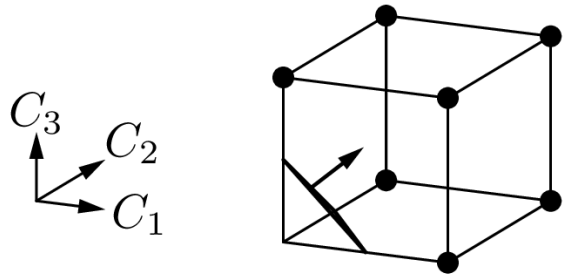


$C_1 = x_1 \wedge \overline{x_2} \wedge \overline{x_3}$

$C_2 = \overline{x_1} \wedge x_2 \wedge x_3$

$C_3 = x_1 \wedge x_2 \wedge x_3$

Second layer (disjunction):



$$D_f = C_1 \vee C_2 \vee C_3$$

Representing Arbitrary Boolean Functions

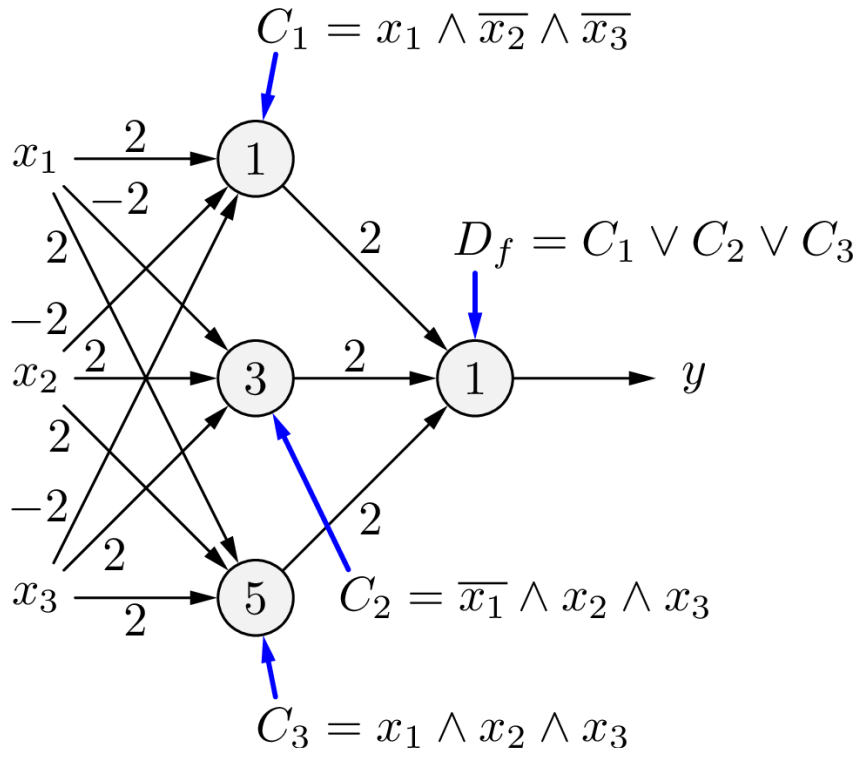
Example:
ternary Boolean function:

x_1	x_2	x_3	y	C_j
0	0	0	0	
1	0	0	1	$x_1 \wedge \overline{x_2} \wedge \overline{x_3}$
0	1	0	0	
1	1	0	0	
0	0	1	0	
1	0	1	0	
0	1	1	1	$\overline{x_1} \wedge x_2 \wedge x_3$
1	1	1	1	$x_1 \wedge x_2 \wedge x_3$

$D_f = C_1 \vee C_2 \vee C_3$

One conjunction for each row where the output y is 1 with literals according to input value.

Resulting network of threshold logic units:

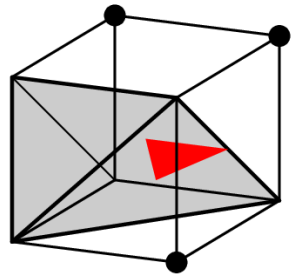


Reminder: Convex Hull Theorem

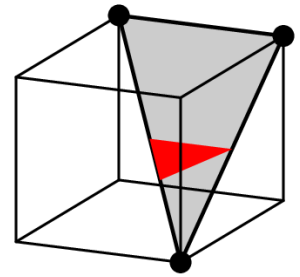
Theorem: Two sets of points in a Euclidean space are **linearly separable** if and only if their convex hulls are disjoint (that is, have no point in common).

Example function on the preceding slide:

$$y = f(x_1, x_2, x_3) = (x_1 \wedge \overline{x_2} \wedge \overline{x_3}) \vee (\overline{x_1} \wedge x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge x_3)$$



Convex hull of points with $y = 0$



Convex hull of points with $y = 1$

- The convex hulls of the two point sets are not disjoint (red: intersection).
- Therefore the function $y = f(x_1, x_2, x_3)$ is not linearly separable.

Training Threshold Logic Units

Training Threshold Logic Units

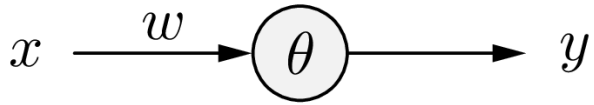
We can decide the values of weights and thresholds by visual interpretation

- Geometric interpretation provides a way to construct threshold logic units with 2 and 3 inputs, but:
  - Not an automatic method (human visualization needed).
  - Not feasible for more than 3 inputs.

- **General idea of automatic training:**
  - Start with random values for weights and threshold.
  - Determine the error of the output for a set of training patterns.
  - Error is a function of the weights and the threshold: $e = e(w_1, \ldots, w_n, \theta)$.
  - Adapt weights and threshold so that the error becomes smaller.
  - Iterate adaptation until the error vanishes.

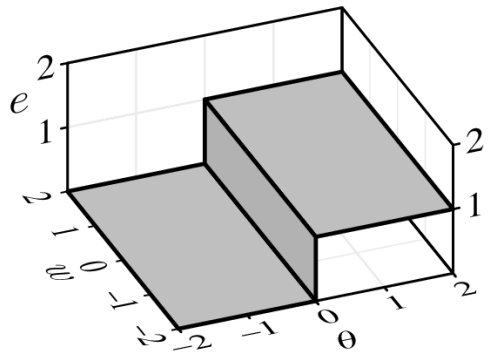
Training Threshold Logic Units

Single input threshold logic unit for the negation $\neg x$.

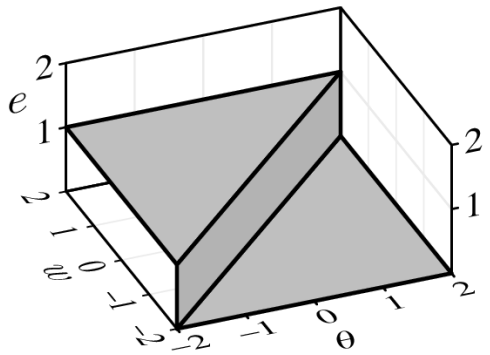


x	y
0	1
1	0

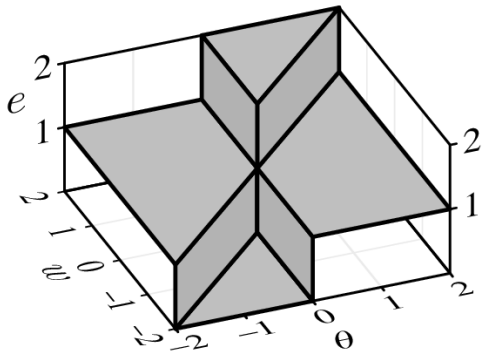
Output error as a function of weight and threshold.



error for $x = 0$



error for $x = 1$

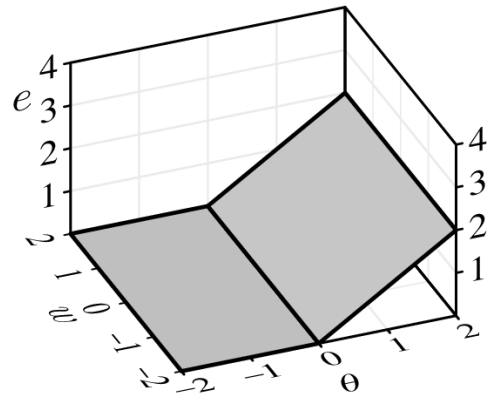


sum of errors

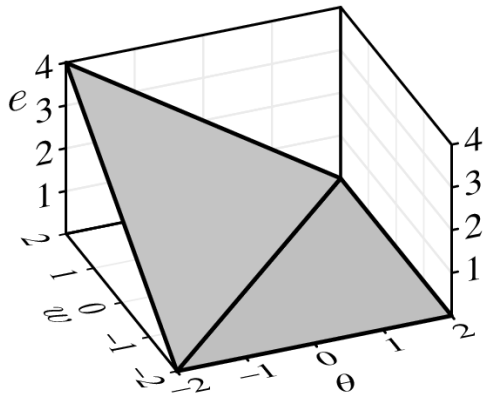
Training Threshold Logic Units

- The error function cannot be used directly, because it consists of plateaus.
- Solution: If the computed output is wrong,
  take into account how far the weighted sum is from the threshold
  (that is, consider "how wrong" the relation of weighted sum and threshold is).

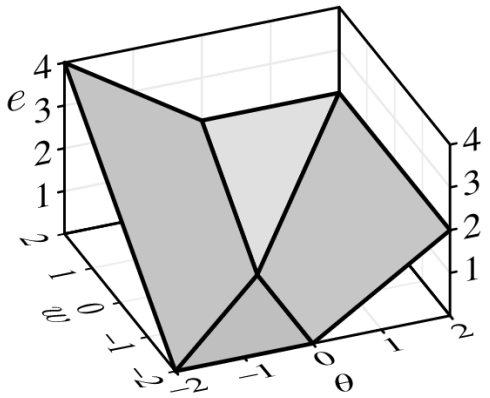
Modified output error as a function of weight and threshold.



error for $x = 0$



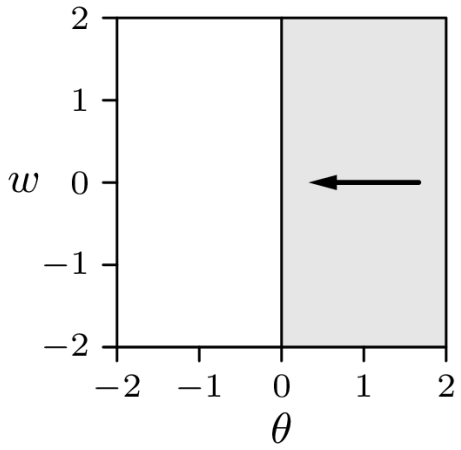
error for $x = 1$



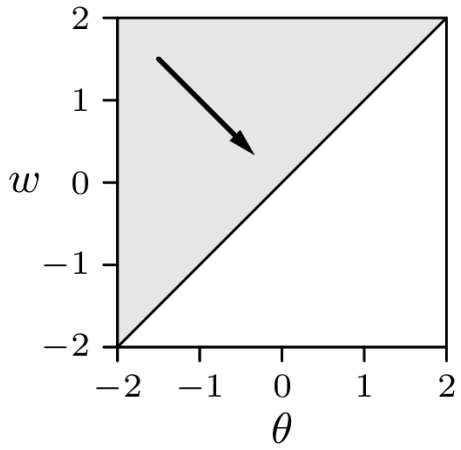
sum of errors

Training Threshold Logic Units

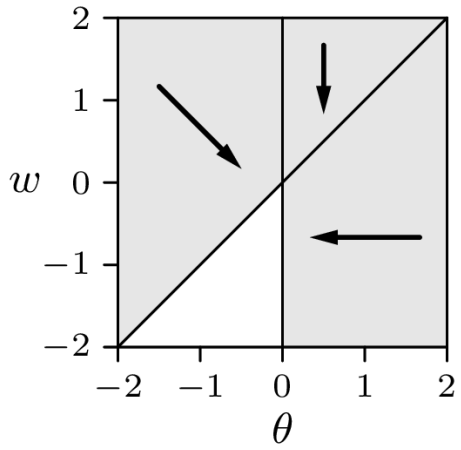
Schemata of resulting directions of parameter changes.



changes for $x = 0$



changes for $x = 1$



sum of changes

- Start at a random point.
- Iteratively adapt parameters
  according to the direction corresponding to the current point.
- Stop if the error vanishes.

Training Threshold Logic Units: Delta Rule

Formal Training Rule: Let $\vec{x} = (x_1, \ldots, x_n)^\top$ be an input vector of a threshold logic unit, o the desired output for this input vector and y the actual output of the threshold logic unit. If $y \neq o$, then the threshold θ and the weight vector $\vec{w} = (w_1, \ldots, w_n)^\top$ are adapted as follows in order to reduce the error:

$$\begin{aligned} \theta^{(\text{new})} &= \theta^{(\text{old})} + \Delta\theta \quad \text{with} \quad \Delta\theta &&= -\eta(o - y), \\ \forall i \in \{1, \ldots, n\}: \quad w_i^{(\text{new})} &= w_i^{(\text{old})} + \Delta w_i \quad \text{with} \quad \Delta w_i &&= \eta(o - y)x_i, \end{aligned}$$

where η is a parameter that is called **learning rate**. It determines the severity of the weight changes. This procedure is called **Delta Rule** or **Widrow–Hoff Procedure** [Widrow and Hoff 1960].

- **Online Training:** Adapt parameters after each training pattern.
- **Batch Training:** Adapt parameters only at the end of each **epoch**, that is, after a traversal of all training patterns.

Training Threshold Logic Units: Delta Rule

```
procedure online_training (var w⃗, var θ, L, η);
var y, e;                                  (* output, sum of errors *)
begin
  repeat                                   (* training loop *)
    e := 0;                                (* initialize the error sum *)
    for all (x⃗, o) ∈ L do begin            (* traverse the patterns *)
      if (w⃗^⊤ x⃗ ≥ θ) then y := 1;          (* compute the output *)
                     else y := 0;          (* of the threshold logic unit *)
      if (y ≠ o) then begin                (* if the output is wrong *)
        θ  := θ  − η(o − y);               (* adapt the threshold *)
        w⃗ := w⃗ + η(o − y)x⃗;               (* and the weights *)
        e  := e  +  |o − y|;               (* sum the errors *)
      end;
    end;
  until (e ≤ 0);                           (* repeat the computations *)
end;                                       (* until the error vanishes *)
```

Training Threshold Logic Units: Delta Rule

```
procedure batch_training (var w⃗, var θ, L, η);
var y, e, θ_c, w⃗_c;                          (* output, sum of errors, sums of changes *)
begin
  repeat                                     (* training loop *)
    e := 0; θ_c := 0; w⃗_c := 0⃗;              (* initializations *)
    for all (x⃗, o) ∈ L do begin              (* traverse the patterns *)
      if (w⃗^⊤ x⃗ ≥ θ) then y := 1;            (* compute the output *)
                     else  y := 0;           (* of the threshold logic unit *)
      if (y ≠ o) then begin                  (* if the output is wrong *)
        θ_c := θ_c − η(o − y);               (* sum the changes of the *)
        w⃗_c := w⃗_c + η(o − y)x⃗;             (* threshold and the weights *)
        e   := e   +  |o − y|;               (* sum the errors *)
      end;
    end;
    θ := θ + θ_c;                            (* adapt the threshold *)
    w⃗ := w⃗ + w⃗_c;                           (* and the weights *)
  until (e ≤ 0);                             (* repeat the computations *)
end;                                         (* until the error vanishes *)
```

Training Threshold Logic Units: Online

Negation problem with initial values $\Theta = 3/2$, $w = 2$ and learning rate $\eta = 1$

epoch	x	o	$\vec{x}\vec{w}$	y	e	$\Delta\theta$	Δw	θ	w
								1.5	2
1	0	1	−1.5	0	1	−1	0	0.5	2
	1	0	1.5	1	−1	1	−1	1.5	1
2	0	1	−1.5	0	1	−1	0	0.5	1
	1	0	0.5	1	−1	1	−1	1.5	0
3	0	1	−1.5	0	1	−1	0	0.5	0
	1	0	0.5	0	0	0	0	0.5	0
4	0	1	−0.5	0	1	−1	0	−0.5	0
	1	0	0.5	1	−1	1	−1	0.5	−1
5	0	1	−0.5	0	1	−1	0	−0.5	−1
	1	0	−0.5	0	0	0	0	−0.5	−1
6	0	1	0.5	1	0	0	0	−0.5	−1
	1	0	−0.5	0	0	0	0	−0.5	−1

$\Delta\Theta = -1(1-0) = -1$

$\Delta w = 1(1-0)0 = 0$

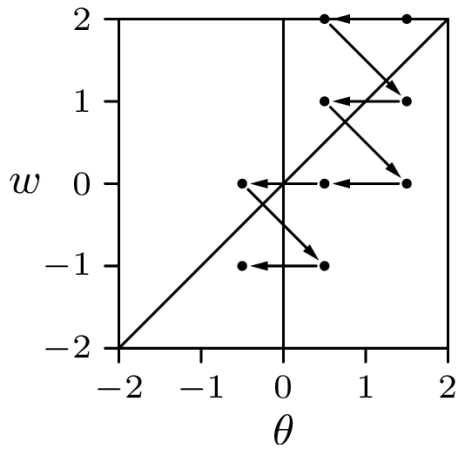
Training Threshold Logic Units: Batch

Negation problem with initial values $\Theta = 3/2$, $w = 2$ and learning rate $\eta = 1$

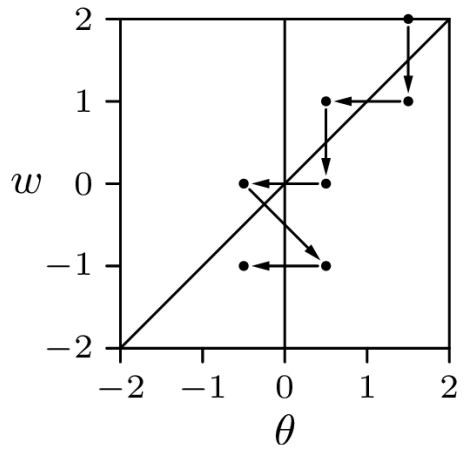
epoch	x	o	$\vec{x}\vec{w}$	y	e	$\Delta\theta$	Δw	θ	w
								1.5	2
1	0	1	−1.5	0	1	−1	0		
	1	0	0.5	1	−1	1	−1	1.5	1
2	0	1	−1.5	0	1	−1	0		
	1	0	−0.5	0	0	0	0	0.5	1
3	0	1	−0.5	0	1	−1	0		
	1	0	0.5	1	−1	1	−1	0.5	0
4	0	1	−0.5	0	1	−1	0		
	1	0	−0.5	0	0	0	0	−0.5	0
5	0	1	0.5	1	0	0	0		
	1	0	0.5	1	−1	1	−1	0.5	−1
6	0	1	−0.5	0	1	−1	0		
	1	0	−1.5	0	0	0	0	−0.5	−1
7	0	1	0.5	1	0	0	0		
	1	0	−0.5	0	0	0	0	−0.5	−1

Training Threshold Logic Units

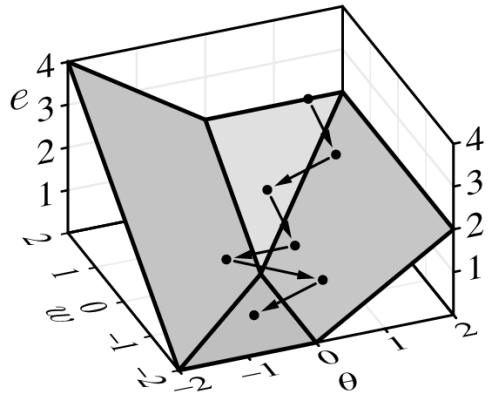
Example training procedure: Online and batch training.



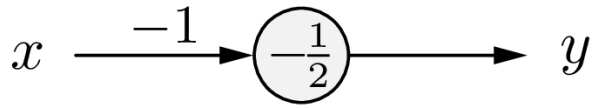
Online Training

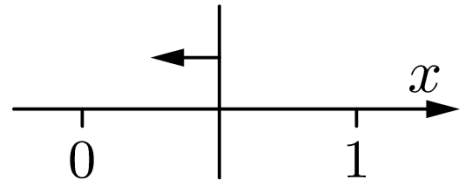


Batch Training



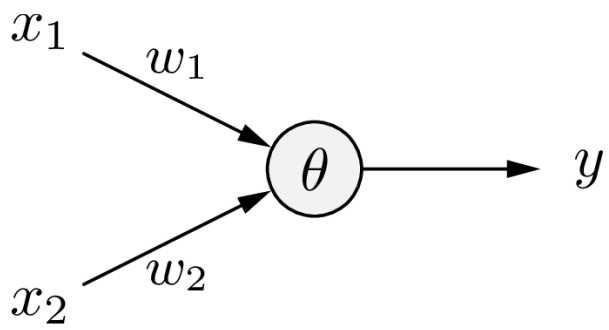
Batch Training



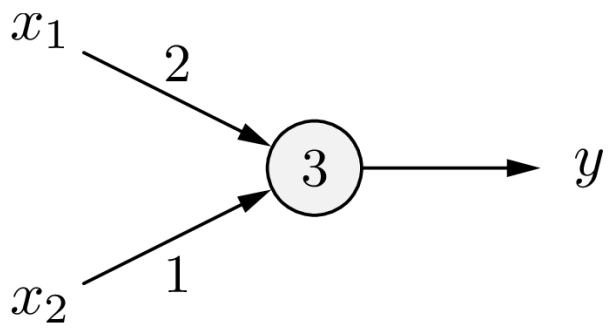


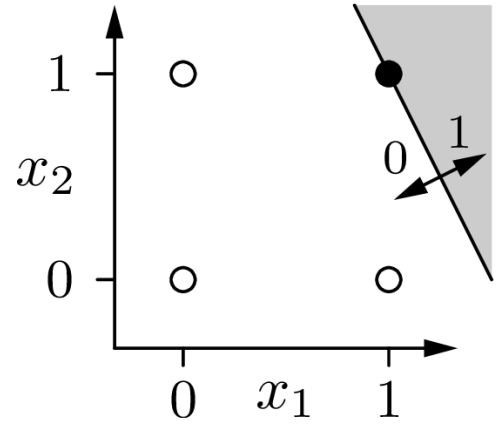
Training Threshold Logic Units: Conjunction

Threshold logic unit with two inputs for the conjunction.



x_1	x_2	y
0	0	0
1	0	0
0	1	0
1	1	1





Training Threshold Logic Units: Conjunction

epoch	x_1	x_2	o	$\vec{x}\vec{w}$	y	e	$\Delta\theta$	Δw_1	Δw_2	θ	w_1	w_2
										0	0	0
1	0	0	0	0	1	−1	1	0	0	1	0	0
	0	1	0	−1	0	0	0	0	0	1	0	0
	1	0	0	−1	0	0	0	0	0	1	0	0
	1	1	1	−1	0	1	−1	1	1	0	1	1
2	0	0	0	0	1	−1	1	0	0	1	1	1
	0	1	0	0	1	−1	1	0	−1	2	1	0
	1	0	0	−1	0	0	0	0	0	2	1	0
	1	1	1	−1	0	1	−1	1	1	1	2	1
3	0	0	0	−1	0	0	0	0	0	1	2	1
	0	1	0	0	1	−1	1	0	−1	2	2	0
	1	0	0	0	1	−1	1	−1	0	3	1	0
	1	1	1	−2	0	1	−1	1	1	2	2	1
4	0	0	0	−2	0	0	0	0	0	2	2	1
	0	1	0	−1	0	0	0	0	0	2	2	1
	1	0	0	0	1	−1	1	−1	0	3	1	1
	1	1	1	−1	0	1	−1	1	1	2	2	2
5	0	0	0	−2	0	0	0	0	0	2	2	2
	0	1	0	0	1	−1	1	0	−1	3	2	1
	1	0	0	−1	0	0	0	0	0	3	2	1
	1	1	1	0	1	0	0	0	0	3	2	1
6	0	0	0	−3	0	0	0	0	0	3	2	1
	0	1	0	−2	0	0	0	0	0	3	2	1
	1	0	0	−1	0	0	0	0	0	3	2	1
	1	1	1	0	1	0	0	0	0	3	2	1

Training Threshold Logic Units: Biimplication

epoch	x_1	x_2	o	$\vec{x}\vec{w}$	y	e	$\Delta\theta$	Δw_1	Δw_2	θ	w_1	w_2
										0	0	0
1	0	0	1	0	1	0	0	0	0	0	0	0
	0	1	0	0	1	−1	1	0	−1	1	0	−1
	1	0	0	−1	0	0	0	0	0	1	0	−1
	1	1	1	−2	0	1	−1	1	1	0	1	0
2	0	0	1	0	1	0	0	0	0	0	1	0
	0	1	0	0	1	−1	1	0	−1	1	1	−1
	1	0	0	0	1	−1	1	−1	0	2	0	−1
	1	1	1	−3	0	1	−1	1	1	1	1	0
3	0	0	1	0	1	0	0	0	0	0	1	0
	0	1	0	0	1	−1	1	0	−1	1	1	−1
	1	0	0	0	1	−1	1	−1	0	2	0	−1
	1	1	1	−3	0	1	−1	1	1	1	1	0

This training never stops!

Training Threshold Logic Units: Convergence

Convergence Theorem for the Delta Rule

Convergence Theorem: Let $L = \{(\vec{x}_1, o_1), \ldots (\vec{x}_m, o_m)\}$ be a set of training patterns, each consisting of an input vector $\vec{x}_i \in \mathbb{R}^n$ and a desired output $o_i \in \{0, 1\}$. Furthermore, let $L_0 = \{(\vec{x}, o) \in L \mid o = 0\}$ and $L_1 = \{(\vec{x}, o) \in L \mid o = 1\}$.
If L_0 and L_1 are linearly separable, that is, if $\vec{w} \in \mathbb{R}^n$ and $\theta \in \mathbb{R}$ exist such that

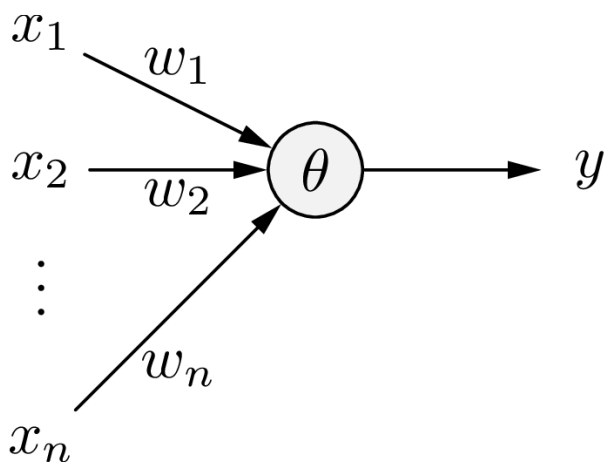
$$\begin{aligned} \forall (\vec{x}, 0) \in L_0 : \quad & \vec{w}^\top \vec{x} < \theta \quad \text{and} \\ \forall (\vec{x}, 1) \in L_1 : \quad & \vec{w}^\top \vec{x} \geq \theta, \end{aligned}$$

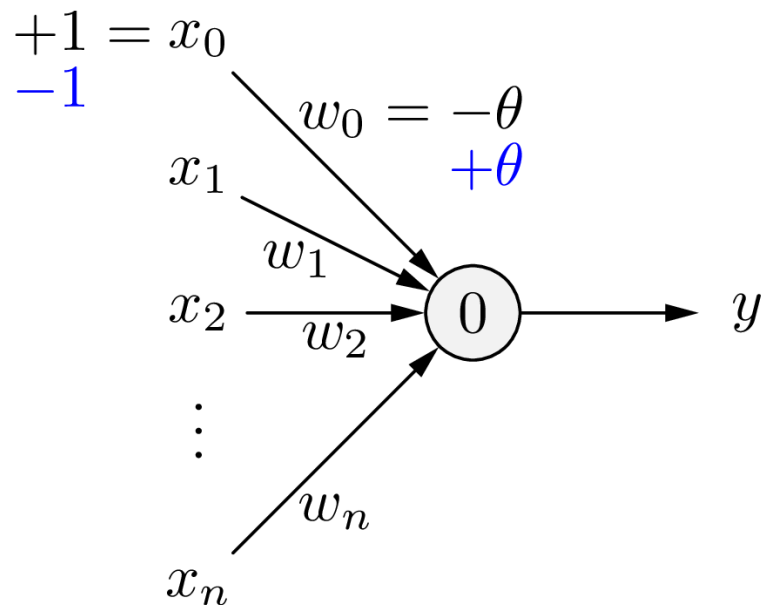
then online as well as batch training terminate.

- The algorithms terminate only when the error vanishes.
- Therefore the resulting threshold and weights must solve the problem.
- For not linearly separable problems the algorithms do not terminate (oscillation, repeated computation of same non-solving $\vec{w}$ and θ).

Training Threshold Logic Units: Delta Rule

Turning the threshold value into a weight:





$$\sum_{i=1}^{n} w_i x_i \geq \theta$$

$$\sum_{i=1}^{n} w_i x_i - \theta \geq 0$$

Training Threshold Logic Units: Delta Rule

Formal Training Rule (with threshold turned into a weight):
Let $\vec{x} = (x_0 = 1, x_1, \ldots, x_n)^\top$ be an (extended) input vector of a threshold logic unit, o the desired output for this input vector and y the actual output of the threshold logic unit. If $y \neq o$, then the (extended) weight vector $\vec{w} = (w_0 = -\theta, w_1, \ldots, w_n)^\top$ is adapted as follows in order to reduce the error:

$$\forall i \in \{0, \ldots, n\}: \quad w_i^{(\text{new})} = w_i^{(\text{old})} + \Delta w_i \qquad \text{with} \qquad \Delta w_i = \eta(o - y)x_i,$$

where η is a parameter that is called **learning rate**. It determines the severity of the weight changes. This procedure is called **Delta Rule** or **Widrow–Hoff Procedure** [Widrow and Hoff 1960].

- Note that with extended input and weight vectors, there is only one update rule (no distinction of threshold and weights).
- Note also that the (extended) input vector may be $\vec{x} = (x_0 = -1, x_1, \ldots, x_n)^\top$ and the corresponding (extended) weight vector $\vec{w} = (w_0 = +\theta, w_1, \ldots, w_n)^\top$.

Training Networks of Threshold Logic Units

- **Single threshold logic units** have strong limitations:
  They **can only compute linearly separable functions.**
- **Networks of threshold logic units**
  can compute arbitrary Boolean functions.
- **Training single threshold logic units** with the delta rule **is easy and fast**
  and guaranteed to find a solution if one exists.
- **Networks of threshold logic units cannot be trained**, because
  - there are no desired values for the neurons of the first layer(s),
  - the problem can usually be solved with several different functions computed by the neurons of the first layer(s) (non-unique solution).
- When this situation became clear,
  neural networks were first seen as a "research dead end".

Only the output neuron can be updated using the desired output.

One network might learn one way to split the input space. Another network might learn a completely different but equally valid way. Both networks would still correctly compute the final output.