Introduction to Neural Networks Computing

CMSC491N/691N, Spring 2001

Notations

units: X_i, Y_j

activation/output: x_i, y_j

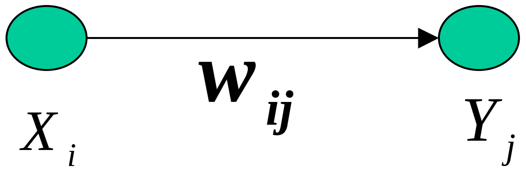
if X_i is an input unit, x_i = input signal

for other units Y_j, $y_j = f(y\_in_j)$

where f(.) is the activation function for Y_j

weights: w_{ij}

from unit i to unit j (other books use w_{ji})



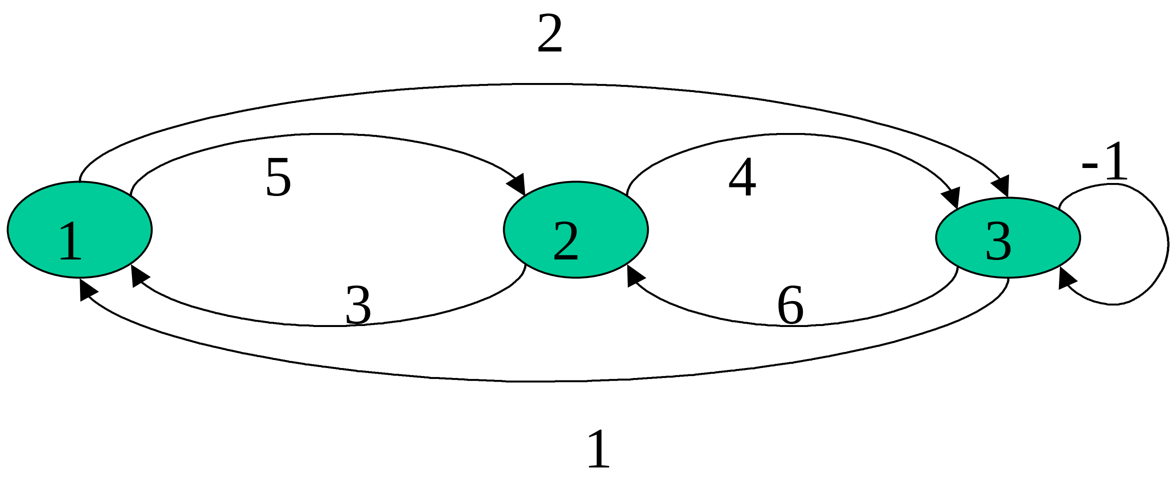
bias: $\boldsymbol{b}_j$ (a constant input)

threshold: q_j (for units with step/threshold

activation function)

weight matrix: W={ $\boldsymbol{w}_{ij}$ }

i: row index; j: column index



$$\begin{array}{cccl} \left\{\begin{matrix} 0 & 5 & 2 \\ 3 & 0 & 4 \\ 1 & 6 & -1 \end{matrix}\right\} & & & \begin{matrix} (w_{1\bullet}) \text{ row vectors} \\ (w_{2\bullet}) \\ (w_{3\bullet}) \end{matrix} \\ w_{\bullet 1} \quad w_{\bullet 2} \quad w_{\bullet 3} & & & \text{column vectors} \end{array}$$

vectors of weights:

$$\boldsymbol{w}_{\bullet j} = (\boldsymbol{w}_{1j}, \ \boldsymbol{w}_{2j}, \ \cdots \boldsymbol{w}_{3j})$$ weights come into unit j

$$\boldsymbol{w}_{i\bullet} = (\boldsymbol{w}_{i1}, \ \boldsymbol{w}_{i2}, \ \cdots \boldsymbol{w}_{i3})$$ weights go out of unit i

$\Delta \boldsymbol{w}_{ij} = \boldsymbol{w}_{ij}(\boldsymbol{new}) - \boldsymbol{w}_{ij}(\boldsymbol{old})$ learning/t raining

$\Delta \boldsymbol{W} = \{\Delta \boldsymbol{w}_{ij}\}$

$\boldsymbol{s} = (\boldsymbol{s}_{1,}\ \boldsymbol{s}_2 \ldots\ldots.\ \boldsymbol{s}_n)$ training input vector

$\boldsymbol{t} = (\boldsymbol{t}, \boldsymbol{t}_2 \ldots\ldots\ \boldsymbol{t}_m)$ training (or target)out put vector

$\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2 \ldots\ldots\ \boldsymbol{x}_n)$ input vector(for computatio n)

a : learning rate

a specifies the scale of $\Delta \boldsymbol{w}_{ij,}$ usually small

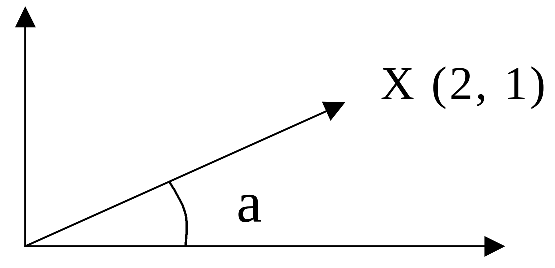
Review of Matrix Operations

Vector: a sequence of elements (the order is important)

e.g., x=(2, 1) denotes a vector

length = sqrt(2*2+1*1)

orientation angle = a



x=(x1, x2, ……, xn), an n dimensional vector

a point on an n dimensional space

column vector:

$$x = \begin{pmatrix} 1 \\ 2 \\ 5 \\ 8 \end{pmatrix}$$

row vector

$$y = (\,1\ 2\ 5\ 8\,) = x^T$$

transpose

$$(x^T)^T = x$$

norms of a vector: (magnitude)

$$\begin{array}{ll} L_1 \text{ norm} & \|x\|_1 = \sum_{i=1}^{n} |x_i| \\ L_2 \text{ norm} & \|x\|_2 = (\sum_{i=1}^{n} x_i^2)^{1/2} \\ L_\infty \text{ norm} & \|x\|_\infty = \max_{1 \le i \le n} |x_i| \end{array}$$

vector operations:

$rx = (rx_1, rx_2, \ldots\ldots rx_n)^T \quad r$: a scaler, x : a column vector

inner (dot) product

x, y are column vectors of same dimension n

$$x^T \bullet y = (x_1, x_2 \ldots\ldots x_n)\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \sum_{i=1}^{n} x_i y_i = (y_1, y_2 \ldots y_n)\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = y^T \bullet x$$

Cross product: $x \times y$

defines another vector orthogonal to the plan formed by x and y.

$$A_{m \times n} = \begin{pmatrix} a_{11} & a_{12} & \cdots\cdots & a_{1n} \\ & \vdots & & \\ a_{m1} & a_{m2} & \cdots\cdots & a_{mn} \end{pmatrix} = \{ a_{ij} \}_{m \times n}$$

a_{ij} : the element on the ith row and jth column

a_{ii} : a diagonal element

w_{ij} : a weight in a weight matrix W

each row or column is a vector

$a_{\bullet j}$: jth column vector

$a_{i \bullet}$: ith row vector

$$A_{m \times n} = (a_{\bullet 1} \ \cdots\cdots \ a_{\bullet n}) = \begin{pmatrix} a_{1\bullet} \\ \vdots \\ a_{m\bullet} \end{pmatrix}$$

a column vector of dimension m is a matrix of mx1

transpose:

$$A_{m\times n}^{T} = \begin{pmatrix} a_{11} & a_{21} & \ldots\ldots & a_{m1} \\ & & & \\ & & & \\ a_{1n} & a_{2n} & \ldots\ldots & a_{mn} \end{pmatrix}$$

jth column becomes jth row

square matrix: $A_{n\times n}$

identity matrix:

$$I = \begin{pmatrix} 1\,0\,\ldots\ldots\,0 \\ 0\,1\ldots\ldots\,0 \\ \\ 0\,0\ldots\ldots\,1 \end{pmatrix} \qquad a_{ij} = \begin{cases} 1 & \mathit{if}\ \ i = j \\ 0 & \mathit{otherwise} \end{cases}$$

symmetric matrix: $\boldsymbol{m = n}$

$$\boldsymbol{A = A^T, or\ \forall i\ a_{\bullet i} = a_{i\bullet},\ or\ \forall ij\ a_{ij} = a_{ji}}$$

matrix operations:

$$rA = (ra_{\bullet 1}, \ldots\ldots ra_{\bullet n}) = (ra_{ij})$$

$$\boldsymbol{x}^T \boldsymbol{A}_{m \times n} = (\boldsymbol{x}_1 \ldots\ldots \boldsymbol{x}_m)(\boldsymbol{a}_{\bullet 1}, \ldots\ldots \boldsymbol{a}_{\bullet n})$$
$$= (\boldsymbol{x}^T \boldsymbol{a}_{\bullet 1}, \ldots\ldots \boldsymbol{x}^T \boldsymbol{a}_{\bullet n})$$

The result is a row vector, each element of which is an inner product of $\boldsymbol{x}^T$ and a column vector $\boldsymbol{a}_{\bullet j}$

product of two matrices:

$$\boldsymbol{A}_{m\times n}\times \boldsymbol{B}_{n\times p}=\boldsymbol{C}_{m\times p}\quad \textbf{\textit{where}}\ \ \boldsymbol{C}_{ij}=\boldsymbol{a}_{i\bullet}\bullet \boldsymbol{b}_{\bullet j}$$

$$\boldsymbol{A}_{m\times n}\times \boldsymbol{I}_{n\times n}=\boldsymbol{A}_{m\times n}$$

vector outer product:

$$\boldsymbol{x}\cdot \boldsymbol{y}^{T}=\begin{pmatrix} \boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{x}_i \\ \vdots \\ \boldsymbol{x}_m \end{pmatrix}(\boldsymbol{y}_1 \ldots\ldots \boldsymbol{y}_n)=\begin{pmatrix} \boldsymbol{x}_1\boldsymbol{y}_1,\ \boldsymbol{x}_1\boldsymbol{y}_2,\ldots\ldots\ \boldsymbol{x}_1\boldsymbol{y}_n \\ \vdots \\ \boldsymbol{x}_m\boldsymbol{y}_1,\ \boldsymbol{x}_m\boldsymbol{y}_2,\ldots\ldots\ \boldsymbol{x}_m\boldsymbol{y}_n \end{pmatrix}$$

Calculus and Differential Equations

- $\dot{x}_i(t)$, the derivative of x_i, with respect to time t
- System of differential equations

$$\begin{cases} \dot{x}_1(t) = f_1(t) \\ \quad\vdots \\ \dot{x}_n(t) = f_n(t) \end{cases}$$

solution: $(x_1(t), \cdots\ x_n(t))$

difficult to solve unless $f_i(t)$ are simple

- Multi-variable calculus: $y(t) = f(x_1(t), x_2(t), \ldots\ldots x_n(t))$

  partial derivative: gives the direction and speed of change of *y*, with respect to x_i

$$y = \sin(x_1) + x_2^2 + e^{-(x_1 + x_2 + x_3)}$$

$$\frac{\partial y}{\partial x_1} = \cos(x_1) - e^{-(x_1 + x_2 + x_3)}$$

$$\frac{\partial y}{\partial x_2} = 2x_2 - e^{-(x_1 + x_2 + x_3)}$$

$$\frac{\partial y}{\partial x_3} = -e^{-(x_1 + x_2 + x_3)}$$

the total derivative: $y(t) = f(x_1(t), x_2(t), \ldots\ldots x_n(t))$

$$\dot{y}(t) = \frac{df}{dt} = \frac{\partial f}{\partial x_1}\dot{x}_1(t) + \ldots\ldots \frac{\partial f}{\partial x_n}\dot{x}_n(t)$$

$$= \nabla f \bullet (\dot{x}_1(t) \ldots\ldots \dot{x}_n(t))^T$$

Gradient of f **:** $\nabla f = (\frac{\partial f}{\partial x_1}, \ldots\ldots \frac{\partial f}{\partial x_n})$

Chain-rule: y is a function of x_i, x_i is a function of t

dynamic system:

$$\begin{cases} \dot{x}_1(t) = f_1(x_1, \ldots x_n) \\ \vdots \\ \dot{x}_n(t) = f_n(x_1, \ldots x_n) \end{cases}$$

- change of x_i may potentially affect other x
- all x_i continue to change (the system evolves)
- reaches equilibrium when $\dot{x}_i = 0 \ \forall i$
- stability/attraction: special equilibrium point

  (minimal energy state)
- pattern of $(x_1, \ldots x_n)$ at a stable state often represents a solution

Chapter 2: Simple Neural Networks for Pattern Classification

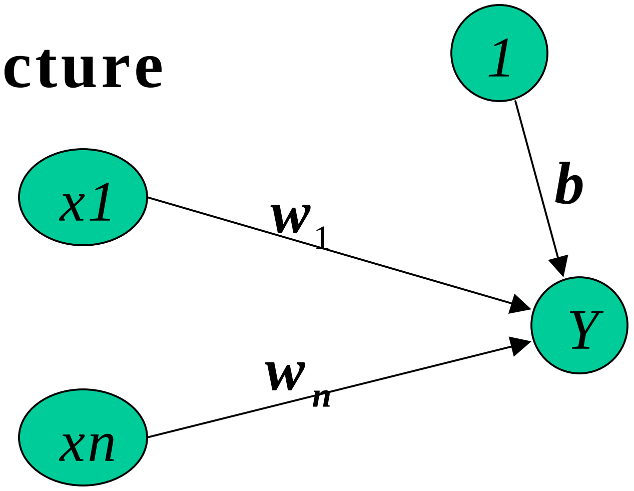
- General discussion
- Linear separability
- Hebb nets
- Perceptron
- Adaline

General discussion

- **Pattern recognition**
  - Patterns: images, personal records, driving habits, etc.
  - Represented as a vector of features (encoded as integers or real numbers in NN)
  - Pattern classification:
    - Classify a pattern to one of the given classes
    - Form pattern classes
  - Pattern associative recall
    - Using a pattern to recall a related pattern
    - **Pattern completion**: using a partial pattern to recall the whole pattern
    - **Pattern recovery:** deals with noise, distortion, missing information

- **General architecture**

Single layer



net input to *Y*: $net = b + \sum_{i=1}^{n} x_i w_i$

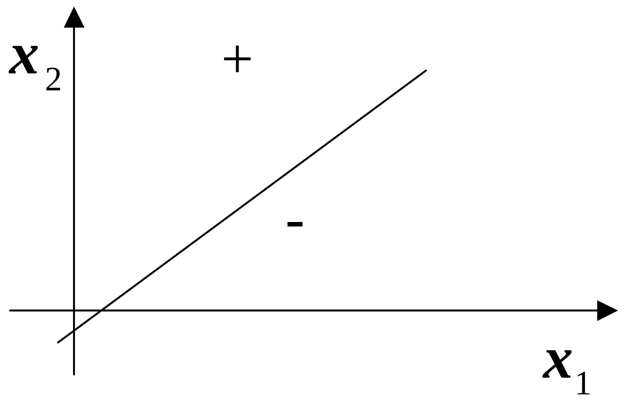
bias ***b*** is treated as the weight from a special unit with constant output 1.

threshold q related to *Y*

output $y = f(net) = \begin{cases} 1 & \text{if } net \geq q \\ -1 & \text{if } net < q \end{cases}$

classify $(x_1, \ldots\ldots x_n)$ into one of the two classes

- **Decision region/boundary**



n = 2, b != 0, θ = 0

$\boldsymbol{b} + \boldsymbol{x}_1\boldsymbol{w}_1 + \boldsymbol{x}_2\boldsymbol{w}_2 = 0$ or

$$\boldsymbol{x}_2 = -\frac{\boldsymbol{w}_1}{\boldsymbol{w}_2}\boldsymbol{x}_1 - \frac{\boldsymbol{b}}{\boldsymbol{w}_2}$$

is a line, called *decision boundary*, which partitions the plane into two decision regions

If a point/pattern $(\boldsymbol{x}_1, \boldsymbol{x}_2)$ is in the positive region, then

$\boldsymbol{b} + \boldsymbol{x}_1\boldsymbol{w}_1 + \boldsymbol{x}_2\boldsymbol{w}_2 \geq 0$, and the output is one (belongs to class one)

Otherwise, $\boldsymbol{b} + \boldsymbol{x}_1\boldsymbol{w}_1 + \boldsymbol{x}_2\boldsymbol{w}_2 < 0$, output –1 (belongs to class two)

n = 2, b = 0, θ != 0 would result a similar partition

– If n = 3 (three input units), then the decision boundary is a two dimensional plane in a three dimensional space

– In general, a decision boundary $\boldsymbol{b} + \sum_{\boldsymbol{i}=1}^{\boldsymbol{n}} \boldsymbol{x}_i \boldsymbol{w}_i = 0$ is a n-1 dimensional hyper-plane in an n dimensional space, which partition the space into two decision regions

– This simple network thus can classify a given pattern into one of the two classes, provided one of these two classes is entirely in one decision region (one side of the decision boundary) and the other class is in another region.

– The decision boundary is determined completely by the weights ***W*** and the bias ***b*** (or threshold *q*).

Linear Separability Problem

- If two classes of patterns can be separated by a decision boundary, represented by the linear equation

  $$\boldsymbol{b} + \sum_{\boldsymbol{i}=1}^{\boldsymbol{n}} \boldsymbol{x}_{\boldsymbol{i}} \boldsymbol{w}_{\boldsymbol{i}} = 0$$

  then they are said to be linearly separable. The simple network can correctly classify any patterns.
- Decision boundary (i.e., ***W*, *b*** or q) of linearly separable classes can be determined either by some learning procedures or by solving linear equation systems based on representative patterns of each classes
- If such a decision boundary does not exist, then the two classes are said to be linearly inseparable.
- Linearly inseparable problems cannot be solved by the simple network , more sophisticated architecture is needed.

- Examples of linearly separable classes

- Logical **AND** function

patterns (bipolar) decision boundary

x1	x2	y
-1	-1	-1
-1	1	-1
1	-1	-1
1	1	1

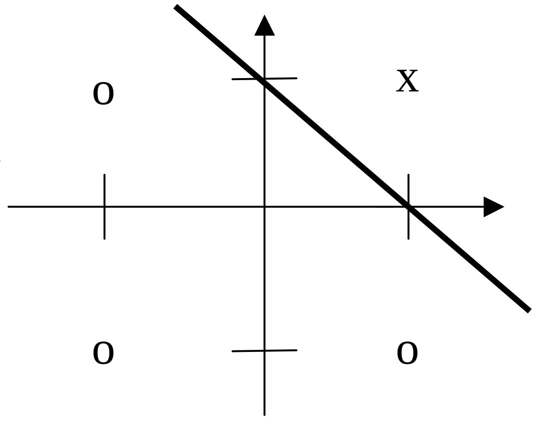
$w1 = 1$

$w2 = 1$

$b = -1$

$\theta = 0$

$-1 + x1 + x2 = 0$



x: class I (y = 1)
o: class II (y = -1)

- Logical **OR** function

patterns (bipolar) decision boundary

x1	x2	y
-1	-1	-1
-1	1	1
1	-1	1
1	1	1

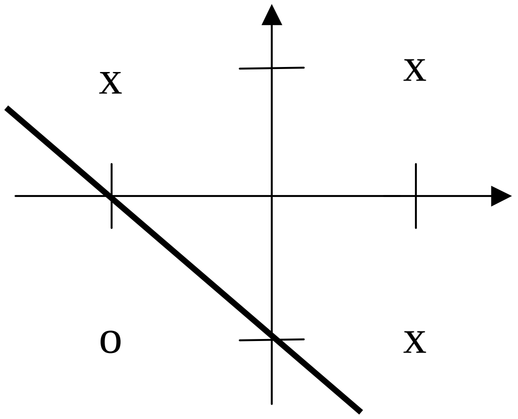
$w1 = 1$

$w2 = 1$

$b = 1$

$\theta = 0$

$1 + x1 + x2 = 0$



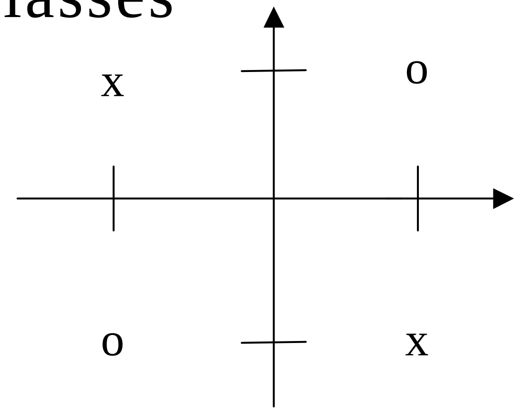
x: class I (y = 1)
o: class II (y = -1)

- Examples of linearly inseparable classes

  - Logical **XOR** (exclusive OR) function

    patterns (bipolar) decision boundary

x1	x2	y
-1	-1	-1
-1	1	1
1	-1	1
1	1	-1



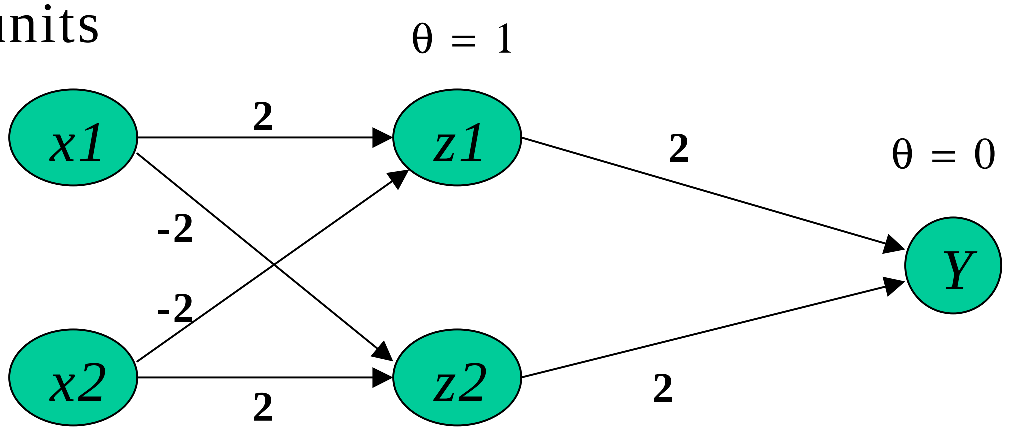
x: class I (y = 1)
o: class II (y = -1)

No line can separate these two classes, as can be seen from the fact that the following linear inequality system has no solution

$$\begin{cases} \boldsymbol{b} - \boldsymbol{w}_1 - \boldsymbol{w}_2 < 0 & (1) \\ \boldsymbol{b} - \boldsymbol{w}_1 + \boldsymbol{w}_2 \geq 0 & (2) \\ \boldsymbol{b} + \boldsymbol{w}_1 - \boldsymbol{w}_2 \geq 0 & (3) \\ \boldsymbol{b} + \boldsymbol{w}_1 + \boldsymbol{w}_2 < 0 & (4) \end{cases}$$

because we have b < 0 from (1) + (4), and b >= 0 from (2) + (3), which is a contradiction

– XOR can be solved by a more complex network with hidden units



(-1, -1)	(-1, -1)	-1
(-1, 1)	(-1, 1)	1
(1, -1)	(1, -1)	1
(1, 1)	(1, 1)	-1

Hebb Nets

- Hebb, in his influential book *The organization of Behavior* (1949), claimed
  - Behavior changes are primarily due to the changes of synaptic strengths (w_{ij}) between neurons I and j
  - w_{ij} increases only when both I and j are "on": the **Hebbian learning law**
  - In ANN, Hebbian law can be stated: w_{ij} increases only if the outputs of both units x_i and y_j have the same sign.
  - In our simple network (one output and n input units)

$$\Delta w_{ij} = w_{ij}(new) - w_{ij}(old) = x_i y$$

or, $\Delta w_{ij} = w_{ij}(new) - w_{ij}(old) = a \; x_i y$

- Hebb net (supervised) learning algorithm (p.49)

```
Step 0.  Initialization: b = 0, wi = 0, i = 1 to n
Step 1.  For each of the training sample s:t do steps 2 -4
             /* s is the input pattern, t the target output of the sample */
Step 2.      xi := si, I = 1 to n                 /* set s to input units */
Step 3.      y := t                               /* set y to the target */
Step 4.      wi := wi + xi * y, i = 1 to n        /* update weight */
             b := b + xi * y                      /* update bias */
```

Notes: 1) $\alpha = 1$, 2) each training sample is used only once.

- Examples: AND function
  - Binary units (1, 0)

(x1, x2, 1)	y=t	w1	w2	b
(1, 1, 1)	1	1	1	1
(1, 0, 1)	0	1	1	1
(0, 1, 1)	0	1	1	1
(0, 0, 1)	0	1	1	1

bias unit

An incorrect boundary:

$1 + x1 + x2 = 0$

Is learned after using each sample once

– Bipolar units (1, -1)

(x1, x2, 1)	y=t	w1	w2	b
(1, 1, 1)	1	1	1	1
(1, -1, 1)	-1	0	2	0
(-1, 1, 1)	-1	1	1	-1
(-1, -1, 1)	-1	2	2	-2

A correct boundary

$-1 + x1 + x2 = 0$

is successfully learned

– It will fail to learn x1 ^ x2 ^ x3, even though the function is linearly separable.

– Stronger learning methods are needed.
  - Error driven: for each sample s:t, compute y from s based on current W and b, then compare y and t
  - Use training samples repeatedly, and each time only change weights slightly ($\alpha << 1$)
  - Learning methods of Perceptron and Adaline are good examples

Perceptrons

- By Rosenblatt (1962)
  - For modeling visual perception (retina)
  - Three layers of units: ***S****ensory*, ***A****ssociation*, and ***R****esponse*
  - Learning occurs only on weights from ***A*** units to ***R*** units (weights from ***S*** units to ***A*** units are fixed).
  - A single ***R*** unit receives inputs from n ***A*** units (same architecture as our simple network)
  - For a given training sample s:t, change weights only if the computed output y is different from the target output t (thus error driven)

- Perceptron learning algorithm (p.62)

```
Step 0. Initialization: b = 0, wi = 0, i = 1 to n
Step 1. While stop condition is false do steps 2-5
Step 2.     For each of the training sample s:t do steps 3 -5
Step 3.         xi := si, i = 1 to n
Step 4.         compute y
Step 5.         If y != t
                    wi := wi + α * xi * t, i = 1 to n
                    b := b + α * t
```

Notes:

- Learning occurs only when a sample has y != t
- Two loops, a completion of the inner loop (each sample is used once) is called an epoch

Stop condition

- When no weight is changed in the current epoch, or
- When pre-determined number of epochs is reached

Informal justification: Consider y = 1 and t = -1

- To move y toward t, w1should reduce net_y
- If xi = 1, xi * t < 0, need to reduce w1 (xi*w1 is reduced)
- If xi = -1, xi * t >0 need to increase w1 (xi*w1 is reduced)

See book (pp. 62-68) for an example of execution

- Perceptron learning rule convergence theorem
  - Informal: any problem that can be represented by a perceptron can be learned by the learning rule
  - ***Theorem***: If there is a $\boldsymbol{W}^1$ such that $\boldsymbol{f}(\boldsymbol{x}(\boldsymbol{p}) \cdot \boldsymbol{W}^1) = \boldsymbol{t}(\boldsymbol{p})$ for all $\boldsymbol{P}$ training sample patterns $\{\boldsymbol{x}(\boldsymbol{p}), \boldsymbol{t}(\boldsymbol{p})\}$, then for any start weight vector $\boldsymbol{W}^0$, the perceptron learning rule will converge to a weight vector $\boldsymbol{W}^*$ such that

    $\boldsymbol{f}(\boldsymbol{x}(\boldsymbol{p}) \cdot \boldsymbol{W}^*) = \boldsymbol{t}(\boldsymbol{p})$ for all $\boldsymbol{p}$. ($\boldsymbol{W}^*$ and $\boldsymbol{W}^1$ may not be the same.)
  - Proof: reading for grad students (pp. 77-79

Adaline

- By Widrow and Hoff (1960)
  - **Ada**ptive **Line**ar Neuron for signal processing
  - The same architecture of our simple network
  - Learning method: **delta rule** (another way of error driven), also called Widrow-Hoff learning rule
  - The delta: ***t – y_in***
    - NOT ***t – y*** because ***y = f(y_in)*** is not differentiable
  - Learning algorithm: same as Perceptron learning except in Step 5:

    b := b + *a* * (***t – y_in***)

    wi := wi + *a* * ***xi **** (***t – y_in***)

- **Derivation of the delta rule**
  - Error for all P samples: mean square error
    $$E = \frac{1}{P}\sum_{p=1}^{P}(t(p) - y\_in(p))^2$$
    - E is a function of W = {w1, ... wn}
  - Learning takes **gradient descent** approach to reduce E by modify W
    - the gradient of E: $\nabla E = (\frac{\partial E}{\partial w_1}, ... \frac{\partial E}{\partial w_n})$
    - $\Delta w_i \propto -\frac{\partial E}{\partial w_i}$
    - $\frac{\partial E}{\partial w_i} = [\frac{2}{P}\sum_{p=1}^{P}(t(p) - y\_in(p))]\frac{\partial}{\partial w_i}(t(p) - y\_in(p)$
      $$= -[\frac{2}{P}\sum_{p=1}^{P}(t(p) - y\_in(p))]x_i$$
    - There for $\Delta w_i \propto -\frac{\partial E}{\partial w_i} = [\frac{2}{P}\sum_{1}^{P}(t(p) - y\_in(p))]x_i$

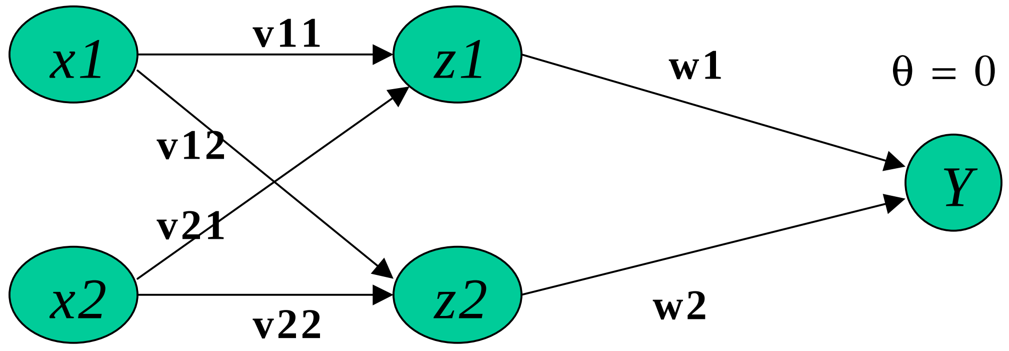
- **How to apply the delta rule**
  - **Method 1 (sequential mode):** change wi after each training pattern by $a\,(\boldsymbol{t}(\boldsymbol{p}) - \boldsymbol{y\_in}(\boldsymbol{p}))\,\boldsymbol{x}_i$
  - **Method 2 (batch mode):** change wi at the end of each epoch. Within an epoch, cumulate $a\,(\boldsymbol{t}(\boldsymbol{p}) - \boldsymbol{y\_in}(\boldsymbol{p}))\,\boldsymbol{x}_i$ for every pattern **(x(p), t(p))**
  - Method 2 is slower but may provide slightly better results (because Method 1 may be sensitive to the sample ordering)
- **Notes:**
  - E monotonically decreases until the system reaches a state with (local) minimum E (a small change of any wi will cause E to increase).
  - At a local minimum E state, $\partial \boldsymbol{E} / \partial \boldsymbol{w}_i = 0 \;\; \forall \boldsymbol{i}$, but E is not guaranteed to be zero

Summary of these simple networks

- Single layer nets have limited representation power (linear separability problem)
- Error drive seems a good way to train a net
- Multi-layer nets (or nets with non-linear hidden units) may overcome linear inseparability problem, learning methods for such nets are needed
- Threshold/step output functions hinders the effort to develop learning methods for multi-layered nets

Why hidden units must be non-linear?

- Multi-layer net with linear hidden layers is equivalent to a single layer net



- Because z1 and z2 are linear unit

  z1 = a1* (x1*v11 + x2*v21) + b1

  z1 = a2* (x1*v12 + x2*v22) + b2

- y_in = z1*w1 + z2*w2

  = x1*u1 + x2*u2 + b1+b2 where

  u1 = (a1*v11+ a2*v12)w1, u2 = (a1*v21 + a2*v22)*w2

  y_in is still a linear combination of x1 and x2.