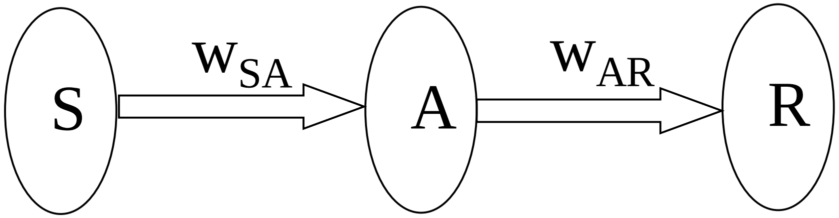
Chapter 2
Single Layer Feedforward Networks

Perceptrons

- By Rosenblatt (1962)
  - For modeling visual perception (retina)
  - A feedforward network of 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).
  - Each ***R*** unit receives inputs from n ***A*** units
  - For a given training sample s:t, change weights between ***A*** and ***R*** only if the computed output y is different from the target output t (error driven)



Perceptrons

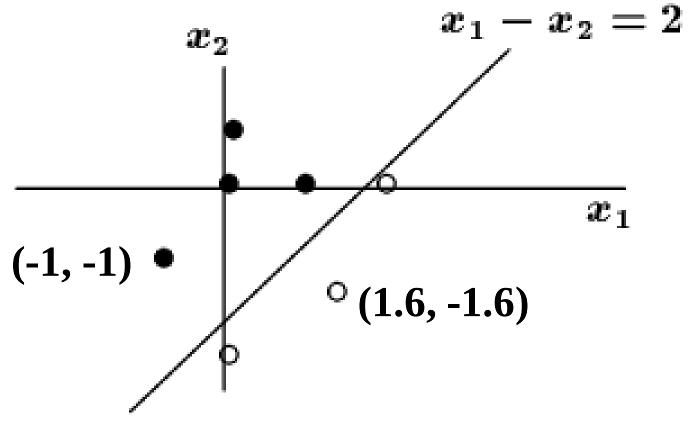
- A simple perceptron
  - Structure:
    - Sing output node with threshold function
    - n input nodes with weights w_i, $i = 1 - n$
  - To classify input patterns into one of the two classes (depending on whether output = 0 or 1)
  - Example: input patterns: (x_1, x_2)
    - **Two groups of input patterns**

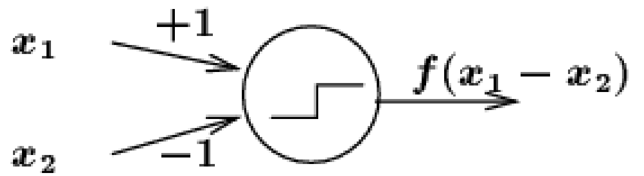
      (0, 0) (0, 1) (1, 0) (-1, -1);

      (2.1, 0) (0, -2.5) (1.6, -1.6)
    - Can be separated by a line on the (x_1, x_2) plane $\boldsymbol{x_1 - x_2 = 2}$
    - Classification by a perceptron with

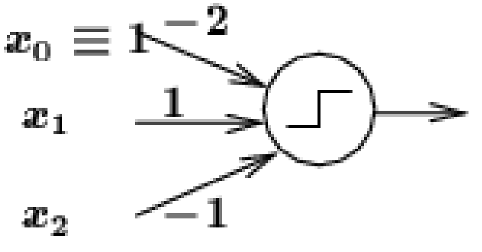
      $w_1 = 1, w_2 = -1, threshold = 2$

Perceptrons





- Implement threshold by a node x_0
  - Constant output 1
  - Weight w_0= - threshold
  - A common practice in NN design



Perceptrons

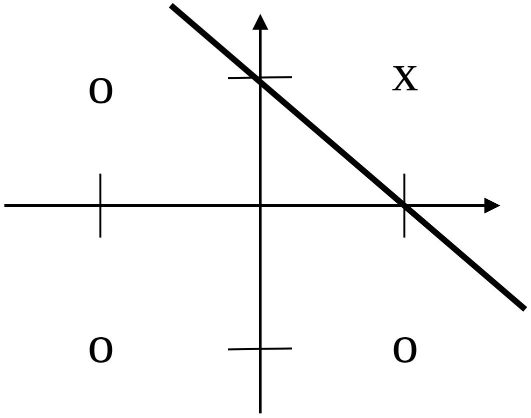
- Linear separability
  - A set of (2D) patterns (x_1, x_2) of two classes is linearly separable if there exists a line on the (x_1, x_2) plane
    - $w_0 + w_1 x_1 + w_2 x_2 = 0$
    - Separates all patterns of one class from the other class
  - A perceptron can be built with
    - 3 input $x_0 = 1$, x_1, x_2 with weights w_0, w_1, w_2
  - n dimensional patterns (x_1,…, x_n)
    - Hyperplane $w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n = 0$ dividing the space into two regions
  - Can we get the weights from a set of sample patterns?
    - If the problem is linearly separable, then YES (by perceptron learning)

- Examples of linearly separable classes

- Logical **AND** function

patterns (bipolar) decision boundary

x1	x2	output	w1 = 1
-1	-1	-1	w2 = 1
-1	1	-1	w0 = -1
1	-1	-1	
1	1	1	-1 + x1 + x2 = 0

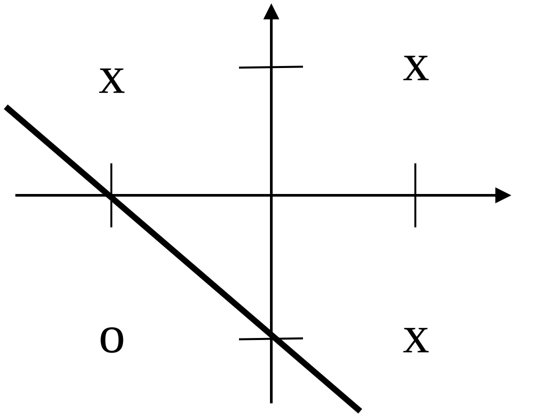


x: class I (output = 1)
o: class II (output = -1)

- Logical **OR** function

patterns (bipolar) decision boundary

x1	x2	output	w1 = 1
-1	-1	-1	w2 = 1
-1	1	1	w0 = 1
1	-1	1	
1	1	1	1 + x1 + x2 = 0



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

Perceptron Learning

- The network
  - Input vector $\boldsymbol{i_j}$ (including threshold input = 1)
  - Weight vector $\boldsymbol{w} = (w_0, w_1, \ldots, w_n)$ $\quad net = w \cdot i_j = \sum_{k=0}^{n} w_k i_{k,j}$
  - Output: bipolar (-1, 1) using the sign node function

$$output = \begin{cases} 1 & \text{if } w \cdot i_j > 0 \\ -1 & \text{otherwise} \end{cases}$$

- Training samples
  - Pairs ($\boldsymbol{i_j}$, *class*($\boldsymbol{i_j}$)) where *class*($\boldsymbol{i_j}$) is the correct classification of $\boldsymbol{i_j}$
- Training:
  - Update $\boldsymbol{w}$ so that all sample inputs are correctly classified (if possible)
  - If an input $\boldsymbol{i_j}$ is misclassified by the current $\boldsymbol{w}$

    $class(\boldsymbol{i_j}) \cdot \boldsymbol{w} \cdot \boldsymbol{i_j} < 0$

    change $\boldsymbol{w}$ to $\boldsymbol{w} + \Delta\boldsymbol{w}$ so that $(\boldsymbol{w} + \Delta\boldsymbol{w}) \cdot \boldsymbol{i_j}$ is closer to *class*($\boldsymbol{i_j}$)

Perceptron Learning

Perceptron Training Algorithm

Algorithm Perceptron:

Start with a randomly chosen weight vector w_0;

Let $k = 1$;

while some input vectors remain misclassified, do

Let i_j be a misclassified input vector;

Let $x_k = class(i_j).i_j$, implying that $w_{k-1} \cdot x_k < 0$;

Update the weight vector to $w_k = w_{k-1} + \eta x_k$;

Increment k;

end-while;

Where $\eta > 0$ is the learning rate

Perceptron Learning

- Justification

$$
\begin{aligned}
(w+\eta\cdot x_k)\cdot i_j &= (w+\eta\cdot class(i_j)\cdot i_j)\cdot i_j \\
&= w\cdot i_j + \eta\cdot class(i_j)\cdot i_j\cdot i_j
\end{aligned}
$$

$$\text{since}\, i_j\cdot i > 0$$

$$(w+\eta\cdot x_k)\cdot i_j - w\cdot i_j = \eta\cdot class(i_j)\cdot i_j\cdot i_j$$

$$\begin{cases} >0 & \text{if } class(i_j)=1 \\ <0 & \text{if } class(i_j)=-1 \end{cases}$$

$$\Rightarrow \text{new}\, net\, \text{moves toward}\ \ class(i_j)$$

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

    $f(i_p \cdot w^*) = class(i_p)$

    (w^* and w^1 may not be the same.)
  - Proof: reading for grad students (Sec. 2.4)

Perceptron Learning

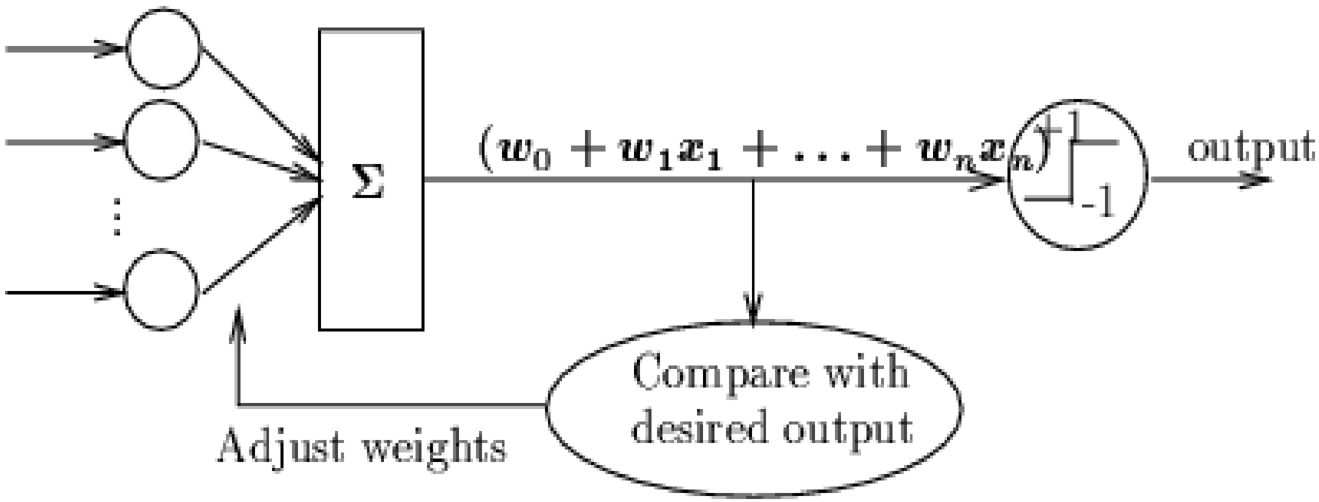
- Note:
  - It is a supervised learning ($class(\boldsymbol{i_j})$ is given for all sample input $\boldsymbol{i_j}$)
  - Learning occurs only when a sample input misclassified (error driven)
- Termination criteria: learning stops when all samples are correctly classified
  - Assuming the problem is linearly separable
  - Assuming the learning rate (η) is sufficiently small
- Choice of learning rate:
  - If η is too large: existing weights are overtaken by $\Delta w = \eta \cdot class(i_j) \cdot i_j$
  - If η is too small (≈ 0): very slow to converge
  - Common choice: η = 1.
- Non-numeric input:
  - Different encoding schema
    ex. Color = (red, blue, green, yellow). (0, 0, 1, 0) encodes “green”

Perceptron Learning

- Learning quality
  - *Generalization*: can a trained perceptron correctly classify patterns not included in the training samples?
    - Common problem for many NN learning models
  - Depends on the quality of training samples selected.
  - Also to some extent depends on the learning rate and initial weights
  - How can we know the learning is ok?
    - Reserve a few samples for testing

Adaline

- By Widrow and Hoff (~1960)
  – **Ada**ptive **lin**ear **e**lements for signal processing
  – The same architecture of perceptrons



  – Learning method: **delta rule** (another way of error driven), also called Widrow-Hoff learning rule

    Try to reduce the mean squared error (MSE) between the net input and the desired out put

Adaline

- Delta rule
  - Let $i_j = (i_{0,j}, i_{1,j}, \ldots, i_{n,j})$ be an input vector with desired output d_j
  - The squared error
    - $E = (d_j - net_j)^2 = (d_j - \sum_l w_l i_{l,j})^2$
    - Its value determined by the weights w_l
  - Modify weights by gradient descent approach
    - $$\begin{aligned} \frac{\partial E}{\partial w_k} &= 2(d_j - \mathrm{net}_j)\frac{\partial}{\partial w_k}(\text{-}\mathrm{net}_j) \\ &= -2(d_j - \mathrm{net}_j)\ i_{k,j}. \end{aligned}$$
    - Change weights in the opposite direction of $\partial E / \partial w_k$

$$\Delta w_k = \eta(d_j - \sum_l w_l i_{l,j}) \cdot i_{k,j} = \eta(d_j - net_j) \cdot i_{k,j}$$

Adaline Learning Algorithm

Algorithm LMS-Adaline;

Start with a randomly chosen weight vector w_0;

Let $k = 1$;

while MSE is unsatisfactory and

computational bounds are not exceeded, do

Let i be an input vector

(chosen randomly or in some sequence)

for which d is the desired output value;

Update the weight vector to

$$w_k = w_{k-1} + \eta(d - w_{k-1} \cdot i)i$$

Increment k;

end-while.

Adaline Learning

- Delta rule in batch mode
  - Based on ***mean*** squared error over all ***P*** samples

$$E = \frac{1}{P}\sum_{p=1}^{P}(d_p - net_p)^2$$

- E is again a function of $w = (w_0, w_1, \ldots, w_n)$
- the gradient of E:

$$\frac{\partial E}{\partial w_k} = \frac{2}{P}\sum_{p=1}^{P}[(d_p - net_p)\frac{\partial}{\partial w_k}(d_p - net_p)]$$

$$= -\frac{2}{P}\sum_{p=1}^{P}[(d_p - net_p)\cdot i_{k,p}]$$

- Therefore $\Delta w_i = -\eta\frac{\partial E}{\partial w_i} = \eta\sum_{p=1}^{P}[(d_p - net_p)\cdot i_{k,p}]$

Adaline Learning

- Notes:
  - Weights will be changed even if an input is classified correctly
  - $\boldsymbol{E}$ monotonically decreases until the system reaches a state with (local) minimum $\boldsymbol{E}$ (a small change of any w_i will cause $\boldsymbol{E}$ to increase).
  - At a local minimum $\boldsymbol{E}$ state, $\partial \mathbf{E} / \partial \mathbf{w}_i = 0 \;\; \forall \boldsymbol{i}$, but $\boldsymbol{E}$ is not guaranteed to be zero (net_j != d_j)
    - This is why Adaline uses threshold function rather than linear function

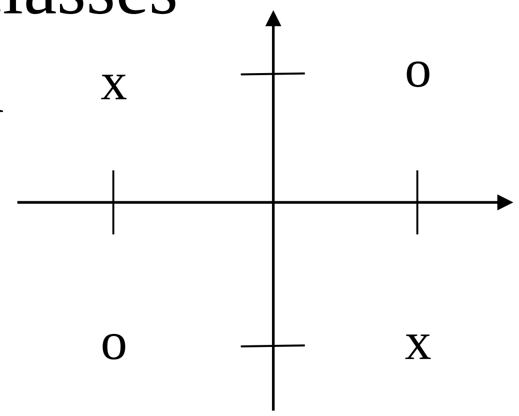
Linear Separability Again

- Examples of linearly inseparable classes

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

  patterns (bipolar) decision boundary

x1	x2	output
-1	-1	-1
-1	1	1
1	-1	1
1	1	-1



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

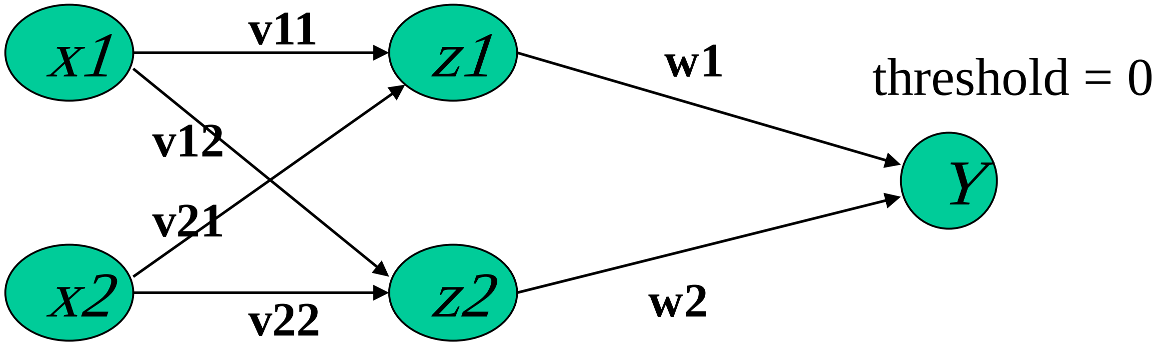
No line can separate these two classes, as can be seen from the fact that the following linear inequality system has no solution

$$\begin{cases} w_0 - w_1 - w_2 < 0 & (1) \\ w_0 - w_1 + w_2 \geq 0 & (2) \\ w_0 + w_1 - w_2 \geq 0 & (3) \\ w_0 + w_1 + w_2 < 0 & (4) \end{cases}$$

because we have $w_0 < 0$ from (1) + (4), and $w_0 >= 0$ from (2) + (3), which is a contradiction

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

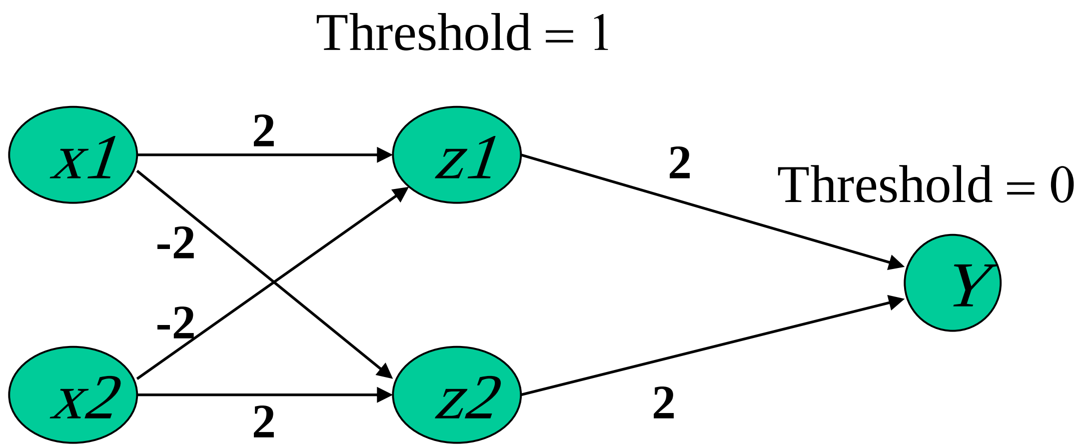
  - net_y = z1*w1 + z2*w2

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

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

    net_y is still a linear combination of x1 and x2.

– 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

Summary

- Single layer nets have limited representation power (linear separability problem)
- Error driven 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