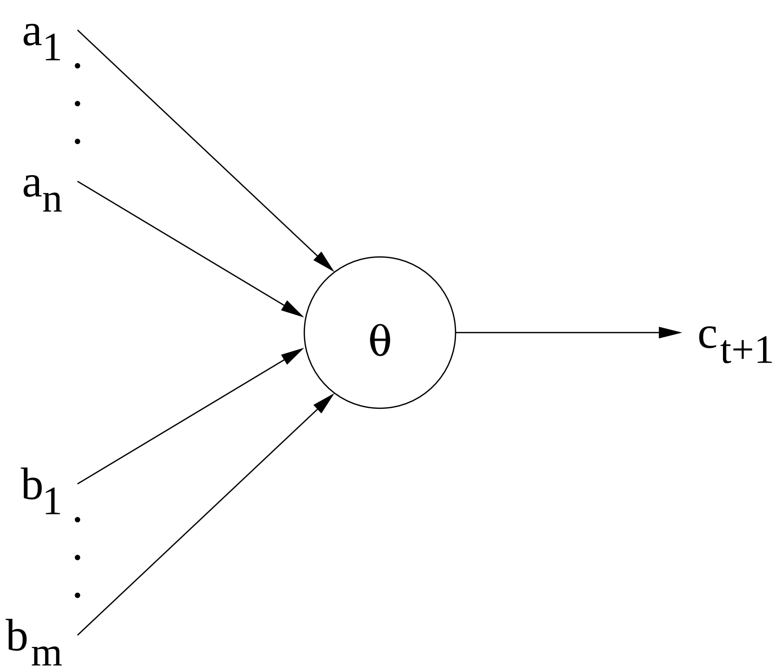
THE PERCEPTRON

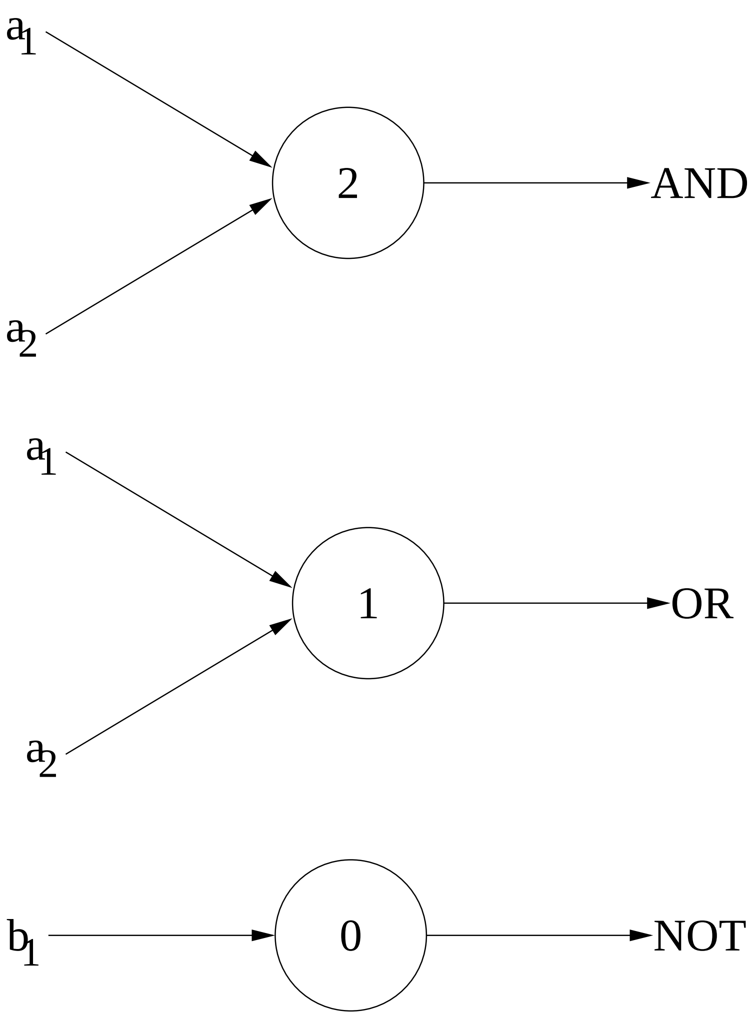
The McCulloch-Pitts Neuron

- The first mathematical model of a neuron [Warren McCulloch and Walter Pitts, 1943]
- Binary activation: *fires* (1) or *not fires* (0)
- Excitatory inputs: the a's, and
  Inhibitory inputs: the b's
- Unit weights and fixed threshold θ
- Absolute inhibition

$$c_{t+1} = \begin{cases} 1 & \text{If } \sum_{i=0}^{n} a_{i,t} \geq \theta \text{ and } b_{1,t} = \cdots = b_{m,t} = 0 \\ 0 & \text{Otherwise} \end{cases}$$



Computing with McCulloch-Pitts Neurons



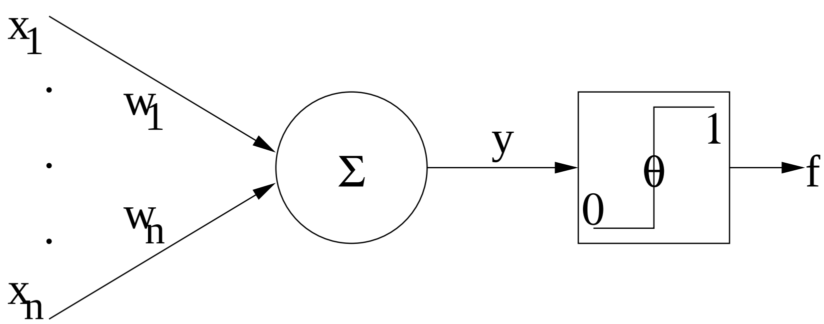
Any task or phenomenon that can be represented as a logic function can be modelled by a network of MP-neurons

- {OR, AND, NOT} is functionally complete
- Any Boolean function can be implemented using OR, AND and NOT
- Canonical forms: CSOP or CPOS forms
- MP-neurons $\Leftrightarrow$ Finite State Automata

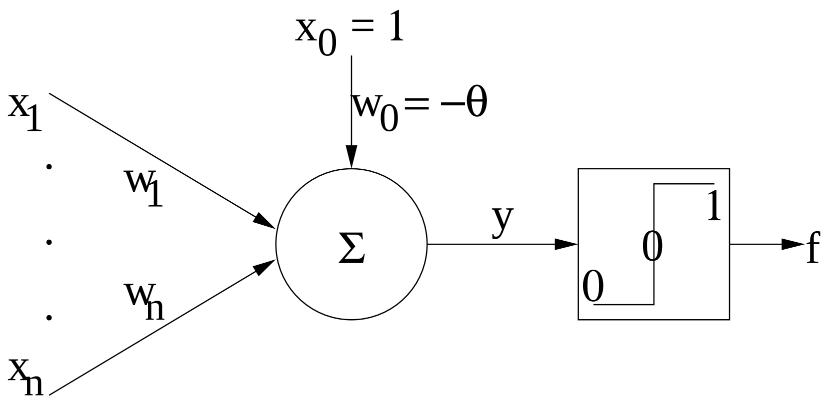
Limitation of MP-neurons and Solution

- Problems with MP-neurons
  - Weights and thresholds are analytically determined. Cannot learn
  - Very difficult to minimize size of a network
  - What about non-discrete and/or non-binary tasks?

- Perceptron solution [Rosenblatt, 1958]
  - Weights and thresholds can be determined analytically or by a learning algorithm
  - Continuous, bipolar and multiple-valued versions
  - Efficient minimization heuristics exist



Perceptron



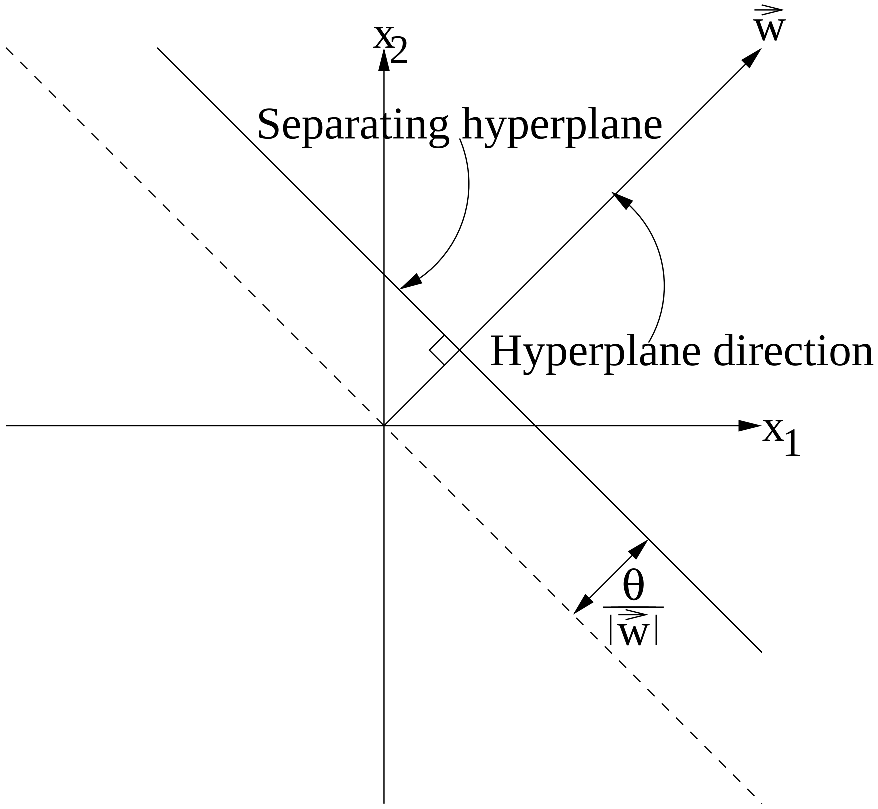
- Architecture
  - Input: $\vec{x} = (x_0 = 1, x_1, \ldots, x_n)$
  - Weight: $\vec{w} = (w_0 = -\theta, w_1, \ldots, w_n)$, θ = bias
  - Net input: $y = \vec{w}\vec{x} = \sum_{i=0}^{n} w_i x_i$
  - Output $f(\vec{x}) = g(\vec{w}\vec{x}) = \begin{cases} 0 & \text{If } \vec{w}\vec{x} < 0 \\ 1 & \text{If } \vec{w}\vec{x} \geq 0 \end{cases}$
- Pattern classification
- Supervised learning
- Error-correction learning

Perceptron Analysis

- Perceptron's *decision boundary*

$$w_1 x_1 + \cdots + w_n x_n = \theta$$

$$w_0 x_0 + w_1 x_1 + \cdots + w_n x_n = 0$$



- All points
  - below the hyperplane have value 0
  - on the hyperplane have the same value
  - above the hyperplane have value 1

Perceptron Analysis
(continued)

- *Linear Separability*
  - A problem (or task or set of examples) is linearly separable if there exists a hyperplane $w_0x_0 + w_1x_1 + \cdots + w_nx_n = 0$ that separates the examples into two *distinct* classes
  - Perceptron can only learn (compute) tasks that are linearly separable.
  - The weight vector $\vec{w}$ of the perceptron correspond to the coefficients of the separating line

- *Non-Linear Separability*
  - Limitations of the perceptron: many real-world problems are highly non-linear
  - Simpe Boolean functions:
    * XOR, EQUALITY, . . . etc.
    * Linear, parity, symmetric or . . . functions

Perceptron Learning Rule

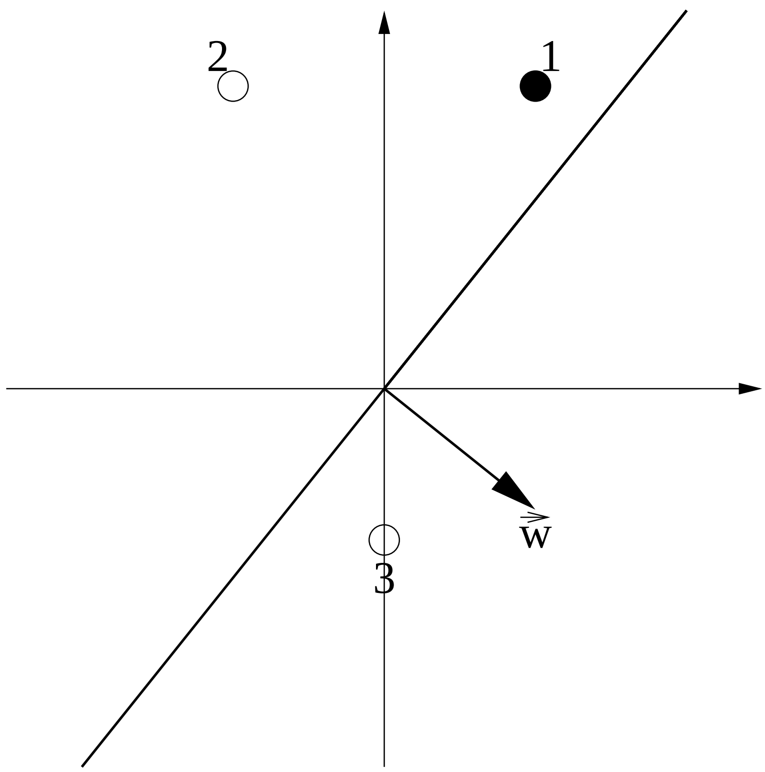
- Test problem
  - Let the set of training examples be

$$[\vec{x}_1 = (1, 2), d_1 = 1]$$

$$[\vec{x}_2 = (-1, 2), d_2 = 0]$$

$$[\vec{x}_3 = (0, -1), d_3 = 0]$$

  - The bias (or threshold) be $b = 0$
  - The initial weight vector be $\vec{w} = (1, 0.8)$



We want to obtain a learning algorithm that finds a weight vector $\vec{w}$ which will correctly classify (separate) the examples.

Perceptron Learning Rule
(continued)

- First input $\vec{x}_1$ is misclassified with positive error. What to do?

- Idea: move hyperplane to separating position

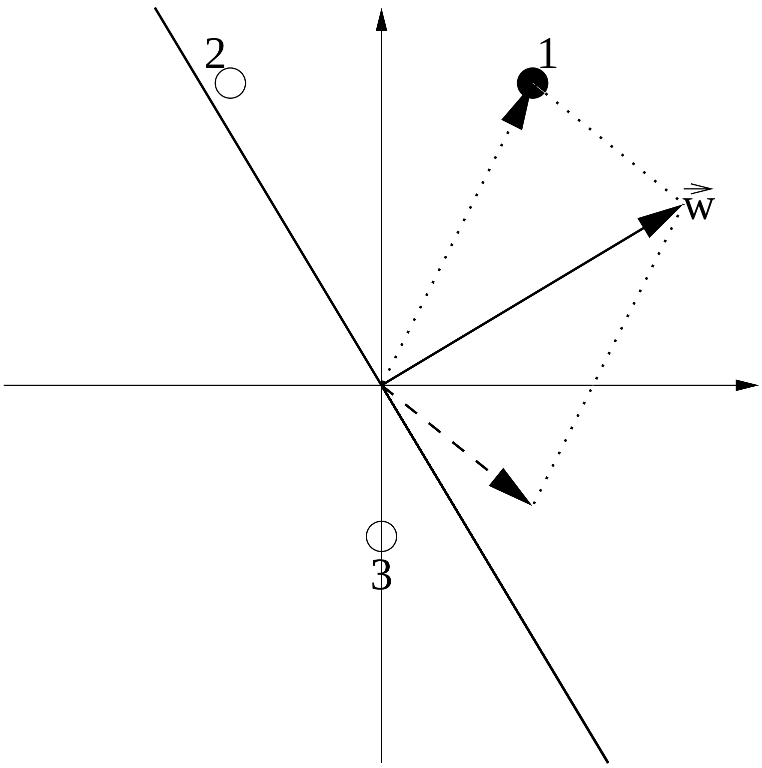
- Solution:

  - Move $\vec{w}$ closer to $\vec{x}_1$: add $\vec{x}_1$ to $\vec{w}$.

    * $\vec{w} = \vec{w} + \vec{x}_1$

  - First rule: *positive error rule*

$$\text{If } d = 1 \text{ and } a = 0 \text{ then } \vec{w}^{new} = \vec{w}^{old} + \vec{x}$$

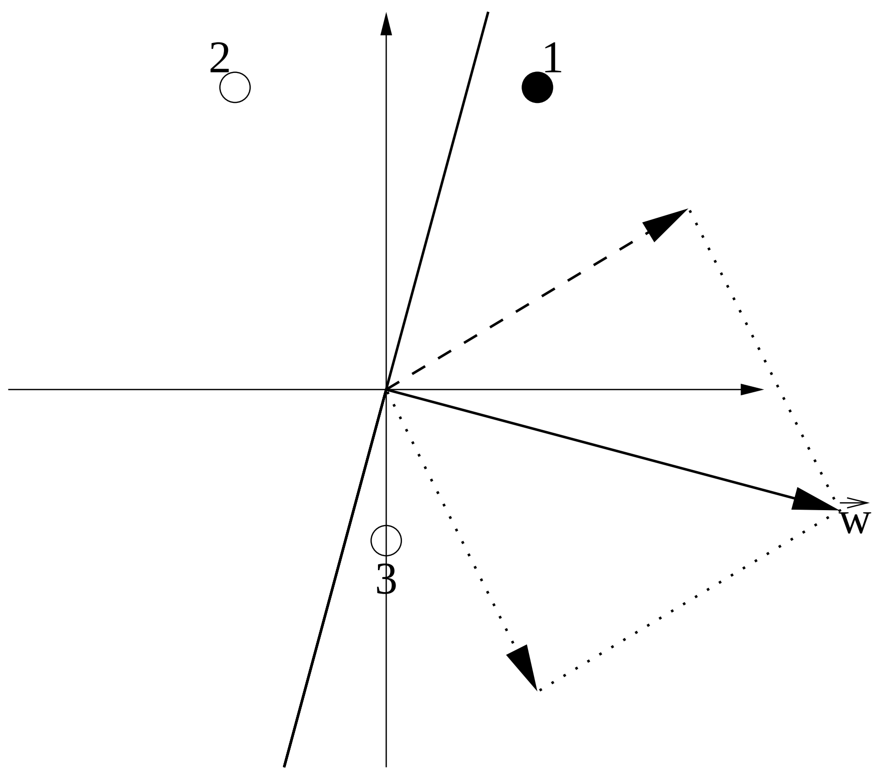


Perceptron Learning Rule
(continued)

- Second input $\vec{x}_2$ is misclassified with negative error

- Solution:
  - Move $\vec{w}$ away from $\vec{x}_2$: substract $\vec{x}_2$ from $\vec{w}$.
    - $\vec{w} = \vec{w} - \vec{x}_2$
  - Second rule: *negative error rule*

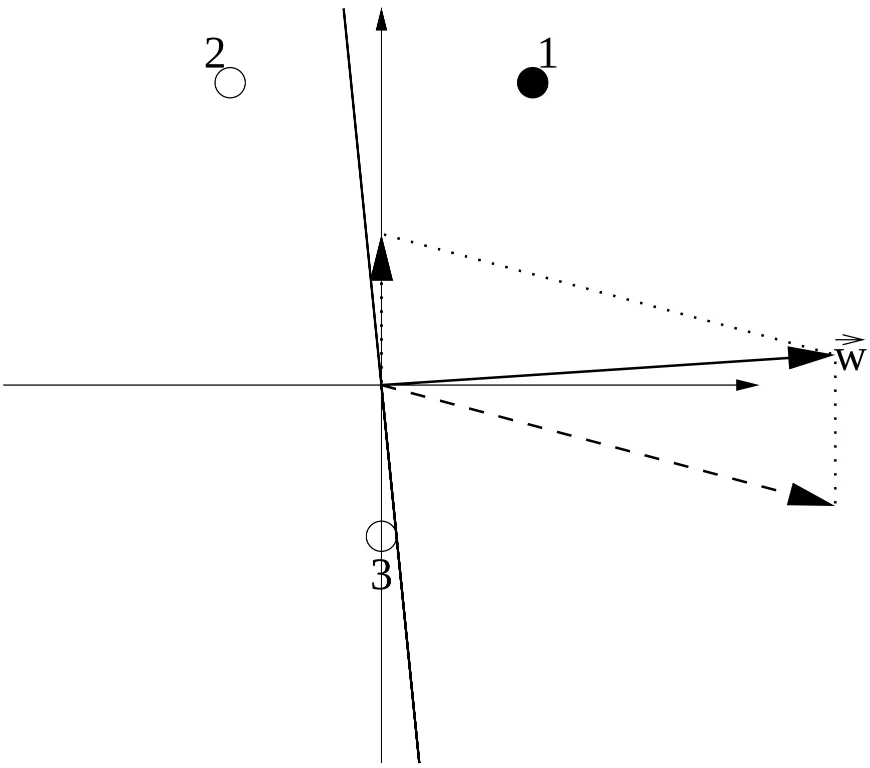
$$\text{If } d = 0 \text{ and } a = 1 \text{ then } \vec{w}^{new} = \vec{w}^{old} - \vec{x}$$



Perceptron Learning Rule
(continued)

- Third input $\vec{x}_3$ is misclassified with negative error

- Move $\vec{w}$ away from to $\vec{x}_3$: $\vec{w} = \vec{w} - \vec{x}_3$



- The perceptron will correctly classify inputs $\vec{x}_1, \vec{x}_2, \vec{x}_3$ if presented to it again. There will be no errors

- Third rule: *no error rule*

$$\text{If } d = a \text{ then } \vec{w}^{new} = \vec{w}^{old}$$

Perceptron Learning Rule
(continued)

- Unified learning rule

$$\vec{w}^{new} = \vec{w}^{old} + \delta\vec{x} = \vec{w}^{old} + (d - a)\vec{x}$$

- With learning rate η

$$\vec{w}^{new} = \vec{w}^{old} + \eta\delta\vec{x} = \vec{w}^{old} + \eta(d - a)\vec{x}$$

- Choice of learning rate η
  - Too large: learning oscillates
  - Too small: very slow learning
  - $0 < \eta \leq 1$. Popular choices:
    * $\eta = 0.5$
    * $\eta = 1$
  - Variable learning rate $\eta = \frac{|\vec{w}\vec{x}|}{|\vec{x}^2|}$
  - Adaptive learning rate
  - . . . etc.

Perceptron Learning Algorithm

```
Initialization:  w⃗_0 = 0⃗;
t = 0;
Repeat
  t = t + 1;
  Error = 0;
  For each training example [x⃗, d_x⃗] do
    net = w⃗ · x⃗;
    a_x⃗ = g(net);
    δ_x⃗ = d_x⃗ − a_x⃗;
    Error = Error + |δ_x⃗|;
    w⃗_{t+1} = w⃗_t + η · δ_x⃗ · x⃗;
    {
      or equivalently,
      For 0 ≤ i ≤ n
        w_{i,t+1} = w_{i,t} + η · δ_x⃗ · x_i;
    }
Until Error = 0;
Save last weight vector;
```

- *Perceptron convergence theorem:* [M. Minsky and S. Papert, 1969] The perceptron learning algorithm terminates if and only if the task is linearly separable

- Cannot learn non-linearly separable functions

Perceptron Learning Algorithm
(continued)

- Termination criteria
  - Assured for small enough η and l.s. functions
  - For non-l.s. functions: halt when number of misclassifications is minimal
- Problem representation
  - Non-numeric inputs: encode into numeric form
  - Multiple-class problem:
    * Use single-layer network
    * Each output node corresponds to one class
    * A u-neuron network can classify inputs into 2^u classes
- Variations of perceptron
  - Bipolar vs. binary encodings
  - Threshold vs. signum functions

Pocket Algorithm

- Robust classification for linearly non-separable problems?

- Find $\vec{w}$ such that such that the number of misclassifications is as small as possible.

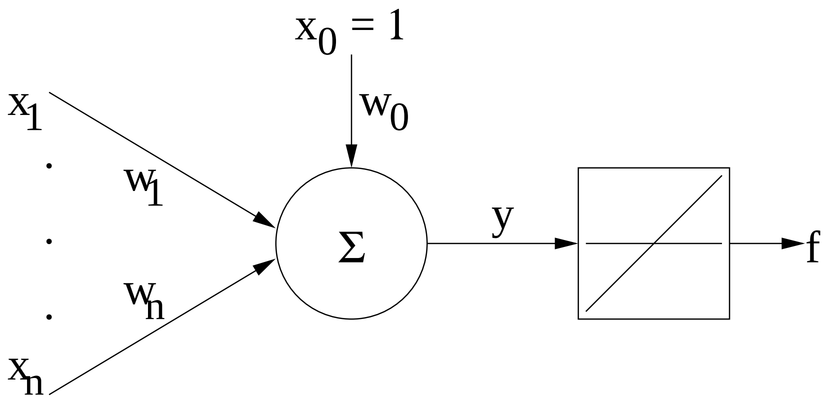
```
Initialization:  w⃗_0 = PerceptronLearning;
Error_{w⃗_0} = number of misclassifications of w⃗_0;
Pocket = w⃗_0;
t = 0;
Repeat
  t = t + 1;
  w⃗_t = PerceptronLearning;
  If Error_{w⃗_t} < Error_{w⃗_{t-1}} Then
    Pocket = w⃗_t;
Until t = MaxIterations;
Best weight so far is stored in Pocket;
```

- Initial weight in `PerceptronLearning` should be random

- Presentation of training examples in `PerceptronLearning` should be random

- Slow but robust learning for non-separable tasks

Adaline



- Architecture
  - Input: $\vec{x} = (x_0 = 1, x_1, \ldots, x_n)$
  - Weight: $\vec{w} = (w_0 = -\theta, w_1, \ldots, w_n)$, $\theta =$ bias
  - Net input: $y = \vec{w}\vec{x} = \sum_{i=0}^{n} w_i x_i$
  - Output $f(\vec{x}) = g(\vec{w}\vec{x}) = \vec{w}\vec{x}$

- Pattern classification

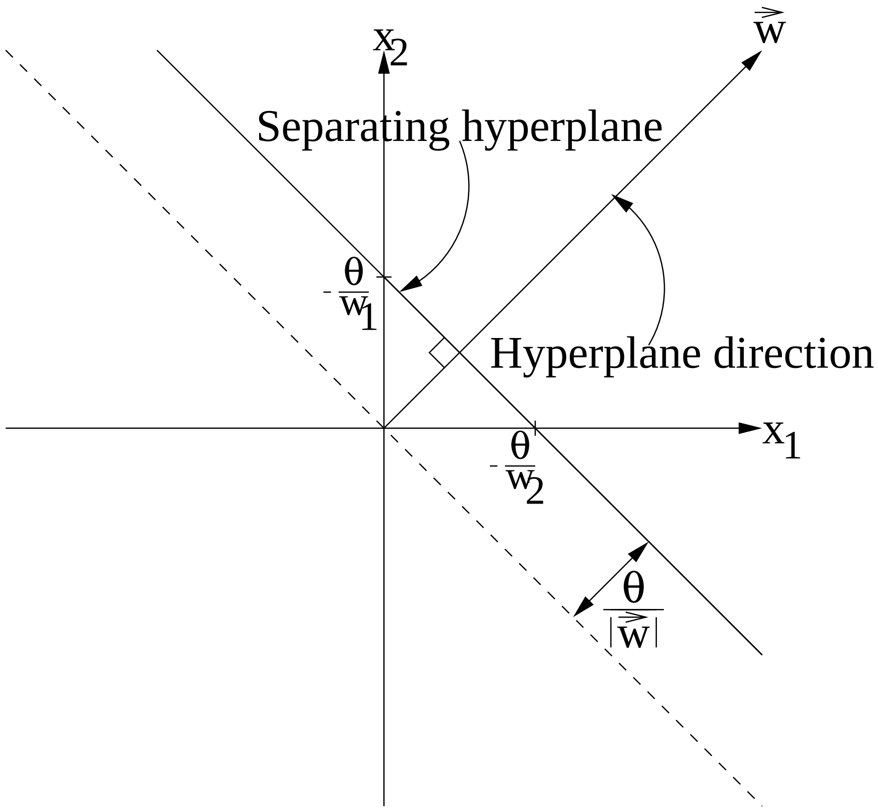
- Supervised learning

- Error-correction learning

Adaline Analysis

- Adaline's *decision boundary*

$$w_0x_0 + w_1x_1 + \cdots + w_nx_n = 0$$



- The Adaline
  - has a decision boundary like the perceptron
  - can be used to classify objects into two categories
  - has same limitation as the perceptron

Adaline Learning Principle

- Data fitting (or linear regression)
  - Set of measurements: $\{(x, d_x)\}$
  - Find w and b such that

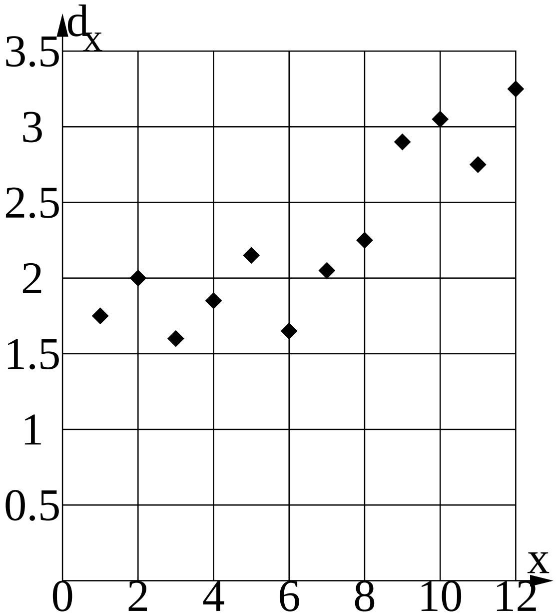
    $$d_x \approx wx + b$$

    or more specifically,

    $$d_i = wx_i + b + \varepsilon_i = y_i + \varepsilon_i$$

    where

    * $\varepsilon_i =$ instantaneous error
    * $y_i =$ linearly fitted value
    * $w =$ line slope, $b = d$-axis intercept (or bias)

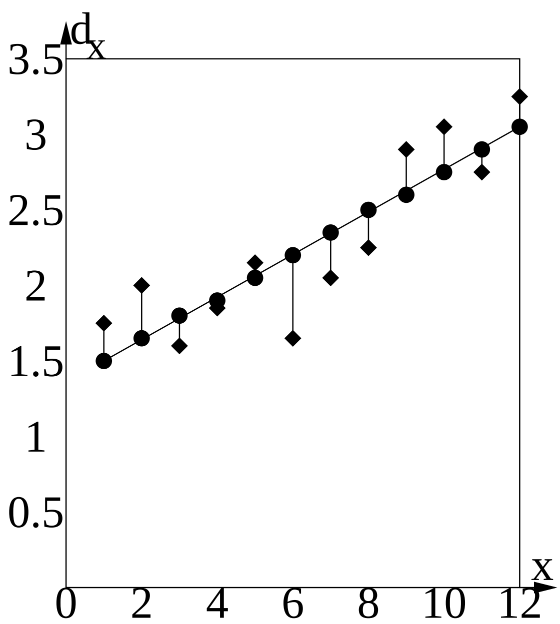


Adaline Learning Principle
(continued)

- Best fit problem: find the best choice of $(\vec{w}, b)$ such that the fitted line passes closest to all points

- Solution: Least squares
  - Minimize sum of squared errors (SSE) or mean of squared errors (MSE)
  - Error $\varepsilon_{\vec{x}} = d_{\vec{x}} - \tilde{d}_{\vec{x}}$ where $\tilde{d}_{\vec{x}} = \vec{w}\vec{x} + b$
  - MSE:

$$J = \frac{1}{N} \sum_{i=1}^{N} \varepsilon^2_{\vec{x}_i}$$



Adaline Learning Principle
(continued)

- The minimum MSE, called the *least mean square* (LMS) can be obtained analytically:

$$\frac{\delta J}{\delta \vec{w}} = 0$$

$$\frac{\delta J}{\delta b} = 0$$

  and solve for $\vec{w}$ and b

- Pattern classification can be interpreted as a linear

- LMS is difficult to obtain for larger dimensions (complex formula) and larger data sets

- Adaline:
  - Learns by minimizing the MSE
  - Not sensitive to noise
  - Powerful and robust learning

Adaline Learning Algorithm

- Gradient descent
  - A learning example: $[\vec{x}, d_{\vec{x}}]$
  - Actual output: $net_{\vec{x}} =$
  - Desired output: $d_{\vec{x}}$
  - Squared error: $E_{\vec{x}} = (d_{\vec{x}} - net_{\vec{x}})^2$
  - Gradient of $E_{\vec{x}}$:

$$\nabla E_{\vec{x}} = \frac{\delta E_{\vec{x}}}{\delta \vec{w}} = (\frac{\delta E_{\vec{x}}}{\delta w_0}, \frac{\delta E_{\vec{x}}}{\delta w_1}, \ldots, \frac{\delta E_{\vec{x}}}{\delta w_n})$$

  - $E_{\vec{x}}$ is minimal if and only if $\nabla E_{\vec{x}} = 0$
  - Negative gradient of $E_{\vec{x}}$:

$$-\nabla E_{\vec{x}}$$

    gives direction of steepest descent to the minimum
  - Gradient descent:

$$\Delta \vec{w} = -\eta \nabla E_{\vec{x}} = -\frac{\delta E_{\vec{x}}}{\delta \vec{w}}$$

Adaline Learning Algorithm
(continued)

- Widrow-Hoff delta rule
  -
$$\begin{aligned}\frac{\delta E_{\vec{x}}}{\delta w_i} &= 2(d_{\vec{x}} - net_{\vec{x}})\frac{\delta(-net_{\vec{x}})}{\delta \vec{w}_i}\\ &= (d_{\vec{x}} - net_{\vec{x}})\frac{\delta(-\sum_{j=0}^{n} w_j x_j)}{\delta \vec{w}_i}\\ &= -(d_{\vec{x}} - net_{\vec{x}})x_i\end{aligned}$$

- $\Rightarrow$ Learning rule:

$$\vec{w}^{new} = \vec{w}^{old} + \eta(d_{\vec{x}} - net_{\vec{x}})\vec{x}$$

Adaline Learning Algorithm
(continued)

```
Initialization:  w⃗_0 = 0⃗;
t = 0;
Repeat
  t = t + 1;
  For each training example [x⃗, d_x⃗] do
    net_x⃗ = w⃗ · x⃗;
    a_x⃗ = g(net_x⃗) = net_x⃗;
    δ_x⃗ = d_x⃗ − a_x⃗;
    w⃗_{t+1} = w⃗_t + η · δ_x⃗ · x⃗;
    {
      or equivalently,
      For 0 ≤ i ≤ n
        w_{i,t+1} = w_{i,t} + η · δ_x⃗ · x_i;
    }
Until MSE(w⃗) is minimal;
Save last weight vector;
```

- Can be used for function approximation task as well