Slide03

Haykin Chapter 3 (Chap 1, 3, 3rd Ed): Single-Layer Perceptrons

CPSC 636-600

Instructor: Yoonsuck Choe

Historical Overview

- McCulloch and Pitts (1943): neural networks as computing machines.
- Hebb (1949): postulated the first rule for self-organizing learning.
- Rosenblatt (1958): perceptron as a first model of supervised learning.
- Widrow and Hoff (1960): adaptive filters using least-mean-square (LMS) algorithm (delta rule).

Multiple Faces of a Single Neuron

What a single neuron does can be viewed from different perspectives:

- Adaptive filter: as in signal processing
- Classifier: as in perceptron

The two aspects will be reviewed, in the above order.

Part I: Adaptive Filter

Adaptive Filtering Problem

- Consider an *unknown* **dynamical system**, that takes m inputs and generates one output.
- Behavior of the system described as its input/output pair:

$$\mathcal{T}: \{\mathbf{x}(i), d(i); i = 1, 2, ..., n, ...\} \text{ where}$$

  $\mathbf{x}(i) = [x_1(i), x_2(i), ..., x_m(i)]^T$ is the input and $d(i)$ the desired response (or target signal).
- Input vector can be either a spatial **snapshot** or a temporal sequence **uniformly spaced in time**.
- There are two important processes in adaptive filtering:
  - Filtering process: generation of output based on the input: $y(i) = \mathbf{x}^T(i)\mathbf{w}(i)$.
  - Adapative process: automatic adjustment of weights to reduce error: $e(i) = d(i) - y(i)$.

Unconstrained Optimization Techniques

- How can we adjust $\mathbf{w}(i)$ to gradually minimize $e(i)$? Note that $e(i) = d(i) - y(i) = d(i) - \mathbf{x}^T(i)\mathbf{w}(i)$. Since $d(i)$ and $\mathbf{x}(i)$ are fixed, only the change in $\mathbf{w}(i)$ can change $e(i)$.
- In other words, we want to *minimize the cost function* $\mathcal{E}(\mathbf{w})$ *with respect to the weight vector* $\mathbf{w}$: Find the optimal solution $\mathbf{w}^*$.
- The **necessary condition** for optimality is

$$\nabla\mathcal{E}(\mathbf{w}^*) = \mathbf{0},$$

  where the **gradient operator** is defined as

$$\nabla = \left[\frac{\partial}{\partial w_1}, \frac{\partial}{\partial w_2}, \dots \frac{\partial}{\partial w_m}\right]^T$$

  With this, we get

$$\nabla\mathcal{E}(\mathbf{w}^*) = \left[\frac{\partial\mathcal{E}}{\partial w_1}, \frac{\partial\mathcal{E}}{\partial w_2}, \dots \frac{\partial\mathcal{E}}{\partial w_m}\right]^T.$$

Steepest Descent

- We want the iterative update algorithm to have the following property:

$$\mathcal{E}(\mathbf{w}(n+1)) < \mathcal{E}(\mathbf{w}(n)).$$

- Define the gradient vector $\nabla\mathcal{E}(\mathbf{w})$ as $\mathbf{g}$.
- The iterative weight update rule then becomes:

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \eta\mathbf{g}(n)$$

  where η is a small learning-rate parameter. So we can say,

$$\Delta\mathbf{w}(n) = \mathbf{w}(n+1) - \mathbf{w}(n) = -\eta\mathbf{g}(n)$$

Steepest Descent (cont'd)

We now check if $\mathcal{E}(\mathbf{w}(n+1)) < \mathcal{E}(\mathbf{w}(n))$.

Using first-order Taylor expansion[†] of $\mathcal{E}(\cdot)$ near $\mathbf{w}(n)$,

$$\mathcal{E}(\mathbf{w}(n+1)) \approx \mathcal{E}(\mathbf{w}(n)) + \mathbf{g}^T(n)\Delta\mathbf{w}(n)$$

and $\Delta\mathbf{w}(n) = -\eta\mathbf{g}(n)$, we get

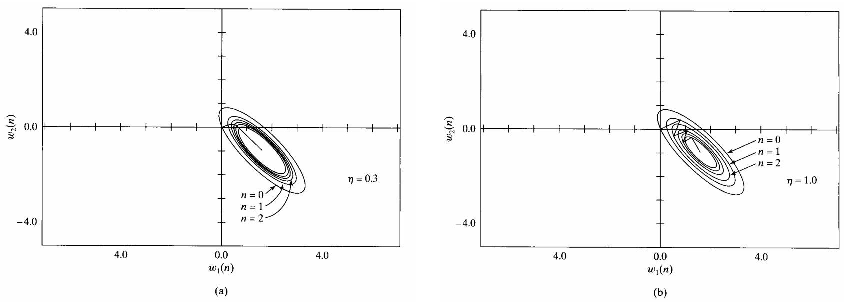
$$\begin{aligned}\mathcal{E}(\mathbf{w}(n+1)) &\approx \mathcal{E}(\mathbf{w}(n)) - \eta\mathbf{g}^T(n)\mathbf{g}(n) \\ &= \mathcal{E}(\mathbf{w}(n)) - \underbrace{\eta\|\mathbf{g}(n)\|^2}_{\text{Positive!}}.\end{aligned}$$

So, it is indeed (for small η):

$$\mathcal{E}(\mathbf{w}(n+1)) < \mathcal{E}(\mathbf{w}(n)).$$

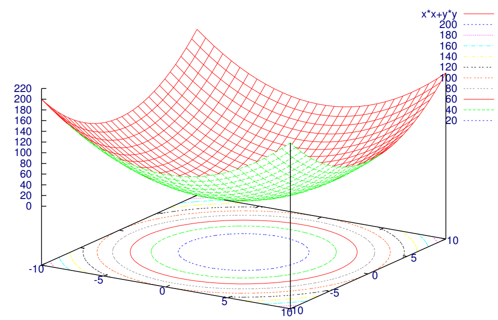
[†] Taylor series: $f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)(x-a)^2}{2!} + \dots$.

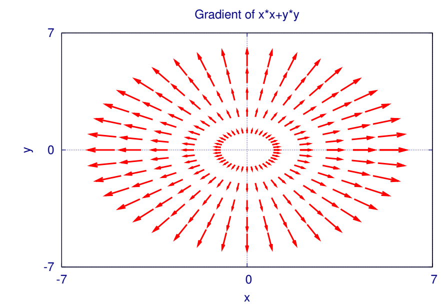
Steepest Descent: Example



- Convergence to optimal $\mathbf{w}$ is very slow.
- Small η: overdamped, smooth trajectory
- Large η: underdamped, jagged trajectory
- η too large: algorithm becomes unstable

Steepest Descent: Another Example





For $f(\mathbf{x}) = f(x, y) = x^2 + y^2$,
$\nabla f(x, y) = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right]^T = [2x, 2y]^T$. Note that (1) the gradient vectors are pointing upward, away from the origin, (2) length of the vectors are shorter near the origin. If you follow $-\nabla f(x, y)$, you will end up at the origin. We can see that the gradient vectors are perpendicular to the level curves.

* The vector lengths were scaled down by a factor of 10 to avoid clutter.

Newton's Method

- Newton's method is an extension of steepest descent, where the second-order term in the Taylor series expansion is used.
- It is generally faster and shows a less erratic meandering compared to the steepest descent method.
- There are certain conditions to be met though, such as the Hessian matrix $\nabla^2 \mathcal{E}(\mathbf{w})$ being positive definite (for an arbitarry $\mathbf{x}$, $\mathbf{x}^T \mathbf{H} \mathbf{x} > 0$).

Gauss-Newton Method

- Applicable for cost-functions expressed as sum of error squares:

$$\mathcal{E}(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{n} e_i(\mathbf{w})^2,$$

  where $e_i(\mathbf{w})$ is the error in the i-th trial, with the weight $\mathbf{w}$.

- Recalling the Taylor series $f(x) = f(a) + f'(a)(x - a)...$, we can express $e_i(\mathbf{w})$ evaluated near $e_i(\mathbf{w_k})$ as

$$e_i(\mathbf{w}) = e_i(\mathbf{w}_k) + \left[\frac{\partial e_i}{\partial \mathbf{w}}\right]^T_{\mathbf{w}=\mathbf{w}_k} (\mathbf{w} - \mathbf{w}_k).$$

- In matrix notation, we get:

$$\mathbf{e}(\mathbf{w}) = \mathbf{e}(\mathbf{w}_k) + \mathbf{J_e}(\mathbf{w}_k)(\mathbf{w} - \mathbf{w}_k).$$

* We will use a slightly different notation than the textbook, for clarity.

Gauss-Newton Method (cont'd)

- $\mathbf{J}_{\mathbf{e}}(\mathbf{w})$ is the **Jacobian matrix**, where each row is the gradient of $e_i(\mathbf{w})$:

$$\mathbf{J}_{\mathbf{e}}(\mathbf{w}) = \begin{bmatrix} \frac{\partial e_1}{\partial w_1} & \frac{\partial e_1}{\partial w_2} & \cdots & \frac{\partial e_1}{\partial w_n} \\ \frac{\partial e_2}{\partial w_1} & \frac{\partial e_2}{\partial w_2} & \cdots & \frac{\partial e_2}{\partial w_n} \\ : & : & & : \\ : & : & & : \\ \frac{\partial e_n}{\partial w_1} & \frac{\partial e_n}{\partial w_2} & \cdots & \frac{\partial e_n}{\partial w_n} \end{bmatrix} = \begin{bmatrix} (\nabla e_1(\mathbf{w}))^T \\ (\nabla e_2(\mathbf{w}))^T \\ : \\ : \\ (\nabla e_n(\mathbf{w}))^T \end{bmatrix}$$

- We can then evaluate $\mathbf{J}_{\mathbf{e}}(\mathbf{w}_k)$ by plugging in actual values of $\mathbf{w}_k$ into the Jabobian matrix above.

Quick Example: Jacobian Matrix

- Given

$$\mathbf{e}(x, y) = \begin{bmatrix} e_1(x, y) \\ e_2(x, y) \end{bmatrix} = \begin{bmatrix} x^2 + y^2 \\ \cos(x) + \sin(y) \end{bmatrix},$$

- The Jacobian of $\mathbf{e}(x, y)$ becomes

$$\mathbf{J}_{\mathbf{e}}(x, y) = \begin{bmatrix} \frac{\partial e_1(x,y)}{\partial x} & \frac{\partial e_1(x,y)}{\partial y} \\ \frac{\partial e_2(x,y)}{\partial x} & \frac{\partial e_2(x,y)}{\partial y} \end{bmatrix} = \begin{bmatrix} 2x & 2y \\ -\sin(x) & \cos(y) \end{bmatrix}.$$

- For $(x, y) = (0.5\pi, \pi)$, we get

$$\mathbf{J}_{\mathbf{e}}(0.5\pi, \pi) = \begin{bmatrix} \pi & 2\pi \\ -\sin(0.5\pi) & \cos(\pi) \end{bmatrix} = \begin{bmatrix} \pi & 2\pi \\ -1 & -1 \end{bmatrix}.$$

Gauss-Newton Method (cont'd)

- Again, starting with

$$\mathbf{e}(\mathbf{w}) = \mathbf{e}(\mathbf{w}_k) + \mathbf{J}_{\mathbf{e}}(\mathbf{w}_k)(\mathbf{w} - \mathbf{w}_k),$$

what we want is to set $\mathbf{w}$ so that the error approaches 0.

- That is, we want to minimize the norm of $\mathbf{e}(\mathbf{w})$:

$$\begin{aligned} \|\mathbf{e}(\mathbf{w})\|^2 &= \|\mathbf{e}(\mathbf{w}_k)\|^2 + 2\mathbf{e}(\mathbf{w}_k)^T \mathbf{J}_{\mathbf{e}}(\mathbf{w}_k)(\mathbf{w} - \mathbf{w}_k) \\ &+ (\mathbf{w} - \mathbf{w}_k)^T \mathbf{J}_{\mathbf{e}}^T(\mathbf{w}_k)\mathbf{J}_{\mathbf{e}}(\mathbf{w}_k)(\mathbf{w} - \mathbf{w}_k). \end{aligned}$$

- Differentiating the above wrt $\mathbf{w}$ and setting the result to 0, we get

$\mathbf{J}_{\mathbf{e}}^T(\mathbf{w}_k)\mathbf{e}(\mathbf{w}_k) + \mathbf{J}_{\mathbf{e}}^T(\mathbf{w}_k)\mathbf{J}_{\mathbf{e}}(\mathbf{w}_k)(\mathbf{w} - \mathbf{w}_k) = \mathbf{0}$, from which we get

$$\mathbf{w} = \mathbf{w}_k - (\mathbf{J}_{\mathbf{e}}^T(\mathbf{w}_k)\mathbf{J}_{\mathbf{e}}(\mathbf{w}_k))^{-1}\mathbf{J}_{\mathbf{e}}^T(\mathbf{w}_k)\mathbf{e}(\mathbf{w}_k).$$

* $\mathbf{J}_{\mathbf{e}}^T(\mathbf{w}_k)\mathbf{J}_{\mathbf{e}}(\mathbf{w}_k)$ needs to be nonsingular (inverse is needed).

Linear Least-Square Filter

- Given m input and 1 output function $y(i) = \phi(\mathbf{x}_i^T \mathbf{w}_i)$ where $\phi(x) = x$, i.e., it is **linear**, and a set of training samples $\{\mathbf{x}_i, d_i\}_{i=1}^n$, we can define the error vector for an arbitrary weight $\mathbf{w}$ as

$$\mathbf{e}(\mathbf{w}) = \mathbf{d} - [\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n]^T \mathbf{w}.$$

where $\mathbf{d} = [d_1, d_2, ..., d_n]^T$. Setting $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n]^T$, we get: $\mathbf{e}(\mathbf{w}) = \mathbf{d} - \mathbf{X}\mathbf{w}$.

- Differentiating the above wrt $\mathbf{w}$, we get $\nabla \mathbf{e}(\mathbf{w}) = -\mathbf{X}^T$. So, the Jacobian becomes $\mathbf{J}_{\mathbf{e}}(\mathbf{w}) = (\nabla \mathbf{e}(\mathbf{w}))^T = -\mathbf{X}$.
- Plugging this in to the Gauss-Newton equation, we finally get:

$$\begin{aligned} \mathbf{w} &= \mathbf{w}_k + (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T(\mathbf{d} - \mathbf{X}\mathbf{w}_k) \\ &= \mathbf{w}_k + (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{d} - \underbrace{(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{X}\mathbf{w}_k}_{\text{This is } \mathbf{I}\mathbf{w}_k = \mathbf{w}_k.} \\ &= (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{d}. \end{aligned}$$

Linear Least-Square Filter (cont'd)

Points worth noting:

- $\mathbf{X}$ does not need to be a square matrix!
- We get $\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{d}$ off the bat partly because the output is linear (otherwise, the formula would be more complex).
- The Jacobian of the error function only depends on the input, and is invariant wrt the weight $\mathbf{w}$.
- The factor $(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$ (let's call it $\mathbf{X}^+$) is like an inverse. Multiply $\mathbf{X}^+$ to both sides of

$$\mathbf{d} = \mathbf{X}\mathbf{w}$$

then we get:

$$\mathbf{w} = \mathbf{X}^+\mathbf{d} = \underbrace{\mathbf{X}^+\mathbf{X}}_{=\mathbf{I}}\mathbf{w}.$$

Linear Least-Square Filter: Example

See `src/pseudoinv.m`.

```
 X = ceil(rand(4,2)*10), wtrue = rand(2,1)*10 , d=X*wtrue, w = inv(X'*X)*X'*d
X =
  10   7
   3   7
   3   6
   5   4

wtrue =
  0.56644
  4.99120

d =
  40.603
  36.638
  31.647
  22.797

w =
  0.56644
  4.99120
```


Least-Mean-Square Algorithm

- Cost function is based on **instantaneous values**.

$$\mathcal{E}(\mathbf{w}) = \frac{1}{2}e^2(\mathbf{w})$$

- Differentiating the above wrt $\mathbf{w}$, we get

$$\frac{\partial\mathcal{E}(\mathbf{w})}{\partial\mathbf{w}} = e(\mathbf{w})\frac{\partial e(\mathbf{w})}{\partial\mathbf{w}}.$$

- Pluggin in $e(\mathbf{w}) = d - \mathbf{x}^T\mathbf{w}$,

$$\frac{\partial e(\mathbf{w})}{\partial\mathbf{w}} = -\mathbf{x}, \text{ and hence } \frac{\partial\mathcal{E}(\mathbf{w})}{\partial\mathbf{w}} = -\mathbf{x}e(\mathbf{w}).$$

- Using this in the steepest descent rule, we get the **LMS algorithm**:

$$\hat{\mathbf{w}}_{n+1} = \hat{\mathbf{w}}_n + \eta\mathbf{x}_n e_n.$$

- Note that this weight update is done with **only one** $(\mathbf{x}_i, d_i)$ pair!

Least-Mean-Square Algorithm: Evaluation

- LMS algorithm behaves like a **low-pass filter**.
- LMS algorithm is simple, model-independent, and thus robust.
- LMS does not follow the direction of steepest descent: Instead, it follows it stochastically (stochastic gradient descent).
- Slow convergence is an issue.
- LMS is sensitive to the input correlation matrix's condition number (ratio between largest vs. smallest eigenvalue of the correl. matrix).
- LMS can be shown to converge if the learning rate has the following property:

$$0 < \eta < \frac{2}{\lambda_{\max}}$$

where $\lambda_{\max}$ is the largest eigenvalue of the correl. matrix.

Improving Convergence in LMS

- The main problem arises because of the fixed η.
- One solution: Use a time-varying learning rate: $\eta(n) = c/n$, as in *stochastic optimization theory*.
- A better alternative: use a hybrid method called *search-then-converge*.

$$\eta(n) = \frac{\eta_0}{1 + (n/\tau)}$$

When $n < \tau$, performance is similar to standard LMS. When $n > \tau$, it behaves like stochastic optimization.

Search-Then-Converge in LMS

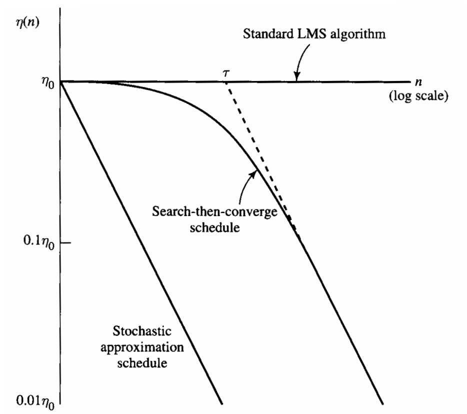
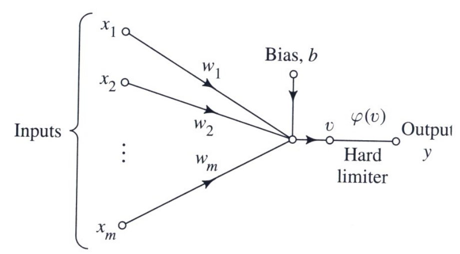


FIGURE 3.5 Learning-rate annealing schedules.

$$\eta(n) = \frac{\eta_0}{n} \text{ vs. } \eta(n) = \frac{\eta_0}{1 + (n/\tau)}$$

Part II: Perceptron

The Perceptron Model

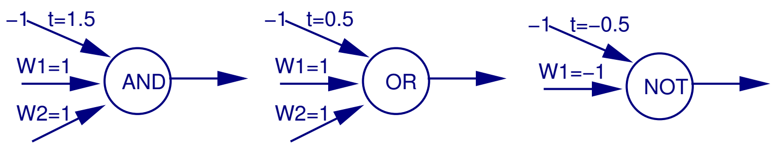


- Perceptron uses a non-linear neuron model (McCulloch-Pitts model).

$$v = \sum_{i=1}^{m} w_i x_i + b, \qquad y = \phi(v) = \begin{cases} 1 & \text{if } v > 0 \\ 0 & \text{if } v \leq 0 \end{cases}$$

- Goal: classify input vectors into two classes.

Boolean Logic Gates with Perceptron Units

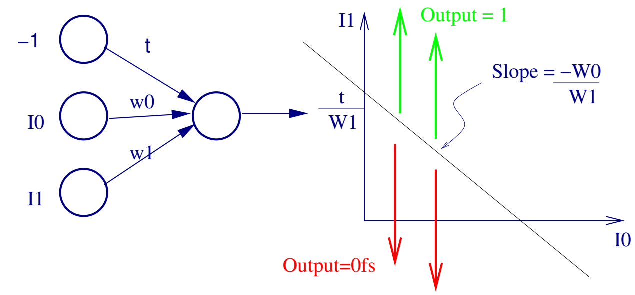


Russel & Norvig

- Perceptrons can represent basic boolean functions.
- Thus, a network of perceptron units can compute any Boolean function.

What about XOR or EQUIV?

What Perceptrons Can Represent



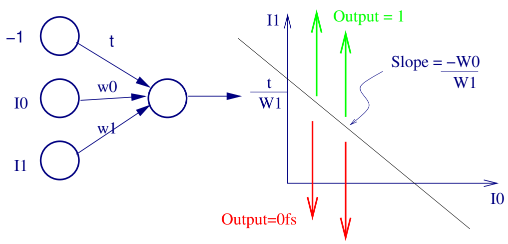
Perceptrons can only represent **linearly separable** functions.

- Output of the perceptron:

$$W_0 \times I_0 + W_1 \times I_1 - t > 0, \text{ then output is } 1$$

$$W_0 \times I_0 + W_1 \times I_1 - t \leq 0, \text{ then output is } 0$$

Geometric Interpretation



- Rearranging

$$W_0 \times I_0 + W_1 \times I_1 - t > 0, \text{ then output is } 1,$$

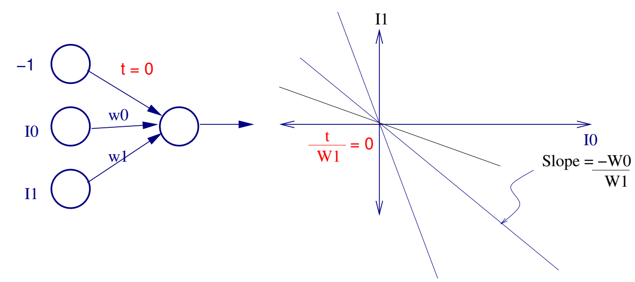
we get (if $W_1 > 0$)

$$I_1 > \frac{-W_0}{W_1} \times I_0 + \frac{t}{W_1},$$

where points above the line, the output is 1, and 0 for those below the line. Compare with

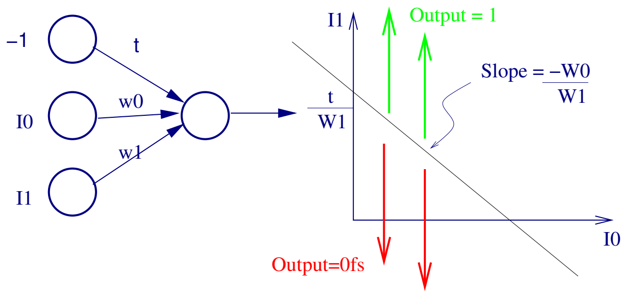
$$y = \frac{-W_0}{W_1} \times x + \frac{t}{W_1}.$$

The Role of the Bias



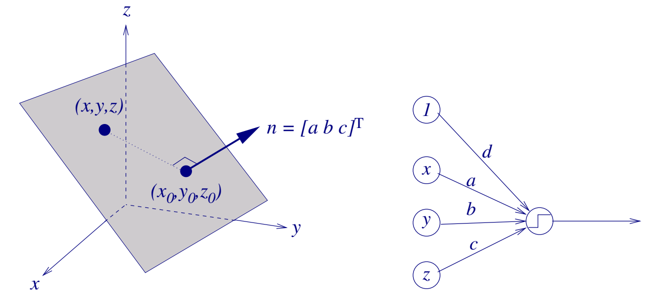
- Without the bias ($t = 0$), learning is limited to adjustment of the slope of the separating line passing through the origin.
- Three example lines with different weights are shown.

Limitation of Perceptrons



- Only functions where the 0 points and 1 points are clearly linearly separable can be represented by perceptrons.
- The geometric interpretation is generalizable to functions of n arguments, i.e. perceptron with n inputs plus one threshold (or bias) unit.

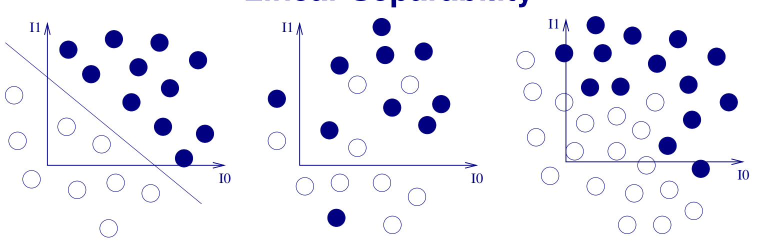
Generalizing to n-Dimensions



http://mathworld.wolfram.com/Plane.html

- $\vec{n} = (a, b, c), \vec{x} = (x, y, z), \vec{x_0} = (x_0, y_0, z_0)$.
- Equation of a plane: $\vec{n} \cdot (\vec{x} - \vec{x_0}) = 0$
- In short, $ax + by + cz + d = 0$, where a, b, c can serve as the weight, and $d = -\vec{n} \cdot \vec{x_0}$ as the bias.
- For n-D input space, the decision boundary becomes a $(n-1)$-D hyperplane (1-D less than the input space).

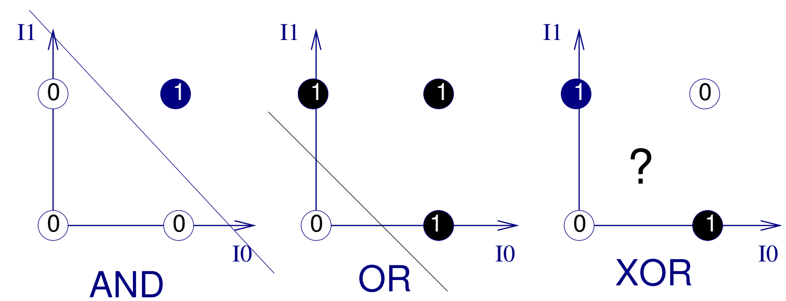
Linear Separability



Linearly–separable Not Linearly–separable Not Linearly–separable

- For functions that take integer or real values as arguments and output either 0 or 1.
- Left: linearly separable (i.e., can draw a straight line between the classes).
- Right: not linearly separable (i.e., perceptrons cannot represent such a function)

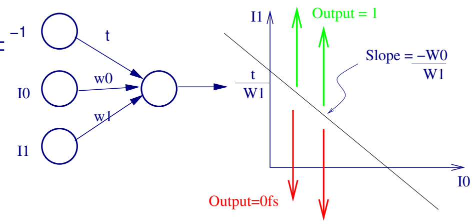
Linear Separability (cont'd)



- Perceptrons cannot represent XOR!
- Minsky and Papert (1969)

XOR in Detail

#	I_0	I_1	XOR
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	0



$W_0 \times I_0 + W_1 \times I_1 - t > 0,$ then output is 1:

1 $\quad -t \leq 0 \quad \rightarrow \quad t \geq 0$

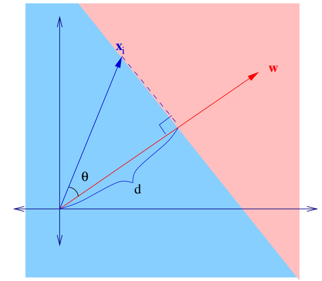
2 $\quad W_1 - t > 0 \quad \rightarrow \quad W_1 > t$

3 $\quad W_0 - t > 0 \quad \rightarrow \quad W_0 > t$

4 $\quad W_0 + W_1 - t \leq 0 \quad \rightarrow \quad W_0 + W_1 \leq t$

$2t < W_0 + W_1 < t$ (from 2, 3, and 4), but $t \geq 0$ (from 1), a contradiction.

Perceptrons: A Different Perspective



$$\mathbf{w}^T\mathbf{x} > b \text{ then, output is 1}$$

$$\mathbf{w}^T\mathbf{x} = \|\mathbf{w}\|\|\mathbf{x}\|\cos\theta > b \text{ then, output is 1}$$

$$\|\mathbf{x}\|\cos\theta > \frac{b}{\|\mathbf{w}\|} \text{ then, output is 1}$$

So, if $d = \|\mathbf{x}\|\cos\theta$ in the figure above is greater than $\frac{b}{\|\mathbf{w}\|}$, then output = 1.

Adjusting $\mathbf{w}$ changes the tilt of the decision boundary, and adjusting the bias b (and $\|\mathbf{w}\|$) moves the decision boundary closer or away from the origin.

Perceptron Learning Rule

- Given a linearly separable set of inputs that can belong to class $\mathcal{C}_1$ or $\mathcal{C}_2$,
- The goal of perceptron learning is to have

$$\mathbf{w}^T\mathbf{x} > 0 \text{ for all input in class } \mathcal{C}_1$$

$$\mathbf{w}^T\mathbf{x} \leq 0 \text{ for all input in class } \mathcal{C}_2$$

- If all inputs are correctly classified with the current weights $\mathbf{w}(n)$,

$$\mathbf{w}(n)^T\mathbf{x} > 0, \text{ for all input in class } \mathcal{C}_1, \text{ and}$$

$$\mathbf{w}(n)^T\mathbf{x} \leq 0, \text{ for all input in class } \mathcal{C}_2,$$

  then $\mathbf{w}(n+1) = \mathbf{w}(n)$ (no change).

- Otherwise, adjust the weights.

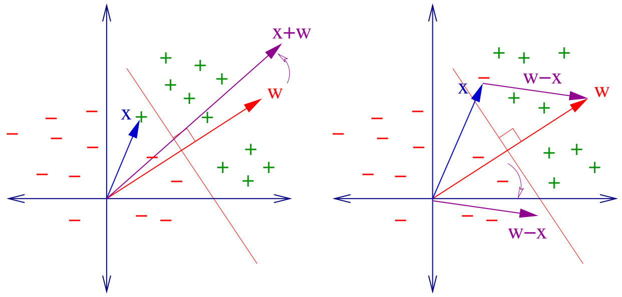
Perceptron Learning Rule (cont'd)

For misclassified inputs ($\eta(n)$ is the learning rate):

- $\mathbf{w}(n+1) = \mathbf{w}(n) - \eta(n)\mathbf{x}(n)$ if $\mathbf{w}^T\mathbf{x} > 0$ and $\mathbf{x} \in \mathcal{C}_2$.
- $\mathbf{w}(n+1) = \mathbf{w}(n) + \eta(n)\mathbf{x}(n)$ if $\mathbf{w}^T\mathbf{x} \leq 0$ and $\mathbf{x} \in \mathcal{C}_1$.

Or, simply $\mathbf{x}(n+1) = \mathbf{w}(n) + \eta(n)e(n)\mathbf{x}(n)$, where $e(n) = d(n) - y(n)$ (the error).

Learning in Perceptron: Another Look



- When a positive example ($\mathcal{C}_1$) is misclassified, $\mathbf{w}(n+1) = \mathbf{w}(n) + \eta(n)\mathbf{x}(n)$.
- When a negative example ($\mathcal{C}_2$) is misclassified, $\mathbf{w}(n+1) = \mathbf{w}(n) - \eta(n)\mathbf{x}(n)$.
- Note the tilt in the weight vector, and observe how it would change the decision boundary.

Perceptron Convergence Theorem

- Given a set of linearly separable inputs, Without loss of generality, assume $\eta = 1$, $\mathbf{w}(0) = \mathbf{0}$.
- Assume the first n examples $\in \mathcal{C}_1$ are all misclassified.
- Then, using $\mathbf{w}(n+1) = \mathbf{w}(n) + \mathbf{x}(n)$, we get

$$\mathbf{w}(n+1) = \mathbf{x}(1) + \mathbf{x}(2) + \ldots + \mathbf{x}(n). \quad (1)$$

- Since the input set is linearly separable, there is at least on solution $\mathbf{w}_0$ such that $\mathbf{w}_0^T\mathbf{x}(n) > 0$ for all inputs in $\mathcal{C}_1$.
  - Define $\alpha = \min_{\mathbf{x}(n)\in\mathcal{C}_1} \mathbf{w}_0^T\mathbf{x}(n) > 0$.
  - Multiply both sides in eq. 1 with $\mathbf{w}_0$, we get:

$$\mathbf{w}_0^T\mathbf{w}(n+1) = \mathbf{w}_0^T\mathbf{x}(1) + \mathbf{w}_0^T\mathbf{x}(2) + \ldots + \mathbf{w}_0^T\mathbf{x}(n). \quad (2)$$

  - From the two steps above, we get:

$$\mathbf{w}_0^T\mathbf{w}(n+1) > n\alpha \quad (3)$$

Perceptron Convergence Theorem (cont'd)

- Using Cauchy-Schwartz inequality

$$\|\mathbf{w}_0\|^2\|\mathbf{w}(n+1)\|^2 \geq \left[\mathbf{w}_0^T\mathbf{w}(n+1)\right]^2$$

- From the above and $\mathbf{w}_0^T\mathbf{w}(n+1) > n\alpha$,

$$\|\mathbf{w}_0\|^2\|\mathbf{w}(n+1)\|^2 \geq n^2\alpha^2$$

So, finally, we get

$$\underbrace{\|\mathbf{w}(n+1)\|^2 \geq \frac{n^2\alpha^2}{\|\mathbf{w}_0\|^2}}_{\text{First main result}} \quad (4)$$

Perceptron Convergence Theorem (cont'd)

- Taking the Euclidean norm of $\mathbf{w}(k+1) = \mathbf{w}(k) + \mathbf{x}(k)$,

$$\|\mathbf{w}(k+1)\|^2 = \|\mathbf{w}(k)\|^2 + 2\mathbf{w}^T(k)\mathbf{x}(k) + \|\mathbf{x}(k)\|^2$$

- Since all n inputs in $\mathcal{C}_1$ are misclassified, $\mathbf{w}^T(k)\mathbf{x}(k) \leq 0$ for $k = 1, 2, \ldots, n$,

$$\|\mathbf{w}(k+1)\|^2 - \|\mathbf{w}(k)\|^2 - \|\mathbf{x}(k)\|^2 = 2\mathbf{w}^T(k)\mathbf{x}(k) \leq 0,$$

$$\|\mathbf{w}(k+1)\|^2 \leq \|\mathbf{w}(k)\|^2 + \|\mathbf{x}(k)\|^2$$

$$\|\mathbf{w}(k+1)\|^2 - \|\mathbf{w}(k)\|^2 \leq \|\mathbf{x}(k)\|^2$$

- Summing up the inequalities for all $k = 1, 2, \ldots, n$, and $\mathbf{w}(0) = \mathbf{0}$, we get

$$\|\mathbf{w}(k+1)\|^2 \leq \sum_{k=1}^{n} \|\mathbf{x}(k)\|^2 \leq n\beta, \quad (5)$$

where $\beta = \max_{\mathbf{x}}(k) \in \mathcal{C}_1 \|\mathbf{x}(k)\|^2$.

Perceptron Convergence Theorem (cont'd)

- From eq. 4 and eq. 5,

$$\frac{n^2\alpha^2}{\|\mathbf{w}_0\|^2} \le \|\mathbf{w}(n+1)\|^2 \le n\beta$$

- Here, α is a constant, depending on the fixed input set and the fixed solution $\mathbf{w}_0$ (so, $\|\mathbf{w}_0\|$ is also a constant), and β is also a constant since it depends only on the fixed input set.
- In this case, if n grows to a large value, the above inequality will becomes invalid (n is a positive integer).
- Thus, n cannot grow beyond a certain $n_{\max}$, where

$$\frac{n_{\max}^2\alpha^2}{\|\mathbf{w}_0\|^2} = n_{\max}\beta$$

$$n_{\max} = \frac{\beta\|\mathbf{w}_0\|^2}{\alpha^2},$$

and when $n = n_{\max}$, all inputs will be correctly classified

Fixed-Increment Convergence Theorem

Let the subsets of training vectors $\mathcal{C}_1$ and $\mathcal{C}_2$ be linearly separable. Let the inputs presented to perceptron originate from these two subsets. The perceptron converges after some n_0 iterations, in the sense that

$$\mathbf{w}(n_0) = \mathbf{w}(n_0+1) = \mathbf{w}(n_0+2) = \ldots.$$

is a solution vector for $n_0 \le n_{\max}$.

TABLE 3.2 Summary of the Perceptron Convergence Algorithm

Variables and Parameters:

$\mathbf{x}(n) = (m+1)$-by-1 input vector
$= [+1, x_1(n), x_2(n), \ldots, x_m(n)]^T$

$\mathbf{w}(n) = (m+1)$-by-1 weight vector
$= [b(n), w_1(n), w_2(n), \ldots, w_m(n)]^T$

$b(n) =$ bias

$y(n) =$ actual response (quantized)

$d(n) =$ desired response

$\eta =$ learning-rate parameter, a positive constant less than unity

1. *Initialization.* Set $\mathbf{w}(0) = \mathbf{0}$. Then perform the following computations for time step $n = 1, 2, \ldots$.

2. *Activation.* At time step n, activate the perceptron by applying continuous-valued input vector $\mathbf{x}(n)$ and desired response $d(n)$.

3. *Computation of Actual Response.* Compute the actual response of the perceptron:

$$y(n) = \mathrm{sgn}[\mathbf{w}^T(n)\mathbf{x}(n)]$$

where $\mathrm{sgn}(\cdot)$ is the signum function.

4. *Adaptation of Weight Vector.* Update the weight vector of the perceptron:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta[d(n) - y(n)]\mathbf{x}(n)$$

where

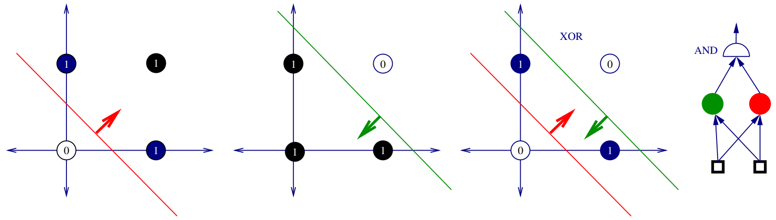
$$d(n) = \begin{cases} +1 & \text{if } \mathbf{x}(n) \text{ belongs to class } \mathcal{C}_1 \\ -1 & \text{if } \mathbf{x}(n) \text{ belongs to class } \mathcal{C}_2 \end{cases}$$

5. *Continuation.* Increment time step n by one and go back to step 2.

Summary

- Adaptive filter using the LMS algorithm and perceptrons are closely related (the learning rule is almost identical).
- LMS and perceptrons are different, however, since one uses linear activation and the other hard limiters.
- LMS is used in continuous learning, while perceptrons are trained for only a finite number of steps.
- Single-neuron or single-layer has severe limits: How can multiple layers help?

XOR with Multilayer Perceptrons



Note: the bias units are not shown in the network on the right, but they are needed.

- Only three perceptron units are needed to implement XOR.
- However, you need two layers to achieve this.