Historical Overview

- 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:
  \[ T : \{x(i), d(i); i = 1, 2, ..., n, \ldots \} \] where \( x(i) = [x_1(i), x_2(i), \ldots, x_m(i)]^T \) 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:
  \[ y(i) = x^T(i)w(i). \]
- Adaptive process: automatic adjustment of weights to reduce error:
  \[ e(i) = d(i) - y(i). \]

Steepest Descent

- We want the iterative update algorithm to have the following property:
  \[ E(w(n+1)) < E(w(n)). \]
- Define the gradient vector \( \nabla E(w) \) as \( g \).
- The iterative weight update rule then becomes:
  \[ w(n+1) = w(n) - \eta g(n) \]
  where \( \eta \) is a small learning-rate parameter. So we can say,
  \[ \Delta w(n) = w(n+1) - w(n) = -\eta g(n) \]

Unconstrained Optimization Techniques

- How can we adjust \( w(i) \) to gradually minimize \( e(i) \)? Note that \( e(i) = d(i) - y(i) = d(i) - x^T(i)w(i). \) Since \( d(i) \) and \( x(i) \) are fixed, only the change in \( w(i) \) can change \( e(i) \).
- In other words, we want to minimize the cost function \( E(w) \) with respect to the weight vector \( w \): Find the optimal solution \( w^* \).
- The necessary condition for optimality is
  \[ \nabla E(w^*) = 0, \]
  where the gradient operator is defined as
  \[ \nabla = \left[ \frac{\partial}{\partial w_1}, \frac{\partial}{\partial w_2}, \ldots, \frac{\partial}{\partial w_m} \right]^T. \]
  With this, we get
  \[ \nabla E(w^*) = \left[ \frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_m} \right]^T. \]

Steepest Descent (cont’d)

We now check if \( E(w(n+1)) < E(w(n)) \).

Using first-order Taylor expansion\(^\dagger\) of \( E(\cdot) \) near \( w(n) \),
\[ E(w(n+1)) \approx E(w(n)) + g^T(n)\Delta w(n) \]
and \( \Delta w(n) = -\eta g(n) \), we get
\[ E(w(n+1)) \approx E(w(n)) - \eta g^T(n)g(n) \]
\[ = E(w(n)) - \eta \|g(n)\|^2. \]
Positive!

So, it is indeed (for small \( \eta \)):
\[ E(w(n+1)) < E(w(n)). \]

\(^\dagger\) Taylor series: \[ f(x) = f(a) + f'(a)(x - a) + \frac{f''(a)(x-a)^2}{2!} + \ldots. \]
Steepest Descent: Example

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

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 $\nabla^2 \mathcal{E}(w)$ being positive definite (for an arbitrary $x, x^T H x > 0$).

Gauss-Newton Method

- Applicable for cost-functions expressed as sum of error squares:
  $$\mathcal{E}(w) = \frac{1}{2} \sum_{i=1}^{n} e_i(w)^2,$$
  where $e_i(w)$ is the error in the $i$-th trial, with the weight $w$.
- Recalling the Taylor series $f(x) = f(a) + f'(a)(x-a)\ldots$, we can express $e_i(w)$ evaluated near $e_i(w_k)$ as
  $$e_i(w) = e_i(w_k) + \left( \frac{\partial e_i}{\partial w} \right)^T_{w=w_k} (w-w_k).$$
- In matrix notation, we get:
  $$\mathbf{e}(w) = \mathbf{e}(w_k) + \mathbf{J}_e(w_k)(w-w_k).$$

For $f(x) = f(x, y) = x^2 + y^2$,
$$\nabla f(x, y) = \left[ \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right]^T = [2x, 2y]^T$$.
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.
Gauss-Newton Method (cont’d)

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

\[
\mathbf{J}_e(\mathbf{w}) = \begin{bmatrix}
\frac{\partial e_1}{\partial w_1} & \frac{\partial e_1}{\partial w_2} & \cdots & \frac{\partial e_1}{\partial w_n} \\
\frac{\partial e_2}{\partial w_1} & \frac{\partial e_2}{\partial w_2} & \cdots & \frac{\partial e_2}{\partial w_n} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial e_n}{\partial w_1} & \frac{\partial e_n}{\partial w_2} & \cdots & \frac{\partial e_n}{\partial w_n}
\end{bmatrix} \quad = \quad \begin{bmatrix}
(\nabla e_1(\mathbf{w}))^T \\
(\nabla e_2(\mathbf{w}))^T \\
\vdots \\
(\nabla e_n(\mathbf{w}))^T
\end{bmatrix}
\]

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

Gauss-Newton Method (cont’d)

- Again, starting with

\[
e(\mathbf{w}) = e(\mathbf{w}_k) + \mathbf{J}_e(\mathbf{w}_k)(\mathbf{w} - \mathbf{w}_k),
\]

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

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

\[
\|e(\mathbf{w})\|^2 = \|e(\mathbf{w}_k)\|^2 + 2\mathbf{e}(\mathbf{w}_k)^T \mathbf{J}_e(\mathbf{w}_k)(\mathbf{w} - \mathbf{w}_k)
+ (\mathbf{w} - \mathbf{w}_k)^T \mathbf{J}_e^T(\mathbf{w}_k)\mathbf{J}_e(\mathbf{w}_k)(\mathbf{w} - \mathbf{w}_k).
\]

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

\[
\mathbf{J}_e^T(\mathbf{w}_k)e(\mathbf{w}_k) + \mathbf{J}_e^T(\mathbf{w}_k)\mathbf{J}_e(\mathbf{w}_k)(\mathbf{w} - \mathbf{w}_k) = 0,
\]
and from which we get

\[
\mathbf{w} = \mathbf{w}_k - (\mathbf{J}_e^T(\mathbf{w}_k)\mathbf{J}_e(\mathbf{w}_k))^{-1}\mathbf{J}_e^T(\mathbf{w}_k)e(\mathbf{w}_k).
\]

- \( \mathbf{J}_e^T(\mathbf{w}_k)\mathbf{J}_e(\mathbf{w}_k) \) needs to be nonsingular (inverse is needed).

Quick Example: Jacobian Matrix

- Given

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

- The Jacobian of \( e(x, y) \) becomes

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

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

\[
\mathbf{J}_e(0.5\pi, \pi) = \begin{bmatrix} \frac{\pi}{2} & \frac{2\pi}{2} \\ -\sin(0.5\pi) & \cos(\pi) \end{bmatrix} = \begin{bmatrix} \frac{\pi}{2} & \frac{2\pi}{2} \\ -1 & -1 \end{bmatrix}.
\]

Linear Least-Square Filter

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

\[
e(\mathbf{w}) = \mathbf{d} - [x_1, x_2, \ldots, x_n]^T \mathbf{w}.
\]

where \( \mathbf{d} = [d_1, d_2, \ldots, d_n]^T \). Setting \( \mathbf{X} = [x_1, x_2, \ldots, x_n]^T \), we get: \( e(\mathbf{w}) = \mathbf{d} - \mathbf{X} \mathbf{w} \).

- Differentiating the above wrt \( \mathbf{w} \), we get \( \nabla e(\mathbf{w}) = -\mathbf{X}^T \). So, the Jacobian becomes \( \mathbf{J}_e(\mathbf{w}) = (\nabla e(\mathbf{w}))^T = -\mathbf{X} \).

- Plugging this in to the Gauss-Newton equation, we finally get:

\[
\mathbf{w} = \mathbf{w}_k + (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T (\mathbf{d} - \mathbf{X} \mathbf{w}_k)
= \mathbf{w}_k + (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{d} - (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{X} \mathbf{w}_k
= (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{d}.
\]

This is \( \mathbf{Iw}_k = \mathbf{w}_k \).
Linear Least-Square Filter (cont’d)

Points worth noting:
- \( X \) does not need to be a square matrix!
- We get \( w = (X^T X)^{-1} X^T d \) 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 \( w \).
- The factor \( (X^T X)^{-1} X^T \) (let’s call it \( X^+ \)) is like an inverse. Multiply \( X^+ \) to both sides of \( d = X w \) then we get:

\[
    w = X^+ d = X^+ X w.
\]

Least-Mean-Square Algorithm

- Cost function is based on instantaneous values.
  
  \[ E(w) = \frac{1}{2} e^2(w) \]

- Differentiating the above wrt \( w \), we get
  
  \[ \frac{\partial E(w)}{\partial w} = e(w) \frac{\partial e(w)}{\partial w}. \]

- Plugging in \( e(w) = d - x^T w \),
  
  \[ \frac{\partial e(w)}{\partial w} = -x, \text{ and hence } \frac{\partial E(w)}{\partial w} = -xe(w). \]

- Using this in the steepest descent rule, we get the LMS algorithm:
  
  \[ \hat{w}_{n+1} = \hat{w}_n + \eta x_n e_n. \]

- Note that this weight update is done with only one \((x_i, d_i)\) 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:
  
  \[ 0 < \eta < \frac{2}{\lambda_{\text{max}}} \]

  where \( \lambda_{\text{max}} \) is the largest eigenvalue of the correl. matrix.

Linear Least-Square Filter: Example

See src/pseudoinv.m.

\[
X = \text{ceil}(\text{rand}(4, 2)*10), \ wtrue = \text{rand}(2, 1)*10, \ d = X \cdot wtrue, \ w = \text{inv}(X' \cdot X) \cdot X' \cdot d
\]

\[
X = \begin{bmatrix} 10 & 7 \\ 3 & 7 \\ 3 & 6 \\ 5 & 4 \end{bmatrix} \\
wtrue = \begin{bmatrix} 0.56644 \\ 4.99120 \end{bmatrix} \\
d = \begin{bmatrix} 40.603 \\ 36.638 \\ 31.647 \\ 22.797 \end{bmatrix} \\
w = \begin{bmatrix} 0.56644 \\ 4.99120 \end{bmatrix}
\]
Improving Convergence in LMS

- The main problem arises because of the fixed $\eta$.
- 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

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

The Perceptron Model

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

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

- 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?

Geometric Interpretation

- Rearranging
  \[ W_0 \times I_0 + W_1 \times I_1 - t > 0, \] 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

- $\vec{n} = (a, b, c)$, $\vec{x} = (x, y, z)$, $\vec{x}_0 = (x_0, y_0, z_0)$.
- 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

- 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)
Perceptron Learning Rule

- Given a linearly separable set of inputs that can belong to class $C_1$ or $C_2$.
- The goal of perceptron learning is to have
  \[ w^T x > 0 \text{ for all input in class } C_1 \]
  \[ w^T x \leq 0 \text{ for all input in class } C_2 \]
- If all inputs are correctly classified with the current weights $w(n)$,
  \[ w(n)^T x > 0, \text{ for all input in class } C_1, \text{ and} \]
  \[ w(n)^T x \leq 0, \text{ for all input in class } C_2, \]
  then $w(n + 1) = w(n)$ (no change).
- Otherwise, adjust the weights.

Perceptron Learning Rule (cont’d)

For misclassified inputs ($\eta(n)$ is the learning rate):
- $w(n + 1) = w(n) - \eta(n)x(n)$ if $w^T x > 0$ and $x \in C_2$.
- $w(n + 1) = w(n) + \eta(n)x(n)$ if $w^T x \leq 0$ and $x \in C_1$.
Or, simply $x(n + 1) = w(n) + \eta(n)e(n)x(n)$, where $e(n) = d(n) - y(n)$ (the error).

Perceptrons: A Different Perspective

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

So, if $\|x\| \cos \theta$ in the figure above is greater than $\frac{b}{\|w\|}$, then output = 1.
Adjusting $w$ changes the tilt of the decision boundary, and adjusting the bias $b$ (and $\|w\|$) moves the decision boundary closer or away from the origin.
Learning in Perceptron: Another Look

- When a positive example ($C_1$) is misclassified,
  \[ w(n + 1) = w(n) + \eta(n)x(n). \]

- When a negative example ($C_2$) is misclassified,
  \[ w(n + 1) = w(n) - \eta(n)x(n). \]

- Note the tilt in the weight vector, and observe how it would change the decision boundary.

Perceptron Convergence Theorem (cont’d)

- Using Cauchy-Schwartz inequality
  \[ \|w_0\|^2 \|w(n + 1)\|^2 \geq \left[w_0^T w(n + 1)\right]^2 \]

- From the above and $w_0^T w(n + 1) > n\alpha$,
  \[ \|w_0\|^2 \|w(n + 1)\|^2 \geq n^2 \alpha^2 \]

So, finally, we get
\[ \|w(n + 1)\|^2 \geq \frac{n^2 \alpha^2}{\|w_0\|^2} \]

First main result

Perceptron Convergence Theorem

- Given a set of linearly separable inputs, Without loss of generality, assume $\eta = 1$, $w(0) = 0$.
- Assume the first $n$ examples $\in C_1$ are all misclassified.
- Then, using $w(n + 1) = w(n) + x(n)$, we get
  \[ w(n + 1) = x(1) + x(2) + \ldots + x(n). \]  \hspace{1cm} (1)

- Since the input set is linearly separable, there is at least one solution $w_0$ such that $w_0^T x(n) > 0$ for all inputs in $C_1$.
  - Define $\alpha = \min_{x(n) \in C_1} w_0^T x(n) > 0$.
  - Multiply both sides in eq. 1 with $w_0$, we get:
    \[ w_0^T w(n + 1) = w_0^T x(1) + w_0^T x(2) + \ldots + w_0^T x(n). \]  \hspace{1cm} (2)

- From the two steps above, we get:
  \[ w_0^T w(n + 1) > n\alpha \]  \hspace{1cm} (3)

Perceptron Convergence Theorem (cont’d)

- Taking the Euclidean norm of $w(k + 1) = w(k) + x(k)$,
  \[ \|w(k + 1)\|^2 = \|w(k)\|^2 + 2w^T(k)x(k) + \|x(k)\|^2 \]

- Since all $n$ inputs in $C_1$ are misclassified, $w^T(k)x(k) \leq 0$ for $k = 1, 2, \ldots, n$,
  \[ \|w(k + 1)\|^2 - \|w(k)\|^2 - \|x(k)\|^2 = 2w^T(k)x(k) \leq 0, \]
  \[ \|w(k + 1)\|^2 \leq \|w(k)\|^2 + \|x(k)\|^2 \]
  \[ \|w(k + 1)\|^2 - \|w(k)\|^2 \leq \|x(k)\|^2 \]

- Summing up the inequalities for all $k = 1, 2, \ldots, n$, and $w(0) = 0$, we get
  \[ \|w(k + 1)\|^2 \leq \sum_{k=1}^{n} \|x(k)\|^2 \leq n\beta, \]  \hspace{1cm} (5)

where $\beta = \max_{x(k) \in C_1} \|x(k)\|^2$. 

37

38

39

40
Perceptron Convergence Theorem (cont’d)

• From eq. 4 and eq. 5,
\[
\frac{n^2 \alpha^2}{\|w_0\|^2} \leq \|w(n+1)\|^2 \leq n\beta
\]

• Here, \( \alpha \) is a constant, depending on the fixed input set and the fixed solution \( w_0 \) (so, \( \|w_0\| \) is also a constant), and \( \beta \) 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 become invalid (\( n \) is a positive integer).

• Thus, \( n \) cannot grow beyond a certain \( n_{\text{max}} \), where
\[
\frac{n_{\text{max}}^2 \alpha^2}{\|w_0\|^2} = n_{\text{max}} \beta
\]
\[
n_{\text{max}} = \frac{\beta \|w_0\|^2}{\alpha^2},
\]
and when \( n = n_{\text{max}} \), all inputs will be correctly classified.

Fixed-Increment Convergence Theorem

Let the subsets of training vectors \( C_1 \) and \( C_2 \) 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
\[
w(n_0) = w(n_0 + 1) = w(n_0 + 2) = \ldots
\]
is a solution vector for \( n_0 \leq n_{\text{max}} \).

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?
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.