Associative Memory

Lecture 4, COMP / ELEC / STAT 502
© E. Merényi

Inception: January 19, 2012 Last updated:January 23, 2014[1]

[1] Sources:
Colin Fyfe, Artificial Neural Networks and Informaton Theory. Course notes, Dept. of Computing and Information Systems, The University of Paisley, 2000
Bobby R. Hunt, Course notes, ECE 631, University of Arizona, 1990.

Outline

Thoughts About Memory

Intelligence, Learning, Pattern Recognition $\cdots$ are related to memory. (But remember: learning $\neq$ memorizing.) What is memory?

- Device (in which to store information)
- Content (that is stored)
- Process (of retrieving or recalling what is stored): "my memory is that$\cdots$" ("as I remember$\cdots$")

How does memory "occur", get created?

- Man made memory device
- Natural memory

We cannot remember an event before it occurred. While the hardware is there in both cases, the content is created by some event, through some encoding process.

Types of Memory

Memories can be

- *Autoassociative*: recall the same pattern as the input
  — especially good for recognizing partial or very noisy patterns

  or

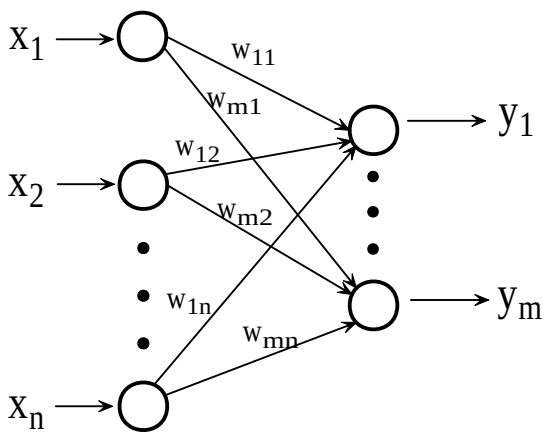
- *Heteroassociative*: recall a pattern different from the input but associated with it

An Associative Memory ANN is a simple model of creating memory.

Heteroassociation

Consider the topology at right, and $\mathbf{x}_k, \mathbf{y}_k$ pattern pairs, $k = 1, \ldots, P$.

$$\mathbf{x}_k = (x_{k1}, \ldots, x_{kn})^T$$
$$\mathbf{y}_k = (y_{k1}, \ldots, y_{km})^T$$
$$\mathbf{W} = \{w_{ij}\} \quad \text{weight matrix}$$



Note: For this part, we will use the pattern index k at the lower right for consistency with the CF notes.

Goal: $\mathbf{y}_k = \mathbf{W}(k)\mathbf{x}_k \quad \forall k$

$$\begin{bmatrix} y_{k1} \\ y_{k2} \\ \vdots \\ y_{km} \end{bmatrix} = \underbrace{\begin{bmatrix} w_{11}(k) & w_{12}(k) & \cdots & w_{1n}(k) \\ w_{21}(k) & w_{22}(k) & \cdots & w_{2n}(k) \\ \vdots & \vdots & \vdots & \vdots \\ w_{m1}(k) & w_{m2}(k) & \cdots & w_{mn}(k) \end{bmatrix}}_{\mathbf{W}(k)} \begin{bmatrix} x_{k1} \\ x_{k2} \\ \vdots \\ x_{kn} \end{bmatrix} \tag{4.1}$$

Construction of Memory

Proposition: To recall P distinct patterns, construct a memory $\mathbf{M}$ as

$$\mathbf{M} = \mathbf{M}(P) = \sum_{k=1}^{P} \mathbf{W}(k) \tag{4.2}$$

where $\mathbf{W}(k)$ recalls one specific pattern, $\mathbf{x}_k$.

$$\mathbf{M}(k) = \mathbf{M}(k-1) + \mathbf{W}(k) \quad \text{with } \mathbf{M}(0) = \mathbf{0} \tag{4.3}$$

Note that using $\mathbf{W}^T$ memory can be created to recall $\mathbf{x}_k$ from $\mathbf{y}_k$.

How do we create the memory matrix $\mathbf{W}(k)$?

Hebb's Rule

Hebb's postulate (Donald Hebb, 1949)

When a neuron A repeatedly and persistently takes part in exciting (firing) neuron B, some growth process or metabolic change occurs in one or both A and B such that A's efficiency (in firing B) is increased $\Longrightarrow$ the synaptic efficiency between A and B increases.

Hebb's learning rule: $\boxed{w \propto AB}$

The weight between Processing Elements A and B is increased proportional to the responses of A and B.

Hebb's rule made a fundamental impact on the development of ANNs ...

Construction of Memory With Hebb's Rule

Using *Hebb's learning rule*:

$$\begin{aligned}
\mathbf{W}(k) &= \mathbf{y}_k \mathbf{x}_k^T \\
&= \begin{bmatrix} y_{k1} \\ y_{k2} \\ \vdots \\ y_{km} \end{bmatrix} \begin{bmatrix} x_{k1} & x_{k2} & \cdots & x_{kn} \end{bmatrix} \\
&= \begin{bmatrix} y_{k1}x_{k1} & y_{k1}x_{k2} & \cdots & y_{k1}x_{kn} \\ y_{k2}x_{k1} & y_{k2}x_{k2} & \cdots & y_{k2}x_{kn} \\ \vdots & \vdots & \vdots & \vdots \\ y_{km}x_{k1} & y_{km}x_{k2} & \cdots & y_{km}x_{kn} \end{bmatrix}
\end{aligned} \tag{4.4}$$

This is also called "outer product rule", or "correlation matrix rule".

Using this $\mathbf{W}(k)$, and assuming $\| \mathbf{x}_k \| = \mathbf{x}_k^T \mathbf{x}_k = 1$

$$\mathbf{y} = \mathbf{W}(k)\mathbf{x}_k = \mathbf{y}_k \mathbf{x}_k^T \mathbf{x}_k = \mathbf{y}_k$$

The k^{th} input pattern is perfectly *recalled*.

For non-normalized inputs, the output vector would still have the same direction as $\mathbf{y}_k$ but scaled by the squared

Crosstalk

When adding more patterns to the memory (P increases), patterns can interfere with one another. Let again

$$\mathbf{M} = \mathbf{M}(P) = \sum_{k=1}^{P} \mathbf{W}(k)$$

$\mathbf{x}_i = (x_{i1}, \ldots, x_{in})^T$ i^{th} input pattern

$\mathbf{y}_i = (y_{i1}, \ldots, y_{im})^T$ desired response to $\mathbf{x}_i$

$\mathbf{y} = (y_1, \ldots, y_m)^T$ actual response to $\mathbf{x}_i$

Then

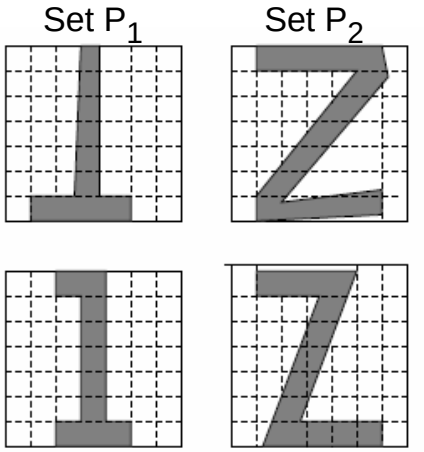
$$\mathbf{y} = \mathbf{M}\mathbf{x}_i = \sum_{k=1}^{P} \mathbf{W}(k)\mathbf{x}_i = \sum_{k=1}^{P} \mathbf{y}_k \mathbf{x}_k^T \mathbf{x}_i = \mathbf{y}_i \underbrace{\mathbf{x}_i^T \mathbf{x}_i}_{\| \mathbf{x}_i \| = 1} + \underbrace{\sum_{k \neq i} \mathbf{y}_k \mathbf{x}_k^T \mathbf{x}_i}_{\text{crosstalk}}$$

$$= \mathbf{y}_i + \mathbf{c}_i$$

Since $\mathbf{x}_k^T \mathbf{x}_i = cos(\angle(\mathbf{x}_k \mathbf{x}_i))$, the crosstalk is minimum ($= 0$) when the input patterns form an orthonormal set. Memory *capacity* is limited.

Using orthonormal patterns is too wasteful ... $\Longrightarrow$ consider BAM

Crosstalk Can Be Useful - an Example



Suppose you have sets of training patterns P_1 and P_2 of the characters "1" and "2", represented by binary 49×1 vectors. Encode the desired responses as

$$\mathbf{y}_1 = (1, 0) \quad \text{for any "1"}$$
$$\mathbf{y}_2 = (0, 1) \quad \text{for any "2"}$$

Then, for any pattern "2", for example,

$$\mathbf{W}(k) = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \underbrace{\begin{bmatrix} x_{k1} & \cdots & x_{kn} \end{bmatrix}}_{\text{any pattern "2"}} = \underbrace{\begin{bmatrix} 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 1 & \cdots & 1 & 0 \end{bmatrix}}_{\text{for example (same pattern "2")}}$$

$$\mathbf{M} = \sum_{k=1}^{P} \mathbf{W}(k) = \begin{bmatrix} \sum_{\mathbf{x}_k \in P_1} \mathbf{x}_k^T \\ \sum_{\mathbf{x}_k \in P_2} \mathbf{x}_k^T \end{bmatrix} \tag{4.5}$$

Crosstalk Can Be Useful - an Example

- Since $\mathbf{y}_1$ and $\mathbf{y}_2$ are orthogonal, the sets of "ones" and "twos" do not interfere.
- However, the memories of all the "ones" are interfering with each other. Similarly, the memories of all the "twos" are interfering with each other. The rows of the memory matrix are formed by the crosstalk between patterns of the same type, and each row is proportional to the mean of the training patterns of the respective type.
- In this case the crosstalk is helpful in forming an idealized, typical version of the pattern "1" and of "2" (as learned from the training data).
- This is one way of storing groups of many interfering patterns, and retrieving the idealized, prototypical version.

Recall of Noisy Patterns

With the memory M created from the example (training) patterns, we want to recognize (recall) patterns similar to the training examples but not exactly the same as any of them.

For simplicity, assume now $\mathbf{x}_k = \mathbf{y}_k$, i.e., autoassociation[2], and $\mathbf{x}_k$ are orthonormal. (I.e., no cross-talk, and M is an *nxn* square matrix.)

Recall a *noisy pattern* $\hat{\mathbf{x}}_k = \mathbf{x}_k + \underbrace{\boldsymbol{\mu}_k}_{\text{noise}}$:

$$\mathbf{y} = \mathbf{M}\hat{\mathbf{x}}_k = \mathbf{M}\mathbf{x}_k + \mathbf{M}\mu_k = \underbrace{\mathbf{y}_k}_{\text{desired}} + \underbrace{\mathbf{M}\mu_k}_{\text{noise remained}} \tag{4.6}$$

The noise is not cleaned — a result of *linear* memory (ANN with linear transfer function). What to do?

[2] Autoassociation is a special case of heteroassociation.

Recall of Noisy Patterns

Try passing the noisy signal back and forth ...

$$\begin{aligned}
\mathbf{y}_k(0) &= \mathbf{M}\hat{\mathbf{x}}_k(0) \text{ forward pass} \\
\hat{\mathbf{x}}_k(1) &= \mathbf{M}^T\mathbf{y}_k(0) \text{ backward pass} \\
\mathbf{y}_k(1) &= \mathbf{M}\hat{\mathbf{x}}_k(1) \text{ forward pass} \\
\hat{\mathbf{x}}_k(2) &= \mathbf{M}^T\mathbf{y}_k(1) \text{ backward pass} \\
&\vdots \\
\mathbf{y}_k(t-1) &= \mathbf{M}\hat{\mathbf{x}}_k(t-1) \text{ forward pass} \\
\hat{\mathbf{x}}_k(t) &= \mathbf{M}^T\mathbf{y}_k(t-1) \text{ backward pass} \qquad (4.7)
\end{aligned}$$

Still linear, we gained nothing:

$$\hat{\mathbf{x}}_k(1) = \mathbf{M}^T\mathbf{y}_k(0) = \underbrace{\mathbf{M}^T\mathbf{M}}_{=\mathbf{I}}\hat{\mathbf{x}}_k(0) = \hat{\mathbf{x}}_k(0)$$

Similarly, $\mathbf{y}_k(1) = \mathbf{y}_k(0)$, ...

Recall of Noisy Patterns

But, introduce a non-linear transfer function, for example a simple threshold:

$$\begin{aligned} y_{k,j} &= 1, \text{ if } \mathbf{M}_j.\hat{\mathbf{x}}_k > 0 \\ y_{k,j} &= -1, \text{ if } \mathbf{M}_j.\hat{\mathbf{x}}_k < 0 \\ \hat{x}_{k,i} &= 1, \text{ if } \mathbf{M}_i^T.\mathbf{y}_k > 0 \\ \hat{x}_{k,i} &= -1, \text{ if } \mathbf{M}_i^T.\mathbf{y}_k < 0 \end{aligned} \tag{4.8}$$

where $y_{k,j}$ is the jth element of $\mathbf{y}_k$, $\mathbf{M}_j$ is the jth row of $\mathbf{M}$, $\mathbf{M}_i^T$ is the ith row of $\mathbf{M}^T$.

We will show that with this, the BAM is bidirectionally stable, i.e., the process (4.7) converges to a stable solution (the desired output) for every input.

Convergence and Stability of the BAM

The proof lies in finding a *Lyapunov function*. The proof will neither tell *what* the solution is, nor how to find a Lyapunov function. (But often it is not too difficult to find one.)

Lyapunov functions can be used for convergence and stability evaluation of non-linear dynamic systems / processes.

We will apply Lyapunov functions to ANN learning.

Lyapunov Function

Let $x_i(t)$ be time dependent variables whose values contain sufficient information at time t to predict the future state of a dynamic system. $x_i(t)$ are the *state variables* of the system. (Example: the weights of an ANN are state variables.)

$\mathbf{x}(t) = (x_1(t), \ldots, x_n(t))$ is the *state vector*
$\mathbf{x}^* = (x_1^*, \ldots, x_n^*)$ denotes *equilibrium state*
$\mathbf{x}(t), \mathbf{x}^* \in \mathcal{L}$, where $\mathcal{L}$ is *the state-space*

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{F}(\mathbf{x}(t)) \quad \text{state-space equation} \tag{4.9}$$

$\mathbf{F}$ is a vector field, (4.9) describes a motion in n-dimensional space.
$\mathbf{F}(\mathbf{x}^*) = 0$, i.e., the velocity (change) vanishes at equilibrium.

Lyapunov Function

A system of 1st order differential equations s.a. eq. (4.9) can describe a large class of non-linear dynamical systems.

$\mathbf{F}(\mathbf{x}(t))$ is vector-valued, and non-linear, in general. (Example: the learning rule of an ANN is a state-space equation.)

The dynamical system is *autonomous* if $\mathbf{F}$ does not depend explicitly on time t. ($\mathbf{x}(t)$ varies with time $\Longrightarrow$ $\mathbf{F}$ implicitly depends on time.) This is our interest. ($\mathbf{F}(\mathbf{x}(\mathrm{t}))$, not $\mathbf{F}(\mathrm{t},\mathbf{x}(\mathrm{t}))$.)

The state-space equation of an ANN — the learning rule, $\mathbf{L}$ — depends implicitly on time t. For example, in the case of the simple perceptron:

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \alpha(y^{target} - y)\mathbf{x} \text{ or}$$

$$\frac{d}{dt}\mathbf{w}(t) = \mathbf{L}(\mathbf{w}(t), \alpha, \mathbf{x}, y) \tag{4.10}$$

Eq. (4.10) $= 0$ when weights stop changing, i.e., $y^{target} \approx y$ for all training inputs.

Lyapunov Theorems

Theorem 1 of Lyapunov:

The equilibrium state $\mathbf{x}^*$ is stable if in a small neighborhood of $\mathbf{x}^*$ there exists a *positive definite function* of the state variables, $f(\mathbf{x}(t))$, such that its derivative wrt time t is negative semidefinite ($f'(\mathbf{x}(t)) \leq 0$ for all $\mathbf{x}$) in that small region.

A small neighborhood of $\mathbf{x}^*$ is the set of all points $\mathbf{x}$ for which $|\mathbf{x} - \mathbf{x}^*| < \boldsymbol{\delta}$ where $\boldsymbol{\delta}$ is the radius of the neighborhood.

Theorem 2 of Lyapunov:

The equilibrium state $\mathbf{x}^*$ is *asymptotically stable* if in a small neighborhood of $\mathbf{x}^*$ there exists a *positive definite function* $f(\mathbf{x}(t))$ such that its derivative wrt time t is negative definite ($f'(\mathbf{x}(t)) < 0$ for all $\mathbf{x}$) in that small region.

Lyapunov Theorems

Definition: $f(\mathbf{x}(t))$ is a positive definite function on the state-space $\mathcal{L}$ if for $\forall \mathbf{x} \in \mathcal{L}$

1. $f(\mathbf{x}(t))$ has continuous partial derivatives wrt the elements of $\mathbf{x}$
2. $f(\mathbf{x}(t)) \geq 0$ with equality iff $\mathbf{x} = \mathbf{x}^*$

A scalar function $f(\mathbf{x}(t))$ that satisfies the requirements of the Lyapunov theorem(s) is called a *Lyapunov function* for the equilibrium state $\mathbf{x}^*$.

If $f(\mathbf{x}(t))$ is a L-function for an existing solution (equilibrium state) $\mathbf{x}^*$, then according to L's Theorem 1 the equilibrium state $\mathbf{x}^*$ is stable if $\frac{d}{dt} f(\mathbf{x}(t)) \leq 0$ for $|\mathbf{x} - \mathbf{x}^*| < \delta$ (similarly, L's Theorem 2 for asymptotic stability).

The point is: Showing the existence of a L-function in a neighborhood is sufficient to prove stability / convergence of the dynamical process in that neighborhood. The L-theorem(s) can be applied without knowing the solution of the state-space equation.

Background: Definitions of Stability and Convergence

The equilibrium state $\mathbf{x}^*$ is *uniformly stable* if for any $\epsilon > 0$ there exists a $\delta > 0$ such that if $\| \mathbf{x}(0) - \mathbf{x}^* \| < \delta$ then $\| \mathbf{x}(t) - \mathbf{x}^* \| < \epsilon$ for all $t > 0$. (The trajectory of the system will stay within a small neighborhood of the equilibrium state if the initial state is close to $\mathbf{x}^*$.)

The equilibrium state $\mathbf{x}^*$ is *convergent* if there exists a $\delta > 0$ such that if $\| \mathbf{x}(0) - \mathbf{x}^* \| < \delta$ then $\mathbf{x}(t) \to \mathbf{x}^*$ as $t \to \infty$. (The trajectory will get closer and closer to the equilibrium in time if the initial state is close to $\mathbf{x}^*$.)

The equilibrium state $\mathbf{x}^*$ is *asymptotically stable* if it is both stable and convergent.

If $\mathbf{F}(\mathbf{x}(t))$ is continuous in all its variables an equilibrium state (solution) $\mathbf{x}^*$ exists but it is not necessarily unique. For $\mathbf{x}^*$ to be a unique solution, the Lipschitz condition must hold: there must exist a C constant s.t.

$$||\mathbf{F}(\mathbf{x}) - \mathbf{F}(\mathbf{u})||_2 \leq C||\mathbf{x} - \mathbf{u}|| \qquad \forall \mathbf{x}, \mathbf{u} \in \mathcal{M}_\delta \subset \mathcal{L}$$

The Lipschitz condition holds if $\frac{\partial F_i}{\partial x_j} < \infty \quad \forall i, j$.

Lyapunov's Direct Method

To show that a dynamical system is convergent / stable,

1. Find a scalar function $f(\mathbf{x}(t))$ of the state variables $\mathbf{x}(t)$ which is continuously differentiable wrt the state variables.
2. Show that $f(\mathbf{x}(t))$ is bounded below (above) if it decreases (increases).
3. Show that the time derivative of $f(\mathbf{x}(t))$ is negative (positive), i.e., $f(\mathbf{x}(t))$ changes monotonously every time the state variables change.

2. and 3. replace (equivalently) the requirement of positive (negative) definiteness, for practical use.

Note 1: The L-function is a sufficient but not necessary condition for convergence / stability!

Note 2: There is no recipe for constructing a L-function. Defining a "small" neighborhood is also an issue. However, in many cases the energy (or squared error) function of the system can serve as a L-function, for example.

Translation To ANN Weight Training

State vector: $\mathbf{w}(t) = (w_1(t), \ldots, w_n(t))$, all weights in the ANN

Equilibrium state (solution): $\mathbf{w}^* = (w_1^*, \ldots, w_n^*)$ where the weights do not change anymore (training converged)

State-space equation: $\frac{d}{dt}\mathbf{w}(t) = \mathbf{L}(\mathbf{w}(t))$, the learning rule of the ANN, prescribes how to update the weights.

To prove convergence of the learning with Lyapunov's direct method,

- A. Find a positive (negative) definite, scalar function $f(\mathbf{w}(t))$ (or, equivalently, $f(\mathbf{w}(t))$ is bounded from below (above)), which is continuously differentiable wrt $w_i(t), i = 1, \cdots, n$.
- B. Show that $f_t'(\mathbf{w}(t)) \leq (\geq)\ 0$, assuming that we are close to $\mathbf{w}^*$ ($\| \mathbf{w} - \mathbf{w}^* \|$ is small).

A + B proves convergence (or stability).

Lyapunov Function For the BAM

To prove convergence (eq.(4.8)) of the BAM, consider the function

$$\begin{aligned} E(\mathbf{x},\mathbf{y}) &= -\mathbf{x}^T\mathbf{M}^T\mathbf{y}, \qquad x_i, y_j \in \{-1,1\} \\ &= -\sum_i \sum_j x_i y_j m_{ij} \end{aligned}$$

Here, $\mathbf{x} = (x_1 \cdots, x_i, \cdots, x_n)$ and $\mathbf{y} = (y_1 \cdots, y_j, \cdots, y_m)$ are the state variables!

Is this a Lyapunov function for the BAM?

- $E(\mathbf{x},\mathbf{y})$ scalar, continuously differentiable function of the $x_i(t), y_j(t)$.
- $E(\mathbf{x},\mathbf{y})$ is bounded below since $E(\mathbf{x},\mathbf{y}) \geq -\sum_i \sum_j |m_{ij}|$.
- Its time derivative is non-increasing: as shown below, $E(\mathbf{x},\mathbf{y})$ decreases (or stays unchanged) every time the system changes (i.e., the state variables change).

Lyapunov Function For the BAM

Let Δx_i be the change at the i^{th} input neuron due to the feedback from the previous output ($\mathbf{y}$) neurons. Then, at any iteration, by (4.8):

$$\Delta x_i = \begin{cases} 2 & \text{if } -1 \to 1 \Longrightarrow \mathbf{M}_i^T.\mathbf{y} > 0 \\ -2 & \text{if } 1 \to -1 \Longrightarrow \mathbf{M}_i^T.\mathbf{y} < 0 \\ 0 & \text{if } 1 \to 1 \text{ or } -1 \to -1 \end{cases} \tag{4.11}$$

and thus

$$\Delta E(\mathbf{x}, \mathbf{y}) = -\Delta \mathbf{x}^T \mathbf{M}^T \mathbf{y} = -\sum_i \Delta x_i \sum_j y_j m_{ij} \leq 0 \tag{4.12}$$

If $\sum_j y_j m_{ij} > 0$ then $\Delta \mathbf{x}_i > 0$, and so $\Delta E(\mathbf{x}_i, \mathbf{y}) < 0$. Similarly, if $\sum_j y_j m_{ij} < 0$ then $\Delta \mathbf{x}_i < 0$, and so $\Delta E(\mathbf{x}_i, \mathbf{y}) < 0$. If $\Delta \mathbf{x}_i = 0$ the sign of $\sum_j y_j m_{ij} < 0$ is irrelevant, the respective partial sum will not contribute to the change of E. A similar argument applies to change at the output neurons. So, any change decreases E. Since E is bounded below the network must stabilize.

Note: Eq. (4.12) in CF 1/2, page 27, has a typo in it.

Convergence Of the BAM

Note that

- Proving convergence of the process (BAM recall, in this example) does not guarantee convergence to the right solution!
- The existence of a Lyapunov function guarantees convergence to, or stabilizing around, *one of the stable states*, i.e., one of the solutions, one of the target patterns $\mathbf{y}_k$.
- The process will converge to, or stabilize around, a solution (retrieve that pattern), for which $E(\mathbf{x}, \mathbf{y})$ is a Lyapunov function in a small neighborhood and which is closest at the start of the process.

Error in Recall

Previously we saw an approach to correct the outside noise during recall. Now we look at crosstalk and outside noise together. Consider the memory (now back to our notation, with pattern index k at upper right):

$$\mathbf{M} = \sum_{k=1}^{P} \mathbf{y}^k \mathbf{x}^{k\,T}$$

and a noisy input pattern $\hat{\mathbf{x}}^l = \mathbf{x}^l + \underbrace{\boldsymbol{\mu}}_{\text{noise}}$ to recall by $\mathbf{M}$.

$$\mathbf{y} = \mathbf{M}\hat{\mathbf{x}}^l = \mathbf{M}\mathbf{x}^l + \mathbf{M}\boldsymbol{\mu} = \underbrace{\mathbf{y}^l \underbrace{\mathbf{x}^{l\,T}\mathbf{x}^l}_{\| \mathbf{x}^l \|}}_{\boxed{\textit{target response}}} + \underbrace{\sum_{k \neq l}^{P} \mathbf{y}^k \mathbf{x}^{k\,T} \mathbf{x}^l}_{\boxed{\textit{crosstalk}}} + \underbrace{\sum_{k=1}^{P} \mathbf{y}^k \mathbf{x}^{k\,T} \boldsymbol{\mu}}_{\boxed{\textit{outside noise}}} \tag{4.13}$$

The crosstalk is caused by $P > n$ or by non-orthogonal set of training patterns.

A Note On Correction Of Crosstalk

- Crosstalk can be avoided by preprocessing, under some circumstances. For example, if the $\{\mathbf{x}^k\}$ patterns are linearly independent, they can be made an orthonormal set by standard orthogonalization algorithms (e.g., Graham-Schmidt).
- Other preprocessing that *approximates* orthonormality, can be *high-pass* filtering: basic low-frequency features of faces, for example, tend to be very similar, while the high-frequency components are more unique and therefore more independent.
- However, such preprocessing is data dependent, therefore potentially requiring human expertese for useful results, and can be tedious.
- A smarter approach: *error correction built into* the memory construction. This suits real-life situations better, and deals with crosstalk and other error.

Optimal Linear Associative Memory (OLAM) - a Classic

Let $\{\mathbf{x}^k\}$ be a set of non-orthogonal but linearly independent patterns (column vectors), $k = 1, \ldots, P$. $n \neq m$, in general.

$$\mathbf{X}: \{\mathbf{x}^1, \ldots, \mathbf{x}^P\} \quad \text{the input pattern matrix,} \qquad \mathbf{x}^k = (x_1^k, \ldots, x_n^k)^T$$

$$\mathbf{Y}: \{\mathbf{y}^1, \ldots, \mathbf{y}^P\} \quad \text{the output pattern matrix,} \qquad \mathbf{y}^k = (y_1^k, \ldots, y_m^k)^T$$

$$\underset{\mathbf{Y}}{\begin{bmatrix} y_1^1 & \cdots & y_1^P \\ \vdots & & \\ y_m^1 & \cdots & y_m^P \end{bmatrix}} \underset{\mathbf{X^T}}{\begin{bmatrix} x_1^1 & \cdots & x_n^1 \\ \vdots & & \\ x_1^P & \cdots & x_n^P \end{bmatrix}} = \underset{\mathbf{M}}{\begin{bmatrix} \sum_{k=1}^P y_1^k x_1^k & \cdots & \sum_{k=1}^P y_1^k x_n^k \\ \vdots & & \\ \sum_{k=1}^P y_m^k x_1^k & \cdots & \sum_{k=1}^P y_m^k x_n^k \end{bmatrix}}$$

$$\mathbf{M} = \sum_{k=1}^{P} \mathbf{y}^k \mathbf{x}^{k\,T} \quad \text{or} \quad \mathbf{M} = \mathbf{Y}\mathbf{X}^T \text{ memory matrix} \tag{4.14}$$

Definition of Recall Error and OLAM

$$\mathbf{M} = \sum_{k=1}^{P} \mathbf{y}^k \mathbf{x}^{k\,T} \quad \text{or} \quad \mathbf{M} = \mathbf{Y}\mathbf{X}^T \text{ memory matrix}$$

$$\hat{\mathbf{y}}^k = \mathbf{M}\mathbf{x}^k \quad \text{actual response to input pattern } \mathbf{x}^k$$

Error of recall (vector / matrix):

$$\mathbf{e}^k = \hat{\mathbf{y}}^k - \mathbf{y}^k \quad \text{or} \quad \mathbf{E} = \mathbf{M}\mathbf{X} - \mathbf{Y} \quad \text{error of recall} \tag{4.15}$$

Total recall error (a scalar quantity, sum of squared errors):

$$E = E(\mathbf{M}) = \| \mathbf{E} \|_2^2 = \sum_{k=1}^{P} \sum_{j=1}^{m} (e_j^k)^2 = \sum_{k=1}^{P} \| \mathbf{e}^k \|_2^2 = \sum_{k=1}^{P} E^k \tag{4.16}$$

A Linear Associative Memory is an OLAM if it minimizes the total recall error E.

OLAM, cont'd

Goal: find the Optimal Linear Associative Memory $\mathbf{M}_o$ that minimizes the error in the Least Square sense, as defined by eq. (4.16).
Fact: It can be shown that the minimum exists and that

$$\mathbf{M}_o = \mathbf{Y}\mathbf{X}^+ \tag{4.17}$$

where $\mathbf{X}^+$ is the *pseudoinverse* of $\mathbf{X}$ (since $\mathbf{M}$ is non-square). $\mathbf{X}^+$ is guaranteed to exist and can be computed as

$$\mathbf{X}^+ = [\mathbf{X}^T\mathbf{X}]^{-1}\mathbf{X}^T \quad \text{if columns of } \mathbf{X} \text{ are independent}$$
$$\mathbf{X}^+ = \mathbf{X}^T[\mathbf{X}\mathbf{X}^T]^{-1} \quad \text{if rows of } \mathbf{X} \text{ are independent}$$

(The proof takes a small course in linear algebra - will not do it here.)

With this approach one does not need to worry about orthogonality of the patterns.

OLAM, cont'd

However, $\mathbf{X}^+$ can be difficult to compute numerically ...

An alternative is to learn $\mathbf{M}$ iteratively — build in the error correction step by step, "on-line". The OLAM above was computed in "batch" mode. To add any new pattern would require to repeat the entire calculation!

This is where learning comes in.

So far, the weight matrix $W(k)$ (and $\mathbf{M} = \sum_{k=1}^{P} W(k)$) was *set* in one step, resulting in a non-ideal memory ... Now we will *learn* $\mathbf{M}_o$.

Next we derive a learning rule that is guaranteed to converge.

Learning Error Correction in the BAM

Let $\hat{\mathbf{M}}(t)$ be the approximation of $\mathbf{M}_o$ at time t; $\hat{\mathbf{M}}(0) = \mathbf{0}$
$\mathbf{e}^k(t) = \mathbf{y}^k - \hat{\mathbf{M}}(t)\mathbf{x}^k$ error at time t in response to $\mathbf{x}^k$ (Omitting t for clarity) learn $\hat{\mathbf{M}}$ s. t. it minimizes

$$E^k(\hat{\mathbf{M}}) = \frac{1}{2} \parallel \mathbf{e}^k \parallel_2^2 \quad \forall k, \quad \text{since} \sum_{k=1}^{P} E^k = \frac{1}{2}E \quad \text{total error}$$

Proposition: Use the *steepest gradient descent* as learning rule,

$$\hat{\mathbf{M}}(t+1) = \hat{\mathbf{M}}(t) - \alpha \nabla_{\hat{\mathbf{M}}} E^k(\hat{\mathbf{M}}) \quad 0 < \alpha < 1 \tag{4.18}$$

$$\underbrace{\hat{\mathbf{M}}(t+1) - \hat{\mathbf{M}}(t)}_{\text{system change}} = -\alpha \nabla_{\hat{\mathbf{M}}} E^k(\hat{\mathbf{M}}) \tag{4.19}$$

The system change is always in the direction opposite to the direction of the error gradient.

Learning Error Correction in the BAM

Compute the RHS of eq. (4.19):

$$E^k(\hat{\mathbf{M}}) = \frac{1}{2} \parallel \mathbf{e}^k \parallel_2^2 = \frac{1}{2}(\mathbf{y}^k - \hat{\mathbf{M}}\mathbf{x}^k)^T(\mathbf{y}^k - \hat{\mathbf{M}}\mathbf{x}^k)$$

$$\nabla_{\hat{\mathbf{M}}} E^k(\hat{\mathbf{M}}) = 2\frac{1}{2}(\mathbf{y}^k - \hat{\mathbf{M}}\mathbf{x}^k)(-\mathbf{x}^k)^T = \underbrace{-(\mathbf{y}^k - \hat{\mathbf{M}}\mathbf{x}^k)}_{-\mathbf{e}^k(t)}\mathbf{x}^{kT} \qquad (4.20)$$

$$= -(\mathbf{y}^k\mathbf{x}^{kT} - \underbrace{\hat{\mathbf{M}}\mathbf{x}^k\mathbf{x}^{kT}}_{feedback\ \ term}) \qquad \text{or, in this form we see the feedback}$$

Update the memory as in eq. (4.19):

$$\hat{\mathbf{M}}(t+1) - \hat{\mathbf{M}}(t) = \alpha \underbrace{[\mathbf{y}^k - \hat{\mathbf{M}}(t)\mathbf{x}^k]}_{\mathbf{e}^k(t)}\mathbf{x}^{kT} \qquad (4.21)$$

Do this for a randomly drawn input pattern $\mathbf{x}^k$ at time t, then increase t ($t = t + 1$). Continue until all errors ($\mathbf{e}^k$ or E^k) are acceptable.

Learning Error Correction in the BAM

This learning process converges because E^k is a Lyapunov function:

a) E^k is a scalar function of the state variables (the weights, the elements m_{ij} of $\mathbf{M}$).

b) E^k is positive definite: $\forall \mathbf{M}, m_{ij} \in \mathcal{R}, E^k(\mathbf{M}) = \| \mathbf{e}^k \|_2^2 \geq 0$ with equality iff $\mathbf{M} = \mathbf{M_o}$.

c) E^k has continuous derivatives wrt the state variables.

d) The derivative of E^k wrt time t is negative semidefinite, as shown next page.

Learning Error Correction in the BAM

The derivative of E^k wrt time t is negative semidefinite:

$$\frac{dE^k}{dt} = \sum_{i=1}^{n}\sum_{j=1}^{m} \frac{\partial E^k}{\partial \hat{\mathbf{M}}_{ij}} \frac{d\hat{\mathbf{M}}_{ij}}{dt} = \sum_{i=1}^{n}\sum_{j=1}^{m} \left[\nabla_{\hat{\mathbf{M}}} E^k(\hat{\mathbf{M}})\right]_{ij} \left[\frac{d\hat{\mathbf{M}}}{dt}\right]_{ij} \tag{4.22}$$

$$\approx \sum_{i=1}^{n}\sum_{j=1}^{m} \underbrace{\left[-\mathbf{e}^k(t)\mathbf{x}^{k\,T}\right]_{ij}}_{\text{from (4.20)}} \alpha \underbrace{\left[\mathbf{e}^k(t)\mathbf{x}^{k\,T}\right]_{ij}}_{\text{from (4.21)}}$$

$$= \underbrace{-\alpha \sum_{i=1}^{n}\sum_{j=1}^{m} \underbrace{\left[\mathbf{e}^k(t)\mathbf{x}^{k\,T}\right]_{ij} \left[\mathbf{e}^k(t)\mathbf{x}^{k\,T}\right]_{ij}}_{\geq 0}}_{\leq 0} \tag{4.23}$$

Caution! α should be "appropriate"!!

MATLAB Code

The MATLAB function to implement the recursive error-correction algorithm that constructs the correlation matrix memory (or matrix associative memory) is posted in file errcorr.m at http://terra.ece.rice.edu/ANNclass502.

Analyze this code to make sure you understand, and do the following exercises.

Data and Exercises

For a simple exercise, try the errcorr.m MATLAB code for the following:

Case A) $P = 3$

$$\mathbf{X} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{and } \mathbf{Y} = \begin{bmatrix} 2 & 1 & -2 \\ -2 & 1 & 4 \\ 3 & -2 & 2 \end{bmatrix} \tag{4.24}$$

I trained with this data set for 3000 iterations. The recall was perfect.

Case B) $P = 4$

$$\mathbf{X} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0.8 \\ 0 & 0 & 1 & 0.2 \end{bmatrix} \quad \text{and } \mathbf{Y} = \begin{bmatrix} 2 & 1 & -2 & 1 \\ -2 & 1 & 4 & 2 \\ 3 & -2 & 2 & 1 \end{bmatrix} \tag{4.25}$$

With this augmentation, the original data are no longer orthogonal. Yet, after 4000 training step, perfect recall was achieved.

Data and Exercises

Try the following:
Let $\mathbf{x^1} = [\mathbf{1}, \mathbf{0}, \mathbf{0}, \mathbf{0}]$, $\mathbf{x^2} = [\mathbf{0}, \mathbf{1}, \mathbf{0}, \mathbf{0}]$ and $\mathbf{x^3} = [\mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{1}]$ input patterns, and $\mathbf{y^1} = [\mathbf{2}, -\mathbf{2}, \mathbf{3}]$, $\mathbf{y^2} = [\mathbf{1}, \mathbf{1}, -\mathbf{2}]$, and $\mathbf{y^3} = [-\mathbf{2}, \mathbf{4}, \mathbf{2}]$ the desired reponse patterns (as on the previous page in Case A).

1. First, construct an associative memory matrix using the outer product rule, and check if that memory produces perfect recall.
2. Second, use the above iterative error-correction procedure to build the memory matrix, for the same pattern associations. Compare the two memory matrices: how much do they differ after n iterations? Suggested parameters for the iterative procedure: mu = 0.1, n=1000, tol = 1e-8.
3. Third, add another input pattern that is not orthogonal to the first three, as in Case B) on the previous page. Repeat steps 1) and 2).

Data and Exercises

A more realistic data set is in the file characters.ascii, posted at http://terra.ece.rice.edu/ANNclass502

- 38 lines of ascii data, 1 line per pattern
- Each line contains one character, represented by 42 binary numbers similarly to the "1" and "2" characters in the previous lecture. Here, each character is a 7 rows x 6 columns matrix. Each pixel has a value of -1 or 1.
- One 42-element vector represents each of the 28 letters, and the ten digits.

a) Create an Associative Memory with the outer product rule and see how good the recall is.

b) Learn an Associative Memory with the error correction iterative algorithm and see how good the recall is in this case.

c) Corrupt some patterns for recall, and see how good the recall is.

d) Corrupt part of the memory, and see how good the recall is for the original uncorrupted patterns.

Summary: Properties of BAM

AM, BAM, MAM (Matrix Associative Memory) are alternative terms.

- **Distributed:** A stored pattern is represented by all elements of the memory, as a result of Hebbian (outer product) rule for memory creation.
- **Content addressable:** The input pattern determines its storage location and its address of retrieval.
- **Robust:** It can recall patterns from corrupted (noisy or partial) instances.
- **Crosstalk exists:** Errors are possible when patterns interfere ($P > n$). Error correcting *learning* of the weights (the memory matrix $\mathbf{M}$) helps.
- **Can learn continuously**: Existing memory does not need to be erased when a new pattern is added - useful in a non-stationary environment.