Heteroassociation and autoassociation

Initialization

\begin{verbatim}
In[10]:=
Off[SetDelayed::write]
Off[General::spell1]
SetOptions[ListPlot, Joined -> True];
SetOptions[ArrayPlot, ColorFunction -> "GrayTones", ImageSize -> Tiny, Frame -> False];
\end{verbatim}

Introduction

Last time

- Linear systems overview
- Introduction to learning and memory

Outer product

Outer product, $gf^T$ models the increase in weight strength when input $f$ is associated with an output $g$:

\begin{verbatim}
Clear[f, g];
Outer[Times, Array[g, 5], Array[f, 4]] // MatrixForm
\end{verbatim}

Learning & recall

1. Learning

Let $(f_n, g_n)$ be a set of input/output activity pairs. Memories are stored by superimposing new weight changes on old ones. Information from many associations is present in each connection strength.
\[ W_{n+1} = W_n + g_n f_n^T \]  \hspace{1cm} (1)

2. Recall

Let \( f \) be an input possibly associated with output pattern \( g \). For recall, the neuron acts as a linear summer:

\[ g = Wf \]  \hspace{1cm} (2)

\[ g_i = \sum_j w_{ij} f_j \]  \hspace{1cm} (3)

3. Condition for perfect recall

If \( \{ f_n \} \) are orthonormal, the system shows perfect recall:

\[ W_n f_m = (g_1 f_1^T + g_2 f_2^T + \ldots + g_n f_n^T) f_m \]
\[ = g_1 f_1^T f_m + g_2 f_2^T f_m + \ldots + g_m f_m^T f_m + \ldots + g_n f_n^T f_m \]  \hspace{1cm} (4)

since,

\[ f_n^T f_m = \begin{cases} 1, & n = m \\ 0, & n \neq m \end{cases} \]  \hspace{1cm} (5)

Today

A common distinction in neural networks is between supervised and unsupervised learning ("self-organization"). The heteroassociative network is supervised, in the sense that a "teacher" supplies the proper output to associate with the input. Learning in autoassociative networks is unsupervised in the sense that they just take in inputs, and try to organize a useful internal representation based on the inputs. What "useful" means depends on the application goal. We explore the idea that if memories are stored with autoassociative weights, it is possible to later "recall" the whole pattern after seeing only part of the whole. Later we'll see how heteroassociative learning can be treated as a special case of autoassociative learning.

- Simulation examples

  Heterassociation

  Autoassociation

  Superposition and interference

Heteroassociation

In this and the next section, we will use Mathematica to simulate a process of association between image representations of the letters, T, I, and P. You will learn more about how to manipulate lists in Mathematica. And you will learn some of the limitations of linear recall. There are several simple exercises/questions you should try to answer.
Simulation of heteroassociative learning - Learning "IT"

- **Stimuli**

If after seeing I, the letter T follows, you might expect that T would become associated with I. The letter I might later act as a stimulus that should elicit T as a response.

```plaintext
In[1]:=  
Imatrix = {
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0},
};

In[2]:=  
Tmatrix = {
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 1, 1, 1, 1, 1, 1, 1},
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0},
};

In[3]:=  
Pmatrix = {
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 1, 1, 1, 1, 1, 1, 1},
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0},
};
```

We now turn our 2D image stimuli into 1D vectors. (We compute the maximum values, maxv and maxi, just to determine plot ranges for use later.)
\textbf{Sidenote: Making images into vectors: Flatten[] and Partition[]}

\textbf{Flatten[]} takes a list of lists and turns it into a list of elements, that is, it removes all of the inner braces:

\textbf{Partition[]} does the reverse of \textbf{Flatten[]} and takes a list of elements and structures it back into a list of lists:

For our purposes, \textbf{Flatten[]} turns a matrix representing a 2D picture (e.g. the letter I, or T) into a vector that we can store in a weight matrix memory. Later, we use \textbf{Partition[]} to turn whatever the matrix remembers into a 2D picture for comparison with the input picture originally learned.
In[17]:=
ArrayPlot[Partition[Tv, size], PlotRange -> {0, maxi}, Mesh -> False]

Out[17]=

Learning an association $I \rightarrow T$

- Learning

Let's use the outer product to represent the change in the synaptic weights (let \texttt{Sweights} be $W$) caused by the simultaneous activity of $T$ and $I$ which, assuming a Hebb-type rule, is proportional to the product of the activities:

In[30]:=
Sweights = Outer[Times, Tv, Iv];

\texttt{Iv} plays the role of $f$, and $Tv$ the role of $g$ in the earlier algebra analysis. And \texttt{Sweights} corresponds to $W$.

Dimensions[Sweights]

Now if sometime later, the weights matrix is "stimulated" with the letter $I$, it produces as a response the letter $T$:

- Recall: Remembering $T$ from stimulus $I$

In[31]:=
response = Sweights.Iv;
ArrayPlot[Partition[response, size], Mesh -> False]

Out[32]=

Note that we expect this result from the rules of algebra: $(T_v \cdot I_v^T) \cdot I_v = T_v \cdot (I_v^T \cdot I_v) = T_v$, because we normalized
the input vectors.

**Exercises**

What if $T v$ was the input?

What if a random vector was the input? What response would you expect?

```math
\text{In[33]:= } \text{response} = \text{Sweights}.Tv; \\
\text{ArrayPlot[Partition[response, size]]}
```

```math
\text{Out[34]=} \text{T}
```

What if you used stimulated the Transpose of Sweights with Tv?

What if you right-multiply $Tv$ by Sweights? (right-multiply means you put the matrix to the right of the vector.)

**Add the association $T \leftrightarrow P$ to the mix**

- Learning $P$ from $T$, storing it with the association between $I$ and $T$

```math
\text{In[39]:= } \text{Sweights} = \text{Sweights} + \text{Outer[Times,Pv,Tv]};
```
Recall: Stimulate with T: Response? P, I, or a mixture?

`response = Sweights.Tv;
ArrayPlot[Partition[response, size]]`

Exercise

If you look carefully, you can see some evidence for interference. Why might you expect this based on the two inputs `Iv` and `Tv`? Try comparing the dot products of the various inputs.

Can you think of a network modification for recall that might help to reduce the interference?

Exercise

The `ArrayPlot` functions don't always give the best way of seeing variations in a function or list. Try `ListPlot[response]`.

What do you notice?

You can also use `Histogram[]` to see how the responses are distributed.

Exercise

The plot function automatically scales the plot range so that white corresponds to the maximum value in the list. Use `Max[Tv]` to find the peak value in a list.

Add a third association I <-> P to the mix

Learning P & I, store it with the associations between I & T, P & T

`Sweights = Sweights + Outer[Times,Iv,Pv];`
Recall: Stimulate with P: Response? T, I, or a mixture?

In[56]:= response = Sweights.Pv; ArrayPlot[Partition[response, size]]

Out[56]=

Summary so far

We’ve seen that the outer product form of Hebbian learning can be used to store associations between pairs of patterns (i.e. vectors). These associations are stored in a matrix that models synaptic strengths between pairs of neurons. We can later use these associations to recall a pattern that was previously associated with another. We can think of this recall as a “stimulus-response” process. However, if the input patterns are not all mutually orthogonal, then we have interference. In other words, as we add patterns, the associations get mixed, and later recall can produce a mixture of responses to a given stimulus. From a machine intelligence perspective, this is not good. But as a psychologist or neuroscientist, it raises the interesting question of whether our own recall makes similar mistakes.

Autoassociation

If \( f = g \), then we have an autoassociative system. There is only one set of units, and each element potentially connects to each other element. Later we will see how the autoassociative architecture is used in non-linear networks. Autoassociation stores information about the relationships between the elements or features of a stimulus pattern (vector). They do unsupervised learning. We can show how this kind of knowledge can be used to predict or reconstruct missing information. Neural networks of this sort build internal models of the statistical structure of the ensemble they are exposed to. An autoassociative network can be as a content-addressable memory.

Reconstructive property

Autoassociation can reconstruct missing parts of a stimulus.

Suppose a whole pattern \( x \), consists of two parts \( \{f_1,f_2,f_3\} \), and \( \{g_1,g_2,g_3,g_4\} \):
\[ x = \begin{pmatrix}
  f_1 \\
  f_2 \\
  \vdots \\
  f_m \\
  g_1 \\
  \vdots \\
  g_n
\end{pmatrix} \]

Let's represent the association of vector \( x \) with itself by the outer product in matrix \( W \):

```plaintext
In[57]:= f = \{f_1, f_2, f_3, 0, 0, 0\};
g = \{0, 0, 0, g_1, g_2, g_3, g_4\};
x = f + g;
W = Outer[Times, x, x];
```

```plaintext
In[61]:= x = f + g
Out[61]= \{f_1, f_2, f_3, g_1, g_2, g_3, g_4\}
```

Sometime later, we input a version of \( x \), but with "missing" elements--i.e. \( x \) with some elements set to zero--namely, vector \( f \). What do we get in response?

```plaintext
In[62]:= W.f
Out[62]= \{f_1^3 + f_1 f_2^2 + f_1 f_3^2, f_1^2 f_2 + f_2^3 + f_2 f_3^2, f_1^2 f_3 + f_2^2 f_3 + f_3^3, f_1^2 g_1 + f_2^2 g_1 + f_3^2 g_1, f_1^2 g_2 + f_2^2 g_2 + f_3^2 g_2, f_1^2 g_3 + f_2^2 g_3 + f_3^2 g_3, f_1^2 g_4 + f_2^2 g_4 + f_3^2 g_4\}
```

```plaintext
In[63]:= Simplify[\%]
Out[63]= \{f_1 (f_1^2 + f_2^2 + f_3^2), f_2 (f_1^2 + f_2^2 + f_3^2), f_3 (f_1^2 + f_2^2 + f_3^2), (f_1^2 + f_2^2 + f_3^2) g_1, (f_1^2 + f_2^2 + f_3^2) g_2, (f_1^2 + f_2^2 + f_3^2) g_3, (f_1^2 + f_2^2 + f_3^2) g_4\}
```

If we replace \((f_1^2 + f_2^2 + f_3^2)\) by \(\alpha\), we see that \(W.f\) is proportional to \(x\). In general, the matrix \(W\) restores \(f\) to the pattern \(x\) up to a scale factor \(\alpha\). We can use Mathematica's replacement operator, \(/.\), to substitute \(\alpha\) for \((f_1^2 + f_2^2 + f_3^2)\)
If \( \mathbf{f} \) was initially normalized to 1, then \( \mathbf{W}.\mathbf{f} \) is would be equal to \( \mathbf{x} \).

**Exercise**

What does it mean to have a "missing" part of a pattern?

**Autoassociation includes heteroassociation as a special case**

At first it may seem that an autoassociative system is a more restrictive type of association than heteroassociation. But suppose we have an input/output pair \( \{\mathbf{f}, \mathbf{g}\} \). If we form a new vector \( \mathbf{f}' \) in which we stack \( \mathbf{f} \) on top of \( \mathbf{g} \), then autoassociation outer product matrix contains within it, the heteroassociation between \( \mathbf{f} \) and \( \mathbf{g} \).

**Question:**

What can you say about the eigenvectors of the weight matrix from autoassociative learning? Take a look at: \( \text{Outer}[-\text{Times},\text{Array}[ff,4],\text{Array}[ff,4]]/\text{MatrixForm} \)

**Hint:** Is the autoassociative matrix symmetric?

In general, is the heteroassociative matrix symmetric?
Autoassociative example with TIP pictures

- Learn about T, learn about I, and store the associations together by superimposing their weight matrices

```math
In[89]:=
Clear[Sweights];
Sweights = Outer[Times, Tv, Tv] + Outer[Times, Iv, Iv];
```

- Sometime later, stimulate the network with an impoverished T, missing some bits

Let's delete the 2nd row of T:

```math
In[91]:=
forgettingTmatrix = Partition[Tv, size];
forgettingTmatrix[[2]] = Table[0, {10}];
forgettingT = Flatten[forgettingTmatrix];
ArrayPlot[Partition[forgettingT, size]]
```

```
Out[94]=
```

- Recall of the original T, from the missing bits

```math
In[95]:=
rememberingT = Sweights.forgettingT;
ArrayPlot[Partition[rememberingT, size]]
```

```
Out[95]=
```
Interference: Corrupt T again, this time with some other random bits missing

Let's do something a little more drastic to T. We'll randomly "delete" pixels of the picture:

In[96]:= pepper = RandomInteger[1, Dimensions[Tv][1]]; peppermatrix = DiagonalMatrix[pepper]; forgettingT = peppermatrix.Tv; ArrayPlot[Partition[forgettingT, size]]

Out[96]=

In[97]:= rememberingT = Sweights.forgettingT; ArrayPlot[Partition[rememberingT, size]]

Out[97]=

A cross-section through the above pattern shows the differences in magnitude between the correct values (T), and the other spurious values.
In[98]:= ListPlot[Partition[rememberingT, size][5], AxesOrigin -> {0, -0.25}, PlotRange -> {-.5, maxi}, Joined -> True]

Out[98]=

- Mathematica note: Recall that you can get information about the options as well as the functions with a ?? query:

??AxesOrigin

AxesOrigin is an option for graphics functions which specifies where any axes drawn should cross. 

Attributes[AxesOrigin] = {Protected}

In[106]:= Dimensions[forgettingT]

Out[106]= {100}
Interference: Corrupt $T$, with added noise

```math
In[101]:= forgettingT = Tv; forgettingT[[59]] = 0.27;
ArrayPlot[Partition[forgettingT, size]]
```

```
Out[101]=
```

```math
In[102]:= rememberingT = Sweights.forgettingT;
ArrayPlot[Partition[rememberingT, size]]
```

```
Out[102]=
```

Although the memory looks pretty good, it is not perfect because although $Tv$ and $Iv$ were almost orthogonal, with a cosine of about .09, they were not perfectly orthogonal. In fact, we can get a measure of how close $rememberingT$ is to $Tv$ in terms of the cosine of the angle between them:

```math
Tv.Normalize[rememberingT]
```

```
0.995682
```

Interference with more autoassociations: $I$, $T$, and now $P$ too

If we have the connection matrix, $Sweights$ store another letter, $P$, then we will begin to get even more interference when we try to recall $T$ from a fragment of $T$:

```math
In[109]:= Sweights = Sweights +
Outer[Times,Pv,Pv];
```
In[110]:= rememberingT = Sweights.forgettingT;
   ArrayPlot[Partition[rememberingT, size]]

Out[110]=

This is because the patterns we've stored are not mutually orthogonal, and in particular, P is too close to I and T:

In[111]:= {Iv.Pv, Tv.Pv, Tv.Iv}

Out[111]= {0.243332, 0.367884, 0.0944911}

We can picture the range of values that rememberingT takes on:

In[112]:= ListPlot[rememberingT, Joined -> False]

Out[112]=

How could we fix the interference problem? We could try to set values below a threshold value to zero.

**Include a threshold. Applying a function over a list**

Define a non-linear threshold, \(\text{step}[x_]\), which when applied to rememberingT removes the interference. A critical parameter is the threshold.

Note: When you define a new function, it is not necessarily "Listable". If not, here are two solutions.

In[113]:= \(\text{step}[x_, t_] := \text{If}[x > t, 1, 0.0];\)
- Change the function attributes

As we saw earlier, you can define your function to be listable with

\[
\text{In}[114]:= \quad \text{SetAttributes[step,Listable]}
\]

Then you can apply step directly to the list rememberingT, and then the function will be applied successively to each element of the list:

\[
\text{In}[115]:= \quad \text{rememberingT}
\]

\[
\text{Out}[115]= \quad \{0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.35166, 0.35166, 0.35166,
0.35166, 0.35166, 0.300669, 0.267261, 0., 0., 0., 0.0843983, 0.,
0.0843983, 0.267261, 0., 0.117806, 0., 0., 0., 0., 0.0843983, 0., 0., 0.267261,
0., 0.117806, 0., 0., 0., 0.0843983, 0.0843983, 0.0843983,
0.35166, 0.0843983, 0.0334077, 0., 0., 0., 0., 0.0843983, 0., 0., 0.267261,
0., 0.0334077, 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.\}
\]

\[
\text{In}[116]:= \quad \text{step[rememberingT,0]}
\]

\[
\text{Out}[116]= \quad \{0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 1, 1, 1, 1, 1, 1, 0., 0., 0.,
1, 0., 0., 1, 0., 0., 0., 0., 0., 1, 0., 0., 0., 0., 1, 0., 1, 0., 0., 0., 0., 0.,
1, 0., 0., 1, 0., 1, 0., 0., 0., 0., 1, 1, 1, 1, 1, 0., 0., 0., 0., 0.,
1, 0., 0., 1, 0., 0., 0., 0., 0., 1, 1, 0., 0., 0., 0., 0., 0.
\}
\]

- Map the function over the list

This means that to apply \text{step}[] to rememberingT, you would have to use the \text{Map}[] function:

The \text{Map}[] function is used often enough, that \text{Mathematica} has a short-hand:

\[
\text{In}[117]:= \quad \text{Clear[f,a,b,c];}
\quad \text{Map[f,{a,b,c}]}
\]

\[
\text{Out}[118]= \quad \{f[a], f[b], f[c]\}
\]

\[
\text{In}[119]:= \quad f \, /\, \{a,b,c\}
\]

\[
\text{Out}[119]= \quad \{f[a], f[b], f[c]\}
\]
If the function has multiple arguments, then use # to identify which function slots get the variable, with the function end defined by &:

\[ \text{In}[120]:= \quad \text{Map}[	ext{step}[, 0]] & , \text{rememberingT}] ; \]

Using the short-hand version:

\[ \text{In}[121]:= \quad \text{step}[, 0] & /@ \text{rememberingT}] ; \]

We'll put it all in one line, together with a slider to adjust the threshold:

\[ \text{In}[122]:= \quad \text{Manipulate}[	ext{ArrayPlot}[	ext{Partition}[\text{step}[, t]] & /@ \text{rememberingT}, \text{size}], \text{ImageSize} \rightarrow \text{Small}], \{t, 0, 1\}] \]

\[ \text{Out}[122]= \]
How do you think recall would be affected if one of the patterns was repeated several times?

How could you transform the input patterns so that they are orthogonal?

The threshold choice is clearly important. How could you make the threshold automatic?

### Autoassociation with pictures

This is the same sort of exercise as above, but uses graylevel patterns, and shows a few tools for importing image data.

### Autoassociation with some other patterns: Einstein or Shannon

Get `einstein64x64.jpg`:

In[123]:= `ieinstein = Import[SystemDialogInput["FileOpen"], "Data"];

But it is usually easier just to drag the picture in:

In[125]:= `ieinstein = ImageData[ ];

In[127]:= `size2 = Dimensions[ieinstein][[1]];

In[128]:= `geinstein = ArrayPlot[ieinstein, ColorFunction -> GrayLevel] ` `einstein = N[Normalize[Flatten[ieinstein]]];

Out[128]=

Get `shannon64x64.jpg`:
In[130]:
ishannon = ImageData[ ];

gshannon = ArrayPlot[ishannon, ColorFunction -> GrayLevel];
(*Save the graphic of ishannon picture for later display*)
shannon = N[Normalize[Flatten[ishannon]]];

Out[131]=

- Note that you can import data from any accessible URL. E.g.

In[133]:=
Import["http://gandalf.psych.umn.edu/users/kersten/kersten-lab/courses/
  NeuralNetworksKoreaUF2011/MathematicaNotebooks/Lect_8_HeterAuto/
einstein64x64.jpg"]

Import["http://gandalf.psych.umn.edu/users/kersten/kersten-lab/courses/
  NeuralNetworksKoreaUF2011/MathematicaNotebooks/Lect_8_HeterAuto/
  shannon64x64.jpg"];

Autoassociatively store einstein

In[134]:= Sweights = Outer[Times,einstein,einstein];

In[135]:= Dimensions[einstein]

How big is Sweights?

Make an einsten picture with some "salt and pepper noise"--random intensities at random locations, einstein64x64missing.jpg:
In[136]:= `forgettingeinstein = einstein;
   Table[forgettingeinstein[[RandomInteger[{1, 4096}]]] =
      RandomReal[{0.0, Max[einstein]}], {1000};
   gforgettingeinstein = ArrayPlot[Partition[forgettingeinstein, size2],
      ColorFunction -> GrayLevel]

Out[138]=

Try to recall all of einstein, with only a part of the original pixels in einstein, i.e. with `forgettingeinstein` as input:

In[139]:= `rememberingeinstein = Swights.forgettingeinstein;
   grememberingeinstein = ArrayPlot[Partition[rememberingeinstein, size2], ColorFunction -> GrayLevel]

In[141]:= Show[GraphicsRow[{geinstein, gforgettingeinstein, grememberingeinstein}]]

Out[141]=

Now superimpose the outerproduct weight matrices for einstein and shannon:

In[142]:= `deltasweights = Outer[Times, shannon, shannon];

In[143]:= `Swights = Swights + deltasweights;

In[144]:= `rememberingeinstein = Swights.forgettinTEGRateinstein;
   grememberingeinstein = ArrayPlot[Partition[rememberingeinstein, size2], ColorFunction -> GrayLevel]
We see there is considerable interference with just two pictures. To deal with this we could ask: how could the images be recoded so as to reduce the interference? Alternatively, we could ask: how could the recall process be improved to make better decisions?

Seeking answers to the first approach will first take us in the direction of trying to understand how inputs could be preprocessed to reduce later interference. Seeking answers to the second approach, we will study non-linear networks that reduce interference during recall.

Next time

Neural networks as statistical pattern processing

Next time we are going ask ourselves what these networks are doing in the sense of information or pattern processing. This will lead naturally to a statistical framework for understanding both heteroassociative and autoassociative networks.

Heterassociation will lead to regression.

Autoassociation will lead to second-order statistical learning.

A deeper understanding of the information processing roles of 1) our learning rule; and 2) our recall mechanism, helps in two ways. First, we will have a better idea of what we need to do to get the network to solve a given problem. Second, we may discover that there is a different problem for which it is better suited. So linear heterassociation recall may be a poor classification model, but it might be a good interpolation model. Linear autoassociation networks may be a poor way to store information about specific memories, but may be a good way of learning about the statistics of ensembles.

As a preview of the latter, consider another application of autoassociation learning, where the goal is not to remember a particular previous input, but rather to learn something about an ensemble of inputs that all belong to the same class. Practical examples are collections of networks that can learn about their own special environments. For example, you want to build an automated driving system. The problem is complex, in part, due to different kinds of demands placed by the driving environment. So you decide you need three "experts": one for single-lane country roads, another for two-lane highways, and another for four-lane divided highways. Now you put them all in one vehicle and let them all monitor their sensory inputs as the vehicle is being driven (by one of them). When given a new environment, these experts "compare
notes" to see how well this new environment fits their internal models or domain of expertise. Which ever expert "knows" the new environment the best, gets to drive the car. This is related to the idea of mental modules. The part of the driving expert that validates the environment could be realized by an autoassociative network. It can do this by testing how well its prediction of the environment's input to its sensors fits the actual sensor measurements.

- Improve learning or recall?

We've seen that the linear associator has problems of interference. How do we improve the network? We have two general strategies: 1) improve the learning, so that linear recall will do better; 2) improve the recall, so that a simple Hebbian outer product rule can still be used.

Later we will derive a new learning rule that builds better association matrices for certain problems like interpolation, regression, and generalization. And we will look at non-linear recall mechanisms that make cleaner classifications for memory problems.

For a specific example, the outerproduct learning rule can be seen as part of a computation that computes autocovariance matrices. These are useful because the input ensemble in some sense "lives" in the space defined by the eigenvectors with the largest eigenvalues. The problem is that to project input vectors into this space using simple matrix multiplication requires one to have the eigenvector matrix, NOT the autocovariance matrix. So we have two possibilities. First, we can find an alternative learning rule that will give us the eigenvector matrix. Or we can look for a different recall rule that will give us the projection we want.