Introduction to Neural Networks
U. Minn. Psy 5038
Spring, 1998

Lecture 8
The Linear associator, continued

Initialization

In[1]:=
```
Off[SetDelayed::write]
Off[General::spell1]
```

Review of linear association model

1. Learning.

Let $\{\mathbf{f_n},\mathbf{g_n}\}$ 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.

$$\mathbf{W_{n+1}} = \mathbf{W_n} + \mathbf{g_n}\mathbf{f_n^T}$$

2. Recall

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

$$\mathbf{g} = \mathbf{W}\mathbf{f}$$

$$g_i = \sum_j w_{ij} f_j$$

If $\{\mathbf{f_n}\}$ are orthonormal, the system shows perfect recall:

$$\mathbf{W_n}\mathbf{f_m} = \left(\mathbf{g_1}\mathbf{f_1^T} + \mathbf{g_2}\mathbf{f_2^T} + \ldots + \mathbf{g_n}\mathbf{f_n^T}\right)\mathbf{f_m}$$

$$= \mathbf{g_1}\mathbf{f_1^T}\mathbf{f_m} + \mathbf{g_2}\mathbf{f_2^T}\mathbf{f_m} + \ldots + \mathbf{g_m}\mathbf{f_m^T}\mathbf{f_m} + \ldots + \mathbf{g_n}\mathbf{f_n^T}\mathbf{f_m}$$

$$= \mathbf{g_m}$$

$$\mathbf{f_n^T f_m} = \begin{cases} 1, & n = m \\ 0, & n \neq m \end{cases}$$

Exercise

Try this either with paper and pencil, or with *Mathematica*.

Let f1 and f2 be two orthogonal, normalized input patterns:

$$f_1 = \frac{1}{\sqrt{3}} \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix}$$

$$f_2 = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

and g1, g2 two output patterns:

$$g_1 = \begin{pmatrix} 3 \\ 1 \\ 2 \end{pmatrix}$$

$$g_2 = \begin{pmatrix} -1 \\ -1 \\ 2 \end{pmatrix}$$

Form the outer product:

$$W = g_1 f_1^T$$

Test for "recall" by feeding f1 as input to W. Stimulate W with f2. What happens? Add the outer product:

$$g_2 f_2^T$$

to the previous W matrix. Now test for recall on stimulation with f1, and f2. What do you find?

Heteroassociation

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.

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

In[4]:=
```
Tmatrix = {
  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 1, 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}};
```

In[5]:=
```
Pmatrix = {
  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
  {0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
  {0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
  {0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
  {0, 1, 1, 1, 1, 1, 0, 0, 0, 0},
  {0, 1, 0, 0, 0, 0, 1, 0, 0, 0},
  {0, 1, 0, 0, 0, 0, 1, 0, 0, 0},
  {0, 1, 0, 0, 0, 0, 1, 0, 0, 0},
  {0, 1, 1, 1, 1, 1, 0, 0, 0, 0},
  {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}};
```

In[6]:=
```
normalize[x_] := N[x/Sqrt[x.x]];
Tv = normalize[Flatten[Tmatrix]];
Iv = normalize[Flatten[Imatrix]];
Pv = normalize[Flatten[Pmatrix]];
```

■ Sidenote: Making images into vectors: Flatten[] and Partition[]

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

In[10]:=
```
Flatten[{{a,b},{c,d}}]
```

Out[10]= $\{a, b, c, d\}$

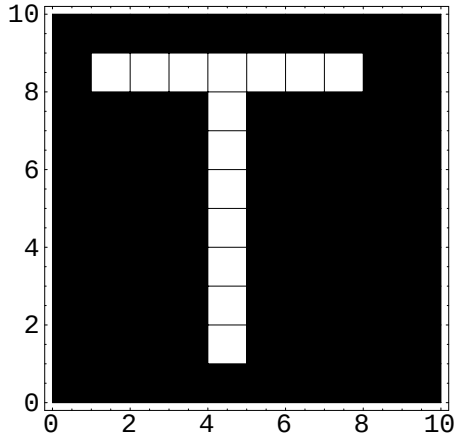
Partition[] is sort of like an inverse for **Flatten[]** and takes a list of elements and structures it back into a list of lists:

In[11]:=
```
Partition[%,2]
```

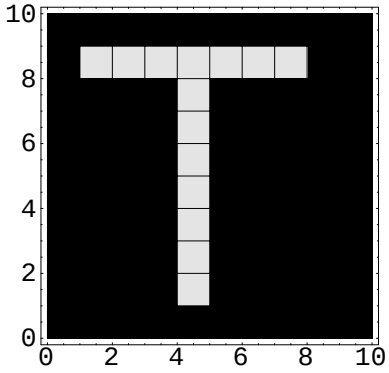
Out[11]= $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$

For our purposes, **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 **Partition[]** to turn whatever the matrix remembers into a 2D picture for comparison with the input picture originally learned.

In[12]:=
```
ListDensityPlot[Tmatrix, PlotRange -> {0, 1}];
```



In[13]:=
```
ListDensityPlot[Partition[Tv,10],PlotRange->{0,.3}];
```



■ Learning an association between I and T

Let's use the outer product to represent the change in the synaptic weights caused by the simultaneous activity of **T** and **I** which assuming a Hebb-type rule, is proportional to the product of the activities:

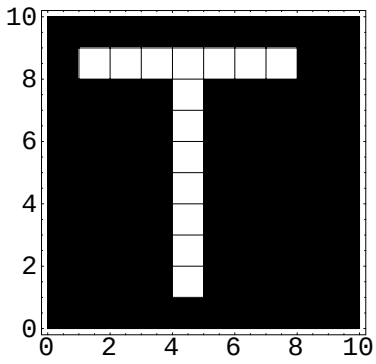
In[14]:=
```
Weights = Outer[Times,Tv,Iv];
```

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 I

In[15]:=
```
response = Weights.Iv;
```

In[16]:=
```
ListDensityPlot[Partition[response,10]];
```



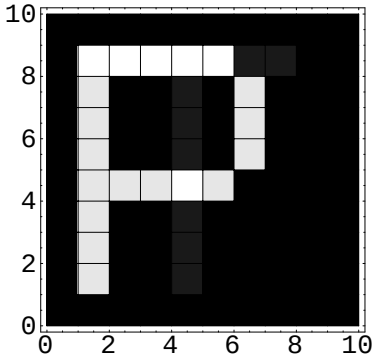
■ Learning P from T, storing it with the association between I and T

In[17]:=
```
Weights = Weights + Outer[Times,Pv,Tv];
```

■ Recall: Stimulate with T: Response? P, I, or a mixture?

In[18]:=
```
response = Weights.Tv;
```

In[19]:=
```
ListDensityPlot[Partition[response,10]];
```



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.

The DensityPlot functions don't always give the best way of seeing variations in a function or list. Try **ListPlot[response]**. What do you notice?

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

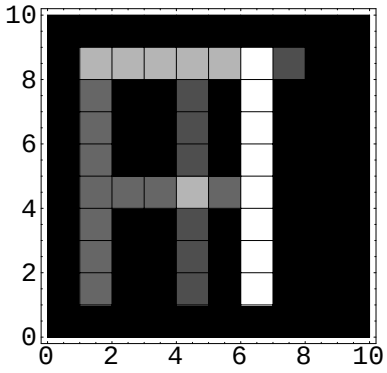
■ **Learning P & I, store it with the associations between I & T, P & T**

In[20]:=
```
Weights = Weights + Outer[Times,Iv,Pv];
```

■ **Recall: Stimulate with P: Response? T, I, or a mixture?**

In[21]:=
```
response = Weights.Pv;
```

In[22]:=
```
ListDensityPlot[Partition[response,10]];
```



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 this architecture is used in non-linear networks. Autoassociation stores information about the relationships between the elements or features of a stimulus. We can show how this kind of knowledge can be used later 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. Practical examples are systems of networks that can learn about their own special environment (e.g. an expert on single-lane country roads, another on two-lane highways, and another on four-lane divided highways, as in the CMU computer driving project). When given a new environment, these experts can "compare notes" to see how well this new environment fits their internal models. Which ever expert "knows" the new environment the best, gets to have its expert driver drive the car!

Reconstructive property

Autoassociation can reconstruct missing parts of a stimulus.

$$\mathbf{x} = \begin{pmatrix} f_1 \\ f_2 \\ \cdot \\ \cdot \\ f_m \\ g_1 \\ \cdot \\ \cdot \\ \cdot \\ g_n \end{pmatrix} \qquad (1)$$

Suppose a whole pattern **x**, consists of two parts {f1,f2,f3}, and {g1,g2,g3,g4}, and the association of vector **x** with itself is represented by **W**:

In[30]:=
```
f={f1,f2,f3,0,0,0,0};
g={0,0,0,g1,g2,g3,g4};
x=f+g;
W=Outer[Times,x,x];
```

In[28]:=
```
x
```

In[29]:=
```
x = f + g
```

Out[29]=
```
{f1, f2, f3, g1, g2, g3, g4}
```

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?

In[34]:=
```
Simplify[W.f]
```

Out[34]=
$$\{f1\,(f1^2 + f2^2 + f3^2),\ f2\,(f1^2 + f2^2 + f3^2),\ f3\,(f1^2 + f2^2 + f3^2),\ (f1^2 + f2^2 + f3^2)\,g1,\ (f1^2 + f2^2 + f3^2)\,g2,\ (f1^2 + f2^2 + f3^2)\,g3,\ (f1^2 + f2^2 + f3^2)\,g4\}$$

So if **f** is normalized, the sum of the squared elements of **f** are equal to 1, and **W.f** is equal to **x**.

The matrix **W** restores **f** to the full pattern **x**.

■ Autoassociation includes heteroassociation

At first it may seem that an autoassociative system is a more restrictive type of association than heteroassociation. But if we form a new vector **f'** in which we stack **f** on top of **g**, then autoassociation matrix contains within it, the heteroassociation between **f** and **g**.

Question:

What can you say about the eigenvectors of the weight matrix from autoassociative learning?

An example with TIP pictures

■ **Learn about T, learn about I, and store the associations together**

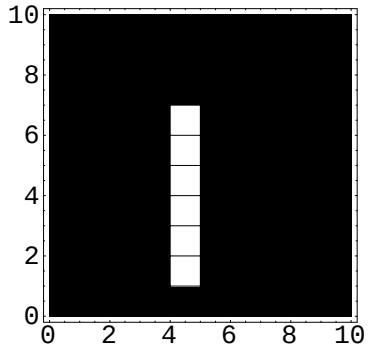
In[35]:=
```
Clear[Weights];
Weights = Outer[Times,Tv,Tv] + Outer[Times,Iv,Iv];
```

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

In[37]:=
```
forgettingT =
Join[Take[Tv,(Dimensions[Tv][[1]]-30)],Table[0, {30}]];
```

Take[m,n] returns a list containing the first n elements of **m**. **Take[m,-n]** returns a list containing the last n elments of **m.**

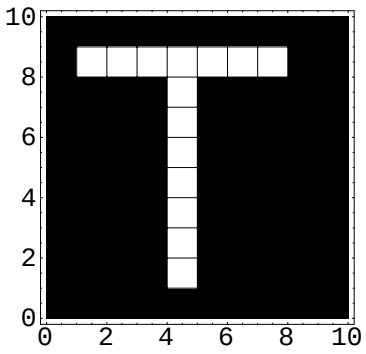
In[38]:=
```
ListDensityPlot[Partition[forgettingT,10]];
```



■ **Recall of the original T, from the missing bits**

In[39]:=
```
rememberingT = Weights.forgettingT;
```

In[40]:=
```
ListDensityPlot[Partition[rememberingT,10]];
```

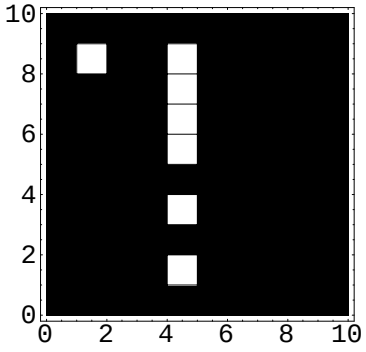


■ Interference: Corrupt T again, this time with some other random bits missing

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

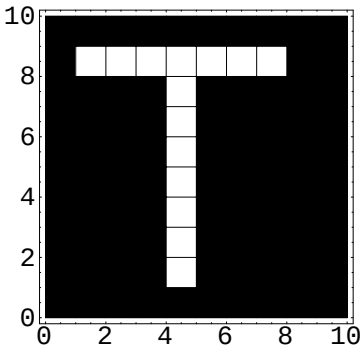
In[41]:=
```
pepper = Table[Random[Integer,1],{Dimensions[Tv][[1]]}];
peppermatrix = DiagonalMatrix[pepper];
forgettingT = peppermatrix.Tv;
```

```
ListDensityPlot[Partition[forgettingT,10]];
```

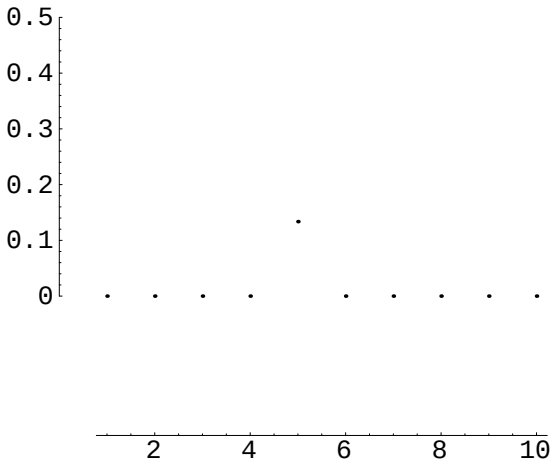


Out[44]=
```
- DensityGraphics -
```

In[45]:=
```
rememberingT = Weights.forgettingT;
ListDensityPlot[Partition[rememberingT,10]];
```



In[47]:=
```
ListPlot[Partition[rememberingT,10][[5]],
    PlotRange->{0,.5},AxesOrigin->{0,-.25}];
```



■ **Note:** You can get information about the options as well as the functions with a **??** query:

In[48]:=
```
??ListPlot
??AxesOrigin
```

ListPlot[{y1, y2, ... }] plots a list of values. The x coordinates for each point are taken to be 1, 2, ... ListPlot[{{x1, y1}, {x2, y2}, ... }] plots a list of values with specified x and y coordinates.

Attributes[ListPlot] = {Protected}

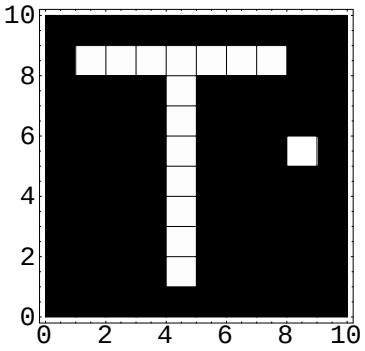
Options[ListPlot] = {AspectRatio −> GoldenRatio^(−1), Axes −> Automatic, AxesLabel −> None, AxesOrigin −> Automatic, AxesStyle −> Automatic, Background −> Automatic, ColorOutput −> Automatic, DefaultColor −> Automatic, Epilog −> {}, Frame −> False, FrameLabel −> None, FrameStyle −> Automatic, FrameTicks −> Automatic, GridLines −> None, ImageSize −> Automatic, PlotJoined −> False, PlotLabel −> None, PlotRange −> Automatic, PlotRegion −> Automatic, PlotStyle −> Automatic, Prolog −> {}, RotateLabel −> True, Ticks −> Automatic, DefaultFont :> $DefaultFont, DisplayFunction :> $DisplayFunction, FormatType :> $FormatType, TextStyle :> $TextStyle}

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

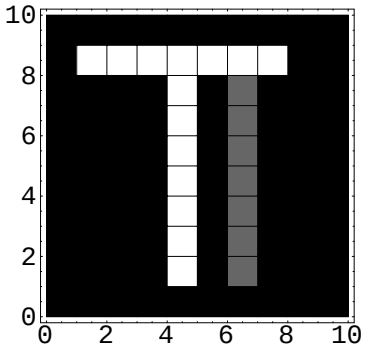
Attributes[AxesOrigin] = {Protected}

■ **Interference: Corrupt T, with added noise**

In[50]:=
```
forgettingT = Tv;
forgettingT[[59]] = .27;
ListDensityPlot[Partition[forgettingT,10]];
```



In[53]:=
```
rememberingT = Weights.forgettingT;
ListDensityPlot[Partition[rememberingT,10]];
```



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:

```
Tv.normalize[rememberingT]
```

```
0.995682
```

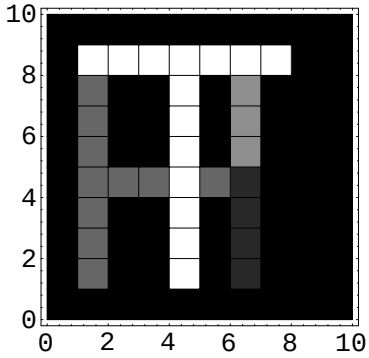
■ Interference with more autoassociations: I, T, and now P too

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

```
Weights = Weights +
           Outer[Times,Pv,Pv];
```

```
rememberingT = Weights.forgettingT;
```

```
ListDensityPlot[Partition[rememberingT,10]];
```



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

```
{Iv.Pv, Tv.Pv, Tv.Iv}
```

```
{0.243332, 0.367884, 0.0944911}
```

Exercise - include a threshold & making functions listable

Define a non-linear threshold, **step[x_]**, which when applied to **rememberingT** removes the interference. A critical paramter is the threshold. How could you make the threshold adaptive?

Note: When you define a new function, it is not necessarily "**Listable**". This means that to apply **step[]** to **remembertingT**, you would have to use the **Map[]** function:

The **Map[]** function is used often enough, that *Mathematica* has a short-hand:

```
Map[f,{a,b,c}]
```

```
{f[a], f[b], f[c]}
```

```
f /@ {a,b,c}
```

```
{f[a], f[b], f[c]}
```

```
Map[step,rememberingT]
```

The following is equivalent:

```
step /@ rememberingT
```

Alternatively, you can define your function to be listable with

```
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:

```
step[rememberingT]
```