Introduction to Neural Networks
U. Minn. Psy 5038

Lecture 8
Examples of heteroassociation and autoassociation

Initialization

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

Introduction

Last time

■ Outer product

Outer product, $\mathbf{g}f^T$ models the increase in weight strength when input **f** is associated with an output **g**:

In[3]:=
```
Outer[Times,Array[g,3],Array[f,3]]//MatrixForm
```

Out[3]//MatrixForm=

$$\begin{pmatrix} f(1)\,g(1) & f(2)\,g(1) & f(3)\,g(1) \\ f(1)\,g(2) & f(2)\,g(2) & f(3)\,g(2) \\ f(1)\,g(3) & f(2)\,g(3) & f(3)\,g(3) \end{pmatrix}$$

■ Learning & recall

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.

$$\mathbf{W}_{n+1} = \mathbf{W}_n + \mathbf{g}_n\,\mathbf{f}_n^T \tag{1}$$

2. **Recall**

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

$$\mathbf{g} = \mathbf{W}\mathbf{f} \tag{2}$$

$$g_i = \sum_j w_{ij}\,f_j \tag{3}$$

3. **Condition for perfect recall**

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

$$\begin{aligned}\mathbf{W}_n\,\mathbf{f}_m &= (\mathbf{g}_1\,\mathbf{f}_1^T + \mathbf{g}_2\,\mathbf{f}_2^T + \ldots + \mathbf{g}_n\,\mathbf{f}_n^T)\,\mathbf{f}_m \\ &= \mathbf{g}_1\,\mathbf{f}_1^T\,\mathbf{f}_m + \mathbf{g}_2\,\mathbf{f}_2^T\,\mathbf{f}_m + \ldots + \mathbf{g}_m\,\mathbf{f}_m^T\,\mathbf{f}_m + \ldots + \mathbf{g}_n\,\mathbf{f}_n^T\,\mathbf{f}_m \\ &= \mathbf{g}_m\end{aligned} \tag{4}$$

since,

$$\mathbf{f}_n^T\,\mathbf{f}_m = \begin{cases} 1, & n = m \\ 0, & n \neq m \end{cases} \tag{5}$$

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 the inputs. What "useful" means depends on the application. 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.

■ Simulations

Heterassociation

Autoassociation

Superposition and interference

Heteroassociation

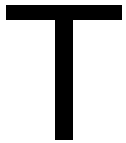
In this and the next section, we will use *Mathematica* to simulate a process of association between image representations of the letters, T, I, and P. You will learn more about how to manipulate lists in *Mathematica*. And you will learn some of the limitations of linear recall. There are several simple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

Simulation of heteroassociative learning - Learning "IT"

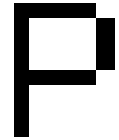
■ Stimuli

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.

In[4]:=
```
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[5]:=
```
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[6]:=
```
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}};
```

General::spell :
Possible spelling error: new symbol name "Pmatrix" is similar to existing symbols {Imatrix, Tmatrix}.

In[7]:=
```
normalize[x_] := N[x/Sqrt[x.x]];
Tv = normalize[Flatten[Tmatrix]];
Iv = normalize[Flatten[Imatrix]];
Pv = normalize[Flatten[Pmatrix]];
size = Dimensions[Imatrix][[1]];
maxv = Max[Flatten[Tmatrix]];
maxi = Max[Tv];
```

■ 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[14]:=
```
Flatten[{{a,b},{c,d}}]
```

Out[14]=
$$\{a, b, c, d\}$$

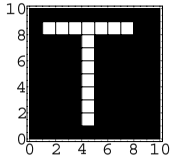
Partition[] is sort of like an inverse for **Flatten[]** and takes a list of elements and structures it back into a list of lists:

In[15]:=
```
Partition[%,2]
```

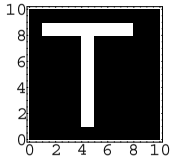
Out[15]=
$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

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[16]:=
```
ListDensityPlot[Tmatrix, PlotRange -> {0, maxv}];
```



In[17]:=
```
ListDensityPlot[Partition[Tv,size],PlotRange->{0,maxi},Mesh->False];
```



Learning an association I -> T

■ Learning

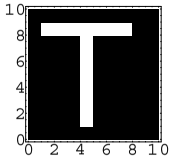
Let's use the outer product to represent the change in the synaptic weights caused by the simultaneous activity of **T** and **I** which assuming a Hebb-type rule, is proportional to the product of the activities:

In[18]:=
```
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

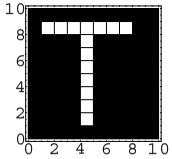
In[19]:=
```
response = Weights.Iv;
ListDensityPlot[Partition[response,size],Mesh->False];
```



Exercise

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

In[21]:=
```
response = Weights.Iv;
ListDensityPlot[Partition[response,size]];
```



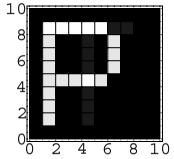
Add the association T -> P to the mix

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

In[23]:=
```
Weights = Weights + Outer[Times,Pv,Tv];
```

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

In[24]:=
```
response = Weights.Tv;
ListDensityPlot[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 **DensityPlot** functions don't always give the best way of seeing variations in a function or list. Try **ListPlot[response]**. What do you notice?

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.

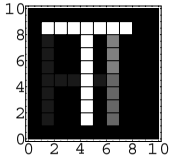
Add the association I -> P to the mix

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

In[26]:=
```
Weights = Weights + Outer[Times,Iv,Pv];
```

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

In[27]:=
```
response = Weights.Iv;
ListDensityPlot[Partition[response,size]];
```



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 pattern (vector). 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.

Reconstructive property

■ **Autoassociation can reconstruct missing parts of a stimulus.**

$$\mathbf{x} = \begin{pmatrix} f_1 \\ f_2 \\ \cdot \\ \cdot \\ f_m \\ g_1 \\ \cdot \\ \cdot \\ \cdot \\ g_n \end{pmatrix} \qquad (6)$$

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 the outer product in matrix **W**:

In[29]:=
```
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[33]:=
```
x = f + g
```

Out[33]=
```
{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, each sum of squared elements of **f** is equal to 1, and **W.f** is equal to **x**. In general, the matrix **W** restores **f** to the pattern **x** up to a scale factor α:

In[35]:=
```
% /. (f1^2 + f2^2 + f3^2) → α
```

Out[35]=
```
{f1 α, f2 α, f3 α, g1 α, g2 α, g3 α, g4 α}
```

Exercise

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

■ **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 outer product 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?

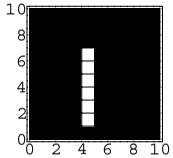
Autoassociative example with TIP pictures

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

In[36]:=
```
Clear[Weights];
Weights = Outer[Times,Tv,Tv] + Outer[Times,Iv,Iv];
```

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

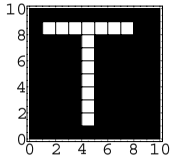
In[38]:=
```
forgettingT =
Join[Take[Tv,(Dimensions[Tv][[1]]-30)],Table[0, {30}]];
ListDensityPlot[Partition[forgettingT,size]];
```



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.**

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

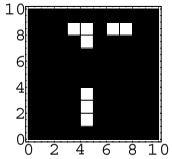
In[40]:=
```
rememberingT = Weights.forgettingT;
ListDensityPlot[Partition[rememberingT,size]];
```



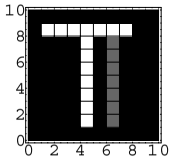
■ **Interference: Corrupt T again, this time with some other random bits missing**

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

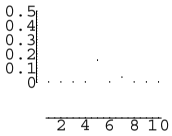
In[42]:=
```
pepper = Table[Random[Integer,1],{Dimensions[Tv][[1]]}];
peppermatrix = DiagonalMatrix[pepper];
forgettingT = peppermatrix.Tv;
ListDensityPlot[Partition[forgettingT,size]];
```



In[46]:=
```
rememberingT = Weights.forgettingT;
ListDensityPlot[Partition[rememberingT,size]];
```



In[48]:=
```
ListPlot[Partition[rememberingT,size][[5]],
    PlotRange->{0,.5},AxesOrigin->{0,-.25}];
```



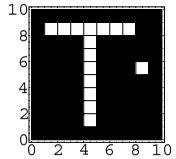
■ **Note:** You can get information about the options as well as the functions with a **??** query:

In[49]:=
```
??ListPlot
??AxesOrigin
```

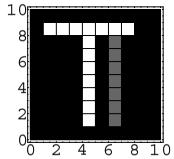
```
ListPlot[{y1, y2, ... }] plots a list of
   values. The x coordinates for each point are taken to be
   1, 2, ... . ListPlot[{{x1, y1}, {x2, y2}, ... }] plots a
   list of values with specified x and y coordinates. More...

Attributes[ListPlot] = {Protected}

Options[ListPlot] =
 {AspectRatio → 1/GoldenRatio, 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. More...

Attributes[AxesOrigin] = {Protected}
```

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

In[51]:=
```
forgettingT = Tv;
forgettingT[[59]] = .27;
ListDensityPlot[Partition[forgettingT,size]];
```



In[54]:=
```
rememberingT = Weights.forgettingT;
ListDensityPlot[Partition[rememberingT,size]];
```



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:

In[56]:=
```
Tv.normalize[rememberingT]
```

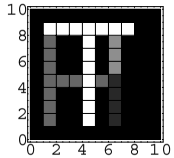
Out[56]= 0.995682

■ **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**:

In[57]:=
```
Weights = Weights +
            Outer[Times,Pv,Pv];
```

In[58]:=
```
rememberingT = Weights.forgettingT;
ListDensityPlot[Partition[rememberingT,size]];
```



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

In[60]:=
```
{Iv.Pv, Tv.Pv, Tv.Iv}
```

Out[60]=
```
{0.243332, 0.367884, 0.0944911}
```

Exercise - Include a threshold. Applying a function over a list

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

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

■ Change the function attributes

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

In[61]:=
```
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:

In[62]:=
```
step[rememberingT]
```

■ Map the function over the list

This means that to apply **step[]** to **rememberingT**, you would have to use the **Map[]** function:

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

In[63]:=
```
Map[f,{a,b,c}]
```

In[64]:=
```
f /@ {a,b,c}
```

In[65]:=
```
Map[step,rememberingT];
```

Map[step, rememberingT] is equivalent to **step /@ rememberingT :**

In[66]:=
```
ListDensityPlot[Partition[step /@ rememberingT, size]]
```

The threshold choice was clearly important. How could you make the threshold adaptive?

Next time

■ Explore a simple application, summed vector memories.

■ Neural networks as statistical pattern processing

Then we are going to step back in a sense, and 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.

Summed vector memories will lead to a discussion of simple ("first-order") statistical learning.

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 learning 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 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.

Appendix

Autoassocation with some other patterns: Einstein, Bush, or Shannon

In[67]:=
```
normalize[x_] := N[x / Sqrt[x.x]];

getinputpattern := Module[{image},
   image = Import[Experimental`FileBrowse[False]];
   image = image /. Graphics → List;
   image = N[image[[1, 1]]]];
```

Get **einstein32x32.jpg**:

In[69]:=
```
einstein = getinputpattern;
size = Dimensions[einstein][[1]];
einstein = normalize[Flatten[einstein]];
```

Get **gw_bush32x32.jpg** (or shannon32x32.jpg):

In[72]:=
```
bush = getinputpattern;
bush = normalize[Flatten[bush]];
```

Get einsten with some blacked out regions, **einstein32x32missing.jpg**


```
forgettingeinstein = getinputpattern;
forgettingeinstein = normalize[Flatten[forgettingeinstein]];
```

Superimpose the outerproduct weight matrices:

```
Weights = Outer[Times,einstein,einstein] + Outer[Times,bush,bush];
```

```
rememberingeinstein = Weights.forgettingeinstein;
ListDensityPlot[Partition[rememberingeinstein,size],Mesh->False];
```

```
ListDensityPlot[Partition[bush,size],Mesh->False];
```