Introduction to Neural Networks
U. Minn. Psy 5038

Lecture 8
Examples of heteroassociation and autoassociation

Initialization

Introduction

Heteroassociation

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.**

$$x = \begin{pmatrix} f_1 \\ f_2 \\ . \\ . \\ f_m \\ g_1 \\ . \\ . \\ . \\ g_n \end{pmatrix} \tag{6}$$

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[28]:=
```
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[32]:=
```
x = f + g
```

Out[32]=
```
{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[33]:=
```
Simplify[W.f]
```

Out[33]=

$$\{f1\,(f1^2+f2^2+f3^2),\ f2\,(f1^2+f2^2+f3^2),\ f3\,(f1^2+f2^2+f3^2),\ (f1^2+f2^2+f3^2)\,g1,\ (f1^2+f2^2+f3^2)\,g2,\ (f1^2+f2^2+f3^2)\,g3,\ (f1^2+f2^2+f3^2)\,g4\}$$

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[34]:=
```
% /. (f1^2 + f2^2 + f3^2) → α
```

Out[34]=

$$\{f1\,\alpha,\ f2\,\alpha,\ f3\,\alpha,\ g1\,\alpha,\ g2\,\alpha,\ g3\,\alpha,\ g4\,\alpha\}$$

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**

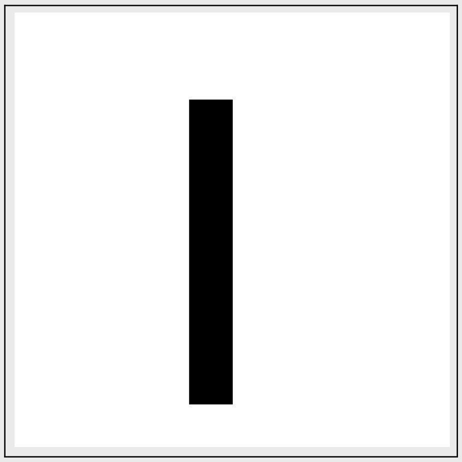
In[35]:=
```
Clear[Weights];
Weights = Outer[Times,Tv,Tv] + Outer[Times,Iv,Iv];
```

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

Let's delete the 2nd row of T:

In[37]:=
```
forgettingTmatrix = Partition[Tv, size];
forgettingTmatrix[[2]] = Table[0, {10}];
forgettingT = Flatten[forgettingTmatrix];
ArrayPlot[Partition[forgettingT, size]]
```

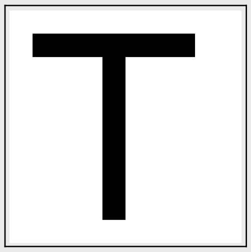
Out[40]=



■ Recall of the original T, from the missing bits

In[41]:=
```
rememberingT = Weights.forgettingT;
ArrayPlot[Partition[rememberingT, size]]
```

Out[41]=

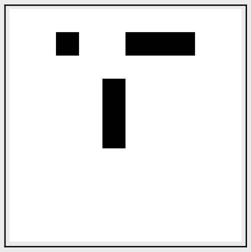


■ Interference: Corrupt T again, this time with some other random bits missing

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

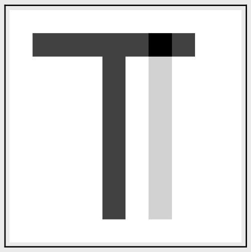
In[42]:=
```
pepper = RandomInteger[1, Dimensions[Tv][[1]]];
peppermatrix = DiagonalMatrix[pepper]; forgettingT = peppermatrix.Tv;
ArrayPlot[Partition[forgettingT, size]]
```

Out[42]=



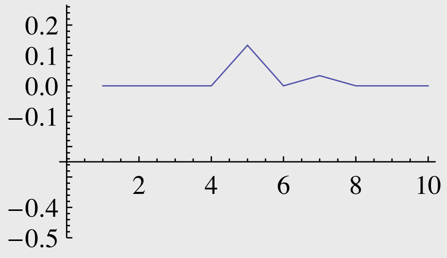
In[43]:=
```
rememberingT = Weights.forgettingT;
ArrayPlot[Partition[rememberingT, size]]
```

Out[43]=



In[44]:=
```
ListPlot[Partition[rememberingT, size][[5]], AxesOrigin → {0, -0.25},
  PlotRange → {-.5, maxi}, Joined → True]
```

Out[44]=



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

In[45]:=
```
??AxesOrigin
```

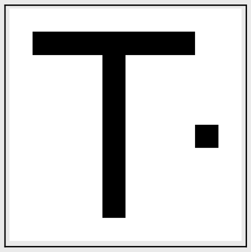
AxesOrigin is an option for two-dimensional graphics
    functions which specifies where any axes drawn should cross. ≫

```
Attributes[AxesOrigin] = {Protected}
```

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

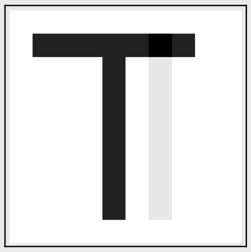
In[46]:=
```
forgettingT = Tv; forgettingT[[59]] = 0.27;
ArrayPlot[Partition[forgettingT, size]]
```

Out[46]=



In[47]:=
```
rememberingT = Weights.forgettingT;
ArrayPlot[Partition[rememberingT, size]]
```

Out[47]=



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[48]:=
```
Tv.Normalize[rememberingT]
```

Out[48]=
```
0.995682
```

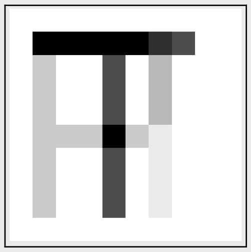
■ Interference with more autoassociations: I, T, and now P too

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[49]:=
```
Weights = Weights +
            Outer[Times,Pv,Pv];
```

In[50]:=
```
rememberingT = Weights.forgettingT;
ArrayPlot[Partition[rememberingT, size]]
```

Out[50]=

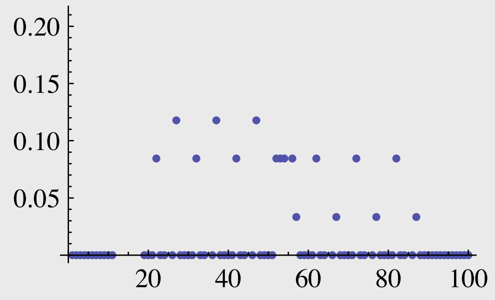


This is because the patterns we've stored are not mutually orthogonal, and in particular, **P** is too close to **I** and **T**:

In[51]:=
```
{Iv.Pv, Tv.Pv, Tv.Iv}
```
Out[51]=
```
{0.243332, 0.367884, 0.0944911}
```

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

In[52]:=
```
ListPlot[rememberingT]
```
Out[52]=



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.

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

In[53]:=
```
step[x_, t_] := If[x > t, 1, 0.0];
```

■ Change the function attributes

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

In[54]:=
```
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[55]:=
```
step[rememberingT,0]
```
Out[55]=
```
{0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1, 1, 1, 1, 1, 1, 1, 0., 0., 0., 1, 0., 0., 1, 0., 1, 0., 0., 0., 0., 1, 0., 0., 1,
  0., 1, 0., 0., 0., 0., 1, 0., 0., 1, 0., 1, 0., 0., 0., 0., 1, 1, 1, 1, 1, 1, 0., 0., 0., 0., 1, 0., 0., 1, 0., 1, 0.,
  0., 0., 0., 1, 0., 0., 1, 0., 1, 0., 0., 0., 0., 1, 0., 0., 1, 0., 1, 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.}
```

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[56]:= Clear[f,a,b,c];
         Map[f,{a,b,c}]
```

```
Out[57]= {f(a), f(b), f(c)}
```

```
In[58]:= f /@ {a,b,c}
```

```
Out[58]= {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 &:

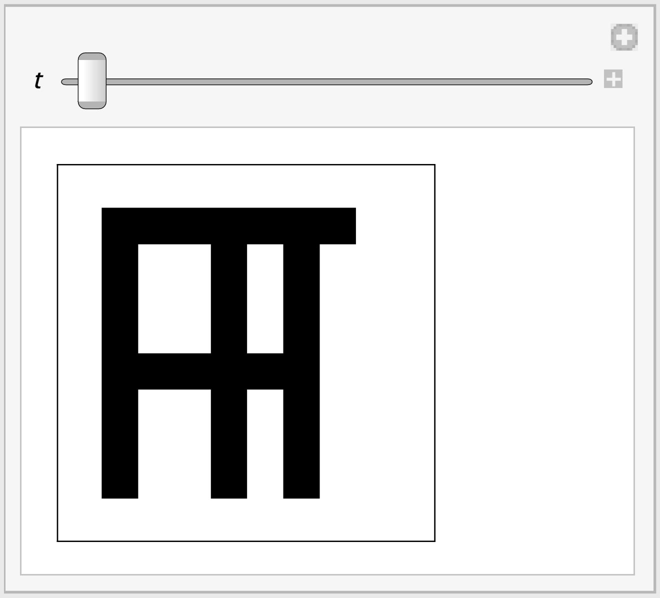
```
In[59]:= Map[step[#,0]&,rememberingT];
```

Using the short-hand version:

```
In[60]:= step[#, 0] & /@ rememberingT;
```

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

In[61]:=
```
Manipulate[ArrayPlot[Partition[step[#, t] & /@ rememberingT, size]],
 {t, 0, 1}]
```

Out[61]=


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

Autoassociation with pictures

Autoassocation with some other patterns: Einstein or Shannon

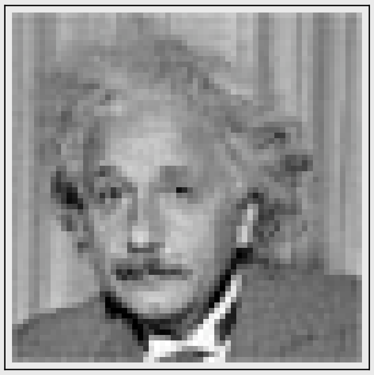
Get **einstein64x64.jpg**:

In[62]:=
```
ieinstein = Import[SystemDialogInput["FileOpen"], "Data"];
```

In[63]:=
```
size2 = Dimensions[ieinstein][[1]];
```

In[64]:=
```
geinstein = ArrayPlot[ieinstein, ColorFunction → GrayLevel]
einstein = N[Normalize[Flatten[ieinstein]]];
```

Out[64]=

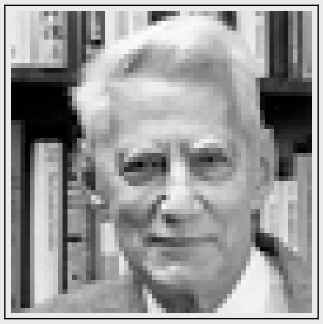


Get **shannon64x64.jpg**:

In[66]:=
```
ishannon = Import[SystemDialogInput["FileOpen"], "Data"];
```

In[67]:=
```
gshannon = ArrayPlot[ishannon, ColorFunction → GrayLevel]
shannon = N[Normalize[Flatten[ishannon]]];
```

Out[67]=



Autoassociatively store einstein

In[69]:=
```
Weights = Outer[Times,einstein,einstein];
```

In[70]:=
```
Dimensions[einstein]
```

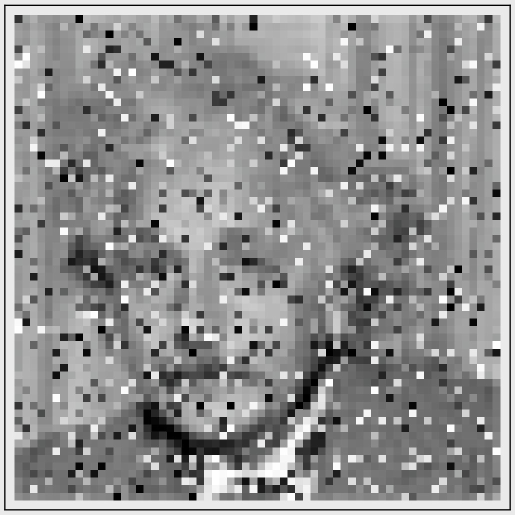
Out[70]=
```
{4096}
```

How big is Weights?

Get einsten with some blacked out regions, **einstein64x64missing.jpg**

In[71]:=
```
forgettingeinstein = einstein;
Table[forgettingeinstein[[RandomInteger[{1, 4096}]]] =
   RandomReal[{0.0, Max[einstein]}], {1000}];
gforgettingeinstein = ArrayPlot[Partition[forgettingeinstein, size2],
  ColorFunction → GrayLevel]
```

Out[73]=

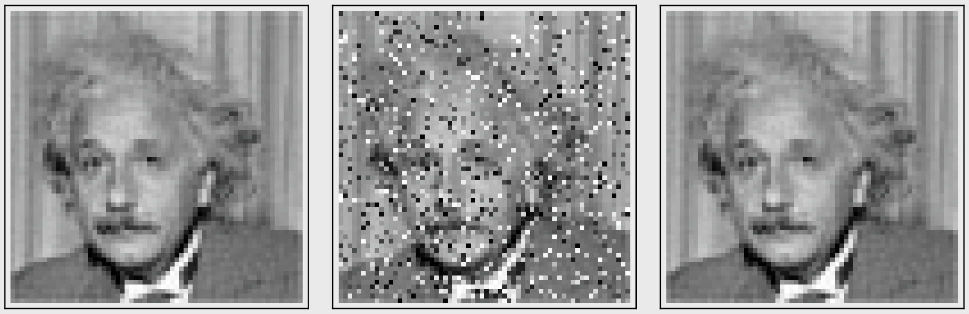


Recall einstein, with only a partial einstein, i.e. with forgettingeinstein as input:

In[74]:=
```
rememberingeinstein = Weights.forgettingeinstein;
grememberingeinstein = ArrayPlot[Partition[rememberingeinstein,size2], Colo
```

In[76]:=
```
Show[GraphicsRow[{geinstein, gforgettingeinstein, grememberingeinstein}]]
```

Out[76]=



Now superimpose the outerproduct weight matrices for einstein and shannon:

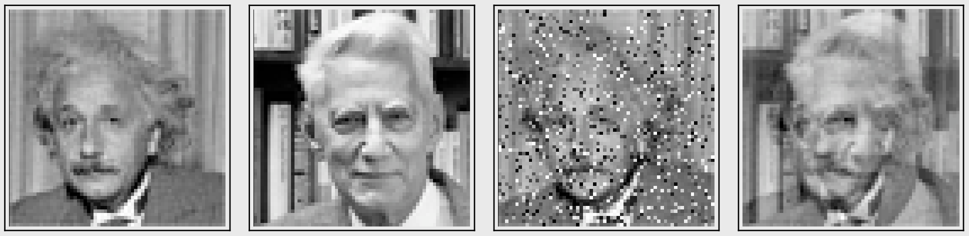
In[77]:=
```
deltaWeights = Outer[Times,shannon,shannon];
```

In[78]:=
```
Weights = Weights + deltaWeights;
```

In[79]:=
```
rememberingeinstein = Weights.forgettingeinstein;
grememberingeinstein = ArrayPlot[Partition[rememberingeinstein,size2], Colo
```

In[81]:=
```
Show[GraphicsRow[{geinstein, gshannon, gforgettingeinstein,
   grememberingeinstein}]]
```

Out[81]=



Next time