Introduction to Neural Networks

Heteroassociation and autoassociation

Initialization

```
In[94]:= Off[SetDelayed::write]
         Off[General::spell1]
         SetOptions[ListPlot, Joined → True];
         SetOptions[ArrayPlot, ColorFunction → "GrayTones",ImageSize->Tiny,Frame→False];
```

Introduction

Last time

Linear systems overview

Introduction to learning and memory

Outer product

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

```
In[98]:= Clear[f, g];
         Outer[Times, Array[g, 5], Array[f, 4]] // MatrixForm
```

Out[99]//MatrixForm=

$$\begin{pmatrix} f[1]\,g[1] & f[2]\,g[1] & f[3]\,g[1] & f[4]\,g[1] \\ f[1]\,g[2] & f[2]\,g[2] & f[3]\,g[2] & f[4]\,g[2] \\ f[1]\,g[3] & f[2]\,g[3] & f[3]\,g[3] & f[4]\,g[3] \\ f[1]\,g[4] & f[2]\,g[4] & f[3]\,g[4] & f[4]\,g[4] \\ f[1]\,g[5] & f[2]\,g[5] & f[3]\,g[5] & f[4]\,g[5] \end{pmatrix}$$

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.

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

2. **Recall**

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

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

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

3. **Condition for perfect recall**

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

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

since,

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

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. 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*. Critically, 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[100]:= 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[101]:= Tmatrix = {
              {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
              {0, 1, 1, 1, 1, 1, 1, 1, 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, 0, 0}};
```

```
In[102]:= Pmatrix = {
              {0, 0, 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, 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, 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.)

```
In[103]:= Tv = N[Normalize[Flatten[Tmatrix]]];
          Iv = N[Normalize[Flatten[Imatrix]]];
          Pv = N[Normalize[Flatten[Pmatrix]]];
          size = Dimensions[Imatrix][[1]];
          maxv = Max[Flatten[Tmatrix]];
          maxi = Max[Tv];
```

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

You’ve already had practice with Flatten[] in the first assignment. And you may have had experience with Matlab and Python’s reshape(), and numpy.reshape() functions. **Flatten[]** takes a list of lists and turns it into a list of elements, that is, it removes all of the inner braces:

```
In[109]:= Flatten[{{a,b},{c,d}}]

Out[109]= {wlist⟦50, 1⟧, wlist⟦50, 2⟧, c, d}
```

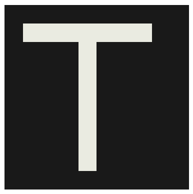
Partition[] does the reverse of **Flatten[]** and takes a list of elements and structures it back into a list of lists:

```
In[110]:= Partition[%,2]

Out[110]= {{wlist⟦50, 1⟧, wlist⟦50, 2⟧}, {c, d}}
```

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.
You can visualize an nxm matrix using MatrixPlot, ArrayPlot or Image. We'll use ArrayPlot:

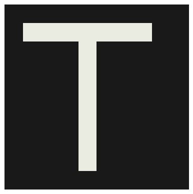
```
In[111]:= ArrayPlot[Tmatrix, PlotRange → {0, maxv}]
```

Out[111]=



```
In[112]:= ArrayPlot[Partition[Tv, size], PlotRange → {0, maxi}, Mesh → False]
```

Out[112]=



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:

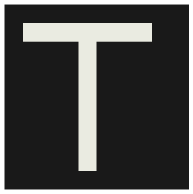
```
In[113]:= weights = Outer[Times,Tv,Iv];

In[114]:= Dimensions[weights]

Out[114]= {100, 100}
```

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[115]:=
```
response = weights.Iv;
ArrayPlot[Partition[response, size], Mesh → False]
```

Out[116]=



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

Exercises

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

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

What if you right-multiply **Tv** by **weights**? (right-multiply means you put the matrix to the right of the vector, i.e. $Tv . (T_v . I_v^T)$)

Add a second association T <--> P to the mix

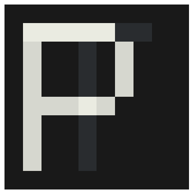
Learning P from T, storing it with the association between I and T

In[117]:=
```
weights = weights + Outer[Times,Pv,Tv];
```

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

In[118]:=
```
response = weights.Tv;
ArrayPlot[Partition[response, size]]
```

Out[119]=



Exercise

You should see 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 ArrayPlot 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.

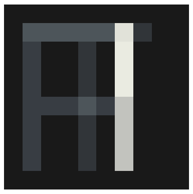
Add a third association I <--> P to the mix

Store it with the associations between I & T, P & T

```
In[120]:= weights = weights + Outer[Times,Iv,Pv];
```

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

```
In[121]:= response = weights.Pv; ArrayPlot[Partition[response, size]]
```

Out[121]=


We see increased interference during recall. But we might have expected this given that the inputs are not orthogonal.

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).

Let’s see 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.

Suppose a vector **x** has two sets of elements shared by parts of vectors **f** and **g**.

$$x = \begin{pmatrix} f_1 \\ f_2 \\ . \\ . \\ f_m \\ 0 \\ . \\ . \\ . \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ . \\ . \\ 0 \\ g_1 \\ . \\ . \\ . \\ g_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ . \\ . \\ f_m \\ g_1 \\ . \\ . \\ . \\ g_n \end{pmatrix}$$

We calculate $x.x^T$, and then later compute: $(x.x^T)$.f

What do we get?

Let’s try this with **x**, consisting 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[122]:= 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[126]:= x = f + g

Out[126]= {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[127]:= Simplify[W.f]
```

$$\text{Out[127]=} \{f1 (f1^2 + f2^2 + f3^2), f2 (f1^2 + f2^2 + f3^2), f3 (f1^2 + f2^2 + f3^2), (f1^2 + f2^2 + f3^2) g1, (f1^2 + f2^2 + f3^2) g2, (f1^2 + f2^2 + f3^2) g3, (f1^2 + f2^2 + f3^2) g4\}$$

If we replace (f1^2+f2^2+f3^2) by α, 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 α:

```
In[128]:= % /. (f1^2 + f2^2 + f3^2) → α

Out[128]= {f1 α, f2 α, f3 α, g1 α, g2 α, g3 α, g4 α}
```

If **f** was initially normalized to 1, then **W.f** is would be equal to **x.**

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 suppose we have an input/output pair {**f**, **g**}. 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**.

```
In[129]:= Clear[f, g];
          fprime = Join[Array[f, 3], Array[g, 3]];

          Outer[Times, fprime, fprime] // MatrixForm
```

Out[131]//MatrixForm=

$$\begin{pmatrix} f[1]^2 & f[1]\,f[2] & f[1]\,f[3] & f[1]\,g[1] & f[1]\,g[2] & f[1]\,g[3] \\ f[1]\,f[2] & f[2]^2 & f[2]\,f[3] & f[2]\,g[1] & f[2]\,g[2] & f[2]\,g[3] \\ f[1]\,f[3] & f[2]\,f[3] & f[3]^2 & f[3]\,g[1] & f[3]\,g[2] & f[3]\,g[3] \\ f[1]\,g[1] & f[2]\,g[1] & f[3]\,g[1] & g[1]^2 & g[1]\,g[2] & g[1]\,g[3] \\ f[1]\,g[2] & f[2]\,g[2] & f[3]\,g[2] & g[1]\,g[2] & g[2]^2 & g[2]\,g[3] \\ f[1]\,g[3] & f[2]\,g[3] & f[3]\,g[3] & g[1]\,g[3] & g[2]\,g[3] & g[3]^2 \end{pmatrix}$$

Note that we are only representing pair-wise relationships (through products) between elements in the vectors. Later we'll see the connection to "correlational structure" and "2nd order statistics".

Question:

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

Hint: Is the autoassociatve matrix symmetric?

In general, is the heterassociative matrix symmetric?

Autoassociative example with TIP pictures

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

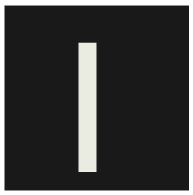
```
In[132]:= Clear[weights];
          weights = Outer[Times,Tv,Tv] + Outer[Times,Iv,Iv];
```

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

Let's delete the 2nd row of T:

```
In[134]:= forgettingTmatrix = Partition[Tv, size];
          forgettingTmatrix[[2]] = Table[0, {10}];
          forgettingT = Flatten[forgettingTmatrix];
          ArrayPlot[Partition[forgettingT, size]]
```

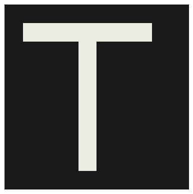
Out[137]=



Recall of the original T, from the missing bits

```
In[138]:= rememberingT = weights.forgettingT; ArrayPlot[Partition[rememberingT, size]]
```

Out[138]=

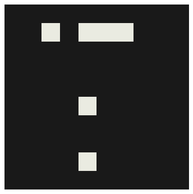


Interference: Corrupt T again, this time with some other random bits missing

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

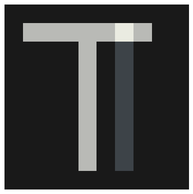
```
In[139]:= pepper = RandomInteger[1, Dimensions[Tv][[1]]]; peppermatrix = DiagonalMatrix[pepper];
forgettingT = peppermatrix.Tv; ArrayPlot[Partition[forgettingT, size]]
```

Out[139]=



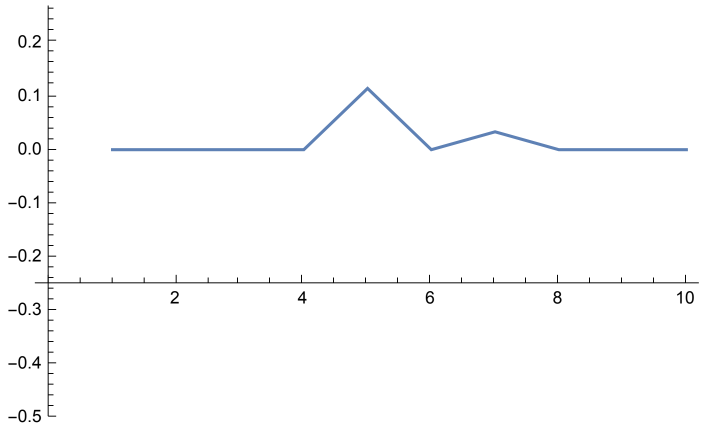
```
In[140]:= rememberingT = weights.forgettingT; ArrayPlot[Partition[rememberingT, size]]
```

Out[140]=



```
In[141]:= ListPlot[Partition[rememberingT, size][[5]],
  AxesOrigin → {0, -0.25}, PlotRange → {-.5, maxi}, Joined → True]
```

Out[141]=



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

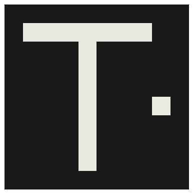
In[142]:=
```
??AxesOrigin
```

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

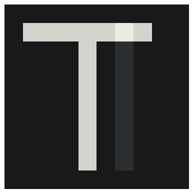
```
Attributes[AxesOrigin] = {Protected}
```

Interference: Corrupt T, with added noise

In[143]:=
```
forgettingT = Tv; forgettingT[[59]] = 0.27; ArrayPlot[Partition[forgettingT, size]]
```

Out[143]=


In[144]:=
```
rememberingT = weights.forgettingT; ArrayPlot[Partition[rememberingT, size]]
```

Out[144]=


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[145]:=
```
Tv.Normalize[rememberingT]
```

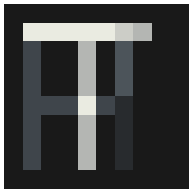
Out[145]=
```
0.995682
```

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[146]:=
```
weights = weights +
            Outer[Times,Pv,Pv];
```

In[147]:=
```
rememberingT = weights.forgettingT; ArrayPlot[Partition[rememberingT, size]]
```

Out[147]=


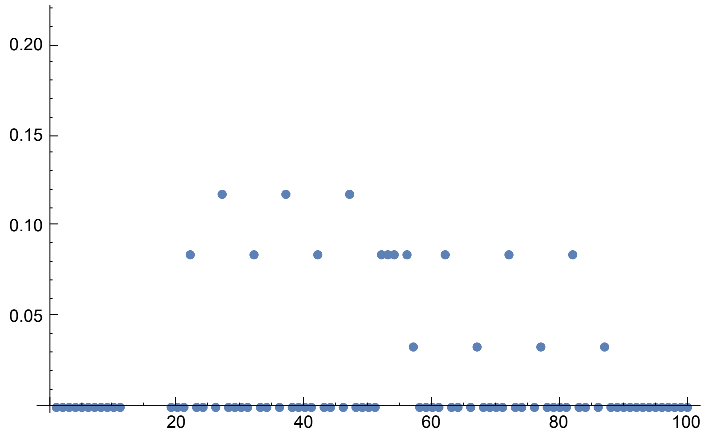
This is because the patterns we've stored are not mutually orthogonal, and in particular, **P** is too close to **I** and **T**:

```
In[148]:= {Iv.Pv, Tv.Pv, Tv.Iv}

Out[148]= {0.243332, 0.367884, 0.0944911}
```

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

```
In[149]:= ListPlot[rememberingT, Joined → False]
```

Out[149]=



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[150]:= 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[151]:= 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[152]:= rememberingT

Out[152]= {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.,
 0.267261, 0., 0.117806, 0., 0., 0., 0., 0.0843983, 0., 0., 0.267261, 0.,
 0.117806, 0., 0., 0., 0., 0.0843983, 0., 0., 0.267261, 0., 0.117806, 0., 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.,
 0.0843983, 0., 0., 0.267261, 0., 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.}

In[153]:= step[rememberingT,0]

Out[153]= {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.}
```

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

Out[155]= {f[a], f[b], f[c]}
```

```
In[156]:= f /@ {a,b,c}

Out[156]= {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[157]:= Map[step[#,0]&,rememberingT];
```

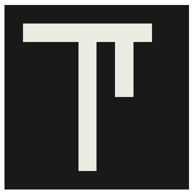
Using the short-hand version:

```
In[158]:= step[#, 0] & /@ rememberingT;
```

```
In[159]:= Dimensions[Partition[step[#, .1] & /@ rememberingT, 10]]
          Dimensions[Flatten[Partition[step[#, .1] & /@ rememberingT, 10]]]
          Flatten[Partition[step[#, .1] & /@ rememberingT, 10]]

Out[159]= {10, 10}

Out[160]= {100}

Out[161]= {0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1, 1, 1, 1, 1, 1, 1, 0., 0., 0.,
           0., 0., 0., 1, 0., 1, 0., 0., 0., 0., 0., 0., 0., 1, 0., 1, 0., 0., 0., 0.,
           0., 0., 0., 1, 0., 1, 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., 0., 0.}
```

```
In[162]:= ArrayPlot[Partition[step[#, .1] & /@ rememberingT, 10]]
```

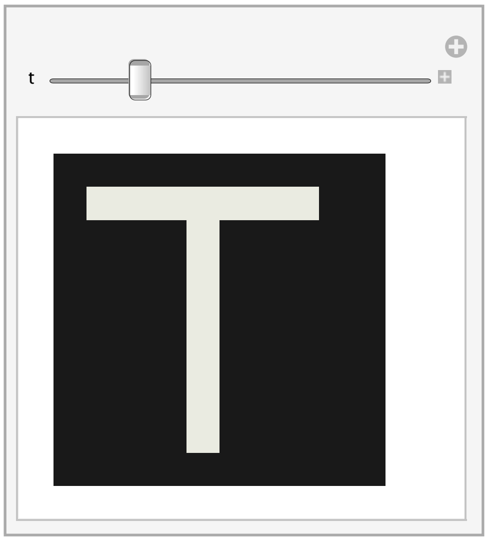
Out[162]=



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

```
In[163]:= Manipulate[ArrayPlot[
            Partition[step[#, t] & /@ rememberingT, size], ImageSize → Small], {t, 0, 1}]
```

Out[163]=



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

Later we'll look at a non-linear recall mechanism that does a better job of restoring the original input from a matrix of memories.

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.

Reading in images

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

```
In[164]:= Import["http://gandalf.psych.umn.edu/users/kersten/kersten-lab/courses/
              NeuralNetworksKoreaUF2011/MathematicaNotebooks/Lect_8_HeterAuto/
              einstein64x64.jpg"];
```

```
In[165]:= Import["http://gandalf.psych.umn.edu/users/kersten/kersten-lab/courses/
              NeuralNetworksKoreaUF2011/MathematicaNotebooks/Lect_8_HeterAuto/
              shannon64x64.jpg"];
```

You can import a file, such as einstein.jpg with a dialog
box:

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

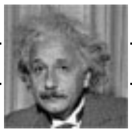
with the size extracted with:

size = Dimensions[ieinstein][[1]];

Or you can just drag an image into an appropriate argument slot of a function. This is what we've done below.

Autoassocation with some other patterns: Einstein or Shannon

Turn einstein64x64.jpg into a 64x64 matrix of intensities:

```
In[166]:= ieinstein = ImageData[
```

```
];
```

```
In[167]:= size2 = Dimensions[ieinstein][[1]]
Out[167]= 64
```

```
In[168]:= geinstein = ArrayPlot[ieinstein, ColorFunction → GrayLevel]
          einstein = N[Normalize[Flatten[ieinstein]]];
Out[168]=
```

Turn shannon64x64.jpg into a 64x64 matrix of intensities:

```
In[170]:= ishannon = ImageData[
```

```
];
```

```
In[171]:= gshannon = ArrayPlot[ishannon, ColorFunction → GrayLevel]
          shannon = N[Normalize[Flatten[ishannon]]];
Out[171]=
```

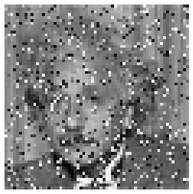
Autoassociatively store einstein

```
In[173]:= weights = Outer[Times,einstein,einstein];
```

How big is weights?

Make an einstein picture with some "salt and pepper noise"--random intensities at random locations, **einstein64x64missing.jpg:**

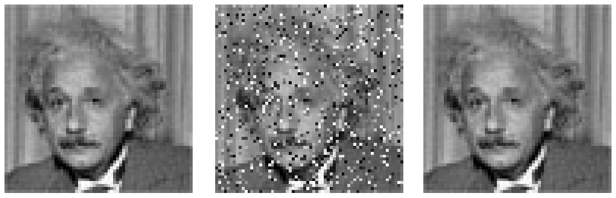
In[175]:=
```
forgettingeinstein = einstein;
Table[forgettingeinstein[[RandomInteger[{1, 4096}]]] =
   RandomReal[{0.0, Max[einstein]}], {1000}];
gforgettingeinstein = ArrayPlot[Partition[forgettingeinstein, size2],
  ColorFunction → GrayLevel]
```

Out[177]=


Recall einstein, with only a part of the original pixels in einstein, i.e. with forgettingeinstein as input:

In[178]:=
```
rememberingeinstein = weights.forgettingeinstein;
grememberingeinstein = ArrayPlot[Partition[rememberingeinstein,size2], ColorFunction → Gra
```

In[180]:=
```
Show[GraphicsRow[{geinstein, gforgettingeinstein, grememberingeinstein}]]
```

Out[180]=


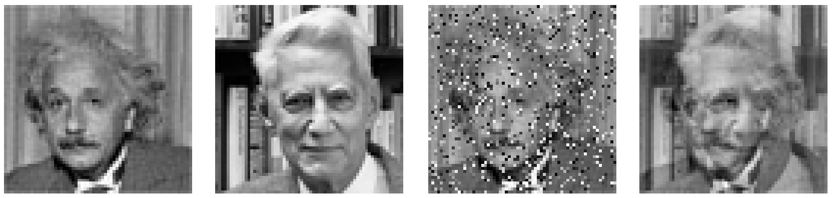
Now superimpose the outerproduct weight matrices for einstein and shannon:

In[181]:=
```
deltaweights = Outer[Times,shannon,shannon];
```

In[182]:=
```
weights = weights + deltaweights;
```

In[183]:=
```
rememberingeinstein = weights.forgettingeinstein;
grememberingeinstein = ArrayPlot[Partition[rememberingeinstein,size2], ColorFunction → Gra
```

In[185]:=
```
Show[GraphicsRow[{geinstein, gshannon, gforgettingeinstein, grememberingeinstein}]]
```

Out[185]=


We see there is considerable interference with just two pictures. And we only have two images so far!

How could the images be recoded so as to reduce the interference?

Try replacing einstein and shannon with edge representations, e.g. using EdgeDetect[]. What if you computed the spectra of each image and used these as inputs to your outer product learning rule?

Where are we headed?

The above simulations helped to show how far we could with a hebbian rule for storage, and a linear rule for recall. The answer is “not very far”.

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 and storage, so that linear recall will do better; 2) improve the recall mechanism, 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.

Neural networks as statistical pattern processing

We are going to step back in a sense, and ask ourselves what these networks are doing in the sense of information or statistical pattern processing. This will lead naturally to a framework for understanding both supervised and unsupervised learning networks from a probablistic perspective, and machine learning.

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 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.

Next time

Introduction to non-linear networks and classifiers. Demonstrate learning and classification with a single-layer perceptron.

In the following lecture, we'll revisit today's linear model but with an improved learning rule, called Widrow-Hoff. This will allow us to make direct ties to traditional methods of regression. Further, an extension of this rule to the "error back-propation" rule will enable us to learn weights in neural networks with more than one layer of weights, multiple layer perceptrons, that can solve non-linear problems of both regression and classification.