Introduction to Neural Networks
U. Minn. Psy 5038

Lecture 10

Non-linear models
The perceptron

Initialization

In[1]:=

```
Off[SetDelayed::write]
Off[General::spell1]
```

Introduction

Last time

- **Summed vector memories**

- **Introduction to statistical learning**

- **Generative modeling and statistical sampling**

Today

- **Non-linear models for classification**

Introduction to non-linear models

By definition, linear models have several limitations on the class of functions they can compute--outputs have to be linear functions of the inputs. However, as we have pointed out earlier, linear models provide an excellent foundation on which to build. On this foundation, non-linear models have moved in several directions.

Consider a single unit with output y, and inputs f_i. One way is to "augment" the richness of the input patterns with higher-order terms to form polynomial mappings, or non-linear regression, as in a Taylor series (Poggio, 1979), going from linear, to quadratic, to higher order functions:

$$y = \sum w_i f_i$$

$$y = \sum w_i f_i + \sum w_{i,j} f_i f_j$$

$$y = \sum w_i f_i + \ldots + \sum w_{i_1,\ldots,i_n} f_{i_1} f_{i_n}$$

The linear Lateral inhibition equations can be generalized using products of input and output terms --"shunting" inhibition (Grossberg).

$$\frac{dy_i}{dt} = -\alpha y_i + (\beta - y_i) f_i - y_i \sum_{i \neq j} w_{ij} f_j$$

A straightforward generalization of the generic connectionist model is to divide the neural output by the squared responses of neighboring units. This is a steady-state model which has been very successful in accounting for a range of neurophysiological receptive field properties in vision (Heeger et al., 1996).

One of the simplest things we can do at this point is to use the generic connectionist neuron with its second stage point-wise non-linearity. Recall that this is an inner product followed by a non-linear sigmoid. Once a non-linearity such as a sigmoid introduced, it makes sense to add more than additional layers of neurons. Much of the modeling of human visual pattern discrimination has used just these "rules-of-the-game", with additional complexities (such as a normalization term above) added only as needed.

A central challenge in the above and all methods which seek general mappings, is to develop techniques to learn the weights, while at the same time avoiding over-fitting (i.e. using too many weights). We'll talk more about this problem later.

These modifications produce smooth functions. If we want to classify rather than regress, we need something abrupt. Generally, we add a sigmoidal squashing function. As the slope of the sigmoid increases, we approach a simple step non-linearity. The neuron then makes discrete (binary) decisions. Recall the McCulloch-Pitts model of the 1940's. Let us look at the Perceptron, an early example of a network built on such threshold logic units.

Classification and the Perceptron

Classification

Previously we introduced the distinction between regression and classification in supervised learning.

Supervised learning: Training set $\{\mathbf{f}_i,\mathbf{g}_i\}$

Regression: Find a function ϕ: **f->g**, i.e. where **g** takes on continuous values.

Classification: Find a function ϕ:**f->{0,1,2,...,n}**, i.e. where $\mathbf{g}_i$ takes on discrete values or labels.

Linear networks can be configured to be supervised or unsupervised learning devices. But linear mappings are severely limited in what they can compute. The specific problem that we focus on in this lecture is that continuous linear mappings don't make discrete decisions.

Let's take a look at the binary classification problem

ϕ: **f** -> {0,1}

We will study both recall (i.e. classification) and learning for the binary classification problem. Learning amounts to adjusting the parameters (e.g. synaptic weights) of the mapping in order to achieve the best classification performance. We will make these learning requirements more precise as we go along.

Pattern classification

One often runs into situations in which we have input patterns with enormous dimensionality, and whose elements are perhaps continuous valued. What we would like to do is classify all members of a certain type. Suppose that **f** is a representation of one of the following 10 input patterns,

{a, A, a, a, A, b, B, b, b, B }

A pattern classifier should make a decision about **f** as to whether it means "a" or "b", regardless of the font type, face or size. In other words, we require a mapping T such that:

T: **f** -> {"a","b"}

In general, a classifier should show *invariance* over variations of instances within each class, while at the same time minimizing the proportion of misclassification errors.

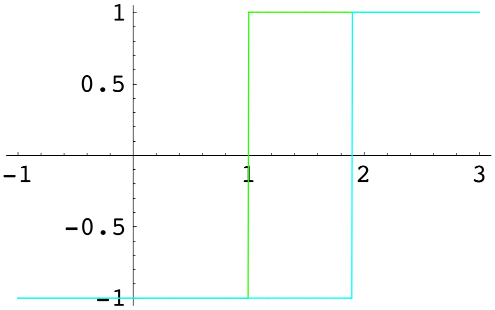
As mentioned above, one of the simplest ways of extending the linear neuron computing element is to include a step threshold function in the tradition of McCulloch & Pitts--the earliest computational model of the neuron. This is special case of the generic connectionist neuron model. With the step threshold, recall that the units are called **Threshold Logic Units** (TLU):

TLU: **f** -> {-1,1}

```
In[3]:= step[x_, θ_] := If[x<θ,-1,1];
```

The TLU neuron's output can be modeled simply as: **step[w.f,θ];**

```
In[4]:=  Plot[{step[x, 1], step[x, 1.9]}, {x, -1, 3}, PlotStyle → {Hue[.3], Hue[.5]}];
```



Our goal will be to find the weight and threshold parameters for which the TLU does a correct classification.

So the TLU has two classes of parameters to learn, weight parameters and a threshold *θ*. If we can put the threshold degree of freedom into the weights, then the math is simpler--we'll only have to worry about learning weights. To see this, we assume the threshold to be fixed at zero, and then augment the inputs with one more input that is always on. Here is a two input TLU, in which we augment it with a third input that is always 1 and whose weight is **-***θ*:

```
In[42]:=  Remove[w,f,waug,faug,θ,w1,w2,f1,f2];
          w = {w1,w2};
          f = {f1,f2};
          waug = {w1,w2,-θ};
          faug = {f1,f2,1};
          θ==w.f
          Solve[0==waug.faug,θ]
```

```
Out[47]=  θ == f1 w1 + f2 w2
```

```
Out[48]=  {{θ → f1 w1 + f2 w2}}
```

So the 3-input augmented unit computes the same inner product as 2-input unit with arbitrary threshold. This is a standard trick that is used often to simplify calculations and theory (e.g. Hopfield network).

Perceptron (Rosenblatt, 1958)

The original perceptron models were fairly sophisticated. There were several layers of **TLU**s. In one early model (Anderson, page 217), there was:

1. An input layer or *retina* of sensory units

2. *Associator* units with lateral connections and

3. *Response* units, also with lateral connections.

Lateral connections between response units functioned as a "winner-take-all" mechanism to produce outputs in which only one response unit was on. (So was the output a distributed code of the desired response?)

The Perceptron in fact is a cartoon of the anatomy between the retina (if it consisted only of receptors, which it does not, it also has horizontal, bipolar, amacrine and ganglion cells), the lateral geniculate nucleus of the thalamus (if it had lateral connections, which if does have, are not a prominent feature) and the visual cortex with feedback from response to associator units, and the lateral connections (V1 cortex in fact does send neurons back to the lateral geniculate nucleus, and does have lateral inhibitory connections).

Perceptrons of this sort are too complex to analyze. It is difficult to to draw general theoretical conclusions about what they can compute and what they can learn. In a curious parallel, and long standing mystery in visual physiology, is the function of the feedback from cortex to thalamus. In order to make the Perceptron theoretically tractable, we will take a look at a simplified perceptron which has just two layers of units (input units and TLUs), and one layer of weights. There is no feedback and there are no lateral connections between units in the same layer. In a nutshell, there is one set of neural TLU elements that receive inputs and send their outputs.

What can this simplified perceptron do?

To simplify further, let's look at a single TLU with just two variable inputs, but three adjustable weights.

Recall performance and Linear separability

■ A two-input simplified perceptron

Assume we have some generative process that is providing data $\{\boldsymbol{f}^k\}$, where each $\boldsymbol{f}^k$ is a two-dimensional vector. So the data set can be represented in a scatter plot. But any particular input can belong to one of two categories, say -1 or 1: $\boldsymbol{f}^k$->{-1,1}. (-1 and 1 could correspond, for example, to "a" and "b".)

For a specific set of weights, the threshold defines a decision line separating one category from another. For a 3 input TLU, this decision line becomes a decision surface separating classes. If we solve **waug.faug==0** for **f2** symbolically, we have an expression for this boundary separating points {f1,f2}

In[52]:=

```
waug = {w1,w2,w3}; (*w3 = -θ*)
faug = {f1,f2,1};
Solve[waug.faug==0,{f2}]
```

Out[54]=

$$\left\{\left\{\mathrm{f2}\to\frac{-\mathrm{f1}\ \mathrm{w1}-\mathrm{w3}}{\mathrm{w2}}\right\}\right\}$$

We can see that f2 is a linear function of f1 for fixed weight values.

■ Define equation for the decision line of a two-input simplified perceptron

Let's write a function for the decision line:

In[55]:=

```
w1 = 0.5; w2 = 0.8; θ = 0.55;
waug = {w1,w2,-θ};
decisionline[f1_,θ_]:= -((-θ + f1*w1)/w2)
```

■ Generative model: Simulate data and network response for a two-input simplified perceptron

Now we generate some random input data and run it through the TLU. Because we've put the threshold variable into the weights, we re-define step[ ] to have a fixed threshold of zero:

In[58]:=

```
Remove[step];
step[x_] := If[x > 0, 1, -1];
```

In[60]:=

```
abovethreshold = Table[{x=Random[],y=Random[],
step[waug.{x,y,1}]},{i,1,20}];
```

We've made an array of 3-element vectors, in which each vector is: {x,y,TLUaug[{x,y}]}. TLUaug is our augmented TLU with the third input set to 1, and weight *θ*.

Question: Why did we use Table[{x=Random[],y=Random[], step[waug.{x,y,1}]},{i,1,20}], rather than Table[{Random[],Random[], step[waug.{Random[],Random[],1}]},{i,1,20}] above?

■ View the data and the responses

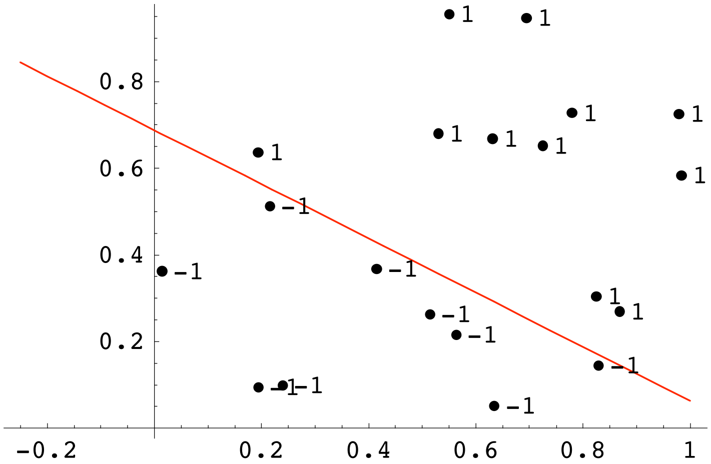
Let's read in the add-on graphics package to use the special plot function called **LabeledListPlot[]**: and plot the outputs and decision line in red.

In[61]:=

```
<<Graphics`Graphics`
```

In[62]:=
```
g1 = Plot[decisionline[f1, θ], {f1, -0.25, 1}, PlotStyle → {RGBColor[1, 0, 0]},
    DisplayFunction → Identity];

g2 = LabeledListPlot[abovethreshold, DisplayFunction → Identity];
Show[g1, g2, DisplayFunction → $DisplayFunction];
```



The red line separates inputs whose inner product with the weights exceeds a threshold of 0.4 (above) from those that do not exceed.

■ Simplified perceptron (TLU network): N-dimensional inputs

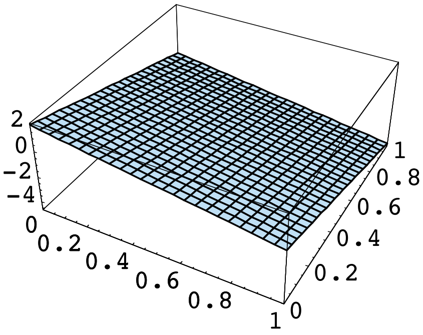
For a three dimensional input TLU, this decision surface is a plane:

In[65]:=
```
w1 = 0.5; w2 = 0.8; w3 = 0.2; θ = 0.4;
w = {w1,w2,w3,-θ};
f = {f1,f2,f3,1};
Solve[w.f==0,{f3}]
```

Out[68]=
```
{{f3 → 5. (−0.5 f1 − 0.8 f2 + 0.4)}}
```

In[69]:=

```
Plot3D[-5.*(-0.4 + 0.5*f1 + 0.8*f2), {f1,0,1},{f2,0,1}];
```



Exercise: Find the algebraic expression for the decision plane

---

For an n-dimensional input TLU, this decision surface is a hyperplane with all the members of one category falling on one side, and all the members of the other category falling on the other side. The hyperplane provides an intuition for the TLU's limited classification capability. For example, what if the features corresponding to the letter "a" fell inside of a circle or radius 1, and the features for "b" fell outside this circle?

Perceptron learning rule

Given a set of classified data, how can we find the perceptron parameters (weights) that specify a good decision plane?

A classic perceptron learning rule (that can be proved to converge) is as follows.

In[72]:=

```
Remove[w, f, c, w1, w2, θ];
w = {w1, w2, -θ}; f = {f1, f2, 1};
```

Suppose we have a training set $\{\mathbf{f}_i,\mathbf{g}_i\}$ where $\mathbf{g}_i$ is either -1 or +1. Imagine we are in the middle of the training,and we have a set of weights **w**. A new $\{\mathbf{f}_i,\mathbf{g}_i\}$ comes along, so we can check to see how our perceptron is doing. Suppose it predicts that the output for $\mathbf{f}_i$ should be $\hat{g}_i$. Then $\hat{g}_i$==$\mathbf{g}_i$ is either true or false. If false the classification is wrong and we can use this information to adjust the weights to make it more likely to get the correct answer the next time.

So we have two basic conditions to address:

If the classification is correct, don't change the weights:

Let **nextW** be the new (adjusted) set of weights:

In[51]:=

```
nextW = w;
```

If the classification is incorrect...

■ ..and the correct answer was +1

Suppose the classification is incorrect AND the response should have been +1. Instead the output was -1 because the inner product was less than zero. We change the weights to improve the chances of getting a positive output next time that input occurs by adding some positive fraction (**c**) of the input to the weights:

In[75]:=

```
nextw = w + c f;
```

Note that the new weights increase the likelihood of making a correct decision because the inner product is bigger than it was, and thus closer to exceeding the zero threshold. We can demonstrate that it is bigger by calculating the difference between the output with the adjusted weights, and the output with the original weights:

In[76]:=

```
Simplify[nextw.f - w.f]
```

Out[76]=

$$c\,(\mathrm{f1}^2 + \mathrm{f2}^2 + 1)$$

This is always positive. In general, **`nextw.f > w.f,`** because **`nextw.f - w.f = c f.f,`**

and **`c f.f > 0.`**

■ ..and the correct answer was -1

If the classification is incorrect AND the response should have been -1, we should change the weights by subtracting a fraction (c) of the incorrect input from the weights. The new weights decrease the likelihood of making a correct decision because the inner product is less, and thus closer to falling below threshold. So next time this input would be more likely to produce a -1 output.

In[78]:=

```
nextw = w - c f;
Simplify[nextw.f - w.f]
```

Out[79]=

$$-c\,(\mathrm{f1}^2 + \mathrm{f2}^2 + 1)$$

Note that the inner product **nextw.f** must now be smaller than before (**`nextw.f < w.f`**), because

nextw.f - w.f < 0

(since **`nextw.f - w.f = -c f.f`**, and **`c f.f > 0,`** as before).

If you are interested in understanding the proof of convergence, take a look at page 222 of the book by Anderson.

Demonstration of perceptron classification (Problem Set 3)

In the problem set you are going to write a program that uses a Perceptron style threshold logic unit (TLU) that learns to classify two-dimensional vectors into "a" or "b" types. The unit will have three inputs: {1,x,y}, where x and y are the coordinates of the data to be classified. The first component, 1 is there because we use the above "trick" used to incorporate the threshold into the weight vector. So three weights will have to be learned: {w1,w2,w3}, where the first can be thought of as the negative of the threshold. It may help to know something more about Conditionals in *Mathematica*.

■ Sidenote: More on conditionals

You have seen how to generate threshold functions using rules. But you can also use conditional statements. For example the following function returns x when Sin[2 Pi x] < 0.5, and returns -1 otherwise:

In[80]:=

```
pet[x_] := If[Sin[2 Pi x] <0.5, x,-1];
```

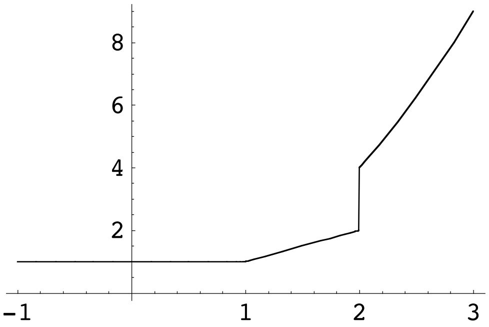
One can define a function over three regions using **Which[]**. **Which[**test1, value1, test2,value2,...**]** evaluates each test in turn, giving the value of the first one that is **True**:

In[81]:=

```
tep[x_] := Which[-1<=x<1,1,
                  1<=x<2, x,
                  2<=x<=3, x^2]
```

In[82]:=

```
Plot[tep[x],{x,-1,3}];
```



■ Generation of synthetic classification data.

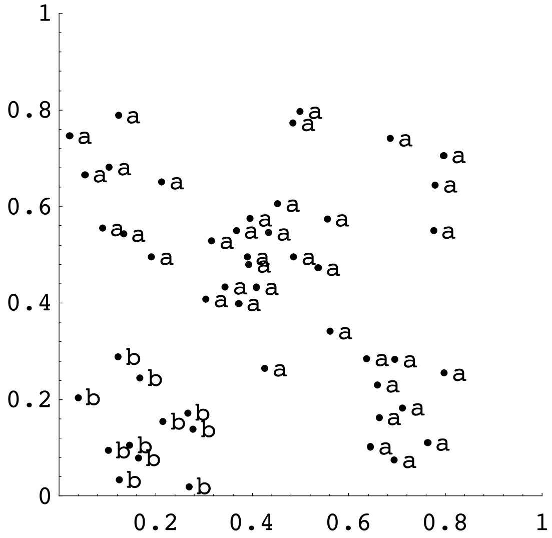
Suppose we generate 50 random points in the unit square, {{0,1},{0,1}} such that for the "a" type points,

x^2+y^2 > **bigradius**^2 and for the "b" points,

x^2+y^2 < **littleradius**.

Each pair of points has its corresponding label, a or b. Depending on the radius values (in this case, 0.25, 0.4), these patterns may or may not be linearly separable because they fall inside or outside their respective circles. The data are stored in **stuff** (Note, we haven't defined **stuff** in this Notebook, so don't try to evaluate the next line--but it could be useful for Problem 6 in PS3).

```
TextListPlot[
  Transpose[RotateLeft[Transpose[Map[Drop[#, {2}] &, stuff]], 1]],
  PlotRange -> {{0, 1}, {0, 1}},
    AspectRatio -> 1];
```



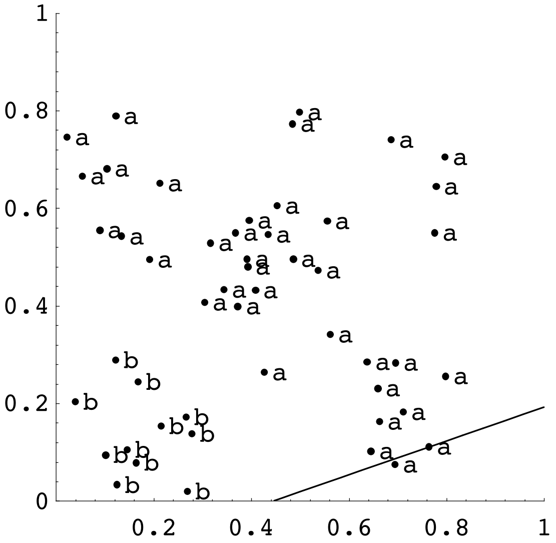
■ Perceptron learning algorithm

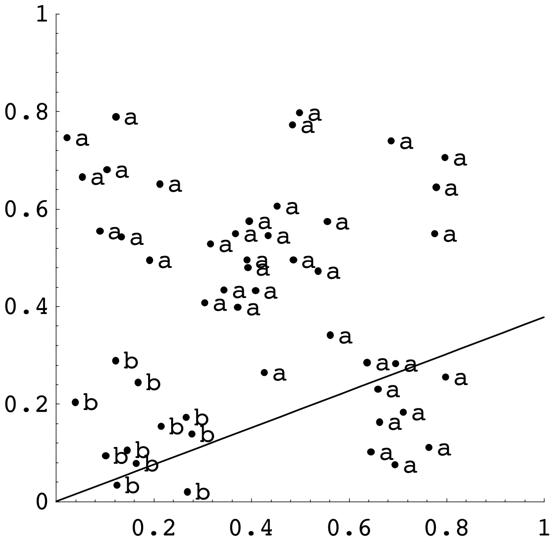
In the problem set you write a program that will run through the training pairs. Start off with a weight vector of: {-.3, -.05, 0.5}. If a point is classified correctly (e.g. as an "a" type), do nothing to the weights. If the point is actually an "a" type, but is incorrectly classified, increment the weights in some proportion (e.g. c = 0.1) of the point vector. If a "b" point is incorrectly classified, decrement the weight vector in proportion (e.g. c = - 0.1) to the values of coordinates of the training point.

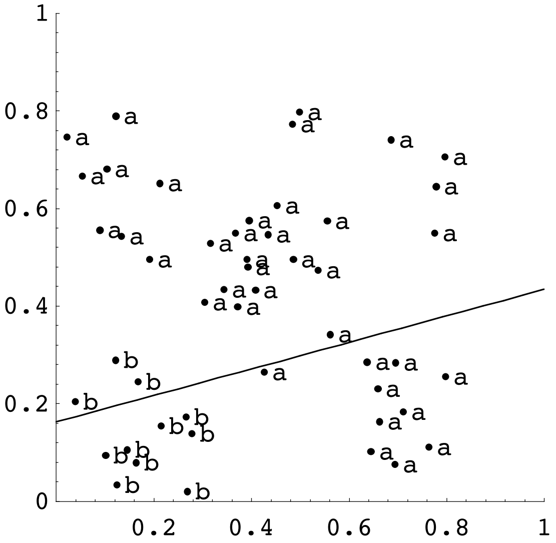
Note that you may have to iterate through the list of training pairs more than once to obtain convergence--remember convergence is guaranteed for linearly separable data sets.

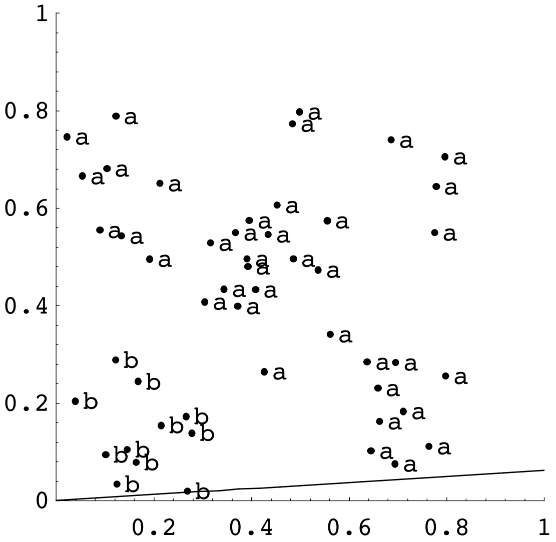
■ Plots of discriminant line

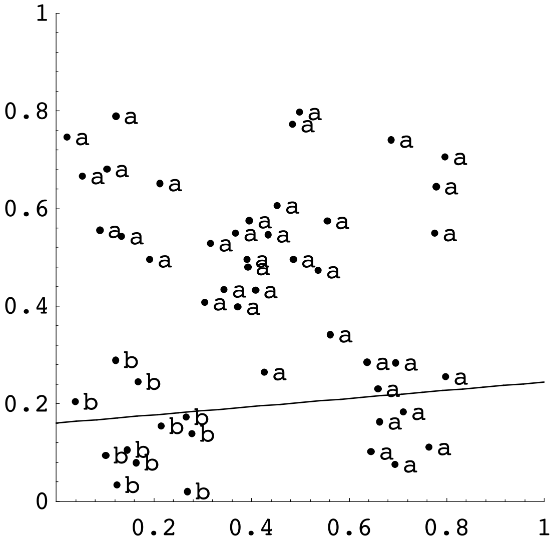
Below we show a series of plots of how the weights evolve through the learning phase. After 150 iterations, percent correct has improved, but still isn't perfect.

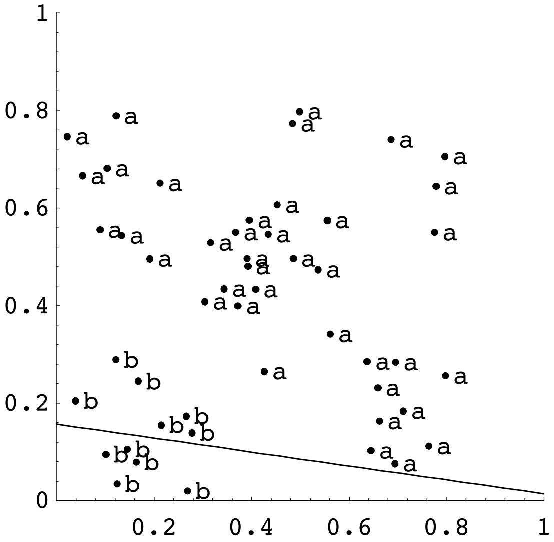


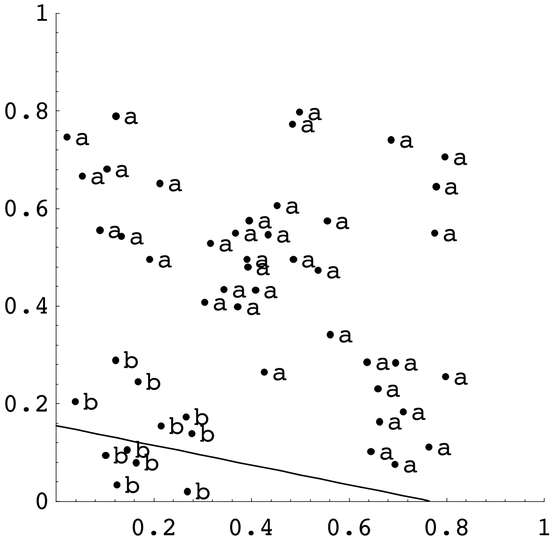


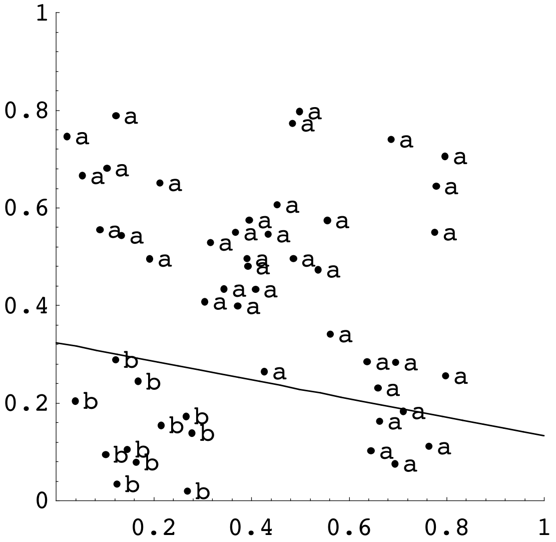

0
0.2
0.4
0.6
0.8
1
0.2
0.4
0.6
0.8
1
a
b

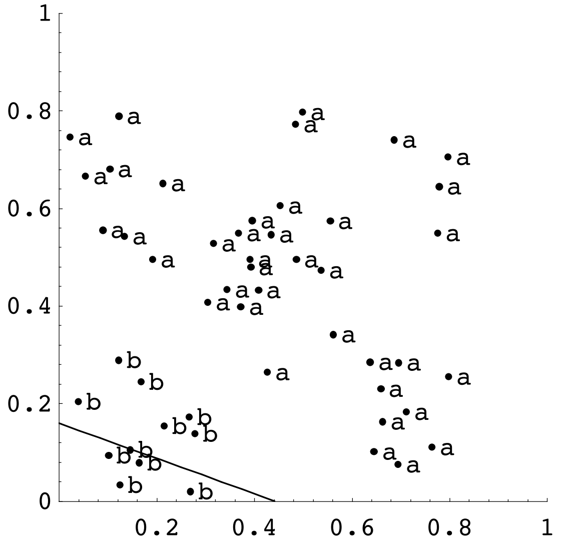

0
0.2
0.4
0.6
0.8
1
0.2
0.4
0.6
0.8
1
a
b

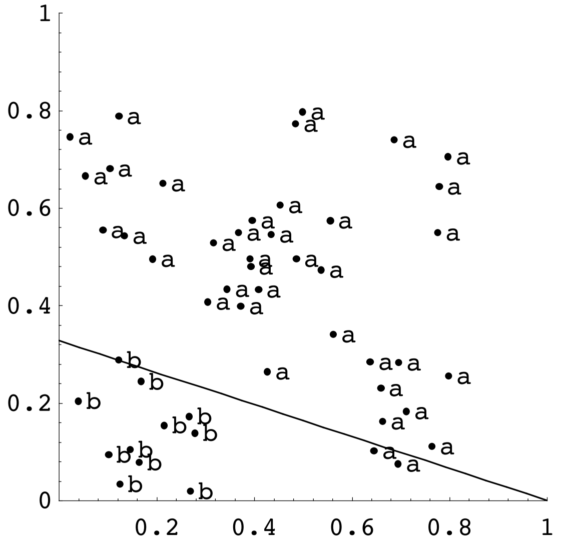

1
0.8
0.6
0.4
0.2
0
0.2
0.4
0.6
0.8
1
a
b


1
0.8
0.6
0.4
0.2
0
0.2
0.4
0.6
0.8
1
a
b


1
0.8
0.6
0.4
0.2
0
0.2
0.4
0.6
0.8
1
a
b


1
0.8
0.6
0.4
0.2
0
0.2
0.4
0.6
0.8
1
a
b





Limitations of Perceptrons (Minksy & Papert, 1969)

XOR

Inclusive vs. Exclusive OR (XOR)

Augmenting the input representation to solve XOR. A special case of polynomial mappings. The idea of augmenting inputs has seen a recent revival with new developments in Support Vector machine learning.

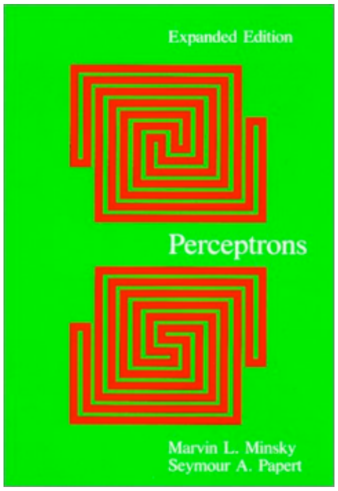
■ Natural constraints and computation of connectedness

Perceptron with natural limitations:

Order-limited: no unit sees more than some maximum number of inputs(e.g. analagous to an upper limit on the number of synapses on a dendritic tree.)

Diameter-limited: no unit sees inputs outside some maximum diameter (e.g. analagous to a neural receptive field, in which a neuron is unresponse to stimulation outside some region on the retina).

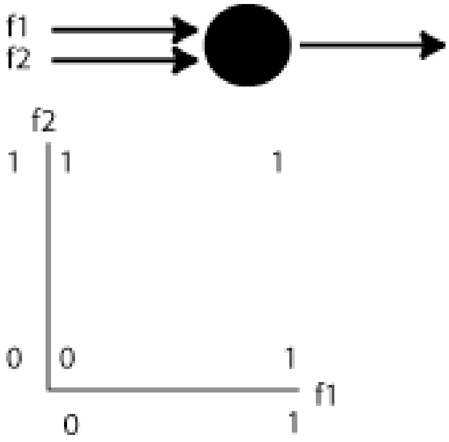
Argument : Connectedness can't be solved with diameter-limited perceptrons.



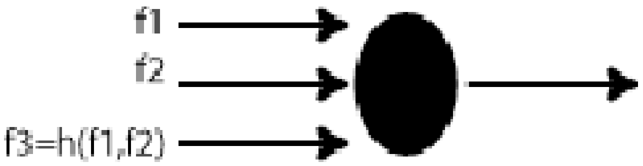
From: http://images.amazon.com/images/P/0262631113.01.LZZZZZZZ.jpg

Exercise: Expanding the input representation

The figure below shows how a 2-input TLU that computes OR maps its inputs to its outputs.



Make a similar truth table for XOR. Plot the logical outputs for the four possible input states. Can you draw a straight line to separate the 1's from the 0's?



What if you added a third input which is a function of original two inputs as in the above figure? (Hint, a logical product). Make a 3D plot of the four possible states,now including the third input as one of the axes.

Future directions for classification networks

Widrow-Hoff and error back-propagation

Later we ask whether there is a learning rule that will work for multi-layer perceptron-style networks. That will lead us (temporarily) back to an alternative method for learning the weights in a linear network. From there we can understanding a famous generalization to non-linear networks for smooth (but including steep sigmoidal non-linearities useful for discrete decisions) function mappings, called "error back-propagation".

These non-linear feedforward networks, with "back-prop" learning increase the computational power for both smooth regression and classification.

Linear discriminant analysis

We will also take a look at classification from the point of view of statistical pattern recognition. In particular, the perceptron is a special case of a linear classifier. In linear discriminants analysis, the idea is to project the data onto a hyperplane whose parameters are chosen so that the classes are maximally separated, in the sense that both the difference between the means (of the two populations) is big and the variation within the classes ("within-class scatter") is small. Intuitively, the idea is to find a decision plane that maximizes the "signal-to-noise" ratio of the classifier.

Support Vector Machines

Within recent years, there has been considerable interest in Support Vector Machine learning. This is a technique which in its simplest form provides a powerful tool for finding non-linear decision boundaries using the trick explored above of increasing the input space. SVM theory has developed a rich body of results dealing with deep issues in the generalization of learning. See the Tutorials at http://www.kernel-machines.org/tutorial.html

References

Duda, R. O., & Hart, P. E. (1973). *Pattern classification and scene analysis*. New York.: John Wiley & Sons.

Grossberg, S. (1982). Why do cells compete? Some examples from visual perception., UMAP Module 484 Applications of algebra and ordinary differential equations to living systems, : Birkhauser Boston Inc., 380 Green Street Cambridge, MA 02139. (pdf1, pdf2)

Heeger, D. J., Simoncelli, E. P., & Movshon, J. A. (1996). Computational models of cortical visual processing. Proc Natl Acad Sci U S A, 93(2), 623-7. (pdf) (See too: Carandini et al).

Minsky, M., & Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA: MIT Press.

Poggio, T. (1975). On optimal nonlinear associative recall. Biological Cybernetics, 19, 201-209.

Vapnik, V. N. (1995). The nature of statistical learning. New York: Springer-Verlag.

© 1998, 2001, 2003, 2005 Daniel Kersten, Computational Vision Lab, Department of Psychology, University of Minnesota.