Introduction to Neural Networks

Non-linear models
The perceptron

Initialization

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

Introduction

Last time

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 explored 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.

We looked at linear recall models which used memories stored with a Hebbian learning rule.

Simulation examples

Hebbian learning with linear recall:

Heterassociation -- supervised learning.
Autoassociation -- unsupervised learning

Superposition led to interference in the linear recall model
->While averaging and interpolation is useful for some tasks,
we also need solutions that will make unambiguous decisions.
-> Classification shows need for non-linear feedforward/recall models

Today

Brief intro to non-linear models

Classification vs. regression

Perceptron as a classic example. Limitations of simple perceptron

Introduction to non-linear models

By definition, linear models have strong 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. Here are a few.
Consider a single unit with output y, and inputs f_i.

1. **Polynomial mappings**. 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:

$$y = \sum w_i f_i$$

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

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

More terms allow mapping of more complicated functions. One can think of the terms $b_n = f_{i_1} f_{i_2} \ldots f_{i_n}$ as basis functions. While giving flexibility, in general the choice of basis functions affects the efficiency with which a function class can be approximated. E.g. contrast Taylor series with Fourier series. Also we need to keep in mind that the more parameters $w_{i_1 \ldots i_n}$ to learn the bigger risk of over-fitting.

2. **Normalization**. Another type of linearity to divide a linear or semi-linear neural output by the squared responses of neighboring units. This is called *divisive normalization*. This is a steady-state model of a version of shunting inhibition which has been very successful in accounting for a range of neurophysiological receptive field properties in vision (Heeger et al., 1996). The idea of normalization has been proposed to underly basic processing functions of the brain, including attention (Carandini & Heeger, 2011) and multi-sensory integration (Ohshiro et al., 2011).

Compare:

$$y_i = f_j - \sum_{j \neq i} w_{ij} f_j$$

with:

$$y_i = \frac{f_i^n}{\sum_{j \neq i} w_{ij} {f_j}^n}$$

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

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

What is the steady-state solution of $\frac{dy_i}{dt} = -\alpha y_i + (\beta - y_i) f_i - y_i \sum_{i \neq j} w_{ij} f_j$ **?**

---

3. **Point-wise non-linearity**. One of the simplest things we can do is to use the generic connectionist neuron with its second stage point-wise non-linearity which we introduced near the beginning of the course--i.e. inner product followed by a non-linear sigmoid. Once a non-linearity such as a sigmoid introduced, it makes sense to add more additional layers of neurons.

Use matrix algebra to show that any linear network that feeds into another linear network is equivalent to a single linear network, with just one layer of weights.

Much of the modeling of human visual pattern discrimination has used just these “rules-of-the-game” (linear matrix multiplication followed by point non-linearities), with additional complexities (such as divisive normalization) added as needed. And together with an appropriate learning rule (such as error back-prop that we will study later) provide state of the art machine vision solutions to some limited problems, such as written character recognition (see LeNet demo link in syllabus).

As mentioned above, a central challenge, in the above and all methods that 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, which can result in failures to generalize). We’ll talk more about this problem later.

The above modifications produce smooth functions. If we want to classify rather than do smooth regression, we need a more abrupt process that forces outputs to extremes, such as 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. We will look at the Perceptron, an early example of a network built on such threshold logic units.

Classification and the Perceptron

Classification

One can make a 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. The linear heterassociative model we looked at last time was doing regression; however, it was motivated by what one could do using a Hebbian learning rule. A statistician might approach the same problem but from the point of view of a statistical constraint such as mimizing the sum of squared differences. We discuss this connection more later.

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 or types,

{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 (e.g. bold or italics) or size. In other words, we require a mapping ϕ = T such that:

T: **f** -> {"a","b"} = {0,1}, or we can use any other set of convenient labels, such as {-1,1}

In general, a classifier should show *invariance* over variations of instances within each class, while at the same time showing *selectivity* between classes--it should minimize the proportion of misclassification errors.

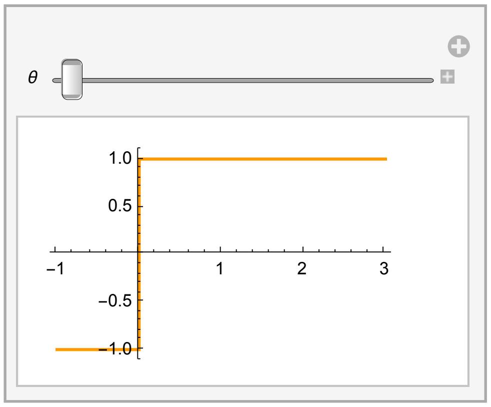
Let's include a step threshold function in the tradition of McCulloch & Pitts, a special case of the generic connectionist neuron model. With the step threshold, recall that the units are called **Threshold Logic Units** (TLU):

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

```
In[3]:= step1[x_, θ_] := If[x<θ,-1,1];
```

```
In[4]:= Manipulate[Plot[step1[x, θ], {x, -1, 3},
          PlotStyle → {Hue[0.1]}, ImageSize → Small], {{θ, 0}, 0, 3}]
```

Out[4]=



The TLU neuron's output can be modeled simply as:

step[w.f, *θ*].

Successful classification depends on having the right values of the weights **w**, and the threshold ***θ***.

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

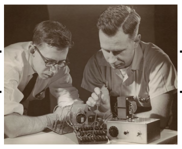
Weights and threshold: Trick to simplify the math

The TLU has two classes of parameters to learn, weight parameters and a threshold *θ*. If we 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, 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*:

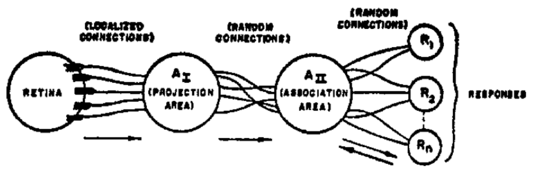
```
In[5]:= (* This point is almost obvious just by thinking, so don't get confused by the Mathematica
        (* It is here to again practice some computer algebra*)

        Remove[w,f,waug,faug,θ,w1,w2,f1,f2];

        w = {w1,w2};
        f = {f1,f2};
        waug = {w1,w2,-θ}; (*define the third, augmented synapse to have weight  of -θ *)
        faug = {f1,f2,1}; (*define the third, augmented input to always be on *)

        (*Now compare the 2-input threshold constraint with the 3-input one*)
        θ==w.f
        Solve[θ==w.f,θ]  (*The solution for θ as threshold*)
        Solve[0==waug.faug,θ](*is the same as the solution for zero as threshold with the "weight"

Out[16]= θ == f1 w1 + f2 w2

Out[17]= {{θ → f1 w1 + f2 w2}}

Out[18]= {{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)

```
Image[
```

```
];
```

In[20]:=



Out[20]=

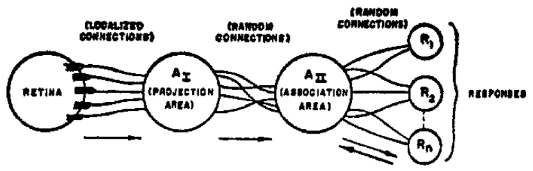


Figure from: Rosenblatt, Frank. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. In, Psychological Review, Vol. 65, No. 6, pp. 386-408, November, 1958.

The original perceptron models were fairly complex. 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 or not?)

With some adjustments, the Perceptron bears some resemblance with 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 it does have, are not a prominent computational 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 complexity are difficult to analyze. It is hard to draw general theoretical conclusions about what they can compute and what they can learn. For example, the computational function of the feedback from cortex to thalamus is still a major mystery in neuroscience.

In order to make the Perceptron theoretically tractable, we will first take a look at an extremely 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 answer this we'll simplify further and look at a single TLU with just two variable inputs, but three adjustable weights.

Visualizing recall behavior and linear separability

A two-input simplified perceptron

Assume we have some generative process that is providing data {**input**$_k$}, where each **input**$_k$ is a two-dimensional vector, and k indexes the data samples k=1,...,N. So the data set can be represented in a 2-D scatter plot with **input** = {f1, f2}. But any particular input can belong to one of two categories, say -1 or 1: **inputs**$_k$ -> {-1,1}.
(-1 and 1 could correspond, for example, to classes with labels "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[21]:= waug = {w1,w2,w3}; (*w3 = -θ*)
         faug = {f1,f2,1};
         Solve[waug.faug==0,{f2}]
```

Out[23]= $\left\{\left\{\mathrm{f2} \to \frac{-\mathrm{f1}\,\mathrm{w1} - \mathrm{w3}}{\mathrm{w2}}\right\}\right\}$

We can see that f2 is a linear function of f1 for fixed weight values w1 and w2.

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

Let's write a function for a particular decision line with arbitrary choices for the weights and threshold. We'll use the weight and threshold choices to artificially classify some inputs, and then use the decision-line function to illustrate the data in a plot below.

```
In[24]:= w1 = 0.5; w2 = 0.8; θ = 0.55;
         waug = {w1,w2,-θ};
         decisionline[f1_,θ_]:= -((-θ + f1*w1)/w2)

In[27]:= decisionline[f1, θ]

Out[27]= -1.25 (-0.55 + 0.5 f1)
```

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 can re-define step[ ] to have a fixed threshold of zero:

```
In[28]:= Remove[step];
         step[x_] := If[x > 0, 1, -1];

In[30]:= inputs = Table[{x = RandomReal[], y = RandomReal[], 1}, {i, 1, 20}];
         responses = step[waug.#] & /@ inputs;
```

An element in the **responses** list is 1 if the weighted sum of the inputs exceeds zero, and is -1 if it is less (or equal to) zero.

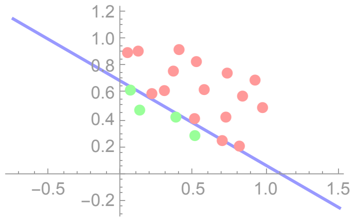
View the input data and the responses

Let's plot up the classified data and decision line.

```
In[32]:= inputsXY = Drop[inputs, None, {3}];
```

```
In[33]:= abovethreshold = Delete[inputsXY, Position[responses, -1]];
         belowthreshold = Delete[inputsXY, Position[responses, 1]];

         g1 = Plot[decisionline[f1, θ], {f1, -.75, 1.5}, PlotStyle → {RGBColor[0, 0, 1]}];
         g2 = ListPlot[{abovethreshold, belowthreshold}, PlotStyle → {Red, Green},
            PlotMarkers → {{"●", Small}, {"●", Small}}, Joined → False];

         (*So inputs classified as above threshold are red,and those below are green.*)

         Show[g1, g2, ImageSize → Small]
```

Out[37]=



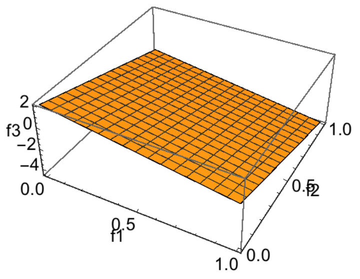
The blue line separates inputs whose inner product with the weights exceeds the threshold from those that do not exceed it.

Side note: simplified perceptron (TLU network) with N-dimensional inputs

For a three dimensional input TLU, this decision surface is a plane. Here is an arbitrary example:

```
In[38]:= 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[41]= {{f3 → 5. (0.4 - 0.5 f1 - 0.8 f2)}}

In[42]:= Plot3D[-5. (-0.4 + 0.5 f1 + 0.8 f2), {f1, 0, 1},
          {f2, 0, 1}, ImageSize → Small, AxesLabel → {f1, f2, f3}]
```

Out[42]=



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

Above we’ve visualized how a perceptron classification rule divides inputs into points that are on one side of a line (or hyperplane) from points on the others. Remember you can think of these points as vectors whose elements represent a discrete set of features characterizing a set of patterns.

Now suppose we have the inputs and their classification assignments, but don’t know the parameters of the mapping function.

In other words, given a set of classified data (i.e. a “teacher” has supplied the correct answers), how can we find the perceptron parameters (weights) that specify a good decision plane? I.e. if we didn't know what the weights were that generated the above data, how could we find a suitable values from the data itself?

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

Suppose we have a training set consisting of N pairs $\{\boldsymbol{f_i}, \boldsymbol{g_i}\}_{i=1 \text{ to } N}$ where $\mathbf{g}_\mathrm{i}$ is either -1 or +1. Imagine we are in the middle of the training, and we have a set of weights **w**. A new training pair $\{\mathbf{f}_\mathrm{i}, \mathbf{g}_\mathrm{i}\}$ comes along, so we can check to see how our perceptron is doing. Suppose it predicts that the output for $\mathbf{f}_\mathrm{i}$ should be $\hat{g}_\mathrm{i}$. Then $\hat{g}_\mathrm{i}$==$\mathbf{g}_\mathrm{i}$ 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.

The question is how to change the weights?

Let’s use our simple 2-D input model:

```
Remove[w, f, c, w1, w2, θ];
```

```
w = {w1, w2, -θ}; f = {f1, f2, 1};
```

We have two basic conditions to address.

If the classification is correct...

...don’t change the weights

So let **nextW** be the new set of weights:

```
In[45]:= nextW = w; (*No change*)
```

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[46]:= 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[47]:= Simplify[nextw.f - w.f]
```

Out[47]= $c\,(1 + f1^2 + f2^2)$

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 an incorrect decision next time 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[48]:= nextw = w - c f;
         Simplify[nextw.f - w.f]
```

Out[49]= $-c\,(1 + f1^2 + f2^2)$

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

Let's look at a program that uses a Perceptron style threshold logic unit (TLU) that learns to classify two-dimensional vectors into "a" or "b" types.

1. 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 indicated by "+1",

x^2+y^2 > **bigradius**^2 and for the "b" points indicated by "-1",

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

Depending on the radius values, these patterns may or may not be linearly separable because they fall inside or outside their respective circles. The data are stored in **stuff**.

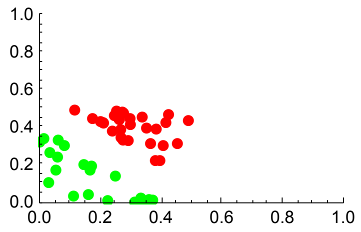
To be concrete, for the "a" type points, let x+y>0.6 and for the "b" points, x+y<0.4. Each pair of points should have its corresponding label, a or b.

```
In[50]:= inputs = {}; size = 50;
         outputs = {};
         While[Length[inputs] < size,
           Which[(x = 0.5 RandomReal[]) + (y = 0.5 RandomReal[]) < 0.4,
            inputs = Append[inputs, {x, y, 1}];
            outputs = Append[outputs, -1], x + y > 0.6, inputs = Append[inputs, {x, y, 1}];
            outputs = Append[outputs, 1]
           ]];
```

For plotting purposes, drop the fixed input (corresponding to the -θ threshold term), and pull out the x-y coordinates above and below threshold.

```
In[53]:= inputsXY = Drop[inputs, None, {3}];
         abovethreshold = Delete[inputsXY, Position[outputs, -1]];
         belowthreshold = Delete[inputsXY, Position[outputs, 1]];
         g3 = ListPlot[{abovethreshold, belowthreshold},
           PlotStyle → {Red, Green}, PlotMarkers → {{"●", Small}, {"●", Small}},
           ImageSize → Small, Joined → False, PlotRange → {{0, 1}, {0, 1}}]
```

Out[55]=



2. Define threshold function

Define a function that is -1 for x <0 and +1 for x>= 0. Here we use the conditional **If[]**.

```
In[56]:= threshold[x_] := If[x<0,-1,1];
```

3. Perceptron learning algorithm

Let's 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), we do nothing to the weights. If the point is actually an "a" type, but is incorrectly classified, we 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.

Let's make a list to record the changes in the weight vector. Note that the three components of the weight vector determine the intercept and the slope of the line separating the "a" and "b" points.

Program

```
In[57]:= w0 = {-.3, -0.05, 0.5};
         w = w0;
         wlist = {};
```

You will probably have to iterate through the list of training pairs several times to obtain convergence--remember convergence is guaranteed for linearly separable data sets. So execute the following cell. Then check the plots of the discriminant line below. Repeat executing this cell until **w** is no longer changing.

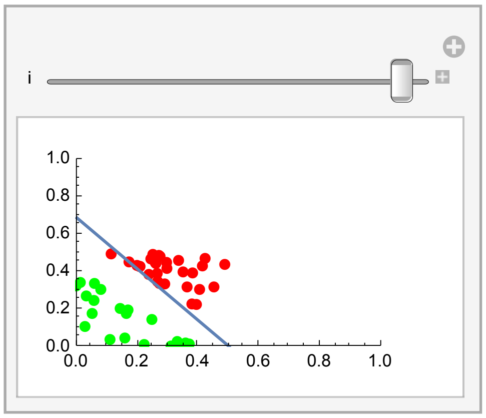
```
In[191]:= i = 0; c = 0.01;

          While[++i<=Length[inputs],
          Which[(outputs[[i]] == 1) && (threshold[w.inputs[[i]]] == 1), w=w,
               (outputs[[i]] == 1) && (threshold[w.inputs[[i]]] == -1), w = w + c (inputs[[i]]),
               (outputs[[i]] == -1) && (threshold[w.inputs[[i]]] == -1), w=w,
               (outputs[[i]] == -1) && (threshold[w.inputs[[i]]] == 1), w = w - c (inputs[[i]])
               ];
          wlist = Append[wlist,w];
          ];
          w
Out[194]= {0.0584899, 0.079961, -0.04}
```

4. Plots of discriminant line

Make a series of plots showing how the weights evolve through the learning phase.
w={a,b,c};

```
In[195]:= Remove[a, b, c2];
          Manipulate[a = wlist⟦i, 1⟧;
           b = wlist⟦i, 2⟧;
           c2 = wlist⟦i, 3⟧;
           Show[g3, Plot[-b x/a - c2/a, {x, 0, 1}, PlotRange → {{0, 2}, {0, 2}},
             AspectRatio → 1, ImageSize → Small]], {i, 1, Length[wlist], 1}]
```

Out[196]=



5. Table of classifications

Make a list {{c1,out1},c2,out2},,,,} where ci is the correct classification, and outi is the output of your threshold logic unit with weight w or w0.

```
In[197]:= responses=threshold[w.#] & /@ inputs;
          Table[{outputs[[i]],responses[[i]]},
                {i,1,Length[inputs]}]//Transpose
Out[198]= {{-1, 1, -1, -1, 1, -1, 1, 1, -1, -1, 1, 1, -1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, -1, 1,
            1, -1, -1, 1, -1, 1, -1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, 1, -1, 1, -1, 1, -1},
           {-1, 1, -1, -1, 1, -1, 1, 1, -1, -1, 1, 1, -1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, -1,
            1, 1, -1, -1, 1, -1, 1, -1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, 1, -1, 1, -1, 1, -1}}
```

6. Proportion of correction classifications - before and after

Let's calculate the proportion of correct classifications before and after the weight vector has been trained.

Before

```
In[199]:= Pc = 0;i=0;
responses0=threshold[w0.#] & /@ inputs;
While[++i <= Length[inputs],
 Which[(outputs[[i]] == 1) && (responses0[[i]] == 1), Pc++,
    (outputs[[i]] == 1) && (responses0[[i]] == -1),Null,
    (outputs[[i]] == -1) && (responses0[[i]] == -1), Pc++,
    (outputs[[i]] == -1) && (responses0[[i]] == 1),Null
    ];
];
Pc/Length[inputs]
```

Out[202]= $\frac{29}{50}$

After:

```
In[203]:= Pc = 0;i=0;
responses=threshold[w.#] & /@ inputs;

While[++i <= Length[inputs],
 Which[(outputs[[i]] == 1) &&
            (responses[[i]] == 1), Pc++,
    (outputs[[i]] == 1) &&
            (responses[[i]] == -1), Null,
    (outputs[[i]] == -1) &&
            (responses[[i]] == -1), Pc++,
    (outputs[[i]] == -1) &&
            (responses[[i]] == 1),Null
    ];
];
Pc/Length[inputs]
```

```
Out[207]= 1
```

Limitations of Perceptrons (Minksy & Papert, 1969)

The above classification algorithm will only achieve perfect solutions for linearly separable data--i.e. in an n-dimensional space, data that can be separated by a hyperplane. Data can fail to be separable for two reasons: 1) the source of the data is linearly separable, but the measured data is corrupted by noise; 2) the source of the data is inherently not linearly separable.

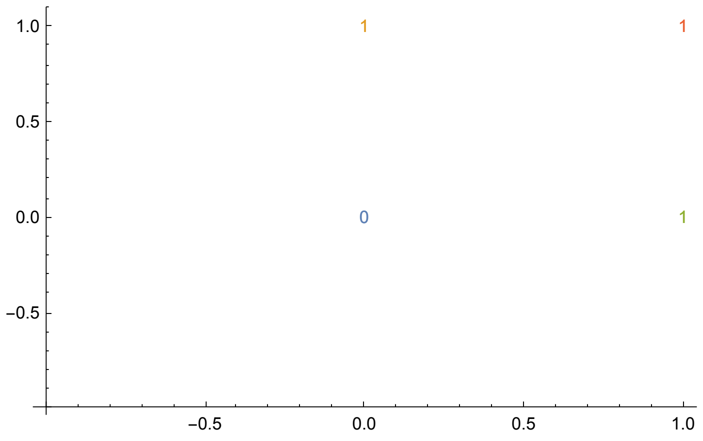
Classic problem of non-linear separability: XOR

Inclusive vs. Exclusive OR (XOR)

Imagine a two feature input space consisting of {{0,0},{0,1},{1,0},{1,1}}. These 2D vectors get mapped to output classes: {0,1,1,1}. You should be able recognize this mapping as **inclusive OR**, which we can represent with the following plot:

```
In[208]:= ListPlot[{{{0, 0}}, {{0, 1}}, {{1, 0}}, {{1, 1}}},
           AxesOrigin → {-1, -1}, PlotMarkers → {{"0"}, {"1"}, "1", "1"}]
```

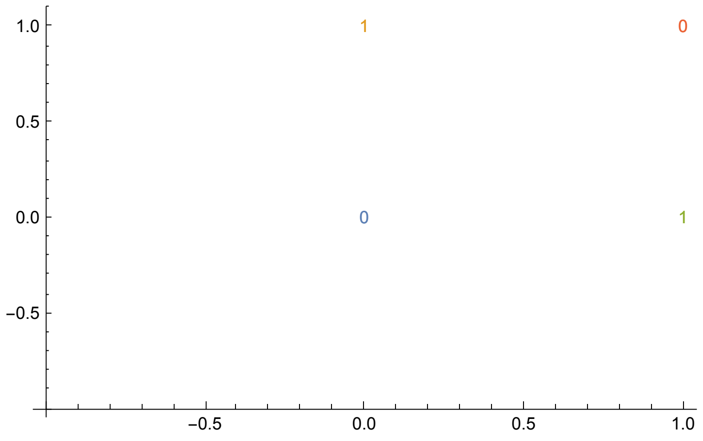
Out[208]=



Compare the OR plot with an **exclusive or** (XOR) which only gives a “true” or “1” output if either of the inputs is “1”, but not both.

```
In[78]:= ListPlot[{{{0, 0}}, {{0, 1}}, {{1, 0}}, {{1, 1}}},
          AxesOrigin → {-1, -1}, PlotMarkers → {{"0"}, {"1"}, "1", "0"}]
```

Out[78]=

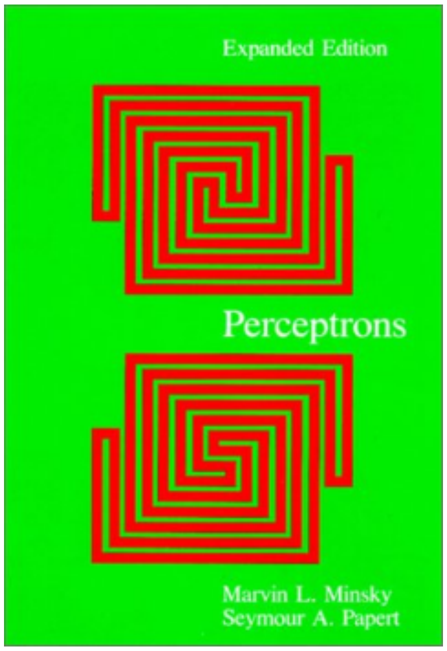


The 0’s and 1’s can no longer be separated by a straight line.

There is an exercise below on one solution to solving XOR by augmenting the input representation with another dimension. A special case of polynomial mappings. The idea of augmenting inputs, i.e. mapping the data into a higher-dimensional space, reappears in Support Vector machine learning.

Natural constraints and computation of connectedness

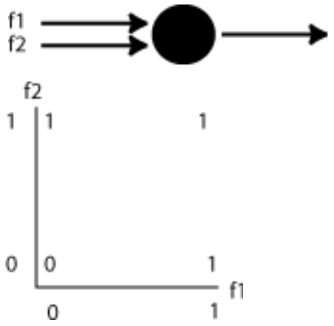
Perceptron with natural limitations:
Order-limited: no unit sees more than some maximum number of inputs (e.g. analogous to an upper limit on the number of synapses on a dendritic tree.)
Diameter-limited: no unit sees inputs outside some maximum diameter (e.g. analogous to a neural receptive field, in which a neuron is unresponsive 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

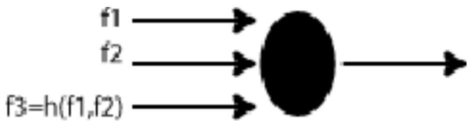
Exercise: Expanding the input representation

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 figure below? (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 then 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. (Note the terminology--linear here refers to the decision boundary, not the network.). In linear discriminant 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

Support Vector Machine learning has provided tools 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.support-vector.net/tutorial.html.

Appendix

Sidenote: More on conditionals

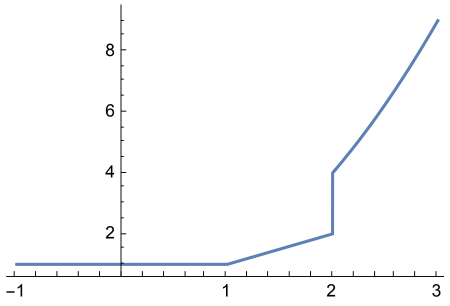
You have seen how to generate threshold functions using rules and the **Piecewise** function. But you can also use conditional statements. For example the following function returns x when Sin[2 Pi x] < 0.5, and returns -1 otherwise:

```
In[79]:= 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[80]:= tep[x_] := Which[-1<=x<1,1,
                         1<=x<2, x,
                         2<=x<=3, x^2]
```

In[81]:= **Plot[tep[x], {x, -1, 3}]**

Out[81]=


References

Carandini, M., & Heeger, D. J. (2011). Normalization as a canonical neural computation. Nature Reviews Neuroscience. doi:10.1038/nrn3136
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)
Gütig, R., & Sompolinsky, H. (2006). The tempotron: a neuron that learns spike timing–based decisions. Nature Neuroscience, 9(3), 420–428. doi:10.1038/nn1643
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.
Ohshiro, T., Angelaki, D. E., & DeAngelis, G. C. (2011). A normalization model of multisensory integration. Nature Publishing Group, 14(6), 775–782. doi:10.1038/nn.2815
Poggio, T. (1975). On optimal nonlinear associative recall. Biological Cybernetics, 19, 201-209.
Rosenblatt, F. (1962) Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms, Spartan Books.
Vapnik, V. N. (1995). The nature of statistical learning. New York: Springer-Verlag.