Introduction to Neural Networks
U. Minn. Psy 5038

Multi-layer non-linear networks
Gradient descent: Learning by error back-propagation

Introduction

Last time

Regression and learning in linear neural networks. Last time we showed 4 different ways to find the generating parameters {2,3} for the following data:

```
In[56]:= rsurface[a_, b_] :=
           N[Table[{x1 = 1 RandomReal[], x2 = 1 RandomReal[],
              a x1 + b x2 + 0.5 RandomReal[] - 0.25}, {120}], 2];
         data = rsurface[2, 3];
         Outdata = data[[All, 3]];
         Indata = data[[All, 1 ;; 2]];
```

Linear regression is so common that *Mathematica* 6 has added the following function to find the least squares parameters directly:

```
In[60]:= LeastSquares[Indata, Outdata]

Out[60]= {2.00148, 2.99808}
```

■ Gradient

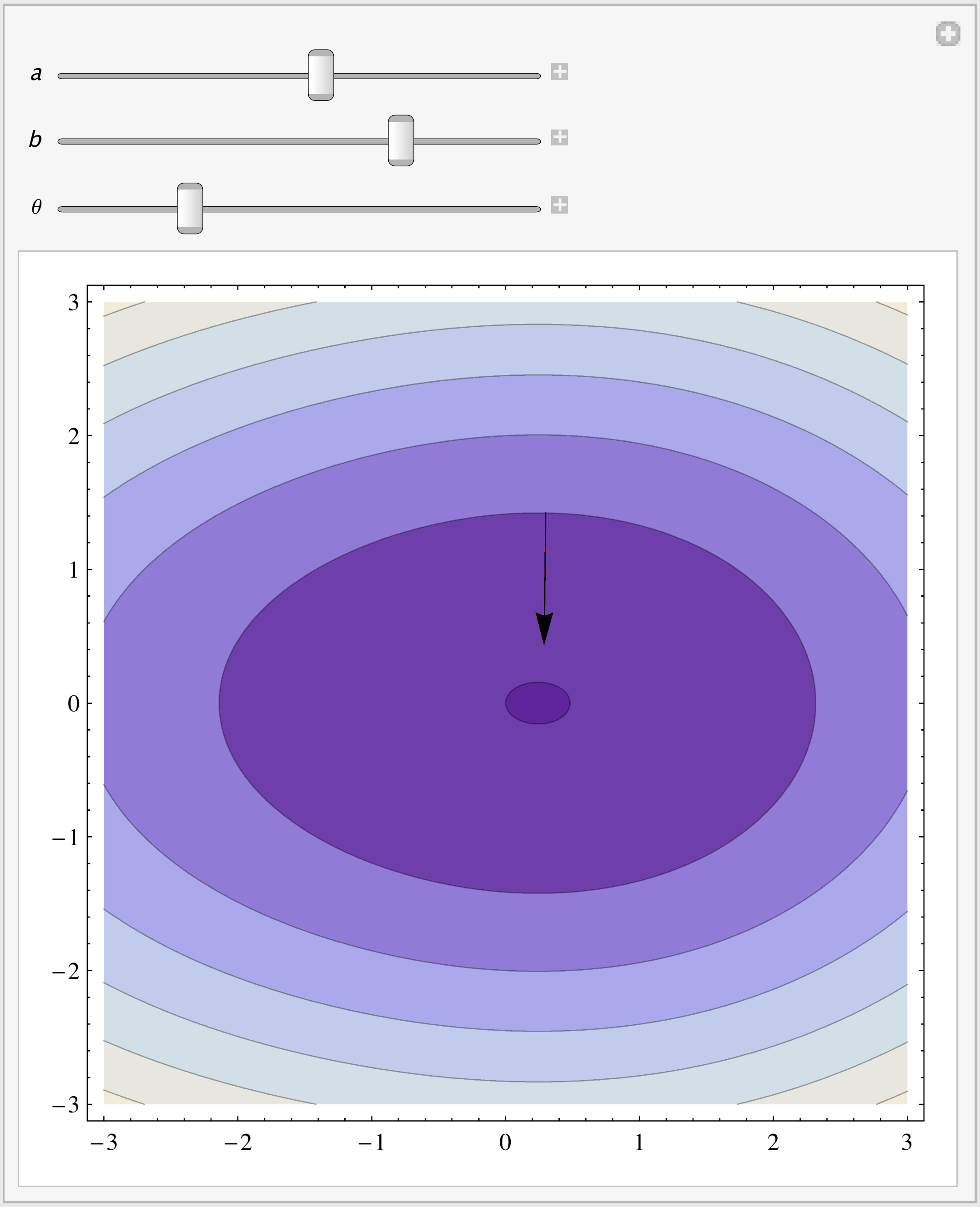
The function f(x,y) is plotted as a contour plot. The demo below calculates and plots the vector **gv** = **∇f**. After normalized to unit length, it is plotted as an arrow in red. The red arrow should always point down the "hill" in the contour plot. You can manipulate the position of the point at which the gradient gets evaluated with the a and b sliders.

The *θ* slider controls the direction of a second vector **un** (in black) whose length is determined by the projection: **∇f.un**

In[61]:=

```
f[x_, y_] := 2 x^2 + 5 y^2 - Sin[x];
g1 = ContourPlot[f[x, y], {x, -3, 3}, {y, -3, 3}, PlotPoints → 50];
Manipulate[
 gv = N[D[2 x^2 + 6 y^2 - Sin[x], {{x, y}}] /. {x → a, y → b}];
 gn = -Normalize[gv];
 un = {Cos[θ], Sin[θ]};
 gm = (un.gn) un;
 g2 = Graphics[{Red, Arrow[{{a, b}, gn + {a, b}}]}];
 g3 = Graphics[Arrow[{{a, b}, gm + {a, b}}]];
 Show[{g1, g2, g3}], {{a, 1}, -3, 3}, {{b, 2}, -3, 3}, {θ, 0, 2 * Pi}]
```

Out[63]=



Today

Learning with the squashing function back in.

Multi-layer networks

Introduction to multi-layer nets: Computation and recall

For linear networks, no computational power is gained by having extra layers:

y1 := W0.y0;
y2 := W1.y1; (1)

is equivalent to:

y2 := W1.(W0.y0) := W1.W0.y0 := W3.y0; (2)

where **W3** is just another matrix. However, if the inner product is followed by a non-linear transformation, then concatenating layers of neural elements is no longer trivial:

y2 := f[**W1.**f[**(W0.y0)**]]; (3)

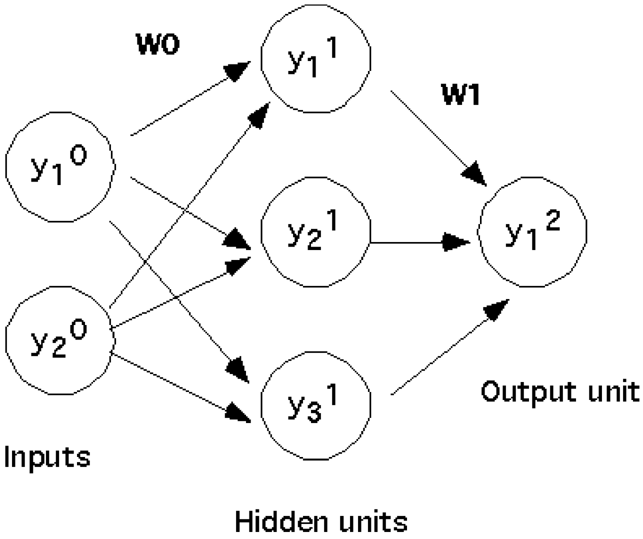
where **f[]**, for example, is a sigmoid.

In[64]:=

```
f[x_] := 1 / (1 + Exp[- (x - .5)]);
```

NOTE: Another notation alert--in this notebook we use f or f2 to represent a scalar squashing function, not an input vector.

■ Example: 2 input units, 3 hidden units, 1 output unit



In[85]:=

```
Clear[y2, f2];
f2[x_] := N[1 / (1 + Exp[- (x - .5) * 100])];
W1 = {{-13.2328, 6.06398, 6.04958}};
W0 = {{-0.937564, -1.09841}, {-9.95589, 3.85642},
   {3.79034, -10.1737}};
y2[y0_] :=  Chop[f2[W1.f2[(W0.y0)]]];
```

What logical function does this network compute?

```
In[91]:=  y2[{1, 0}]

Out[91]=  {1.}
```

What logical function will the above network compute with the following weights?

```
In[75]:=  W0 = {{1.9009195689645, 1.9251997561975753}, {2.177312432892121, 2.19839207
            {-2.8297064197335953, -2.8648165030618693}};
            W1 = {{1.1241018372719358, 1.7600034188676417, -10.048562036295957}};
```

```
In[77]:=  Clear[y2];
          y2[y0_] := Chop[f2[W1.f2[(W0.y0)]]];
```

```
In[83]:=  y2[{1, 0}]

Out[83]=  {1.}
```

■ Logistic function - a smooth, differentiable non-linearity

As we will see shortly, it can be important for theoretical reasons, to have a non-linearity which is smooth enough to be differentiable. **D[]** returns an expression for the derivative, so to define a function that is the derivatie of another we write:

```
In[92]:=  Df1[x_] := D[f[t],t] /.t->x
          Df1[x_] := Evaluate[D[f[t],t]]
```

The more direct way is to use "operators" or functionals that take functions as inputs and return functions as outputs. **Derivative[]**, or **f'[x]** return functions:

```
In[94]:=  (*Df[x_] := Derivative[f[x],x]*)
          Df[x_] := f'[x]
```

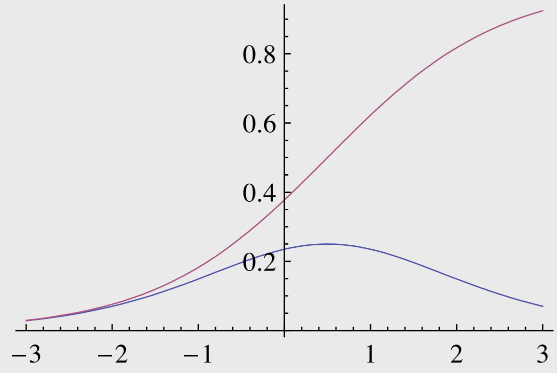
Note that the derivative has a particularly simple expression in terms of f[x], which you can verify by comparing Simplify[Df[x]] with Simplify[hh[x]], or Simplify[Df[x]-hh[x]]:

```
In[98]:=  hh[x_] := f[x](1-f[x]);
```

Here is a plot of the sigmoid and its derivative:

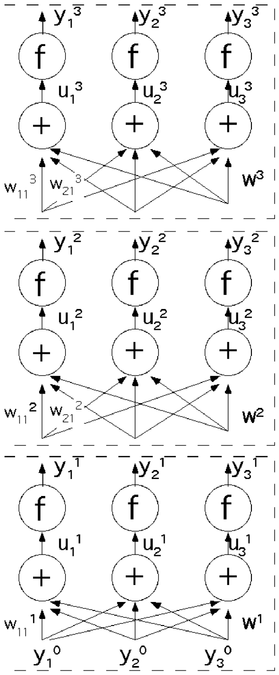
```
In[97]:=  Plot[{Df[x], f[x]}, {x, -3, 3}]
```

Out[97]=



■ Generic multi-layer feedforward net

OK, so suppose we have a multi-layer network with 3 layers of weights. The output from the first layer is: $\mathbf{y}^{\mathbf{1}}$ = f[$\mathbf{u}^{\mathbf{1}}$] = f[$\mathbf{W}^{\mathbf{1}}.\mathbf{y}^{\mathbf{0}}$]. The output of the second layer is: $\mathbf{y}^{\mathbf{2}}$ = f[$\mathbf{u}^{\mathbf{2}}$] = f[$\mathbf{W}^{\mathbf{2}}.\mathbf{y}^{\mathbf{1}}$]. And so forth.



Multi-layer nets: Learning and the error back-propagation algorithm

So we've seen that the non-linear generic neuron allows increased computational power when we add an extra layer of weights. But suppose we don't know the weights, but want to determine them through supervised learning?

How to assign the weights? For any complex system that is required to achieve a target goal, for the system to work, each component must contribute towards the goal. If the goal is not met, one has to figure out which component needs to be fixed. If the goal is met, each component contributed something towards the goal. How does one assign the credit for success or failure to a component? This problem is called the *credit-assignment problem*..

In particular, for the above multi-layer network, how do we adjust the weights in a way appropriate for learning a given input/output relation?

■ Gradient descent again, but this time the network is non-linear.

$$\{x^p, t^p\},\ \ \mathrm{y}^0 = x^p,\ \ \text{training pairs p} = 1,\ldots,\mathrm{M}$$

For a given input $\{y^0 = x^p\}$, we feed forward the information to the last layer (layer L) to produce an output $\{\mathbf{y}=\boldsymbol{y}^L\}$. We compare the output to the target value supplied by the "teacher" $\{\mathrm{t} = t^p\}$, and compute the error:

$$\mathrm{E}(\{\mathrm{w}_{ij}^{\lambda}\}) = \sum_{k=1}^{N} (\mathrm{y}_k(\mathrm{y}^0; \mathrm{w}_{ij}^{\lambda}) - \mathrm{t}_k)^2$$

where we've summed over all N output units. (A subscript means the component of the corresponding vector of activity.) λ indexes the weight layers going from λ=1 to L. Note that the y's at any point after the input depend on the h's (the weigthed sum before the non-linearity), each of which in turn depends on w_{ij}^{λ}s before it.

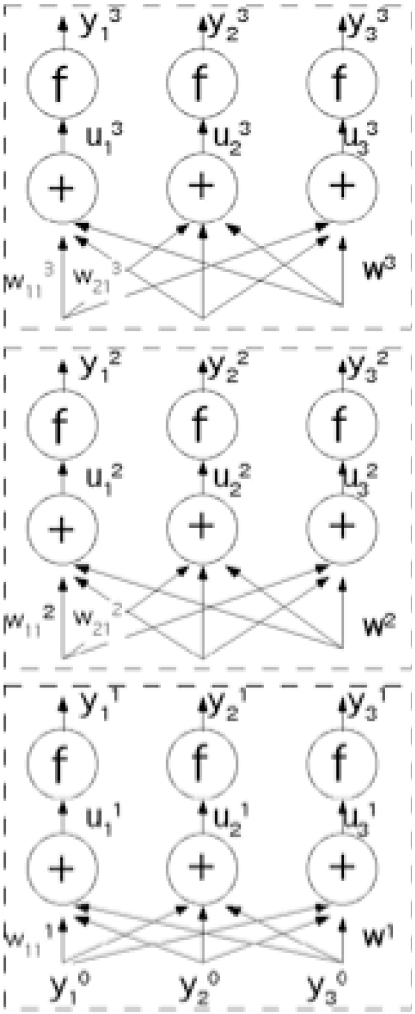
The trick is to find out how to assign credit (and blame) for the error to each of the weights. Gradient descent provides the answer. Adjust the weights such that:

$$\text{new } \mathbf{w}_{ij}^{\lambda} = \text{previous } \mathbf{w}_{ij}^{\lambda} + \Delta\mathbf{w}_{ij}^{\lambda} \tag{4}$$

where

$$\Delta\mathrm{w}_{ij}^{\lambda} = -\eta \frac{\partial \mathrm{E}}{\partial \mathrm{w}_{ij}^{\lambda}}$$

Again, this formula means that if we calculate the gradient $\nabla E=\partial_{w^{\lambda}_{ij}}\ E$, its negative direction points in the direction of steepest descent. Thus if we update the weight vector to point in this direction, then at the next step we should in general have a lower value of E. I.e. E(new w^{λ}_{ij}) < E(previous w^{λ}_{ij}).



Let's see how to calculate the gradient for one layer. Then we will generalize the result to an expression that tells us how to adjust the ijth weight for the connection in the λth layer.

In[27]:=

```
Remove["Global`*"]
```

In[102]:= $\mathtt{(1\,/\,2)\,*\,\left(t_j - y_j[w_{ij}]\right)\,\hat{}\,2}$

We start with the error term for the jth training pair. Note that $y_j$depends on the inputs x, but these are fixed. At this point, we care about the variable weights.

Out[102]= $$\frac{1}{2}\left(t_j - y_j(w_{\text{ij}})\right)^2$$

Take the derivate with respect to w_{ij}

In[28]:= $\mathtt{eq1 = D\left[(1\,/\,2)\,*\,\left(t_j - y_j[w_{ij}]\right)\,\hat{}\,2,\; w_{ij}\right]}$

Out[28]= $$-\left(t_j - y_j(w_{\text{ij}})\right)\,y_j'(w_{\text{ij}})$$

y depends on h which in turn depends on the w's, so using the chain rule in calculus, we can write $y_j{}' \left(w_{ij}\right)$ as:

In[29]:= $\mathtt{eq2 = D[y_j[h_j[w_{ij}]],\; w_{ij}]}$

Out[29]= $$h_j'(w_{\text{ij}})\,y_j'\left(h_j(w_{\text{ij}})\right)$$

But $y_j'\left(h_j\right)$ is the derivative of the squashing function f: $y_j'\left(h_j\right)$=f'(h_j) So we can substitute f' for y in eq2

In[30]:= $\mathtt{eq3 = eq2\;\;/.\;\;y_j{}'[h_j[w_{ij}]] \to f\,'\,[h_j]}$

Out[30]= $$f'(h_j)\,h_j'(w_{\text{ij}})$$

Now h is given by:

In[31]:= $$\mathbf{h_j = \sum_{k=1}^{N} x_k\, w_{j,k}}$$

Out[31]= $$\sum_{k=1}^{N} x_k\, w_{j,k}$$

Let's pick an arbitrary input input, say i=5. Suppose we take the derivative with respect to $w_{j,i}$, with i = 5? Then with respect to $w_{j,5}$, we have:

In[32]:= $D\left[\sum_{k=1}^{10} \mathbf{x}_k \, \mathbf{w}_{j,k},\ \mathbf{w}_{j,5}\right]$

Out[32]= x_5

So from this we can guess that in general (i.e. any i) we should have:

$$\Delta w_{ij} = -\left(t_j - y_j\left(w_{ij}\right)\right) f'\left(h_j\right) x_i$$

This is the "delta-rule".

We don't have direct access to the hidden units, and the problem is how to find the error signals for the hidden layers. It turns out that the delta error terms can be propagated back in terms of a weighted sum of the delta terms at the level above. We will see how this update rule gets applied to the weights starting from the top--i..e. those closest to the outputs, and then applied recursively down towards the inputs. Hence "back-propagation". We will relabel $x_k$to be y_k^λ, where λ indicates the layer. So y_k^0corresponds to the set of inputs to the network (bottom of figure).

To derive the recursive rules to relate deltas at an earlier layer to the next higher layer takes some work and careful booking with indices that I won't go through here.

Let's look at a summary of the algorithm that results.

1. Initialize the weights to small random values

2. Pick a pattern from the input/output collection, say the pth pattern: {x^p,t^p}: Calculate a delta term (analogous to the Widrow-Hoff rule) for the output layer L:

$$\partial_i^L = \left(t_i^p - y_i^L\left(x^p\right)\right) f'\left(u_i^L\right)$$

(See where it is useful to have an expression for the derivative of the squashing function f().)

3. Propagate the errors back through the layers:

$$\partial_i^\lambda = f'(u_i^\lambda) \sum_{k=1}^{N} \partial_k^{\lambda+1} w_{ki}^{\lambda+1} \qquad \lambda = \mathrm{L}-1,\ldots,1$$

...the error back propagation or "back prop" part.

4. Calculate weight adjustments (analogous to the outerproduct part of the Widrow-Hoff) and update using:

$$\Delta w_{ij}^\lambda = \quad \eta \partial_i^\lambda y_j^{\lambda-1}$$

5. Repeat steps 2 to 4 until convergence.

One can accumulate the weight adjustments for each training pair, and then update them all at once.

$$(\Delta w_{ij}^{\lambda})_p = \quad \eta \partial_i^{\lambda} y_j^{\lambda - 1}$$

$$\Delta w_{ij}^{\lambda} = \sum_{p=1}^{M} (\Delta w_{ij}^{\lambda})_p$$

In practice, updating the weights after each training pair often works better than accumulating a bunch of input/output pairs, and then computing the cumulative global error. The reason is that by randomly sampling a training pair, the "descent" may actually climb the global error function defined by the entire set. As we will see later with the Boltzmann machine, occasional climbing is useful to avoid local minima.

There are a number of derivations and illustrations on the web that you might find useful. See for example, http://galaxy.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

Backprop simulation example: XOR

It is well-known that with appropriate weights, 2 weight layers with 3 hidden units can solve the XOR problem. But this is still a tough problem to learn, mainly because it requires that two very different inputs map to the same output. Let's try learning the weights.

Reading in packages

So far, we've only used standard packages that are part of the *Mathematica* package. For these exercises, we will use a publicly available package written by James A. Freeman which can be downloaded from:

http://www.mathsource.com/cgi-bin/MathSource/Publications/BookSupplements/Freeman-1993/0205-906

Different computer systems have different ways of handling directory structures. You can find out where your current default directory is by typing:

```
In[33]:= Directory[]
```

```
Out[33]= /Users/kersten
```

Then you can set your default directory (for example) to read custom packages, or to save your data in a particular place:

SetDirectory["myfavoritedirectory"]

...but you'll have to determine where your Backpropagation.m file is, and set the directory appropriately. On the Mac, you select the file icon, and then use Get Info (under the Finder file menu) to determine where the file is. After you've set the directory, you can read in the package:

```
In[34]:= <<Backpropagation.m

         Get::noopen : Cannot open Backpropagation.m. ≫

Out[34]= $Failed
```

That said, it is usually easier to bring up a window for file browsing using:

```
In[103]:= Import[SystemDialogInput["FileOpen"], "Package"];
```

and then find and select **Backpropagation.m.**

Standard backprop

We will first try a straightforward implementation of the algorithm described above , called **bpnStandard[]**, which is in **Backpropagation.m**. You can open up the package and see how this and other functions are defined. But we replicate it here to show that the basic operations of error backpropagation are rather straightforward. You don't have to execute the following function because if you've read it in, it is defined in **Backpropagation.m**.

■ The bpnStandard backprop function

Some sample inputs:

```
ioPairsXOR = { {{0.9,0.9},{0.1}}, {{0.1,0.1},{0.1}}, {{0.1,0.9},{0.9}},
{{0.9,0.1},{0.9}} };
```

```
bpnStandard[inNumber_,hidNumber_,outNumber_,ioPairs_,eta_,numIters_] :=
  Module[{errors,hidWts,outWts,ioP,inputs,outDesired,hidOuts,  outputs, outE
    hidWts = Table[Table[Random[Real,{-0.1,0.1}],{inNumber}],{hidNumber}];
    outWts = Table[Table[Random[Real,{-0.1,0.1}],{hidNumber}],{outNumber}];
    errors = Table[
                        (* select ioPair *)
    ioP=ioPairs[[Random[Integer,{1,Length[ioPairs]}]]];
    inputs=ioP[[1]];
    outDesired=ioP[[2]];
                        (* forward pass *)
    hidOuts = sigmoid[hidWts.inputs];
    outputs = sigmoid[outWts.hidOuts];
                        (* determine errors and deltas *)
    outErrors = outDesired-outputs;
```

```
outDelta= outErrors (outputs (1-outputs));
```

$$\partial_i^L = \left(t_i^p - y_i^L\left(x^p\right)\right) f'\left(u_i^L\right)$$

Note that the identity that we derived earlier relating the derivative of the sigmoid to the sigmoid was used in the above *Mathematica* line.

```
hidDelta=(hidOuts (1-hidOuts)) Transpose[outWts].outDelta;
                        (* update weights *)
```

$$\partial_i^\lambda = f'(u_i^\lambda)\sum_{k=1}^{N} \partial_k^{\lambda+1} w_{ki}^{\lambda+1} \qquad \lambda = \mathrm{L}-1,\ldots,1$$

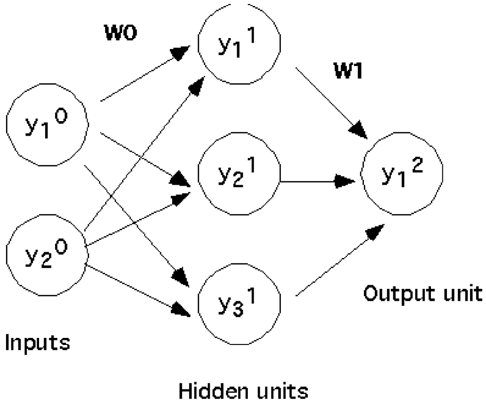
```
outWts += eta Outer[Times,outDelta,hidOuts];
hidWts += eta Outer[Times,hidDelta,inputs];
```

$$\Delta w_{ij}^{\lambda} = \eta \partial_i^{\lambda} y_j^{\lambda-1}$$

(Note the C-style notation: x += a, is the same as: x = x + a)

```
(* add squared error to Table *)
      outErrors.outErrors,{numIters}];  (* end of Table *)
 Return[{hidWts,outWts,errors}];
 ];
```

Running the algorithm



We will first try a standard backpropagation network with 2 input units, 3 hidden layer units, and 1 output unit (same network diagram as at the beginning of this notebook). The learning constant **eta**, we will set to 5. And let's try it for 1500 iterations.

bpnStandard[] expects a list with the input/output pairs set up as follows (for an XOR training set).

In[104]:=
```
ioPairsXOR = { {{0.9,0.9},{0.1}}, {{0.1,0.1},{0.1}}, {{0.1,0.9},{0.9}},
{{0.9,0.1},{0.9}} };
```

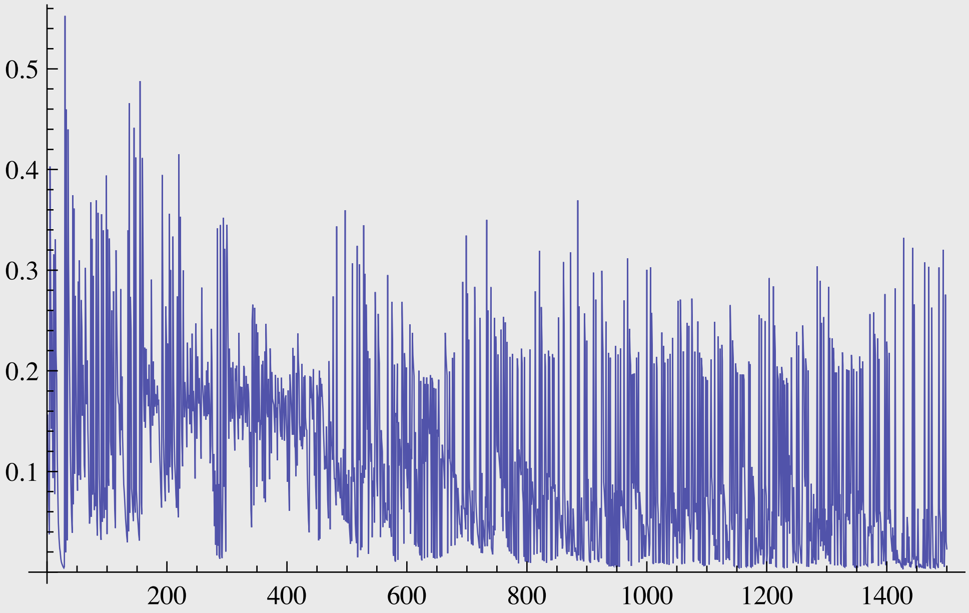
In[105]:=
```
Timing[outs=bpnStandard[2,3,1,ioPairsXOR,5,1500];]
```

Out[105]=
```
{0.251502, Null}
```

Did the net converge? No. The errors wiggle all over and never settle down near zero.

In[106]:=
```
ListPlot[outs⟦3⟧, Joined → True]
```

Out[106]=



We can see specifically where it is failing by calling **bpnTest[]**:

In[107]:=
```
bpnTest[outs[[1]],outs[[2]],ioPairsXOR];
```

```
Output 1 = {0.604353} desired = {0.1} Error = {−0.504353}


Output 2 = {0.172017} desired = {0.1} Error = {−0.0720165}


Output 3 = {0.695837} desired = {0.9} Error = {0.204163}


Output 4 = {0.76935} desired = {0.9} Error = {0.13065}


Mean Squared Error = 0.0795775
```

Improving the standard algorithm by preventing overlearning of certain patterns

The above network had a hard time learning the first pattern. Even with more iterations, you may discover the network to be stuck in a local minimum of the error function.

You could try more units in the hidden layer. This probably won't help much.

One trick to improve the odds of convergence is avoid over-learning certain patterns. The function **bpnMomentumSmart** sets a maximum acceptable error for a given pattern to 0.1, and then if the error is less than that for a given iteration, the weights are not updated. The idea is to concentrate more on learning the associations where the net shows some obstinance. (This network also uses a momentum term, the weight of this term is determined by alpha, which below is set to 0.9. Momentum is discussed below)

But you have to be lucky. I tried the following several times, before the algorithm nailed it:

In[112]:=

```
outs={0,0,0,0};
Timing[
outs=bpnMomentumSmart[2,3,1,ioPairsXOR,2.0,0.9,1300];]
```

New hidden−layer weight matrix:

$$\begin{pmatrix} 3.5845 & -9.41898 \\ -0.783742 & -1.43191 \\ -9.6124 & 3.76858 \end{pmatrix}$$

New output−layer weight matrix:

$$\begin{pmatrix} 5.67589 & -12.4747 & 5.89713 \end{pmatrix}$$

Output 1 = {0.19237} desired = {0.1} Error = {−0.0923703}

Output 2 = {0.196917} desired = {0.1} Error = {−0.0969173}

Output 3 = {0.94727} desired = {0.9} Error = {−0.0472703}

Output 4 = {0.804331} desired = {0.9} Error = {0.0956687}

Mean Squared Error = 0.00732805

Out[113]=

```
{0.194452, Null}
```

In[114]:=

```
ioPairsXOR
```

Out[114]=

$$\begin{pmatrix} \{0.9, 0.9\} & \{0.1\} \\ \{0.1, 0.1\} & \{0.1\} \\ \{0.1, 0.9\} & \{0.9\} \\ \{0.9, 0.1\} & \{0.9\} \end{pmatrix}$$

```
In[115]:= bpnTest[outs[[1]],outs[[2]],ioPairsXOR];
```

```
Output 1 = {0.19237} desired = {0.1} Error = {−0.0923703}

Output 2 = {0.196917} desired = {0.1} Error = {−0.0969173}

Output 3 = {0.94727} desired = {0.9} Error = {−0.0472703}

Output 4 = {0.804331} desired = {0.9} Error = {0.0956687}

Mean Squared Error = 0.00732805
```

Momentum

A standard modification to backprop that typically has a significant effect on learning speed is a **momentum term**. The idea, as the name suggests, is to keep the weight changes moving in about the same direction that they have been going. For example, for standard backprop, the hidden units were updated as:

```
In[44]:= hidWts += eta Outer[Times,hidDelta,inputs];
```

With momentum, the weights are updated in the same way except that alpha times the previous weight update is added in:

```
In[45]:= hidLastDelta =
         eta Outer[Times,hidDelta,inputs]+ alpha hidLastDelta;
         hidWts += hidLastDelta;
```

The momentum term was included in **`bpnMomentumSmart[].`**

Exercise: Can you find better learning parameters eta, and momentum term alpha?

Exercise: Try learning OR and AND.

```
In[47]:= ioPairsOR = { {{0.9,0.9},{0.9}}, {{0.1,0.1},{0.1}}, {{0.1,0.9},{0.9}},
         {{0.9,0.1},{0.9}} };
```

In[48]:=

```
Timing[outs=bpnStandard[2,3,1,ioPairsOR,5,1500];]
```

Out[48]=

```
{0.253577, Null}
```

In[49]:=

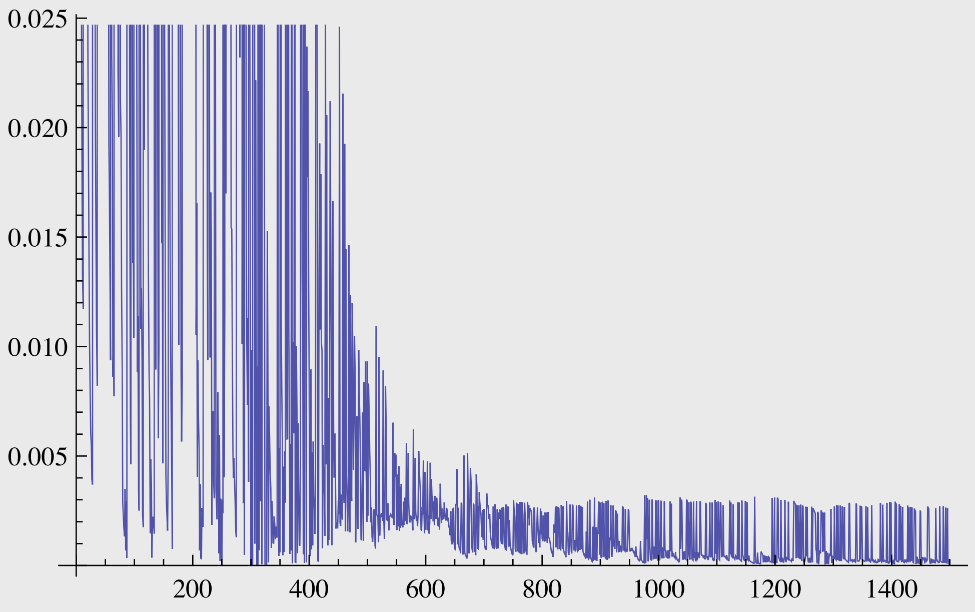
```
bpnTest[outs[[1]],outs[[2]],ioPairsOR];
```

```
Output 1 = {0.951026} desired = {0.9} Error = {−0.0510257}

Output 2 = {0.110089} desired = {0.1} Error = {−0.0100893}

Output 3 = {0.886146} desired = {0.9} Error = {0.0138544}

Output 4 = {0.884828} desired = {0.9} Error = {0.0151718}

Mean Squared Error = 0.000781886
```

In[50]:=

```
ListPlot[outs⟦3⟧, Joined → True]
```

Out[50]=



```
In[51]:=  outs[[1]]
          outs[[2]]
```

```
Out[51]=  ( 2.57106    2.5303   )
          ( −2.71095  −2.73164  )
          ( −2.04329  −2.0842   )
```

```
Out[52]=  ( 3.14513  −6.52994  −4.1613 )
```

Exercise: Can you find a learning sequence which improves the odds of standard backpropagation finding a solution to the XOR problem?

References

Rumelhart, D. E., McClelland, J. L., & Group, and the PDP Research Group (1986). Parallel Distributed Processing. Explorations in the Microstructure of Cognition . Cambridge, Massachusetts: MIT Press. See chapter 9 in Vol. I, page 318.

Freeman, J. A. (1994). Simulating Neural Networks with Mathematica . Reading, MA: Addison-Wesley Publishing Company.

http://www.mathsource.com/cgi-bin/MathSource/Publications/BookSupplements/Freeman-1993/0205-906