Introduction to Neural Networks
U. Minn. Psy 5038
Spring, 1998

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

Introduction to multi-layer nets

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

```
y1 := W0.y0;
y2 := W1.y1;
```

is equivalent to:

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

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)]];
```

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

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

As we will see later, it can be important for theoretical reasons, to have a non-linearity which is smooth enough to be differentiable:

```
Df[x_] := D[f[t],t] /.t->x
```

```
Df[x]
```

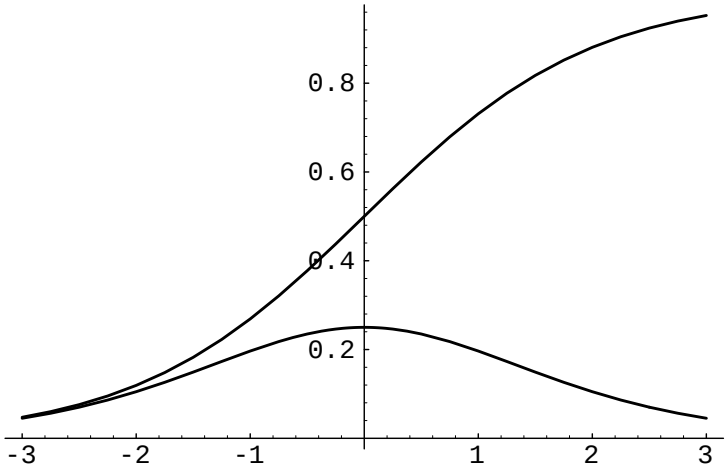
$$\frac{1}{E^{x}\ (1 + E^{-x})^{2}}$$

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

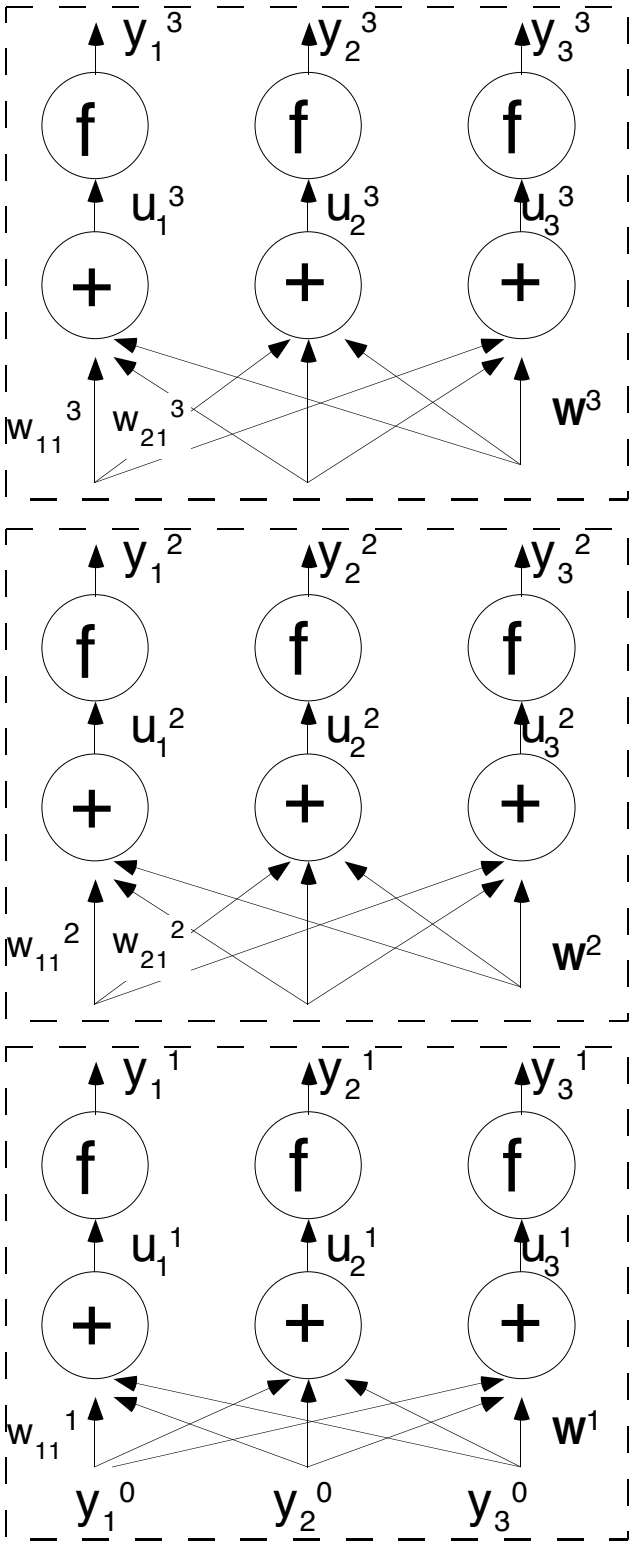
```
h[x_] := f[x](1-f[x]);
```

Here is a plot of the sigmoid and its derivative:

```
Plot[{Df[x],f[x]},{x,-3,3}];
```



OK, so suppose we have a multi-layer network with inputs y0. The output of the first layer is: $\mathbf{y^1} = f[\mathbf{u^1}] = f[\mathbf{W^1.y^0}]$. The output of the second layer is: $\mathbf{y^2} = f[\mathbf{u^2}] = f[\mathbf{W^2.y^1}]$. And so forth.



$$\Delta w_{ij}^{\lambda} = -\eta \frac{\partial E^2}{\partial w_{ij}^{\lambda}}$$

The problem is 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?

Backpropagation algorithm

We can approach the problem in the same way as we did for the linear network. Calculate an error term which is a function of the weights (with parameters determined by the input/output pairs), and then adjust the weights in such a way as to move down the error surface.

I won't go through the derivation here. It is complex mainly because of having to keep track of lots of indices as the chain rule from calculus is applied.

Here is 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:

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

3. Propagate the errors back through the layers:

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

4. Calculate weight adjustments and update.

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

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. As we discussed in the previous lecture on the Widrow-Hoff rule, this is true gradient descent on the error surface defined by the entire training ensemble.

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

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

In practice, updating the weights after each training pair often works better. 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.

Backprop simulation example: XOR

It is well-known that with appropriate weights, a 3 layer net 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:

```
Directory[]
```

```
DT-4110AV-1:Applications:Programming:Mathematica 2.2.2
```

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

```
SetDirectory["Macintosh HD:Lect_12"]
```

```
<<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

```
bpnStandard[inNumber_,hidNumber_,outNumber_,ioPairs_,eta_,numIters_] :=
 Module[{errors,hidWts,outWts,ioP,inputs,outDesired,hidOuts,  outputs, outErrors,outDelta,hidDelta},
  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));
```

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

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

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

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

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

(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. 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).

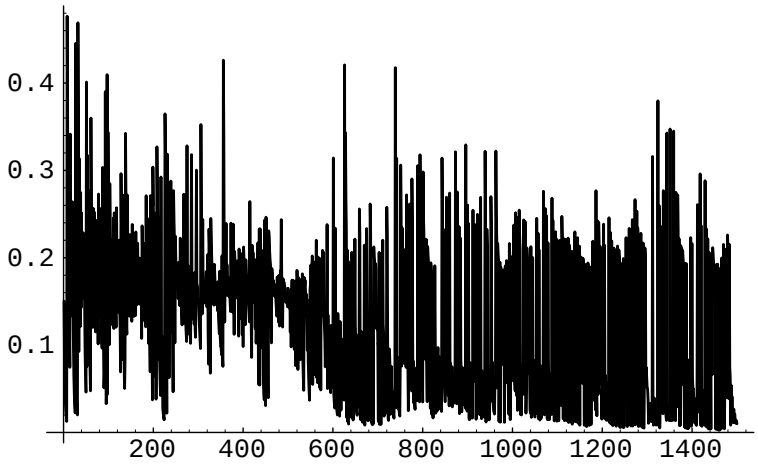
```
ioPairsXOR = { {{0.1,0.1},{0.1}}, {{0.1,0.9},{0.9}},
{{0.9,0.1},{0.9}}, {{0.9,0.9},{0.1}} };
```

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

```
{33.5833 Second, Null}
```

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

```
ListPlot[outs[[3]],PlotJoined->True];
```



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

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

```
 Output 1 = {0.204165} desired = {0.1} Error = {-0.104165}
 Output 2 = {0.762727} desired = {0.9} Error = {0.137273}
 Output 3 = {0.813478} desired = {0.9} Error = {0.0865222}
 Output 4 = {0.668159} desired = {0.1} Error = {-0.568159}
Mean Squared Error = 0.0899962
```

Improving the standard algorithm by preventing overlearning of certain patterns

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

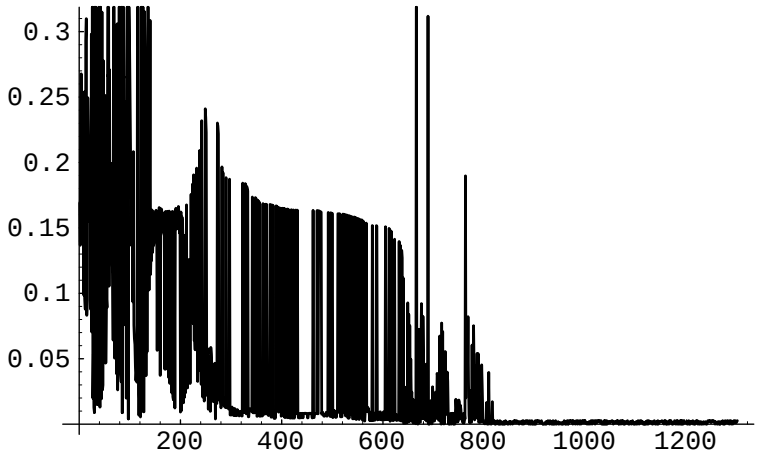
You could try more units in the hidden layer. This probably won't help much.

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 I hit it:

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

```
New hidden-layer weight matrix:
{{-0.937564, -1.09841}, {-9.95589, 3.85642}, {3.79034, -10.1737}}
New output-layer weight matrix:
{{-13.2328, 6.06398, 6.04958}}
 Output 1 = {0.151955} desired = {0.1} Error = {-0.0519553}
 Output 2 = {0.904261} desired = {0.9} Error = {-0.00426053}
 Output 3 = {0.865767} desired = {0.9} Error = {0.034233}
 Output 4 = {0.144143} desired = {0.1} Error = {-0.0441427}
Mean Squared Error = 0.0014595
```




```
{25.7667 Second, Null}
```

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:

```
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:

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

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

■ Optional exercise

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

References

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

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