

## Introduction to Neural Networks U. Minn. Psy 5038

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

#### Introduction to multi-layer nets: Computation and recall

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

$$\begin{aligned} y_1 &:= W_0 \cdot y_0; \\ y_2 &:= W_1 \cdot y_1; \end{aligned} \quad (1)$$

is equivalent to:

$$y_2 := W_1 \cdot (W_0 \cdot y_0) := W_1 \cdot W_0 \cdot y_0 := W_3 \cdot y_0; \quad (2)$$

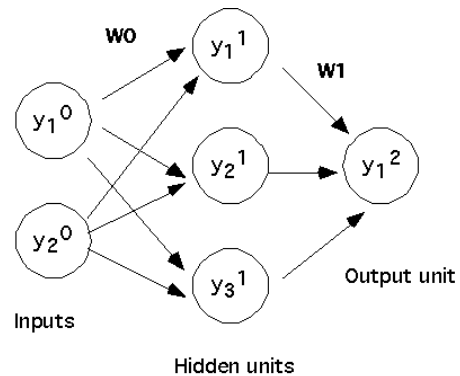
where  $W_3$  is just another matrix. However, if the inner product is followed by a non-linear transformation, then concatenating layers of neural elements is no longer trivial:

$$y_2 := f[W_1 \cdot f(W_0 \cdot y_0)]; \quad (3)$$

where  $f[\ ]$ , for example, is a sigmoid.

$$f[x_] := 1 / (1 + \text{Exp}[-(x - .5)]);$$

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



```

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?

$y_2[\{0, 0\}]$

$\{0\}$

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

```

W0 = {{-1.196538262126091, -0.8796120440036393}, {-9.222528381458348,
3.637813962288577},
{3.633609140128183, -8.88333887824766}};
W1 = {{-13.8761692810796, 5.999907050231006, 6.165721544099243}};
  
```

$\text{Chop}[y_2[\{1, 1\}]]$

$\{0\}$

$y_2[\{0, 1\}]$

$\{1.\}$

#### ■ 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 derivative of another we write:

```

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:

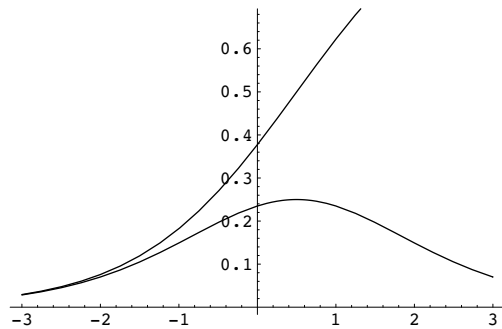
```
(*Df[x_] := Derivative[f[x],x]*)
Df[x_] := f'[x]
```

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[h[x]]`, or `Simplify[Df[x]-h[x]]`:

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

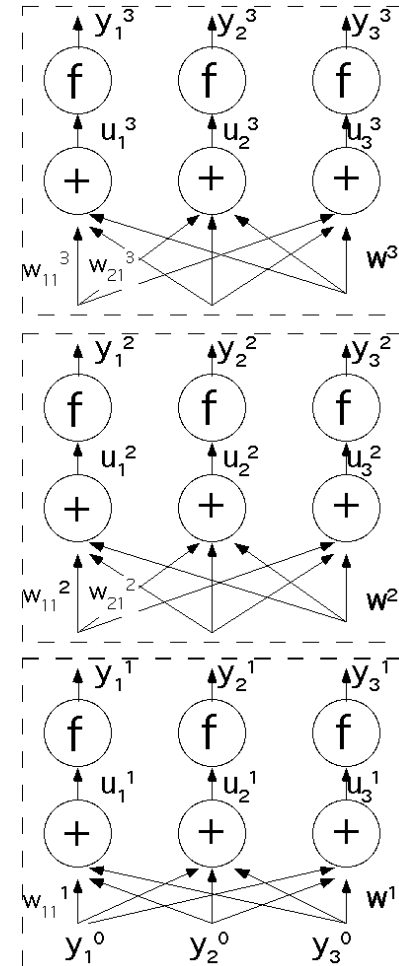
Here is a plot of the sigmoid and its derivative:

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



### ■ Generic multi-layer feedforward net

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

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

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

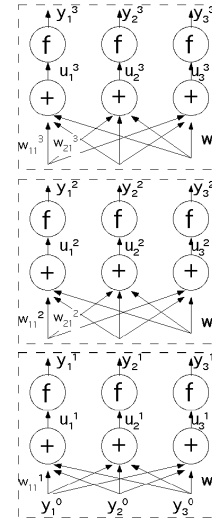
$$E(\{w_{ij}^\lambda\}) = \sum_{k=1}^N (y_k(y^0; w_{ij}^\lambda) - t_k)^2$$

where we've summed over all N output units. (A subscript means the component of the corresponding vector of activity.) 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:

$$\text{new } w_{ij}^\lambda = \text{previous } w_{ij}^\lambda + \Delta w_{ij}^\lambda \quad (4)$$

where

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



...but we need to calculate the gradient, which will be an expression that tells us how to adjust the  $ij$ th weight for the connection in the  $\lambda$ th layer.

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. (\*Project Idea: Use *Mathematica*\*).

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  $p$ th pattern:  $\{x^p, t^p\}$ : Calculate a delta term (analogous to the Widrow-Hoff rule) for the output layer L:

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

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:

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

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

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

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

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.

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

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

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.

## 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:

```
Directory[]
```

```
PowerbookHD:Applications (Mac OS 9):Programming:Mathematica 4.1 Files
```

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:

```
<<Backpropagation.m
```

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

```
Import[Experimental`FileBrowse[False]];
```

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,
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));

```

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

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 *)

```

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

```

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

```

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

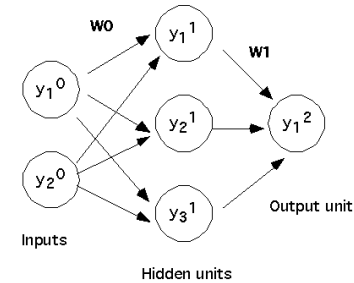
(Note the C-style notation:  $x += a$ , is the same as:  $x = x + a$ )

```

(* 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).

```

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}} };

```

```

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

```

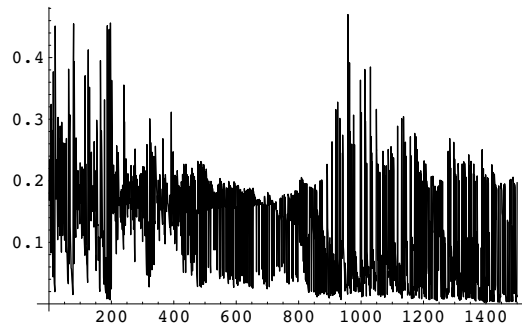
```

{3.18333 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 = {0.537691} desired = {0.1} Error = {-0.437691}
```

```
Output 2 = {0.157867} desired = {0.1} Error = {-0.0578666}
```

```
Output 3 = {0.792102} desired = {0.9} Error = {0.107898}
```

```
Output 4 = {0.531681} desired = {0.9} Error = {0.368319}
```

```
Mean Squared Error = 0.0855556
```

### 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:

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

New hidden-layer weight matrix:

```
{{-4.21221, -8.00302}, {-3.32035, -6.66567}, {-0.13548, -5.89153}}
```

New output-layer weight matrix:

```
{{-12.2988, -2.2086, 5.15553}}
```

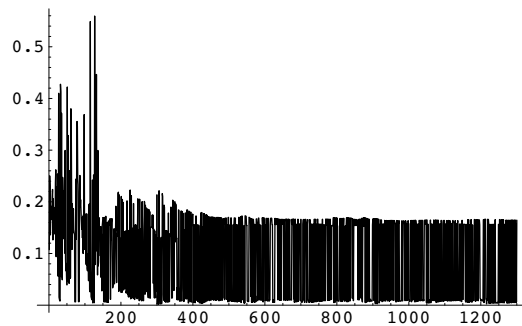
```
Output 1 = {0.505536} desired = {0.1} Error = {-0.405536}
```

```
Output 2 = {0.172073} desired = {0.1} Error = {-0.0720735}
```

```
Output 3 = {0.503818} desired = {0.9} Error = {0.396182}
```

```
Output 4 = {0.820374} desired = {0.9} Error = {0.0796261}
```

Mean Squared Error = 0.0832385



```
{2.81667 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[]`.

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

**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>

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