SECTION 9 Classification (Cont'd)

```
drive <- "D:"
code.dir <- paste(drive, "DATA/Data Mining R-Code", sep="/")
data.dir <- paste(drive, "DATA/Data Mining Data", sep="/")
#
source(paste(code.dir, "CreateGrid.r", sep="/"))
source(paste(code.dir, "ClassBoundary.r", sep="/"))
source(paste(code.dir, "confusion_ex.r", sep="/"))
```

Neural Networks

The basic concept in neural networks is the weighting of inputs to produce the desired output result. Suppose we consider a modification of an example that is familiar to students - you write 4 tests worth 10% each, do 2 assignments worth 5% each and a final exam worth 50%. Your mark in the course will be found from

$$0.1 * t_1 + 0.1 * t_2 + 0.1 * t_3 + 0.1 * t_4 + 0.05 * a_1 + 0.05 * a_2 + 0.5 * f_1$$

If this mark exceeds a threshold (say 50%) you pass, otherwise you fail. In this case, you are told what the weighting will be.

Now suppose that your professor does not tell you what the weights will be but instead decides to set the weights so that the students that (s)he thinks should pass will pass and the others will fail. The professor would then have to start off with weights (maybe at random) and see what happens to each student. If a student that the professor feels should pass exceeds the threshold, no change needs to be made in the weights. On the other hand, if the expected result does not occur, the weights need to be altered. Of course, changing the weights means that the professor then has to check on how these new weights affect the outcome for students whose grades were calculated using the old weights. The process will thus involve going though the student marks a number of times until no changes occur. Note that there may be cases for which the result will not be that which the professor desired - for example, if two students have the same marks and the professor felt one should pass and the other fail, it is impossible to find weights to produce the desired result.

This type of process can be done with a simple neural net - a Threshold Logic Unit (TLU, shown below):

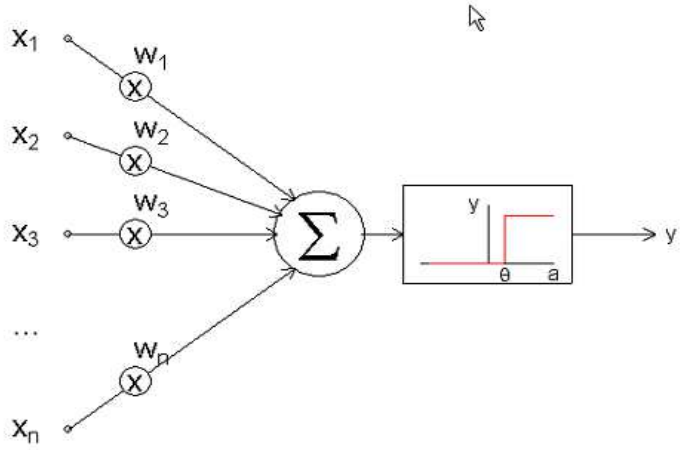


Figure 1. Threshold Logic Unit

Here the *n* inputs (marks, signals) $x_1, x_2, ..., x_n$ with weights $w_1, w_2, ..., w_n$ are put into an adder, so that the total input to a node is

$$a = x_1 w_1 + x_2 w_2 + ... + x_n w_n = \sum_{i=1}^{n} x_i w_i = \mathbf{x} \cdot \mathbf{w}$$

Suppose the node has a *threshold* (call it θ) such that if $a \geq \theta$ the node will 'fire' so the output is

$$y = \begin{cases} 1 & \text{if } a \geq \theta \\ 0 & \text{if } a < \theta \end{cases}$$

In a simple case of a TLU having 2 inputs (with values of 0 or 1) with initial weights of 1 and a threshold of 1.5, we get a response table

x_1	x_2	w_1	w_2	$a = \mathbf{x} \cdot \mathbf{w}$	θ	y
0	0	1	1	0	1.5	0
0	1	1	1	1	1.5	0
1	0	1	1	1	1.5	0
1	1	1	1	2	1.5	1

These inputs could be considered to be points at the corners of a rectangle in two dimensions (see below). The TLU classifies them into 2 classes (0 - fail, 1 - pass).

If we look at the situation in this way, we see that we will have a line

$$\mathbf{x} \cdot \mathbf{w} = \theta \qquad \text{or} \qquad \mathbf{x} \cdot \mathbf{w} - \theta = 0$$

$$\text{or} \qquad x_1 w_1 + x_2 w_2 = \theta$$

$$\text{so} \qquad x_2 = -\left(\frac{w_1}{w_2}\right) x_1 + \left(\frac{\theta}{w_2}\right)$$

$$\text{i.e.} \qquad x_2 = A x_1 + B$$

that divides the plane. In our example the slope $A = -1$ and the y-intercept $B = 1.5$.

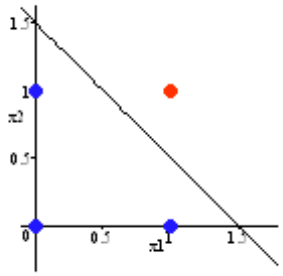


Figure 2. Logical AND

(We note that this is a logical AND). If this is what we wish to obtain, then we expect

x_1	x_2	t
0	0	0
0	1	0
1	0	0
1	1	1

The *t* value is the result that we want or the *target* value (if we *know* the results we want, we are using 'supervised' learning).

Suppose that we do not know the weights but are required to find them by '*training*'. i.e. we are given a set of weights (possibly random) and a threshold and have to adjust the weights and threshold based on the results that they produce.

For example, if we start with $w_1 = 0.2,\ w_2 = 0.4,$ and $\theta = 0.3,$ then

$$\begin{aligned} x_2 &= -\left(\frac{0.2}{0.4}\right)x_1 + \left(\frac{0.3}{0.4}\right) \\ &= -0.5x_1 + 0.75 \end{aligned}$$

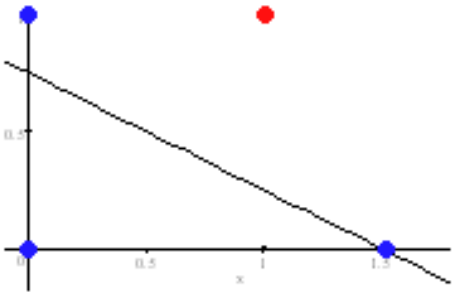


Figure 3. Linear non-separation

which does not do the desired classification (logical AND).

How can we go about adjusting (*training*) the **weights** and **threshold**?

Because both are to be trained, we will consider a variation of the preceeding by letting the threshold be treated as another "weight" attached to an input of –1. So the weight vector now becomes

$$\mathbf{w} = [w_1, w_2, \theta]$$

and the input vector becomes

$$\mathbf{x} = [x_1, x_2, -1] = [\mathbf{v}, -1]$$

so now $\mathbf{x} \cdot \mathbf{w} = x_1 w_1 + x_2 w_2 - \theta$. This means that we have a new type of node that fires if the "activation signal" a is ≥ 0. That is

$$y = \begin{cases} 1 & \text{if} \quad a = \mathbf{x} \cdot \mathbf{w} \geq 0 \\ 0 & \text{if} \quad a = \mathbf{x} \cdot \mathbf{w} < 0 \end{cases}$$

What will happen if we adjust a weight? Let the first weight become $w_1 + \delta_{w_1}$ so

$$\begin{aligned}\mathbf{x}\cdot\mathbf{w}^* &= x_1(w_1+\delta_{w_1})+x_2w_2-\theta\\ &= x_1w_1+x_2w_2-\theta+x_1\delta_{w_1}\\ &= \mathbf{x}\cdot\mathbf{w}+x_1\delta_{w_1}\end{aligned}$$

Similarly we modify the second weight w_2. From this we see that if we have an activation level that is "too low", we could 'add' $[\delta_{w_1}\ \ \delta_{w_2}]$ to the weight vector (similarly for too high). If we look at

Response from the neural net y	Correct or desired response t	Change	$t-y$
0	0	no	0
0	1	+	1
1	0	–	–1
1	1	no	0

we note that any change will involve $(t-y)$. One possibility would be to add $(t-y)(x_1,x_2)=(t-y)\mathbf{v}$ but this might overshoot or overcorrect so we use a *learning* rate (a *damping* factor) α and adjust the weights (including threshold) by

$$\Delta\mathbf{w} = \alpha[(t-y)\mathbf{v}]$$

To help illustrate, we need some functions. The following function sets

$$y=\begin{cases}1 & \mathbf{x}\cdot\mathbf{w}\geq\mathbf{0}\\ 0 & \mathbf{x}\cdot\mathbf{w}<\mathbf{0}\end{cases}$$

and returns $\Delta\mathbf{w}$ = *learning rate* $*$ (*class* $-y$) $*$ **input** (*class* is the t in the previous formula).

Note the use of `formatC` in the printing of the results. It allows us to display the results as floating point values with 4 digits after the decimal point and a total width of 7.

```
comp.delta.w <- function (input, w, class, rate){
   activation <- t(input)%*%w;       # [x1,x2,-1].[w1,w2,thresh]
      # if activation > 0  y <- 1 (fire)
   y <- (activation > 0)*1
   factor <- rate*(class - y);
   cat(formatC(c(w, input, activation, y, class, factor, factor*input,
               1-(class==y)), format="f",
               digit=4, width=7), "\n")
   return(list(delta=factor*input, Err=1-(class == y)))
}
```

The following plots the data with classes coloured, puts the separating line on it, and updates the weights.

Note: `pch` sets the plotting character and `cex` sets the character size.

```
#============================================================
# Plot and update the weights
#============================================================
f.update.weights <- function (inputs, w, target, rate, XR, YR){
   Total.Error <- 0
   for (i in 1:(length(inputs[,1]))){
      plot(XR, YR, type = "n",
           main=paste(floor(w[1]*100+.5)/100, floor(w[2]*100+.5)/100,
           floor(w[3]*100+.5)/100),
           xlab="", ylab="")
      mtext(paste("b=",floor(w[3]/w[2]*100+.5)/100,
                  " m=",floor(w[1]/w[2]*100)/100),side=1, line=0.5, cex=.8)
      points (inputs[,1], inputs[,2], col=target+2, pch=19, cex=1.2)
      plot.line (w)
      points(inputs[i,1], inputs[i,2], col="blue", pch="x", cex=2)
      delta.w <- comp.delta.w(inputs[i,], w, target[i], rate)
      w <- w + delta.w$delta
      Total.Error <- Total.Error + delta.w$Err
   }
   return (list(w=w, Err=Total.Error))
}
#============================
# Plot a coloured line (w = [w1,w2,θ])
#============================
plot.line <- function (w){
   if (w[2] != 0){
      inter <- w[3]/w[2]
      slope <- w[1]/w[2]
      abline (inter,slope, col="red")
   }
   else {  # vertical line at xx
      xx <- w[3]/w[1]
      abline (v=xx, col="red")
   }
}
```

The function `f.AND` runs through the inputs (up to 200 times) and displays the results.

```
#====================
#  Main function
#====================
f.AND <- function(inputs, class, w, rate, XR, YR) {
      # Now use up to 200 training cycles to get the weights
   T.Err <- 0
   for (i in (1:200)) {
        # Display the inputs (coloured by class)
     cat("  w1      w2      b0      x1      x2     -1       act.     y",
           "     target  delta    dw1     dw2     db     \n")
     old.w <- w
```

```
    res <- f.update.weights (inputs, w, class, rate, XR, YR)
    w <- res$w
    cat("weights = ",w," old ",old.w, "Err = ", res$Err, "\n")
    if (res$Err == 0) {
       break
    }
    ret <- readline("Press Enter...")
    if (ret == "q") break
  }
}
```

Set the data for the problem:

```
rate <- 0.25;          # Learning rate
x1 <- c(0,0,-1)
x2 <- c(0,1,-1)
x3 <- c(1,0,-1)
x4 <- c(1,1,-1)
oldpar <- par(mfrow = c(5,4),mar=c(2,1,2,1),xaxt="n",yaxt="n")
f.AND(rbind(x1,x2,x3,x4), c(0,0,0,1), c(-.2, 0.2, -0.5), rate, -.5:1.5, -.5:1.5)
par(oldpar)
```

```
  w1      w2      b0      x1      x2    -1       act.     y      target  delta
-0.2000  0.2000 -0.5000  0.0000  0.0000 -1.0000  0.5000  1.0000  0.0000 -0.2500  0
-0.2000  0.2000 -0.2500  0.0000  1.0000 -1.0000  0.4500  1.0000  0.0000 -0.2500  0
-0.2000 -0.0500  0.0000  1.0000  0.0000 -1.0000 -0.2000  0.0000  0.0000  0.0000  0
-0.2000 -0.0500  0.0000  1.0000  1.0000 -1.0000 -0.2500  0.0000  1.0000  0.2500  0
weights =  0.05 0.2 -0.25  old  -0.2 0.2 -0.5 Err =  3
```

e.g. [–0.2000 0.2000 – 0.5000]•[0.0000 0.0000 – 1.0000]= 0.5 > 0 so the predicted class is $y = 1$. The true class is 0 so $delta = rate * (target - y) = -0.25$ and $delta * x$ = [0.0000 0.0000 0.2500]. The last column indicates if $y == target$.

```
  w1      w2      b0      x1      x2    -1       act.     y      target  delta
 0.0500  0.2000 -0.2500  0.0000  0.0000 -1.0000  0.2500  1.0000  0.0000 -0.2500  0
 0.0500  0.2000  0.0000  0.0000  1.0000 -1.0000  0.2000  1.0000  0.0000 -0.2500  0
 0.0500 -0.0500  0.2500  1.0000  0.0000 -1.0000 -0.2000  0.0000  0.0000  0.0000  0
 0.0500 -0.0500  0.2500  1.0000  1.0000 -1.0000 -0.2500  0.0000  1.0000  0.2500  0
weights =  0.3 0.2 0  old  0.05 0.2 -0.25 Err =  3

  w1      w2      b0      x1      x2    -1       act.     y      target  delta
 0.3000  0.2000  0.0000  0.0000  0.0000 -1.0000  0.0000  0.0000  0.0000  0.0000  0
 0.3000  0.2000  0.0000  0.0000  1.0000 -1.0000  0.2000  1.0000  0.0000 -0.2500  0
 0.3000 -0.0500  0.2500  1.0000  0.0000 -1.0000  0.0500  1.0000  0.0000 -0.2500 -0
 0.0500 -0.0500  0.5000  1.0000  1.0000 -1.0000 -0.5000  0.0000  1.0000  0.2500  0
weights =  0.3 0.2 0.25  old  0.3 0.2 0 Err =  3

  w1      w2      b0      x1      x2    -1       act.     y      target  delta
 0.3000  0.2000  0.2500  0.0000  0.0000 -1.0000 -0.2500  0.0000  0.0000  0.0000  0
 0.3000  0.2000  0.2500  0.0000  1.0000 -1.0000 -0.0500  0.0000  0.0000  0.0000  0
 0.3000  0.2000  0.2500  1.0000  0.0000 -1.0000  0.0500  1.0000  0.0000 -0.2500 -0
 0.0500  0.2000  0.5000  1.0000  1.0000 -1.0000 -0.2500  0.0000  1.0000  0.2500  0
weights =  0.3 0.45 0.25  old  0.3 0.2 0.25 Err =  2

  w1      w2      b0      x1      x2    -1       act.     y      target  delta
 0.3000  0.4500  0.2500  0.0000  0.0000 -1.0000 -0.2500  0.0000  0.0000  0.0000  0
 0.3000  0.4500  0.2500  0.0000  1.0000 -1.0000  0.2000  1.0000  0.0000 -0.2500  0
 0.3000  0.2000  0.5000  1.0000  0.0000 -1.0000 -0.2000  0.0000  0.0000  0.0000  0
```

```
 0.3000  0.2000  0.5000  1.0000  1.0000 -1.0000  0.0000  0.0000  1.0000  0.2500  0
weights =  0.55 0.45 0.25  old  0.3 0.45 0.25 Err =  2

  w1      w2      b0      x1      x2     -1       act.     y      target  delta
 0.5500  0.4500  0.2500  0.0000  0.0000 -1.0000 -0.2500  0.0000  0.0000  0.0000  0
 0.5500  0.4500  0.2500  0.0000  1.0000 -1.0000  0.2000  1.0000  0.0000 -0.2500  0
 0.5500  0.2000  0.5000  1.0000  0.0000 -1.0000  0.0500  1.0000  0.0000 -0.2500 -0
 0.3000  0.2000  0.7500  1.0000  1.0000 -1.0000 -0.2500  0.0000  1.0000  0.2500  0
weights =  0.55 0.45 0.5  old  0.55 0.45 0.25 Err =  3

  w1      w2      b0      x1      x2     -1       act.     y      target  delta
 0.5500  0.4500  0.5000  0.0000  0.0000 -1.0000 -0.5000  0.0000  0.0000  0.0000  0
 0.5500  0.4500  0.5000  0.0000  1.0000 -1.0000 -0.0500  0.0000  0.0000  0.0000  0
 0.5500  0.4500  0.5000  1.0000  0.0000 -1.0000  0.0500  1.0000  0.0000 -0.2500 -0
 0.3000  0.4500  0.7500  1.0000  1.0000 -1.0000  0.0000  0.0000  1.0000  0.2500  0
weights =  0.55 0.7 0.5  old  0.55 0.45 0.5 Err =  2

  w1      w2      b0      x1      x2     -1       act.     y      target  delta
 0.5500  0.7000  0.5000  0.0000  0.0000 -1.0000 -0.5000  0.0000  0.0000  0.0000  0
 0.5500  0.7000  0.5000  0.0000  1.0000 -1.0000  0.2000  1.0000  0.0000 -0.2500  0
 0.5500  0.4500  0.7500  1.0000  0.0000 -1.0000 -0.2000  0.0000  0.0000  0.0000  0
 0.5500  0.4500  0.7500  1.0000  1.0000 -1.0000  0.2500  1.0000  1.0000  0.0000  0
weights =  0.55 0.45 0.75  old  0.55 0.7 0.5 Err =  1

  w1      w2      b0      x1      x2     -1       act.     y      target  delta
 0.5500  0.4500  0.7500  0.0000  0.0000 -1.0000 -0.7500  0.0000  0.0000  0.0000  0
 0.5500  0.4500  0.7500  0.0000  1.0000 -1.0000 -0.3000  0.0000  0.0000  0.0000  0
 0.5500  0.4500  0.7500  1.0000  0.0000 -1.0000 -0.2000  0.0000  0.0000  0.0000  0
 0.5500  0.4500  0.7500  1.0000  1.0000 -1.0000  0.2500  1.0000  1.0000  0.0000  0
weights =  0.55 0.45 0.75  old  0.55 0.45 0.75 Err =  0
```

We see that, in the last set, the output *y* is identical to the target value (shown by the *delta* values being 0) and so the TLU has been "trained".

In the following figures, *m* is the slope and *b* is the *y*-intercept. In the figures, the blue × indicates which case is being 'presented' for training (we use a similar process with self-organizing maps (SOM)).

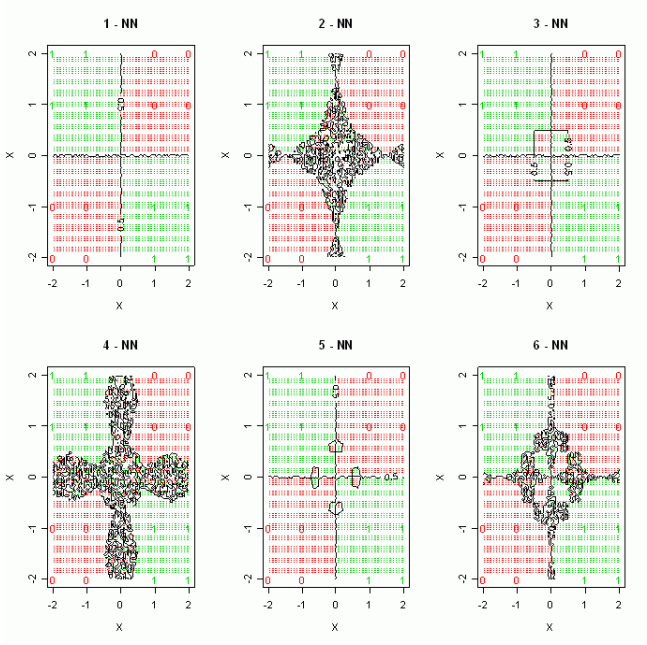


Figure 4. First 5 passes

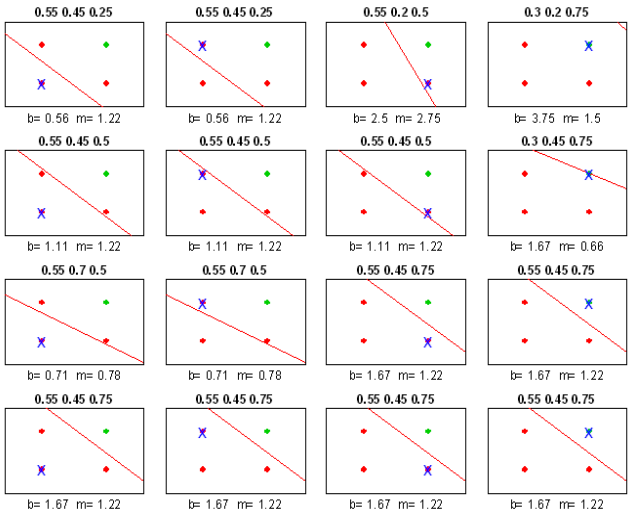


Figure 5. Next 4 passes

The *linear separator* has the equation

$$b0 + w1 * x + w2 * y = 0$$

$$\text{or} \quad y = -\left(\frac{b0}{w2} + \frac{w1}{w2}x\right)$$

$$= b + mx$$

Note: This simple training algorithm is very restricted in its ability to be extended to deal with more complicated partitioning using more nodes.

We have a value $(t - y)$ that is a measure of the error for a set of weights. Rather than consider the output y (which is not continous), we will consider what happens with the activation $a = \mathbf{x} \cdot \mathbf{w} = x_1 w_1 + ... + x_n w_n - \theta$. Suppose we consider as our error for a single node

$$E_i = \frac{1}{2}(t - a)^2$$

(the $\frac{1}{2}$ is there to remove the 2 that appears on differentiation of E_i).

We will try to find the set of weights **w** that give the minimum error. To do this we note that if a is considered to be a function of **w** (t is constant), then

$$\frac{\partial E_i}{\partial w_j} = -(t - a)\frac{\partial a}{\partial w_j} = -(t - a)\frac{\partial(x_1 w_1 + x_2 w_2 + ... + x_n w_n - \theta)}{\partial w_j}$$

$$= \begin{cases} -(t - a)x_j & j = 1, 2, ..., n \\ +(t - a)\,1 & \theta \end{cases}$$

The error curve is quadratic in **w**. What happens if we make a change in w_j that is proportional to the slope of the error curve?

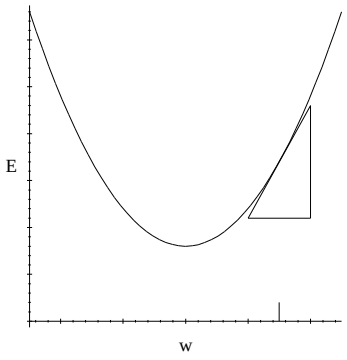


Figure 6. Gradient

i.e. if we let

$$\begin{aligned}\Delta w_j &= -\alpha \times \text{slope} \\ &= -\alpha \frac{\partial E_i}{\partial w_j} \\ &= \alpha(t-a)x_j\end{aligned}$$

$\left(\alpha = \textit{learning}\text{ rate}\right)$ then

$$\begin{aligned}\Delta E_i &\approx \frac{\partial E_i}{\partial w_j}\Delta w_j \\ &= -\alpha(t-a)^2 x_j^2 < 0\end{aligned}$$

so we move down towards the minimum.

When we did the original version, we used the fact that we had a 1 if the activation $a \geq 0$ and 0 if $a < 0$. We need to have our target activations take on the same behaviour. For this, we will use t to be either -1 or $+1$.
We can replace the `comp.delta.w` function with a new version that uses the *gradient*.

```
comp.delta.w <- function (input.i, w, target, rate){
   activation <- t(input.i)%*%w
         # As before [x1,x2,x3].[w1,w2,w3] = [x1,x2,-1].[w1,w2,b]
   factor <- rate*(target - activation)
   cat(formatC(c(w, input.i, activation, target,  factor,
                 factor*input.i, 0.5*(target-activation)^2),
               format="f", digit=4,width=7), "\n")
   return(list(delta=factor*input.i, Err=0.5*(target-activation)^2))
}
```

and make a slight change to the main function

```
f2.AND.G <- function(inputs, class, w, rate, XR, YR) {
      # Now use up to 200 training cycles to get the weights
```

```
   T.Err <- 9999999
   for (i in (1:200)) {
        # Display the inputs (coloured by class)
     cat("  w1      w2      b       x1      x2    -1        act.",
           "  target  delta    dw1     dw2     db     \n")
     old.w <- w
     res <- f.update.weights (inputs, w, class, rate, XR, YR)
     w <- res$w
     cat("weights = ",w," old ",old.w,"\n")
     cat("Err = ", res$Err, "\n")
     if (abs(res$Err - T.Err) > 0.01*res$Err) {
        T.Err <- res$Err
        ret <- readline("Press Enter...")
        if (ret == "q") break
      } else {
        break
     }
  }
}
x1 <- c(0,0,-1)
x2 <- c(0,1,-1)
x3 <- c(1,0,-1)
x4 <- c(1,1,-1)
oldpar <- par(mfrow = c(5,4), mar=c(2,1,2,1), xaxt="n", yaxt="n")
f2.AND.G(rbind(x1,x2,x3,x4), c(-1,-1,-1,1), c(-.2, 0.2, -0.5), rate,
-.5:1.5,-.5:1.5)
par(oldpar)
```

```
  w1      w2      b       x1      x2    -1        act.   target  delta    dw1
-0.2000  0.2000 -0.5000  0.0000  0.0000 -1.0000  0.5000 -1.0000 -0.3750  0.0000  0
-0.2000  0.2000 -0.1250  0.0000  1.0000 -1.0000  0.3250 -1.0000 -0.3312  0.0000 -0
-0.2000 -0.1312  0.2062  1.0000  0.0000 -1.0000 -0.4063 -1.0000 -0.1484 -0.1484  0
-0.3484 -0.1312  0.3547  1.0000  1.0000 -1.0000 -0.8344  1.0000  0.4586  0.4586  0
weights =  0.1101563 0.3273438 -0.1039063  old  -0.2 0.2 -0.5
Err =  3.861548
```

e.g. $[-0.2000 \;\; 0.2000 \;\; -0.5000] \cdot [0.0000 \;\; 0.0000 \;\; -1.0000] = 0.5$. The true class is -1
$rate * (target - activation) = 0.25 * (-1.5) = -0.375$ so
$delta * x = [0.0000 \;\; 0.0000 \;\; 0.3750]$. The last column is
$0.5 * (target - activation)^2 = 1.5^2/2 = 1.125$.

```
  w1      w2      b       x1      x2    -1        act.   target  delta    dw1
 0.1102  0.3273 -0.1039  0.0000  0.0000 -1.0000  0.1039 -1.0000 -0.2760  0.0000  0
 0.1102  0.3273  0.1721  0.0000  1.0000 -1.0000  0.1553 -1.0000 -0.2888  0.0000 -0
 0.1102  0.0385  0.4609  1.0000  0.0000 -1.0000 -0.3507 -1.0000 -0.1623 -0.1623  0
-0.0522  0.0385  0.6232  1.0000  1.0000 -1.0000 -0.6368  1.0000  0.4092  0.4092  0
weights =  0.3570496 0.4477356 0.2139954  old  0.1101563 0.3273438 -0.1039063
Err =  2.827031

  w1      w2      b       x1      x2    -1        act.   target  delta    dw1
 0.3570  0.4477  0.2140  0.0000  0.0000 -1.0000 -0.2140 -1.0000 -0.1965  0.0000  0
 0.3570  0.4477  0.4105  0.0000  1.0000 -1.0000  0.0372 -1.0000 -0.2593  0.0000 -0
 0.3570  0.1884  0.6698  1.0000  0.0000 -1.0000 -0.3128 -1.0000 -0.1718 -0.1718  0
 0.1852  0.1884  0.8416  1.0000  1.0000 -1.0000 -0.4680  1.0000  0.3670  0.3670  0
weights =  0.5522269 0.555414 0.474629  old  0.3570496 0.4477356 0.2139954
```

```
Err =  2.160428

  w1      w2      b       x1      x2     -1        act.   target  delta    dw1
 0.5522  0.5554  0.4746  0.0000  0.0000 -1.0000 -0.4746 -1.0000 -0.1313  0.0000  0
 0.5522  0.5554  0.6060  0.0000  1.0000 -1.0000 -0.0506 -1.0000 -0.2374  0.0000 -0
 0.5522  0.3181  0.8433  1.0000  0.0000 -1.0000 -0.2911 -1.0000 -0.1772 -0.1772  0
 0.3750  0.3181  1.0206  1.0000  1.0000 -1.0000 -0.3275  1.0000  0.3319  0.3319  0
weights =  0.706878 0.6499282 0.6886811  old  0.5522269 0.555414 0.474629
Err =  1.721121

  w1      w2      b       x1      x2     -1        act.   target  delta    dw1
 0.7069  0.6499  0.6887  0.0000  0.0000 -1.0000 -0.6887 -1.0000 -0.0778  0.0000  0
 0.7069  0.6499  0.7665  0.0000  1.0000 -1.0000 -0.1166 -1.0000 -0.2209  0.0000 -0
 0.7069  0.4291  0.9874  1.0000  0.0000 -1.0000 -0.2805 -1.0000 -0.1799 -0.1799  0
 0.5270  0.4291  1.1672  1.0000  1.0000 -1.0000 -0.2112  1.0000  0.3028  0.3028  0
weights =  0.8297923 0.7318663 0.864451  old  0.706878 0.6499282 0.6886811
Err =  1.430988

  w1      w2      b       x1      x2     -1        act.   target  delta    dw1
 0.8298  0.7319  0.8645  0.0000  0.0000 -1.0000 -0.8645 -1.0000 -0.0339  0.0000  0
 0.8298  0.7319  0.8983  0.0000  1.0000 -1.0000 -0.1665 -1.0000 -0.2084  0.0000 -0
 0.8298  0.5235  1.1067  1.0000  0.0000 -1.0000 -0.2769 -1.0000 -0.1808 -0.1808  0
 0.6490  0.5235  1.2875  1.0000  1.0000 -1.0000 -0.1150  1.0000  0.2787  0.2787  0
weights =  0.9277692 0.8022292 1.008743  old  0.8297923 0.7318663 0.864451
Err =  1.239578

  w1      w2      b       x1      x2     -1        act.   target  delta    dw1
 0.9278  0.8022  1.0087  0.0000  0.0000 -1.0000 -1.0087 -1.0000  0.0022  0.0000  0
 0.9278  0.8022  1.0066  0.0000  1.0000 -1.0000 -0.2043 -1.0000 -0.1989  0.0000 -0
 0.9278  0.6033  1.2055  1.0000  0.0000 -1.0000 -0.2777 -1.0000 -0.1806 -0.1806  0
 0.7472  0.6033  1.3860  1.0000  1.0000 -1.0000 -0.0355  1.0000  0.2589  0.2589  0
weights =  1.006081 0.8621967 1.127163  old  0.9277692 0.8022292 1.008743
Err =  1.113613

  w1      w2      b       x1      x2     -1        act.   target  delta    dw1
 1.0061  0.8622  1.1272  0.0000  0.0000 -1.0000 -1.1272 -1.0000  0.0318  0.0000  0
 1.0061  0.8622  1.0954  0.0000  1.0000 -1.0000 -0.2332 -1.0000 -0.1917  0.0000 -0
 1.0061  0.6705  1.2871  1.0000  0.0000 -1.0000 -0.2810 -1.0000 -0.1798 -0.1798  0
 0.8263  0.6705  1.4668  1.0000  1.0000 -1.0000  0.0300  1.0000  0.2425  0.2425  0
weights =  1.068833 0.9129927 1.224327  old  1.006081 0.8621967 1.127163
Err =  1.031035

  w1      w2      b       x1      x2     -1        act.   target  delta    dw1
 1.0688  0.9130  1.2243  0.0000  0.0000 -1.0000 -1.2243 -1.0000  0.0561  0.0000  0
 1.0688  0.9130  1.1682  0.0000  1.0000 -1.0000 -0.2553 -1.0000 -0.1862  0.0000 -0
 1.0688  0.7268  1.3544  1.0000  0.0000 -1.0000 -0.2856 -1.0000 -0.1786 -0.1786  0
 0.8902  0.7268  1.5330  1.0000  1.0000 -1.0000  0.0840  1.0000  0.2290  0.2290  0
weights =  1.119231 0.9558044 1.304034  old  1.068833 0.9129927 1.224327
Err =  0.977192

  w1      w2      b       x1      x2     -1        act.   target  delta    dw1
 1.1192  0.9558  1.3040  0.0000  0.0000 -1.0000 -1.3040 -1.0000  0.0760  0.0000  0
 1.1192  0.9558  1.2280  0.0000  1.0000 -1.0000 -0.2722 -1.0000 -0.1819  0.0000 -0
 1.1192  0.7739  1.4100  1.0000  0.0000 -1.0000 -0.2907 -1.0000 -0.1773 -0.1773  0
 0.9419  0.7739  1.5873  1.0000  1.0000 -1.0000  0.1285  1.0000  0.2179  0.2179  0
weights =  1.159793 0.9917371 1.369408  old  1.119231 0.9558044 1.304034
Err =  0.9423397
```

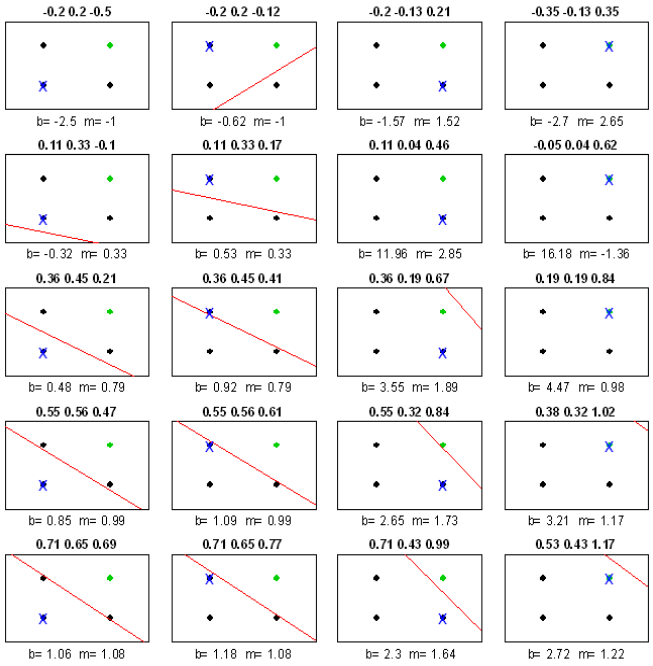


Figure 7. First 5 passes

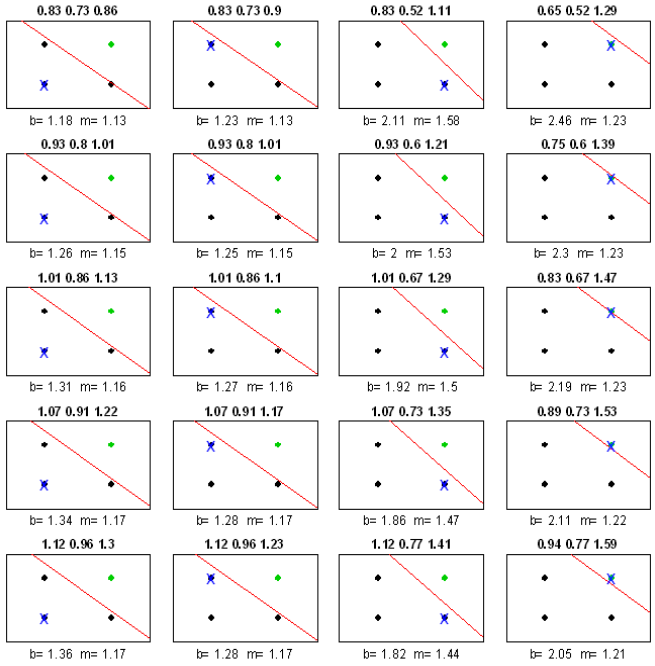


Figure 8. Second 5 passes

```
  w1      w2      b       x1      x2     -1        act.   target  delta    dw1
 1.1598  0.9917  1.3694  0.0000  0.0000 -1.0000 -1.3694 -1.0000  0.0924  0.0000  0
 1.1598  0.9917  1.2771  0.0000  1.0000 -1.0000 -0.2853 -1.0000 -0.1787  0.0000 -0
 1.1598  0.8131  1.4557  1.0000  0.0000 -1.0000 -0.2959 -1.0000 -0.1760 -0.1760  0
 0.9838  0.8131  1.6317  1.0000  1.0000 -1.0000  0.1651  1.0000  0.2087  0.2087  0
weights =  1.192501 1.021792 1.423018  old  1.159793 0.9917371 1.369408
Err =  0.9199996

Press Enter...
  w1      w2      b       x1      x2     -1        act.   target  delta    dw1
 1.1925  1.0218  1.4230  0.0000  0.0000 -1.0000 -1.4230 -1.0000  0.1058  0.0000  0
 1.1925  1.0218  1.3173  0.0000  1.0000 -1.0000 -0.2955 -1.0000 -0.1761  0.0000 -0
 1.1925  0.8457  1.4934  1.0000  0.0000 -1.0000 -0.3009 -1.0000 -0.1748 -0.1748  0
 1.0177  0.8457  1.6682  1.0000  1.0000 -1.0000  0.1952  1.0000  0.2012  0.2012  0
weights =  1.218922 1.046857 1.466975  old  1.192501 1.021792 1.423018
Err =  0.905868

Press Enter...
  w1      w2      b       x1      x2     -1        act.   target  delta    dw1
 1.2189  1.0469  1.4670  0.0000  0.0000 -1.0000 -1.4670 -1.0000  0.1167  0.0000  0
 1.2189  1.0469  1.3502  0.0000  1.0000 -1.0000 -0.3034 -1.0000 -0.1742  0.0000 -0
 1.2189  0.8727  1.5244  1.0000  0.0000 -1.0000 -0.3055 -1.0000 -0.1736 -0.1736  0
 1.0453  0.8727  1.6980  1.0000  1.0000 -1.0000  0.2200  1.0000  0.1950  0.1950  0
weights =  1.240296 1.067708 1.503013  old  1.218922 1.046857 1.466975
Err =  0.8970905
```

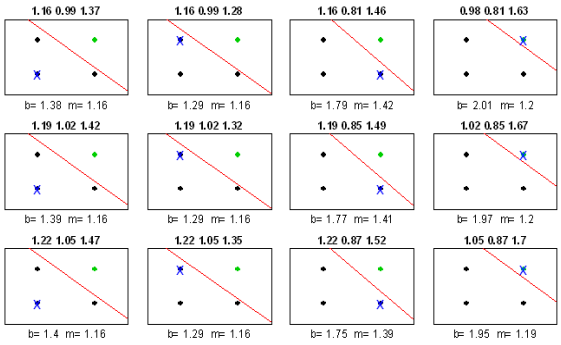


Figure 9. Final passes

We see that we have a correct pass through the training set (all the activations *a* and targets have the same sign). Further iterations may reduce the error.

The type of problems described above are problems in which we can do a ***linear separation***. A simple example in which this will **not** work is the logical XOR function

x_1	x_2	t
0	0	0
0	1	1
1	0	1
1	1	0

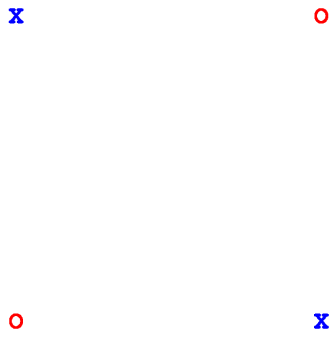


Figure 10. Logical XOR

It is not possible to separate the classes with a straight line

In order ***to solve problems that are not linearly separable*** we need to introduce a more complicated structure for the neural network. This will come in the form of one (or more) additional layers of nodes. Because they are not visible as input or output layers, they are referred to as *hidden layers*.

A network with one hidden layer can be pictured as:

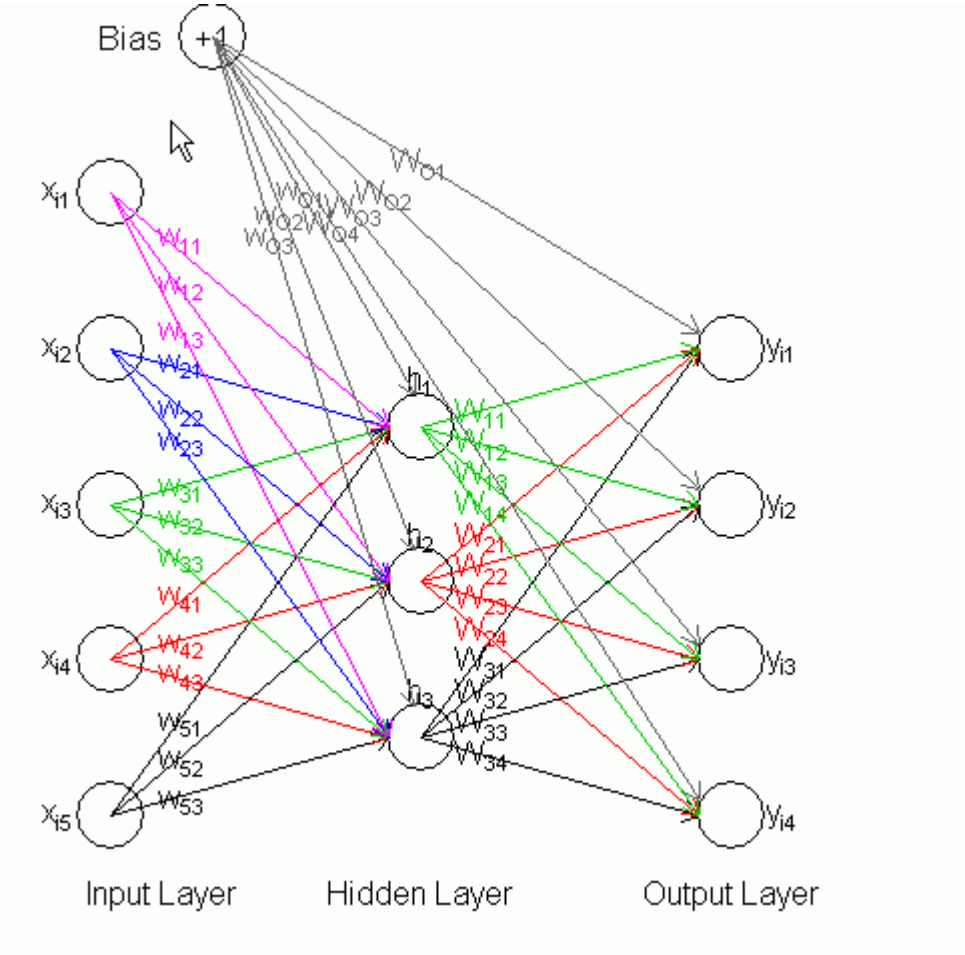
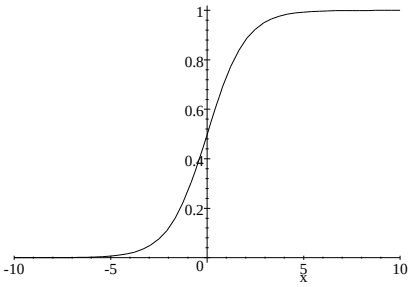


Figure 11. Feed-forward Neural Network

As before, we take x_i as inputs and w_{ij} as weights.

The hidden nodes will have activation functions (as do the output nodes) but, in order to allow us to update the weights, we will need an activation function that behaves in a manner similar to a step function but is continuous (so that we can differentiate it).

A *sigmoidal function* such as



or

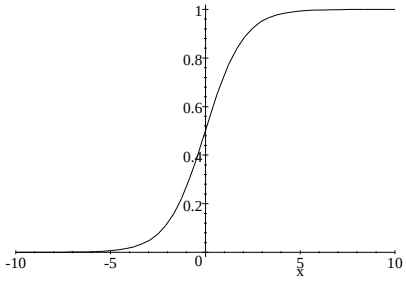


Figure 12. $f(x) = e^x/(1 + e^x)$

Figure 13. $f(x) = \tanh x$

is typically used (recall the Logistic Regression).

In our network above, we can take a common *hidden* node activation function (f_h) and a common *output* activation function (f_o) to be

$$1/(1 + e^{-x})$$

(Note: We could have different functions at different nodes).

We will 'present' the N cases to the neural network where, for case i, we have

$$\mathbf{X}_i^T = \begin{bmatrix} 1 & x_{i1} & x_{i2} & \cdots & x_{ip} \end{bmatrix}$$

(Note that the 1 is for the bias unit and is similar to the 1 in regression that corresponds to the presence of an intercept term).

The weights from p inputs (and bias) to the hidden node m (H_m) are

$$\mathbf{w}_m^T = \begin{bmatrix} w_{0m} & w_{1m} & w_{2m} & \ldots & w_{pm} \end{bmatrix}$$

so the input to H_m for case i *is*

$$\begin{aligned} z_m &= w_{0m}1 + w_{1m}x_{i1} + w_{2m}x_{i2} + \ldots + w_{pm}x_{ip} \\ &= w_{0m} + \sum_{i=1}^{p} w_{im}x_i \\ &= \mathbf{w}_m^T\mathbf{X}_i \end{aligned}$$

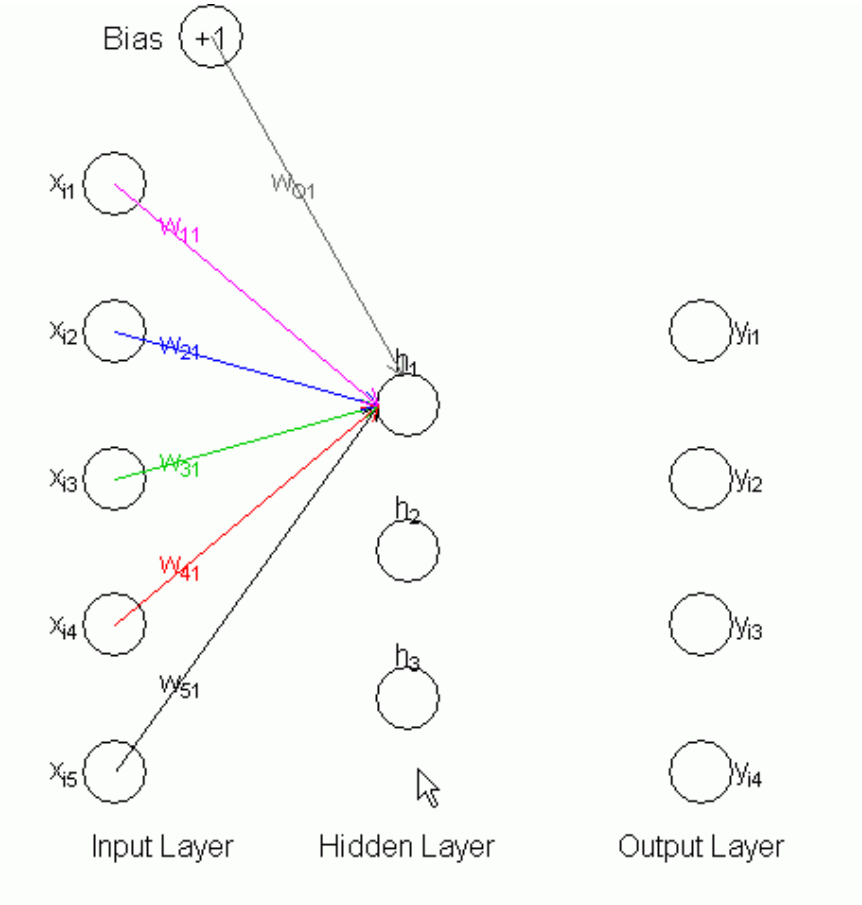


Figure 14 Inputs to hidden node

This input is 'evaluated' by the activation function f_H and the output from a hidden node due to case i from H_m is

$$T_{im} = f_H(w_{0m} + w_{1m}x_{i1} + w_{2m}x_{i2} + \ldots + w_{pm}x_{ip})$$
$$= f_H\left(w_{0m} + \sum_{i=1}^{p} w_{im}x_i\right) = f_H(\mathbf{w}_m^T\mathbf{X}_i)$$

The weights from all (M) hidden nodes to the output node O_k are given by

$$\mathbf{W}_k^T = \left[W_{0k}\ W_{1k}\ W_{2k}\ \ldots\ W_{Mk}\right]$$

so that the input to O_k is for case i is

$$Z_k = W_{0k} + \sum_{m=1}^{M} W_{mk} f_H\left(w_{0m} + \sum_{i=1}^{p} w_{im}x_i\right)$$
$$= W_{0k} + \sum_{m=1}^{M} W_{mk} T_{im}$$

or

$$Z_k = W_{0k} + W_{1k}T_{i1} + W_{2k}T_{i2} + \ldots.W_{Mk}T_{iM}$$
$$= \mathbf{W}_k^T\ \mathbf{T}_i$$

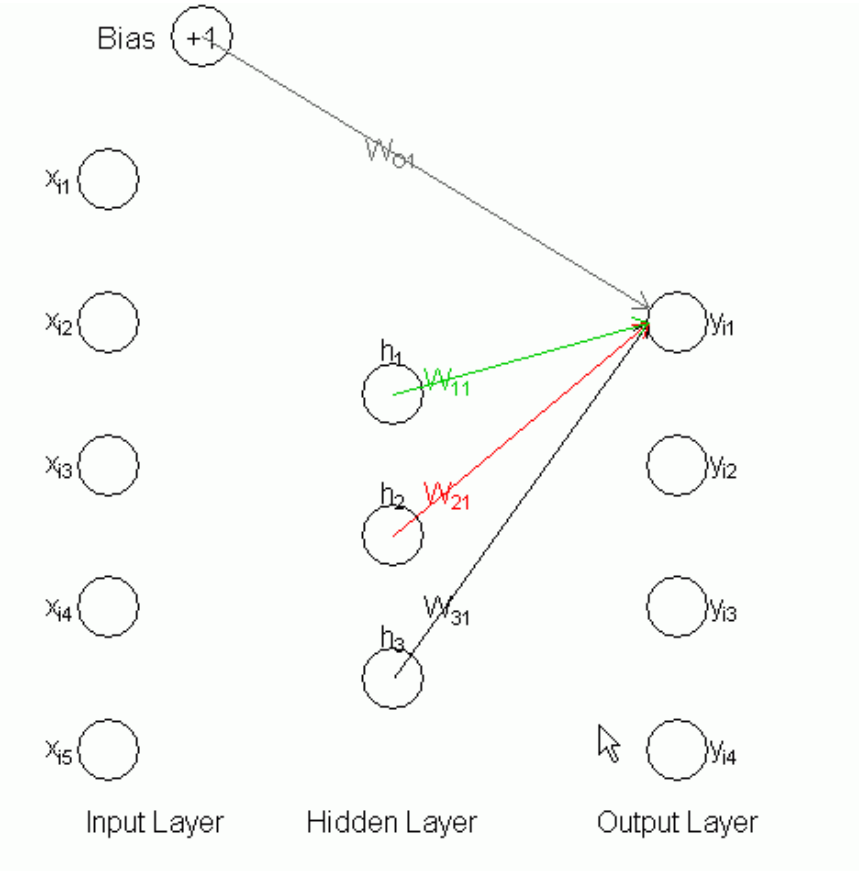


Figure 15. Inputs to output node

The output from O_k for case i is

$$f_O(W_{0k} + W_{1k}T_{i1} + W_{2k}T_{i2} + ...W_{Mk}T_{iM}) = f_O\left(\mathbf{W}_k^T\ \mathbf{T}_i\right)$$

To find the "error" at O_k we take the difference between this output and the "true" value at the node - t_{ik} - and square it

$$E_{ik} = \left(t_{ik} - f_O\left(\mathbf{W}_k^T\ \mathbf{T}_i\right)\right)^2$$

The error from all output nodes for case i is

$$E_i = \sum_{k=1}^{K}\left(t_{ik} - f_O\left(\mathbf{W}_k^T\ \mathbf{T}_i\right)\right)^2$$

so the total error of the network for all N input values is

$$E = \sum_{i=1}^{N} E_i = \sum_{i=1}^{N}\sum_{k=1}^{K}\left(t_{ik} - f_O\left(\mathbf{W}_k^T\ \mathbf{T}_i\right)\right)^2$$

We wish to choose weights which minimize E so we start by differentiating E with respect to the weight from H_m to O_k :

$$\begin{aligned}
\frac{\partial E_i}{\partial W_{mk}} &= \frac{\partial}{\partial W_{mk}}\sum_{k=1}^{K}\left(t_{ik} - f_O\left(\mathbf{W}_k^T\ \mathbf{T}_i\right)\right)^2 \\
&= \frac{\partial}{\partial W_{mk}}\sum_{k=1}^{K}(t_{ik} - f_O(W_{0k} + W_{1k}T_{i1} + W_{2k}T_{i2} + \cdots + W_{Mk}T_{iM}))^2 \\
&= -2\left(t_{ik} - f_O\left(\mathbf{W}_k^T\ \mathbf{T}_i\right)\right)\frac{\partial f_O(W_{0k} + W_{1k}T_{i1} + W_{2k}T_{i2} + \cdots + W_{Mk}T_{iM})}{\partial W_{mk}} \\
&= -2\left(t_{ik} - f_O\left(\mathbf{W}_k^T\ \mathbf{T}_i\right)\right)f_O'\left(\mathbf{W}_k^T\ \mathbf{T}_i\right)T_{im}
\end{aligned}$$

and also differentiate with respect to the weight from each input node I_l to H_m :

$$\begin{aligned}\frac{\partial E_i}{\partial w_{lm}} &= \frac{\partial}{\partial w_{lm}} \sum_{k=1}^{K} \left(t_{ik} - f_O\left(\mathbf{W}_k^T\ \mathbf{T}_i\right)\right)^2 \\ &= -\sum_{k=1}^{K} 2\left(t_{ik} - f_O\left(\mathbf{W}_k^T\ \mathbf{T}_i\right)\right) \frac{\partial f_O\left(\mathbf{W}_k^T\ \mathbf{T}_i\right)}{\partial w_{lm}} \\ &= -\sum_{k=1}^{K} 2\left(t_{ik} - f_O\left(\mathbf{W}_k^T\ \mathbf{T}_i\right)\right) f_O'\left(\mathbf{W}_k^T\ \mathbf{T}_i\right) \frac{\partial (W_{0k} + W_{1k}T_{i1} + W_{2k}T_{i2} + \cdots + W_{Mk}T_{iM})}{\partial w_{lm}}.\end{aligned}$$

Note that

$$\begin{aligned}W_{0k} + W_{1k}T_{i1} + W_{2k}T_{i2} + ...W_{Mk}T_{iM} = {} & W_{0k} + W_{1k}\, f_H(w_{01} + w_{11}x_{i1} + w_{21}x_{i2} + \cdots + w_{p1}x_{ip}) \\ & + W_{2k} f_H(w_{02} + w_{12}x_{i1} + w_{22}x_{i2} + \cdots + w_{p2}x_{ip}) \\ & + ... + W_{Mk} f_H(w_{0M} + w_{1M}x_{i1} + w_{2M}x_{i2} + \cdots + w_{pM}x_{ip})\end{aligned}$$

so

$$\frac{\partial E}{\partial w_{lm}} = -\sum_{k=1}^{K} 2\left(t_{ik} - f_O\left(\mathbf{W}_k^T\ \mathbf{T}_i\right)\right) f_O'\left(\mathbf{W}_k^T\ \mathbf{T}_i\right) W_{mk} f_H'(\mathbf{w}_m^T \mathbf{X}_i) x_{il}$$

With these derivatives, it is possible to do a *gradient descent update* (with a learning rate α_s) so that at step $(s+1)$:

$$W_{mk}^{(s+1)} = W_{mk}^{(s)} - \alpha_s \sum_{i=1}^{N} \frac{\partial E_i}{\partial w_{lm}^{(s)}}$$

$$\text{and } w_{lm}^{(s+1)} = w_{lm}^{(s)} - \alpha_s \sum_{i=1}^{N} \frac{\partial E}{\partial w_{lm}^{(s)}}$$

This process is referred to as *back-propagation*.

In order to illustrate the neural network consider the following:

Example: Neural Network on Flea Beetles

```
source(paste(code.dir, "ReadFleas.r", sep="/"))
```

We know the correct classes for the flea beetles (given by `flea.species`). We will use that as our target values for the neural net.

`nnet` expects the target information to be given in the form of a `class.ind` (that is a matrix with the the columns indicating classes for the cases - the transpose is used to make the dispaly more compact).

```
t(class.ind(flea.species))
```

```
   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13] [,14]
C     1    1    1    1    1    1    1    1    1     1     1     1     1     1
Hk    0    0    0    0    0    0    0    0    0     0     0     0     0     0
Hp    0    0    0    0    0    0    0    0    0     0     0     0     0     0
   [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22] [,23] [,24] [,25] [,26]
C      1     1     1     1     1     1     1     0     0     0     0     0
Hk     0     0     0     0     0     0     0     0     0     0     0     0
Hp     0     0     0     0     0     0     0     1     1     1     1     1
   [,27] [,28] [,29] [,30] [,31] [,32] [,33] [,34] [,35] [,36] [,37] [,38]
C      0     0     0     0     0     0     0     0     0     0     0     0
Hk     0     0     0     0     0     0     0     0     0     0     0     0
Hp     1     1     1     1     1     1     1     1     1     1     1     1
   [,39] [,40] [,41] [,42] [,43] [,44] [,45] [,46] [,47] [,48] [,49] [,50]
C      0     0     0     0     0     0     0     0     0     0     0     0
Hk     0     0     0     0     0     1     1     1     1     1     1     1
Hp     1     1     1     1     1     0     0     0     0     0     0     0
   [,51] [,52] [,53] [,54] [,55] [,56] [,57] [,58] [,59] [,60] [,61] [,62]
C      0     0     0     0     0     0     0     0     0     0     0     0
Hk     1     1     1     1     1     1     1     1     1     1     1     1
Hp     0     0     0     0     0     0     0     0     0     0     0     0
   [,63] [,64] [,65] [,66] [,67] [,68] [,69] [,70] [,71] [,72] [,73] [,74]
C      0     0     0     0     0     0     0     0     0     0     0     0
Hk     1     1     1     1     1     1     1     1     1     1     1     1
Hp     0     0     0     0     0     0     0     0     0     0     0     0
```

We can find how many of each species are present:

```
apply(class.ind(flea.species),2,sum)
C Hk Hp
21 31 22
```

We try for the simplest possible solution (i.e. the **smallest hidden layer**) consistent with a good result (low misclassification). The `size` parameter indicates the number of nodes in the hidden layer. If we omit the hidden layer `size=0` we must indicate `skip=T`. (See `?nnet`.)

In order to enable us to look at how well the classifier works we will divide the data into a training and a test set using the functions from previous lectures.

```
"%w/o%" <- function(x,y) x[!x %in% y]
```

```
#=======================================
# Set the indices for the training/test sets
#=======================================
get.train <- function (data.sz, train.sz) {
      # Take subsets of data for training/test samples
      # Return the indices
   train.ind <- sample(data.sz, train.sz)
   test.ind <- (1:data.sz) %w/o% train.ind
   list(train=train.ind, test=test.ind)
}
Train.sz <- 52
(tt.ind <- get.train(dim(df.flea)[1], Train.sz))

$train
 [1] 52 61 13 12  7  8 33 40 16  4 22 60 58 23  6  9 66 21 48 65 38  5  3 41 27 42
[28] 71 32 43 74 50 51  1 34 47 69 45 56 28 10 67 24 37 68 18  2 62 54 64 57 19
$test
 [1] 11 14 15 17 20 25 26 29 30 31 35 36 39 44 46 49 53 55 59 63 70 72
```

(For reproducibility we will use these values rather than the random ones.)

```
tt.ind$train <- c(52,61,13,12,7,8,33,40,16,4,22,60,58,23,6,9,66,21,48,65,38,
                  5,3,41,27,42,73,71,32,43,74,50,51,1,34,47,69,45,56,28,10,
                  67,24,37,68,18,2,62,54,64,57,19)
tt.ind$test <-
c(11,14,15,17,20,25,26,29,30,31,35,36,39,44,46,49,53,55,59,63,70,72)
apply(class.ind(flea.species[tt.ind$train]),2,sum)

 C Hk Hp
16 22 14

apply(class.ind(flea.species[tt.ind$test]),2,sum)

C Hk Hp
5  9  8
```

We see that we have 16/21 **C**, 22/31 **Hk**, and 14/22 **Hp** in the training set and 5/21 **C**, 9/31 **Hk**, and 8/22 **Hp** in the test set.

NOTE: There is always a danger that the classes may be represented in an inappropriate fashion in the training set. For example, if there are no representatives of class **C** in the training set then the neural network will never classify anything as class **C**.

We will start with no hidden layer (`size=0`), connections from the input to the output layer (`skip=T`), range for initial random weights [–0.1, 0.1] (`range=0.1`), weight decay parameter 0.0005 (`decay=5e-4`), and a maximum of 500 iterations (`maxit=500`):

```
flea.nn <- nnet(df.flea[tt.ind$train,], class.ind(flea.species[tt.ind$train]),
size=0, skip=T, rang=0.1, decay=5e-4, maxit=500)

# weights:  21
initial  value 38.003178
```

```
iter  10 value 25.673878
iter  20 value 17.542510
iter  30 value 4.471108
iter  40 value 3.329062
iter  50 value 3.298340
iter  60 value 3.212452
iter  70 value 3.146040
iter  80 value 3.129967
iter  90 value 3.114434
iter 100 value 3.105363
iter 110 value 3.098298
iter 120 value 3.095578
iter 130 value 3.095131
iter 140 value 3.094971
iter 150 value 3.094963
final  value 3.094956
converged
```

The neural net has 21 weights (each of the 6 inputs is connected to each output as is the bias, for a total of 7×3).

```
summary(flea.nn)

a 6-0-3 network with 21 weights
options were - skip-layer connections  decay=5e-04
 b->o1 i1->o1 i2->o1 i3->o1 i4->o1 i5->o1 i6->o1
 -3.16   0.65  -0.03  -9.88   2.80  -3.83   0.53
 b->o2 i1->o2 i2->o2 i3->o2 i4->o2 i5->o2 i6->o2
  0.02   0.79  -0.26   0.19  -0.49   0.57  -0.75
 b->o3 i1->o3 i2->o3 i3->o3 i4->o3 i5->o3 i6->o3
  0.03  -0.57   0.24   0.24   0.20  -0.14   0.21
```

The above output shows the connections and the corresponding weights.

We can investigate the results of this (and many other functions) by use of expressions of the form `flea.nn[i]`.

```
flea.nn[11]
$wts
 [1] -3.16234631  0.64955583 -0.02654692 -9.87994802  2.79725402 -3.83345049
 [7]  0.52646205  0.02468363  0.79217231 -0.25674016  0.19259577 -0.48754806
[13]  0.57420624 -0.75152200  0.02583289 -0.57245962  0.23516685  0.23850064
[19]  0.19531903 -0.14429595  0.21283729
```

(We could also use `flea.nn$wts`.)

```
t(flea.nn$fitted.values)
   52        61           13           12            7            8       33
C   0 1.0000000 1.000000e+00 1.0000000000 8.993480e-01 9.999995e-01 0.000000
Hk  1 0.9999208 8.611791e-06 0.0000000000 6.284825e-07 0.000000e+00 0.000000
Hp  0 0.0000000 0.000000e+00 0.0006301502 3.469448e-07 4.432988e-07 0.999971
          40           16            4        22           60           58
C  0.0000000 1.086797e-05 9.876069e-01 0.0000000 0.06561292 1.087303e-05
Hk 0.0000000 2.812531e-04 0.000000e+00 0.0000000 0.99999945 1.000000e+00
Hp 0.9975123 0.000000e+00 2.131585e-05 0.9985126 0.00000000 0.000000e+00
            23           6           9        66           21           48
C  0.0005646545 1.000000000 0.944343721 0.0000000 0.000000e+00 0.0001635885
Hk 0.0000000000 0.000000000 0.000000000 0.9999484 0.000000e+00 1.0000000000
Hp 0.9974711616 0.001730916 0.003764649 0.0000000 2.489253e-05 0.0000000000
```

```
          65           38            5           3 41         27        42 73
C  0.06260702 5.306533e-05 1.0000000000 0.999994919  0 0.07486592 0.0000000  0
Hk 0.99442958 0.000000e+00 0.0000000000 0.003979733  0 0.00000000 0.0000000  1
Hp 0.00000000 9.999784e-01 0.0003857919 0.000000000  1 0.99994208 0.9983447  0
   71 32       43        74           50           51           1           34
C   0  0 0.000000 0.0000000 1.540079e-06 1.462929e-06 9.999757e-01 9.045183e-06
Hk  1  0 0.000000 0.9916207 1.000000e+00 1.000000e+00 3.837811e-07 0.000000e+00
Hp  0  1 0.999999 0.0000000 0.000000e+00 0.000000e+00 0.000000e+00 1.000000e+00
   47 69 45 56        28          10 67         24        37 68          18
C   0  0  0  0 0.0000000 1.000000e+00  0 0.03017668 0.0000000  0 1.000000000
Hk  1  1  1  1 0.0000000 0.000000e+00  1 0.00000000 0.0000000  1 0.000000000
Hp  0  0  0  0 0.9999996 9.697477e-05  0 0.99782310 0.9991032  0 0.001895541
             2 62 54           64        57          19
C  1.000000e+00  0  0 1.095646e-05 0.0000000 1.000000000
Hk 0.000000e+00  1  1 1.000000e+00 0.9999972 0.008531558
Hp 1.746592e-06  0  0 0.000000e+00 0.0000000 0.000000000
```

(We get the same information from `t(predict(flea.nn, df.flea[tt.ind$train,]))`.)

The fitted class for each case is given by the **largest value** for the case. To determine which class has been predicted for which case, we can use:

```
max.col(flea.nn$fitted.values)
 [1] 2 2 1 1 1 1 3 3 1 1 3 2 2 3 1 1 2 1 2 2 3 1 1 3 3 3 2 2 3 3 2 2 2 1 3 2 2 2
[39] 2 3 1 2 3 3 2 1 1 2 2 2 2 1
```

It is difficult to see what the classes are in this form so a useful method for testing the accuracy of classification is the confusion matrix.

```
confusion.expand(max.col(flea.nn$fitted.values), flea.species[tt.ind$train])
         true
object    C    Hk   Hp   | Row Sum
  1       14   1    0    | 15
  2       1    21   0    | 22
  3       1    0    14   | 15
  ------- ---- ---- ---- | ----
  Col Sum 16   22   14   | 52
attr(,"error")
[1] NaN
attr(,"mismatch")
[1] 1
```

(Watch out! The confusion matrix displays the classes in **alphabetical** order - not the given order!)

Note the use of `max.col` in the confusion matrix. It does as the name suggests - determines which column has the maximum value (note that ties are broken at random).

Here 14 of class **C** have been classified as class 1, 1 as class 2, and 1 as class 3, 21 of the **Hk** were classified as being in class 2, and 1 as class 1, while all the **Hp** are classified as class 3. Thus we have an error rate of 3/52 or 5.8%. The problem is that we are obtaining this error rate by using the same data that went into creating the model. Because we have reserved some data for testing, we can use it to get a more accurate representation of the error rate. We first need to determine how the test data can be classified. It is typical of classifiers that

they have a **prediction** method so that we can see how the cases can be classified.

```
predict(flea.nn, df.flea[tt.ind$test,])
```

```
               C          Hk           Hp
11 0.3513851292 1.61017e-05 0.000000e+00
14 1.0000000000 0.00000e+00 4.722636e-07
15 1.0000000000 0.00000e+00 4.124901e-06
17 0.0433338860 0.00000e+00 1.977635e-06
20 1.0000000000 0.00000e+00 1.260786e-06
25 0.9999701704 0.00000e+00 9.999987e-01
26 0.0000000000 0.00000e+00 9.999587e-01
29 0.0000000000 0.00000e+00 9.999708e-01
30 0.0000000000 0.00000e+00 1.000000e+00
31 0.9997923461 0.00000e+00 9.999988e-01
35 0.0000000000 0.00000e+00 9.976731e-01
36 0.0000000000 0.00000e+00 9.999655e-01
39 0.0626155629 0.00000e+00 9.999328e-01
44 0.0000000000 1.00000e+00 0.000000e+00
46 0.9998812554 1.00000e+00 0.000000e+00
49 0.0000000000 1.00000e+00 0.000000e+00
53 0.0004629659 1.00000e+00 0.000000e+00
55 0.0000000000 1.00000e+00 0.000000e+00
59 0.0000922833 1.00000e+00 0.000000e+00
63 0.0000000000 1.00000e+00 0.000000e+00
70 0.0000000000 9.99988e-01 0.000000e+00
72 0.0000000000 1.00000e+00 0.000000e+00
```

```
confusion.expand(max.col(predict(flea.nn,
df.flea[tt.ind$test,])),flea.species[tt.ind$test])
```

```
         true
object    C    Hk   Hp   | Row Sum
  1       5    0    0    | 5
  2       0    9    0    | 9
  3       0    0    8    | 8
  ------- ---- ---- ---- | ----
  Col Sum 5    9    8    | 22
attr(,"error")
[1] NaN
attr(,"mismatch")
[1] 1
```

Here we have a perfect classification, but remember we are dealing with a small data set that we have seen can be separated quite well.

We might also try a neural net that has one node (`size=1`) in the hidden layer:

```
flea.nn <- nnet(df.flea[tt.ind$train,], class.ind(flea.species[tt.ind$train]),
                size=1, rang=0.1, decay=5e-4, maxit=500)
```

```
# weights:  13
initial  value 39.061500
iter  10 value 34.003834
iter  20 value 27.969509
iter  30 value 18.598327
iter  40 value 17.403684
iter  50 value 17.302171
iter  60 value 16.063792
```

```
iter  70 value 13.520698
iter  80 value 11.554444
iter  90 value 11.269187
iter 100 value 11.108513
iter 110 value 11.006646
iter 120 value 11.005451
iter 130 value 11.001329
final  value 11.000691
converged
```

(Notice that we have fewer weights.)

```
summary(flea.nn)
a 6-1-3 network with 13 weights
options were - decay=5e-04
 b->h1 i1->h1 i2->h1 i3->h1 i4->h1 i5->h1 i6->h1
  3.24   0.07  -0.10   0.03  -0.06   0.32  -0.01
 b->o1 h1->o1
 -0.02  -1.75
 b->o2 h1->o2
 -7.13  14.50
 b->o3 h1->o3
  2.39 -34.26
```

This describes the network in terms of what nodes it contains, how they are connected, and the weights.

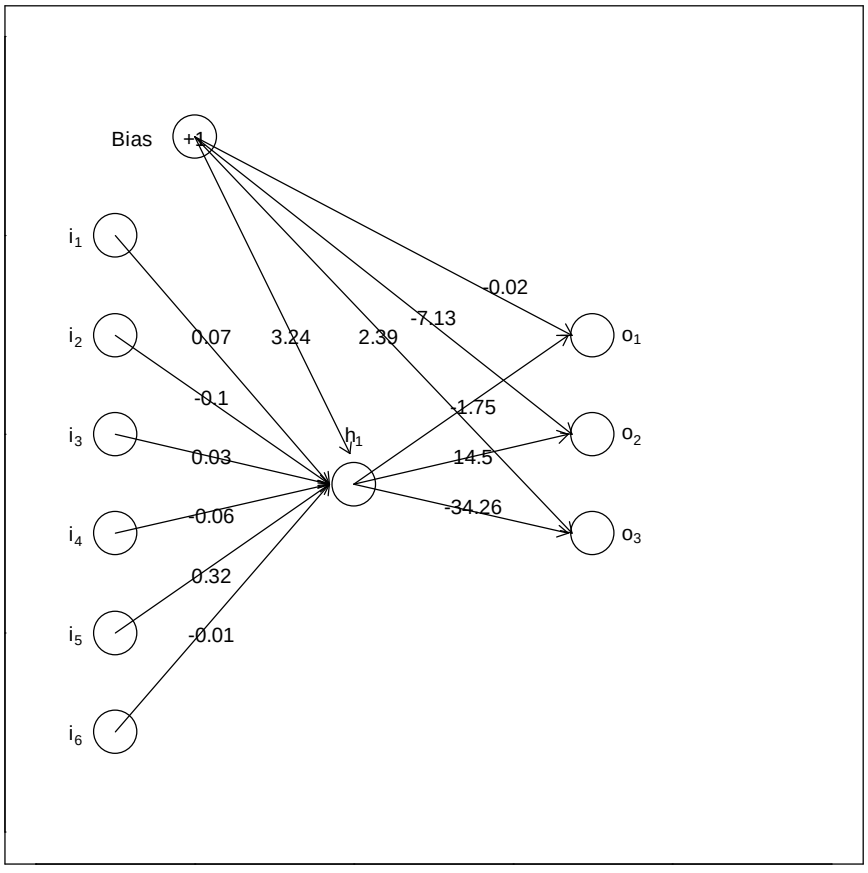


Figure 16. Neural net for flea beetles

```
confusion.expand(max.col(flea.nn$fitted.values), flea.species[tt.ind$train])
         true
object    C    Hk   Hp   | Row Sum
  1       16   0    0    | 16
  2       0    22   0    | 22
  3       0    0    14   | 14
  ------- ---- ---- ---- | ----
  Col Sum 16   22   14   | 52
```

Here all the cases are correctly classified.

Suppose we look at how this neural network will predict the class by looking at the first case:

```
df.flea[1,]
  tars1 tars2 head aede1 aede2 aede3
1   191   131   53   150    15   104
```

To see how we get our result, look at the input vector (the `1` is the bias component)

```
(i.case.1 <- c(1, as.matrix(df.flea[1,])))
[1]   1 191 131  53 150  15 104
```

and the weights to the hidden node:

```
(w.i.h <- flea.nn$wts[1:7])
[1]  3.239477309  0.069190234 -0.098314207  0.026150991 -0.064498570
[6]  0.318486158 -0.009023526
```

The 'signal' to the hidden node is thus:

```
(to.h <- sum(i.case.1*w.i.h))
[1] -0.8742865
```

which is then passed through the sigmoidal function of the hidden layer to give the hidden layer output:

```
(from.h <- 1/(1+exp(-to.h)))
[1] 0.2943632
```

The input to the output nodes comes from the bias and the hidden node i.e. `c(1, from.h)` and the weights are:

```
(w.h.o1 <- flea.nn$wts[8:9])
[1] -0.01884662 -1.74534267
(w.h.o2 <- flea.nn$wts[10:11])
[1] -7.132326 14.497100
(w.h.o3 <- flea.nn$wts[12:13])
[1]   2.385936 -34.257359
```

so the inputs to the output nodes are:

```
(to.o1 <- sum(c(1, from.h)*w.h.o1))
[1] -0.5326112
(to.o2 <- sum(c(1, from.h)*w.h.o2))
[1] -2.864914
(to.o3 <- sum(c(1, from.h)*w.h.o3))
[1] -7.698168
```

and the outputs are:

```
from.o1 <- 1/(1+exp(-to.o1))
from.o2 <- 1/(1+exp(-to.o2))
from.o3 <- 1/(1+exp(-to.o3))
cat(from.o1, from.o2, from.o3, "\n")
0.3699081 0.05391547 0.0004534519
```

Using

```
predict(flea.nn, df.flea[1,])
          C          Hk          Hp
1 0.3699081 0.05391547 0.0004534519
```

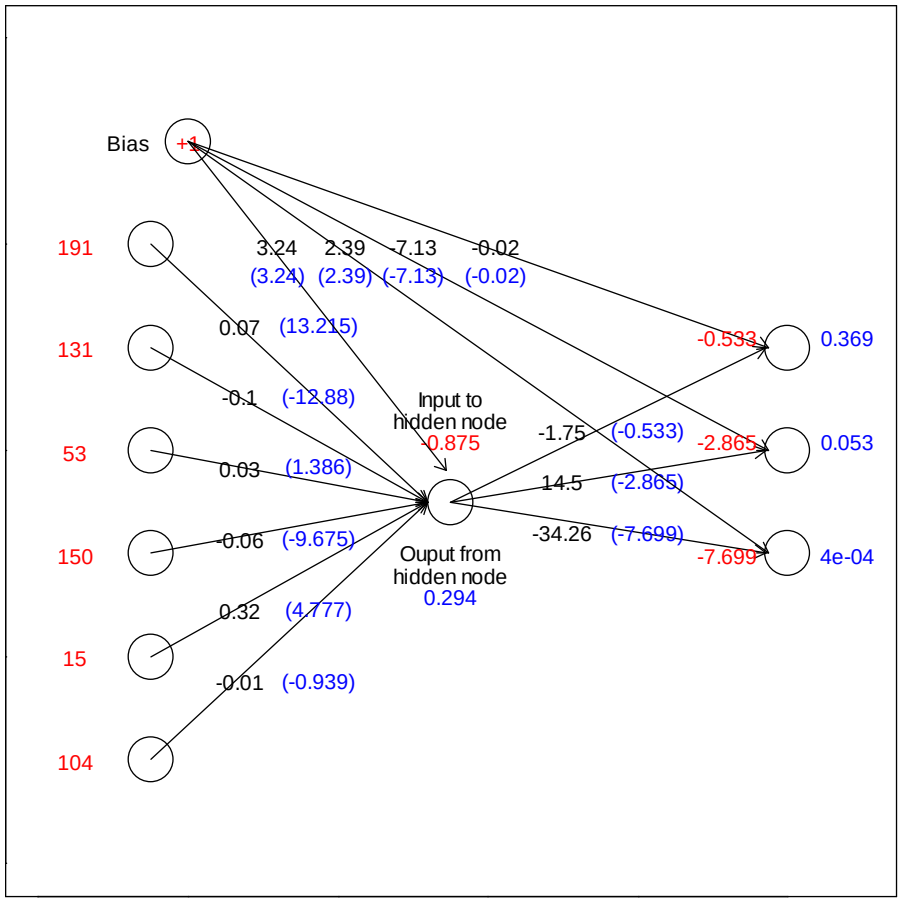


Figure 17. Results for first case

In the Figure 17 above, the black numbers represent the weights, the blue numbers represent the output from a neuron and the red numbers represent the inputs.

Now look at the test cases.

```
confusion.expand(max.col(predict(flea.nn,
df.flea[tt.ind$test,])),flea.species[tt.ind$test])
          true
object     C    Hk    Hp    | Row Sum
  1        5    0     0     | 5
  2        0    9     0     | 9
  3        0    0     8     | 8
  ------- ---- ---- ---- | ----
  Col Sum 5    9     8     | 22
attr(,"error")
[1] NaN
attr(,"mismatch")
[1] 1
```

It would appear that this neural network is a good classifier.

Suppose that we rerun with the same values (omitting some output):

```
flea.nn <- nnet(df.flea[tt.ind$train,], class.ind(flea.species[tt.ind$train]),
size=1, rang=0.1, decay=5e-4, maxit=500)
confusion(max.col(flea.nn$fitted.values), flea.species[tt.ind$train])
```

```
# weights:  13
initial  value 38.378513
iter  10 value 34.004917
...
iter 160 value 18.370829
final  value 18.370750
converged
      true
object  C Hk Hp
     1  2  0  0
     2 14 22  0
     3  0  0 14
attr(,"error")
[1] NaN
attr(,"mismatch")
[1] 1
```

Different final value and # of iterations.

```
# weights:  13
initial  value 39.608202

...

final  value 18.549964
converged
      true
object  C Hk Hp
     1  1  0  0
     2 15 22  0
     3  0  0 14
attr(,"error")
[1] NaN
attr(,"mismatch")
[1] 1
```

Different final value and # of iterations.

```
# weights:  13
initial  value 38.641683
...
iter  90 value 15.034509
final  value 15.034442
converged
      true
object  C Hk Hp
     1 16  0 14
     2  0 22  0
attr(,"error")
[1] NaN
attr(,"mismatch")
[1] 1
```

Different classification.

```
# weights:  13
initial  value 37.963702
...

final  value 14.778896
converged
      true
object  C Hk Hp
     1 16  0 14
     2  0 22  0
attr(,"error")
[1] NaN
attr(,"mismatch")
[1] 1
```

Different classification.

```
# weights:  13
initial  value 38.419771
iter  10 value 34.000557
final  value 34.000455
converged
      true
object  C Hk Hp
     2 16 22 14
attr(,"error")
[1] NaN
attr(,"mismatch")
[1] 1
```

Different classification.

We find that, ***even with the same parameters***, we get different classifications, different numbers of iterations, and, even for the same classification, different minima (`final value`). Why?

To help us understand what has happened, suppose we run the neural network 10 times on the flea beetle data using one hidden node, the 6 inputs variables and 2 outputs (the number of classes). If we use random initialization we find that we get the following minima at termination:

```
[ 1] 48.51438 48.51438 48.51446 15.00773 15.42307 15.00780 48.51439 48.51446
[ 9] 48.51443 15.41893
```

The answer is that the `nnet` routine ***initializes the weights at random***, so we start at different locations - and then tries to find the absolute minimum in a 13-dimensional weight space.

To try to get an idea of what is happening, we can take the data and change it so that there are only two input vectors, 2 classes (and hence 2 output nodes). With reduction of the data, we can get down to 7 weights:

```
targets <- class.ind(c(rep("C", 5), rep("Hb", 4)))
plot(df.flea[1:9,1],df.flea[1:9,2], pch=targets+1)
```

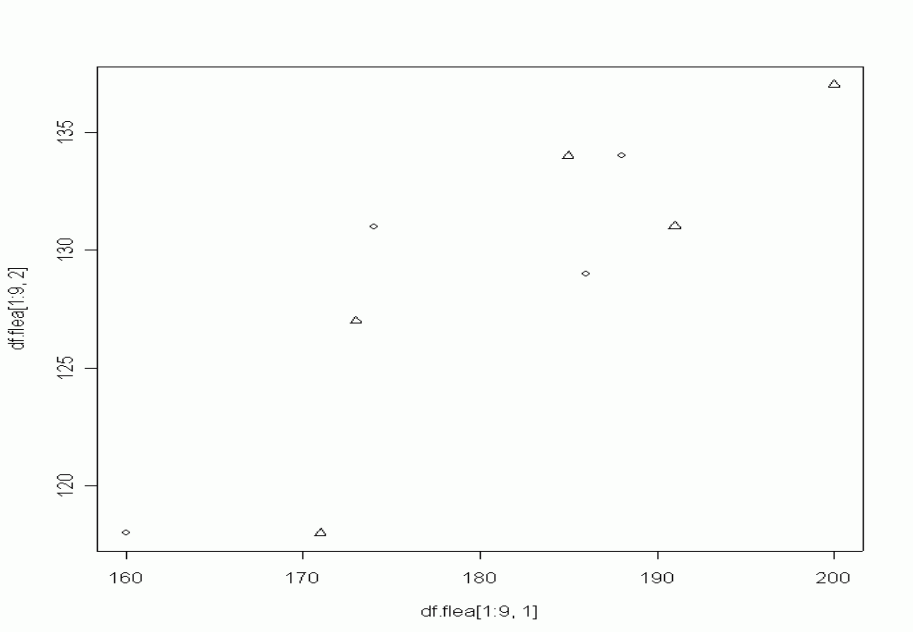


Figure 18.

Now run the following several times:

```
flea.nn <- nnet(df.flea[1:9,1:2], targets, size=1, rang=0.1, decay=5e-4, maxit=200
# weights:  7
initial  value 4.529719
iter  10 value 4.444539
```

```
...
final  value 4.444498
converged
# weights:  7
initial  value 4.456285
...
final  value 2.982019 converged
# weights:  7
initial  value 4.496130
...
final  value 2.765461
converged
# weights:  7
initial  value 4.474008
...
final  value 2.780494
```

Again we see different final values.

It is interesting to look at what happens if we fix the initialization for 5 of the weights and let the other 2 vary over a rectangle.

The following shows a ***perspective plot*** and a ***contour plot*** of the minima attained for the various 'free' weights:

```
library(rgl)
z <- matrix(0, 81, 81)
x<-1:81
y<-1:81
for (i in x) {
   for (j in y) {
for (i in x) {
   for (j in y) {
      mywts <- c((i-41)/200, (j-41)/200, 0, 0, 0, 0, 0)
      flea.nn <- nnet(df.flea[1:9,1:2], targets, Wts=mywts, size=1,
                     rang=0.1, decay=5e-4, maxit=200,trace=F)
      z[i,j] <- flea.nn$value
      cat("x = ",i," y = ",j," min = ", flea.nn$value," wts =
         ",floor(1000*flea.nn$wts)/1000,"\n")
   }
}
```

```
x =  1  y =  1  min =  2.780487  wts =  -1.549 -5.527 8.013 3.711 -4.403 -3.712 4.
x =  1  y =  2  min =  2.780484  wts =  -1.619 -5.505 7.982 3.715 -4.407 -3.716 4.
x =  1  y =  3  min =  2.967719  wts =  1.678 -5.951 8.06 0.916 -5.114 -0.917 5.113
x =  1  y =  4  min =  2.780537  wts =  -1.331 -5.538 8.028 3.69 -4.385 -3.691 4.384
x =  1  y =  5  min =  2.967729  wts =  1.544 -5.965 8.08 0.917 -5.126 -0.918 5.125
x =  1  y =  6  min =  2.780551  wts =  -1.286 -5.557 8.054 3.699 -4.389 -3.7 4.388
x =  1  y =  7  min =  2.780498  wts =  -1.534 -5.556 8.055 3.712 -4.406 -3.713 4.
x =  1  y =  8  min =  2.780496  wts =  -1.511 -5.503 7.978 3.698 -4.391 -3.699 4.
x =  1  y =  9  min =  2.967752  wts =  1.443 -5.987 8.111 0.914 -5.12 -0.915 5.119
x =  1  y =  10  min =  2.967774  wts =  1.39 -5.949 8.06 0.919 -5.115 -0.92 5.114
x =  1  y =  11  min =  2.967721  wts =  1.642 -5.943 8.05 0.915 -5.117 -0.916 5.116
...
x =  1  y =  40  min =  2.780484  wts =  -1.608 -5.513 7.993 3.719 -4.412 -3.72 4.
x =  1  y =  41  min =  4.444497  wts =  -0.004 -0.061 -0.044 0.222 0.001 -0.223 -
x =  1  y =  42  min =  2.765456  wts =  1.582 5.408 -7.846 -0.691 4.946 0.69 -4.947
```

```
x =  1  y =  43  min =  2.780488  wts =  -1.541 -5.517 7.999 3.712 -4.404 -3.713 4
...
x =  2  y =  38  min =  2.780486  wts =  -1.57 -5.52 8.004 3.714 -4.407 -3.715 4.406
x =  2  y =  39  min =  2.780486  wts =  -1.567 -5.509 7.988 3.707 -4.401 -3.708 4
x =  2  y =  40  min =  2.780543  wts =  -1.536 -5.606 8.127 3.691 -4.381 -3.692 4
x =  2  y =  41  min =  4.444497  wts =  -0.005 -0.065 -0.046 0.222 0.001 -0.223 -
x =  2  y =  42  min =  2.765472  wts =  1.464 5.343 -7.75 -0.691 4.938 0.69 -4.939
x =  2  y =  43  min =  2.780489  wts =  -1.553 -5.531 8.019 3.718 -4.412 -3.719 4
...
x =  3  y =  24  min =  2.781137  wts =  -0.748 -5.418 7.849 3.827 -4.516 -3.828 4
x =  3  y =  25  min =  2.780488  wts =  -1.532 -5.512 7.992 3.71 -4.403 -3.711 4.
x =  3  y =  26  min =  2.780491  wts =  -1.584 -5.54 8.033 3.698 -4.391 -3.699 4.
x =  3  y =  27  min =  4.444514  wts =  -0.19 -0.07 0 0.222 -0.001 -0.223 0
x =  3  y =  28  min =  4.444514  wts =  -0.19 -0.065 0 0.222 -0.001 -0.223 0
x =  3  y =  29  min =  4.444514  wts =  -0.19 -0.06 0 0.222 -0.001 -0.223 0
x =  3  y =  30  min =  4.444514  wts =  -0.19 -0.055 0 0.222 -0.001 -0.223 0
x =  3  y =  31  min =  2.780486  wts =  -1.569 -5.523 8.008 3.717 -4.409 -3.718 4
x =  3  y =  32  min =  2.780491  wts =  -1.507 -5.525 8.011 3.716 -4.409 -3.717 4
x =  3  y =  33  min =  2.780548  wts =  -1.297 -5.54 8.03 3.699 -4.388 -3.7 4.387
x =  3  y =  34  min =  2.780499  wts =  -1.468 -5.531 8.018 3.721 -4.412 -3.722 4
...
x =  4  y =  20  min =  2.780486  wts =  -1.566 -5.506 7.983 3.709 -4.402 -3.71 4.
x =  4  y =  21  min =  2.780506  wts =  -1.429 -5.519 8.001 3.703 -4.397 -3.704 4
x =  4  y =  22  min =  2.967788  wts =  1.367 -6.024 8.162 0.917 -5.118 -0.918 5.
x =  4  y =  23  min =  2.780491  wts =  -1.505 -5.518 8.001 3.713 -4.406 -3.714 4
x =  4  y =  24  min =  4.444515  wts =  -0.185 -0.085 0 0.222 -0.001 -0.223 0
x =  4  y =  25  min =  4.444514  wts =  -0.185 -0.08 0 0.222 -0.001 -0.223 0
x =  4  y =  26  min =  4.444514  wts =  -0.185 -0.075 0 0.222 -0.001 -0.223 0
x =  4  y =  27  min =  4.444514  wts =  -0.185 -0.07 0 0.222 -0.001 -0.223 0
x =  4  y =  28  min =  4.444513  wts =  -0.185 -0.065 0 0.222 -0.001 -0.223 0
x =  4  y =  29  min =  4.444513  wts =  -0.185 -0.06 0 0.222 -0.001 -0.223 0
x =  4  y =  30  min =  4.444513  wts =  -0.185 -0.055 0 0.222 -0.001 -0.223 0
x =  4  y =  31  min =  2.780522  wts =  -1.423 -5.552 8.049 3.741 -4.432 -3.742 4
x =  4  y =  32  min =  2.781296  wts =  -0.37 -5.571 8.068 3.719 -4.412 -3.72 4.411
x =  4  y =  33  min =  2.780542  wts =  -1.313 -5.505 7.98 3.736 -4.429 -3.737 4.
x =  4  y =  34  min =  2.780495  wts =  -1.584 -5.544 8.038 3.688 -4.381 -3.689 4
x =  4  y =  35  min =  2.780493  wts =  -1.521 -5.512 7.992 3.701 -4.392 -3.702 4
x =  4  y =  36  min =  2.780527  wts =  -1.353 -5.54 8.03 3.716 -4.406 -3.717 4.405
x =  4  y =  37  min =  2.780487  wts =  -1.551 -5.522 8.006 3.709 -4.402 -3.71 4.
x =  4  y =  38  min =  2.967742  wts =  1.478 -5.972 8.09 0.917 -5.13 -0.918 5.129
x =  4  y =  39  min =  4.444513  wts =  -0.185 -0.045 -0.041 0.222 0.011 -0.223 -
x =  4  y =  40  min =  2.780504  wts =  -1.438 -5.526 8.011 3.706 -4.397 -3.707 4
x =  4  y =  41  min =  4.444498  wts =  -0.007 -0.071 -0.049 0.222 0.003 -0.223 -
x =  4  y =  42  min =  2.982028  wts =  -1.526 6.122 -8.298 -3.638 4.557 3.637 -4
x =  4  y =  43  min =  2.780488  wts =  -1.531 -5.515 7.996 3.709 -4.402 -3.71 4.
x =  4  y =  44  min =  2.765450  wts =  1.521 5.395 -7.826 -0.691 4.924 0.69 -4.925
...
x =  5  y =  20  min =  2.780489  wts =  -1.528 -5.516 7.997 3.712 -4.404 -3.713 4
x =  5  y =  21  min =  2.780484  wts =  -1.625 -5.515 7.997 3.715 -4.408 -3.716 4
x =  5  y =  22  min =  4.444515  wts =  -0.18 -0.095 0 0.222 0 -0.223 0
x =  5  y =  23  min =  4.444514  wts =  -0.18 -0.09 0 0.222 -0.001 -0.223 0
x =  5  y =  24  min =  4.444514  wts =  -0.18 -0.085 0 0.222 -0.001 -0.223 0
x =  5  y =  25  min =  4.444513  wts =  -0.18 -0.08 0 0.222 -0.001 -0.223 0
x =  5  y =  26  min =  4.444513  wts =  -0.18 -0.075 0 0.222 -0.001 -0.223 0
x =  5  y =  27  min =  4.444513  wts =  -0.18 -0.07 0 0.222 -0.001 -0.223 0
x =  5  y =  28  min =  4.444512  wts =  -0.18 -0.065 0 0.222 -0.001 -0.223 0
x =  5  y =  29  min =  4.444512  wts =  -0.18 -0.06 0 0.222 -0.001 -0.223 0
x =  5  y =  30  min =  4.444512  wts =  -0.18 -0.055 0 0.222 -0.001 -0.223 0
x =  5  y =  31  min =  4.444511  wts =  -0.18 -0.05 0 0.222 -0.001 -0.223 0
x =  5  y =  32  min =  2.967738  wts =  1.498 -5.964 8.079 0.918 -5.125 -0.919 5.
x =  5  y =  33  min =  2.780486  wts =  -1.557 -5.516 7.998 3.71 -4.402 -3.711 4.
x =  5  y =  34  min =  2.780535  wts =  -1.336 -5.533 8.021 3.694 -4.385 -3.695 4
...
```

```
x =  6  y =  18  min =  2.780489  wts =  -1.533 -5.518 8 3.711 -4.405 -3.712 4.404
x =  6  y =  19  min =  2.967737  wts =  1.526 -5.988 8.111 0.915 -5.113 -0.916 5.
x =  6  y =  20  min =  4.444515  wts =  -0.175 -0.105 0 0.222 0 -0.223 0
x =  6  y =  21  min =  4.444514  wts =  -0.175 -0.1 0 0.222 0 -0.223 0
x =  6  y =  22  min =  4.444514  wts =  -0.175 -0.095 0 0.222 0 -0.223 0
x =  6  y =  23  min =  4.444513  wts =  -0.175 -0.09 0 0.222 -0.001 -0.223 0
x =  6  y =  24  min =  4.444513  wts =  -0.175 -0.085 0 0.222 -0.001 -0.223 0
x =  6  y =  25  min =  4.444513  wts =  -0.175 -0.08 0 0.222 -0.001 -0.223 0
x =  6  y =  26  min =  4.444512  wts =  -0.175 -0.075 0 0.222 -0.001 -0.223 0
x =  6  y =  27  min =  4.444512  wts =  -0.175 -0.07 0 0.222 -0.001 -0.223 0
x =  6  y =  28  min =  4.444511  wts =  -0.175 -0.065 0 0.222 -0.001 -0.223 0
x =  6  y =  29  min =  4.444511  wts =  -0.175 -0.06 0 0.222 -0.001 -0.223 0
x =  6  y =  30  min =  4.444511  wts =  -0.175 -0.055 0 0.222 -0.001 -0.223 0
x =  6  y =  31  min =  4.444511  wts =  -0.175 -0.05 0 0.222 -0.001 -0.223 0
x =  6  y =  32  min =  2.967729  wts =  1.608 -5.988 8.111 0.915 -5.118 -0.916 5.
x =  6  y =  33  min =  2.780487  wts =  -1.543 -5.52 8.003 3.716 -4.409 -3.717 4.
```

It can be seen that, in some cases, the weights for a similar minima are quite similar:

```
x =  6  y =  21  min =  4.444514  wts =  -0.175 -0.1 0 0.222 0 -0.223 0
x =  6  y =  22  min =  4.444514  wts =  -0.175 -0.095 0 0.222 0 -0.223 0
x =  6  y =  23  min =  4.444513  wts =  -0.175 -0.09 0 0.222 -0.001 -0.223 0
x =  6  y =  24  min =  4.444513  wts =  -0.175 -0.085 0 0.222 -0.001 -0.223 0
```

while in other cases there are considerable differences:

```
x =  4  y =  38  min =  2.967742  wts =  1.478 -5.972 8.09 0.917 -5.13 -0.918 5.129
x =  4  y =  42  min =  2.982028  wts =  -1.526 6.122 -8.298 -3.638 4.557 3.637 -4
persp3d(x,y,z, col = "lightblue")
contour((x-41)/200,(y-41)/200,z, nlevels=10,col=c(1,2,3,4,5,6,7,8,9,10))
```

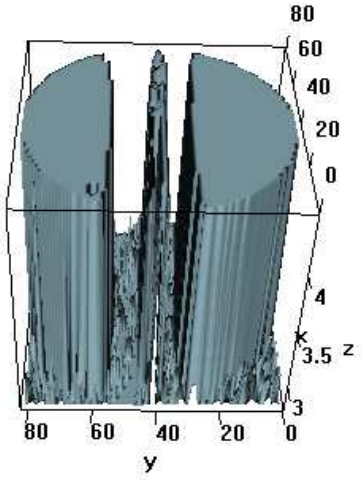


Figure 19 (**a**) Perspective plot

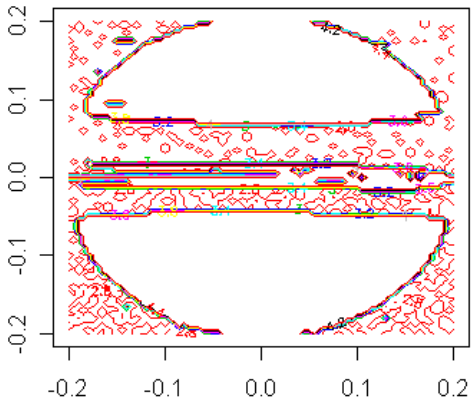


Figure 19 (**b**) Contour plot

We see from the images above that slight changes in the initial weights (only two of the seven changed - the other five started at 0) resulted in convergence to quite different minima. Keep in mind that the images represent the minima to which the process converges based on the initial values of the two "free" weights.

The minima have values in the following range:

```
range(z)
[1] 2.765447 4.444516
```

This had the weights initialized as `c((i-41)/200, (j-41)/200, 0, 0, 0, 0, 0).`

The following use the same code with different fixed weights - 0.5 rather than 0:

```
mywts <- c((i-41)/200, (j-41)/200, 0.5, 0.5, 0.5, 0.5, 0.5)
```

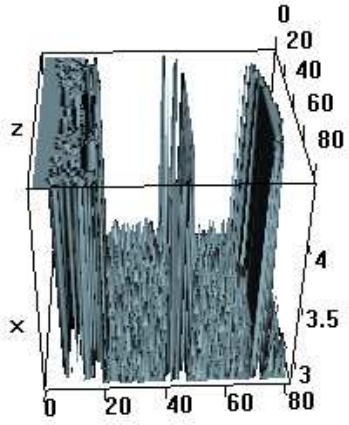


Figure 20 (**a**) Perspective plot

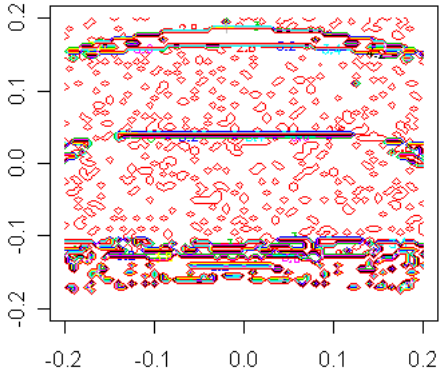


Figure 20 (**b**) Contour plot

```
range(z)
[1] 2.765447 4.444488
```

Again with the fixed weights at 1.0:

```
mywts <- c((i-41)/200, (j-41)/200, 1, 1, 1, 1, 1)
```

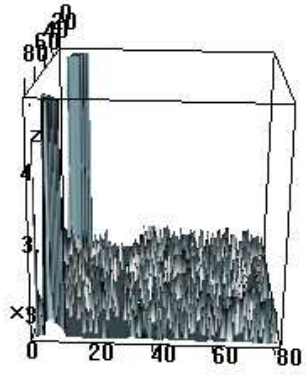


Figure 21 (**a**) Perspective plot

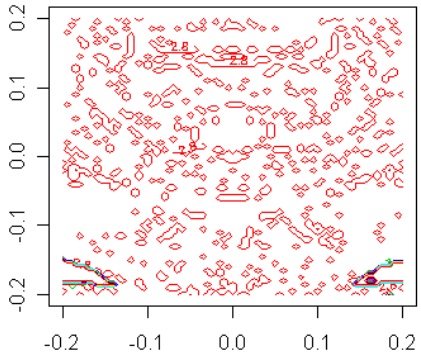
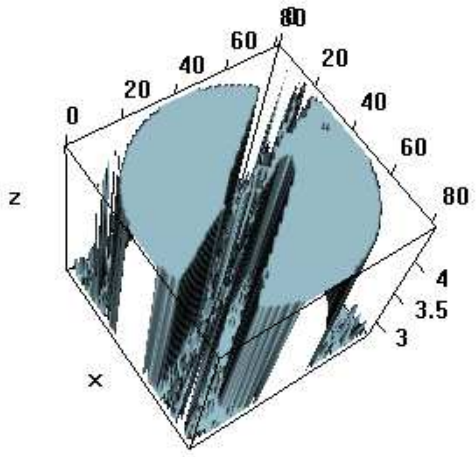


Figure 21 (**b**) Contour plot

```
range(z)
[1] 2.765447 4.444475
```

The following sets the first weight and the last four to 0:

```
mywts <- c(0, (i-41)/200, (j-41)/200, 0, 0, 0, 0)
```



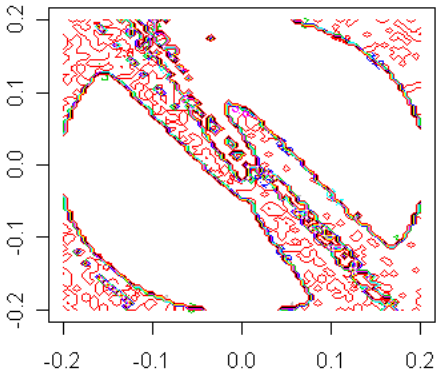


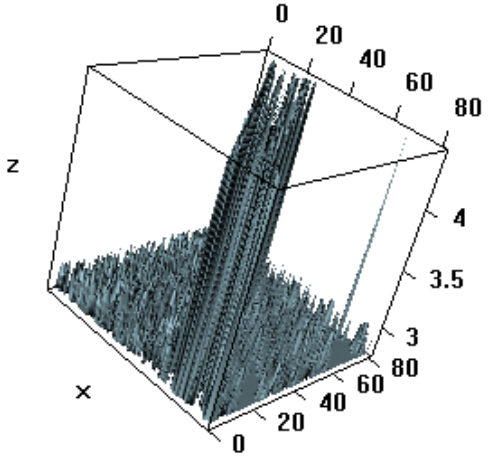
Figure 22 (**a**) Perspective plot

Figure 22 (**b**) Contour plot

```
range(z)
[1] 2.765447 4.444515
```

and then to 1.0:

```
mywts <- c(1, (i-41)/200, (j-41)/200, 1, 1, 1, 1)
```



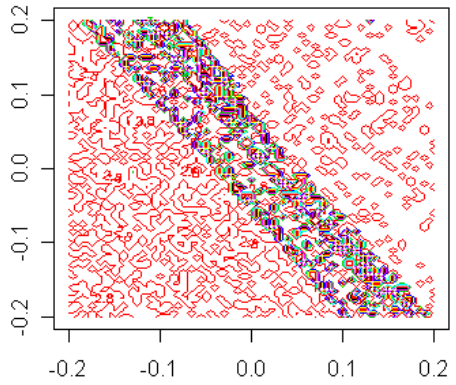


Figure 23 (**a**) Perspective plot

Figure 23 (**b**) Contour plot

```
range(z)
[1] 2.765447 4.444516
```

and then the first three to 0 and the last two to 1.0:

```
mywts <- c(0, 0, 0, (i-41)/200, (j-41)/200, 1, 1)
```

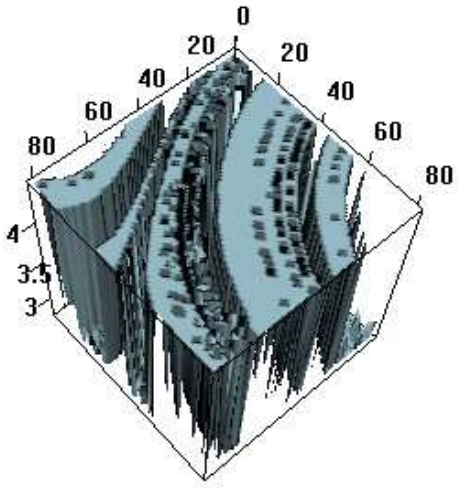


Figure 24 (**a**) Perspective plot

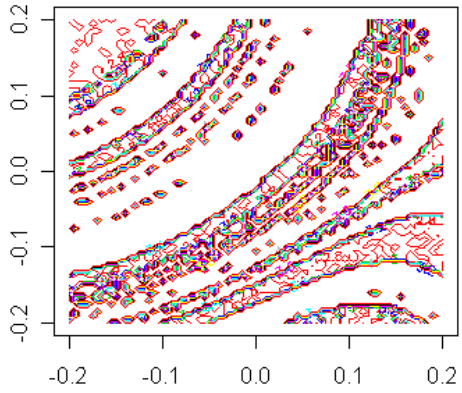


Figure 24 (**b**) Contour plot

```
range(z)[1] 2.765447 4.444507
```

We see that there are several different minima for this data. Depending on the initial weights, we may converge to a relative minimum rather than the absolute minimum. Care must be taken to avoid such suboptimal solutions.

As before, we will try neural networks on the synthetic data intoduced in the previous lecture.

We read in the function for plotting the examples, the data, and the training/test set indices:

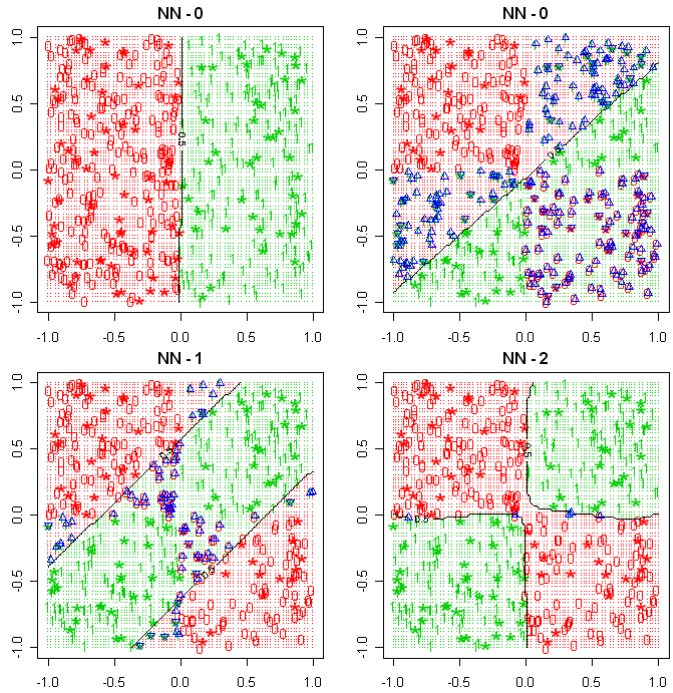
```
source(paste(code.dir, "example_display.r", sep="/"))
load(paste(data.dir, "syn.Rdata", sep="/"))
load(paste(data.dir, "synTrainTest.Rdata", sep="/"))
   #
syn.train.class <- res$tp[tt.ind[[1]],]
syn.train.data <- res$data[tt.ind[[1]],]
   #
syn.test.class <- res$tp[tt.ind[[2]],]
syn.test.data <- res$data[tt.ind[[2]],]
```

The following is the usual menu function to allow us to run neural nets on the different data sets with different numbers of hidden nodes:

```
f.menu <- function(){
   while (TRUE) {
      cat("Enter the number of the data set","\n")
      cat ("0 to quit\n")
      resN <- menu(c("1:2", "1/3:2/4", "1:2/3:4", "1:2 rotated", "1/3:2/4 rotated"
                     "1:2/3:4 rotated", "Hole", "y^2"))
      if (resN == 0) break
      size <- readline("Number of nodes - 0 to 20: ")
      example.display(syn.train.data, syn.train.class[,resN], syn.test.data,
            syn.test.class[,resN],  c(100,100), paste("NN -", size), model.exp, predict
            as.integer(size))
   }
}
f.menu()
```

The following illustrates ***the nature of the distributions of data that can be classified using neural networks***. Note that, if the class boundary is a straight line, no hidden layer is required. A hidden layer with at least one node is required if the class boundary is curved. Note that for the circle, several examples are given with the same number of nodes in the hidden layer with different results, indicating that the results are dependent on the initial weights.

In the confusion matrices, the results are given as $\frac{training}{test}$ where there is a difference.



true		
object	0	1
0	220 / 59	0
1	0	180 / 41

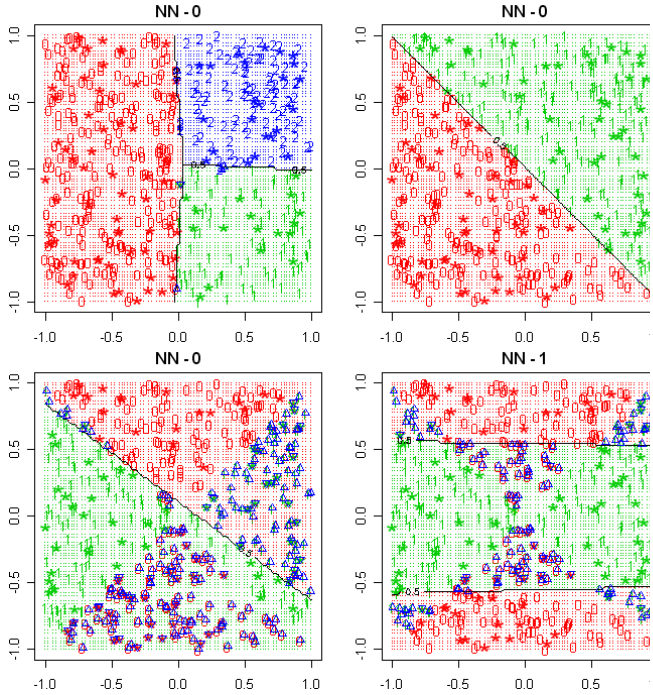
true		
object	0	1
0	118 / 25	110 / 26
1	85 / 18	87 / 31

true		
object	0	1
0	159 / 38	18 / 9
1	44 / 5	179 / 48

true		
object	0	1
0	201 / 43	3 / 0
1	2 / 0	196 / 57

Figure 25.

Set 1	Set 2
Set 2	Set 2



true			
object	0	1	2
0	218 / 58	0 / 1	0 / 0
1	1 / 0	85 / 17	2 / 0
2	1 / 1	0 / 0	93 / 23

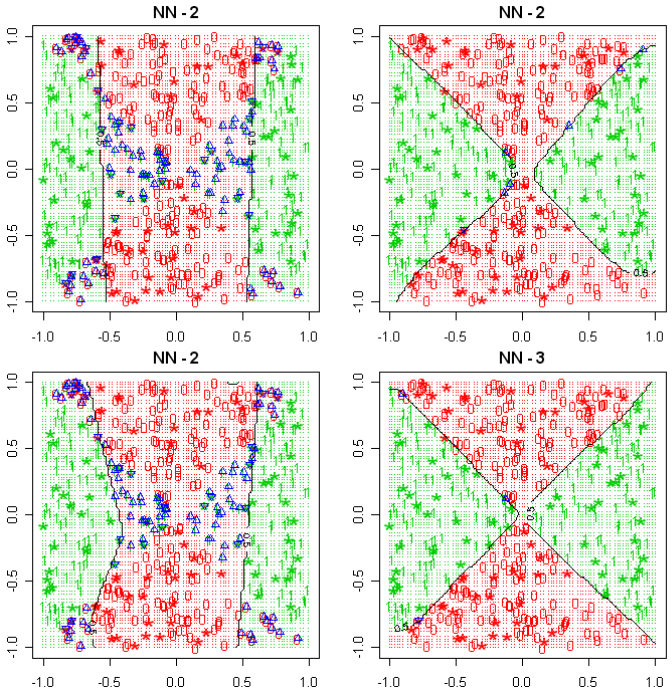
true		
object	0	1
0	202 / 54	0
1	0	19 / 46

true		
object	0	1
0	102 / 19	80 / 23
1	97 / 27	121 / 31

true		
object	0	1
0	136 / 35	51 / 9
1	63 / 11	15 / 45

Figure 26.

Set 3	Set 4
Set 5	Set 5



true		
object	0	1
0	163/35	53/17
1	36/10	148/37

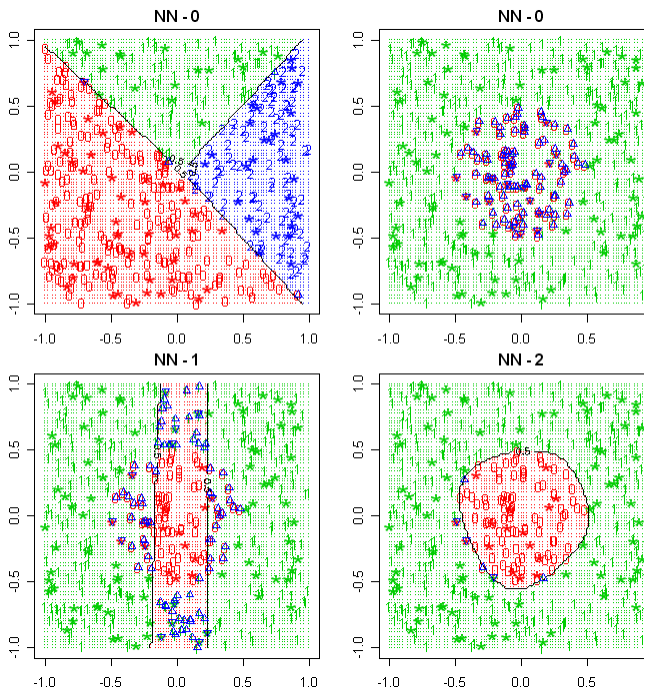
true		
object	0	1
0	196/45	3/0
1	3/1	198/54

true		
object	0	1
0	164/38	49/14
1	35/8	152/40

true		
object	0	1
0	198/46	0/2
1	1/0	201/52

Figure 27.

Set 5	Set 5
Set 5	Set 5



true			
object	0	1	2
0	201/53	1/0	1/0
1	0/1	104/19	0
2	1/0	1/0	91/27

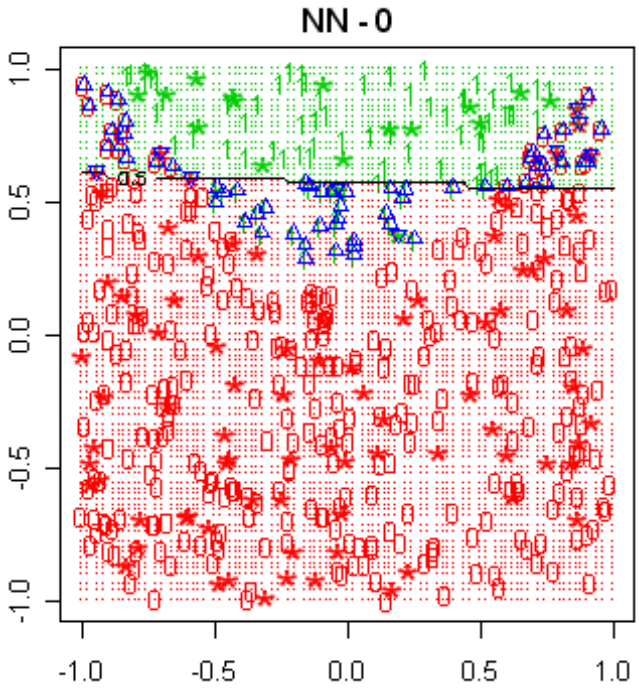
true		
object	1	(
1	317/84	8/1

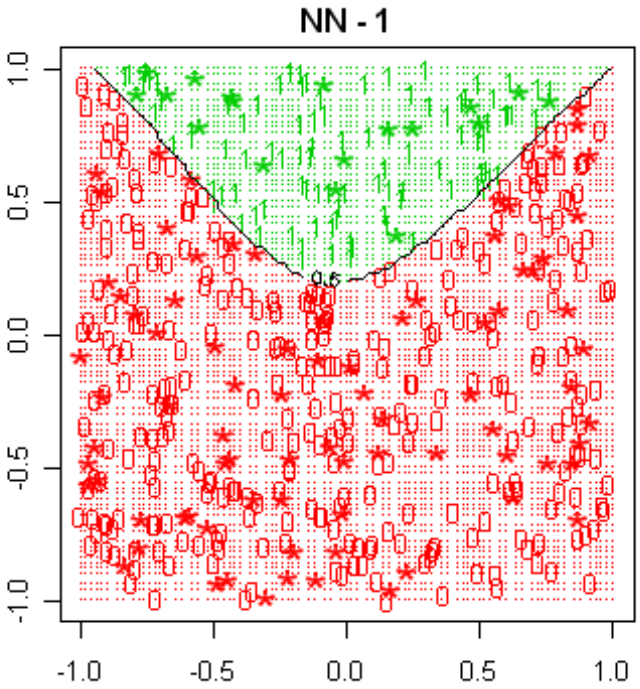
true		
object	0	1
0	48/10	39/9
1	35/6	278/75

true		
object	0	1
0	82/14	2/1
1	1/2	31/8

Figure 28.

Set 6	Set 7
Set 7	Set 7





	true	
object	0	1
0	278 75	32 2
1	26 7	64 16

	true	
object	0	1
0	304 82	0 0
1	0	99 18

Figure 29. Set 8 Set 8

Weight decay

In all of our examples we use a value of 0.0005 for the weight decay without an understanding of what it does or if this is an appropriate value. We will now look at what happens if we use different values of the weight decay in one of our synthetic examples and then look at the role of this parameter.

One of the problems with examining the effects of changing a parameter is that, unless we specify the initial weights, the `nnet` function will assign them for us.

We will use the following weight set for our illustration:

```
Wts <- c(-0.076,0.034,-0.046,0.076,-0.056,0.024,0.029,-0.002,-0.013,0.096,
         0.02,-0.1,-0.03,0.094,0.079,-0.077,0.092,0.021,-0.1,-0.029,0.067,
         -0.005,-0.066,-0.036,0.07,-0.045,-0.073,-0.05,-0.002,0.006,0.026,
         -0.027,-0.004,-0.01,-0.067,0.073,-0.028,0.016,-0.002,0.048,-0.004,
         -0.06,0,0.085,-0.084,0.012,-0.076,-0.048,-0.088,-0.029,0.017,0.076)
```

Because the `example.display` and `plot.class.boundary` functions are not flexible enough to use for this purpose, a revised version is used:

```
wt.decay.demo <- function(train.data, train.class, size, decay, Wts) {
   plot.class(train.data, train.class, {}, {},
              paste("Set = ",7, " Size = ", size," Decay = ", decay))
      #
   model <-nnet(data.frame(train.data), class.ind(train.class),
                skip=F, softmax = T, size=size, decay=decay, maxit = 1000,
   trace = F, Wts=Wts)
   res <- f.create.grid(train.data, c(100,100))
   xp <- res$xp
   yp <- res$yp
   zp <- res$zp
   dimnames(zp)[[2]]<- dimnames(train.data)[[2]]
   Z <- as.integer(predict(model, zp, type="class"))
   points(zp[,1], zp[,2], pch=".", col= (matrix(Z, length(xp), length(yp))+2))
   for (c in 1:2) {
      contour(xp, yp, matrix(as.numeric(Z==(c-1)), length(xp), length(yp)),
              add=T, levels=0.5, labex=0, col=1)
   }
   model$wts
}
```

This combines parts of both functions and allows us to alter both the decay and initial weights.

As can be seen in the function, we will use synthetic set 7 for this example. First, we will have a weight decay of 0 (the numbers that follow are the final weights):

```
wt.decay.demo(syn.train.data, syn.train.class[,7], 10, 0, Wts)

 [1]   46.952944  -72.321578  -99.543309  -67.505512 -128.530383 -112.573710   24.
 [8]    2.406699  -86.910964    4.655850   -5.063192    9.925622  165.129026  -52.
[15] -111.858845   73.052638   20.531977 -147.603460  -17.720219  -20.236981 -131.
[22]  -63.040967    7.908969   56.876267   30.960780  -98.432307   39.383626  -19.
[29]  -42.131693    9.172620 -207.027488   78.284520 -197.114643   41.715497  168.
[36] -142.902458   80.409175   93.197333   48.933154  106.898010 -161.048555  206.
[43]  -78.311520  197.195643  -41.809497 -168.752360  142.899458  -80.485175  -93.
[50]  -48.964154 -106.833010  161.120555
```

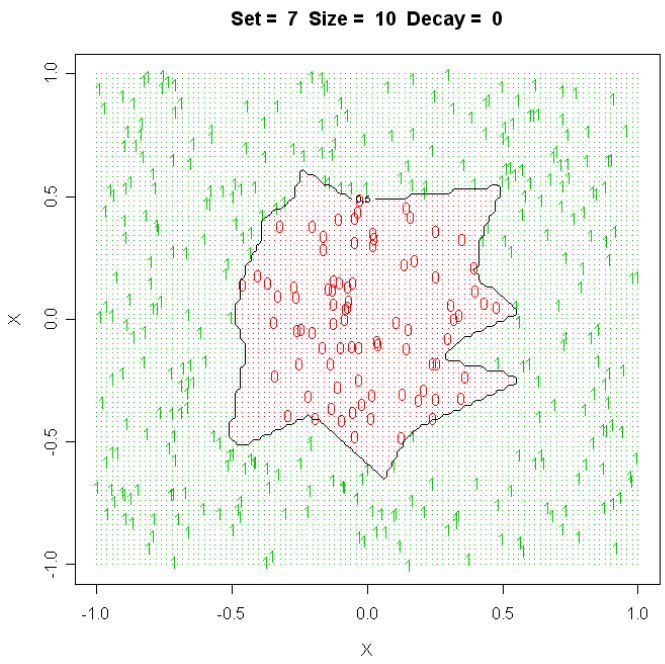


Figure 30.

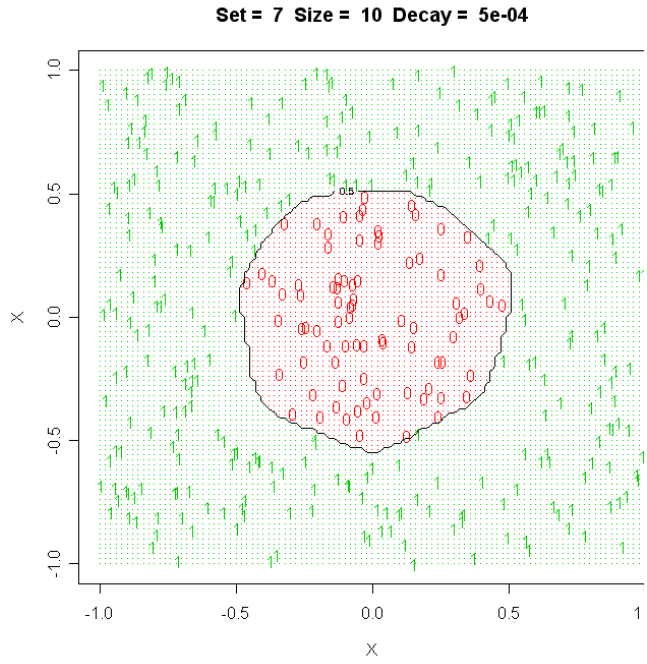


Figure 31.

then our default value of 0.0005:

```
wt.decay.demo(syn.train.data, syn.train.class[,7], 10, 0.0005, Wts)
 [1]  -5.87928080   5.03279020  12.19698210  -4.85386908  11.01464585   2.96233865
 [7]   2.15728880  -0.18055929   0.07299016  -6.39541033 -15.08979694   1.96283858
[13]   2.17110174  -0.18986848   0.06882179  -2.99942496   5.81146042   4.75170420
[19]  -5.80564095 -12.47637429  -8.74936714   8.63444402 -17.02970141  12.67236964
[25]   7.85763930  -0.30151434  18.40550548  -6.86009708  -7.98902089  13.30337368
[31]  -3.64575415 -11.07458910  -9.05655467  -3.29344431 -12.86662984  -3.29296441
[37]  -6.43938568 -10.88162586  13.14176398  11.87692163 -13.07925292   3.64575466
[43]  11.07458950   9.05655347   3.29344570  12.86663065   3.29296445   6.43938681
[49]  10.88162693 -13.14176352 -11.87692260  13.07925186
```

and finally 0.02:

```
wt.decay.demo(syn.train.data, syn.train.class[,7], 10, 0.02, Wts)
 [1]  3.104485804 -1.455452680 -7.090912984 -2.877788537 -5.136496553  5.016078302
 [7]  0.119925527 -0.008408174  0.029998843  0.120244244 -0.009551926  0.032034089
[13] -2.423589572 -3.290603562 -4.717679548 -2.305383046  3.733994023 -4.991637432
[19] -2.883649417  6.541906313  2.752506359  0.120419263 -0.009514419  0.030042892
[25]  1.829243424 -3.362771870  3.387582234 -2.401569808 -6.188614088 -1.550652177
[31] -0.672603942  5.595841465 -6.059654330 -0.357572600 -0.357828448 -4.605786190
[37] -4.879244836 -5.863651265 -0.357939658  3.040065787 -5.058816013  0.672605305
[43] -5.595840383  6.059651082  0.357576369  0.357830653  4.605786310  4.879247883
[49]  5.863654152  0.357940901 -3.040068393  5.058813126
```

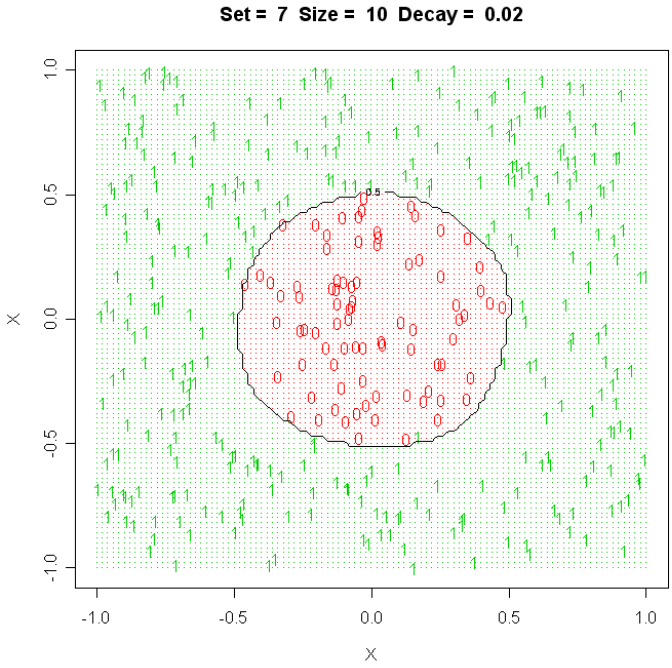


Figure 32.

When we look at the final weights, we see that the magnitudes are decreasing as the weight decay value increases. It is also obvious from the figures that increasing the weight decay has resulted in a smoother curve. In ***data smoothing***, we found that wider windows resulted in smoother curves that did not fit the training data as well but might be better at predicting for test data (i.e.we might be less likely to overfit the data).

When we were deriving the equations for the neural network, we tried to *minimize the error* term

$$E = \sum_{i=1}^{N}\sum_{k=1}^{K}\left(t_{ik} - f_O\left(\mathbf{W}_k^T\ \mathbf{T}_i\right)\right)^2$$

We also know that the absolute minimum would come when the network fits every training case exactly (*overfitting*).

To avoid this, we can add a ***penalty term*** to the error to give an error term in the form of

$$E + \lambda_1 \sum_{l=0}^{p} w_{lm}^2 + \lambda_2 \sum_{k=0}^{M} W_{mk}^2$$

(for simplicity, the ***tuning parameters*** λ_1 and λ_2 have been replaced by a single value in `nnet`). The effect of such a penalty term is to force (*shrink*) the coefficients towards 0 for increasing values of λ_1 and λ_2.

Recall that it was mentioned that we might split our data into ***three*** sets. Here is a situation in which that might be required. The first set could be used to *train the data* for various values of λ_1 and λ_2; the second could be used to *determine the best values* of λ_1 and λ_2 to use; and the third set to *test the model*.

As before, it might be instructive to look at the separating surfaces in three dimensions.

We will look at the ***flea beetles*** in three dimensions using variables 2, 3, and 5. The classes are not as well separated in these directions so the surfaces may be a bit more complex.

We use the function `disp.3d` from earlier in order to find the points at which the classes change:

```
library(MASS)
```

We create the three dimensional dataset:

```
first <- 5
second <- 3
third <- 2
data.for.3d <- data.frame(d.flea[,first],d.flea[,second],d.flea[,third])
colnames(data.for.3d) <- colnames(d.flea[,c(first,second,third)])
```

Because neural networks can converge to different minima depending on the initial weights, we will set:

```
Wts <-
c(-0.99,10.71,-4.93,0.95,-15.76,-1.8,0.37,0.16,-15.34,13.5,11.89,10.6,-8.59,
        -13.63,-9.84,-6.11,12.75)
```

as our initial values.

Because we would like ***surfaces other than planes***, we will use two (2) nodes in the hidden layer for one example and 10 nodes in the hidden layer for the second example:

```
flea.nn.3d.2 <- nnet(data.for.3d, class.ind(flea.species), size=2, skip=F,
rang=0.1,
                     decay=5e-4, maxit=500, Wts=Wts)
```

```
# weights:  17
initial  value 17.170432
iter  10 value 15.382686
iter  20 value 15.381768
final  value 15.381755
converged

summary(flea.nn.3d.2)
a 3-2-3 network with 17 weights
options were - decay=5e-04
 b->h1 i1->h1 i2->h1 i3->h1
 -0.94  10.74  -4.94   0.95
 b->h2 i1->h2 i2->h2 i3->h2
-15.73  -1.79   0.38   0.16
 b->o1 h1->o1 h2->o1
-15.33  13.50  11.88
 b->o2 h1->o2 h2->o2
 10.62  -8.59 -13.63
 b->o3 h1->o3 h2->o3
 -9.83  -6.12  12.74

confusion.expand(max.col(flea.nn.3d.2$fitted.values), flea.species)
         true
object    C    Hk   Hp   | Row Sum
  1       16   1    0    | 17
  2       5    29   0    | 34
  3       0    1    22   | 23
  ------- ---- ---- ---- | ----
  Col Sum 21   31   22   | 74
attr(,"error")
[1] NaN
attr(,"mismatch")
[1] 1

flea.nn.3d.2[11]      # or flea.nn.3d.2$wts
$wts
 [1] -0.9429123  10.7362345  -4.9361434  0.9533044 -15.7269710  -1.7932020
 [7]  0.3789444   0.1601158 -15.3255563 13.4954494  11.8781092  10.6156708
[13] -8.5906243 -13.6286737  -9.8275448 -6.1223647  12.7414013
```

We get the points at which the classes change:

```
data.3d.2 <- disp.3d(flea.nn.3d.2, data.for.3d, "net", 60)
```

and display them:

```
library(rgl)
plot3d(d.flea[,c(first, second, third)], type="s", col = species+1, size=0.5)
points3d(data.3d.2, size=2)
```

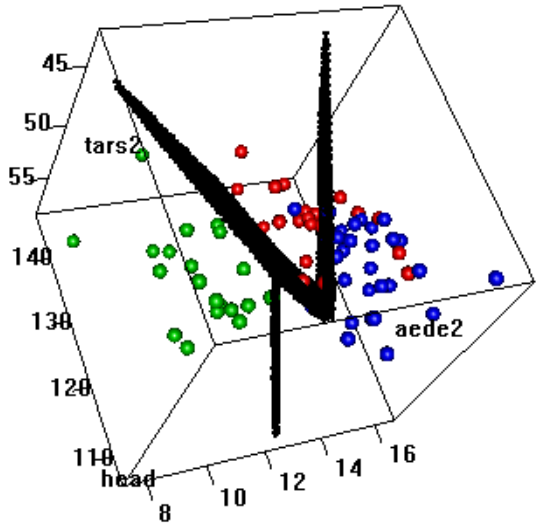


Figure 33. 2 nodes in hidden layer

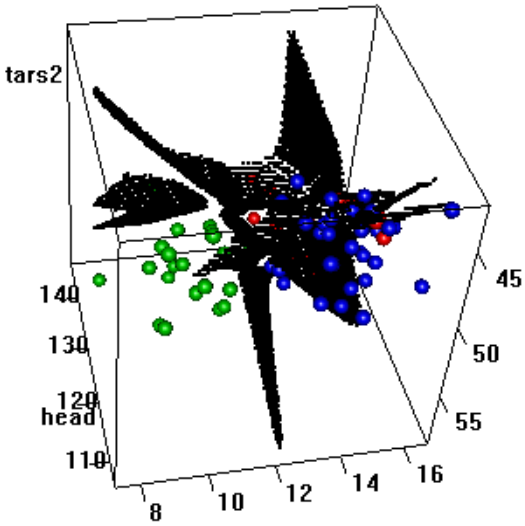


Figure 34. 10 nodes in hidden layer

Now repeat with 10 nodes in the hidden layer:

```
Wts <- c(16.54,2.17,-0.53,-0.17,-0.01,0.05,0.07,0.18,7.36,2.86,2.32,-1.3,17.5,
         -0.87,1.29,-0.54,-1.03,5.59,-0.82,-0.2,0,0.01,0.05,0.14,1.78,-3.16,
         -10.6,4.61,-5.24,5.15,6.69,-3.25,-6.26,0.58,8.4,-3.23,-2.99,8.87,
         -4.52,0.98,-4.44,-12.29,-4.47,-3.67,-15.14,1.44,-1.77,7.25,11.69,
         13.3,8.87,0.71,12.26,0.74,4.37,14.72,8.72,-0.75,-7.52,-11.99,-13.7,
         -8.6,0.76,-0.38,0.76,-0.65,1.88,-7.29,0.84,1.77,-0.55,0.73,-4.73)
flea.nn.3d.10 <- nnet(data.for.3d, class.ind(flea.species), size=10, skip=F,
rang=0.1,
                      decay=5e-4, maxit=500, Wts=Wts)
confusion.expand(max.col(flea.nn.3d.10$fitted.values), flea.species)
         true
object     C    Hk    Hp    | Row Sum
  1        20   0     0     | 20
  2        1    31    0     | 32
  3        0    0     22    | 22
  ------- ---- ---- ---- | ----
  Col Sum 21    31    22    | 74
attr(,"error")
[1] NaN
attr(,"mismatch")
[1] 1
```