Artificial Neural Networks 2

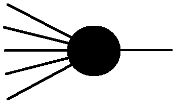
Roger Barlow
The University of Huddersfield

CODATA School on Data Science

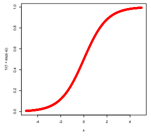
Autumn 2021

First lecture: Artificial Neural Networks

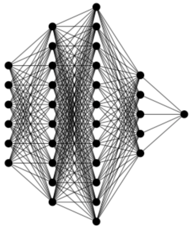
Nodes



Weighted sum fed through Threshold Function

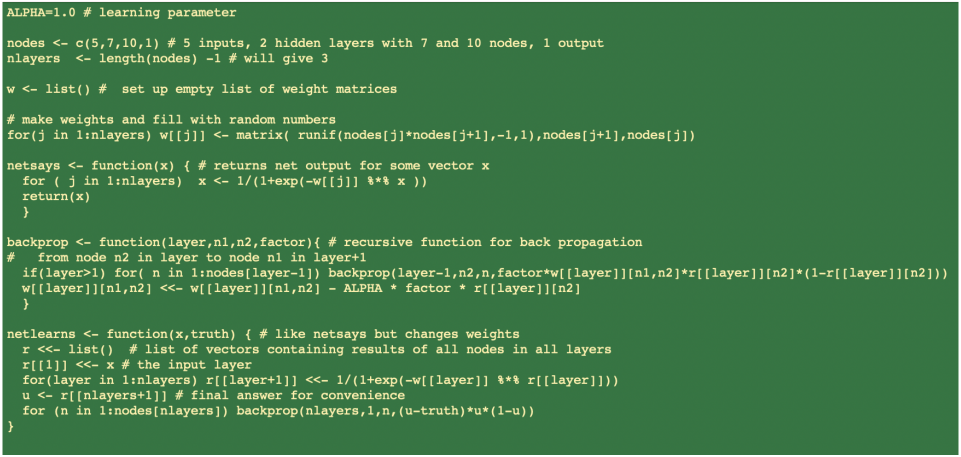


Put together into
Multilayer Perceptron



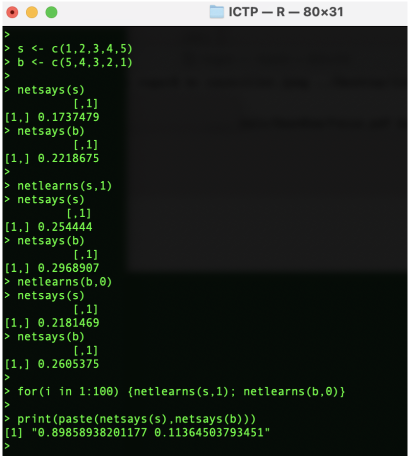
Now to turn pictures into programs...

This is a complete neural network program: type it in



```
ALPHA=1.0 # learning parameter

nodes <- c(5,7,10,1) # 5 inputs, 2 hidden layers with 7 and 10 nodes, 1 output
nlayers  <- length(nodes) -1 # will give 3

w <- list() #  set up empty list of weight matrices

# make weights and fill with random numbers
for(j in 1:nlayers) w[[j]] <- matrix( runif(nodes[j]*nodes[j+1],-1,1),nodes[j+1],nodes[j])

netsays <- function(x) { # returns net output for some vector x
  for ( j in 1:nlayers)  x <- 1/(1+exp(-w[[j]] %*% x ))
  return(x)
  }

backprop <- function(layer,n1,n2,factor){ # recursive function for back propagation
#   from node n2 in layer to node n1 in layer+1
  if(layer>1) for( n in 1:nodes[layer-1]) backprop(layer-1,n2,n,factor*w[[layer]][n1,n2]*r[[layer]][n2]*(1-r[[layer]][n2]))
  w[[layer]][n1,n2] <<- w[[layer]][n1,n2] - ALPHA * factor * r[[layer]][n2]
  }

netlearns <- function(x,truth) { # like netsays but changes weights
  r <<- list()  # list of vectors containing results of all nodes in all layers
  r[[1]] <<- x # the input layer
  for(layer in 1:nlayers) r[[layer+1]] <<- 1/(1+exp(-w[[layer]] %*% r[[layer]]))
  u <- r[[nlayers+1]] # final answer for convenience
  for (n in 1:nodes[nlayers]) backprop(nlayers,1,n,(u-truth)*u*(1-u))
}
```

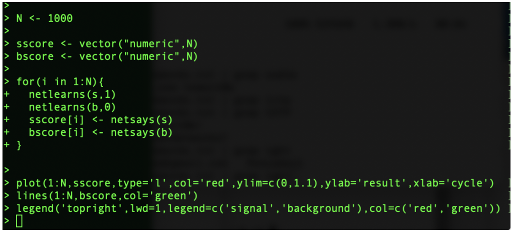
Fire up R and run your program. Then...



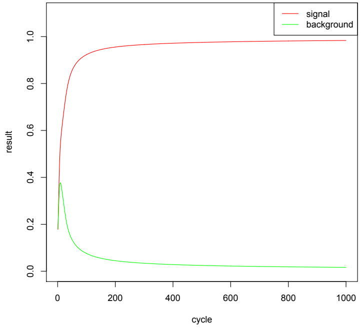
```
>
> s <- c(1,2,3,4,5)
> b <- c(5,4,3,2,1)
>
> netsays(s)
          [,1]
[1,] 0.1737479
> netsays(b)
          [,1]
[1,] 0.2218675
>
> netlearns(s,1)
> netsays(s)
         [,1]
[1,] 0.254444
> netsays(b)
          [,1]
[1,] 0.2968907
> netlearns(b,0)
> netsays(s)
          [,1]
[1,] 0.2181469
> netsays(b)
          [,1]
[1,] 0.2605375
>
> for(i in 1:100) {netlearns(s,1); netlearns(b,0)}
>
> print(paste(netsays(s),netsays(b)))
[1] "0.89858938201177 0.11364503793451"
>
```

Print the `w` matrices just to check they're what you expect (not shown)
Define signal and background target vectors
See what the network makes of them - at this stage it will be random rubbish
Train it on one signal event
It increases the score for signal but also the score for background
Training it one one background event nudges it down again
Do this many times, and sense emerges

Bit more exploring



```
>
> N <- 1000
>
> sscore <- vector("numeric",N)
> bscore <- vector("numeric",N)
>
> for(i in 1:N){
+   netlearns(s,1)
+   netlearns(b,0)
+   sscore[i] <- netsays(s)
+   bscore[i] <- netsays(b)
+ }

>
> plot(1:N,sscore,type='l',col='red',ylim=c(0,1.1),ylab='result',xlab='cycle')
> lines(1:N,bscore,col='green')
> legend('topright',lwd=1,legend=c('signal','background'),col=c('red','green'))
> 
```



Best to re-set weights, perhaps by re-running your program.
Choose a large number and declare two empty vectors
Train many times and save the results
Plot them
After a bit the network settles down and provides virtually perfect separation
This is unrealistic in that the training events are always the same. But it's good to ensure the program is working

Conclusion

It's easy to create a Neural Network using R.
As you train it, its performance improves.
More realistic examples and more details of performance next time.

Activity for you

Run this and look at the effect of changing alpha.
What happens when it is small?
How large can it be before the network fails to train?
Then try changing the network topology. Explore larger and smaller networks.

Present some result(s) as a plot and be ready to show it at a class zoom meeting