Chapter 7

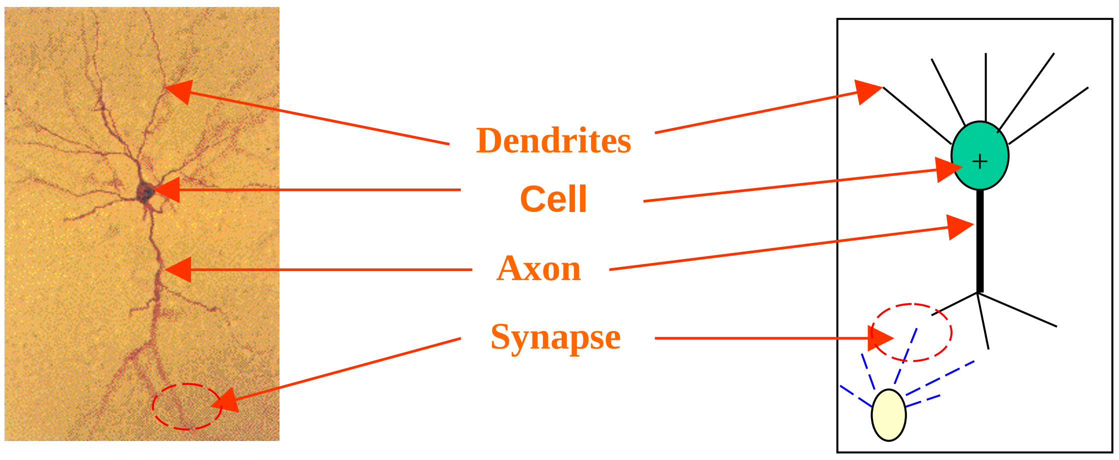
Artificial Neural Network

The Binary Logistic regression, multinomial logit and Classification Tree are some commonly used statistical techniques for classification. A completely different approach is the Artificial Neural Network (**ANN**). This approach is to mimic the functions and mechanism of our brain. Although how our brain memorizes, recognizes and generalizes pattern is still a mystery, if not completely, some aspects of the structure of brain are well-known at least to the neuroscientists. This provides a useful and workable model to build an ANN.

In 1943, McCulloch and Pits postulated a simple mathematical model to explain how biological neurons work. There is not much progress in this area until 1970 when the computer was invented. In 1970's, Hopfield invented **back- propagation** algorithm to train neural networks so as to match the fitted output with the actual demanded response as much as possible. Since then, there are many successful examples and applications in the labs. In 1980's, research moved from the labs to commercial world, typical applications like detecting fraud credit card transactions, real estate appraisal, and data mining. Before we go into the topics of ANN, we need to learn a bit about the structure our human brains.

7.1 Human brain and Artificial Neuron

Our brain consists of approximately 100 billion of some specific type of cell known as neuron. Each of these neurons is typically connect with thousands to 10,000 other neurons. Most importantly, unlike other cells, these neurons will not regenerate. It is widely accepted that these neurons responsible for our ability for memorizing, learning, generalizing and thinking. Within the neuron, there are four items: Dendrites, cell body, Axon and Synapse. These neurons are connected to form a huge and very complicated network. The exact function of these neurons is still a mystery but a very simple mathematical model that mimics these neurons provides a surprising good performance in pattern recognition, classification and prediction.

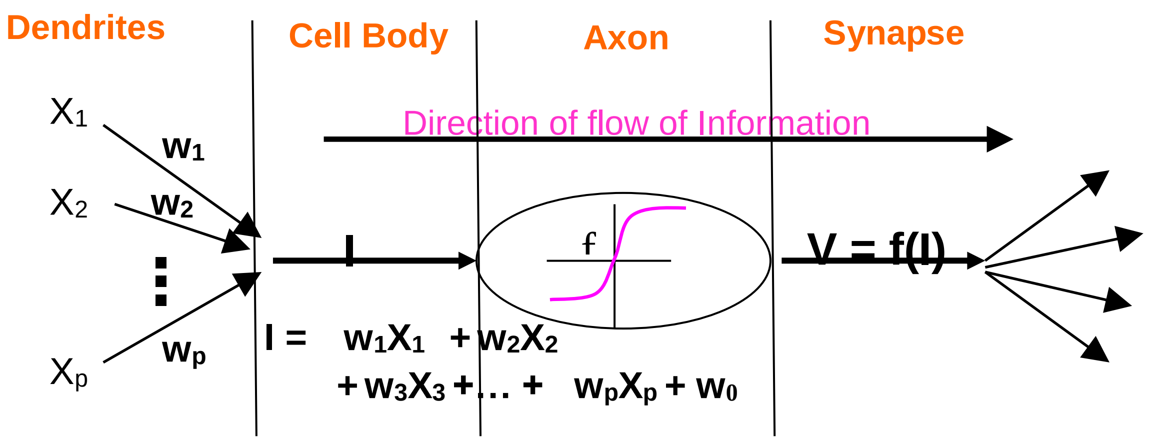


Hippocampal Neurons

Source: heart.cbl.utoronto.ca/ ~berj/projects.html

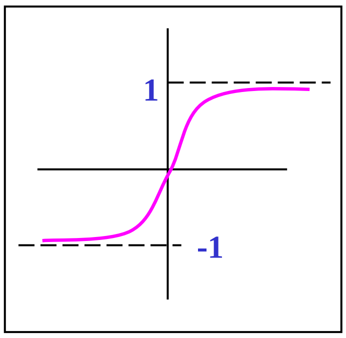
Schematic

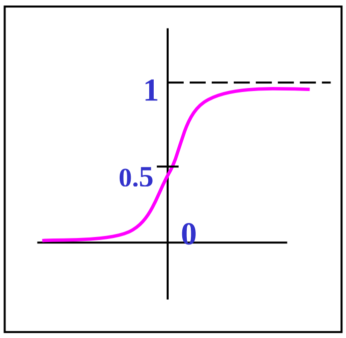
Dendrites responsible for receiving information; cell body is for processing information; axon amplifying the signal carried by the processed information to other neurons and synapse is the junction between axon end and dendrites of other neurons. To mimic this neuron, we have the following artificial neuron:

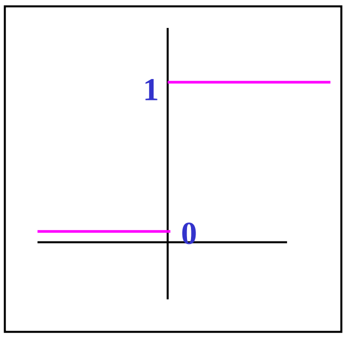

Dendrites
Cell Body
Axon
Synapse
Direction of flow of Information
X1
w1
X2
w2
Xp
wp
I
f
V = f(I)
I = w1X1 + w2X2 + w3X3 + ... + wpXp + w0

$x_1, \ldots, x_p$ are the inputs received from other neurons or our outside environment. The total input I is formed from the linear combination of these inputs with weights $w_0, w_1, \ldots, w_p$. (w_0 is known as the bias). The **transfer** function (or **Activation** function) converts the the input I to output V=f(I). The output V will go to other neurons as input.

There are some commonly used transfer functions inside the Axon:







Tanh (hyperbolic tangent)
$f(x) = (e^x - e^{-x}) / (e^x + e^{-x})$

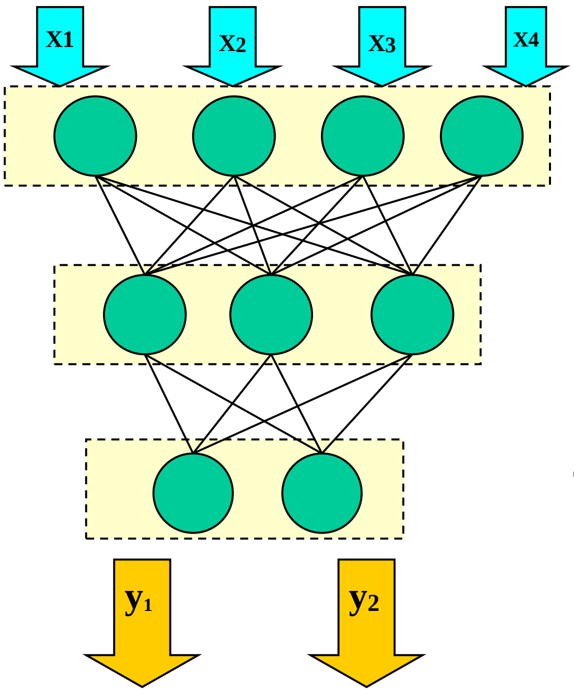
Logistic
$f(x) = e^x / (1 + e^x)$
$= 1 / (1 + e^{-x})$

Threshold
$$f(x) = \begin{cases} 0 \text{ if } x < 0 \\ 1 \text{ if } x >= 0 \end{cases}$$

Usually, logistic transfer function is used; that is why artificial neural network is closely related to logistic regression. It is easy to show that the derivative of the logistic function is $f'(x) = f(x)[1 - f(x)]$, as seen before in Chapter 5.

7.2 Feed-forward Network

These artificial neurons are connected from one layer to other to form a network.



Input Layer
- Each neuron gets ONLY one input, directly from outside

Hidden Layer
- Connects Input and Output layers

Output Layer
- Output of each neuron directly goes to outside

In this network, we have three layers: input layer, hidden layer and output layer. The number of neurons in the input, hidden and output is respectively 4, 3, and 2. This is known as a 4-3-2 ANN.

Remarks:

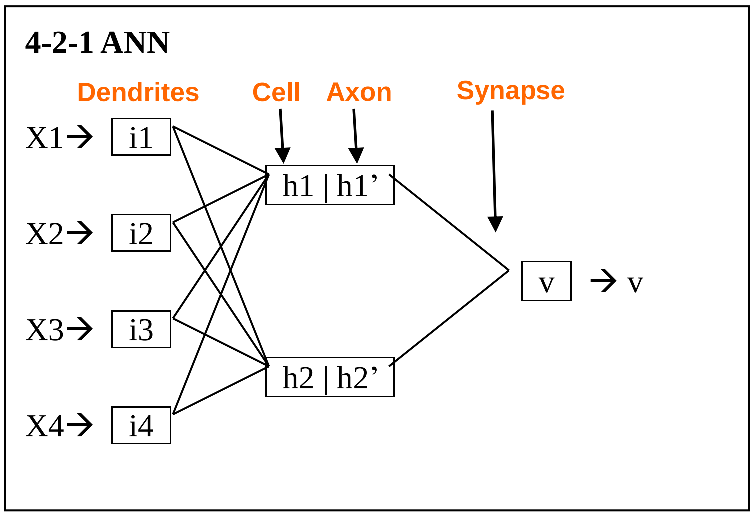
1. The number of hidden layers can be zero, one, two, …, etc.
2. The number of neurons in the input and output layer are determined by the nature of the problem, but the number of neurons in the hidden layer is user-defined.
3. Within each layer, neurons are not connected to each other. Neurons in one layer are connected only to neurons in the (forward) next layer (Feed-forward).
4. Each line joining the neuron is associated by a weight w_{ij}. These weights are unknown parameters need to be estimated from the training dataset.

7.3 ANN using R

Given the input X and output Y in the training dataset, we have to find the w_{ij} such that the sum of square error $E = (Y - V)'(Y - V)$ is minimized, where V is the prediction from the ANN, that is *V=f(X)*. However, the algebraic form of V is too complex to write down explicitly. The objective function E has many local minima and usually the **Back-propagation** algorithm is used to find the minimum. R has a built-in function *nnet()* inside the library *nnet* to implement the ANN. Let us illustrate this by our examples.

```
> library(nnet)                                 # load library nnet
> d<-read.csv("iris.csv")                       # read in data
# ANN with single hidden layer and linear output unit
> iris.nn<-nnet(d[,1:4],d[,5],size=2,linout=T)  # linear output
# weights:  13
initial  value 304.982032
iter  10 value 4.946966
iter  20 value 4.849086
iter  30 value 4.847140
iter  40 value 4.846500
iter  50 value 4.843899
iter  60 value 4.837284
iter  70 value 4.811071
iter  80 value 4.445104
iter  90 value 2.190095
iter 100 value 1.743840
final  value 1.055574
stopped after 100 iterations
> summary(iris.nn)                     # summary of output
a 4-2-1 network with 13 weights
options were - linear output units     # weights
 b->h1 i1->h1 i2->h1 i3->h1 i4->h1
 -4.96 -11.21 -14.22  38.14  20.95
 b->h2 i1->h2 i2->h2 i3->h2 i4->h2
153.65  35.55  37.28 -75.82 -64.98
  b->o  h1->o  h2->o
  1.99   1.00  -0.99
```

The above ANN is represented as the following diagram:



$$
\begin{aligned}
h_1 &= -4.96 - 11.21x_1 - 14.22x_2 + 38.14x_3 + 20.95x_4 \\
h_2 &= 153.65 + 35.55x_1 + 37.28x_2 - 75.82x_3 - 64.98x_4 \\
h_1^{'} &= \exp(h_1)/[1+\exp(h_1)] \qquad h_2^{'} = \exp(h_2)[1+\exp(h_2)] \\
v &= 1.99 + h_1^{'} - 0.99h_2^{'}
\end{aligned} \tag{7.1}
$$

Let us use the 1st, 51th and 101th observations from the iris data set to illustrate how this ANN makes prediction. The 1st observation is x=(5.1, 3.5, 1.4, 0.2)'. According to the formulae in (7.1),

h1 = -4.96+(-11.21)(5.1)+(-14.22)(3.5)+(38.14)(1.4)+(20.95)(0.2)=-54.315,
h2 = 153.65+(35.55)(5.1)+(37.28)(3.5)+(-75.82)(1.4)+(-64.98)(0.2)=346.291,
h1'=exp(h1)/(1+exp(h1))=0, h2'=exp(h2)/(1+exp(h2))=1, and
v = 1.99+h1'-0.99h2'=1.

Similarly for the 51th observation, x=(7, 3.2, 4.7, 1.4)', h1=79.654, h2=74.47, h1'=1, h2'=1, and v=2; and the 101th observation, x=(6.3, 3.3, 6, 2.5)', h1=158.71, h2=-116.7 h1'=1, h2'=0 and v=2.99 (round to the nearest integer, see Section 7.4).

In fact, v is stored in iris.nn$fitted.values, and we can produce the classification table using the following:

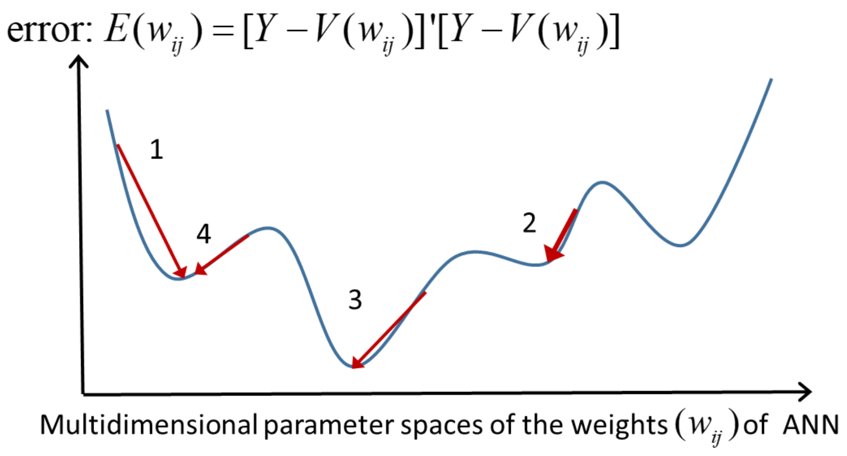
```
> pred<-round(iris.nn$fit)         # round the fitted values
> table(pred,d[,5])                # classification table
   pred
     1  2  3
  1 50  0  0
  2  0 49  0
  3  0  1 50
```

Let us try the HSI example. Instead of specifying the model in the argument, we can input the matrix of input and vector of output in the function nnet as follows:

```
> d<-read.table("fin-ratio.csv")   # read in data
> names(d)                         # to see what is in d
[1] "EY"    "CFTP"  "ln_MV" "DY"    "BTME"  "DTE"   "HSI"
> x<-d[,1:6]                       # create the matrix of input
> fin.nn<-nnet(x,d$HSI,size=2,linout=T,maxit=200)   # set max. it to 200
# weights:  17
initial  value 50.245625
iter  10 value 25.359299
iter 200 value 7.332069
final  value 7.332069
stopped after 200 iterations
> summary(fin.nn)
a 6-2-1 network with 17 weights
options were - linear output units
 b->h1 i1->h1 i2->h1 i3->h1 i4->h1 i5->h1 i6->h1
 12.76  -0.03   0.01  -1.33  -0.01   1.00   0.04
 b->h2 i1->h2 i2->h2 i3->h2 i4->h2 i5->h2 i6->h2
 11.93  -0.04  -0.05  -1.24  -0.02   1.09   0.04
  b->o  h1->o  h2->o
  0.65 -16.57  15.93

> pred<-round(fin.nn$fit)              # prediction
> table(pred,d$HSI)                    # classification table
    pred
       0   1
  0 645    7
  1   3   25
```

A major problem of ANN is that the solution depends on the starting values. When we run `nnet()` each time, we may obtain different results. This is illustrated in the graph below.


error: $E(w_{ij})=[Y-V(w_{ij})]'[Y-V(w_{ij})]$
1
4
3
2
Multidimensional parameter spaces of the weights (w_{ij}) of ANN

The error function, while depends on the weights (w_{ij}) of the ANN, contains a lot of local minima in the multi-dimensional spaces of w_{ij}. As nnet() randomly assign an initial set of parameters w_{ij} and use backpropagation (essentially a gradient steepest descent [Newton] method) for minimizing the error function. The resulting weights w_{ij} of the ANN may only be the parameter at a local minimum of the error function. We need to run nnet() several times from different sets of initial parameters in order to get the real optimal weight w_{ij} of ANN from the true global minimum of the error function. Therefore, we have to run `nnet()` several times and seek for the best possible result. Here is an improved version of the `nnet()` function:

```
# improved nnet()
# Try nnet(x,y) k times and output the best trial
# x is the matrix of input variable
# y is the dependent value; y must be factor if linout=F is used
library(nnet)
ann<-function(x,y,size,maxit=100,linout=F,try=5) {
  ann1<-nnet(y~.,data=x,size=size,maxit=maxit,linout=linout)
  v1<-ann1$value          # save the value for the first trial

  for (i in 2:try) {
    ann<-nnet(y~.,data=x,size=size,maxit=maxit,linout=linout)
    if (ann$value<v1) {   # check if the current value is better
      v1<-ann$value       # save the best value
      ann1<-ann           # save the results
    }
  }
  ann1                    # return the results
}
```

This function allows users to specify number of trials and save the best result. For example:

```
> d<-read.csv("fin-ratio.csv")
> source( “ann.r” )              # load ann() function
> fin.nn<-ann(d[,1:6],d[,7],size=2,linout=T,try=10)  # try 10 times
> fin.nn$value                   # display the best result
[1] 2.883506

> summary(fin.nn)                # weights correspond to the best result
a 6-2-1 network with 17 weights
options were - linear output units
 b->h1 i1->h1 i2->h1 i3->h1 i4->h1 i5->h1 i6->h1
188.58   9.10  10.57 -19.61  -0.73  -3.85   0.35
 b->h2 i1->h2 i2->h2 i3->h2 i4->h2 i5->h2 i6->h2
202.15  11.82  14.85 -13.72  12.13  -7.45   0.75
  b->o  h1->o  h2->o
  0.00  -0.94   0.94

> pred<-round(fin.nn$fit)
> table(pred,d$HSI)
   pred
      0   1
  0 645   0
  1   3  32
```

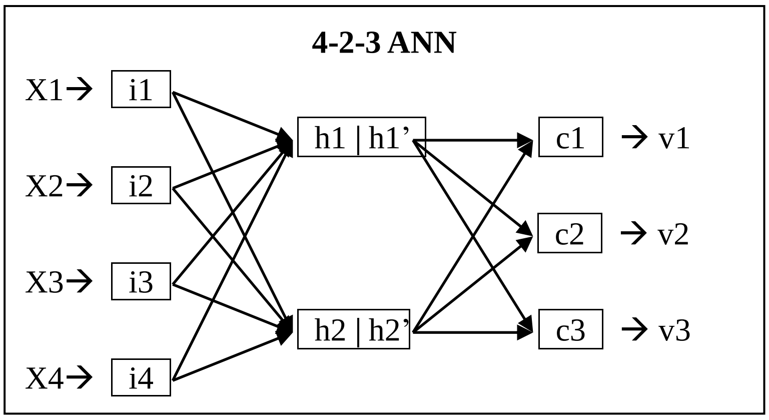
7.4 ANN with logistic output

We have seen the *nnet()* function with linear output option *linout=T* in R. The output is a linear function of the values in the neurons in the hidden layer and it is a real number. When using this in a classification problem, like iris flower or HSI example, we have to round the output to its nearest integer. However, we may use the default option *linout=F* and then *k* different logistic functions are used to connect the neurons and the output. This can be interpreted as the probability of an observation belongs to that group. Let us look at how to use this logistic output option in *nnet()*.

```
> d<-read.csv("iris.csv")          # read in data
> y<-as.factor(d[,5])              # change the output to factor
> x<-d[,1:4]                       # define input x
> iris.nn<-ann(x,y,size=2,maxit=200,try=10)     # try 10 times with logistic
> iris.nn$value                    # best value
[1] 4.922006
> summary(iris.nn)                 # display weights
a 4-2-3 network with 19 weights
options were - softmax modelling
  b->h1  i1->h1  i2->h1  i3->h1  i4->h1
   0.24  -16.23  -33.03   54.34    0.59
  b->h2  i1->h2  i2->h2  i3->h2  i4->h2
 398.41  217.56  215.06 -378.74 -302.15
  b->o1  h1->o1  h2->o1
  19.35 -110.32   55.19
  b->o2  h1->o2  h2->o2
 -10.01   53.17   69.59
  b->o3  h1->o3  h2->o3
  -9.62   56.68 -124.63
> pred<-max.col(iris.nn$fit)       # find col. no. of max. of fitted values
> table(pred,d[,5])                # classification table
pred  1  2  3
   1 50  0  0
   2  0 49  0
   3  0  1 50
```

Note that the fitted values in the output have k columns. Suppose that the fitted values are respectively c_{ij}, for j=1,…,k. Then the probability of the i-th observation belonging to the the j-th group is $p_{ij} = \exp(c_{ij})/\Sigma_{j=1}^{g}\exp(c_{ij})$. Normally speaking, we predict that the i-th observation belongs to group j if p_{ij} is maximum in the i-th row. Since p_{ij} is maximum if and only if c_{ij} is maximum. Therefore we assign the label to pred according to the max. column index in each row of iris.nn$fit. By using this method, we have only 1 error case out of 150 cases and the error rate is 0.67%!

Since in the iris flower example, the output has three levels, the diagram for this ANN with logistic output is as follows:



Note that for logistic output, the output of i^{th} record is $v_j = \text{logistic}\ (c_{ij}) = \Pr\{y_i = j\}$.

Now we try the other HSI example. Note that in this example, the output variable is binary (*k=2*). The fitted value in the output has only one column, i.e. the probability of belonging to group 1.

```
> d<-read.csv("fin-ratio.csv")        # read in data
> x<-d[,1:6]                          # define x
> y<-as.factor(d$HSI)                 # change y to factor
> fin.nn<-ann(x,y,size=2,maxit=200,try=10)
> fin.nn$value                        # best value
[1] 5.101526
> summary(fin.nn)
a 6-2-1 network with 17 weights
options were - entropy fitting
  b->h1  i1->h1  i2->h1  i3->h1  i4->h1  i5->h1  i6->h1
1158.65  -57.15   54.27 -123.28   -0.75  -18.64   10.61
  b->h2  i1->h2  i2->h2  i3->h2  i4->h2  i5->h2  i6->h2
 -15.26  -23.33   10.54    5.94   -0.27   -3.61   18.91
   b->o   h1->o   h2->o
 -26.85 -203.91   30.54
> pred<-1*(fin.nn$fit>1/2)            # create label
> table(pred,d$HSI)                   # classification table
pred   0   1
   0 647   0
   1   1  32
```

We assign the label to *pred* according to the probability given in *fin.nn$fit*. Note that there is only 1 error case out of totally *680* cases and the error rate is *0.14%*!

7.5 Training and testing dataset

To fully understand how to make predictions on a testing dataset using a trained ANN, let us consider the IRIS dataset again and randomly choose *120* observations as training data and remaining *30* observations as testing data.

```
> d<-read.csv("iris.csv")            # read in data
> set.seed(12345)                    # set random seed
> id<-sample(1:150,size=30,replace=F) # index for testing dataset ≈ 85% of all data
> d1<-d[-id,]                        # create training dataset
> d2<-d[id,]                         # create testing dataset
> iris.nn<-ann(d1[,1:4],d1[,5],size=2,linout=T,maxit=200,try=10)   # linear output
> iris.nn$value                                                     # best value
[1] 0.9750302
> summary(iris.nn)                                                  # weights
a 4-2-1 network with 13 weights
options were - linear output units
  b->h1  i1->h1  i2->h1  i3->h1  i4->h1
  83.34   71.66   82.18 -131.74  -73.63
  b->h2  i1->h2  i2->h2  i3->h2  i4->h2
  -4.96   -0.93   -6.20   10.35    4.87
   b->o   h1->o   h2->o
   1.97   -0.98    1.00
```

There is an item *wts* in *iris.nn* contains all the weights in this ANN. We first create two matrices *w1* and *w2* containing the weights for *i->h* and *h->o* as follows:

```
> w1<-matrix(iris.nn$wts[1:10],nr=2,byrow=T)    # weights matrix from i to h
> w2<-matrix(iris.nn$wts[11:13],nr=1,byrow=T)   # weights matrix from h to o
> w1
          [,1]       [,2]     [,3]       [,4]       [,5]
[1,] 83.337618 71.6612716 82.17638 -131.73741 -73.632762
[2,] -4.961494 -0.9324202 -6.19693   10.35123   4.865432
> w2
         [,1]      [,2]      [,3]
[1,] 1.974976 -0.975008 0.9995585
```

Let us apply these weights to the testing dataset *d2*. First we need to define the logistic function as follows:

```
> logistic<-function(x) { 1/(1+exp(-x)) }  # define the logistic function
> x<-cbind(1,d2[,1:4])                     # form the matrix x, note that a column
> dim(x)                                   # of ones corresponds to the bias in w1
[1] 30  5
> out1<-logistic(w1%*%t(x))                # This is the output h'
> out1<-rbind(1,out1)                      # append a row of ones on top of out1
> dim(out1)                                # corresponds to the bias in w2
[1]  3 30
```

Note that we first create a matrix *x* from *d2*, with a column of ones in the front corresponds to the bias in *w1*. This is the output value of *h'*. In order to compute the fitted value, we need to append a row of ones on top of *out1* which corresponds to the bias in *w2*. Finally, we multiply *w2* to *out1* to obtain the fitted values.

```
> out2<-t(w2%*%out1)      # compute fitted values
> dim(out2)
[1] 30  1
> pr<-round(out2)         # round to nearest integer
> table(pr,d2[,5])        # classification table for testing dataset
pr   1  2  3
  1  9  0  0
  2  0  8  0
  3  0  1 12
```

The built-in function *predict()* will perform the above calculations and gives the same prediction.

```
> pr<-predict(iris.nn,d2)     # apply the saved ann to testing data d2
> table(round(pr),d2[,5])

     1  2  3
  1  9  0  0
  2  0  8  0
  3  0  1 12
```

Although we use linear output option in this example, similar commands can be used for logistic output option.

```
y<-as.factor(d1[,5])                               # change y to factor
iris.nn<-ann(d1[,1:4],y,size=2,maxit=200,try=10)   # logistic output
iris.nn$value                                      # display best value

pr<-predict(iris.nn,d2)          # apply saved output iris.nn to d2
pred<-max.col(pr)                # find col. index of max. pr
table(pred,d2[,5])               # classification table
pred  1  2  3
   1  9  0  0
   2  0  8  0
   3  0  1 12
```

Now we try the fin-ratio.csv and randomly select 544 records as training dataset.

```
d<-read.csv("fin-ratio.csv")      # read in data
set.seed(12345)                   # set random seed
id<-sample(1:680,size=544)        # index for testing dataset ≈ 85% of all data
d1<-d[id,]                        # testing dataset
d2<-d[-id,]                       # training dataset
y<-as.factor(d1$HSI)              # change y to factor
fin.nn<-ann(d1[,1:6],y,size=2,maxit=200,try=10)
fin.nn$value                      # display best value
[1] 4.359164
```

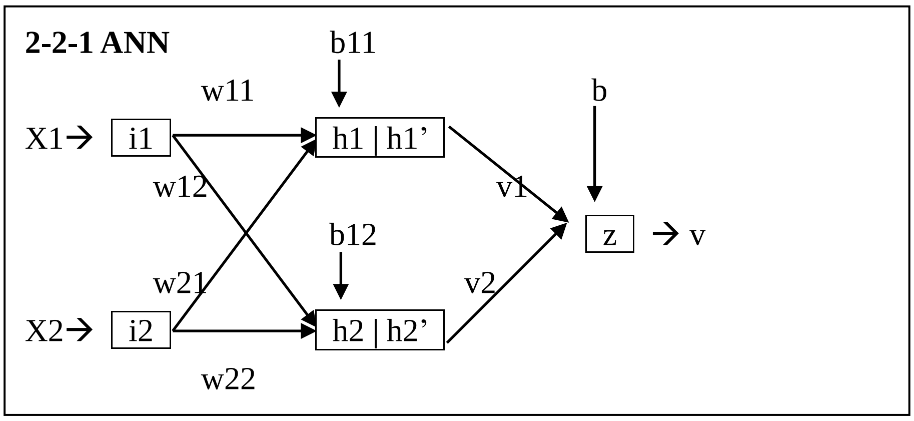
```
summary(fin.nn)                          # summary
a 6-2-1 network with 17 weights
options were - entropy fitting
  b->h1  i1->h1  i2->h1  i3->h1  i4->h1  i5->h1  i6->h1
 256.46    1.27    4.89  -26.89   -1.33   -5.91    0.76
  b->h2  i1->h2  i2->h2  i3->h2  i4->h2  i5->h2  i6->h2
   3.94  -58.21  -85.12    8.86    4.43  -11.26   52.11
   b->o   h1->o   h2->o
-164.64  -82.15  167.98

pred<-1*(fin.nn$fit>1/2)
table(pred,d1$HSI)               # classification table for training data d1
pred   0   1
   0 515   0
   1   1  28

pr<-predict(fin.nn,d2)           # apply saved output ann to testing data d2
pred<-1*(pr>1/2)
table(pred,d2$HSI)               # classification table for testing data d2
pred   0   1
   0 130   0
   1   2   4
```

7.6 Training the neural network using Backpropagation

As seen from the above output, training the neural network is to find the appropriate weight w such that the function $E=(Y-V)'(Y-V)$ is minimized using back-propagation algorithm. Let us illustrate it by a simple *2-2-1* ANN example with logistic output.



Suppose that we have 4 input vector, target vector and initial weights as follow:

$$X=\begin{pmatrix}0.4 & 0.7\\ 0.8 & 0.9\\ 1.3 & 1.8\\ -1.3 & -0.9\end{pmatrix}, Y=\begin{pmatrix}0\\0\\1\\0\end{pmatrix},\ W_1=\begin{pmatrix}b_{11} & w_{11} & w_{12}\\ b_{12} & w_{21} & w_{22}\end{pmatrix}=\begin{pmatrix}0.1 & -0.2 & 0.1\\ 0.4 & 0.2 & 0.9\end{pmatrix},\ W_2=\begin{pmatrix}b\\ v_1\\ v_2\end{pmatrix}=\begin{pmatrix}0.2\\ -0.5\\ 0.1\end{pmatrix}$$

First we compute the output from the ANN using the initial weights as follows:

```
> logistic<-function(x) {1/(1+exp(-x))}                  # define activiation function
> x<-matrix(c(0.4,0.7,0.8,0.9,1.3,1.8,-1.3,-0.9),ncol=2,byrow=T)   # input x
> x<-cbind(1,x)                                         # attach a column of ones
> y<-c(0,0,1,0)                                         # target value
> w1<-matrix(c(0.1,-0.2,0.1,0.4,0.2,0.9),byrow=T,nrow=2) # hidden layer weights
> w2<-c(0.2,-0.5,0.1)                                   # output layer weights

> h<-rbind(1,logistic(w1%*%t(x)))     # compute h'
> out<-logistic(w2%*%h)               # compute output value
> e1<-y-out                           # compute output error
> sse<-sum(e1^2)                      # compute sum of squared error
> e1
           [,1]       [,2]     [,3]       [,4]
[1,] -0.5034926 -0.5064967 0.490446 -0.4875784
> sse
[1] 0.9883136
```

Next we update the output layer weights *w2* and hidden layer weights *w1* by

```
> lr<-0.5                          # learning rate:η
> del2<-out*(1-out)*e1             # output layer:δ2 = out*(1-out)*e1
> del_w2<-2*lr*del2%*%t(h)         # compute Δw2 = 2 ηh'δ2
> new_w2<-w2+del_w2                # new output layer weights:w2 = w2+Δw2

> del1<-h*(1-h)*(w2%*%del2)        # hidden layer: δ1 = h' *(1-h' )*(δ2'w2)
> del1<-del1[c(2,3),]              #
> del_w1<-2*lr*del1%*%x            # compute Δw1 = 2 η δ1*x
> new_w1<-w1+del_w1                # new hidden layer weights:w1 = w1+Δw1

> err<-y-logistic(new_w2%*%rbind(1,logistic(new_w1%*%t(x))))  # new error
> sse<-sum(err^2)                  # final error
> sse
[1] 0.8330266
```

scale up the learning rate (*lr = 1.1*lr*) at next iteration if *SSE* decreases or scale down *lr* (*lr = 0.5*lr*) if *SSE* increases.

7.6.1 Mathematical details of Backpropagation algorithm

Let us consider a p-q-r ANN with logistics transfer function and logistics output.

Input to Hidden layer: $h_j = \sum_{i=1}^{p} w_{ji} x_i$, $j = 1,\ldots,q$

Hidden layer to output: $v_k = f(z_k)$, where $z_k = \sum_{j=1}^{q} w_{kj} f(h_j)$, $k = 1,\ldots,r$, where

f(x) is the transfer function.

The objective function needs to minimize is $E = (y-v)'(y-v)$.

Note that if the transfer function *f(x)* is logistic, then $f'(x) = f(x)[1-f(x)]$.

The Backpropagation algorithm is the gradient steepest descent method with variable-step length for minimizing the error function E.

1) Law of total derivative gives, symbolically

$$\Delta E = \frac{\partial E}{\partial w_1}\Delta w_1 + \frac{\partial E}{\partial w_2}\Delta w_2$$

2) The direction of $(\Delta w_2, \Delta w_2)$ that can maximize ΔE is along $\left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}\right)$, that is the reason why the method is called steepest descent.

The general algorithm for updating W is

$W^{(t+1)} = W^{(t)} + \Delta W^{(t)}$, $\Delta W^{(t)} = -\eta\,(\partial E/\partial W)|_{W^{(t)}}$, where η is the learning rate, which is kept at low level, so that no exaggerated step size would be taken. Also the negative sign means going inward as the vector $\left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}\right)$ is pointing outward.

(I) Recall that in Section 7.6, $\Delta W_2 = 2\eta\, h'\delta_2$, where $\delta_2 = e_1 \times out \times (1-out)$, also recall $e_1 = y - v$.
To see this, we first compute the gradient for the weights W_2 (from hidden layer to output).

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial z_k}\cdot\frac{\partial z_k}{\partial w_{kj}} = \frac{\partial E}{\partial v_k}\cdot\frac{\partial v_k}{\partial z_k}\cdot\frac{\partial z_k}{\partial w_{kj}} = -2(y-v)\ f'(z)\ f(h_j) = -2(y-v)\ f(z)(1-f(z))\ f(h_j)$$
$$= -2e_1 \times out \times (1-out) \times f(h_j)$$

Therefore, $\Delta W_2 = -\eta\left(\frac{\partial E}{\partial w}\right) = 2\eta\ e_1\ out(1-out)\ f(h_j) = 2\eta\ \delta_2\ f(h_j)$.

(II) Next recall that in Section 7.6, $\Delta W_1 = 2\eta\ \delta_1 x$, where $\delta_1 = h(1-h)\times(\delta_2 w_j)$. To see this, finally, we compute the gradient for the weights W_1 (from input to hidden layer).

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial f(h_j)}\cdot\frac{\partial f(h_j)}{\partial h_j}\cdot\frac{\partial h_j}{\partial w_{ji}}.$$

Note that $\frac{\partial E}{\partial f(h_j)} = \frac{\partial E}{\partial v}\cdot\frac{\partial v}{\partial z}\cdot\frac{\partial z}{\partial f(h_j)} = -2(y-v)\ f'(z)\ w_j = -2\delta_2 w_j$,

$$\frac{\partial f(h_j)}{\partial h_j} = h(1-h) \text{ and } \frac{\partial h_j}{\partial w_{ji}} = x_i .$$

Therefore, $\frac{\partial E}{\partial w_{ji}} = -2\delta_2 w_j \; h(1-h) \; x_i$ and $\Delta W_1 = -\eta \left(\frac{\partial E}{\partial w} \right) = 2\eta \; \delta_2 \; w_j \; h(1-h) \; x_i$.

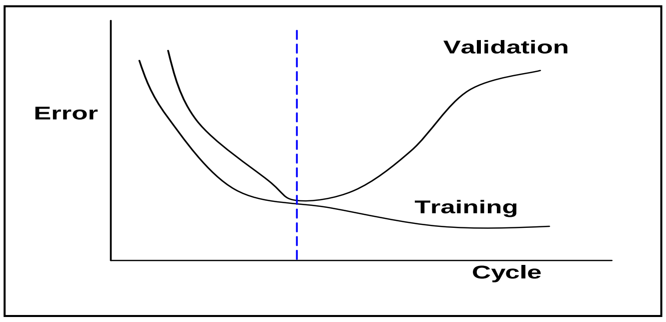
7.7 Practical considerations

Usually the training data are scaled before feeding into the ANN so that each input has equal importance (or of similar unit of measurement). The output or target data are also scaled if the output activation function has a limited range. There are two popular scaling methods:

1. Range: Scale the data *x* into the range [0, 1] by $x' = (x - \min) / (\max - \min)$ where *max* and *min* is the maximum and minimum of *x* respectively.
2. Standardize score: Scale the data *x* using standardize score $z = (x - \bar{x}) / s$.

In the last two examples, we have seen that a very simple single hidden layer with two neurons perform well in classification. It is mainly because there are many parameters we can choose to produce the outputs as close to the training data as possible. However, a very good fit to the training does not mean that it also has good prediction power on those unseen data. The network may recognize patterns-of-one by memorizing instead of generalizing the training data. A good indictor for this situation is that the network works well on training sample but not on the test sample. This is known as **over-fitting**.

Usually in data mining, the dataset is divided into training dataset and testing (validation) dataset. We can monitor the error sum of square from the training dataset as well as the prediction error sum of square from the testing dataset. A typical situation is as follows:



When we start training the ANN, both the error sum of square and the prediction error sum of square will decrease. We will stop the iteration when the prediction error sum of square starts to increase. The network is **over-trained** if it is beyond this point. In practice, there is some questions needed to consider:

1. Size of the hidden layer

The capacity of the network to recognize patterns increases with the number of neurons in the hidden layer. However, the drawback is there are too many parameters and the risk of over-fitting. The size of the hidden layer should never be more than twice as large as the input layer. A good strategy is to start with the same size as the input layer. If the network is over-fitted, reduce the size of the hidden layer. If the network is not sufficiently accurate, increase the size.

2. Number of hidden layer

Increase the number of hidden layer will increase the number of parameters, and the time needed to train the network. However, more hidden layer may be needed for multiple output values.

3. Number of inputs

Increase the number of inputs will increase the time needed to train the network and probably converge to an inferior solution. A hybrid approach is to use other techniques (such as principal component, classification tree) to select important input variables.

4. Size of the training sample and testing sample

Normally speaking, the size of the training sample should be at least 5 to 10 times the number of weights in the network and the size of the testing sample should be around 1% to 30% of the total sample size.

5. Linear output

Since the linear output in ANN gives real numbers, it can be used as a nonlinear regression model as well.

7.8 Neural Network using EXCEL

There is a good implementation of ANN using EXCEL found in the internet developed by Dr. Saha. The file *ch7-nn-iris.xls* contains the iris data. Inside the EXCEL file, there are seven separate sheets.

1. ReadMe sheet contains the basic information for using this file. Basically, this file can handle up to 10,000 observations and 50 predictor variables.

2. Data are stored in L24 of the Data sheet. We can specify which variable is output, continuous, categorical or omit. We can copy and paste your own data here to build our own classification tree. Note that there should be exactly 1 output variable in the data sheet.

3. In the User Input sheet, we can specify the parameters for the ANN: number of inputs, sample size, number of hidden layer and the size of hidden layer. Other parameters are related to the back- propagation algorithm for training the dataset. In particular, the initial weights are chosen randomly from a range [-w, w], the learning parameter in (0,1). If this parameter is too large, it may miss the global minimum. Other the other hand, if this is too small; it may trap in a local

minimum. In summary, there is no clear optimal strategy on choosing these parameters. We need to use trial-and-error approach to see their effect on the prediction error sum of square.

4. The Output sheet contains the graph of the error sum of square and the prediction error sum of square as well as the percentage of misclassification at each iteration.

5. In the Calc sheet, the final weights and the misclassification table are given.

6. In the Profile sheet, we can enter an input x and create the profile of the output as a function of a particular input variable.

7. In the Lift Chart sheet, the score of a particular output class is sorted in descending order and the percentage of correct class is plotted. This is another graphical method to show the accuracy of the ANN. See more detail in ReadMe sheet.

7.9 Conclusion

ANN provides an interesting model for classification. The strength and weakness are summarized as follow:

Strength:

- Neural networks are versatile, can be used for predicting continuous y as well.
- Can produce good results in complicated domains.
- Can handle continuous, ordinal and nominal data types.

Weakness:

- Work like a black box and cannot explain results.
- May converge to an inferior solution.
- Too many parameters need to use trial-and-error.

Reference:

Supplement 12A of Applied Multivariate Statistical Analysis, 5th ed., Richard Johnson and Dean Wichern, Prentice Hall.