Chapter 5 Artificial Neural Network

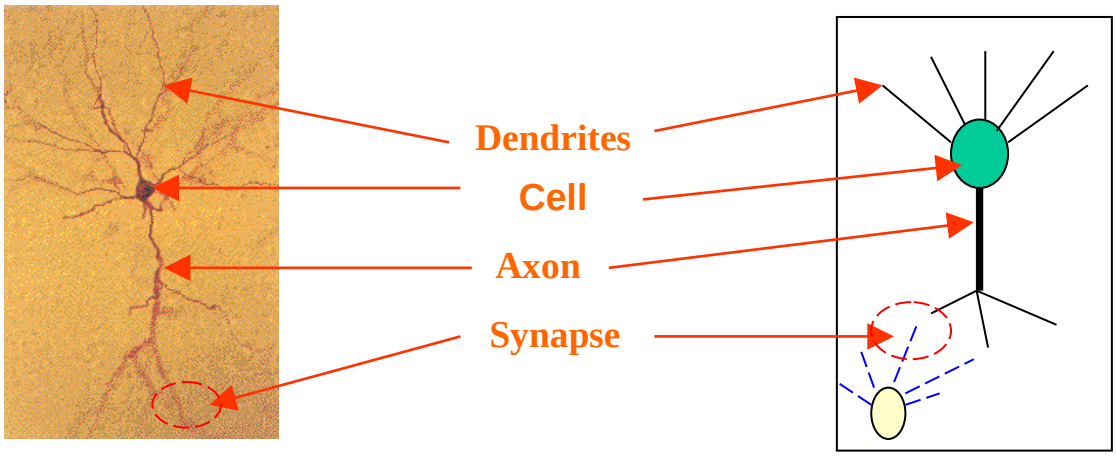
Artificial Neural Network (ANN) is an active area of research in Data Mining. This approach is to mimic the functions and mechanism of our brain. Although how our brain memorizes, recognizes and generalizes pattern is still a mystery, some aspects of the structure of brain are well-known at least to the neuroscientists. This provides a useful and workable model to build an ANN.

First, ANN was developed in the research laboratory. With the increase in the computing power and fast and efficient algorithms for ANN, it was tested in many areas of applications. Since then, ANN has many successful applications in engineering, business, finance, risk management, artificial intelligence etc.

In 1943, McCulloch (a neurophysiologist) and Pitts (a mathematician) 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 1982, Hopfield invented the associative neural network; and in 1986, Rumelhard, Hinton and Williams applied **back-propagation** algorithm to train neural networks. Since then, there are many successfully 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. Recent development of **Deep Learning**, which based on ANN has great achievements in many applications. This becomes the most active research area in Artificial Intelligence. Before we go into ANN, we need learn a bit about the structure our human brains.

5.1 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 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 complicate network. The exact function of these neurons is still a mystery but a very simple mathematical model that mimic these neurons provide a surprising good performance in pattern recognition, classification and prediction.

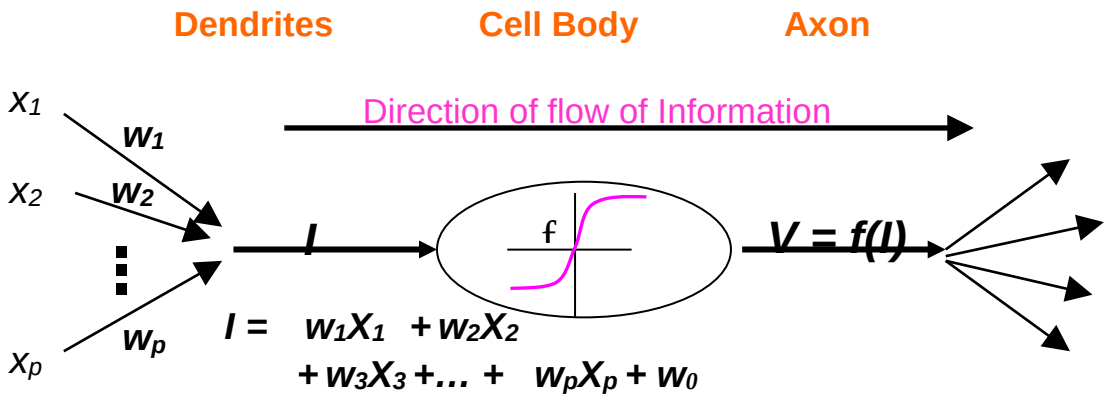


Hippocampal Neurons

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

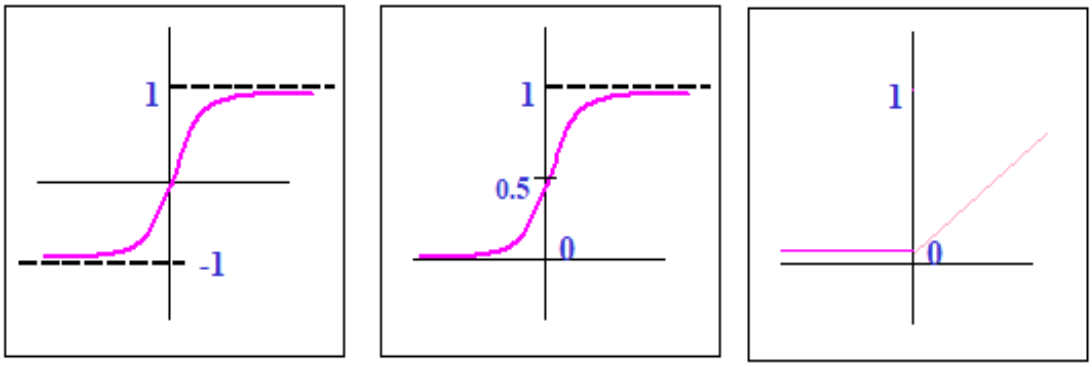
Schematic

Dendrites responsible for receiving information; cell body is for process information; axon carries 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:



$x_1,...,x_p$ are the inputs received from our neurons or environment. The total input I is formed from the linear combinations of these inputs with weights $w_0, w_1,..., w_p$. (w_0 is known as the bias). The **transfer** function (or **Activation** function) converts 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 cell body:



Tanh

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

Logistic

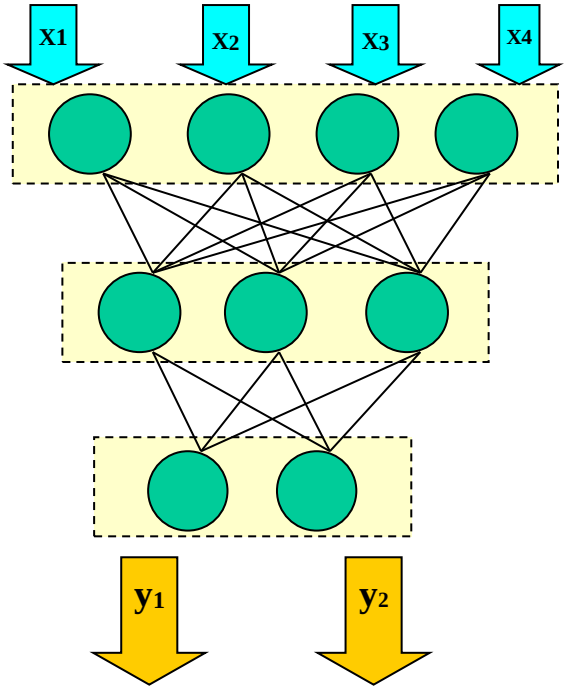
$$f(x) = e^{x}/(1+e^{x}) = 1/(1+e^{-x})$$

Rectificed Linear Unit (ReLU)

$$f(x) = \max(0, x)$$

5.2 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 layer 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 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.

5.3 ANN using R

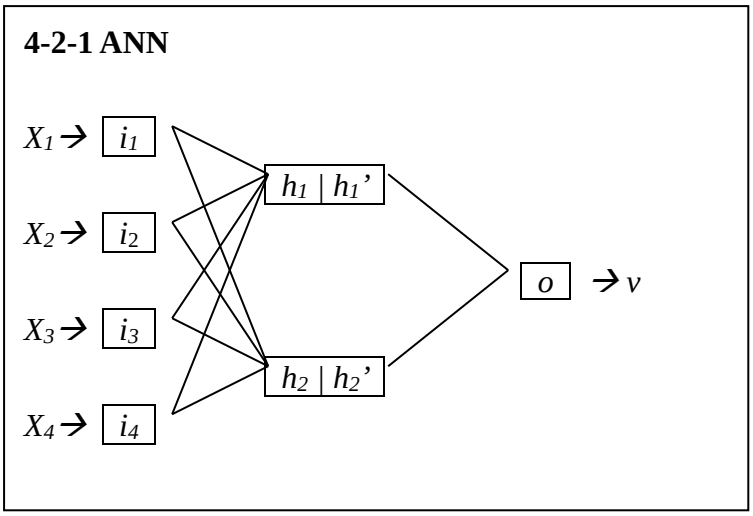
Given the input X and output Y in the training dataset, we have to find the w_{ij} such that the sum of square error $E = \sum_{i=1}^{n} (Y_i - V_i)^2$ is minimized, where V_i is the prediction from the ANN, that is $V_i = f(X_i)$. However, the algebraic form of V_i is too complicate to write down explicitly. 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 2 neurons in the hidden layer and linear output unit
> iris.nn<-nnet(Species~Sepal_len+Sepal_wid+Petal_len+Petal_wid,data=d,
    size=2,linout=T)
# 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
```

We can turn off the display of value in iter by adding the option `trace=F` in nnet() function as follows:

```
> iris.nn<-nnet(Species~Sepal_len+Sepal_wid+Petal_len+Petal_wid,data=d,
    size=2,linout=T,trace=F)
```

The above ANN is represented as the following diagram:



$$h_1 = -4.96 - 11.21x_1 - 14.22x_2 + 38.14x_3 + 20.95x_4$$
$$h_2 = 153.65 + 35.55x_1 + 37.28x_2 - 75.82x_3 - 64.98x_4$$
$$h_1' = \exp(h_1)/[1+\exp(h_1)] \qquad h_2' = \exp(h_2)[1+\exp(h_2)] \qquad (5.1)$$
$$v = 1.99 + h_1' - 0.99h_2'$$

Let us use the *1, 51* and *101* 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 (5.1),
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.

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

```
> pred<-round(iris.nn$fitted.values)   # prediction
> table(pred,d$Species)                # classification table
   pred
    1  2  3
  1 50  0  0
  2  0 49  0
  3  0  1 50
```

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. Therefore, we have to run *nnet()* several times and save the best 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
# The default option of trace is set to F

library(nnet)
ann<-function(x,y,size,maxit=100,linout=F,trace=F,try=5) {
  ann1<-nnet(y~.,data=x,size=size,maxit=maxit,linout=linout,trace=trace)
  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,trace=trace)
    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 results. For example:

```
> source("ann.r")                       # load ann()
> iris.nn<-ann(d[,1:4],d[,5],size=2,linout=T,try=10)     # ann with 10 times
> iris.nn$value                         # best value among 10 trials
[1] 0.980673
> summary(iris.nn)                      # summary of output
a 4-2-1 network with 13 weights
options were - linear output units
  b->h1  i1->h1  i2->h1  i3->h1  i4->h1
 100.34   20.63   14.97  -87.18 -139.94
  b->h2  i1->h2  i2->h2  i3->h2  i4->h2
-335.23 -340.06 -316.06  525.50  458.66
   b->o   h1->o   h2->o
   2.00   -1.00    0.98

> pred<-round(iris.nn$fitted.values)   # prediction
> table(pred,d$Species)                # classification table
   pred
     1  2  3
  1 50  0  0
  2  0 49  0
  3  0  1 50
```

Note that the best (smallest) objective function value among 10 trials is 0.980673.

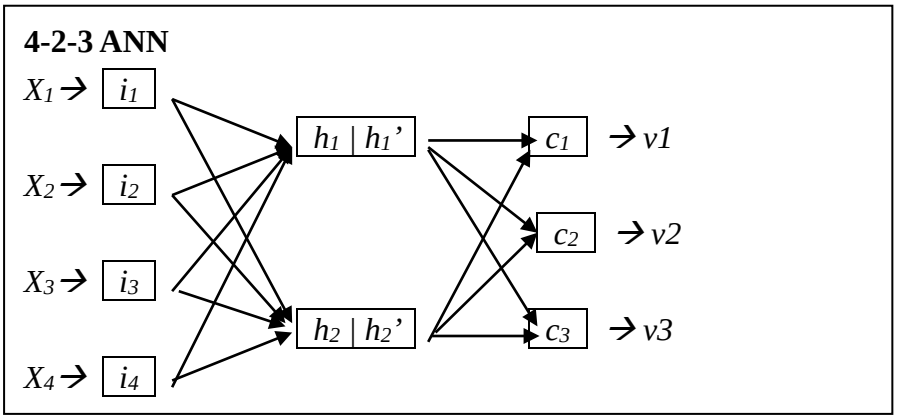
5.4 ANN with logistic output

We have seen the *nnet()* or *ann()* function with linear output option *linout=T* in R. This means that 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, we have to round the number to its nearest integer. However, the default option in R is *linout=F* and *k* different logistic functions are used to relate the neurons and the output. Let us look at how to use this option in *nnet()* or *ann()*.

```
> 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 we have to change *y* to a **factor object** in order to use *linout=F*. The fitted values in the output have *k* columns. Suppose that the fitted values is c_{ij}, for *j=1,…,k*. Then the probability of the *i*-th observation belongs to the the *j*-th group is $p_{ij} = \exp(c_{ij}) / \Sigma_{j=1}^{g} \exp(c_{ij})$. Normally speaking, we predict 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. Here we have only 1 error case out of 150 cases and the error rate is 0.67%!

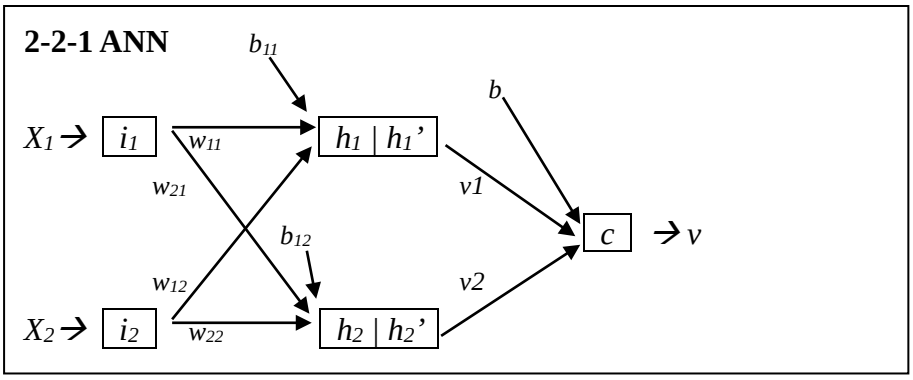
Since in the iris flower example, the output has three levels, the diagram for this ANN with logistic output is as follow:



Note that for logistic output $v_j = \text{logistic}(c_{ij}) = \Pr\{y_i = j\}$.

5.5 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.



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

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

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

```
> 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 W_2 and hidden layer weights W_1 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
```

Note that the new weights *new_w1* and *new_w2* are computed and the *SSE* decreases from 0.988 to 0.833. We can update the *w1* by *new_w1* and *w2* by *new_w2* and iterate until *SSE* is small or the maximum number of iteration exceeded. Note that we can 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.

Usually the training data are scaled before feeding into the ANN so that each input has equal importance. The *stand()* and *scale()* functions in Chapter 1 can be used to rescale the input *x* before using ANN.

5.6 ANN for prediction

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

```
> d<-read.csv("iris.csv")                              # read in data
> n<-nrow(d)                                           # get sample size
> r<-0.8                                               # get sampling ratio
> set.seed(12345)
> id<-sample(1:n,size=round(r*n),replace=F)            # generate id
> d1<-d[id,]                                           # training dataset
> d2<-d[-id,]                                          # testing dataset
> iris.nn<-ann(d1[,1:4],d1[,5],size=2,linout=T,maxit=200,try=10)
> iris.nn$value
[1] 0.9750302
> summary(iris.nn)
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 follow:

```
> 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(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
```

We can directly apply the built-in *predict()* function on the testing dataset *d2*.

```
> pr<-predict(iris.nn,d2)       # pr is a vector of length 30
> table(round(pr),d2[,5])       # round pr to nearest integer
     1  2  3
  1  9  0  0
  2  0  8  0
  3  0  1 12                    # error rate = 1/30 = 3.33%
```

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

```
> y<-as.factor(d1[,5])       # change y to factor
> iris.nn<-ann(d1[,1:4],y,size=2,maxit=200,try=10)
> iris.nn$value
[1] 0.8753883
> w1<-matrix(iris.nn$wts[1:10],nr=2,byrow=T)
> w2<-matrix(iris.nn$wts[11:19],nr=3,byrow=T)
> x<-cbind(1,d2[,1:4])       # form the matrix x, note that a column
> out1<-logistic(w1%*%t(x))  # This is the output h'
> out1<-rbind(1,out1)        # append a row of ones on top of out1
> out2<-t(w2%*%out1)         # compute fitted values
> pr<-max.col(out2)          # find the column corresponds to max
> table(pr,d2[,5])           # classification table
pr   1  2  3
  1  9  0  0
  2  0  8  0
  3  0  1 12
```

Or we can directly apply the *predict()* function to the testing dataset *d2* as follow:

```
> pr<-predict(iris.nn,d2)    # pr is a matrix 30x3
> dim(pr)
[1] 30  3
> pr<-max.col(pr)            # select col. index of max. in each row
> table(pr,d2[,5])           # classification table
pr   1  2  3
  1  9  0  0
  2  0  8  0
  3  0  1 12
```

Let us try the *HMEQ* example using columns *DEROG, DELINQ* and *DEBTINC* (columns *8,9,13*) which appears in *CTREE*. Note that y is binary, the fitted values is *Pr{Y=1}* instead of c_{ij} in the IRIS example. Hence it is a 3-3-1 ANN.

```
> d<-read.csv("hmeq1.csv")                          # read in dataset
> n<-nrow(d)                                        # get sample size
> r<-2/3                                            # set sampling ratio
> set.seed(123)                                     # set the random seed
> id<-sample(1:n,size=round(r*n),replace=F)         # generate id
> d1<-d[id,]                                        # training dataset
> d2<-d[-id,]                                       # testing dataset
> y1<-as.factor(d1[,1])                             # convert y to factor
> y2<-as.factor(d2[,1])
> source("ann.r")                                   # load ann()
> hmeq.nn<-ann(d1[,c(8,9,13)],y1,size=3,maxit=500,try=30) # use logistic output
> summary(hmeq.nn)                                  # summary of output
> hmeq.nn$value                                     # display best value
[1] 409.6365

> pred<-(hmeq.nn$fitted.values>0.5)          # fitted value is Pr{Y=1}
> table(pred,y1)                             # classification table for training data
       y1
pred       0    1
  FALSE 1867  109
  TRUE     1   68                           # training error rate = 110/2045 = 5.38%

> pr<-predict(hmeq.nn,d2)                    # predict on testing data
> pred<-(pr>0.5)
> table(pred,y2)                             # classification table for testing data
         y2
pred      0   1
  FALSE 920  57
  TRUE    1  44                             # testing error rate = 58/1022 = 5.68%
```

Now let us using standardize *x* before using ANN.

```
> source("stand.r")
> z<-stand(d[,c(8,9,13)])
> z1<-z[id,]
> z2<-z[-id,]
> hmeq.nn<-ann(z1,y1,size=3,maxit=500,try=30)   # use logistic output
> pred<-round(hmeq.nn$fitted.values)  # prediction
> summary(hmeq.nn)                    # summary of output
> hmeq.nn$value                       # display best value
[1] 409.6376
> table(pred,y1)                      # classification table for training data
       y1
pred    0    1                       # training error rate = 111/2045 = 5.43%
   0 1866  109
   1    2   68
> pr<-predict(hmeq.nn,z2)             # predict on testing data
> pred<-(pr>0.5)
> table(pred,y2)                      # classification table for testing data
       y2
pred      0   1                     # testing error rate = 58/1022 = 5.68%
  FALSE 920  57
  TRUE    1  44
```

Finally, we can try the rescale *x* into [*0,1*] before using ANN.

```
> source("scale.r")
> z<-scale.con(d[,c(8,9,13)])
> z1<-z[id,]
> z2<-z[-id,]
> hmeq.nn<-ann(z1,y1,size=3,maxit=500,try=30)   # use logistic output
> hmeq.nn$value                                 # display best value
> hmeq.nn$value                                 # display best value
[1] 409.6626
> summary(hmeq.nn)                              # summary of output
> pred<-round(hmeq.nn$fitted.values)            # prediction
> table(pred,y1)                  # classification table for training data
       y1
pred      0    1                  # training error rate = 110/2045 = 5.38%
  FALSE 1867  109
  TRUE     1   68
> pr<-predict(hmeq.nn,z2)         # predict on testing data
> pred<-(pr>0.5)
> table(pred,y2)                  # classification table for testing data
        y2
pred      0   1                   # testing error rate = 58/1022 = 5.68%
  FALSE 920  57
  TRUE    1  44
```

Although all the variables in Titanic dataset are categorical, we can recode them into numeric values and fit them using ANN. The following function *codechr()* is to recode an input character vector *v* into numeric vector. The weights in ANN will be trained and adjusted to give reasonable prediction.

```
# function to recode character vector v to numeric vector
codechr<-function(v) {
  n<-length(v)                          # length of v
  h<-levels(v)                          # store levels in v to h
  k<-length(h)                          # no. of categories in v
  p<-outer(v,h,"==")                    # nxk matrix of logical values
  q<-matrix(1:k,nrow=n,ncol=k,byrow=T)  # nxk matrix, each row is 1:k
  apply(p*q,1,sum)                      # output numeric code
}
```

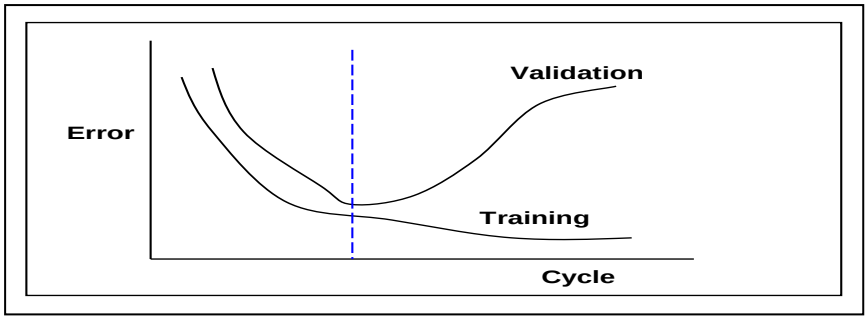
```
> d<-read.csv("titanic.csv")      # read in data
> n<-nrow(d)                      # get sample size
> v1<-d[,1]                       # recode column 1 to x1
> x1<-codechr(v1)
> v2<-d[,2]                       # recode column 2 to x2
> x2<-codechr(v2)
> v3<-d[,3]                       # recode column 3 to x3
> x3<-codechr(v3)
> v4<-d[,4]                       # recode column 4 to x4
> x4<-codechr(v4)
> x<-cbind(x1,x2,x3,x4)           # combine column-wise to form x
> set.seed(12345)                 # set random seed
> r<-0.9                          # set sampling ratio
> id<-sample(1:n,round(r*n))      # generate id
> d1<-x[id,]                      # training data
> d2<-x[-id,]                     # testing data

# linear output
> titan.nn<-ann(d1[,1:3],d1[,4],size=3,linout=T,maxit=200,try=20)
> titan.nn$value                  # display best value
[1] 303.7198

> pr<-round(titan.nn$fit)         # prediction for training data
> table(pr,d1[,4])                # classification table for training data
pr     1    2                     # training error rate=(395+19)/1981=20.90%
>   1 1322  395
  2   19    245

> pr<-predict(titan.nn,d2)        # prediction for testing data
> table(round(pr),d2[,4])         # classification table for testing data
pr    1   2                       # testing error rate=47/220=21.36%
  1 148  46
  2   1  25
```

5.7 Practical considerations

Usually the training data are scaled before feeding into the ANN so that each input has equal importance. 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 $x' = (x - \min)/(\max - \min)$
   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 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**. 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 follow:



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 need considering:

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 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.

5.8 Conclusion

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

Strength:

- Neural networks are flexible, can be used for modeling continuous y as well.
- Can produce good results in complicated domains.
- Can handles 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 be trial-and-error.

Reference:
Chapter 5 of Introduction to Data Mining by Tan, Steinbach and Kumar, Addison Wesley.

Appendix: Mathematical details of Backpropagation algorithm

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

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

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

and $f(x)$ 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 descent method with variable step length for minimizing E. 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.

Recall that in section 5.5, $\Delta W_2 = 2\eta \, h' \delta_2$ where $\delta_2 = e_1 \times out \times (1 - out)$.

First, we compute the gradient for the weights W_2 (from hidden layer to output).

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

Therefore $\Delta W_2 = 2\eta \; e_1 \; out(1 - out) \; f(h_j) = 2\eta \; \delta_2 \; f(h_j)$.

Next recall that in section 5.5, $\Delta W_1 = 2\eta \; \delta_1 x$ where $\delta_1 = h(1 - h) \times (\delta_2 w_j)$.

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

$\frac{\partial f(h_j)}{\partial h_j} = h(1 - h)$ 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 $\Delta W_1 = 2\eta \; \delta_2 \; w_j \; h(1 - h) \; x_i$.