11/10/16

Implementation of Neural networks and NN Commands Using R Programming

Authors,

Aswath Shakthi(117014012)

P Hema prasaath(117014032)

M A Sundeep(117014100)

K Natraj(117014062)

Aru.Ranjith(117014076)

Contents

1 Introduction:

In machine learning and cognitive science, an artificial neural network (ANN) is a network inspired by biological neural networks (the central nervous systems of animals, in particular the brain) which are used to estimate or approximate functions that can depend on a large number of inputs that are generally unknown.

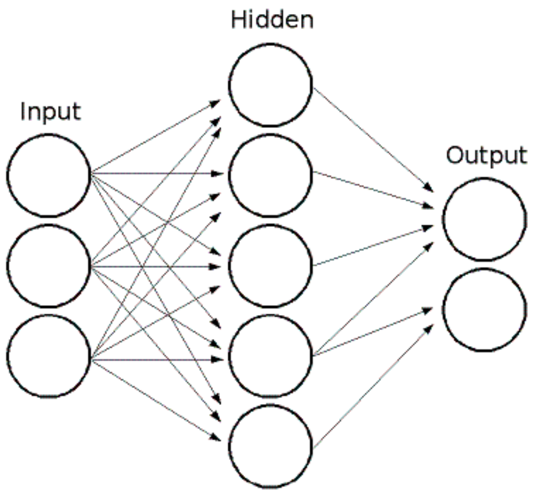
1.1. Specifications:

Artificial neural networks are typically specified using three things:

1) **Architecture**: specifies what variables are involved in the network and their topological relationships—for example the variables involved in a neural network might be the weights of the connections between the neurons, along with activities of the neurons

2) **Activity Rule**: Most neural network models have short time-scale dynamics: local rules define how the activities of the neurons change in response to each other. Typically the activity rule depends on the weights (the parameters) in the network.

3) **Learning Rule:** The learning rule specifies the way in which the neural network's weights change with time. This learning is usually viewed as taking place on a longer time scale than the



time scale of the dynamics under the activity rule. Usually the learning rule will depend on the activities of the neurons. It may also depend on the values of the target values supplied by a teacher and on the current value of the weights.

1.2 Background

Examinations of humans central nervous systems inspired the concept of artificial neural networks. In an artificial neural network, simple artificial nodes, known as "neurons", "neurodes", "processing elements" or "units", are connected together to form a network which mimics a biological neural network.

There is no single formal definition of what an artificial neural network is. However, a class of statistical models may commonly be called "neural" if it possesses the following characteristics:

1. contains sets of adaptive weights, i.e. numerical parameters that are tuned by a learning algorithm
2. is capable of approximating non-linear functions of their inputs.

2. Exploring the packages-Training, Validating and Testing

In RStudio, there are several packages(CRAN) for implementing Neural Networks.Some of the packages are **Neuralnet**, **nnet**, **NeuralNetTools**, **Neural**.The upcoming Topics will explain on the above packages and the commands used for NN implementation.

Packages are available on,

https://cran.r-project.org/web/packages/neuralnet/index.html

https://cran.r-project.org/web/packages/NeuralNetTools/index.html

https://cran.r-project.org/web/packages/nnet/index.html

Now, we will see the commands and function that are available in the packages.

1. Plotnet:

Description:

Plots a neural interpretation diagram for a neural network object.

Syntax:

```
plot(mod)
```

Example:

```
XOR <- c(0,1,1,0)
xor.data <- data.frame(expand.grid(c(0,1), c(0,1)), XOR)
print(net.xor <- neuralnet( XOR~Var1+Var2, xor.data, hidden=2, rep=5))
plot(net.xor, rep="best")
```

Output:

```
 neuralnet(formula = XOR ~ Var1 + Var2, data = xor.data, hidden = 2,
   rep = 5)

$response
  XOR
1   0
2   1
3   1
4   0
$covariate
     [,1] [,2]
[1,]    0    0
[2,]    1    0
[3,]    0    1
[4,]    1    1

$weights
$weights[[1]]
$weights[[1]][[1]]
                [,1]         [,2]
[1,] -0.001616656726 -1.303218129
[2,]  4.270201045089 -3.466875427
[3,]  4.344267708625 -3.128125885

$weights[[1]][[2]]
              [,1]
[1,]  0.1313163203
[2,]  0.5386353240
[3,] -1.6855128795
```

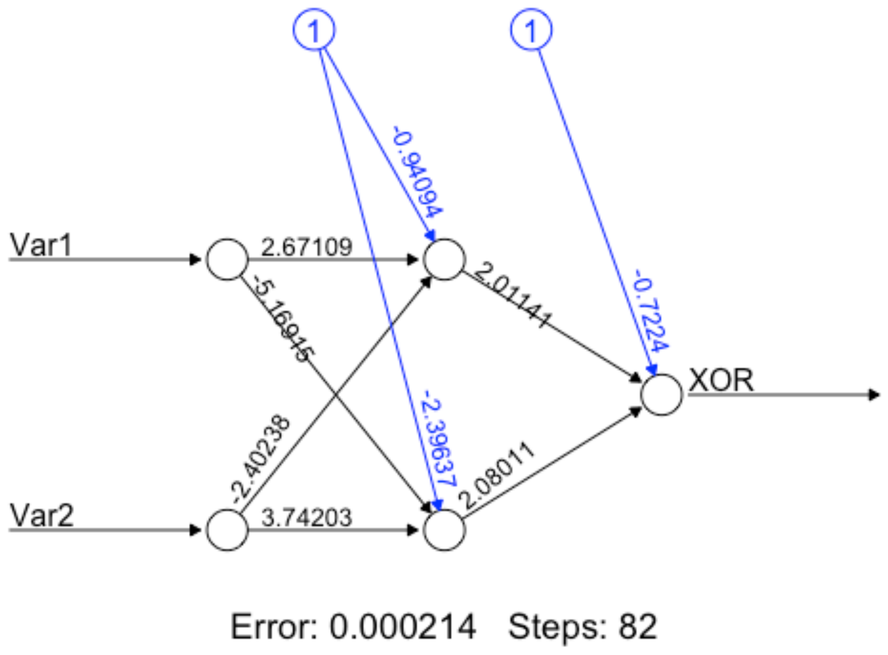
```
$weights[[2]]
$weights[[2]][[1]]
            [,1]         [,2]
[1,]  0.643416028 -0.8914476782
[2,] -2.277201570 -6.3303151357
[3,]  7.016143762 -6.4649093613

$weights[[2]][[2]]
            [,1]
[1,]  1.080370858
[2,] -0.578579946
[3,] -2.406317811

$weights[[3]]
$weights[[3]][[1]]
           [,1]        [,2]
[1,] 1.137282028 -1.170456552
[2,] 3.216831548 -3.549350875
[3,] 4.060329733 -3.046803268

$weights[[3]][[2]]
             [,1]
[1,] -0.3525517362
[2,]  1.0295508865
[3,] -1.6600933546

$weights[[4]]
$weights[[4]][[1]]
             [,1]        [,2]
[1,] -0.9409379672 -2.396368011
[2,]  2.6710865479 -5.169147503
[3,] -2.4023788114  3.742029232
```

```
$weights[[4]][[2]]
             [,1]
[1,] -0.7223993772
[2,]  2.0114148692
[3,]  2.0801091748

$weights[[5]]
$weights[[5]][[1]]
            [,1]          [,2]
[1,] -1.562557942 -0.7500639113
[2,]  1.436865952 -5.4162620655
[3,] -4.693166211 -4.1210846056

$weights[[5]][[2]]
             [,1]
[1,]  0.5022176801
[2,]  1.0479792926
[3,] -2.0661751538

$startweights
$startweights[[1]]
$startweights[[1]][[1]]
            [,1]          [,2]
[1,] 0.3051826224 -0.5278849826
[2,] 1.0667239251 -0.9668754271
[3,] 0.7758677086  0.9718741155

$startweights[[1]][[2]]
             [,1]
[1,] -0.3392012874
[2,] -0.1055185457
[3,] -2.0597366015
```

```
$startweights[[2]]
$startweights[[2]][[1]]
             [,1]          [,2]
[1,] -0.1259039166  0.08930106245
[2,] -1.9601989914  0.36968486433
[3,]  1.4477437618 -1.29765308325

$startweights[[2]][[2]]
             [,1]
[1,]  0.7569447669
[2,] -1.0594044834
[3,] -2.7115579103

$startweights[[3]]
$startweights[[3]][[1]]
            [,1]          [,2]
[1,] 2.4639591476 -0.79387501935
[2,] 0.1168315477 -1.31187375528
[3,] 1.2919297334  0.05319673243

$startweights[[3]][[2]]
             [,1]
[1,] -0.6236926186
[2,]  0.7517275041
[3,] -1.6905249511

$startweights[[4]]
$startweights[[4]][[1]]
             [,1]         [,2]
[1,] -0.7090210663 -0.3355880404
[2,] -0.9394067611 -0.9498257938
[3,]  0.1612567086  0.7205505954
```

```
$startweights[[4]][[2]]
            [,1]
[1,] -1.6486360555
[2,]  0.6813600422
[3,] -0.4199814469
$startweights[[5]]
$startweights[[5]][[1]]
            [,1]         [,2]
[1,] -1.3635011286 -0.81489553133
[2,]  1.0982032067 -1.64786206550
[3,] -0.5931662108 -0.02108460565
$startweights[[5]][[2]]
            [,1]
[1,]  0.2157338233
[2,] -0.1525558499
[3,] -0.9035428978
$generalized.weights
$generalized.weights[[1]]
          [,1]         [,2]
1 40.20052560065 37.98107315691
2  0.35106927255  0.33257229551
3  0.42287902321  0.39617035291
4  0.01168482417  0.01076030378

$generalized.weights[[2]]
            [,1]           [,2]
1 2622.72531226261 1747.71467839865
2   12.23469411242  -34.77788826992
3    0.04133871539    0.03203317717
4    0.02407680836   -0.07389720328

$generalized.weights[[3]]
          [,1]         [,2]
```

```
1 51.04723080224 51.31879584213
2  0.40888994593  0.42457938815
3  0.44872166906  0.41722156062
4  0.01473904657  0.01400505476

$generalized.weights[[4]]
            [,1]          [,2]
1   16.86146018  -24.43914575
2   54.09200126  -48.72790826
3 -470.40031359  331.31223592
4  505.81219040 -475.36813715

$generalized.weights[[5]]
             [,1]            [,2]
1 129.29472177965   56.0580345463
2  36.59187733369 -110.8656417780
3   0.34959667560    0.2195938483
4   0.04941377772   -0.1552557071
```



2.Garson:

Description:

The garson function uses Garson's algorithm to evaluate relative variable importance. This function identifies the relative importance of explanatory variables for a single response variable by deconstructing the model weights. The importance of each variable can be determined by identifying all weighted connections between the layers in the network. That is, all weights connecting the specific input node that pass through the hidden layer to the response variable are identified.

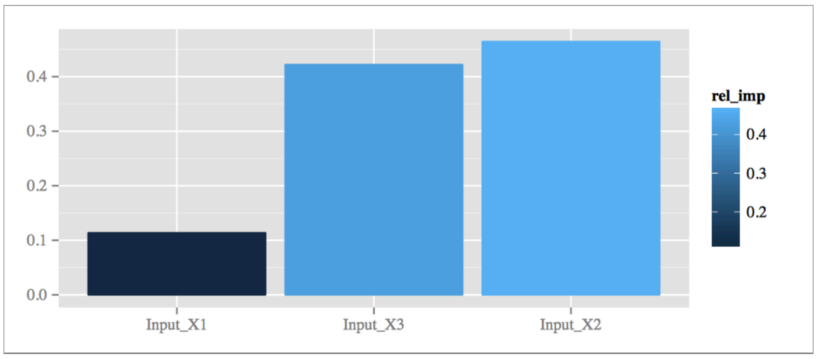
Syntax:

```
garson(mod)
```

Example:

```
1. library(nnet)
   data(neuraldat)
 set.seed(123)
 mod <- nnet(Y1 ~ X1 + X2 + X3, data = neuraldat, size = 5)
 garson(mod)
2. library(RSNNS)
   data(neuraldat)
   x <- neuraldat[, c('X1', 'X2', ‘X3’)].
   y <- neuraldat[, 'Y1']
   mod <- mlp(x, y, size = 5)
   garson(mod, ‘Y1')
```

Output:



The algorithm currently only works for neural networks with one hidden layer and one response variable.

3. Olden:

Description:

The olden function is an alternative and more flexible approach to evaluate variable i importance. The function calculates iportance as the product of the raw input-hidden and hidden-output connection weights between each input and output neuron and sums the product across all hidden neurons.

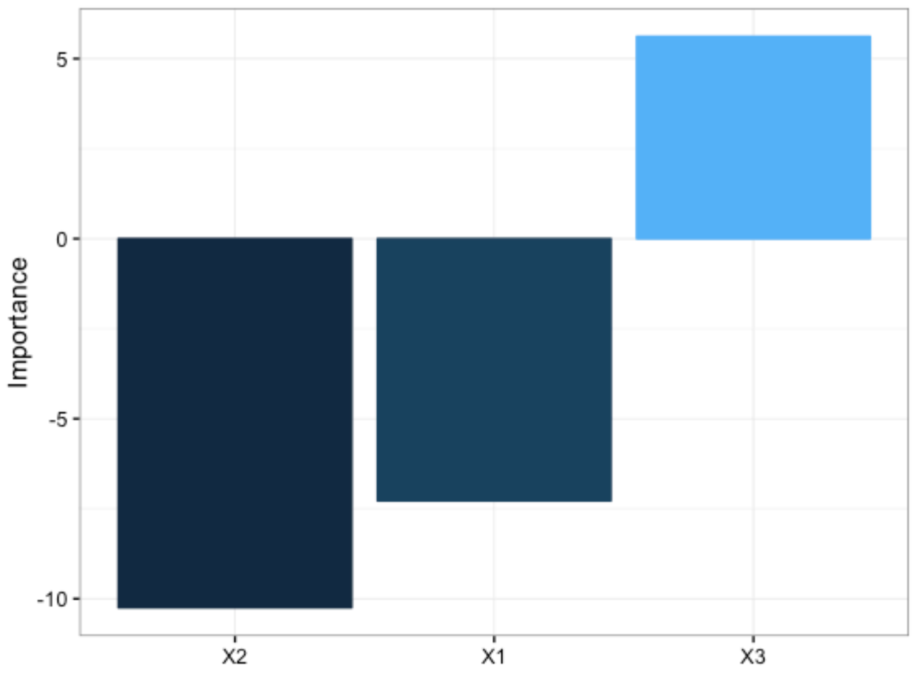
Syntax:

```
olden(mod)
```

Example:

```
library(nnet)
data(neuraldat)
set.seed(123)
mod <- nnet(Y1 ~ X1 + X2 + X3, data = neuraldat, size = 5)
olden(mod)
```

Output:



This ‘Olden’ method calculates variable importance as the product of the raw input-hidden and hidden-output connection weights between each input and output neuron and sums the product across all hidden neurons.An additional advantage is that Olden’s algorithm is capable of evaluating neural networks with multiple hidden layers wheras Garson’s was developed for networks with a single hidden layer.

4.lekprofile:

Description:

Creates optional barplot of constant values of each variable for each group used with lekprofile.It Conducts a sensitivity analysis of model responses in a neural network to input variables using Lek’s profile method

Syntax:

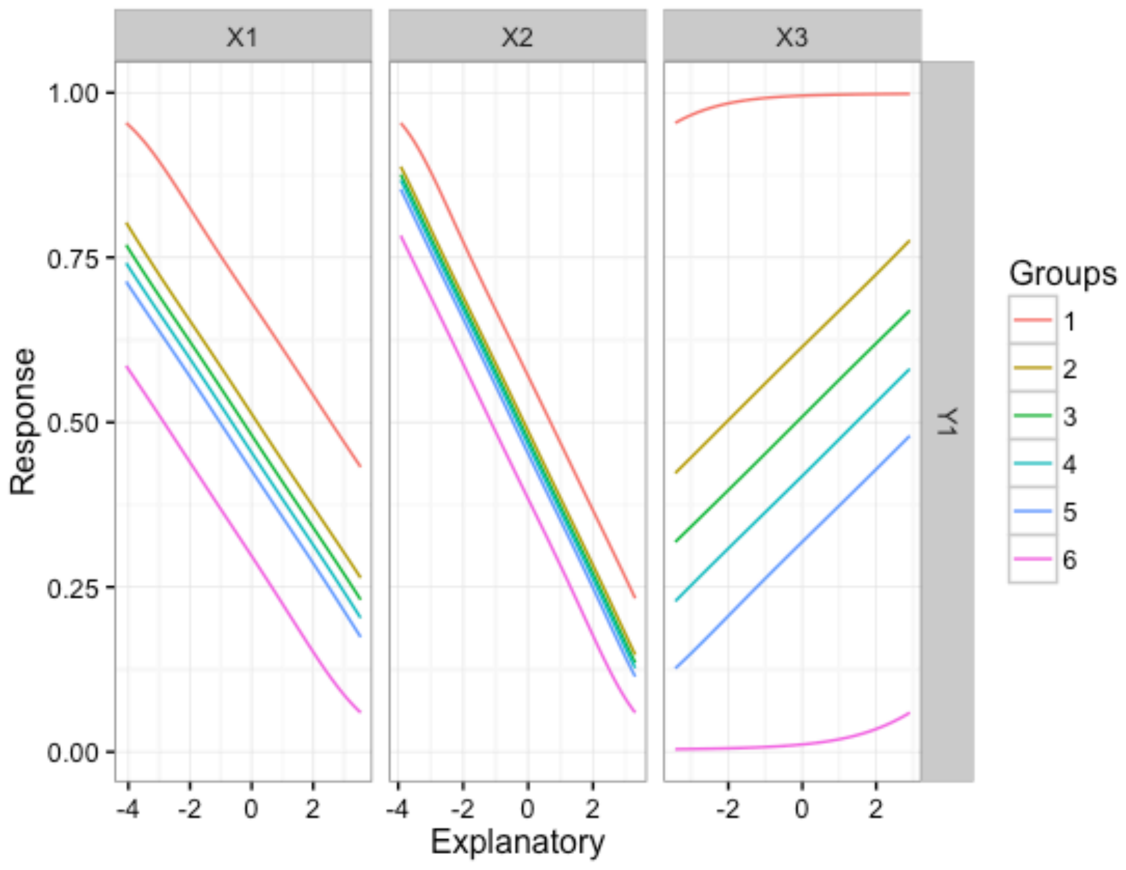
lekprofile(mod)

Example:

```
library(nnet)
set.seed(123)
mod <- nnet(Y1 ~ X1 + X2 + X3, data = neuraldat, size = 5)
lekprofile(mod)
```

Output:

```
# weights:  26
initial  value 259.012592
iter  10 value 0.986480
iter  20 value 0.225311
iter  30 value 0.139585
iter  40 value 0.098961
iter  50 value 0.038200
iter  60 value 0.022839
iter  70 value 0.013774
iter  80 value 0.008530
iter  90 value 0.005172
iter 100 value 0.003044
final  value 0.003044
stopped after 100 iterations
```



6.NeuralSkips:

Description:

Get weights for the skip layer in a neural network, only valid for networks created using skip = TRUE with the nnet function.

syntax:

```
neuralskips(mod)
```

Example:

```
library(nnet)
mod <- nnet(Y1 ~ X1 + X2 + X3, data = neuraldat, size = 5, linout = TRUE,
 skip = TRUE)
```

```
neuralskips(mod)
```

Output:

```
# weights:  29
initial  value 4492.774979
iter  10 value 4.858765
iter  20 value 0.010940
final  value 0.000057
converged
$`out 1`
[1] -0.07181016884 -0.10195449983  0.05379557297
```

7. **Neuralweights**:

Description:

Gets weights for a neural network in an organized list by extracting values from a neural network object. This function is generally not called by itself.

Syntax:

```
neuralweights(mod)
```

Example:

```
library(nnet)
mod <- nnet(Y1 ~ X1 + X2 + X3, data = neuraldat, size = 5, linout = TRUE)
neuralweights(mod)
```

Output:

```
# weights:  26
initial  value 1508.888597
iter  10 value 32.525118
iter  20 value 1.521364
iter  30 value 0.228338
iter  40 value 0.010819
iter  50 value 0.001629
iter  60 value 0.001381
iter  70 value 0.000676
iter  80 value 0.000271
iter  90 value 0.000170
final  value 0.000095
converged
$struct
[1] 3 5 1
$wts
$wts$`hidden 1 1`
[1]  2.3485980554 -0.4121534383  0.0511177694  0.1809830376
$wts$`hidden 1 2`
[1]  1.49351874669 -0.10923950731 -0.15712541398  0.08586107554
$wts$`hidden 1 3`
[1]  0.1677368698 -1.6618393780  0.2674810175  4.7864726191
$wts$`hidden 1 4`
[1]  1.3360278549  0.1920666105  0.2783487277 -0.1514935229
$wts$`hidden 1 5`
[1]  2.667134626 -3.788926911 -5.133233599  2.546932271
$wts$`out 1`
[1] -0.97662625116642 -0.00125014801061  2.62204687687865 -0.00002107152938
[5] -0.87322025857395  0.00053412858842
```

8. Pred_sens:

Description:

Get predicted values for Lek Profile method, used iteratively in lekprofile.Gets predicted output for a model’s response variable based on matrix of explanatory variables that are restricted following Lek’s profile method. The selected explanatory variable is sequenced across a range of values. All other explanatory variables are held constant at the values in grps.

Syntax:

```
pred_sens(mod)
```

Example:

```
library(nnet)
data(neuraldat)
```

```
set.seed(123)
mod <- nnet(Y1 ~ X1 + X2 + X3, data = neuraldat, size = 5)
mat_in <- neuraldat[, c('X1', 'X2', 'X3')]
grps <- apply(mat_in, 2, quantile, seq(0, 1, by = 0.2))
pred_sens(mat_in, mod, 'X1', 100, grps, 'Y1')
```

Output:

```
$`1`
         Y1          x_vars
1 0.9541241669  -4.0752252496
2 0.7004218869  -0.2676721922
3 0.4321254163   3.5398808652
$`2`
         Y1        x_vars
1 0.8031473629 -4.0752252496
2 0.5318837441 -0.2676721922
3 0.2639300861  3.5398808652
$`3`
         Y1        x_vars
1 0.7692306575 -4.0752252496
2 0.4996430559 -0.2676721922
3 0.2308000994  3.5398808652
$`4`
         Y1        x_vars
1 0.7416347968 -4.0752252496
2 0.4730633951 -0.2676721922
3 0.2030688883  3.5398808652
$`5`
         Y1        x_vars
1 0.7134240472 -4.0752252496
2 0.4454745064 -0.2676721922
3 0.1741574249  3.5398808652
```

```
$`6`
          Y1        x_vars
1 0.58556437663 -4.0752252496
2 0.31615855023 -0.2676721922
3 0.05948843882  3.5398808652
```

Gets predicted output for a model's response variable based on matrix of explanatory variables that are restricted following Lek's profile method. The selected explanatory variable is sequenced across a range of values. All other explanatory variables are held constant at the values in grps.

9.Lekgrps:

Description:

Create optional barplot of constant values of each variable for each group used with lekprofile

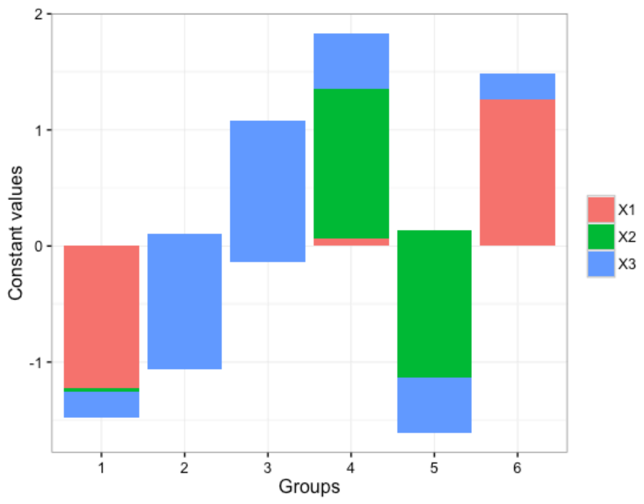
Syntax:

```
lekgrps(grps)
```

Example:

```
x <- neuraldat[, c('X1', 'X2', 'X3')]
grps <- kmeans(x, 6)$center
lekgrps(grps)
```

Output:



10. Compute:

Description :

compute, a method for objects of class nn, typically produced by neuralnet. Computes the outputs of all neurons for specific arbitrary covariate vectors given a trained neural network. Please make sure that the order of the covariates is the same in the new matrix or dataframe as in the original neural network.

Syntax:

```
compute(x, covariate, rep = 1)
```

Value:

compute returns a list containing the following components:

neurons - a list of the neurons' output for each layer of the neural network. net.result - a matrix containing the overall result of the neural network.

Example:

```
Var1 <- runif(5, 0, 10)
 sqrt.data <- data.frame(Var1, Sqrt=sqrt(Var1))
print(net.sqrt <- neuralnet(Sqrt~Var1,  sqrt.data, hidden=3,threshold=0.01))
compute(net.sqrt, (1:3)^2)$net.result
```

Output:

```
  neuralnet(formula = Sqrt ~ Var1, data = sqrt.data, hidden = 3,threshold = 0.01)
$response
       Sqrt
1 2.5147353194
2 2.0142846722
3 0.9708523607
4 2.9363159277
5 2.2308830814


$covariate
         [,1]
```

```
[1,] 6.3238937268
[2,] 4.0573427407
[3,] 0.9425543062
[4,] 8.6219512275
[5,] 4.9768393231

$model.list
$model.list$response
[1] "Sqrt"


$model.list$variables
[1] "Var1"
$err.fct
    function (x, y)
   {
       1/2 * (y - x)^2
   }
attr(,"type")
[1] "sse"
$act.fct
function (x)
{
   1/(1 + exp(-x))
}
attr(,"type")
[1] "logistic"
$linear.output
[1] TRUE


$data
        Var1         Sqrt
1 6.3238937268 2.5147353194
2 4.0573427407 2.0142846722
```

```
3 0.9425543062 0.9708523607
4 8.6219512275 2.9363159277
5 4.9768393231 2.2308830814

$net.result
$net.result[[1]]
         [,1]
1 2.5534751892
2 1.9890062237
3 0.9658478552
4 2.9074554951
5 2.2442259157



$weights
$weights[[1]]
$weights[[1]][[1]]
             [,1]         [,2]          [,3]
[1,]  1.1439011338 -0.6599525328 -1.1782906091
[2,] -0.3491588206  0.3735797729 -0.2257163831

$weights[[1]][[2]]
            [,1]
[1,]  1.283941114
[2,] -1.115743857
[3,]  1.999042381
[4,] -1.965541666

$startweights
$startweights[[1]]
$startweights[[1]][[1]]
             [,1]        [,2]          [,3]
[1,]  1.1977435006 -0.398656002 -0.7492694485
```

```
[2,] -0.1984836726  1.055843847  0.1909967384


$startweights[[1]][[2]]
             [,1]
[1,] -0.8775103110
[2,] -1.6529608274
[3,] -0.2787727242
[4,] -2.5114378264
$generalized.weights
$generalized.weights[[1]]
           [,1]
1 -0.05043846166
2 -0.15007486538
3 10.18593862642
4 -0.02033721187
5 -0.09271282224


$result.matrix
                                1
error                0.001587889173
reached.threshold      0.009159657113
steps              99.000000000000
Intercept.to.1layhid1  1.143901133774
Var1.to.1layhid1      -0.349158820570
Intercept.to.1layhid2 -0.659952532777
Var1.to.1layhid2       0.373579772903
Intercept.to.1layhid3 -1.178290609118
Var1.to.1layhid3      -0.225716383076
Intercept.to.Sqrt      1.283941114247
1layhid.1.to.Sqrt     -1.115743856985
1layhid.2.to.Sqrt      1.999042380808
1layhid.3.to.Sqrt     -1.965541665790
```

```
attr(,"class")
[1] "nn"
> compute(net.sqrt, (1:3)^2)$net.result
           [,1]
[1,] 0.9851683844
[2,] 1.9720191967
[3,] 2.9479268252
```

11.confidence.interval

Description :

confidence.interval is a method for objects of class nn, typically produced by neuralnet. Calculates confidence intervals of the weights (White, 1989) and the network information criteria NIC (Murata et al. 1994). All confidence intervals are calculated under the assumption of a local identification of the given neural network. If this assumption is violated, the results will not be reasonable.

Syntax:

confidence.interval(x, alpha = 0.05)

x is a Neural Network and alpha is set of confidence levels to (1-alpha)

Values:

confidence.interval returns a list containing the following components:

lower.ci - a list containing the lower confidence bounds of all weights of the neural net- work differentiated by the repetitions.

upper.ci - a list containing the upper confidence bounds of all weights of the neural net- work differentiated by the repetitions.

Nic - a vector containg the information criteria NIC for every repetition.

Example:

```
data(infert, package="datasets"

print(net.infert <- neuralnet(case~parity+induced+spontaneous,

infert, err.fct="ce", linear.output=FALSE))
```

```
confidence.interval(net.infert)
```

Output:

$data has not been listed, due to large amount of data

# Lower.ci is the lower confidence bound of all weight of NN differentiated by repitition.

# Upper.ci is the upper confidence bound of all weight of NN differentiated by repitition.

```
attr(,"class")
[1] "nn"
> confidence.interval(net.infert)
$lower.ci
$lower.ci[[1]]
$lower.ci[[1]][[1]]
             [,1]
[1,]  0.2345640490
[2,] -0.7567348448
[3,] -5.6856487157
[4,] -8.7680906435
  $lower.ci[[1]][[2]]
             [,1]
[1,] -0.7294807748
[2,] -6.4404053437
```

```
$upper.ci

$upper.ci[[1]]

$upper.ci[[1]][[1]]

            [,1]

[1,] 2.8601519305

[2,] 4.5377684856

[3,] 0.7721780638

[4,] 0.9780033487

$upper.ci[[1]][[2]]

             [,1]

[1,]  4.2847824968

[2,] -0.7900958121

$nic

[1] 135.6847209
```

12.Gwplot:

Description :

gwplot, a method for objects of class nn, typically produced by neuralnet. Plots the generalized weights (Intrator and Intrator, 1993) for one specific covariate and one response variable.

Usage:

```
  gwplot(x, rep = NULL, max = NULL, min = NULL, file = NULL,
selected.covariate = 1, selected.response = 1,
highlight = FALSE, type="p", col = "black", ...)
```

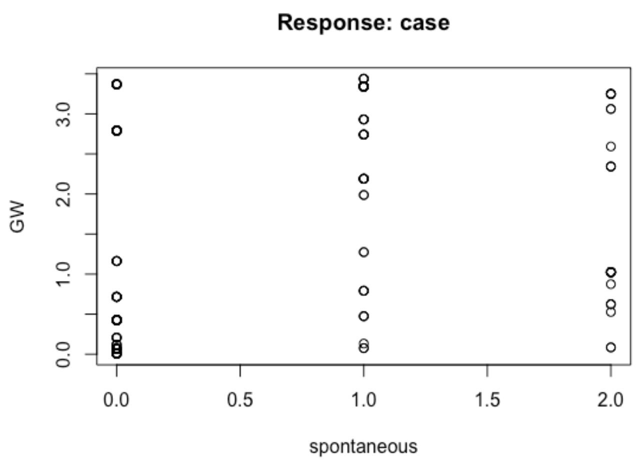
Examples

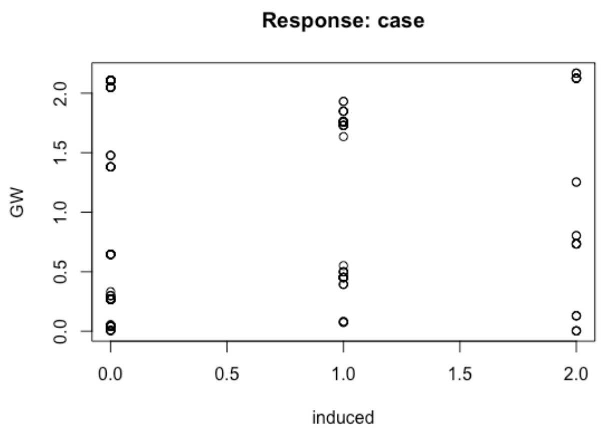
```
data(infert, package="datasets")
print(net.infert <- neuralnet(case~parity+induced+spontaneous, infert,
        err.fct="ce", linear.output=FALSE, likelihood=TRUE))
gwplot(net.infert, selected.covariate="parity")
gwplot(net.infert, selected.covariate="induced")
    gwplot(net.infert, selected.covariate="spontaneous")
```

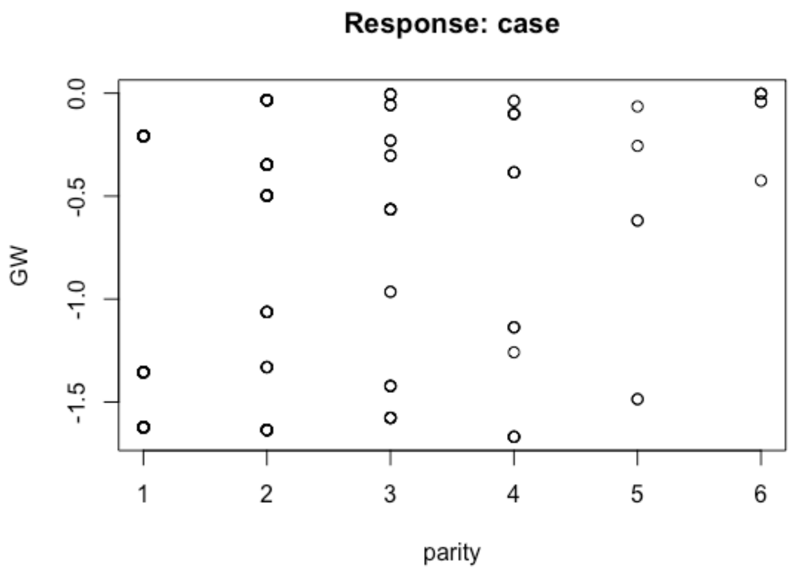
Output:

```
$generalized.weights
$generalized.weights[[1]]
        [,1]              [,2]            [,3]
1   -0.424424322088  0.551433013660 0.874333503465
2   -1.354818424689 1.760246922731 2.790987882103
3   -0.002377772417 0.003089319206 0.004898319864
4   -0.100471546368 0.130537588723 0.206975977960
5   -1.422695191271 1.848435765844 2.930817123834
6   -1.668850237967 2.168252543919 3.437907771501
7   -0.207218670440 0.269228717537 0.426879932842
8   -0.033151536713 0.043072111671 0.068293680948
9   -1.622017717427 2.107405423256 3.341430638526
10  -0.033151536713 0.043072111671 0.068293680948
11  -0.347704681420 0.451755072366 0.716287536919
12  -1.668850237967 2.168252543919 3.437907771501
.              .                  .                .
.              .                  .                .
.              .                     .             .
.          .                      .                .
139 -0.347704681420 0.451755072366 0.716287536919
```

140	-1.330684863556	1.728891409738	2.741271643051
141	-1.635595255546	2.125046030481	3.369400987665
142	-1.354818424689	1.760246922731	2.790987882103
143	-0.347704681420	0.451755072366	0.716287536919
144	-1.635595255546	2.125046030481	3.369400987665
145	-1.354818424689	1.760246922731	2.790987882103
146	-0.207218670440	0.269228717537	0.426879932842
147	-0.033151536713	0.043072111671	0.068293680948
148	-1.062967562204	1.381059886872	2.189761765206
149	-0.207218670440	0.269228717537	0.426879932842
150	-1.622017717427	2.107405423256	3.341430638526
151	-0.564467219335	0.733383653271	1.162828273000
152	-1.422695191271	1.848435765844	2.930817123834
153	-0.207218670440	0.269228717537	0.426879932842
154	-1.354818424689	1.760246922731	2.790987882103
155	-0.207218670440	0.269228717537	0.426879932842
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
242	-1.062967562204	1.381059886872	2.189761765206
243	-1.622017717427	2.107405423256	3.341430638526
244	-1.622017717427	2.107405423256	3.341430638526
245	-0.207218670440	0.269228717537	0.426879932842
246	-1.635595255546	2.125046030481	3.369400987665
247	-1.622017717427	2.107405423256	3.341430638526







2.1. Applications:

The utility of artificial neural network models lies in the fact that they can be used to infer a function from observations. This is particularly useful in applications where the complexity of the data or task makes the design of such a function by hand impractical.

The tasks artificial neural networks are applied to tend to fall within the following broad categories:

- *Function approximation, or regression analysis, including time series prediction, fitness approximation and modeling*
- *Classification, including pattern and sequence recognition, novelty detection and sequential decision making*
- *Data processing, including filtering, clustering, blind source separation and compression*
- *Robotics, including directing manipulators, prosthesis.*
- *Control, including Computer numerical control*

2.2. Criticism

2.2.1. Training issues:

A common criticism of neural networks, particularly in robotics, is that they require a large diversity of training for real-world operation. This is not surprising, since any learning machine needs sufficient representative examples in order to capture the underlying structure that allows it to generalise to new cases. A large amount of his research is devoted to (1) extrapolating multiple training scenarios from a single training experience, and (2) preserving past training diversity so that the system does not become overtrained (if, for example, it is presented with a series of right turns – it should not learn to always turn right). These issues are common in neural networks that must decide from amongst a wide variety of responses

2.2.2. Theoretical Issues:

Although neural nets do solve a few toy problems, their powers of computation are so limited.No neural network has ever been shown that solves computationally difficult problems such as the n-Queens problem, the travelling salesman problem, or the problem of factoring large integers. Aside from their utility, a fundamental objection to artificial neural networks is that they fail to reflect how real neurons function. Back propagation is at the heart of most artificial neural networks and not only is there no evidence of any such mechanism in natural neural networks, it seems to contradict the fundamental principle of real neurons that information can only flow forward along the axon.How information is coded by real neurons is not yet known.

2.2.3. Hardware issues:

To implement large and effective software neural networks, considerable processing and storage resources need to be committed. While the brain has hardware tailored to the task of processing signals through a graph of neurons, simulating even a most simplified form on von Neumann architecture may compel a neural network designer to fill many millions of database rows for its connections – which can consume vast amounts of computer memory and hard disk space. Furthermore, the designer of neural network systems will often need to simulate the transmission of signals through many of these connections and their associated neurons – which must often be matched with incredible amounts of CPU processing power and time.

3. Conclusion:

Although it is true that analyzing what has been learned by an artificial neural network is difficult, it is much easier to do so than to analyze what has been learned by a biological neural network.Furthermore, the packages and commands are thereby studied using RStudio and CRAN Repository packages.