neuralnet: Training of Neural Networks

by Frauke Günther and Stefan Fritsch

Abstract Artificial neural networks are applied in many situations. **neuralnet** is built to train multi-layer perceptrons in the context of regression analyses, i.e. to approximate functional relationships between covariates and response variables. Thus, neural networks are used as extensions of generalized linear models.
neuralnet is a very flexible package. The backpropagation algorithm and three versions of resilient backpropagation are implemented and it provides a custom-choice of activation and error function. An arbitrary number of covariates and response variables as well as of hidden layers can theoretically be included.
The paper gives a brief introduction to multi-layer perceptrons and resilient backpropagation and demonstrates the application of **neuralnet** using the data set `infert`, which is contained in the R distribution.

Introduction

In many situations, the functional relationship between covariates (also known as input variables) and response variables (also known as output variables) is of great interest. For instance when modeling complex diseases, potential risk factors and their effects on the disease are investigated to identify risk factors that can be used to develop prevention or intervention strategies. Artificial neural networks can be applied to approximate any complex functional relationship. Unlike generalized linear models (GLM, McCullagh and Nelder, 1983), it is not necessary to prespecify the type of relationship between covariates and response variables as for instance as linear combination. This makes artificial neural networks a valuable statistical tool. They are in particular direct extensions of GLMs and can be applied in a similar manner. Observed data are used to train the neural network and the neural network learns an approximation of the relationship by iteratively adapting its parameters.

The package **neuralnet** (Fritsch and Günther, 2008) contains a very flexible function to train feed-forward neural networks, i.e. to approximate a functional relationship in the above situation. It can theoretically handle an arbitrary number of covariates and response variables as well as of hidden layers and hidden neurons even though the computational costs can increase exponentially with higher order of complexity. This can cause an early stop of the iteration process since the maximum of iteration steps, which can be defined by the user, is reached before the algorithm converges. In addition, the package provides functions to visualize the results or in general to facilitate the usage of neural networks. For instance, the function `compute` can be applied to calculate predictions for new covariate combinations.

There are two other packages that deal with artificial neural networks at the moment: **nnet** (Venables and Ripley, 2002) and **AMORE** (Limas et al., 2007). **nnet** provides the opportunity to train feed-forward neural networks with traditional backpropagation and in **AMORE**, the TAO robust neural network algorithm is implemented. **neuralnet** was built to train neural networks in the context of regression analyses. Thus, resilient backpropagation is used since this algorithm is still one of the fastest algorithms for this purpose (e.g. Schiffmann et al., 1994; Rocha et al., 2003; Kumar and Zhang, 2006; Almeida et al., 2010). Three different versions are implemented and the traditional backpropagation is included for comparison purposes. Due to a custom-choice of activation and error function, the package is very flexible. The user is able to use several hidden layers, which can reduce the computational costs by including an extra hidden layer and hence reducing the neurons per layer. We successfully used this package to model complex diseases, i.e. different structures of biological gene-gene interactions (Günther et al., 2009). Summarizing, **neuralnet** closes a gap concerning the provided algorithms for training neural networks in R.

To facilitate the usage of this package for new users of artificial neural networks, a brief introduction to neural networks and the learning algorithms implemented in **neuralnet** is given before describing its application.

Multi-layer perceptrons

The package **neuralnet** focuses on multi-layer perceptrons (MLP, Bishop, 1995), which are well applicable when modeling functional relationships. The underlying structure of an MLP is a directed graph, i.e. it consists of vertices and directed edges, in this context called neurons and synapses. The neurons are organized in layers, which are usually fully connected by synapses. In **neuralnet**, a synapse can only connect to subsequent layers. The input layer consists of all covariates in separate neurons and the output layer consists of the response variables. The layers in between are referred to as hidden layers, as they are not directly observable. Input layer and hidden layers include a constant neuron relating to intercept synapses, i.e. synapses that are not directly influenced by any covariate. Figure 1 gives an example of a neural network with one hidden layer that consists of three hidden neurons. This neural network models the relationship between the two co-

variates A and B and the response variable Y. **neuralnet** theoretically allows inclusion of arbitrary numbers of covariates and response variables. However, there can occur convergence difficulties using a huge number of both covariates and response variables.

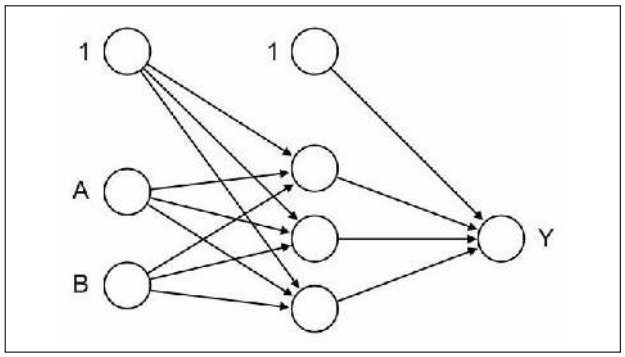

Figure 1: Example of a neural network with two input neurons (A and B), one output neuron (Y) and one hidden layer consisting of three hidden neurons.

To each of the synapses, a weight is attached indicating the effect of the corresponding neuron, and all data pass the neural network as signals. The signals are processed first by the so-called integration function combining all incoming signals and second by the so-called activation function transforming the output of the neuron.

The simplest multi-layer perceptron (also known as perceptron) consists of an input layer with n covariates and an output layer with one output neuron. It calculates the function

$$o(\mathbf{x}) = f\left(w_0 + \sum_{i=1}^{n} w_i x_i\right) = f\left(w_0 + \mathbf{w}^T\mathbf{x}\right),$$

where w_0 denotes the intercept, $\mathbf{w} = (w_1,\ldots,w_n)$ the vector consisting of all synaptic weights without the intercept, and $\mathbf{x} = (x_1,\ldots,x_n)$ the vector of all covariates. The function is mathematically equivalent to that of GLM with link function f^{-1}. Therefore, all calculated weights are in this case equivalent to the regression parameters of the GLM.

To increase the modeling flexibility, hidden layers can be included. However, Hornik et al. (1989) showed that one hidden layer is sufficient to model any piecewise continuous function. Such an MLP with a hidden layer consisting of J hidden neurons calculates the following function:

$$\begin{aligned} o(\mathbf{x}) =& f\left(w_0 + \sum_{j=1}^{J} w_j \cdot f\left(w_{0j} + \sum_{i=1}^{n} w_{ij} x_i\right)\right) \\ =& f\left(w_0 + \sum_{j=1}^{J} w_j \cdot f\left(w_{0j} + \mathbf{w_j}^T\mathbf{x}\right)\right), \end{aligned}$$

where w_0 denotes the intercept of the output neuron and w_{0j} the intercept of the jth hidden neuron. Additionally, w_j denotes the synaptic weight corresponding to the synapse starting at the jth hidden neuron and leading to the output neuron, $\mathbf{w_j} = (w_{1j},\ldots,w_{nj})$ the vector of all synaptic weights corresponding to the synapses leading to the jth hidden neuron, and $\mathbf{x} = (x_1,\ldots,x_n)$ the vector of all covariates. This shows that neural networks are direct extensions of GLMs. However, the parameters, i.e. the weights, cannot be interpreted in the same way anymore.

Formally stated, all hidden neurons and output neurons calculate an output $f(g(z_0,z_1,\ldots,z_k)) = f(g(\mathbf{z}))$ from the outputs of all preceding neurons $z_0,z_1,\ldots,z_k$, where $g : \mathrm{R}^{k+1} \to \mathrm{R}$ denotes the integration function and $f : \mathrm{R} \to \mathrm{R}$ the activation function. The neuron $z_0 \equiv 1$ is the constant one belonging to the intercept. The integration function is often defined as $g(\mathbf{z}) = w_0 z_0 + \sum_{i=1}^{k} w_i z_i = w_0 + \mathbf{w^T z}$. The activation function f is usually a bounded nondecreasing nonlinear and differentiable function such as the logistic function ($f(u) = \frac{1}{1+e^{-u}}$) or the hyperbolic tangent. It should be chosen in relation to the response variable as it is the case in GLMs. The logistic function is, for instance, appropriate for binary response variables since it maps the output of each neuron to the interval $[0,1]$. At the moment, **neuralnet** uses the same integration as well as activation function for all neurons.

Supervised learning

Neural networks are fitted to the data by learning algorithms during a training process. **neuralnet** focuses on supervised learning algorithms. These learning algorithms are characterized by the usage of a given output that is compared to the predicted output and by the adaptation of all parameters according to this comparison. The parameters of a neural network are its weights. All weights are usually initialized with random values drawn from a standard normal distribution. During an iterative training process, the following steps are repeated:

- The neural network calculates an output $\mathbf{o}(\mathbf{x})$ for given inputs $\mathbf{x}$ and current weights. If the training process is not yet completed, the predicted output $\mathbf{o}$ will differ from the observed output $\mathbf{y}$.
- An error function E, like the sum of squared errors (SSE)

$$E = \frac{1}{2}\sum_{l=1}^{L}\sum_{h=1}^{H}(o_{lh} - y_{lh})^2$$

or the cross-entropy

$$\begin{aligned} E = -\sum_{l=1}^{L}\sum_{h=1}^{H} &(y_{lh}\log(o_{lh}) \\ &+ (1-y_{lh})\log(1-o_{lh})), \end{aligned}$$

measures the difference between predicted and observed output, where $l = 1,\ldots,L$ indexes the

observations, i.e. given input-output pairs, and $h = 1, \ldots, H$ the output nodes.

- All weights are adapted according to the rule of a learning algorithm.

The process stops if a pre-specified criterion is fulfilled, e.g. if all absolute partial derivatives of the error function with respect to the weights ($\partial E/\partial w$) are smaller than a given threshold. A widely used learning algorithm is the resilient backpropagation algorithm.

Backpropagation and resilient backpropagation

The resilient backpropagation algorithm is based on the traditional backpropagation algorithm that modifies the weights of a neural network in order to find a local minimum of the error function. Therefore, the gradient of the error function ($dE/d\mathbf{w}$) is calculated with respect to the weights in order to find a root. In particular, the weights are modified going in the opposite direction of the partial derivatives until a local minimum is reached. This basic idea is roughly illustrated in Figure 2 for a univariate error-function.

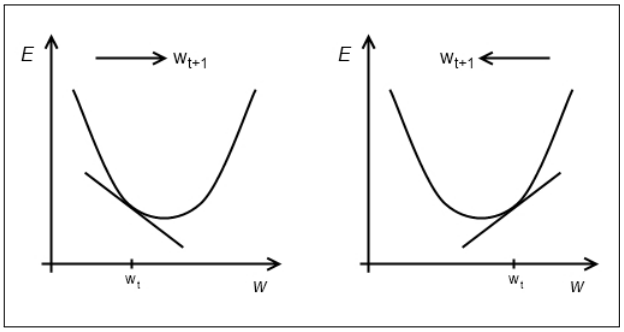

Figure 2: Basic idea of the backpropagation algorithm illustrated for a univariate error function $E(w)$.

If the partial derivative is negative, the weight is increased (left part of the figure); if the partial derivative is positive, the weight is decreased (right part of the figure). This ensures that a local minimum is reached. All partial derivatives are calculated using the chain rule since the calculated function of a neural network is basically a composition of integration and activation functions. A detailed explanation is given in Rojas (1996).

neuralnet provides the opportunity to switch between backpropagation, resilient backpropagation with (Riedmiller, 1994) or without weight backtracking (Riedmiller and Braun, 1993) and the modified globally convergent version by Anastasiadis et al. (2005). All algorithms try to minimize the error function by adding a learning rate to the weights going into the opposite direction of the gradient. Unlike the traditional backpropagation algorithm, a separate learning rate η_k, which can be changed during the training process, is used for each weight in resilient backpropagation. This solves the problem of defining an over-all learning rate that is appropriate for the whole training process and the entire network. Additionally, instead of the magnitude of the partial derivatives only their sign is used to update the weights. This guarantees an equal influence of the learning rate over the entire network (Riedmiller and Braun, 1993). The weights are adjusted by the following rule

$$w_k^{(t+1)} = w_k^{(t)} - \eta_k^{(t)} \cdot \operatorname{sign}\left(\frac{\partial E^{(t)}}{\partial w_k^{(t)}}\right),$$

as opposed to

$$w_k^{(t+1)} = w_k^{(t)} - \eta \cdot \frac{\partial E^{(t)}}{\partial w_k^{(t)}},$$

in traditional backpropagation, where t indexes the iteration steps and k the weights.

In order to speed up convergence in shallow areas, the learning rate η_k will be increased if the corresponding partial derivative keeps its sign. On the contrary, it will be decreased if the partial derivative of the error function changes its sign since a changing sign indicates that the minimum is missed due to a too large learning rate. Weight backtracking is a technique of undoing the last iteration and adding a smaller value to the weight in the next step. Without the usage of weight backtracking, the algorithm can jump over the minimum several times. For example, the pseudocode of resilient backpropagation with weight backtracking is given by (Riedmiller and Braun, 1993)

```
for all weights{
    if (grad.old*grad>0){
        delta := min(delta*eta.plus, delta.max)
        weights := weights - sign(grad)*delta
        grad.old := grad
    }
    else if (grad.old*grad<0){
        weights := weights + sign(grad.old)*delta
        delta := max(delta*eta.minus, delta.min)
        grad.old := 0
    }
    else if (grad.old*grad=0){
        weights := weights - sign(grad)*delta
        grad.old := grad
    }
}
```

while that of the regular backpropagation is given by

```
for all weights{
    weights := weights - grad*delta
}
```

The globally convergent version introduced by Anastasiadis et al. (2005) performs a resilient backpropagation with an additional modification of one learning rate in relation to all other learning rates. It

is either the learning rate associated with the smallest absolute partial derivative or the smallest learning rate (indexed with i), that is changed according to

$$\eta_i^{(t)} = -\frac{\sum_{k;k\neq i}\eta_k^{(t)}\cdot\frac{\partial E^{(t)}}{\partial w_k^{(t)}}+\delta}{\frac{\partial E^{(t)}}{\partial w_i^{(t)}}},$$

if $\frac{\partial E^{(t)}}{\partial w_i^{(t)}} \neq 0$ and $0 < \delta \ll \infty$. For further details see Anastasiadis et al. (2005).

Using neuralnet

neuralnet depends on two other packages: **grid** and **MASS** (Venables and Ripley, 2002). Its usage is leaned towards that of functions dealing with regression analyses like `lm` and `glm`. As essential arguments, a formula in terms of *response variables ~ sum of covariates* and a data set containing covariates and response variables have to be specified. Default values are defined for all other parameters (see next subsection). We use the data set `infert` that is provided by the package **datasets** to illustrate its application. This data set contains data of a case-control study that investigated infertility after spontaneous and induced abortion (Trichopoulos et al., 1976). The data set consists of 248 observations, 83 women, who were infertile (cases), and 165 women, who were not infertile (controls). It includes amongst others the variables `age`, `parity`, `induced`, and `spontaneous`. The variables `induced` and `spontaneous` denote the number of prior induced and spontaneous abortions, respectively. Both variables take possible values 0, 1, and 2 relating to 0, 1, and 2 or more prior abortions. The age in years is given by the variable `age` and the number of births by `parity`.

Training of neural networks

The function `neuralnet` used for training a neural network provides the opportunity to define the required number of hidden layers and hidden neurons according to the needed complexity. The complexity of the calculated function increases with the addition of hidden layers or hidden neurons. The default value is one hidden layer with one hidden neuron. The most important arguments of the function are the following:

- `formula`, a symbolic description of the model to be fitted (see above). No default.
- `data`, a data frame containing the variables specified in `formula`. No default.
- `hidden`, a vector specifying the number of hidden layers and hidden neurons in each layer. For example the vector (3,2,1) induces a neural network with three hidden layers, the first one with three, the second one with two and the third one with one hidden neuron. Default: 1.
- `threshold`, an integer specifying the threshold for the partial derivatives of the error function as stopping criteria. Default: 0.01.
- `rep`, number of repetitions for the training process. Default: 1.
- `startweights`, a vector containing prespecified starting values for the weights. Default: random numbers drawn from the standard normal distribution
- `algorithm`, a string containing the algorithm type. Possible values are `"backprop"`, `"rprop+"`, `"rprop-"`, `"sag"`, or `"slr"`. `"backprop"` refers to traditional backpropagation, `"rprop+"` and `"rprop-"` refer to resilient backpropagation with and without weight backtracking and `"sag"` and `"slr"` refer to the modified globally convergent algorithm (grprop). `"sag"` and `"slr"` define the learning rate that is changed according to all others. `"sag"` refers to the smallest absolute derivative, `"slr"` to the smallest learning rate. Default: `"rprop+"`
- `err.fct`, a differentiable error function. The strings `"sse"` and `"ce"` can be used, which refer to 'sum of squared errors' and 'cross entropy'. Default: `"sse"`
- `act.fct`, a differentiable activation function. The strings `"logistic"` and `"tanh"` are possible for the logistic function and tangent hyperbolicus. Default: `"logistic"`
- `linear.output`, logical. If `act.fct` should not be applied to the output neurons, `linear.output` has to be `TRUE`. Default: `TRUE`
- `likelihood`, logical. If the error function is equal to the negative log-likelihood function, `likelihood` has to be `TRUE`. Akaike's Information Criterion (AIC, Akaike, 1973) and Bayes Information Criterion (BIC, Schwarz, 1978) will then be calculated. Default: FALSE
- `exclude`, a vector or matrix specifying weights that should be excluded from training. A matrix with n rows and three columns will exclude n weights, where the first column indicates the layer, the second column the input neuron of the weight, and the third neuron the output neuron of the weight. If given as vector, the exact numbering has to be known. The numbering can be checked using the provided plot or the saved starting weights. Default: `NULL`
- `constant.weights`, a vector specifying the values of weights that are excluded from training and treated as fixed. Default: `NULL`

The usage of `neuralnet` is described by modeling the relationship between the case-control status (`case`) as response variable and the four covariates `age`, `parity`, `induced` and `spontaneous`. Since the response variable is binary, the activation function could be chosen as logistic function (default) and the error function as cross-entropy (`err.fct="ce"`). Additionally, the item `linear.output` should be stated as `FALSE` to ensure that the output is mapped by the activation function to the interval $[0,1]$. The number of hidden neurons should be determined in relation to the needed complexity. A neural network with for example two hidden neurons is trained by the following statements:

```
> library(neuralnet)
Loading required package: grid
Loading required package: MASS
>
> nn <- neuralnet(
+       case~age+parity+induced+spontaneous,
+       data=infert, hidden=2, err.fct="ce",
+       linear.output=FALSE)
> nn
Call:
  neuralnet(
  formula = case~age+parity+induced+spontaneous,
  data = infert, hidden = 2, err.fct = "ce",
  linear.output = FALSE)

1 repetition was calculated.

        Error Reached Threshold Steps
1 125.2126851    0.008779243419  5254
```

Basic information about the training process and the trained neural network is saved in `nn`. This includes all information that has to be known to reproduce the results as for instance the starting weights. Important values are the following:

- `net.result`, a list containing the overall result, i.e. the output, of the neural network for each replication.
- `weights`, a list containing the fitted weights of the neural network for each replication.
- `generalized.weights`, a list containing the generalized weights of the neural network for each replication.
- `result.matrix`, a matrix containing the error, reached threshold, needed steps, AIC and BIC (computed if `likelihood=TRUE`) and estimated weights for each replication. Each column represents one replication.
- `startweights`, a list containing the starting weights for each replication.

A summary of the main results is provided by `nn$result.matrix`:

```
> nn$result.matrix
                                      1
error                   125.212685099732
reached.threshold         0.008779243419
steps                  5254.000000000000
Intercept.to.1layhid1     5.593787533788
age.to.1layhid1          -0.117576380283
parity.to.1layhid1        1.765945780047
induced.to.1layhid1      -2.200113693672
spontaneous.to.1layhid1  -3.369491912508
Intercept.to.1layhid2     1.060701883258
age.to.1layhid2           2.925601414213
parity.to.1layhid2        0.259809664488
induced.to.1layhid2      -0.120043540527
spontaneous.to.1layhid2  -0.033475146593
Intercept.to.case         0.722297491596
1layhid.1.to.case        -5.141324077052
1layhid.2.to.case         2.623245311046
```

The training process needed 5254 steps until all absolute partial derivatives of the error function were smaller than 0.01 (the default threshold). The estimated weights range from -5.14 to 5.59. For instance, the intercepts of the first hidden layer are 5.59 and 1.06 and the four weights leading to the first hidden neuron are estimated as -0.12, 1.77, -2.20, and -3.37 for the covariates `age`, `parity`, `induced` and `spontaneous`, respectively. If the error function is equal to the negative log-likelihood function, the error refers to the likelihood as is used for example to calculate Akaike's Information Criterion (AIC).

The given data is saved in `nn$covariate` and `nn$response` as well as in `nn$data` for the whole data set inclusive non-used variables. The output of the neural network, i.e. the fitted values $o(\mathbf{x})$, is provided by `nn$net.result`:

```
> out <- cbind(nn$covariate,
+              nn$net.result[[1]])
> dimnames(out) <- list(NULL,
+                  c("age","parity","induced",
+                    "spontaneous","nn-output"))
> head(out)
     age parity induced spontaneous    nn-output
[1,]  26      6       1           2 0.1519579877
[2,]  42      1       1           0 0.6204480608
[3,]  39      6       2           0 0.1428325816
[4,]  34      4       2           0 0.1513351888
[5,]  35      3       1           1 0.3516163154
[6,]  36      4       2           1 0.4904344475
```

In this case, the object `nn$net.result` is a list consisting of only one element relating to one calculated replication. If more than one replication were calculated, the outputs would be saved each in a separate list element. This approach is the same for all values that change with replication apart from `net.result` that is saved as matrix with one column for each replication.

To compare the results, neural networks are trained with the same parameter setting as above using **neuralnet** with `algorithm="backprop"` and the package **nnet**.

```
> nn.bp <- neuralnet(
+          case~age+parity+induced+spontaneous,
+          data=infert, hidden=2, err.fct="ce",
+          linear.output=FALSE,
+          algorithm="backprop",
+          learningrate=0.01)
> nn.bp
Call:
  neuralnet(
  formula = case~age+parity+induced+spontaneous,
  data = infert, hidden = 2, learningrate = 0.01,
  algorithm = "backprop", err.fct = "ce",
  linear.output = FALSE)

1 repetition was calculated.

      Error Reached Threshold Steps
1 158.085556    0.008087314995     4
>
>
> nn.nnet <- nnet(
+            case~age+parity+induced+spontaneous,
+            data=infert, size=2, entropy=T,
+            abstol=0.01)
# weights:  13
initial  value 158.121035
final  value 158.085463
converged
```

`nn.bp` and `nn.nnet` show equal results. Both training processes last only a very few iteration steps and the error is approximately 158. Thus in this little comparison, the model fit is less satisfying than that achieved by resilient backpropagation.

neuralnet includes the calculation of generalized weights as introduced by Intrator and Intrator (2001). The generalized weight $\tilde{w}_i$ is defined as the contribution of the ith covariate to the log-odds:

$$\tilde{w}_i = \frac{\partial \log\left(\frac{o(\mathbf{x})}{1-o(\mathbf{x})}\right)}{\partial x_i}.$$

The generalized weight expresses the effect of each covariate x_i and thus has an analogous interpretation as the ith regression parameter in regression models. However, the generalized weight depends on all other covariates. Its distribution indicates whether the effect of the covariate is linear since a small variance suggests a linear effect (Intrator and Intrator, 2001). They are saved in `nn$generalized.weights` and are given in the following format (rounded values)

```
> head(nn$generalized.weights[[1]])
      [,1]       [,2]      [,3]      [,4]
1 0.0088556 -0.1330079 0.1657087 0.2537842
2 0.1492874 -2.2422321 2.7934978 4.2782645
3 0.0004489 -0.0067430 0.0084008 0.0128660
4 0.0083028 -0.1247051 0.1553646 0.2379421
5 0.1071413 -1.6092161 2.0048511 3.0704457
6 0.1360035 -2.0427123 2.5449249 3.8975730
```

The columns refer to the four covariates `age` ($j = 1$), `parity` ($j = 2$), `induced` ($j = 3$), and `spontaneous` ($j = 4$) and a generalized weight is given for each observation even though they are equal for each covariate combination.

Visualizing the results

The results of the training process can be visualized by two different plots. First, the trained neural network can simply be plotted by

```
> plot(nn)
```

The resulting plot is given in Figure 3.

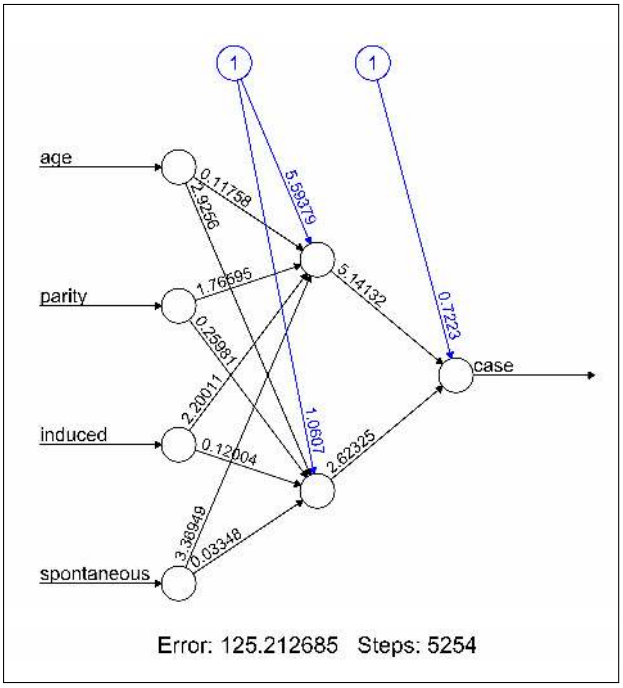

Figure 3: Plot of a trained neural network including trained synaptic weights and basic information about the training process.

It reflects the structure of the trained neural network, i.e. the network topology. The plot includes by default the trained synaptic weights, all intercepts as well as basic information about the training process like the overall error and the number of steps needed to converge. Especially for larger neural networks, the size of the plot and that of each neuron can be determined using the parameters `dimension` and `radius`, respectively.

The second possibility to visualize the results is to plot generalized weights. `gwplot` uses the calculated generalized weights provided by `nn$generalized.weights` and can be used by the following statements:

```
> par(mfrow=c(2,2))
> gwplot(nn,selected.covariate="age",
+        min=-2.5, max=5)
> gwplot(nn,selected.covariate="parity",
+        min=-2.5, max=5)
> gwplot(nn,selected.covariate="induced",
```

```
+        min=-2.5, max=5)
> gwplot(nn,selected.covariate="spontaneous",
+        min=-2.5, max=5)
```

The corresponding plot is shown in Figure 4.

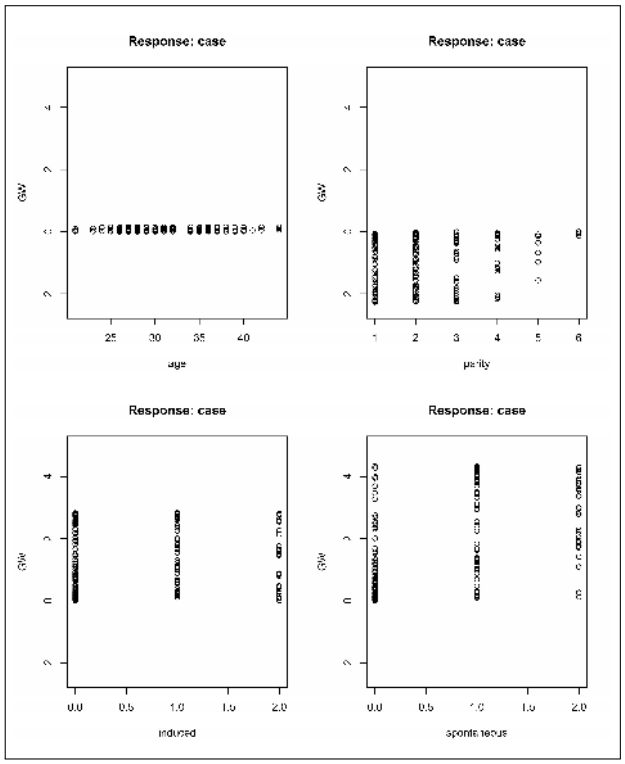

Figure 4: Plots of generalized weights with respect to each covariate.

The generalized weights are given for all covariates within the same range. The distribution of the generalized weights suggests that the covariate `age` has no effect on the case-control status since all generalized weights are nearly zero and that at least the two covariates `induced` and `spontaneous` have a nonlinear effect since the variance of their generalized weights is overall greater than one.

Additional features

The `compute` function

`compute` calculates and summarizes the output of each neuron, i.e. all neurons in the input, hidden and output layer. Thus, it can be used to trace all signals passing the neural network for given covariate combinations. This helps to interpret the network topology of a trained neural network. It can also easily be used to calculate predictions for new covariate combinations. A neural network is trained with a training data set consisting of known input-output pairs. It learns an approximation of the relationship between inputs and outputs and can then be used to predict outputs $o(\mathbf{x}_{new})$ relating to new covariate combinations $\mathbf{x}_{new}$. The function `compute` simplifies this calculation. It automatically redefines the structure of the given neural network and calculates the output for arbitrary covariate combinations.

To stay with the example, predicted outputs can be calculated for instance for missing combinations with `age=22`, `parity=1`, `induced` ≤ 1, and `spontaneous` ≤ 1. They are provided by `new.output$net.result`

```
> new.output <- compute(nn,
                 covariate=matrix(c(22,1,0,0,
                                    22,1,1,0,
                                    22,1,0,1,
                                    22,1,1,1),
                                  byrow=TRUE, ncol=4))
> new.output$net.result
          [,1]
[1,] 0.1477097
[2,] 0.1929026
[3,] 0.3139651
[4,] 0.8516760
```

This means that the predicted probability of being a case given the mentioned covariate combinations, i.e. $o(\mathbf{x})$, is increasing in this example with the number of prior abortions.

The `confidence.interval` function

The weights of a neural network follow a multivariate normal distribution if the network is identified (White, 1989). A neural network is identified if it does not include irrelevant neurons neither in the input layer nor in the hidden layers. An irrelevant neuron in the input layer can be for instance a covariate that has no effect or that is a linear combination of other included covariates. If this restriction is fulfilled and if the error function equals the neagtive log-likelihood, a confidence interval can be calculated for each weight. The **neuralnet** package provides a function to calculate these confidence intervals regardless of whether all restrictions are fulfilled. Therefore, the user has to be careful interpreting the results.

Since the covariate `age` has no effect on the outcome and the related neuron is thus irrelevant, a new neural network (`nn.new`), which has only the three input variables `parity`, `induced`, and `spontaneous`, has to be trained to demonstrate the usage of `confidence.interval`. Let us assume that all restrictions are now fulfilled, i.e. neither the three input variables nor the two hidden neurons are irrelevant. Confidence intervals can then be calculated with the function `confidence.interval`:

```
> ci <- confidence.interval(nn.new, alpha=0.05)
> ci$lower.ci
[[1]]
[[1]][[1]]
              [,1]          [,2]
[1,]   1.830803796  -2.680895286
[2,]   1.673863304  -2.839908343
[3,]  -8.883004913 -37.232020925
```

```
[4,] -48.906348154 -18.748849335

[[1]][[2]]
              [,1]
[1,]  1.283391149
[2,] -3.724315385
[3,] -2.650545922
```

For each weight, `ci$lower.ci` provides the related lower confidence limit and `ci$upper.ci` the related upper confidence limit. The first matrix contains the limits of the weights leading to the hidden neurons. The columns refer to the two hidden neurons. The other three values are the limits of the weights leading to the output neuron.

Summary

This paper gave a brief introduction to multi-layer perceptrons and supervised learning algorithms. It introduced the package **neuralnet** that can be applied when modeling functional relationships between covariates and response variables. **neuralnet** contains a very flexible function that trains multilayer perceptrons to a given data set in the context of regression analyses. It is a very flexible package since most parameters can be easily adapted. For example, the activation function and the error function can be arbitrarily chosen and can be defined by the usual definition of functions in R.

Acknowledgements

The authors thank Nina Wawro for reading preliminary versions of the paper and for giving helpful comments. Additionally, we would like to thank two anonymous reviewers for their valuable suggestions and remarks.

We gratefully acknowledge the financial support of this research by the grant PI 345/3-1 from the German Research Foundation (DFG).

Bibliography

H. Akaike. Information theory and an extension of the maximum likelihood principle. In Petrov BN and Csaki BF, editors, *Second international symposium on information theory*, pages 267–281. Academiai Kiado, Budapest, 1973.

C. Almeida, C. Baugh, C. Lacey, C. Frenk, G. Granato, L. Silva, and A. Bressan. Modelling the dsty universe i: Introducing the artificial neural network and first applications to luminosity and colour distributions. *Monthly Notices of the Royal Astronomical Society*, 402:544–564, 2010.

A. Anastasiadis, G. Magoulas, and M. Vrahatis. New globally convergent training scheme based on the resilient propagation algorithm. *Neurocomputing*, 64:253–270, 2005.

C. Bishop. *Neural networks for pattern recognition*. Oxford University Press, New York, 1995.

S. Fritsch and F. Günther. *neuralnet: Training of Neural Networks*. R Foundation for Statistical Computing, 2008. R package version 1.2.

F. Günther, N. Wawro, and K. Bammann. Neural networks for modeling gene-gene interactions in association studies. *BMC Genetics*, 10:87, 2009. http://www.biomedcentral.com/1471-2156/10/87.

K. Hornik, M. Stichcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.

O. Intrator and N. Intrator. Interpreting neural-network results: a simulation study. *Computational Statistics & Data Analysis*, 37:373–393, 2001.

A. Kumar and D. Zhang. Personal recognition using hand shape and texture. *IEEE Transactions on Image Processing*, 15:2454–2461, 2006.

M. C. Limas, E. P. V. G. Joaquín B. Ordieres Meré, F. J. M. de Pisón Ascacibar, A. V. P. Espinoza, and F. A. Elías. *AMORE: A MORE Flexible Neural Network Package*, 2007. URL http://wiki.r-project.org/rwiki/doku.php?id=packages:cran:amore. R package version 0.2-11.

P. McCullagh and J. Nelder. *Generalized Linear Models*. Chapman and Hall, London, 1983.

M. Riedmiller. Advanced supervised learning in multi-layer perceptrons - from backpropagation to adaptive learning algorithms. *International Journal of Computer Standards and Interfaces*, 16:265–278, 1994.

M. Riedmiller and H. Braun. A direct method for faster backpropagation learning: the rprop algorithm. *Proceedings of the IEEE International Conference on Neural Networks (ICNN)*, 1:586–591, 1993.

M. Rocha, P. Cortez, and J. Neves. Evolutionary neural network learning. *Lecture Notes in Computer Science*, 2902:24–28, 2003.

R. Rojas. *Neural Networks*. Springer-Verlag, Berlin, 1996.

W. Schiffmann, M. Joost, and R. Werner. Optimization of the backpropagation algorithm for training multilayer perceptrons. Technical report, University of Koblenz, Insitute of Physics, 1994.

G. Schwarz. Estimating the dimension of a model. *Ann Stat*, 6:461–464, 1978.

D. Trichopoulos, N. Handanos, J. Danezis, A. Kalandidi, and V. Kalapothaki. Induced abortion and secondary infertility. *British Journal of Obstetrics and Gynaecology*, 83:645–650, 1976.

W. Venables and B. Ripley. *Modern Applied Statistics with S*. Springer, New York, fourth edition, 2002. URL http://www.stats.ox.ac.uk/pub/MASS4. ISBN 0-387-95457-0.

H. White. Learning in artificial neural networks: a statistical perspective. *Neural Computation*, 1:425–464, 1989.

Frauke Günther
University of Bremen, Bremen Institute for Prevention Research and Social Medicine
guenther@bips.uni-bremen.de

Stefan Fritsch
University of Bremen, Bremen Institute for Prevention Research and Social Medicine