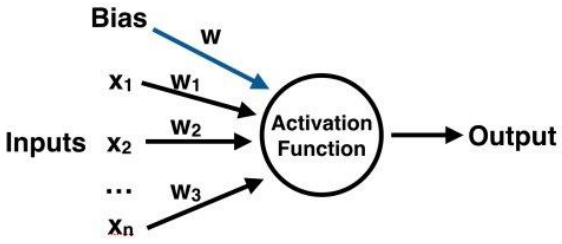
Business Data Mining (IDS 572)

Neural Network Model

Neural Networks are a machine learning framework that attempts to mimic the learning pattern of natural biological neural networks. Biological neural networks have interconnected neurons with dendrites that receive inputs, then based on these inputs they produce an output signal through an axon to another neuron. We will try to mimic this process through the use of Artificial Neural Networks (ANN), which we will just refer to as neural networks. The process of creating a neural network begins with the most basic form, a single perceptron.

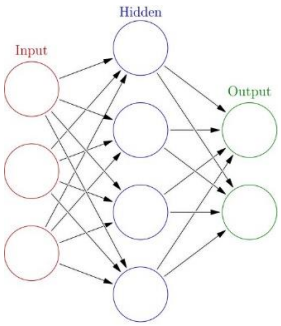
The Perceptron

A perceptron has one or more inputs, a bias, an activation function, and a single output. The perceptron receives inputs, multiplies them by some weight, and then passes them into an activation function to produce an output. There are many possible activation functions to choose from, such as the identity function, a logistic function, a step function etc. We also make sure to add a bias to the perceptron, this avoids issues where all inputs could be equal to zero (meaning no multiplicative weight would have an effect).



Once we have the output we can compare it to a known label and adjust the weights accordingly (the weights usually start off with random initialization values). We keep repeating this process until we have reached a maximum number of allowed iterations, or an acceptable error rate.

To create a neural network, we simply begin to add layers of perceptrons together, creating a multi-layer perceptron model of a neural network. You'll have an input layer which directly takes in your feature inputs and an output layer which will create the resulting outputs. Any layers in between are known as hidden layers because they don't directly "see" the feature inputs or outputs.



Creating a neural network in R

To construct a neural network model, we'll use ISLR's built in College Data Set which has several features of a college and a categorical column indicating whether or not the School is Public or Private.

```
> # install.packages("ISLR")
> library(ISLR)
```

Data Preprocessing

It is important to normalize data before training a neural network on it. The neural network may have difficulty converging before the maximum number of iterations allowed if the data is not normalized. There are a lot of different methods for normalization of data. We will use the built-in scale() function in R to easily accomplish this task.

Usually it is better to scale the data from 0 to 1, or -1 to 1. We can specify the center and scale as additional arguments in the scale() function. For example:

```
> # Create vector of column Max and Min values
> maxs = apply(College[ , 2:18], 2, max) # apply(x, margin, function). If margin = 1, function is
  applied on the rows and when margin  = 2, function is applied on the columns
> mins = apply(College[ , 2:18], 2, min)
> # Use scale() and convert the resulting matrix to a data frame

> # scale(data, center, scale); If center is a numeric vector with length equal to the number of
columns of data, then each column of data has the corresponding value from center subtracted
from it. If center is TRUE then centering is done by subtracting the column means (omitting NAs)
of data from their corresponding columns, and if center is FALSE, no centering is done. The value
of scale determines how column scaling is performed (after centering). If scale is a numeric
vector with length equal to the number of columns of data, then each column of data is divided
by the corresponding value from scale. If scale is TRUE then scaling is done by dividing the
(centered) columns of data by their standard deviations if center is TRUE, and the root mean
square otherwise. If scale is FALSE, no scaling is done.

> scaled.data = as.data.frame(scale(College[ , 2:18], center = mins, scale = maxs - mins))
```

Let us now split our data into a training set and a test set. We will run our neural network on the training set and then see how well it performed on the test set.

We will randomly split the data into a training set and test set.

```
> set.seed(1234)
> ind = sample(2, nrow(College), replace = T, prob = c(0.7, 0.3))
> TrainData = College[ind == 1, ]
> TestData = College[ind == 2, ]
```

To obtain a neural network we can use the function nnet() from the CART package as follows,

```
> library(nnet)
> nn  = nnet(Private ~ ., data=TrainData, linout=F, size=10, decay=0.01, maxit=1000)
```

This function call will build a neural network with a single hidden layer, in this case formed by 10 hidden units (the parameter size specifies how many nodes should be in the hidden layer). Moreover, the weights will be learned with a weight updating rate of 0.01 (the parameter decay). The parameter linout (linear out-put) indicates that the target variable is continuous or not. The maxit parameter sets the maximum number of iterations of the weight convergence algorithm. The nnet() function uses the back-propagation algorithm as the basis of an iterative process of updating the weights of the neural network, up to a maximum of maxit cycles. This iterative process may take a long time to compute for large datasets.
The above function call creates a neural network with 17 input units (predictor variables) connected to 10 hidden units, which will then be linked to a single output unit. We can see the final weights of these connections by issuing,

```
> summary(nn)
> # You could also use wts to get the best weights found and fitted.values to get the fitted values on training data
> nn$wts
> nn$fitted.values
```

This neural net can be used to make predictions for our test period,

```
> nn.preds = predict(nn, TestData)
```

Notice we still have results between 0 and 1 that are more like probabilities of belonging to each class. To get the predicted classes we can use change the type argument.

```
>nn.preds = predict(nn, TestData, type = "class")
```

Now let's create a simple confusion matrix

```
>table(TestData$Private, nn.preds)
```

```
     nn.preds
       No Yes
No     54  11
Yes    12 161
```

Notice that all variables used in neural network model should be continuous. You can use model.matrix() function that transfer all the factor variable to a set of dummy variables.

We can also use the neuralnet package to construct a neural network model.

```
> install.packages("neuralnet")
> require(neuralnet)
```

The function neuralnet() from this package is used for training a neural network. This function 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 (response ~ inputs).
- data: a data frame containing the variables specified in formula.
- 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

We use the data set infert that is provided by the package datasets to illustrate the neuralnet function. This data set contains data of a case-control study that investigated infertility after spontaneous and induced abortion. 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.

1. Education	0 = 0-5 years
	1 = 6-11 years
	2 = 12+ years
2. age	age in years of case
3. parity	count
4. number of prior	0 = 0
induced abortions	1 = 1
	2 = 2 or more
5. case status	1 = case
	0 = control
6. number of prior	0 = 0
spontaneous abortions	1 = 1
	2 = 2 or more
7. matched set number	1-83
8. stratum number	1-63

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)
> nn = neuralnet(case ~ age+parity+induced+spontaneous, data=infert, hidden=2, err.fct="ce", linear.output=FALSE)
```

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 in provided by nn$result.matrix

```
                                  1
error                   158.085555448697
reached.threshold         0.002114813655
steps                    34.000000000000
Intercept.to.1layhid1     0.233498761639
age.to.1layhid1           3.195901197857
parity.to.1layhid1       -2.729680341227
induced.to.1layhid1      -0.840795801424
spontaneous.to.1layhid1   0.674028668650
Intercept.to.1layhid2     2.942726509671
age.to.1layhid2           1.652164203840
parity.to.1layhid2        0.308431564445
induced.to.1layhid2       3.929388169168
spontaneous.to.1layhid2  -0.309789104102
Intercept.to.case        -1.698515087905
1layhid.1.to.case        -0.628639390613
1layhid.2.to.case         1.640087908907
```

The training process needed 34 steps until all absolute partial derivatives of the error function were smaller than 0.01 (the default threshold). The estimated weights range from −2.72 to 3.92. For instance, the intercepts of the first hidden layer are 0.23 and 2.94 and the four weights leading to the first hidden neuron are estimated as 3.2, -2.7, −0.84, and 0.67 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$data for the whole data set inclusive non-used variables. The output of the neural network, i.e. the fitted values, is provided by nn$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 0.3346859468
[2,]  42      1       1           0 0.3346859468
[3,]  39      6       2           0 0.3346859468
[4,]  34      4       2           0 0.3346859468
[5,]  35      3       1           1 0.3346859468
[6,]  36      4       2           1 0.3346859468
```

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.

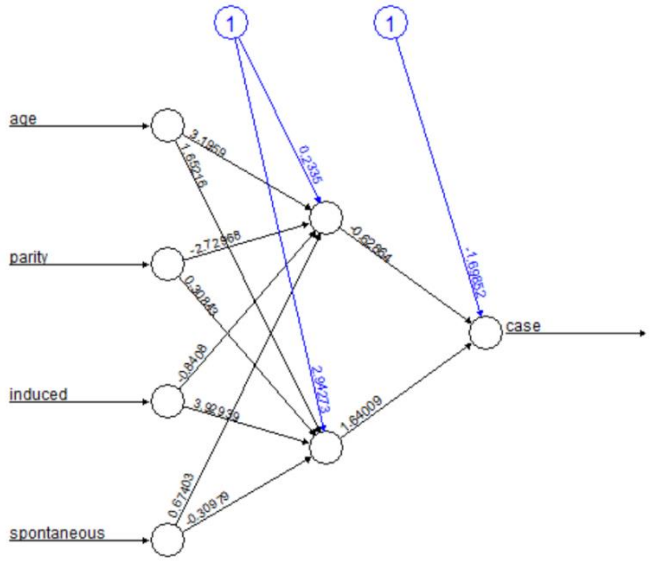
To compare the results, neural networks are trained with the same parameter setting as above using neuralnet with algorithm="backprop" and the package nnet.

```
> 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
> nn.nnet = nnet( case ~ age + parity + induced + spontaneous, data = infert, size=2, entropy=T,
  abstol=0.01)
```

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.

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

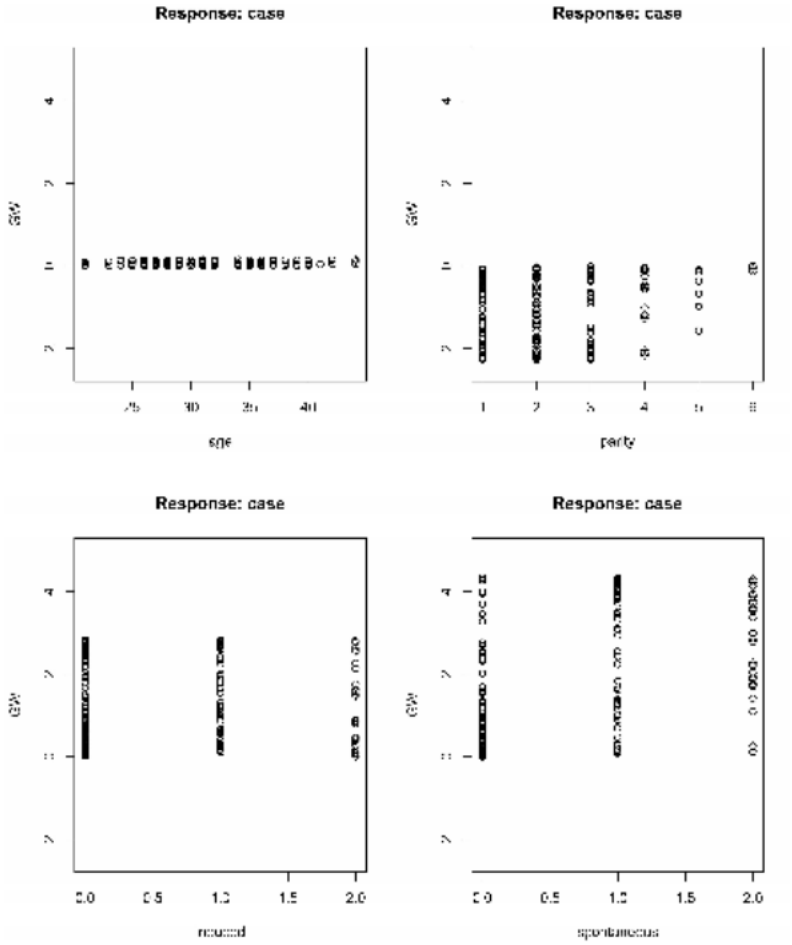
```
>plot(nn)
```



Error: 158.085555 Steps: 34

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 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.

The mlp() (multi-layer perceptron) function from RSNNS package can be also used for constructing a neural network model.

```
> # mlp(Vector of Inputs, Target variable, size = 5, maxit = 100, linOut = T, learnFunc =
  "Std_Backpropagation")
```

> # size = number of units in the hidden layer(s). maxit = maximum number of iterations to learn, learnFunc = function used for learning, linOut sets the activation function of the output units to linear or logistic