Appendix R11: Deep Learning

Neural Networks

J. Alberto Espinosa

2/3/2023

Contents


```
RNGkind(sample.kind = "default")
```

Basic Prep Work

I will use the **Boston** housing house values data set from the **{MASS}** library. I will also store the model specification formula in formula object named **formula.medv**, so that I can reuse the same formula in the various models later on.

```
library(MASS) # Contains the Needed Boston housing data set

# Store the model formula to re-use later

formula.medv <- medv ~ lstat + crim + age + chas
```

NN Training Process

Neural networks are trained by: (1) selecting arbitrary weights (i.e., coefficients) for all the inputs (i.e., predictors); (2) predicting the outcome; (3) evaluating the cross-validation (CV) cost (i.e., test error or deviance) of the predictions; (4) back propagation to make adjustments to the input weights; and (5) doing it again many times. The starting points are usually extreme values, that is very small weight and very large weights. Each iteration increases the small weights by a fraction and reduces the large weighs by a fraction, which is expected to progressively reduce the model cost. This process is called **gradient descent** and the idea is that the model converges to a solution as we further adjustments the input weights – i.e., the gradient (i.e., derivative or tangent) of the cost curve approximates zero.

The Min-Max Normalization Method

Using the raw data can be problematic because of scale issues. Variables with large values will have a stronger influence on the initial set of random weights selected and on the gradient descent process than variables with small values. This may cause the final weights to vary sharply and the model may not even converge to a solution. This problem is minimize if we normalize the data to a similar scale. There are various normalization methods, with the two most popular being **standardization** (i.e., z-scores with a mean of 0 an standard deviation of 1) or a -1 to +1 scale normalization using a **Min-Max** function. You can standardize the data with z-scores (i.e., center the variables and divide by their standard deviation), but this affects dummy variables. If you don't have any dummy variables, standardizing with z-scores is OK. If there are dummy variables, you can normalize all other predictors, except dummy variables. A simpler approach is to use other popular normalization methods, like Max-Min, in which the variable is transformed as a deviation (difference) from the minimum value for that variable, divided by the largest difference between maximum and minimum values, which does not affect dummy variables. Most experts recommend the **Min-Max** normalization method, which is what I use in these illustrations.

In the **Min-Max method, the smallest value of x is subtracted from the highest value of x, yielding the range or span of the data. The Min-Max value of a variable is obtained by subtracting the smallest value of x from the raw value of the variable, divided by the Min-Max range of the data. This ensures that all variable are normalized to a - 1 to + 1 scale. To do this efficiently, we can create a function:

```
normalize <- function(x) {
  return ((x - min(x)) / (max(x) - min(x)))
}
```

This way, if we want to normalize a value of a variable named `lstat`, all we need to do is compute `lstat.n <- normalize(lstat)`. We can also use the `lapply()` function to normalize the entire data set.

```
Boston.n <- as.data.frame(lapply(Boston, normalize))
```

Prep the Data

Important Note about Dummy and categorical variables. `neuralnet()` requires quantitative predictors. If you have a factor dummy variable, convert it to numeric, an if you have a categorical variable, transform the variable to the respective binary variables, in numeric format. This can be done by hand or using the model.matrix() function. For example, if your data set is called `my.data` and the model formula is `y ~ x1 + x2 + x3 + etc.`, and some of these predictors are categorical, this command will convert the categorical x's into the respective binary variables:

```
my.data.q <- model.matrix(~ x1 + x2 + x3 + etc.)
```

But you then need to add the outcome to the dataset as follows:

```
my.data.q$y <- my.data$y
```

Now, let's split the data into train and test subset. The NN training methods don't need this subset splitting, but we will compare models using **Random Splitting Cross Validation** for illustration purposes.

```
set.seed(1) # Set an arbitrary seed

tr.size <- 0.7 # Set an arbitrary train sample size

train <- sample(1:nrow(Boston.n),
                tr.size * nrow(Boston.n))

Boston.n.train <- Boston.n[train, ] # Train subset
Boston.n.test <- Boston.n[-train, ] # Test subset
```

Multiple Linear Regression (for comparison)

I start by training a plain OLS regression model, so that we can make some comparisons with the **Neural Network (NN)** outcomes.

```
lm.fit <- lm(formula.medv, data = Boston.n.train)

summary(lm.fit) # Display regression output summary
```

```
##
## Call:
## lm(formula = formula.medv, data = Boston.n.train)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.29798 -0.08964 -0.03139  0.04503  0.49409
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.60274    0.01859  32.414  < 2e-16 ***
## lstat       -0.82020    0.05148 -15.932  < 2e-16 ***
```

```
## crim        -0.09668    0.09443  -1.024    0.307
## age          0.05003    0.03183   1.572    0.117
## chas         0.11665    0.02729   4.274 2.48e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1382 on 349 degrees of freedom
## Multiple R-squared:  0.5815, Adjusted R-squared:  0.5767
## F-statistic: 121.2 on 4 and 349 DF,  p-value: < 2.2e-16
```

Compare actual against predicted values for the first few records

```
head(data.frame("Actual" = Boston.n.test$medv,
                "Predicted" = predict(lm.fit, Boston.n.test)))
```

```
##       Actual  Predicted
## 5  0.6933333 0.54762311
## 6  0.5266667 0.55270020
## 7  0.3977778 0.39330138
## 8  0.4911111 0.25635040
## 9  0.2555556 0.01431038
## 10 0.3088889 0.29746357
```

```
pred.mdev <- predict(lm.fit, Boston.n.test)

mse.lm <- round(mean((Boston.n.test$medv - pred.mdev)^2 ),
                digits = 4)

paste("The OLS RSCV Test MSE is", mse.lm)
```

```
## [1] "The OLS RSCV Test MSE is 0.0161"
```

Neural Networks

neuralnet() Function and Arguments Overview

There are many libraries and function in R to train neural networks, including `{neuralnet}`, `{nnet}`, `{deepnet}`, `{h2o}`, `{MXNET}`, `{tensorflow}`, `{MXNet}`, etc. You can also train NNs with the `{caret}` package using any of these methods. In this illustration I will use the `neuralnet()` function in the `{neuralnet}` library, which allows for multiple layers and renders good NN diagrams.

Notes on `neuralnet()` key attributes (the values shown are the defaults, which you can omit to accept the default, or change as needed):

- `hidden = 1` - a vector with the number of neurons in each layer, e.g., `hidden = 1` is the default, which is 1 layer with 1 neuron. `hidden = 3` has 1 layer with 3 neurons; `hidden=c(4, 2)` has 2 layers, the first one with 4 neuribs and the second one with 2.

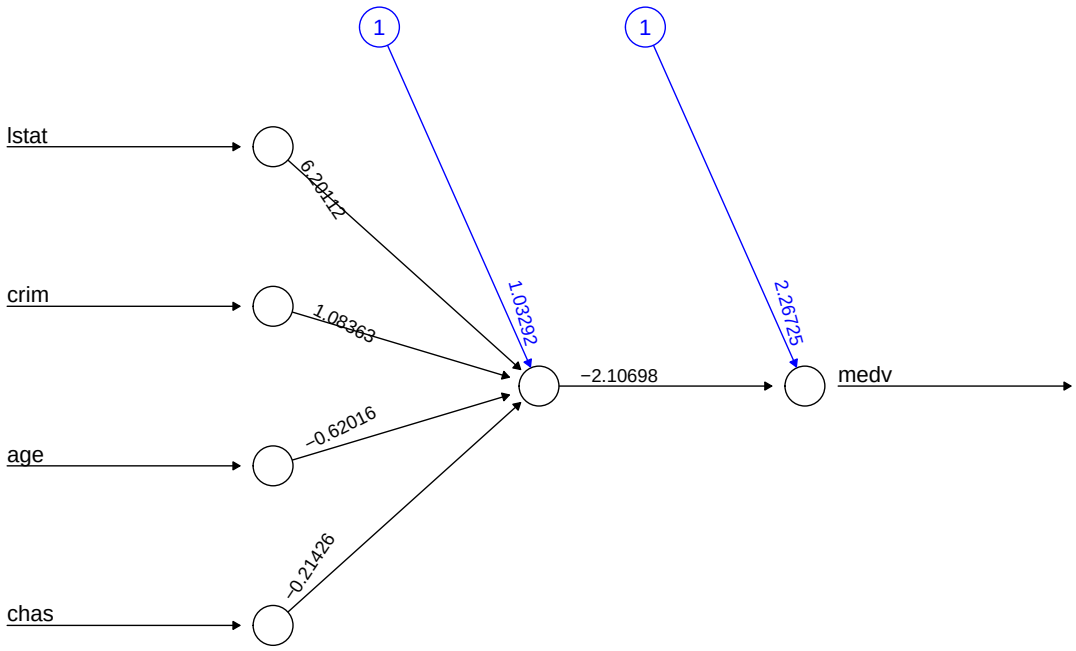
- `threshold = 0.01` - is an approximation threshold used in gradient descent derivatives. The default value is 0.01, which is also the smallest value needed for `neuralnet()` to work. If the model does not converge, then increase the threshold progressively to 0.1, 0.2, etc. If the model still does not converge, increase it to 1, 10, 100, etc.
- `stepmax = 10 ^ 5` - is the maximum number of steps for training a neural network. Each epoch or iteration requires a number of steps to complete. The model training will stop if the model has not converged after the stepmax. Increasing the value of stepmax will make it more likely that the model will converge, but it will take substantially longer.
- `rep = 1` - the number of times you want the neuralnet training to run. Usually, 1 is sufficient, but due to the random nature of training NN's, if you want more randomness and are willing to wait, you can use more reps.
- `algorithm = "rprop+"` - is the internal algorithm to calculate the neural network. Available methods include: "rprop+" (resilient back propagation), "rprop-", "backprop" (back propagation), "sag", "slr". The default usually works fine.
- `err.fct = "sse"` - is the method used to calculate the cost or error. You can use `"ce"` for cross-entropy, which can be used for classification models.
- act.fct = "logistic"`- is the activation function;`"logistic"`and`"tanh"`‘ are popular functions for classification models.
- `linear.output = T` - leave default as `T` for **quantitative** models or change to `F` for **classification** models, along with the act.fct activation function

Training NNs with a Quantitative Outcome

Let's start by loading the `{neuralnet}` library and training a NN with only 1 neuron and 1 layer (i.e., we omit the `hidden = 1` parameter because it is the default.

Technical Note: Notice that I used the parameter `rep = "best"` in the `plot()` function. The reason for this is that, even if you only us `rep = 1`, `neuralnet()` doesn't know which repetition to graph. So, if you render the plot from the code chunk by pressing the play icon, the graph will render fine, but will not print when you knit into a document. Adding the `rep = "best"` parameter tells knitr to graph the best of the repetitions. In a nutshell, if you want the graphs to print in the knitted document, you must include this parameter in the `plot()` function.

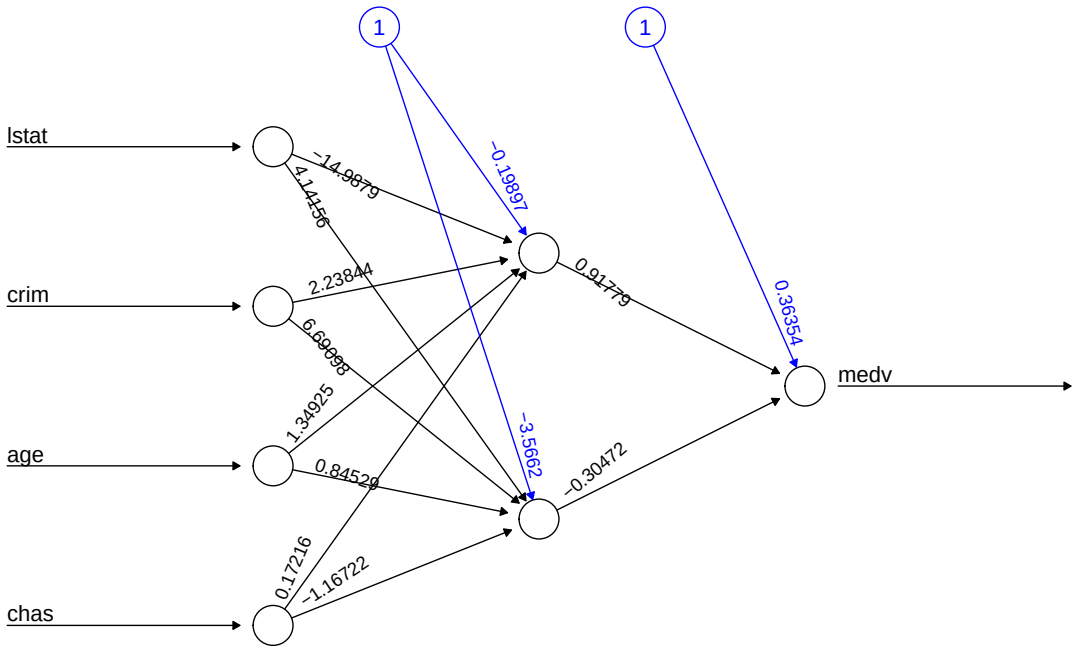
```
library(neuralnet)
# 1 Neuron, 1 Layer

net.fit.1 <- neuralnet(formula.medv,
                       data = Boston.n,
                       threshold = 0.01)

plot(net.fit.1, rep = "best")
```



Error: 3.092343 Steps: 14555

Let's now train a NN with 2 neurons in 1 layer.

```
net.fit.2 <- neuralnet(formula.medv,
                       data = Boston.n,
                       hidden = 2,
                       threshold = 0.01)

plot(net.fit.2, rep = "best")
```

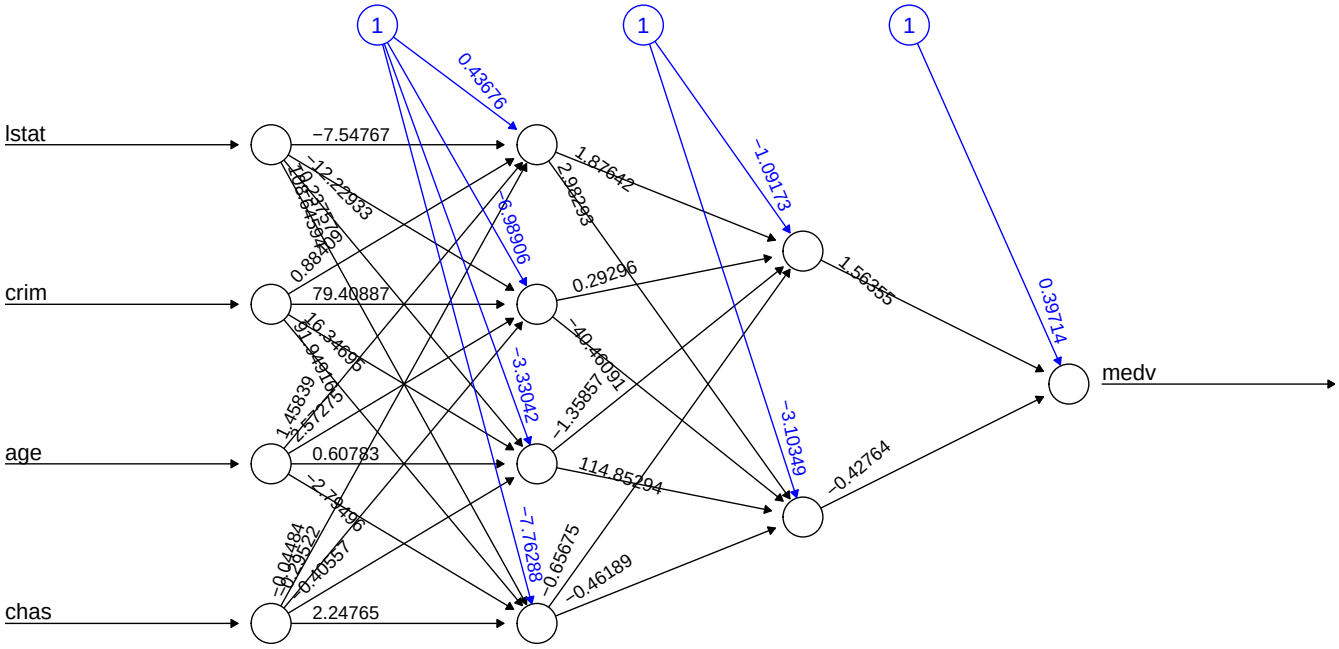


Error: 2.797015 Steps: 2258

Let's now try a deeper learning NN with 2 layers, the first one with 4 Neurons, and the second one with 2 Neurons.

Technical Note: If the algorithm does not converge increase the `threshold` to 0.1, 0.2, or even larger and/or increase the `stepmax` to a large value, for example `10 ^ 5`. Note that this will increase the computational time substantially.

```
net.fit.4.2 <- neuralnet(formula.medv,
                         data = Boston.n,
                         hidden = c(4, 2),
                         threshold = 0.01,
                         stepmax = 10 ^ 5)

plot(net.fit.4.2, rep = "best")
```



Error: 2.329962 Steps: 8323

Training, Predicting and Testing NNs with a Quantitative Outcome

We use the {neuralnet}predict() function and test sub-sample to make predictions:

```
pred.nnet <- predict(net.fit.4.2, Boston.n.test)

pred.nnet[1:15] # Display a few predictions
```

```
##  [1] 0.5514481 0.5846659 0.3423339 0.2584102 0.2219462 0.2778396 0.2470746
##  [8] 0.3379197 0.4220022 0.3004611 0.2280545 0.2738761 0.3656999 0.3568591
## [15] 0.2524821
```

```
data.frame("Actual" = Boston.n.test$medv,
           "Predicted" = pred.nnet,
           "Difference" = Boston.n.test$medv - pred.nnet)[1:15, ]
```

```
##       Actual Predicted  Difference
## 5  0.6933333 0.5514481  0.14188519
## 6  0.5266667 0.5846659 -0.05799923
## 7  0.3977778 0.3423339  0.05544389
## 8  0.4911111 0.2584102  0.23270088
## 9  0.2555556 0.2219462  0.03360934
## 10 0.3088889 0.2778396  0.03104925
## 11 0.2222222 0.2470746 -0.02485237
## 12 0.3088889 0.3379197 -0.02903078
```

```
## 17 0.4022222 0.4220022 -0.01978002
## 18 0.2777778 0.3004611 -0.02268328
## 21 0.1911111 0.2280545 -0.03694334
## 26 0.1977778 0.2738761 -0.07609834
## 30 0.3555556 0.3656999 -0.01014438
## 32 0.2111111 0.3568591 -0.14574797
## 34 0.1800000 0.2524821 -0.07248214
```

Now lets compute the **RSCV Test MSE** and display it side by side with the OLS result:

```
mse.nnet <- mean( (Boston.n.test$medv - pred.nnet) ^ 2)
mse.nnet <- round(mse.nnet, digits = 4)

cbind("OLS MSE" = mse.lm,
      "Neural Net MSE" = mse.nnet)
```

```
##      OLS MSE Neural Net MSE
## [1,]  0.0161         0.0083
```

Training NNs with a Classification Outcome

Let's first read the data

```
heart <- read.table("Heart.csv",
                    sep = ",",
                    header = T,
                    stringsAsFactors = T)
```

Let's now do some data prep work. First, let convert factor variables to numeric

```
heart$famhist <- as.numeric(heart$famhist)
```

Next, let's normalize the data with the `normalize()` function defined above.

```
heart.n <- as.data.frame(lapply(heart, normalize))
```

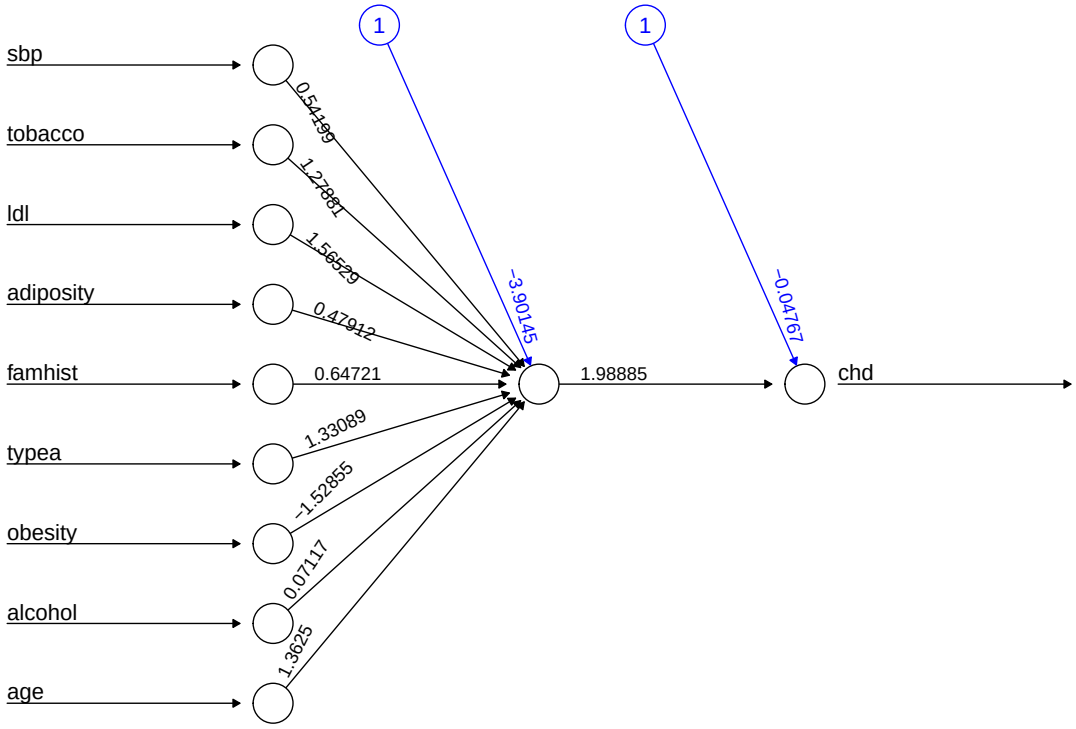
Then, we need to specifying the model specification formula object, for convenience

```
formula.chd <- chd ~ sbp + tobacco + ldl +
                     adiposity + famhist + typea +
                     obesity + alcohol + age
```

Technical Note: the `neuralnet()` algorithm below may take a long time to run and will not necessary converge. You can manipulate the `threshold` and `stepmax` if you are having issues getting your model to converge, or you can reduce the number of hidden layers and nodes. Of course, increasing the `stepmax` and `threshold` increase substantially the amount of time needed to estimate the NN.

Let's start by loading the `{neuralnet}` library and training a NN with only 1 neuron and 1 layer (i.e., we omit the `hidden = 1` parameter). Notice that we use the parameter `act.fct = "logistic"` to use the logit activation function.

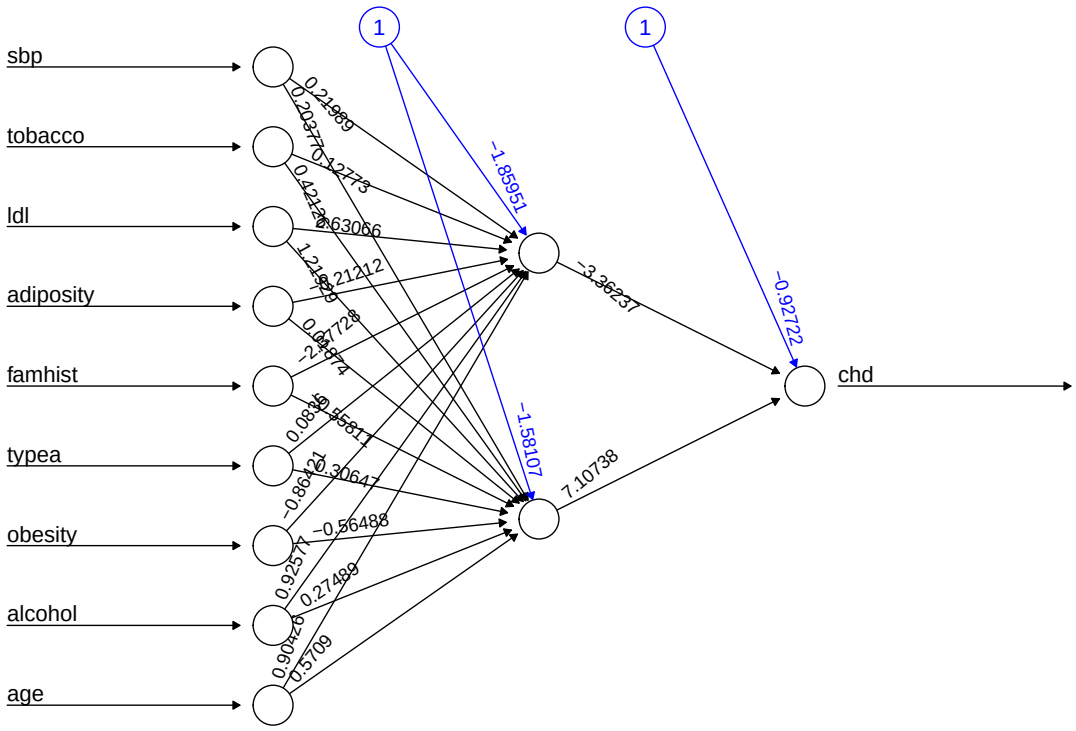
```
heart.nnet.1 <- neuralnet(formula.chd,
                          data = heart.n,
                          threshold = 0.01,
                          act.fct = "logistic")

plot(heart.nnet.1, rep = "best")
```



Error: 39.280415 Steps: 1256

Let's now train a NN with 2 neurons in 1 layer.

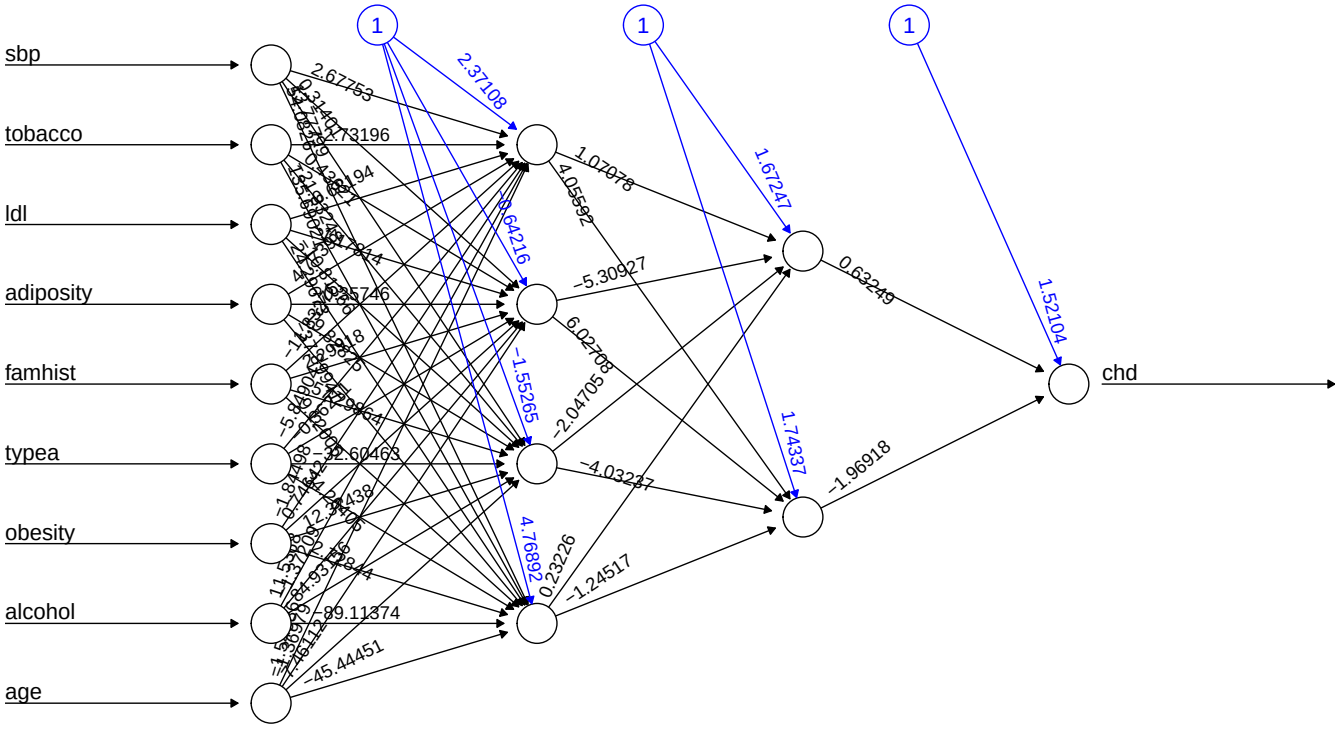
```
heart.nnet.2 <- neuralnet(formula.chd,
                          data = heart.n,
                          hidden = 2,
                          threshold = 0.01,
                          act.fct = "logistic")

plot(heart.nnet.2, rep = "best")
```



Let's now try a deeper learning NN with 2 layers, the first one with 4 Neurons, and the second one with 2 Neurons.

Technical Note: If the algorithm does not converge increase the `threshold` to 0.1, 0.2. I increased it to 0.5 because this model did not converge a couple of times. You can also increase the `stepmax` to a large value, for example `10 ^ 5`. Note that this will increase the computational time substantially.

```
heart.nnet.4.2 <- neuralnet(formula.chd,
                            data = heart.n,
                            hidden = c(4,2),
                            threshold = 0.05,
                            act.fct = "logistic")

plot(heart.nnet.4.2, rep = "best")
```



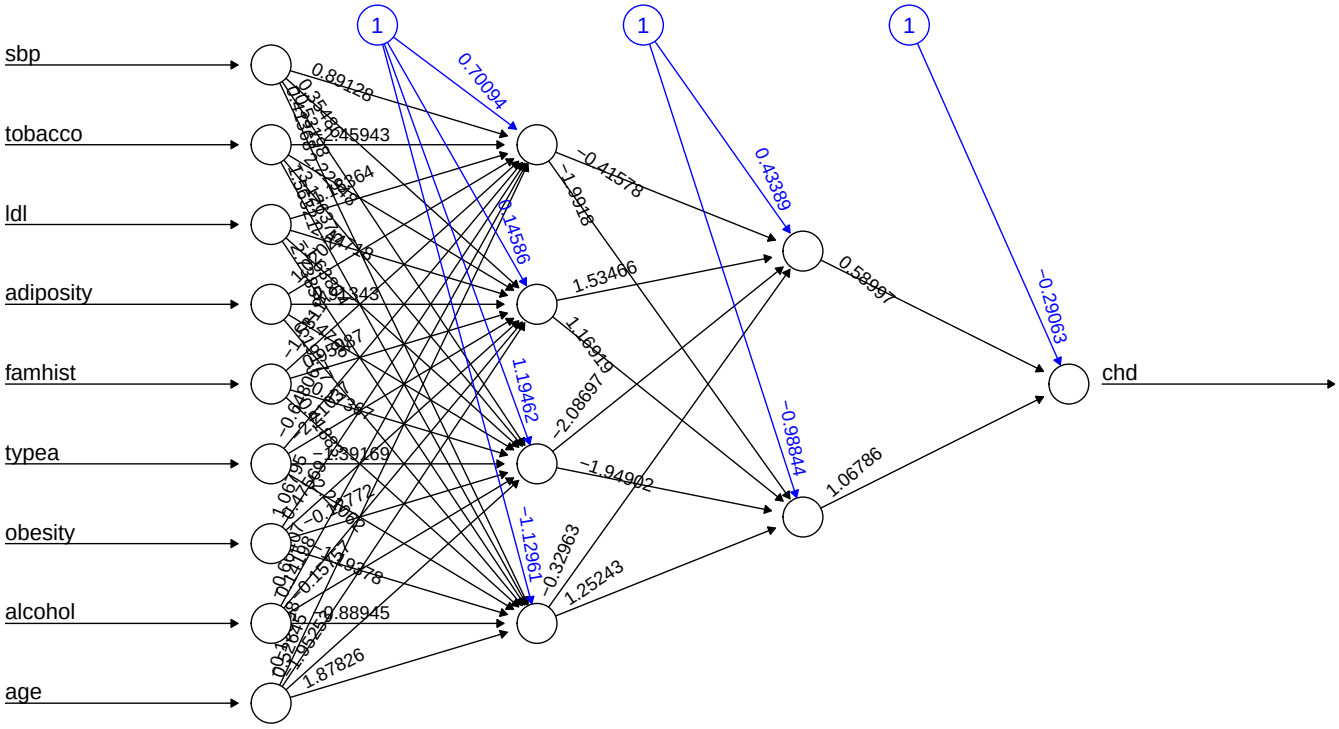
Training, Predicting and Testing NNs with a Classification Outcome

Firs, let's split the data into train and test subsets

```
set.seed(1)

tr.size <- 0.7

train <- sample(1:nrow(heart.n),
                tr.size * nrow(heart.n))

heart.n.train <- heart.n[train,]
heart.n.test <- heart.n[-train,]
```

Tran the NN with the train subset

```
heart.nnet.4.2 <- neuralnet(formula.chd,
                            data = heart.n.train,
                            hidden = c(4,2),
                            threshold = 0.5,
                            act.fct = "logistic")

plot(heart.nnet.4.2, rep = "best")
```



Error: 26.71112 Steps: 59

Now, let's predict **probability** outcomes and convert them to classifications using a threshol of 0.5.

```
pred.prob <- predict(heart.nnet.4.2, heart.n.test)

thresh <- 0.5 # Set the classification threshold

pred.prob.class <- ifelse(pred.prob > thresh, 1, 0)
```

Let's display a few results

```
results <- cbind(round(pred.prob, digits = 3),
                 pred.prob.class,
                 heart.n.test$chd)[1:15, ]

colnames(results) <- c("Probability",
                       "Classification",
                       "Actual")

results
```

```
##    Probability Classification Actual
## 5        0.722              1      1
## 6        0.549              1      0
## 7        0.215              0      0
## 8        0.661              1      1
## 9        0.228              0      0
## 10       0.438              0      1
```

```
## 11       0.304              0      1
## 17       0.815              1      0
## 18       0.760              1      1
## 21       0.028              0      1
## 30       0.347              0      1
## 32       0.276              0      1
## 34       0.371              0      1
## 46       0.251              0      0
## 47       0.829              1      1
```

Confusion Matrix

Let's now use the RSCV testing results to build a cross-validation confusion matri and display it.

```
conf.mat <- table("Predicted" = pred.prob.class,
                  "Actual" = heart.n.test$chd)

conf.mat # Display matrix
```

```
##           Actual
## Predicted  0  1
##         0 75 31
##         1 15 18
```

Let's now compute the fit statistics

```
TruN <- conf.mat[1,1] # True negatives
TruP <- conf.mat[2,2] # True positives
FalN <- conf.mat[1,2] # False negatives
FalP <- conf.mat[2,1] # False positives

TotN <- TruN + FalP # Total actual negatives
TotP <- TruP + FalN # Total actual positives

TotNpr <- TruN + FalN # Total negative predictions
TotPpr <- TruP + FalP # Total positive predictions

Tot <- TotN + TotP # Total

# Do a quick check of the computations
cbind(TruN, TruP, FalN, FalP, TotN, TotP, TotNpr, TotPpr, Tot)
```

```
##      TruN TruP FalN FalP TotN TotP TotNpr TotPpr Tot
## [1,]   75   18   31   15   90   49    106     33 139
```

Let's add totals and labels to the confusion matrix. Notice that I use the `knitr::kable()` function to render the table with a nicer format.

```
conf.mat.totals <- cbind(conf.mat,
                         c(TotNpr, TotPpr))

conf.mat.totals <- rbind(conf.mat.totals,
                         c(TotN, TotP, Tot))

colnames(conf.mat.totals) <-
  rownames(conf.mat.totals) <-
     c("No","Yes", "Total")

knitr::kable(conf.mat.totals,
             format = "simple",
             caption = "Confusion Matrix, Prob Thresh > 0.5")
```

Table 1: Confusion Matrix, Prob Thresh > 0.5

	No	Yes	Total
No	75	31	106
Yes	15	18	33
Total	90	49	139

Confusion Matrix Accuracy and Error Rates

```
Accuracy.Rate <- (TruN + TruP) / Tot
Error.Rate <- (FalN + FalP) / Tot
Sensitivity <- TruP / TotP
Specificity <- TruN / TotN
FalseP.Rate <- 1 - Specificity
```

We can now label the results and display them

```
nnet.rates.50 <- round(c(Accuracy.Rate, Error.Rate,
                         Sensitivity, Specificity,
                         FalseP.Rate),
                       digits = 3)

names(nnet.rates.50) <- c("Accuracy", "Error",
                          "Sensitivity", "Specificity",
                          "False Positives")

nnet.rates.50
```

```
##        Accuracy           Error     Sensitivity     Specificity False Positives
##           0.669           0.331           0.367           0.833           0.167
```

Try on your own: change the classification threshold from prob > 0.5 to > 0.3 and then to > 0.7 and compute the respective confusion matrices and fit statistics and see how sensitivity and specificity change.

Also, try fitting a classification tree and a logistic model with the same data and compare the fit statistics across all 3 models. You will need to use thie formula for the pred.prob.class:

`pred.prob.class <- ifelse(pred.prob > 0.3, 1, 0)` and `pred.prob.class <- ifelse(pred.prob > 0.7, 1, 0)`