Fitting a neural network in R

Bankbintje

21 februari 2016

Dit is een samenvatting van een artikel op R-bloggers:

<http://www.r-bloggers.com/fitting-a-neural-network-in-r-neuralnet-package/>.

# The dataset

We are going to use the Boston dataset in the MASS package. The Boston dataset is a collection of data about housing values in the suburbs of Boston. Our goal is to predict the median value of owner-occupied homes (medv) using all the other continuous variables available

require("MASS")

## Loading required package: MASS

set.seed(500)  
library(MASS)  
data <- Boston

First we need to check that no datapoint is missing, otherwise we need to fix the dataset.

apply(data,2,function(x) sum(is.na(x)))

## crim zn indus chas nox rm age dis rad   
## 0 0 0 0 0 0 0 0 0   
## tax ptratio black lstat medv   
## 0 0 0 0 0

There is no missing data, good. We proceed by randomly splitting the data into a train and a test set, then we fit a linear regression model and test it on the test set. Note that I am using the gml() function instead of the lm() this will become useful later when cross validating the linear model.

index <- sample(1:nrow(data),round(0.75\*nrow(data)))  
## create training set  
train <- data[index,]  
## create test set  
test <- data[-index,]  
## fit the model  
lm.fit <- glm(medv~., data=train)

Summarize the model

summary(lm.fit)

##   
## Call:  
## glm(formula = medv ~ ., data = train)  
##   
## Deviance Residuals:   
## Min 1Q Median 3Q Max   
## -14.9143 -2.8607 -0.5244 1.5242 25.0004   
##   
## Coefficients:  
## Estimate Std. Error t value Pr(>|t|)   
## (Intercept) 43.469681 6.099347 7.127 5.50e-12 \*\*\*  
## crim -0.105439 0.057095 -1.847 0.065596 .   
## zn 0.044347 0.015974 2.776 0.005782 \*\*   
## indus 0.024034 0.071107 0.338 0.735556   
## chas 2.596028 1.089369 2.383 0.017679 \*   
## nox -22.336623 4.572254 -4.885 1.55e-06 \*\*\*  
## rm 3.538957 0.472374 7.492 5.15e-13 \*\*\*  
## age 0.016976 0.015088 1.125 0.261291   
## dis -1.570970 0.235280 -6.677 9.07e-11 \*\*\*  
## rad 0.400502 0.085475 4.686 3.94e-06 \*\*\*  
## tax -0.015165 0.004599 -3.297 0.001072 \*\*   
## ptratio -1.147046 0.155702 -7.367 1.17e-12 \*\*\*  
## black 0.010338 0.003077 3.360 0.000862 \*\*\*  
## lstat -0.524957 0.056899 -9.226 < 2e-16 \*\*\*  
## ---  
## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1  
##   
## (Dispersion parameter for gaussian family taken to be 23.26491)  
##   
## Null deviance: 33642 on 379 degrees of freedom  
## Residual deviance: 8515 on 366 degrees of freedom  
## AIC: 2290  
##   
## Number of Fisher Scoring iterations: 2

Predict the values of the test set

pr.lm <- predict(lm.fit,test)

Calculate the MSE of the test set. Since we are dealing with a regression problem, we are going to use the mean squared error (MSE) as a measure of how much our predictions are far away from the real data.

MSE.lm <- sum((pr.lm - test$medv)^2)/nrow(test)  
MSE.lm

## [1] 21.62976

# Preparing to fit the neural network

Before fitting a neural network, some preparation need to be done. Neural networks are not that easy to train and tune.

As a *first step*, we are going to address data preprocessing.

It is good practice to normalize your data before training a neural network. I cannot emphasize enough how important this step is: depending on your dataset, avoiding normalization may lead to useless results or to a very difficult training process (most of the times the algorithm will not converge before the number of maximum iterations allowed). You can choose different methods to scale the data (z-normalization, min-max scale, etc…). I chose to use the min-max method and scale the data in the interval [0,1]. Usually scaling in the intervals [0,1] or [-1,1] tends to give better results. We therefore scale and split the data before moving on:

maxs <- apply(data, 2, max)   
mins <- apply(data, 2, min)  
  
scaled <- as.data.frame(scale(data, center = mins, scale = maxs - mins))  
  
train\_ <- scaled[index,]  
test\_ <- scaled[-index,]

# Parameters

As far as I know there is no fixed rule as to how many layers and neurons to use although there are several more or less accepted rules of thumb. Usually, if at all necessary, one hidden layer is enough for a vast numbers of applications. As far as the number of neurons is concerned, it should be between the input layer size and the output layer size, usually 2/3 of the input size. At least in my brief experience testing again and again is the best solution since there is no guarantee that any of these rules will fit your model best.

Since this is a toy example, we are going to use 2 hidden layers with this configuration: 13:5:3:1. The input layer has 13 inputs, the two hidden layers have 5 and 3 neurons and the output layer has, of course, a single output since we are doing regression.

Let’s fit the net:

require("neuralnet")

## Loading required package: neuralnet  
## Loading required package: grid

library(neuralnet)  
n <- names(train\_)  
f <- as.formula(paste("medv ~", paste(n[!n %in% "medv"], collapse = " + ")))  
nn <- neuralnet(f,data=train\_,hidden=c(5,3),linear.output=T)

A couple of notes:

* For some reason the formula y~. is not accepted in the neuralnet() function. You need to first write the formula and then pass it as an argument in the fitting function.
* The hidden argument accepts a vector with the number of neurons for each hidden layer, while the argument linear.output is used to specify whether we want to do regression linear.output=TRUE or classification linear.output=FALSE

The neuralnet package provides a nice tool to plot the model.This is the graphical representation of the model with the weights on each connection:

plot(nn)

The black lines show the connections between each layer and the weights on each connection while the blue lines show the bias term added in each step. The bias can be thought as the intercept of a linear model.

The net is essentially a black box so we cannot say that much about the fitting, the weights and the model. Suffice to say that the training algorithm has converged and therefore the model is ready to be used.