

Fitting a neural network in R; neuralnet package

[Michy Alice](#)

Neural networks have always been one of the most fascinating machine learning model in my opinion, not only because of the fancy backpropagation algorithm, but also because of their complexity (think of deep learning with many hidden layers) and structure inspired by the brain. Neural networks have not always been popular, partly because they were, and still are in some cases, computationally expensive and partly because they did not seem to yield better results when compared with simpler methods such as support vector machines (SVMs). Nevertheless Neural Networks have, once again, raised attention and become popular.

In this post we are going to fit a simple neural network using the neuralnet package and fit a linear model as a comparison.

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.

```
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)))
```

<i>crim</i>	<i>zn</i>	<i>indus</i>	<i>chas</i>	<i>nox</i>	<i>rm</i>	<i>age</i>	<i>dis</i>	<i>rad</i>	<i>tax</i>
<i>ptratio</i>									
0	0	0	0	0	0	0	0	0	0
0									
<i>black</i>	<i>lstat</i>	<i>medv</i>							
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 `glm()` 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)))
train <- data[index,]
test <- data[-index,]
lm.fit <- glm(medv~., data=train)
summary(lm.fit)
pr.lm <- predict(lm.fit,test)
MSE.lm <- sum((pr.lm - test$medv)^2)/nrow(test)
```

The `sample(x,size)` function simply outputs a vector of the specified size of randomly selected samples from the vector `x`. By default the sampling is without replacement: `index` is essentially a random vector of indices.

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.

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,]
```

Note that `scale` returns a matrix that needs to be coerced into a `data.frame`.

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:

```
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)
```

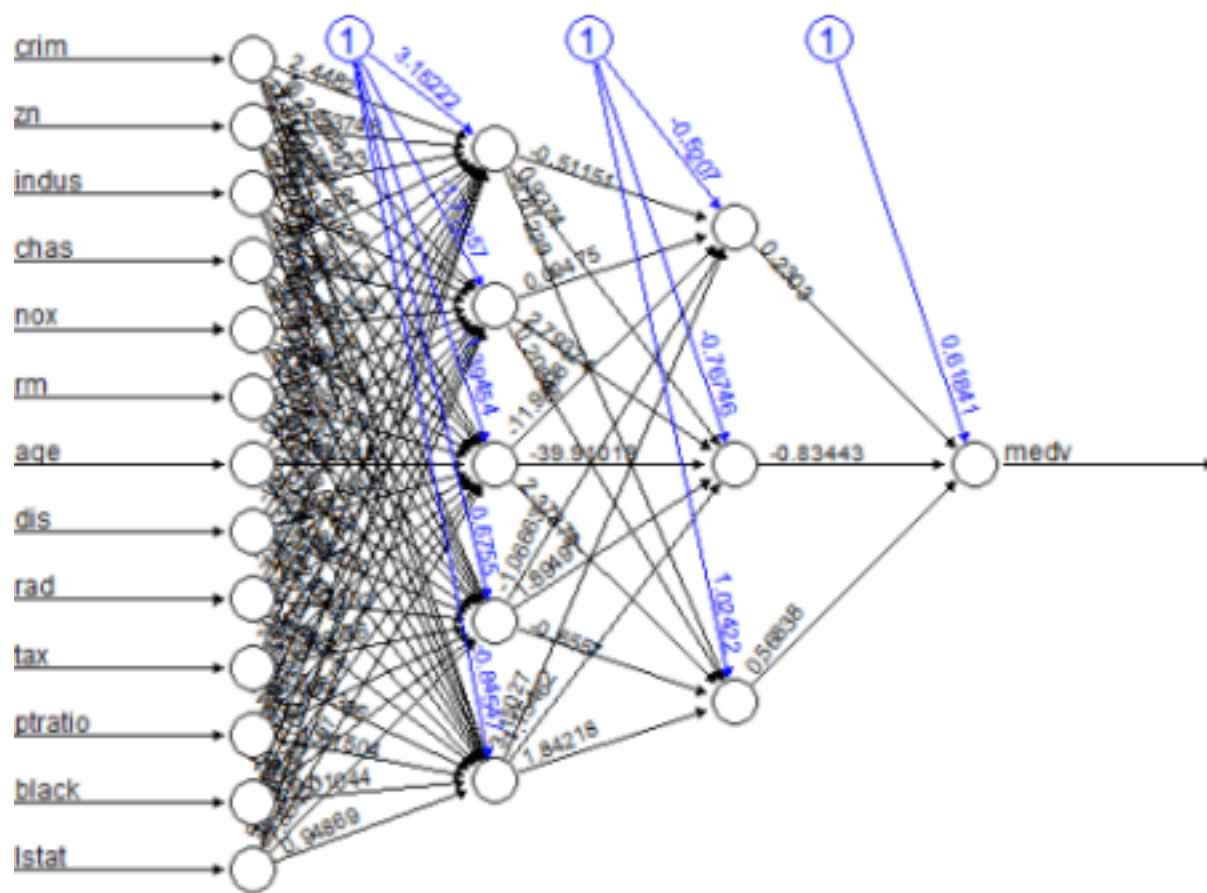
A couple of notes:

- For some reason the formula $y \sim .$ 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:

```
plot(nn)
```

This is the graphical representation of the model with the weights on each connection:



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.

Predicting medv using the neural network

Now we can try to predict the values for the test set and calculate the MSE. Remember that the net will output a normalized prediction, so we need to scale it back in order to make a meaningful comparison (or just a simple prediction).

```
pr.nn <- compute(nn,test_[,1:13])
```

```
pr.nn_ <- pr.nn$net.result*(max(data$medv)-min(data$medv))+min(data$medv)
test.r <- (test_$medv)*(max(data$medv)-min(data$medv))+min(data$medv)
```

```
MSE.nn <- sum((test.r - pr.nn_)^2)/nrow(test_)
```

we then compare the two MSEs

```
print(paste(MSE.lm,MSE.nn))
```

```
"21.6297593507225 10.1542277747038"
```

Apparently the net is doing a better work than the linear model at predicting medv. Once again, be careful because this result depends on the train-test split performed above. Below, after the visual plot, we are going to perform a fast cross validation in order to be more confident about the results.

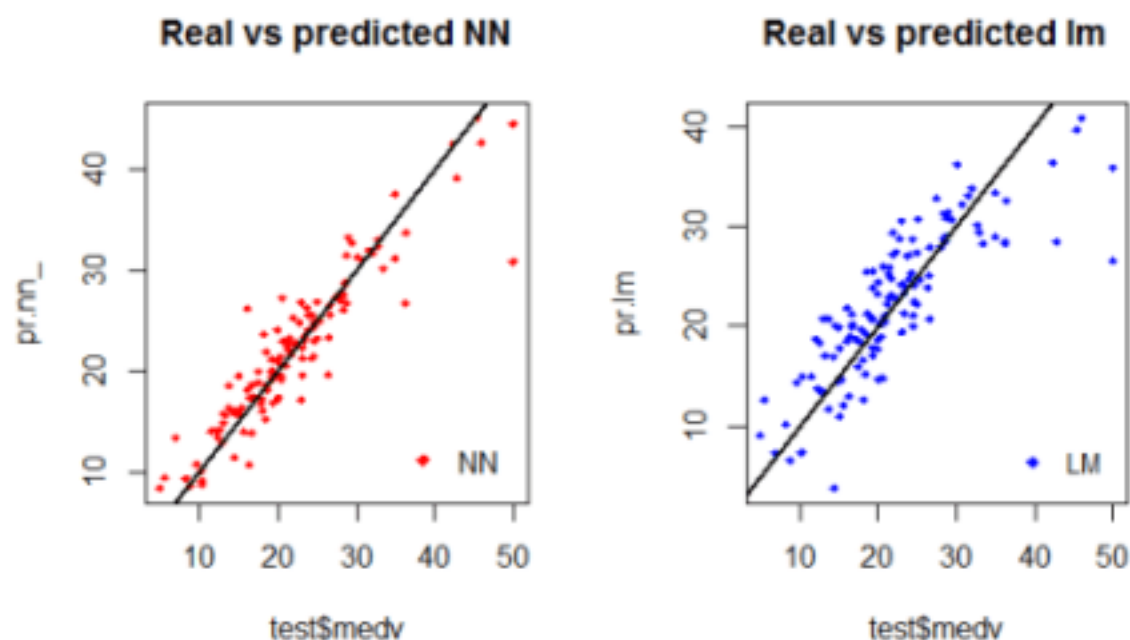
A first visual approach to the performance of the network and the linear model on the test set is plotted below

```
par(mfrow=c(1,2))
```

```
plot(test$medv,pr.nn_,col='red',main='Real vs predicted NN',pch=18,cex=0.7)
abline(0,1,lwd=2)
legend('bottomright',legend='NN',pch=18,col='red', bty='n')
```

```
plot(test$medv,pr.lm,col='blue',main='Real vs predicted lm',pch=18, cex=0.7)
abline(0,1,lwd=2)
legend('bottomright',legend='LM',pch=18,col='blue', bty='n', cex=.95)
```

The output plot



By visually inspecting the plot we can see that the predictions made by the neural network are (in general) more concentrated around the line (a perfect alignment with the line would indicate a MSE of 0 and thus an ideal perfect prediction) than those made by the linear model.

```
plot(test$medv,pr.nn_,col='red',main='Real vs predicted NN',pch=18,cex=0.7)
points(test$medv,pr.lm,col='blue',pch=18,cex=0.7)
abline(0,1,lwd=2)
legend('bottomright',legend=c('NN','LM'),pch=18,col=c('red','blue'))
```

A perhaps more useful visual comparison is plotted below:

A (fast) cross validation

Cross validation is another very important step of building predictive models. While there are different kind of cross validation methods, the basic idea is repeating the following process a number of time:

train-test split

- Do the train-test split
- Fit the model to the train set
- Test the model on the test set
- Calculate the prediction error
- Repeat the process K times

Then by calculating the average error we can get a grasp of how the model is doing.

We are going to implement a fast cross validation using a for loop for the neural network and the `cv.glm()` function in the `boot` package for the linear model.

As far as I know, there is no built-in function in R to perform cross validation on this kind of neural network, if you do know such a function, please let me know in the comments. Here is the 10 fold cross validated MSE for the linear model:

```
library(boot)
set.seed(200)
lm.fit <- glm(medv~.,data=data)
cv.glm(data,lm.fit,K=10)$delta[1]
```

23.83560156

Now the net. Note that I am splitting the data in this way: 90% train set and 10% test set in a random way for 10 times. I am also initializing a progress bar using the `plyr` library because I want to keep an eye on the status of the process since the fitting of the neural network may take a while.

```
set.seed(450)
cv.error <- NULL
k <- 10
```

```
library(plyr)
```



```

pbar <- create_progress_bar('text')
pbar$init(k)

for(i in 1:k){
  index <- sample(1:nrow(data),round(0.9*nrow(data)))
  train.cv <- scaled[index,]
  test.cv <- scaled[-index,]

  nn <- neuralnet(f,data=train.cv,hidden=c(5,2),linear.output=T)

  pr.nn <- compute(nn,test.cv[,1:13])
  pr.nn <- pr.nn$net.result*(max(data$medv)-min(data$medv))+min(data$medv)

  test.cv.r <- (test.cv$medv)*(max(data$medv)-
min(data$medv))+min(data$medv)

  cv.error[i] <- sum((test.cv.r - pr.nn)^2)/nrow(test.cv)

  pbar$step()
}

```

After a while, the process is done, we calculate the average MSE and plot the results as a boxplot

```

mean(cv.error)
10.32697995

cv.error
17.640652805  6.310575067 15.769518577  5.730130820 10.520947119  6.121160840
6.389967211  8.004786424 17.369282494  9.412778105

```

The code for the box plot:

```

boxplot(cv.error,xlab='MSE CV',col='cyan',
        border='blue',names='CV error (MSE)',
        main='CV error (MSE) for NN',horizontal=TRUE)

```

The code above outputs the following boxplot:

As you can see, the average MSE for the neural network (10.33) is lower than the one of the linear model although there seems to be a certain degree of variation in the MSEs of the cross validation. This may depend on the splitting of the data or the random initialization of the weights in the net. By running the simulation different times with different seeds you can get a more precise point estimate for the average MSE.

A final note on model interpretability

Neural networks resemble black boxes a lot: explaining their outcome is much more difficult than explaining the outcome of simpler model such as a linear model. Therefore, depending on the kind of application you need, you might want to take into account this factor too. Furthermore, as you have seen above, extra care is needed to fit a neural network and small changes can lead to different results.

A gist with the full code for this post can be found [here](#).

Thank you for reading this post, leave a comment below if you have any

question.

To **leave a comment** for the author, please follow the link and comment on their blog: [DataScience+](#).

[R-bloggers.com](#) offers **[daily e-mail updates](#)** about [R](#) news and [tutorials](#) on topics such as: [Data science](#), [Big Data](#), [R jobs](#), visualization ([ggplot2](#), [Boxplots](#), [maps](#), [animation](#)), programming ([RStudio](#), [Sweave](#), [LaTeX](#), [SQL](#), [Eclipse](#), [git](#), [hadoop](#), [Web Scraping](#)) statistics ([regression](#), [PCA](#), [time series](#), [trading](#)) and more...