How a monopolist can learn from market reactions using an artificial neural network.

Luigi Battistoni

June 2016

Turin

Abstract

The paper studies how a monopolist can understand the shape of the demand curve of the buyers. Initially he can only sell at random prices in order to collect a significative number of *price quantity* couple. After having formed a database of the order of 100 couples it is shown the difference between two methods: an empirical one, where the monopolist continues to use random prices, and one using Neural Networks, where he uses the datas collected in order to predict how the market will react to a new price. The demand curves obtained are then used to maximize the profit of the monopolist. All this process is repeated for 3 different cases: a never-changing demand curve, an always-changing demand curve and a curve that changes every 10 time ticks. An additional case is then studied: it shows the results of these two methods if the database is poor (10 couples instead of 100). At the end of the paper an appendix is added, where is explained how a neural network works and some more details are given about the package *NeuralNet.*

How a monopolist can learn from market reactions
using an artificial neural network

Introduction

The main objective of this paper is to determine if a monopolist can use neural networks in order to maximize the profit. First of all the world we are working in must be described: the monopolist is by definition the only seller of this world. This means that he has the power of choosing the price, that is unique. The quantity is chosen by buyers, given different prices. So the goal of a monopolist is to determine the best couple "price quantity" . Buyers' actions are governed by their demand curve, meaning that whether they will decide to buy or not a quantity of goods given their price is already written in this curve. The crucial fact is that the monopolist doesn't know the demand curve of the buyers. This means that he can only sell at random prices and try to figure out the demand curve. This is the main problem of the monopolist.

One solution could be to collect enough couple "price quantity" and try to plot the demand curve. This is not the most efficient solution, in fact not only it requires a lot of time, but we made no assumptions on the demand curve itself. This means that it could not be constant in time. With this approach datas collected by monopolist in a very large time are not necessarily meaningful.

A more powerful and time-saving method is to use neural networks. The neural network's job is to determine the relationship between price and quantity. The idea is that monopolist should collect a serie of datas (the couple price quantity), then he tells the neural network that, given in input the list of prices, the output is the list of quantities. What neural network will do is to learn how input and output are linked. Now monopolist can use the network to know what will be the quantity associated with a price. This means that he has built the

demand curve and can maximize the profit using simple computation.

Of course this means that the monopolist only requires a very little list of datas compared to the first method. Then the neural network will give him all the other datas. Moreover this method is accurate even if the demand curve changes. It is obvious that after some time has passed, if the demand curve continues to change, the results will be less accurate. All the monopolist will have to do is to feed the network with new datas, so that it can learn the demand curve once again.

The maximization of the profit is calculated with the usual formula

$$profit = (price - costs) * quantity$$

where costs are determined in the following way:

$$cost = 0.00001 * quantity \wedge (2.2) + 10 * quantity \wedge (-0.5)$$

Computing these two formulas for each "price quantity" couple will give the monopolist a list of profits. The highest value is the one we were looking for and its associated "price quantity" couple is the one the monopolist is going to use.

Software description

The languages used are *NetLogo* and *R*. The procedures that generate monopolist's random prices, the demand curve and the monopolist's costs are imported by the *Terna's program*, called *monopolio004*, whose code is further reported. This code is used in order to generate the 4 files containing the "random-price quantity" datas. The first file contains 100 prices and quantities obtained from a demand curve that changes at every time tick. The second file contains 100 prices and quantities obtained from a demand curve that is generated only at the beginning of the process. So all these datas are taken from the same demand curve. The third one contains 100 prices and quantities and the demand curve changes every 10 time ticks. The last one contains only 10 prices and quantities, obtained from the same demand curve. All these files are saved as .txt documents and are fed into neural network by *main_program*.

```
;-------------------------
globals [x whole_price_list marginal_revenue_list marginal_revenue_list_ma monopolist_p
        expected_q cycle_# monopolist_gain0 monopolist_gain future_price_correction
        future_price_correction0]

breed [consumers consumer]

consumers-own [q price_list]

;-------------------------
to setup

create-consumers consumer_number
ask consumers [fd 10]

set monopolist_p random-float max_p

foreach n-values shop_n [?]
  [ask (patch random-pxcor random-pycor)
    [set pcolor lime] ]

end

;-------------------------

to run_consumers
if (cycle_# mod 10 = 0) [

ask consumers [
  set heading random 360
  fd random 5
  ]

fix_q_in_consumers

build_demand_curve

]

set cycle_# cycle_# + 1
end ;to run_consumers

;-------------------------

to fix_q_in_consumers
```

```
ask consumers [
  set q (random max_q) + 1
  set price_list n-values q
    [random-float max_p]
  set price_list sort-by [?1 > ?2] price_list
  ]

set-current-plot "quantities"
  set-current-plot-pen "n with q" plot-pen-reset
    histogram [q] of consumers
  set-current-plot-pen "q" plot-pen-reset
    foreach n-values consumer_number [?] [
      ; show q-of consumer ?
      plotxy ? [q] of consumer ?
      ]

end ;

;-------------------------

to build_demand_curve

set whole_price_list []
foreach n-values consumer_number [?]
  [set whole_price_list sentence whole_price_list [price_list] of consumer ?
  ]
set whole_price_list sort-by [?1 > ?2] whole_price_list
end

;-----------------------------------

to-report vc
  ifelse no_vc
  [report 0]
  [report 0.00001 * x ^ 3.2 + 10 * x ^ 0.5]
end

;-------------------------

to-report avc
  report vc / x
end

;--------------------------
```

The procedure *setup* creates the consumers and then *fix_q_in_consumers* assigns each

consumer a random quantity in range 1 to *max_q* and a random price associated to that quantity, from 0 to *max_p*. The demand curve is now built by *build_demand_curve:* all prices are put in a list and are sorted from the highest value to the lowest. *Report-vc* and *report-avc* are used in order to calculate costs from quantities. Now two procedures are implemented from the original *Terna's program*:

```
;-----------------------------------

to monopolist_random_price

  set random_price (random-float max_p)
;  show random_price

end

; -----------------

to compare

  monopolist_random_price
  fix_q_in_consumers
  build_demand_curve

  set counter 0
  set compared_list []
  foreach whole_price_list
    [ if ? >= random_price
      [ set compared_list sentence compared_list ?
        set counter (counter + 1)]
    ;  [ set compared_list sentence compared_list 0 ]
    ]
  show compared_list
  show random_price
  show counter
end

;-------------------------------------
```

This code and *monopolio004* are now bounded in the new program *generate_datas* , whose job is to create the .txt files that will be read by *main_program*. Here follows a description of the procedures added by the previous code. *monopolist_random_price* simply generates a random number from 0 to *max_p*. This number is the random price decided by the

monopolist. Now the *compare* procedure shows how many units are bought using that random price. Running these two last procedures 100 times will generate 100 random prices and its bought quantity, being the demand curve not-changing. This is how the second data file has been created. By running together these two procedures and *fix_q_in_consumers* and *build_demand_curve* one can generate the first data file, so 100 quantities and prices obtained from an always changing demand curve. Now the generation of the data files has ended and the main program is asked to elaborate them.

The main program, named *main_program*, uses both *NetLogo* and *R*, connected by the package *Rserve*. *R* is used to compute the graphic part and the neural network, by the package *NeuralNet*. First of all *R* must be launched, then one has to input in command shell *library(Rserve)* and *Rserve()*. In this way *Rserve* will be running.

IMPORTANT: before using *main_program* one must ask *R* the directory where it is working in. This is done simply writing *getwd()* in *R* shell. The usual working directory is *C:/User/Documents* but it might vary, so it is important to understand where it is in order to let the program work. After having found the working directory the folder *MyProgram*, that contains all the .txt files, must be placed in that directory. In this way *Rserve* will be able to read the files. If one does not want to do this process all strings in *main_program* containing final directories must be changed with the wanted ones.

The code of *main_program* is here reported.

```
;-----------------------------------

extensions [rserve]

to show_cost

  let avc 0
  let d 0
  foreach n-values 300 [?]
  [ set d (1 + ?)
    set avc (0.00001 * d ^ 2.2 + 10 * 1 / (d ^ 0.5))
```

```
   show list (d) (avc)
   ]

end

;----------------- Rserve part ---------------

to init
  rserve:init 6311 "localhost"
  print rserve:isConnected
end

;---------------------------------------------

to read_dati1
  rserve:eval "x<-read.table(file='MyProgram/dati1.txt')"
end

;---------------------------------------------

to read_dati2
  rserve:eval "x<-read.table(file='MyProgram/dati2.txt')"
end

;---------------------------------------------

to read_dati3
  rserve:eval "x<-read.table(file='MyProgram/dati3.txt')"
end

;---------------------------------------------

to read_dati4
  rserve:eval "x<-read.table(file='MyProgram/dati4.txt')"
end

;---------------------------------------------

to start_neuralnet
  rserve:eval "library(neuralnet)"
end

;---------------------------------------------

to R
```

```
  rserve:eval "ti<-as.data.frame(x[,1])"
  rserve:eval "to<-as.data.frame(x[,2])"
  rserve:eval "minto<-min(to)"
  rserve:eval "maxto<-max(to)"
  rserve:eval "to_scaled<-(to-minto)/(maxto-minto)"
  rserve:eval "td<-cbind(ti,to_scaled)"
  rserve:eval "colnames(td)<-c('price','quantity')"
  rserve:eval "net.qq<-neuralnet(quantity~price,td,hidden=2,threshold=0.01,rep=15)"
  rserve:eval "net.qq"
  rserve:eval "to_test<-compute(net.qq, ti)"
  rserve:eval "to_testOriginalScale<-to_test$net.result*(maxto-minto)+minto"
  show rserve:get "to_testOriginalScale"
  show rserve:get "to"

end

;-----------------------------------------------

to export
  rserve:eval "ex<-cbind(ti,to,to_testOriginalScale)"
  rserve:eval "colnames(ex)<-c('prezzo','quantita','quantita calcolata')"
  rserve:eval "write.table(ex,file='MyProgram/mydata.txt')"
  show "export done"
end

;-----------------------------------------------

to cost

  rserve:eval "costi<-read.table(file='MyProgram/costi.txt')"
  rserve:eval "cost_theory<-( 0.00001 * x[,2] ^ (2.2) + 10 * x[,2] ^ (-0.5))"
 ; show rserve:get "cost_theory"
  rserve:eval "cost_net<-( 0.00001 * to_testOriginalScale  ^ (2.2) + 10 *
to_testOriginalScale  ^ (-0.5) )"
 ; show rserve:get "cost_net"
  rserve:eval "profit_theory<-(x[,1] - cost_theory) * x[,2]"
  rserve:eval "profit_net<-(x[,1] - cost_net) * to_testOriginalScale"
 ; show rserve:get "profit_theory"
 ; show rserve:get "profit_net"
  rserve:eval "profit<-cbind(ti, to, profit_theory, to_testOriginalScale, profit_net)"
  rserve:eval "colnames(profit)<-c('prezzo','quantita teorica','profitto teorico','quantita
calcolata','profitto calcolato')"
  rserve:eval "write.table(profit,file='MyProgram/profit.txt')"
  show "done"
  rserve:eval "maxpt<-max(profit_theory)"
  rserve:eval "maxpn<-max(profit_net)"
```

```
  show "max profit theory"
  show rserve:get "maxpt"
  show "max profit net"
  show rserve:get "maxpn"

end

;-----------------------------------

to graph
  rserve:eval "split.screen(c(2,2))"
  rserve:eval "screen(1)"
  rserve:eval "plot(profit[,2],profit[,1],xlab='quantity   empirical',ylab='price',
main='quantity empirical - price')"
  rserve:eval "screen(2)"
  rserve:eval "plot(profit[,4],profit[,1],xlab='quantity   network',ylab='price',
main='quantity network - price')"
  rserve:eval "screen(3)"
  rserve:eval "plot(profit[,2],profit[,3],xlab='quantity   empirical',ylab='profit   empirical',
main='quantity empirical - profit empirical')"
  rserve:eval "screen(4)"
  rserve:eval "plot(profit[,4],profit[,5],xlab='quantity   network',ylab='profit   network',
main='quantity network - profit network')"

end

;-------------------------------

to dev-off
  rserve:eval "dev.off()"
end


;-------------------------------
to r-idle ;; a "forever" method
  rserve:eval "Sys.sleep(0.01)"
end

;----------------------------------------
```

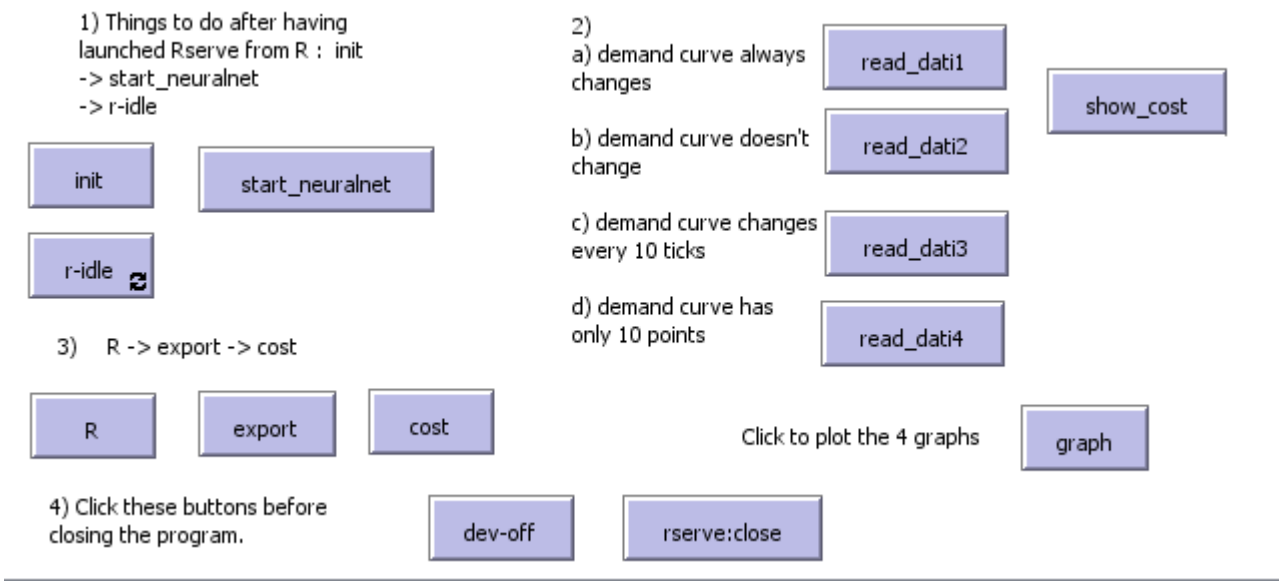

Figure 1
The Shell of *main_program*.

The code is essentially formed by 4 parts:

- The first part connects *NetLogo* with *R* and starts the package *NeuralNet*. This is done with the procedures *init* and *start_neuralnet*. *R-idle* must be running in order to see the plots made by the command *graph*.
- The second part simply asks the user what kind of datas he wants to use: datas obtained by an always changing demand curve or a static one. ".txt" documents created by *generate_datas* are used.
- The third part is the core of the program and where *NeuralNet* is used. The column-matrix of prices, named *ti*, is used as input of the network and the column-matrix of quantites, named *to*, is first scaled and then used as output. Since *NeuralNet* needs input and output files to be in the form of data-frames, the two matrix are built in this form. This means that they are labelled and sticked together in the matrix *td*. Then *NeuralNet* is asked to find the relationship between input and output, and then it shows its results. For a more detailed description of the neural network (and the package *NeuralNet)* see the Appendix. After this the results are saved in a .txt file as a matrix. Finally the procedure *cost* computes every cost and profit associated with every quantity and price. Then it reports the maximum profit and its quantity and price associated.

– The last part is graphic: the *graph* procedure plots the demand curves (the one calculated by the program and the one found by the network) and the profits (calculated by program and by network). Last commands are to shut down *Rserve*.

Results

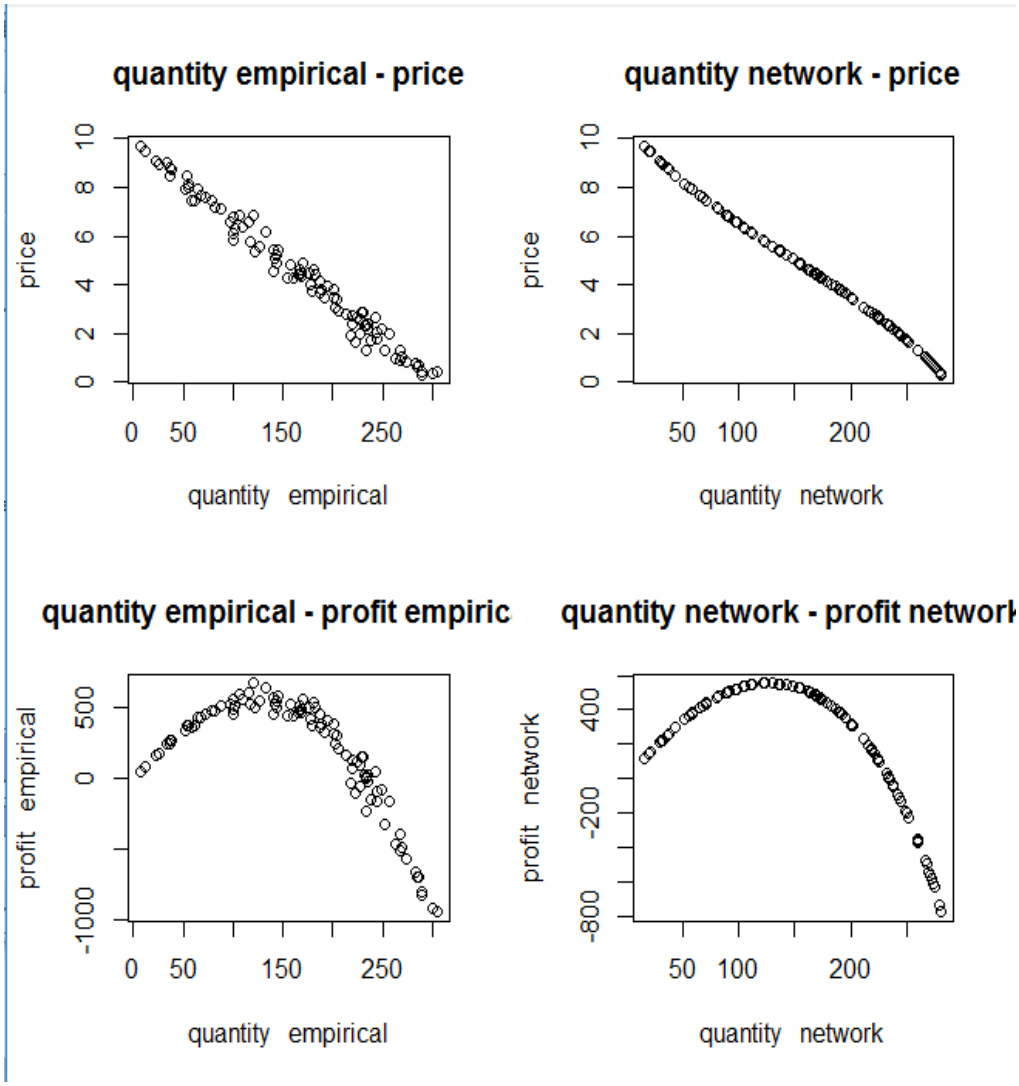

Figure 2
Demand curve always changes.

In **Figure 2** have been used datas obtained from an always changing demand curve.Even if the original datas are spreaded, the neural network has learned the relation and its results are quite linear. While the quantity associated with the maximum profit is circa the same (128 for empirical one and 123 for network's), the value of its profit is lower (673 in the first graph and 564 in the one generated by network).This can be explained taking a closer look

to the first graphic: it's clear that the fact that the demand curve always changes spreads out widely the profit, expecially in the medium range of quantity values (from 50 to 250). This is caused by the costs formula's structure, that is highly influenced by fluctuations of quantity values in this range. Graphically it can be seen that the neural network did a sort of mean value around the maximum value area, that in the first graph is pretty wide and with a large error associated.

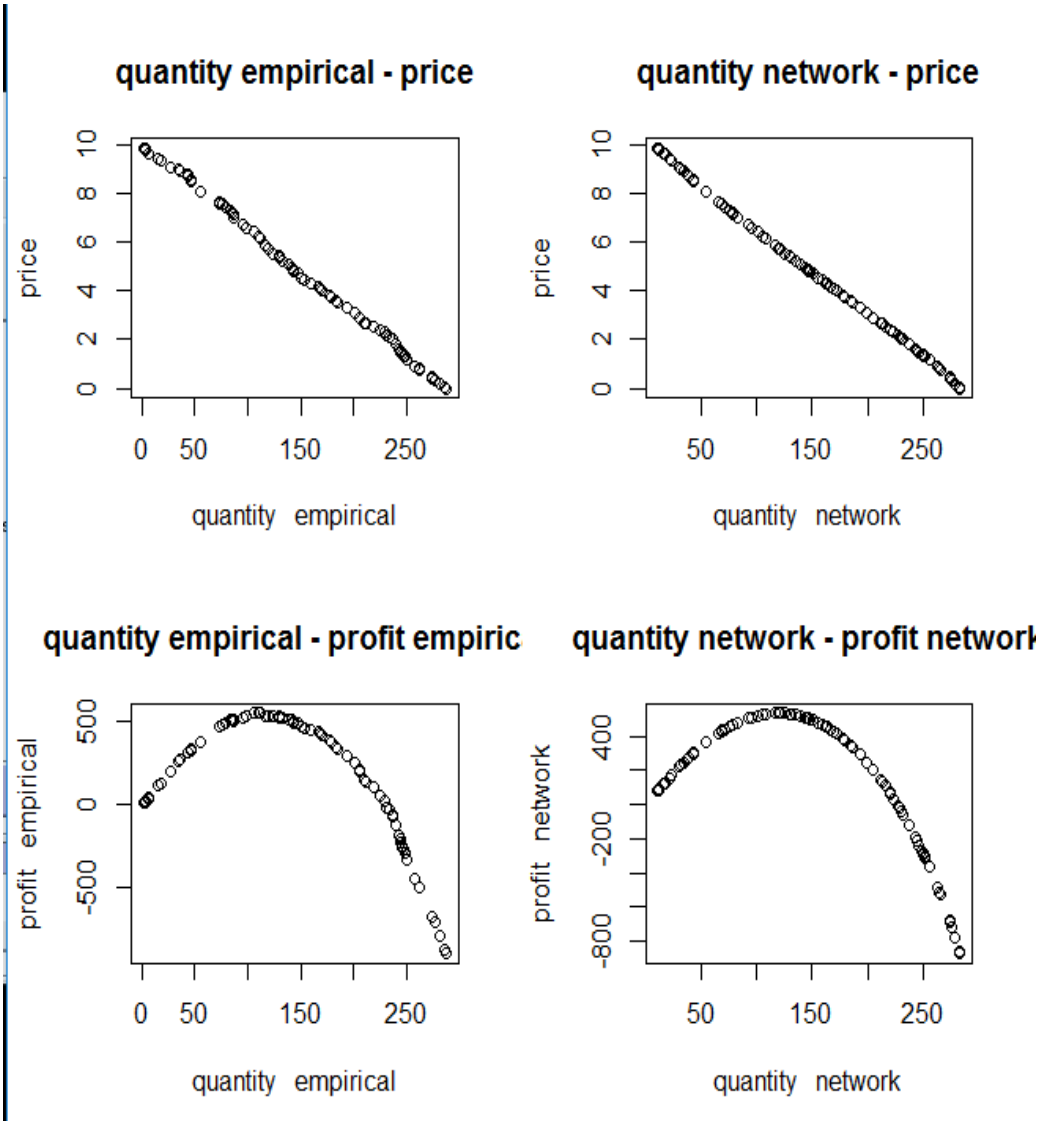

Figure 3
Demand curve never changes.

In **Figure 3** the demand curve never changes. This case is less complex than the previous one, in fact the curve generated by the program is more regular and the neural network is able to plot a precise curve. Maximum profits obtained by the two methods are closer than

the first case: the program found a max profit of 549 and the network of 533, so they differ of a 3% instead of a 17%. The quantity is 117 for empirical method and 118 for neural network. It is clear that the results of the static demand curve and the changing one can not be drastically different, in fact the parameters of the problem are the same: the number of buyers, the maximum price and the maximum quantity. Is however interesting to understand how the network can anticipate the results of the monopolist and build a better demand curve and profit curve.

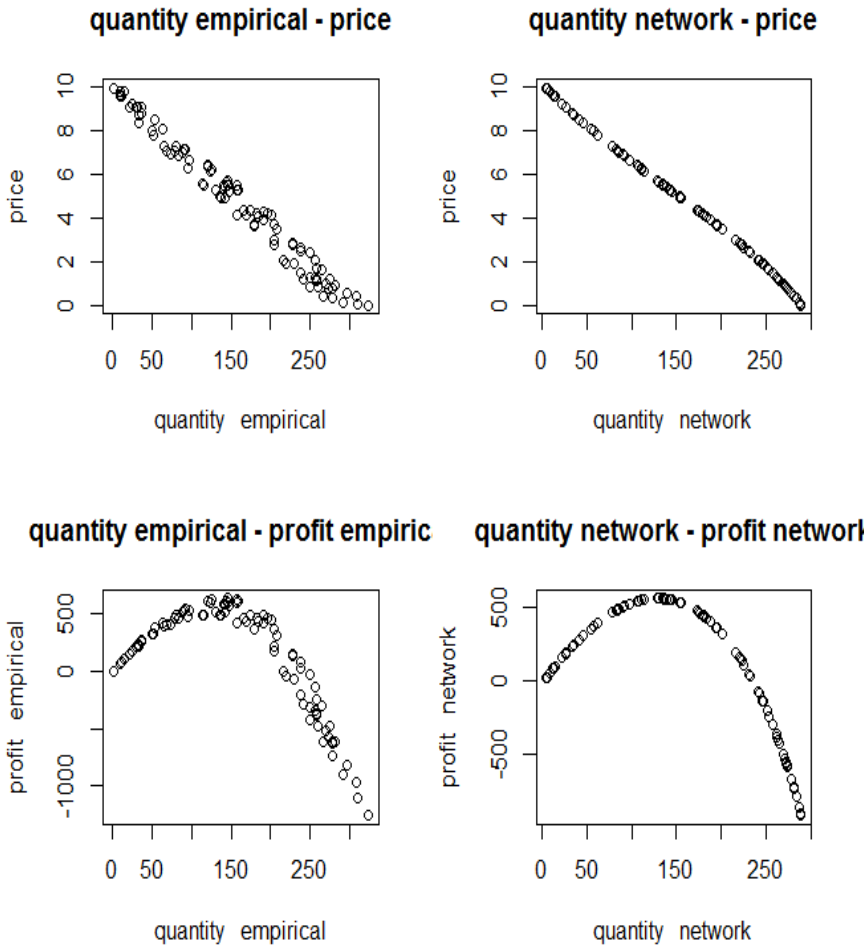

Figure 4
Demand curve changes every 10 time ticks.

In **Figure 4** the demand curve changes every 10 time ticks. As one would expect it is a mixture of the previous two cases: the demand curve and the profit curve are spreaded, but less than the first case. No more interesting results are added by this case. Quantity for empirical method is 148 associated to a maximum profit of 637, while the quantity for

neural network's method is 139 associated to a maximum profit of 543.

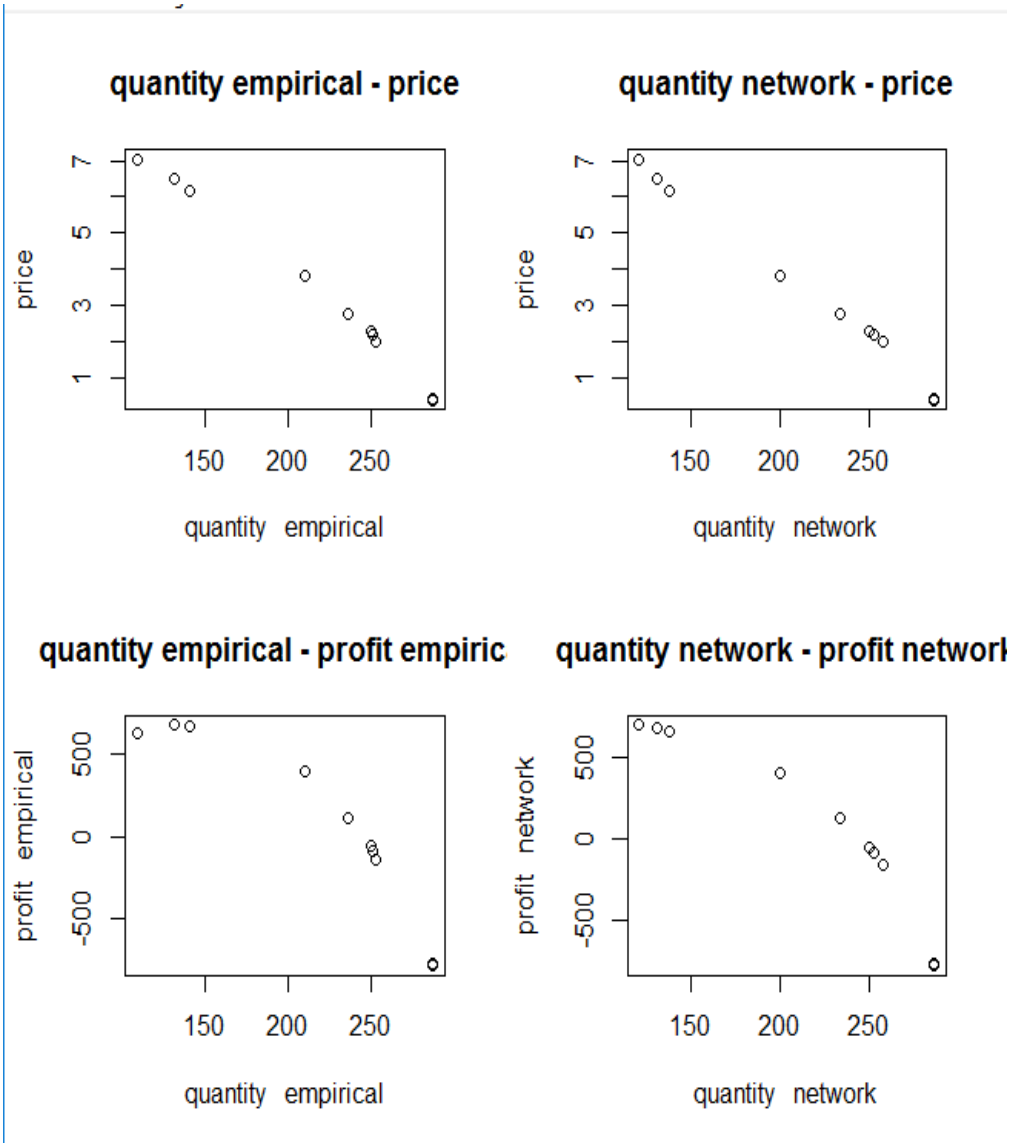

Figure 5
Demand curve never changes, but there are only 10 datas.

In **Figure 5** a limit case is preseted: only 10 datas have been recorded, with a static demand curve. Here the maximum profits are 675 and 668 for the calculated one and the network one respectively. As one could imagine 10 datas are a really poor database and the results are imprecise. Even if the quantity associated with the max profit is still on 130, as in the other cases, the profit value is way bigger than the second case, precisely of 21%. This error is greater than the always changing demand curve. We can point out that if the curve is static 100 cases are enough. The more cases will be added the more the "first method", the empirical one, will be precise. The neural network would be more precise too, but the effort

of collecting more cases is not rewarded in precision, so it would be pretty useless. In case of a changing demand curve, so the more realistic and complex one, the empirical method is always worst than the network one. This is because collecting infinite datas from a curve that changes faster than the speed of data collection is pointless. In fact 1000 datas of this type are just the same as 100, as they would spread the same way. Using neural network instead is more time efficient, but it should be remarked that its result is not permanent. This means that after an indefinite amount of time the couple quantity price obtained will not be the best anymore, because of the wide fluctuations of the demand curve. The amount of time is indefinite because the fluctuations are random, but at that time one should collect the 100 previous datas and feed them into the network again. So a new couple quantity price will be obtained, and so on.

APPENDIX: Neural Networks using NeuralNet

An artificial neural network is a model that simulates the activity of the neurons and the synapsis of an human brain. It simulates a brain in the sense that it can actually elaborate informations, learn and adapt to different situations and it has a tolerance towards errors. Various images of networks used in *main_program* will be shown further. The neural network's logic can be explained in few words, even if the actual computation is quite complex: it is given in input neurons the "input datas" and in output neurons the "output datas". What the network will do is computing these datas to find the best relationship between input and output. In order to do this it uses *hidden neurons* contained in *hidden layers*. Number of hidden neurons and layers is determined by the complexity of the relation between input and output. This will be seen in a more detailed way further. When input and output datas are given to neural network it is said to be a *supervised method*. The advantages of using a neural network is that it can describe and solve a problem only using empirical datas and that it doesn't need any rule. The disadvantage is that in order to solve a specific problem it is built a specific neural network.

The way the neural network learns is the following: it modifies the weights of the synapsis in order to reduce the difference (error) between the output value and the output generated

by the network itself. Iterating this process the network can learn. The simplest case to examine is the one of a network with no hidden layers (i.e. *The perceptron*, that is a binary classificator whose activation function is the sign function). Its weight's adjustment formula is here reported

$$w\,(k+1) = w(k) + A * (\,y - y'(k)\,) * x$$

A is the learning rate, this means that for $A = 0$ the weight at k+1 iteration is influenced by the weight at k iteration (this means that it doesn't change). For $A = 1$ the weight is going to be changed. The values of y and $y'(k)$ are respectively the output value and the neural network's output value at k iteration. Their difference is the error, that is multiplied by x, that is the value of the input (training set).

The problem of having no hidden layers is that it can't be found a solution to every situation, in fact a no-hidden-layers network can only solve separable linear problems. This is why the network used to solve the monopolist's problem has an hidden layer. In general with 2 layers it can be solved mostly every problem, but in this case is quite useless the introduction of another layer (and not time efficient). So how does a multi layer neural network learn? It uses the gradient method: the goal is to determine the vector of weights W that minimizes the sum of the quadratic errors, in order to change the weights of the hidden and output neurons. The error function is given by

$$E(w) = ½ * \sum (\,y(i) - y'(i)\,)^2$$

where the sum is over the number of neurons and y and y' are the same objects of the perceptron. This function $E(w)$ is used in order to compute the iteration of the weights:

$$w(k+1) = w(k) - A * (\,\partial E(w) / \partial w(k)\,)$$

The weights are plotted like a point on the error surface, in that point there is a different slope of the surface respect to the axes formed by every single weight. This is the meaning of the partial derivative. Then the weights are changed using the previous formula. Now the network follows direction associated with the most negative gradient. This process continues untill the network reaches a fixed threshold value. In *main_program* 15 repetitions are made in order to avoid that the algorithm stops in a local minimum of the error surface, in fact we are looking for global minimums.

In *main_program* it is used the *R* package *NeuralNet*. This useful package uses the packages *grid* and *MASS* and allows us to build a neural network in order to solve the monopolist's problem. Moreover it is included a graphical part that plots the shape of the network and the values of weights. This will be shown later. The datas that we want to be the input and output of the network must be in form of *data-frames.* This means that they must be labeled and put in a matrix. The *R* command that computes the network is

net.name<-neuralnet(output~input,dataframe,hidden=2,threshold=0.01,rep=15)

The hidden command is used to tell the network how many hidden neurons we want to use, the threshold is used to stop the process when it is reached that value and *rep* is how many repetitions we want the network to do. The the repetition with the lowest error it will be used in order to solve the monopolist's problem.

Here are plotted some of the networks generated by *main_program:*

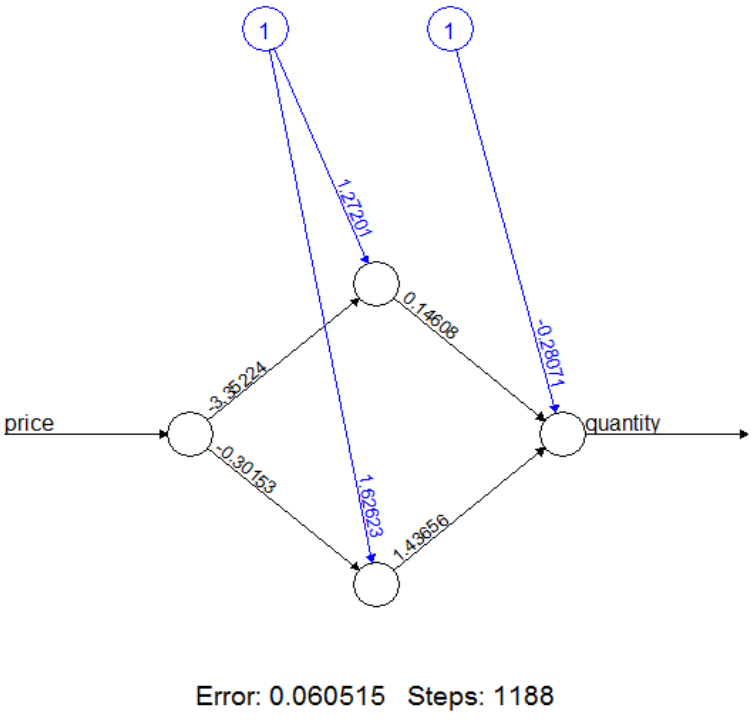

Figure 6
Demand curve always changes (Dati1)

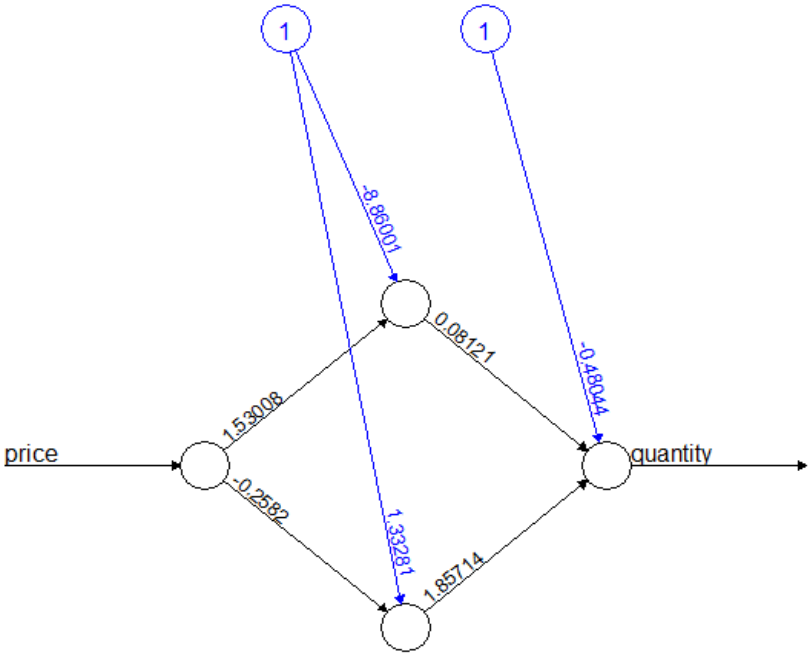

Figure 7
Demand curve never changes (Dati2)

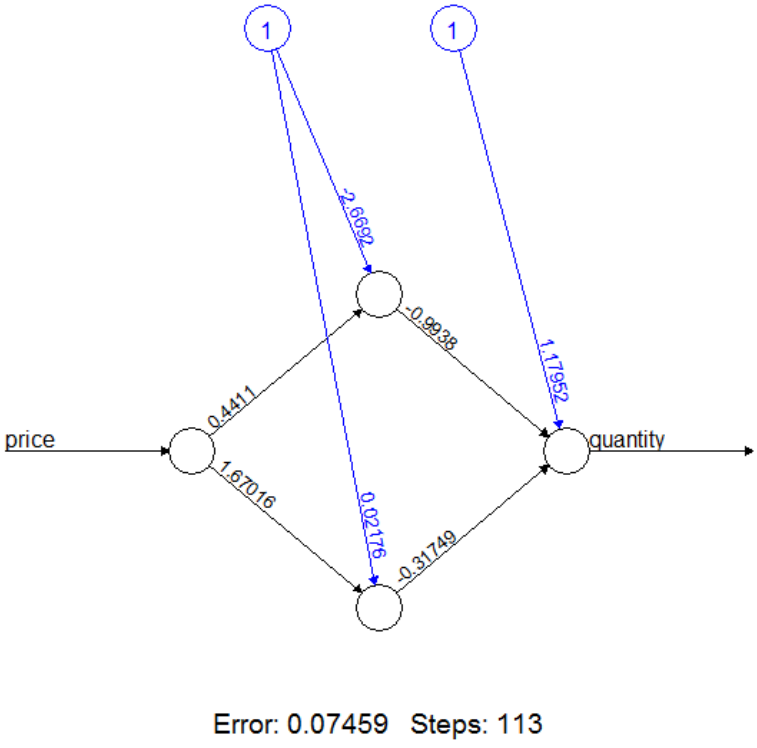

Figure 8
Demand curve changes every 10 time ticks (Dati3)

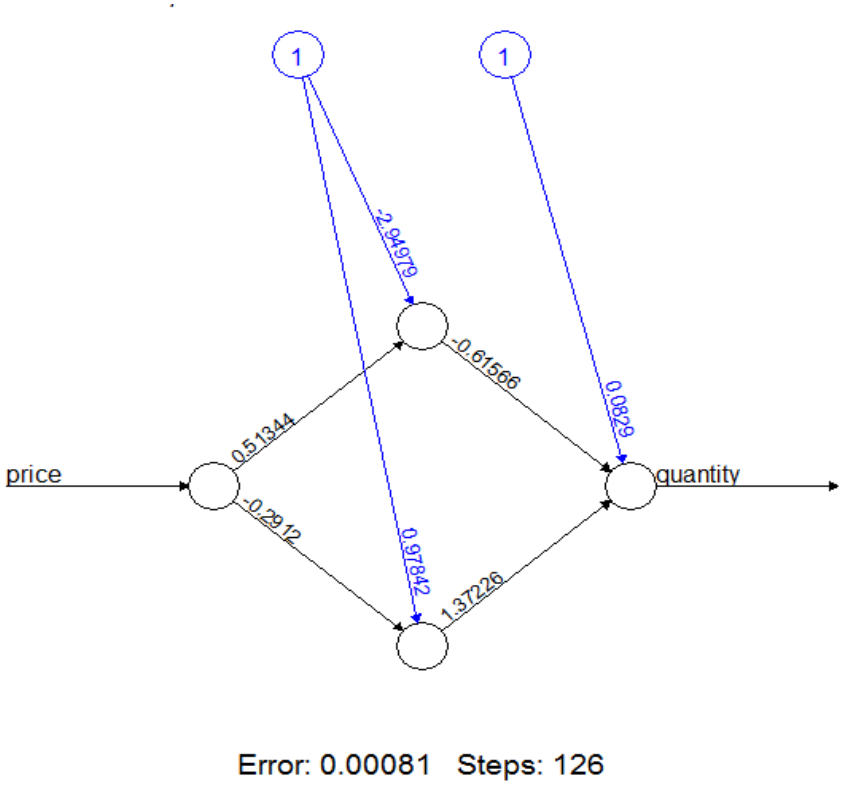

Figure 9
Demand curve never changes, but only 10 datas are available (Dati4)

As it can be seen one layer of two hidden neurons is sufficient in order to solve this problem. In this way the neural network can work at its maximum time efficiency (other hidden layers or neurons would have been useless).

In **Figure 6** and **Figure 8** error is way greater than other two cases (more than 10 times). Let us think about what kind of datas are given to neural network: in **Figure 7** the demand curve is always the same while in **Figure 6** and **8** demand curve changes. Datas of a static demand curve are obviously more regular than datas of a changing one. This means that while the neural network is trying to understand the curve's shape the datas of a static curve will be less spreaded compared to datas of a changing one, resulting in a better approximation and a minor error.

The low error of **Figure 9** instead is probably due to the poor database used by neural network. This means that the results are not more precise than previous cases, but that the datas are so few that a realistic evaluation of error and demand curve is not possible.