Remi HARDY,

DSTI S18, ANN Exam

Goal: evaluate several methods and Neural Network related functions, to approximate the Time-series AirPassenger comprising 144 records.

RESULTS SUM-UP:

Method	Model	Neural Net Cost	MAPE Score	Fitting	Comments
Linear Regression	Linear	NA	NA	Poor	Gives only a trend
ARIMA	MA(1)	NA	NA	Very good	Only simulated a prediction on 12 periods
RBFN	1 layer K=44	44 bases	6.4%	Good	Test 100%
MLP	3 hidden layers	(1,1,1) = 3 neurons	1.75%	Excellent	Training 100% Test 100%
MLP	3 hidden layers	(2,1,1) = 4 neurons	2.5%	Excellent	Training 70% Test 30%
MLP	3 hidden layers	(14,16,19) = 49 neurons	2%	Excellent	Training 70% Test 30% Higher cost
GRNN	Sigma = 0.1	144 activation + 2 summation	2.15%	Excellent	Test 100%
LSTM					Not covered

1. **Linear Regression:**

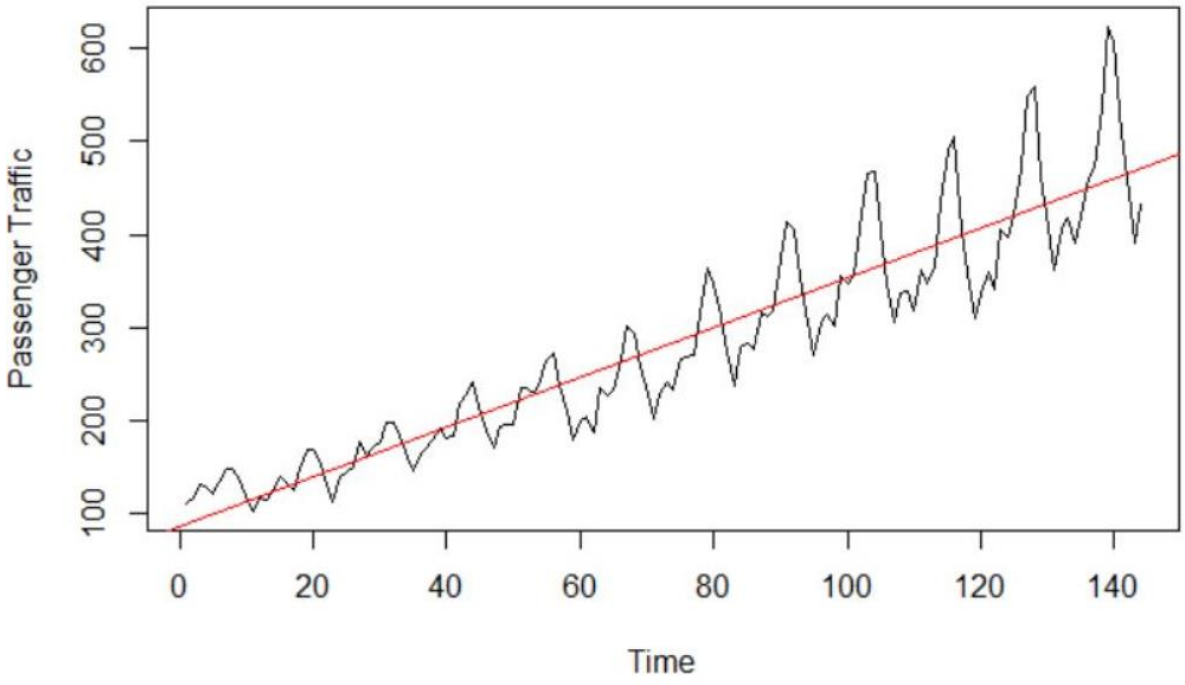
Files: the code is available in *linear_regression.R* (in R)

2 methods:

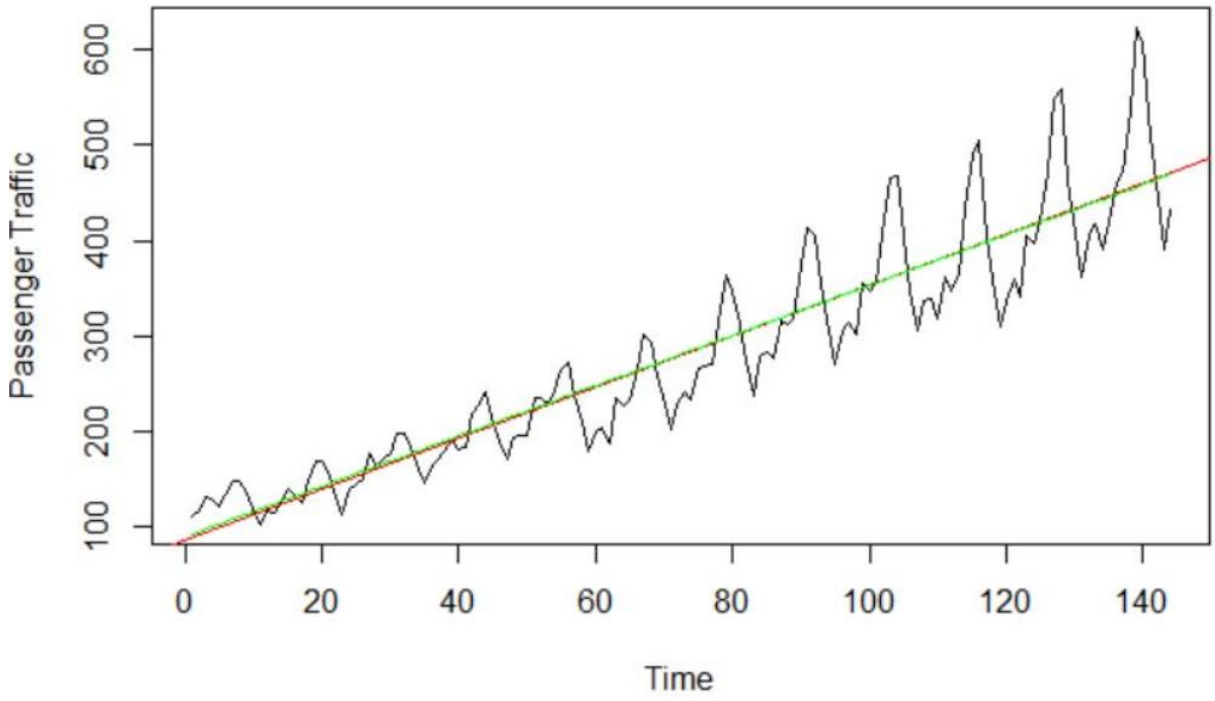
- Using *lm()* function from R
- My own iterative method

Legend:
Lm() function in RED

Results and Plot:



Legend:
Own linear regression function in GREEN (superposed)

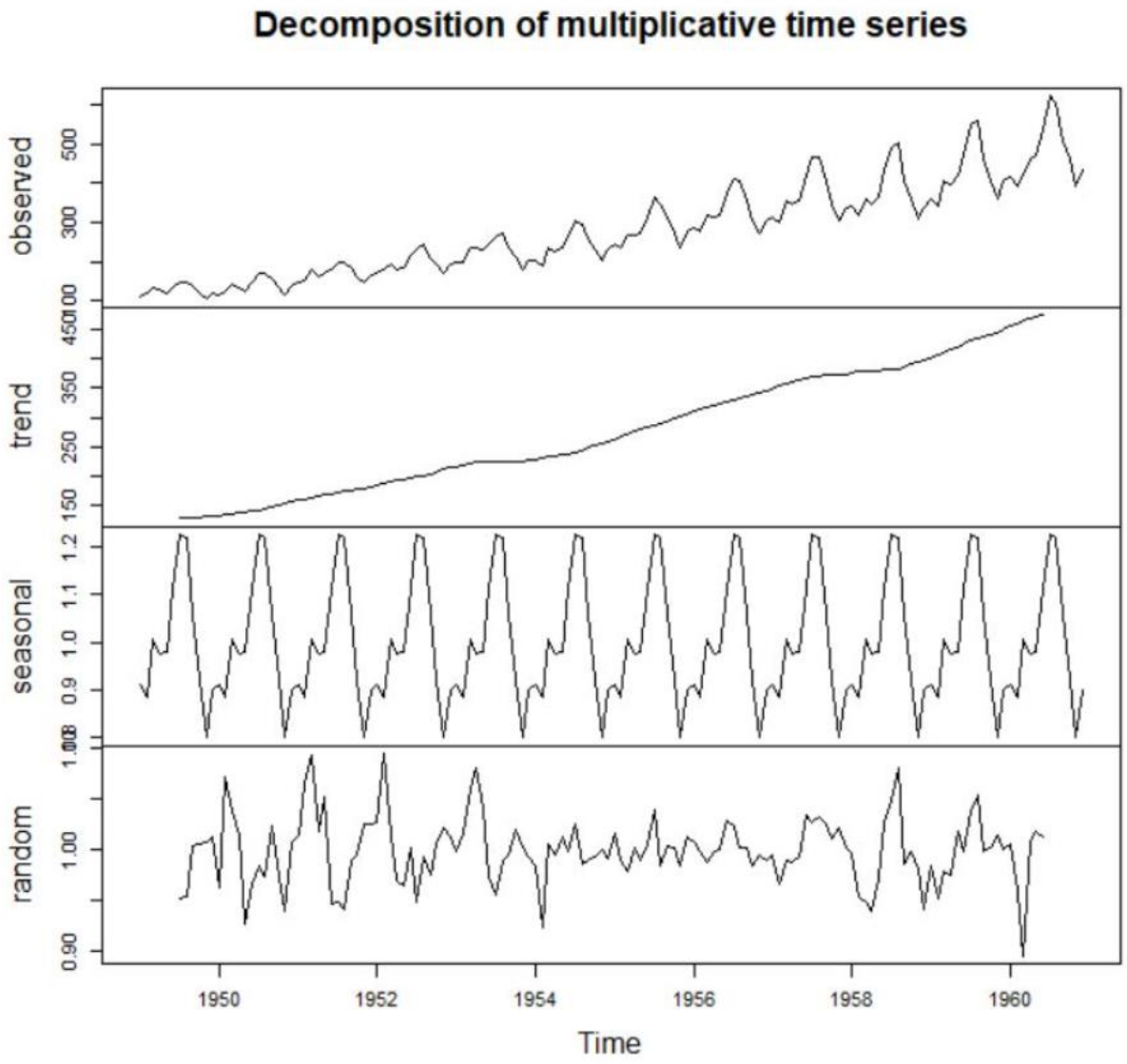


The 2 methods give similar results.
The approximation of the function is not good as one could expect it.
It only gives a trend.

2. ARIMA

Files: the code is available in *arima.R* (in R)

Obviously, the time-series is not stationary as a trend and seasonality can be clearly seen.
This is confirmed by Dickey Fuller test:
With
H0: the time-series is not stationary
H1: the time-series is stationary
Using the *adf.test* function, a p-value=0.7807 is obtained, that leads to conclude that we cannot reject H0 (ie there is a good chance that the time-series is not stationary)
decompose() function in R can be used to extract trend, seasonal components, and residuals (random).

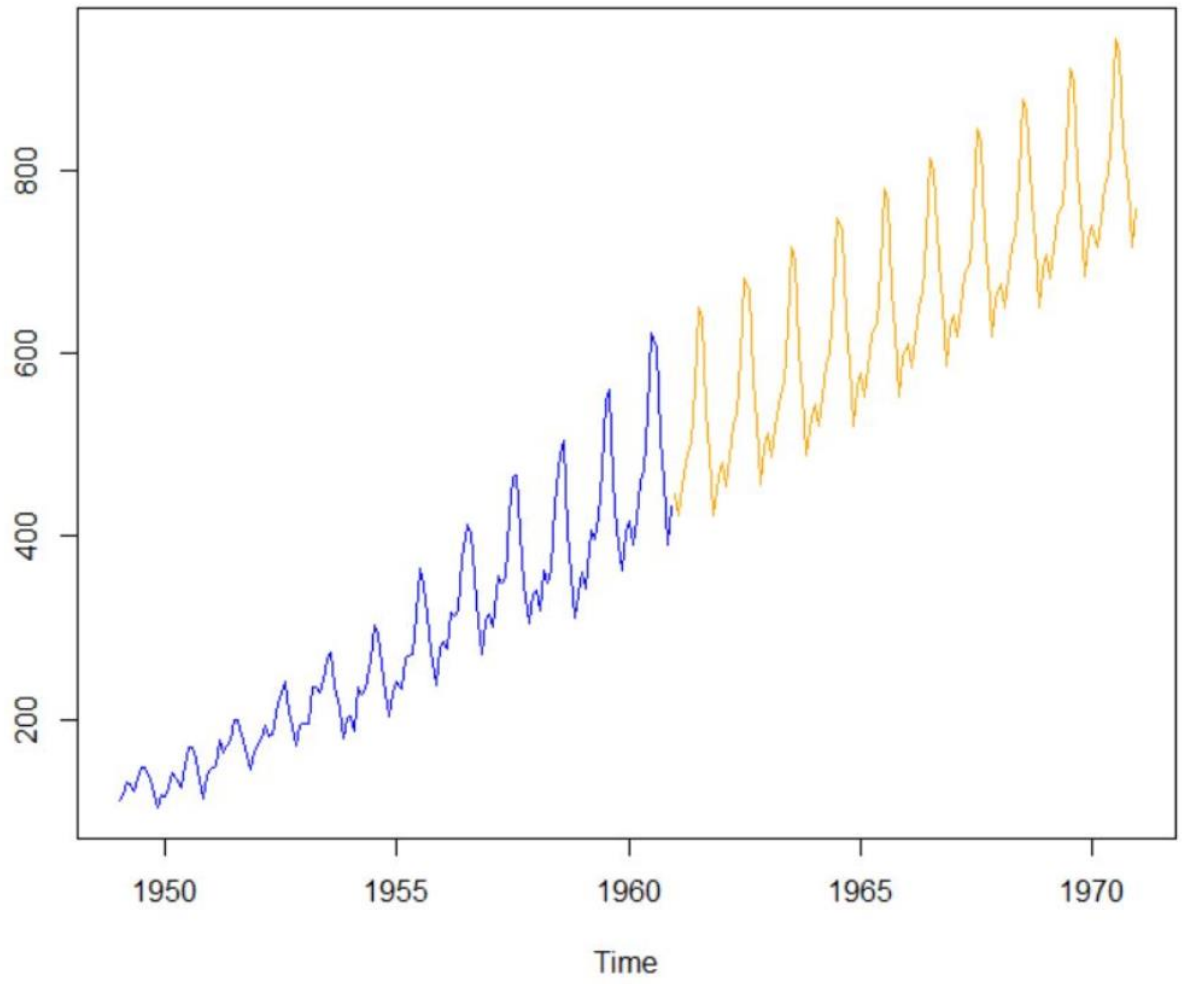


The ARIMA model is defined by analyzing ACF (auto correlation), PACF (partial autocorrelation) and normality of residuals.
To remove the trend, differentiation is needed, with lag=1 that is enough.

An MA1 (moving average type) ARIMA model with parameters (p=0 ; d=1 ; p=1) can be identified.
Then, this model can be sued to do some prediction on the next 12 (for ex) periods.

Legend:
The initial data is in BLUE
The prediction is in ORANGE

Results and Plot:



The prediction seems quite good (although there is no data to confirm it).
The multiplicative effect **may not** be modelled properly as the amplitude of seasonality remains constant over the predicted values.

3. RBFN Radial Basis Function Neural Network

Files: the code is available in *RBFnet.py* (in Python)

The code and the documentation were taken from the blog below:
https://pythonmachinelearning.pro/using-neural-networks-for-regression-radial-basis-function-networks/

The code was adapted to:

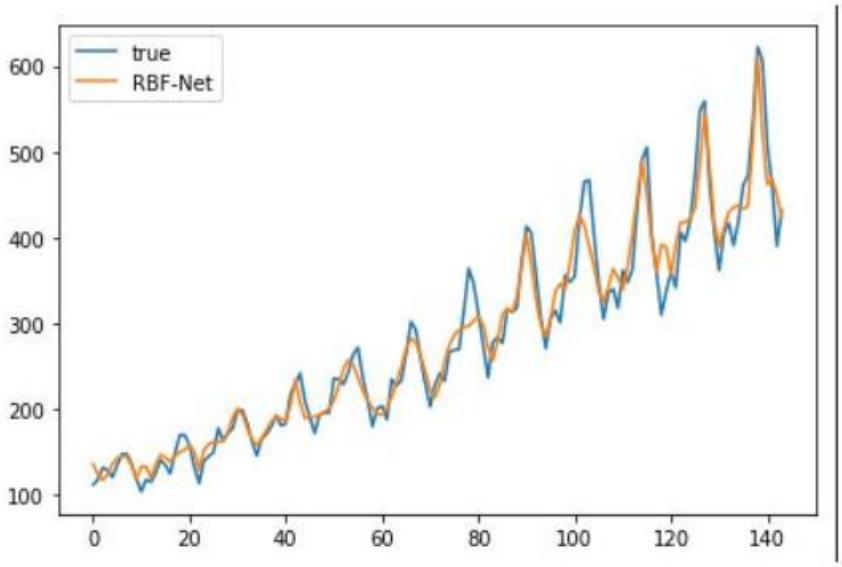
- our dataset
- calculate some scores to evaluate the approximation quality: SSE, MSE, MAPE
- test a range of kernels k (ie number of gaussians to use to approximate the function) and keep the best one according to the scores

Legend:
The initial data is in BLUE
The prediction is in ORANGE

Results and Plot:

Out of a range of k in (40 ..70), I get the best result with k=44 and a MAPE score of 6.4%.

```
k=40 ; sse=123330.295399 ; mse=856.460384715 ; mape=0.0687707732804
k=44 ; sse=88816.8058974 ; mse=616.783374287 ; mape=0.0647912759028
```



The data set is approximated by a smooth function. The method gives interesting results but not as accurate as what is obtained from the MLP other methods (next sections).

4. Using MLPRegressor from SciKit-Learn

The fit function requires arrays of shape (n_samples, n_features).
As our dataset is a time series (144 data points), it was not very clear initially if I had to feed the fit function with (1 sample, 144 features) or (144 samples, 1 feature).
So, I tested both.

Eventually, the following blogs gave also some hints:
http://www.machinelearningtutorial.net/2017/01/28/python-scikit-simple-function-approximation/
https://machinelearningmastery.com/time-series-forecasting-supervised-learning/

A. 1 sample, 144 features:

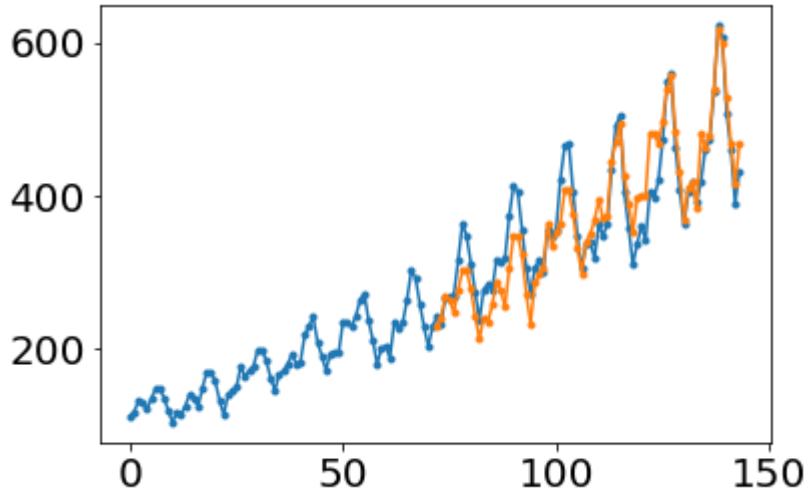
Files: the code is available in *mlp_1_sample.py* (in Python)

Building the data structures for MLPRegressor with this method has the following limitations:

- The number of features must be the same always
- This leads to train on 72 features and test on 72 features
- Therefore, we cannot train on 100 features (70%) and test on 44 features (30%)
- The fitting is not excellent
-

Legend:
The initial data is in BLUE
The prediction is in ORANGE

Results and Plot:



Another script (not provided) allowed to test several configurations.
Eventually 3 layers (2,2,7) gave the best result, with a MAPE score of 7.8%
Even if the fitting is not excellent, the complexity of the neural net is reduced.

B. 144 samples, 1 feature:

Although the method above seems to give some decent result, some reading (blogs) made me doubt whether it is the proper one. So, I reconsidered and changed the way the organization of the data and the training of the network were done.
I also ran 2 approximations:

- over the whole dataset for function approximation (train on 100%, test on 100%)
- with different train data set and test data set (train on 70%, test on 30%)

Preliminary:
The information below is valid for the whole section.

As performance indicator I will use the MAPE score Mean Absolute Percentage Error defined as follows and implemented in the code as a separate function.

$$\mathrm{M} = \frac{1}{n}\sum_{t=1}^{n}\left|\frac{A_t - F_t}{A_t}\right|,$$

where A_t is the actual value and F_t is the forecast value.

The MLPRegressor was tuned by empirical method by comparing a number of runs over several neural net configs. I choose a defined configuration giving consistently the best results based on 3 scores (SSE, MSE, MAPE). This process is not presented in this document.

Eventually, I choose:

- 3 hidden layers giving a better efficiency: I could get lower MAPE score with a reduced number of neurons thus a reduced complexity.
  This might appear as LoveFactor (LF) in the simulation snapshots,
  with LF=Nneurons * MAPE score.
- Activation function: RELU
- Solver: LBFGS
- Iteration: 1000 to limit the run time

The same configuration was consistently run over all the simulations, with no further parameter change.

Function approximation with training 100% / test 100% of the data set

Files: the code is available in *mlp_approx_all.py* (in Python)

RUN(1,10,10,10) function is used as follows: 1 is an internal parameter and should not be changed. The next 3 parameters (10,10,10) are the max number of neurons for the 3 hidden layers, that the program will iterate on, to find the best combination (layer 1 will be iterated from 1 to 10, layer 2 from 1 to 10, layer 3 from 1 to 10, 10x10x10 = 1000 iterations altogether)

As mentioned at the beginning if this section, the data fed to the MLPRegressor function were re-organized in the following way:

Initial data set : 144 values 112,118,132,129 ….. 390, 432 ;
X_train : 112,118,132,129 ….. 390 ;
Y_train : 118,132,129 ….. 390, 432 ;

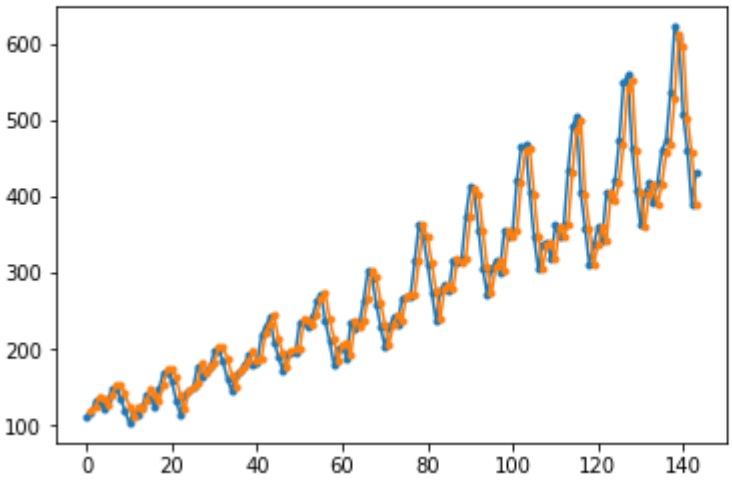
The main idea is: the neural network is trained by the value at time t, to provide the response at time t+1.

To get the approximation result on the whole data set, X_pred is therefore the same vector as X_train.

Legend:
The initial data is in BLUE
The function approximation is in ORANGE

Results and Plot:
With 3 hidden layers comprising 1 neuron each, a pretty good fit and MAPE score of 1.7% are obtained.

```
N neurons=3       layers=1/1/1          mape=0.0175937096557
```



Function approximation with training 70% / test 30% of the data set

Files: the code is available in *mlp_investig_70_30.py* (in Python), and m*lp_L15_best_211_70_30.py* (in Python).
Mlp_investig_70_30.py is to investigate several NN configuration and get the one giving the best results.
Mlp_l15_best_211_70_20.py is the code implementing only the final selected model.

Notice: the MAPE score here cannot be directly compared with the last section (100%/100%) as it is not calculated on the same test part of the approximation (100% for the first, 30% for this one).

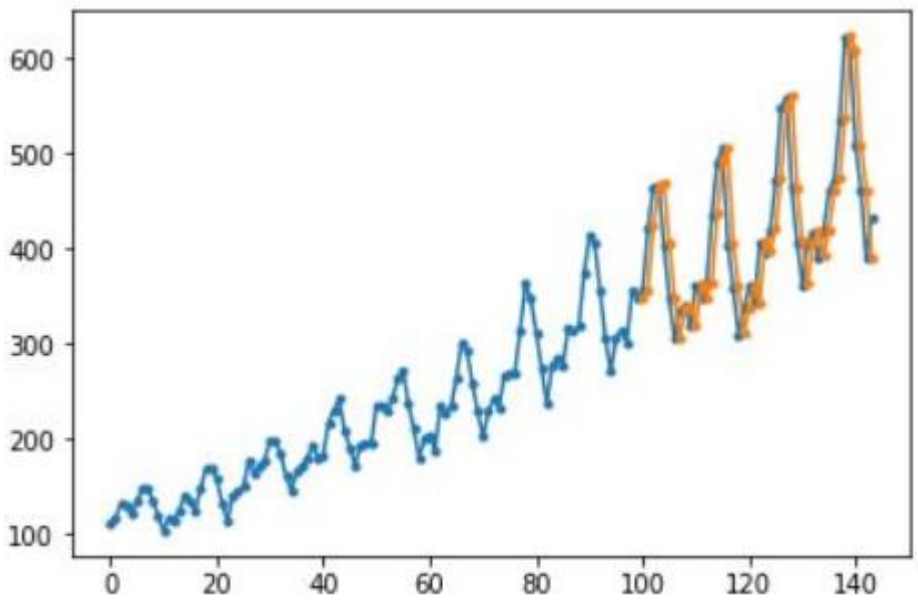
Again, the data fed to the MLPRegressor function were re-organized in the following way, but split in 2 parts for training (first 70%, roughly from 0 to 100 sample index) and test (last 30% roughly from 101 to 144 sample index)

Initial data set: 144 values 112,118,132,129 ….. ;
X_train: 112,118,132,129 ….. ; based on 100 values
Y_train: 118,132,129 ….. ; based on 100 values
X_pred: ; based on 44 values
Y_pred: ; 44 values

Legend:
The initial data is in BLUE
The function approximation is in ORANGE

Results and Plot:

```
Lag=1
N neurons=3     layers=1/1/1        mape=0.462574219104
N neurons=4     layers=1/1/2        mape=0.462574219104
N neurons=5     layers=1/1/3        mape=0.462574219104
N neurons=6     layers=1/1/4        mape=0.462574216923
N neurons=9     layers=1/1/7        mape=0.462574149782
N neurons=12    layers=2/2/8        mape=0.462574101755
N neurons=8     layers=2/4/2        mape=0.0991115936954
N neurons=13    layers=2/4/7        mape=0.0991115526913
N neurons=11    layers=2/5/4        mape=0.0988519113562
N neurons=14    layers=2/9/3        mape=0.0988443566975
N neurons=16    layers=3/7/6        mape=0.0988067647201
N neurons=16    layers=3/8/5        mape=0.098802264622
N neurons=24    layers=7/8/9        mape=0.0987829016363
```



The MAPE score = 9.8% is lower than previously, as one could anticipate it, because the training is done on the first 70% of the data set rather than 100%.
However, please notice, that for the previous simulation on 100% of the data set, the function may have over-fitted the data.

So far, the network was fed with only 1 feature. In order to try to achieve better results, I ran the experiment to feed the network with more features (ie more than 1 input vector).
The idea was to build features based on a sliding window, so that the network can be trained and learn from sequences of the time-series.

X_train is built as follows:
With nlag=1, 1 input vector of the data from samples [0:99] is built
With a nlag=5, 5 input vectors of the data are built

- Vector 1 : samples [0:95]
- Vector 2 : samples [1:96]
- Vector 3 : samples [2:97]
- Vector 4 : samples [3:98]
- Vector 5 : samples [4:99]

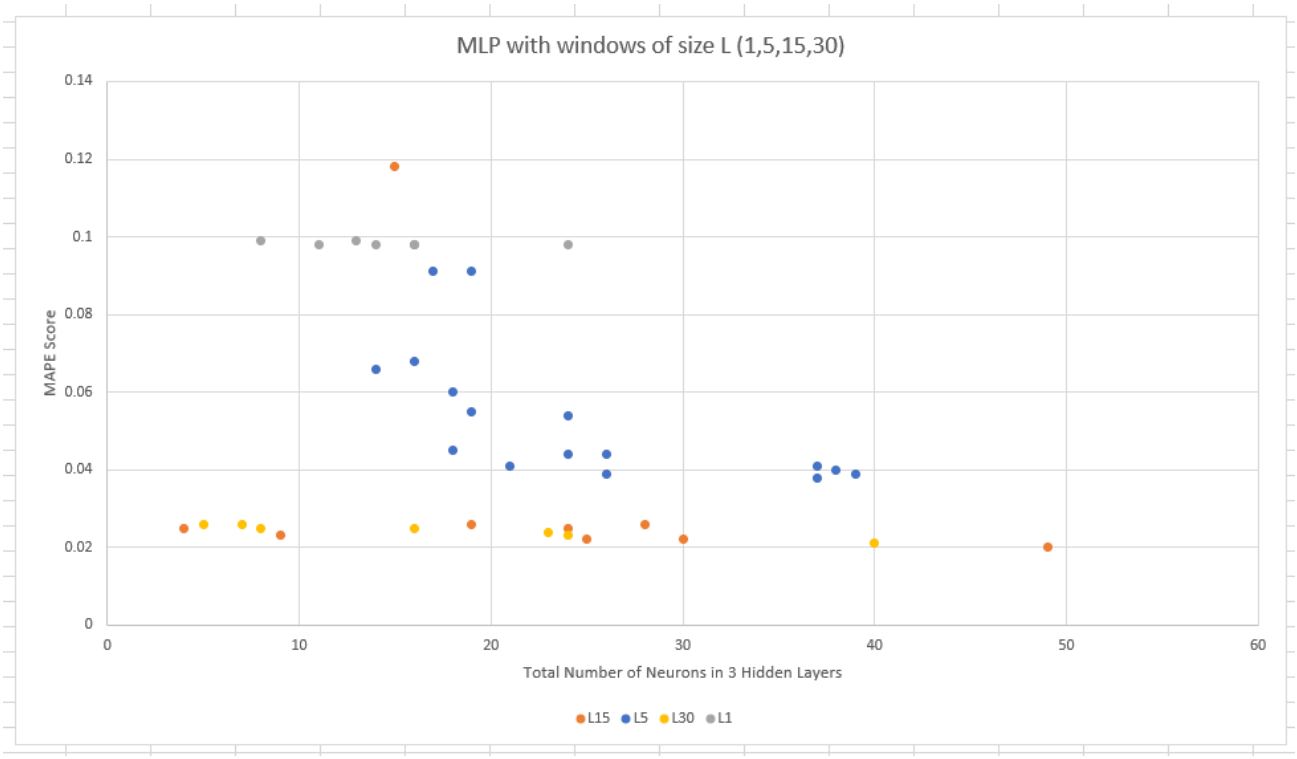
Y_train is shifted accordingly by the lag number

To be compliant with the MLPRegressor function format, X_pred follows the same formatting as for X_train but on the test part of the data set.

The simulations were run:

- for different window sizes: 99 (lag=1), 95 (lag=5) , 85 (lag=15), 70 (lag=30).
- On 3 hidden layers based configuration, each receiving an iteration from 1 to 20 neurons (8000 tested configurations altogether)

The results are summed up in the graph below (and in the excel sheet mlp_analysis_70_30.xlsx):



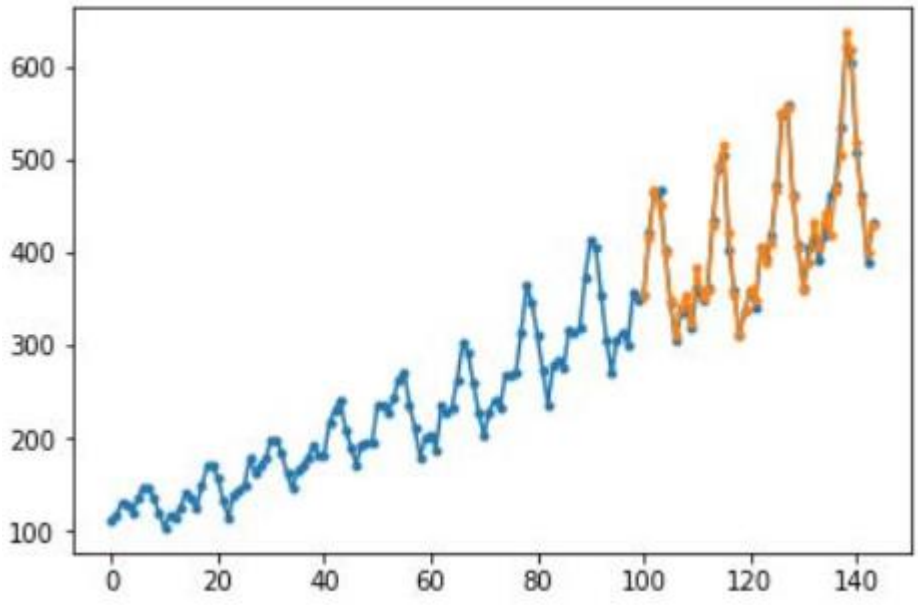
Y axis shows the MAPE score
X axis shows the total number of Neurons in the 3 hidden layers (ie complexity, cost)

We can see that:

- the score gets better with the increase of the lag (lag=15 better than lag=1 or 5)
- increasing the lag beyond 15 does not bring a major increase of the performance
- increasing the number of neurons gives better results
- [bottom left] the best tradeoff performance / complexity is obtained by a (2,1,1) configuration, lag=15, total number of neurons 4, MAPE score = 0.025 (2.5%)
- [bottom right] the best performance is obtained by a configuration comprising 49 neurons, lag15, MAPE score = 0.02 … although we can state that the small increase in performance in done at the cost of a great increase complexity.

Below the results for lag=15 and the best configuration (2,1,1):

```
Lag=15
N neurons=3     layers=1/1/1        mape=0.425633154398     lovefact=1.27689946319
N neurons=5     layers=1/1/3        mape=0.425633076817     lovefact=2.12816538408
N neurons=9     layers=1/1/7        mape=0.425633076188     lovefact=3.83069768569
N neurons=11    layers=1/1/9        mape=0.425633074937     lovefact=4.68196382431
N neurons=12    layers=1/1/10       mape=0.425633072051     lovefact=5.10759686461
N neurons=20    layers=1/2/17       mape=0.425633065346     lovefact=8.51266130691
N neurons=21    layers=1/2/18       mape=0.425633064012     lovefact=8.93829434425
N neurons=13    layers=1/3/9        mape=0.425632957779     lovefact=5.53322845113
N neurons=20    layers=1/3/16       mape=0.425632141494     lovefact=8.51264282988
N neurons=15    layers=1/12/2       mape=0.118278931874     lovefact=1.77418397811
N neurons=19    layers=1/16/2       mape=0.0263706256819    lovefact=0.501041887955
N neurons=28    layers=1/17/10      mape=0.0262735653726    lovefact=0.735659830433
N neurons=4     layers=2/1/1        mape=0.0256420388674    lovefact=0.10256815547
N neurons=24    layers=2/4/18       mape=0.0256409868519    lovefact=0.615383684446
N neurons=9     layers=2/5/2        mape=0.0238802339034    lovefact=0.21492210513
N neurons=30    layers=5/17/8       mape=0.0223530234969    lovefact=0.670590704906
N neurons=25    layers=10/8/7       mape=0.0222174631153    lovefact=0.555436577883
N neurons=49    layers=14/16/19     mape=0.0205196149442    lovefact=1.00546113227
```



4. GRNN General Regression Neural Network

Files: the code is available in *gnrr.py* (in Python)

Sources:
The analysis and code development are based on the following article:
https://easyneuralnetwork.blogspot.com/2013/07/grnn-generalized-regression-neural.html

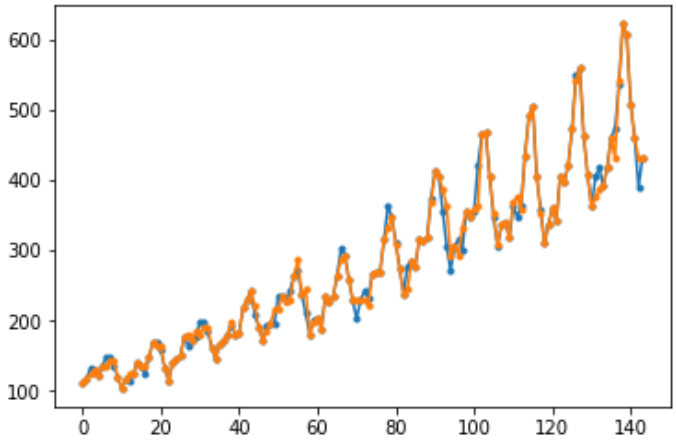
The only unknown parameter is *sigma* that can be tuned to an optimum value where the error is very small.

Eventually, a minimum for sigma = 0.1 is obtained, with a MAPE score of 0.021, reaching the level of our best approximation results (using MLP for example).

Legend:
The initial data is in BLUE
The function approximation is in ORANGE

Results and Plot:

```
sigma=1.0       mape_score=0.0422139642941      sse_score=38526.1302582
sigma=0.1       mape_score=0.0215949608312      sse_score=23172.6666667
sigma=0.01      mape_score=0.0215949608312      sse_score=23172.6666667
sigma=0.001     mape_score=0.0215949608312      sse_score=23172.6666667
sigma=0.0001    mape_score=0.0215949608312      sse_score=23172.6666667
Out[77]: [<matplotlib.lines.Line2D at 0x1ed9eee97f0>]
```



5. LSTM Long Short-Term Memory

There are also other techniques to work out timeseries with neural networks, such as LSTM (long short-term memory) nets that seem to be quite effective and popular.

Please find some blog references below:

Sources:
https://www.altumintelligence.com/articles/a/Time-Series-Prediction-Using-LSTM-Deep-Neural-Networks

or this one, using Keras:

https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/

Unfortunately, I had some trouble to install Tensor Flow and – eventually - was running out of time to investigate further such a method. Therefore, I cannot bring any analysis on this topic within the assignment scope but keep it for future personal work.