A Report on

FASHION RETAIL FORECASTING BY EVOLUTIONARY NEURAL NETWORKS

By:

N. Sai Abhishek Siddhartha (2015A4PS0322H)

& S. Gagan Aditya Reddy (2015B4A40571H)

For the course

Supply Chain Management (ME F421)

Birla Institute of Technology and Science, Pilani

Hyderabad Campus

(April 2019)

Table of Contents

Introduction

Accounting for demand uncertainty is one of the biggest hurdles in supply chain management and retailers have been employ numerous methods to avoid stock out due to demand uncertainty. Primary method for this is calculating and having safety stock. However, having a significant safety stock can also be detrimental as it can shoot up inventory costs and, in the fields, where the products go in and out of trend very quickly, viz., fashion retailing, having significant safety stock may lead to losses if the products are unsold before they are out of trend.

Thus, this is where demand forecasting is used to avoid stock-out and maintain a high inventory fill rate. A few methods used for modelling and predicting the demand are accurate response policy or quick response policy or any of the numerous forecasting models (Holt's model, Winter's model, etc.). Some fashion retail firms also make preliminary forecasts and then revise their forecast in multiple stages as they further acquire market information. By obtaining more information of fashion products (and products closely related or similar to fashion) we can reduce the forecast error which in turn can reduce the inventory holding costs.

Here, we discuss training a multi-layer neural network model with a given data set of past sales records and train it to forecast for the future. However, this is not easy as great care is required when adopting neural forecasting models. The neural network topology can be addressed by a simple trial-and-error method of varying the number of hidden nodes or can also be addressed by considering more elaborate methods like pruning or constructive algorithms. These may still not be accurate as it may be stuck at a local minima and not go over the complete data.

Evolutionary Computation can be picked up as an alternative to this as it does a global multi-point search which can quickly locate areas of high quality despite a complex and large search space. Thus, we employ a combination of these two, Evolutionary Neural Network (ENN) which is a hybrid combination of the two and is currently widely adopted. In general, most ENN approaches use indirect parametric representation and encode each of the factors separately. We, however, consider an ENN approach with direct binary representation to every single neural network connection for time series forecasting. Also, for the ENN approach we take Bayesian Information Criterion (BIC) is used as the fitness function to speed up the searching process.

$$BIC = N\ ln\left(\frac{SSE}{N}\right) + Pln(N)$$

Where, N denotes the number of training cases, P is the number of parameters (in case of linear models) or equal to the number of connection weights (in case of non-linear models) and SSE is the sum of squared errors, given by-

$$SSE = \sum_{i=1}^{L} (Y_i - T_i)^2$$

Where, $(Y_i - T_i)$ is the error term, Y_i is the forecasted term and T_i is the target value.

A pre searching mechanism can also be taken up to further speed up the searching process. We employ a "time lagged feed-forward network" which enables us to create training cases from the data by adopting a sliding time window to perform time series modelling with a multi-layer perception. A sliding time window is defined as the sequence $\langle k_1, k_2, k_3, \dots k_i \rangle$ for a network with i inputs and time lags.

We will be using a basic, fully connected topology, one hidden layer, bias and direct shortcut connections from input to output nodes with sigmoid function as the activation function to account for non-linearity. Hence, the general model can be written as-

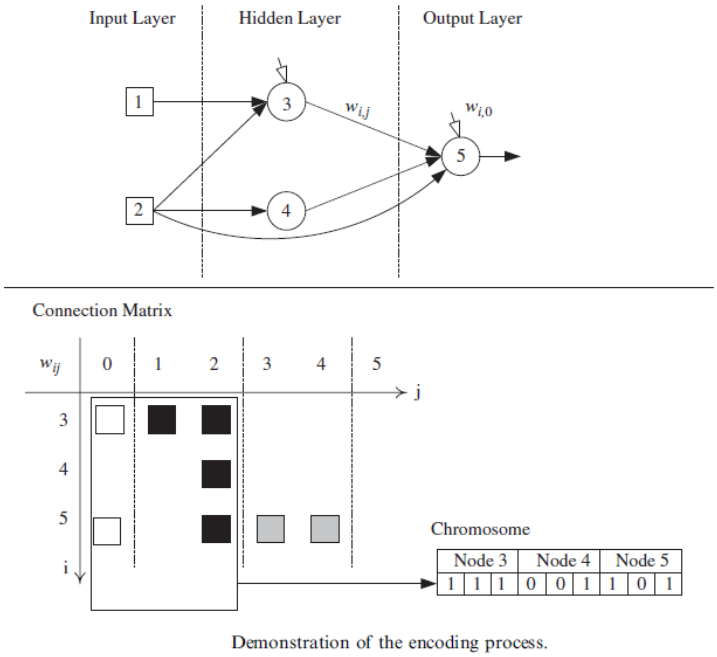
$$\widehat{x_t} = w_{out,0} + \sum_{i=1}^{I} x_t - k_i w_{out,i} + \sum_{j=I+1}^{out-1} f\left(\sum_{i=1}^{I} x_t - k_i w_{i,j} + w_{j,0}\right) w_{out,j}$$

Where, $w_{i,j}$ denotes the weight of the connection from node j to i, "out" denotes output node, f(x) is the sigmoid function, i.e., $\mathrm{f(x)} = \left(1/(1+e^{-x})\right)$, $K_i$denotes the lag of the input i and I in the number of input neurons.

Neural Network Modelling

For optimum forecasting results, selection of a good sliding time window and a neural structure are crucial. We use genetic algorithm in the model selection process as it is suitable in cases where we expect non-linearity.

In genetic algorithm, we have various potential solutions (individuals) to a problem evolving simultaneously (population). Each solution/individual is coded by a string (chromosome) of symbols (genes) taken from a well-defined alphabet. Following this, each solution is assigned a numeric value (fitness) that gives us how appropriate the solution is as a whole. This set of solutions is replaced by off springs, a combination of existing solutions by genetic operators (like crossover and mutation) to give new solutions for which fitness is again calculated. The solution with higher fitness at the end of all the iterations is chosen.



Demonstration of the encoding process.

In general, while designing multilayer network for time-series forecasting, trial and error procedures can be used. However, with evolutionary design we can have ENNs with indirect parametric representation and encode factors (like number of input and hidden nodes, initial weights, activation functions, learning rates, etc.).

From statistical analysis for the time series forecasting with ENN, we know that they are often modelled by small networks. To ensure that the network does not grow too large, fix that a connection exists only if its connection value is 1, as shown in the figure alongside.

The performance criterion, such as SSE is considered for the fitness function. We use this in the validation set to ensure that we do not have a problem of overfitting and is useful when we have limited training data. We choose to use the data from the most recent week to forecast as the week for validation are likely to share the same feature because the sales of fashion are heavily dependent on short trends which spike up. However, this approach is accurate only if we choose the number of hidden nodes (H) appropriately as it may lead to overfitting or may not find a proper small structure.

If the parameter H is large, the approach fails to successfully zero down on a simple network for forecasting and can be very time consuming and has a high chance of overfitting if the value is too less.

Thus, as discussed previously, we employ a pre-search algorithm, which finds an approximate maximum hidden neuron network It decides if the network is overfitting by observing the MSE and gradient in learning is taken. Doing this can greatly speed up the process to end up with a structure of two hidden nodes which does not overfit the data. After this we evaluate evolutionary algorithm as shown below.

BEGIN

Initialize time (t ¼ 0).

Generate the initial neural forecasting population (P_0).

Evaluate the individuals (BIC computation) in P_0.

WHILE NOT ($t<G_{max}$) DO

Select from P_t a number of individuals for reproduction.

Breed the offspring by applying to the genetic operators.

Evaluate the offspring (BIC computation).

Select the offspring to insert into the next population (P_{t+1}).

Select the survivors from P_t to be reinserted into P_{t+1}.

Increase the current time (t: ¼ t+1).

END

Methodology

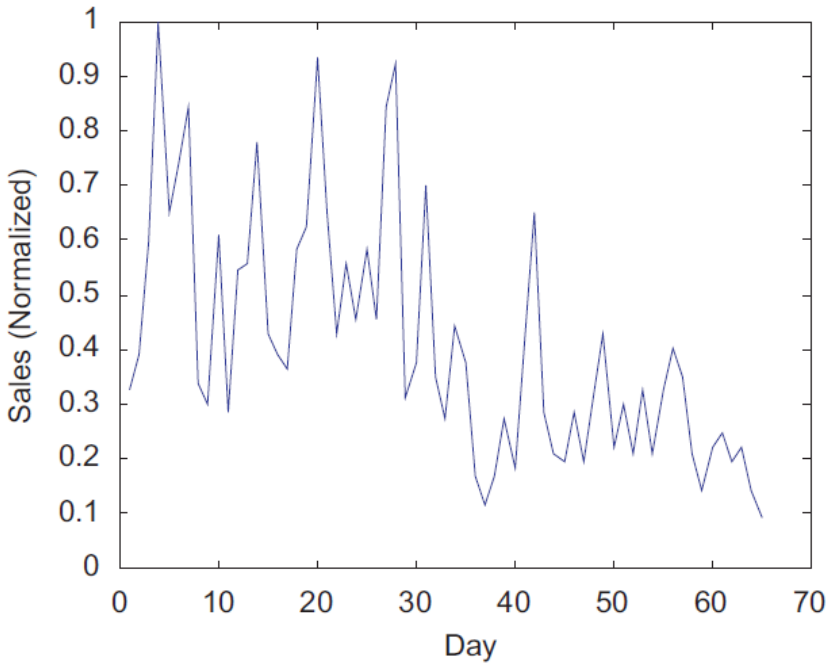
First, we extract the data from the plots given in the Reference [1] and then normalise it in 0-1 scale. Our aim is to replicate the results in the aforementioned paper and compare the said results of using an ENN with an ANN. For implementing the Evolutionary network implemented in the paper, we take the same parameters, i.e., max no of hidden nodes as H=7, reducing it to a simple 2 neuron network with the aid of the pre-search algorithm. After that we test by using a linear activation function to ensure that the data is non-linear. If it is, we continue with sigmoid function as suggested in the paper. We run the code and obtain the results for the same, using BIC fitness condition to get the fit for each of the iteration of chromosomes. Genetic operations are performed to obtain fitter ones from the existing population of chromosomes. The results are documented and compared.

We also construct an ANN to compare the results of the ENN with. We run and test the code for different number of Hidden nodes without using a pre-search algorithm to test how much variation in computation power and time required is. Also, we compare how accurately each forecast and how much time it takes to arrive at the forecast, for possible implementation for forecasting in fashion retail industry.

Results and Inferences

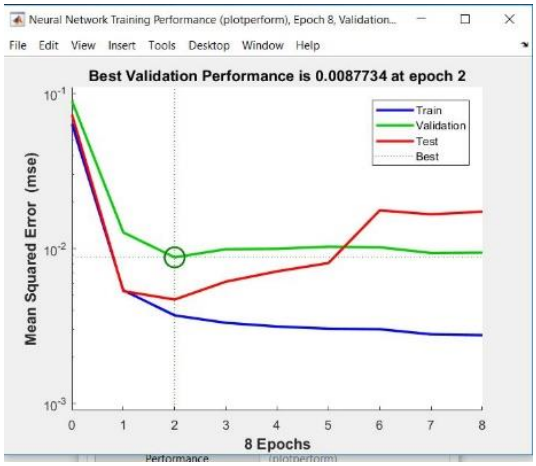
Characteristics of the Network constructed and comparison with the paper

First the data is obtained for reference [1] and then normalised as follows:

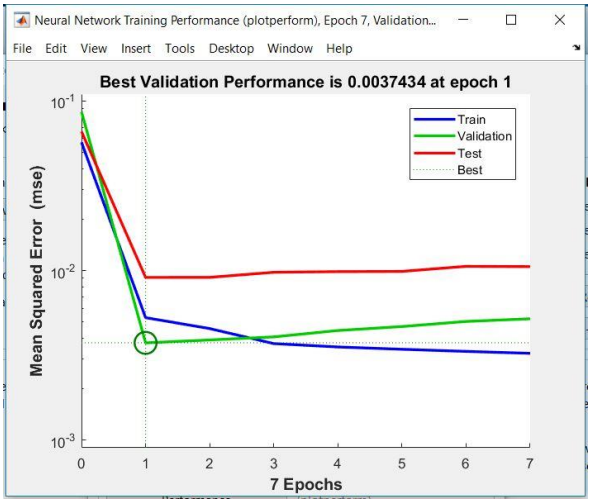


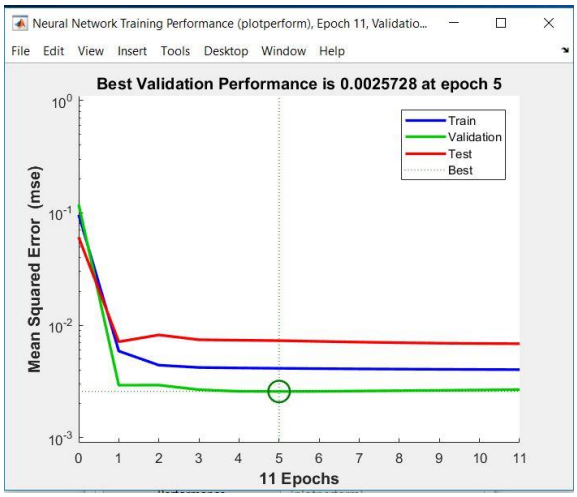
Following this, we test for different activation functions.

a. Linear – as we can see from the result, the MSE for the test value is very high and diverges from the training and validation sets. Thus, we eliminate this and conclude that the data is non-linear. We now test with sigmoid activation functions.



b. Log sigmoid and tan sigmoid – the MSE are clearly lesser and these fit the data way better than non-linear. Tan sigmoid fits the data better than log sigmoid and is chosen and the rest of the ENN is coded. The first figure is log sigmoid and second is tan sigmoid.

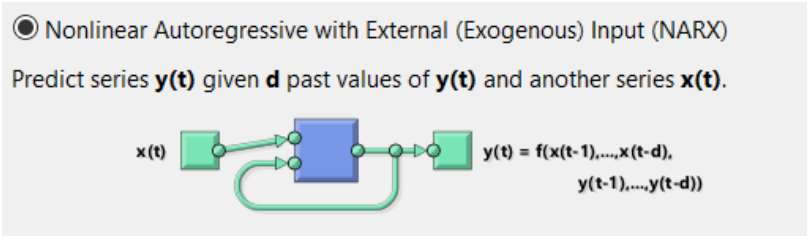


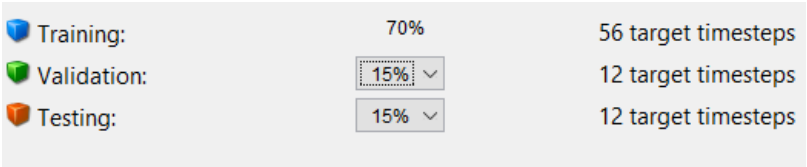


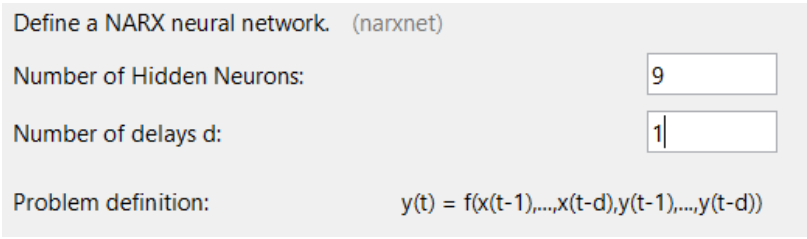
MSE FOR EACH ACTIVATION FUNCTION

0.0025	***Tansigmoid***
0.0037	***Logsigmoid***
0.0087	***Linear***

Other parameters for the ANN are –







We tried for various no of hidden neurons on trial-and-error basis. We finally settled at the above values after testing for various other parameters and comparing if the errors converged to zero.

The fitness for each iteration in ENN is as shown below:

```
p = population(p_count, i_length, i_min, i_max)
fitness_history = [grade(p, target),]
for i in xrange(100):
    p = evolve(p, target)
    fitness_history.append(grade(p, target))
for datum in fitness_history:
    print (datum/1000)
```

```
0.32613
0.20761
0.13798
0.07961
0.02806
0.01519
0.01573
0.01354
0.013
0.013
0.013
0.01348
0.01311
0.01288
0.01027
0.0
0.0
0.0
0.0
0.00305
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.00324
0.0
```

Thus, the ENN we modelled successfully fit the data given. We evaluated this for out of the 1st time window. We obtained ⟨2 3 4 7 8 9 10 13 14⟩ while in the paper, the result obtained was ⟨2 3 4 8 10 11 12 13 14⟩. This might be due to a difference in the initial population size considered for evolution which we weren't able to take a huge one due to limited computational power. Despite this, we did obtain satisfactory results with significant accuracy

Comparison of ENN with traditional SARIMA model

Model	*ENN*	*SARIMA*
MSE for 5th week	0.0329	0.0855
6th Week	0.0712	0.0651
7th Week	0.0177	0.0095
8th Week	0.0023	0.0046

Here, we can clearly see that ENN are more accurate than traditional SARIMA approach. They also consume less time for forecasting overall when we consider the accuracy obtained. Thus, using ENNs are very much desirable for high accuracy forecast.

Conclusions

We have designed and coded an Evolutionary Neural Networks (ENN) for sales forecasting in fashion retailing. The data for the weekly sales is from Reference [1] and we constructed a neural network to forecast for the future weeks of sales with the given data as the training set. Although, the retail sales data, though following a general trend, can be noisy and random. However, we have seen that the ENN approach can be used to obtain a reasonably accurate forecast.

One major drawback of this method is that it requires increasingly more computational power with increasing amount of data we take. We can work around this by employing pre-search approach and using BIC as the fitness function. These can help the ENN to converge much faster, as much as half the time of traditional ENNs. The decreased computation time can be of a huge help in fashion retail forecasting as the number of inventory turns are high and sales forecasting occurs just as often, if not more.

Computation power for ENN is lower compared to ANN and are more desirable to be deployed to actual use. Though still it is a considerable amount of time and power, it is well worth the forecasts as we can keep giving it data and the model grows more accurate with more the amount of data it is given. Compared to traditional SARIMA, it is more accurate and feasible to implement.

References

[1] K. F. C. T. M. &. Y. Y. Au, Fashion retail forecasting by evolutionary neural networks, International Journal of Production Economics, 615-630., 2008.

[2] R. M. N. J. Cortez P., Time series forecasting by evolutionary neural networks, Artificial Neural Networks in Real-Life Applications, 47–70, 2006.

[3] S. K. Y. &. B. H. S. Panigrahi, “Time series forecasting using evolutionary neural network,” *International Journal of Computer Applications,* vol. 75, no. 10, 2013.

[4] S. A. J. Bhadouria, “Development of ANN Models for Demand Forecasting,” 2017.

Appendix

The code used for constructing the ENN is as follows-

```
In [1]: lol = [1.25E-
04,0.15379,0.06553,0.24993,0.04977,0.24993,0.07262,0.24993,0.06553,0
        l
        o
        l

Out[1]:
[0.000125,
         0.15379,
         0.06553,
         0.24993,
         0.04977,
         0.24993,
         0.07262,
         0.24993,
         0.06553,
         0.17993,
         0.08366,
         0.16993,
         0.15694,
         0.24993]

In [11]: def create_population(self, count):
             """Create a population of random networks.
              Args:
             count (int): Number of networks to generate, aka
             the size of the population
             """
             pop = []
             for _ in range(0, count):
                 # Create a random network.
                 network = Network(self.nn_param_choices)
                 network.create_random()

                 # Add the network to our population.
                 pop.append(network)

             return pop

In [16]: from random import randint
         from operator import add
         import functools
In [13]: def individual(length, min, max):
             return [randint(min,max) for x in range(length)]
```

```
In [14]: def population(count, length, min, max):
             return [individual(length, min, max) for x in
             range(count)]

In [51]: from keras.models import Sequential
         from keras.layers import Dense
         from sklearn import datasets, linear_model
         from sklearn.model_selection import train_test_split
         import numpy as np def fitness(individual, target):
             X_train = np.multiply(individual,lol)
             X_train = X_train.reshape(1,14)
             target = np.array[target]
             Y_train = target model = Sequential()
             model.add(Dense(14, input_dim=14, activation='relu'))
             model.add(Dense(7, activation='relu'))
             model.add(Dense(1, activation='relu'))
             # Compile model
             model.compile(loss='binary_crossentropy', optimizer='ad
             metrics=['accuracy
             # Fit the model
             model.fit(X_train, Y_train, epochs=150)
             # evaluate the model
             scores = model.evaluate(X_train, Y_train)
             return scores

In [52]: X_train = np.multiply(individual(14,0,1),lol)
         sef = X_train.T sef.shape
         X_train.reshape(1,14)

Out[52]: array([[ 0. , 0.15379, 0.06553, 0. , 0. , 0.24993, 0.07262,
0.24993, 0.06553, 0. , 0.08366, 0.16993, 0. , 0.24993]])

In [53]: from random import random, randint
         def grade(pop, target):
             summed = 0
             for x in pop:
             summed = fitness(x, target) + summed
             return summed / (len(pop) * 1.0)
         def evolve(pop, target, retain=0.2, random_select=0.05,
    mutate=0.01):
             graded = [ (fitness(x, target), x) for x in pop]
             graded = [ x[1] for x in sorted(graded)]
             retain_length = int(len(graded)*retain)
             parents = graded[:retain_length]
```

```
            # randomly add other individuals to promote genetic
            diversity
            for individual in graded[retain_length:]:
                if random_select > random():
                    parents.append(individual)
            # mutate some individuals
            for individual in parents:
                if mutate > random():
                    pos_to_mutate = randint(0,
                    len(individual)-1)
                    # this mutation is not ideal, because it
                    # restricts the range of possible values,
                    # but the function is unaware of the min/max
                    # values used to create the individuals,
                    individual[pos_to_mutate] = randint(
                        min(individual),
                        max(individual))

            # crossover parents to create children
            parents_length = len(parents) desired_length =
            len(pop) - parents_length children = []

            while len(children) < desired_length:
                male = randint(0,
                parents_length-1)
                female = randint(0, parents_length-1)
                if male != female:
                    male = parents[male]
                    female = parents[female]
                    half = len(male) / 2
                    child = male[:half] + female[half:]
                    children.append(child)
                    parents.extend(children)
                    return parents

In [54]: target = 0.23569
         p_count = 100
         i_length = 14
         i_min = 0
         i_max = 1
         p = population(p_count, i_length, i_min, i_max)
         fitness_history = [grade(p, target),]
         for i in xrange(100):
         p = evolve(p, target)
         fitness_history.append(grade(p, target))
         for datum in fitness_history:
             print(datum)
```