Automated Neural Network Structure Determination via Discrete Particle Swarm Optimization (for Non-Linear Time Series Models)

Alex Kalos
Research & Engineering Sciences, Core Research & Development
The Dow Chemical Company
Freeport, Texas, 77541
USA

Abstract: - Due to their universal function approximation capability, artificial neural networks have enjoyed widespread use from research and engineering, to finance and banking applications. However, there are two barriers to their acceptance as mainstream data mining tools: 1) the trial-and-error nature of designing the optimal structure, suitable for a particular application and 2) difficulty of interpretation of results. In this paper, we attempt to address the former aspect by describing the use of a discrete version of Particle Swarm Optimization for automating the design of neural networks. The methodology is applied to a case study for selecting the optimal structure for multivariate nonlinear time series models for the day-ahead forecasting of electricity prices.

Key-Words: - Neural network structure optimization, Discrete Particle Swam Optimization

1 Introduction

With the exception of self-organizing maps (SOM), particularly Kohonen type networks which are inherently self-adaptive [1], [2], [3], the general class of feedforward neural networks has traditionally required a fair amount of manual tuning and manipulation. Because of this, despite their universal function approximation capability [4], neural networks are often avoided or even viewed with mistrust and this has provided a barrier to their use by non-experts. Contributions have been made recently addressing various aspects of this general problem, including self-tuning of network training parameters [5], automated determination of the number of inputs via genetic algorithms [6], improvements in the training algorithms for radial basis function networks [7] and automated basis function selection [8], and the use of alternate training algorithms instead of back propagation, such as genetic programming [9], or Levenberg-Marquardt optimization [10] for estimating network weights. Evolutionary computing techniques have been used to cooperatively co-evolve sub-components of neural network structures [11].

Recently, we reported a heuristic method of growing neural network structures via combinatorial exhaustive search of various network structural elements, such as the variable input pattern, the number of nodes in the hidden layer and the lag structure in nonlinear time series models [10].

Although the heuristic exhaustive search approach guarantees finding the optimal structure, it is computationally intensive requiring the evaluation of a very large number of neural networks. On the other hand, the heuristic method also showed that several suitable models exist, near the optimum point. Therefore, it appeared reasonable to attempt to cast the heuristic approach within an optimization framework, with the expectation that even if the "best" possible structure could not be found, at least one or several useful models would be obtained. We considered the use of various optimization techniques and finally settled on Particle Swarm Optimization (PSO) because it is a) relatively simple to implement, b) has a small number of control parameters and it is robust in the sense that one set of parameters is suitable for a variety of applications, with explicit control over the exploration vs. exploitation modes, and c) because of its population-oriented nature, it has the potential to return an entire set of near-optimum models.

The methodology is applicable to classical "static" feed-forward neural networks. Here, we use it for developing nonlinear multivariate autoregressive time series forecasting models for energy price indices (natural gas and electrical price indices). The system was implemented in *Mathematica* [12].

2 Methodology

2.1 General Problem

The general problem we wish to solve falls in the class of multivariate autoregressive models (1). Elements that need to be tuned include the number and identity of inputs (i.e., which of the potential candidate variables to include), the number of nodes in the hidden layer of the network and the optimal lag structure:

$$\hat{y}(t) = f(y(t-1)...y(t-l), x_k(t-1)..x_k(t-L_k)) \quad (1)$$

where, k=1…p, p is the number of x (input) variables, y is the output variable, t is time, l and L_k are the lag steps for the y and each of the x variables respectively, and $\hat{y}$ is the predicted output.

In the case study discussed later, we wish to find a network that can make a step-ahead prediction for an energy index (e.g., the natural gas opening price on the New York Mercantile Exchange (NYMEX)), based on historical data of that index as well as other energy price indices and related data. Ideally, the best combination of elements would be found automatically within an optimization framework.

2.2 Optimization Framework

2.2.1 Classic Particle Swarm Optimization (PSO).

PSO is a stochastic optimization technique inspired by bird flocking or fish schooling that was introduced by Kennedy and Eberhart [13], [14]. It shares many features with other evolutionary techniques in that it is population based and uses random techniques to search for the optimum fitness, however, unlike genetic algorithms it has no evolutionary operators such as crossover or mutation. In PSO, particles are "thrown" into the search/state space and they "fly" in this space by adjusting their position in a direction that is essentially a weighted average of three possible components: the direction that the particle is already on, the direction of its best prior fitness, and the direction of the best fitness achieved by either the particle's neighbors or the entire population. The contribution of each of these components can be adjusted by control parameters. The basic algorithm of classic PSO is shown in (2):

$$\begin{cases} v_{t+1} = v_t + c_1 r_1 (p_{i,t} - x_t) + c_2 r_2 (p_{g,t} - x_t) \\ x_{t+1} = x_t + v_{t+1} \end{cases} \quad (2)$$

where,

v_t = particle's velocity at time step t

x_t = particle's position at time step t

$p_{i,t}$ = particle's best previous position, at time step t

$p_{g,t}$ = global best previous position, at time step t

r_1, r_2 = random numbers between (0,1)

c_1, c_2 = "exploitation" and "exploration" rates

Many variations of the basic algorithm have been reported, including hybrids of PSO with other local optimizers [15].

One reason for selecting PSO particularly for this application, where automation is the primary motivation, is that it has few adjustable parameters, and it is robust in the sense that one version with a few slight modifications can work well for a wide range of problems.

2.2.2 Adaptation of PSO to Operate in Discrete Parameter Space.

Generally, in the classic version of PSO, both the parameter search space and the state space are continuous. In our case, the state space (i.e., the fitness function RMSE, resulting from neural network evaluation) is continuous, but the parameter search space is discrete. We need to have integers, not fractional values, for the number of nodes (M) and the number of lag steps (L), $(M, L) \in Integers$. Also, a certain input variable to the network is either used or is not. In the later case, the parameters are binary. Binary versions of PSO have been reported [16]. It is relatively easy to formulate a generalized version of PSO, discrete in the search space, by restricting velocity step changes to take on only integer values and as a consequence for the positions to also be integers, $(x, v) \in Integers$ in (2). For example, for a given range in velocity changes of {-10, 10}, the following velocity changes are calculated (see Table 1). The Velocity for several iterations are shown, which result in the positions shown in Table 2.

Table 1. Velocity changes via discrete PSO for a 5-particle population in a dimension with velocity range {-10,10}.

	Velocity Changes				
i	**P1**	**p2**	**p3**	**p4**	**p5**
1	-8	-7	8	-3	-3
2	-8	2	8	0	-2
3	-8	10	-2	3	-4
4	4	3	-8	2	-4
5	10	-8	-2	-1	2
6	10	-6	7	-2	8
7	-5	5	6	1	8
8	-8	10	-3	4	8
9	2	2	-8	2	-2
10	9	-8	-2	2	-8
11	2	-6	7	-1	-2

Table 2. Particle positions after discrete velocity adjustments shown in Table 1.

	Particle Positions				
i	P1	p2	p3	p4	p5
1	18	7	24	18	15
2	10	8	31	18	14
3	2	18	28	21	10
4	6	21	20	22	6
5	16	12	18	21	8
6	26	6	25	18	16
7	21	11	30	19	24
8	12	21	27	22	31
9	14	22	20	24	28
10	23	14	18	26	20
11	26	8	25	25	18

2.2.3 Encoding/Decoding of Input Variable Patterns

In driving toward implementing the evolution of neural network structures within an optimization framework, it would be convenient to represent possible input variable patterns as a single numbers, rather than combinatorial lists. The optimizer would then manipulate a single number, instead of the actual variable patterns themselves. For example, if we have three input variables to the neural network, the following is the complete set of possible variable patterns: {{1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2,3}}.

In our initial explorations, where we performed an exhaustive evaluation of all possible variable patters, this was indeed the way we would generate such patterns - the algorithm would then step through each pattern and evaluate each of these patterns as individual neural networks. Now, in our optimization approach, we aim for the optimizer to choose to generate neural nets for only a small subset of these patterns.

One approach would be to have the optimizer explicitly control the presence/absence of each variable. For example, we could have the optimizer generate patterns like {0, 0, 1}, {0, 1, 0}, {1, 0 1}, etc, where the first pattern would mean to use only the third input variable, and the last pattern would indicate to use input variables one and three. There are two problems with this approach: a) as the number of input variables increases we would end up with a high-dimensional but rather shallow parameter search space, and b) which is more of an implementation issue, due to how the optimizers are set up, each of the input variables would have to be coded as a specific optimization parameter, which would require some wrapper around the function in order to make it generic, as the number of variables change from problem to problem.

An alternative would be to let the optimizer manipulate just one number. This number would essentially take the range of the 2^p - 1(where *p* is the number of variables). For example, again for three potential input variables, the maximum value for this optimization parameter would be 7. The binary representation of 7 is 111, the list-oriented representation being {1, 1, 1}. The optimizer would then be constrained to generate an integer between 0 and 7. So, for this example, the value (0 or 1) of each binary digit in this number would indicate whether the corresponding variable would be used.

In addition to permitting a very simple and compact representation of potentially complex variable patterns via just a single integer, what is also useful about this is that it can significantly cut down the number of parameters that have to be explicitly manipulated by the optimizer, so this should speed up computing. As it is, what goes on inside the objective function is so complex and non-linear already, using a single indirectly manipulated variable vs. many directly manipulated ones, should not be of consequence in terms of "contributing" to the non-linear behavior. Furthermore, this representation paves the way for potential implementation using other evolutionary techniques (e.g., genetic algorithms) which work well using binary encoded representations.

Finally, this approach facilitates visualization of the optimization process. For example, it is easy to see all twelve input variables on one 2-dimensional plot). In addition, the methodology is easy to extend. Such an extension is explored later, where the lag structure is "tied" to the input variable pattern.

2.3 Neural Network Evaluation – Fitness Function

To generate the fitness values, we evaluate the neural networks according to the procedure that was reported for heuristically growing of neural networks [10]. A brief overview is provided below:

2.3.1 Data Pre-processing

We partition the data into three sets: the *training set (*consisting of the *estimation*, *validation* subsets), and the *test* set. There are various strategies for partitioning the data. Since we are building time series models, we have opted to partition the data such that the sets are consecutive in time. This kind of partitioning allows to objectively evaluate our models' performance in the future. We try to adhere as much as possible to the 80/20 split [17] between the training/test sets, so we end up with, i.e., 60:20:20 for estimation, validation, and test sets respectively. Finally, the target output variable as well as all potential input variables are standardized to avoid instabilities in the resulting neural network models [4].

2.3.2 Neural Network Weight Estimation

The Levenberg-Marquardt (LM) [18] optimization method is used for estimating the network weights [19]. This method is a hybrid of the Gauss-Newton (NG) conjugate-gradient method and of the steepest descent (SD) method, as a result it works well for ill-defined minimization problems (i.e., where the Hessian is ill-defined [4]). Equation (3) shows the learning law for the LM method:

$$\Delta_w = -(J^T J + \lambda I)^{-1} J^T \varepsilon \qquad (3)$$

where ε is the error vector, J is the Jacobian of the error vector with respect to the weights, J^T is the transpose of the Jacobian, I is the identity matrix and λ is the control parameter.

The LM method is more robust than standard back propagation (SB) [20], due to this switching between the two modes, which ensures less entrapment in local minima, with quick search at near minimum conditions. On the other hand SB, being strictly a steepest descent method, is more prone to getting stuck in local minima, requiring manual tuning of the momentum parameter. The other advantage of LM is that the control parameter is manipulated automatically, and so it is preferred especially in this type of application which requires a large number of unattended runs.

The stop-search (or early stopping) method of cross-validation is used for determining the optimal set of weights for a given neural network [4], [21], [22]. With this method (see Figure 1), the weights are adjusted during training on the basis of the root mean-squared error (RMSE) shown in (4), of the *estimation* data set, but the accepted set of weights is that which corresponds to the minimum RMSE for the *validation* data set.

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - f(w_i, x_{ki}))^2} \qquad (4)$$

for k=1…p, where p is the number of x (input) variables, y is the measured output variable, N is the number of (x_{ki}, y_i) observations, w is the weight vector determined during training, and f is the neural network model.

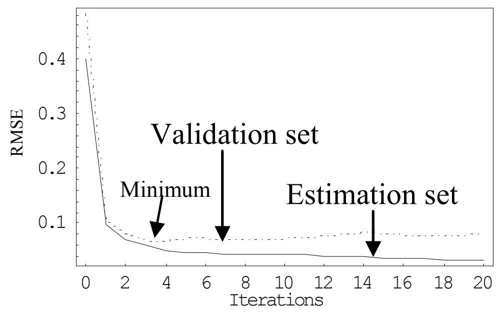

Fig. 1. Illustration of the Stop-Search Method of Cross-Validation.

3 Case Study

We wish to find a model to forecast the day-ahead on-peak electricity price in the southern region of the United States. Daily values of on-peak and off-peak electricity prices from various regions in the United States from the Electric Reliability Council of Texas (ERCOT) database [23], as well as natural gas prices from the New York Mercantile Exchange (NYMEX) database [24] are used (see Table 3). The data spans from 8-March-2002 to 24-Feb-2004, a total of 493 data points with a 64%:18%:18% split between the estimation, validation and test sets respectively.

Table 3. Variables used in Case Study

Variable	Description
x1	Seller on-peak electricity price
x2	North USA on-peak electricity price
x3	Houston USA on-peak electricity price
x4	West USA on-peak electricity price
x5	North USA off-peak electricity price
x6	Houston USA off-peak electricity price
x7	West USA off-peak electricity price
x8	South USA off-peak electricity price
x9	NYMEX natural gas opening price
x10	NYMEX natural gas high price
x11	NYMEX natural gas low price
x12	NYMEX natural closing price
y	South USA on-peak electricity price

3.1 Optimization Runs

3.1.1 Neural Networks with a Constant Number of Nodes and a Fixed lag structure (Case 1)

This is the simplest case, where the number of nodes in the single hidden layer is kept fixed to M=3, and the number of lags is also kept fixed to L=4. Only the variable pattern, encoded to vary from 1 to 2^{12} - 1 = 4095, as described in section 2.2.3, is controlled by PSO.

3.1.2 Neural Networks with a Variable Number of Nodes and a Fixed Lag Structure (Case 2)

In this case, the number of nodes is also a parameter optimized by PSO, but it is an ordinary parameter (not encoded as is the input variable pattern) and is allowed to vary between {0, 3}, inclusive.

3.1.3 Optimization Including the Lag Structure (Cases 3 and 4)

The simplest way of allowing optimization of the lags is to use one global parameter for all variables in the input variable pattern. In this case, the value returned by PSO is used to set the number of lag steps between {1, L_{max}}, inclusive. The value of L_{max} is determined by examining the spectral density plots and autocorrelation plots from classical auto regressive/ moving average (ARMA) time series analysis [25]. For this case study the value L_{max} = 4 was used. This is a somewhat rigid lag structure in that every input variable has to use the same number of lag steps and the same offset from the y variable.

Table 4. Decoding of the Lag Structure Corresponding to the Optimizer Manipulated Parameter for a 2-Lag Step Input Variable

Manipulated parameter	Lag structure	Meaning
0	{0 ,0, 0}	variable not used at all
1	{0 ,0, 1}	variable only, no lags
2	{0 ,1, 0}	t-1 lag step only
3	{0 ,1, 1}	variable and t-1 lag step
4	{1 ,0, 0}	t-2 lag step only
5	{1 ,0, 1}	variable and t-2 lag step
6	{1 ,1, 0}	t-1 and t-2 lag steps
7	{1 ,1, 1}	variable and both lag steps

Another approach to incorporate a more flexible lag structure into the optimization framework would be to “tie” the lags to the input variable pattern. Otherwise, if the input variable pattern and lag structure are decoupled, the optimizer may generate lag structures for non-existent input variable patterns or visa-versa. To accommodate this, an extension of encoding/decoding scheme can be used where each input variable and its corresponding lag structure is represented by a single number. For example, if an input variable has a maximum of two lag steps the manipulated parameter would range from 0 to 7, and each value would correspond to the pattern shown in Table 4. The disadvantage of this approach is that for our case study, we would have 13 parameters to optimize instead of 3; 12 coded variables for each input and its corresponding lag structure, plus 1 for the number of nodes, as opposed to 1 coded variable for all 12 inputs, plus 1 for the number of nodes, plus 1 for the global number of lag steps.

3.2 Evaluation of Optimization Performance

The main observations are that by using discrete PSO: a) we often found at least one model with the best or at least a near-best fitness value, and b) we did so with a substantially reduced number of neural network evaluations that would have been required otherwise.

To illustrate, we first performed one exhaustive evaluation of all networks that can be constructed from all possible combinations of the 12 input variables (singlets, pairs, triplets, etc), with 0 to 3 nodes (Case 3, for a total of 16380 networks); and another, where the number of lag steps was also varied between 1 to 4 (Case 4, for a total of 65520 networks). It is noted that when the number of nodes is 0, this reduces to a linear system. The plot of the fitness values for Case 3 is shown in Fig. 2. The corresponding plot for Case 4 is shown in Fig 3. The plot of the top 200 models with the fitness values sorted according to best fitness (lowest RMSE) for case 3 is shown in Fig. 4.

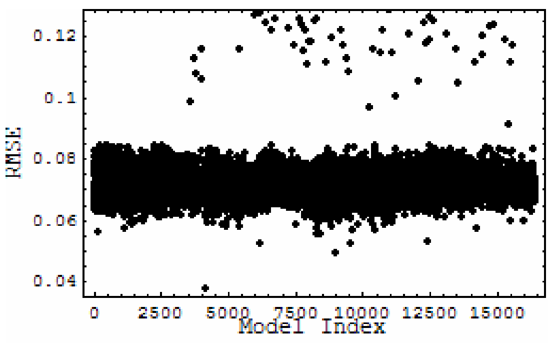

Fig. 2. Fitness values for Case 3.

For each of these cases, 100 PSO runs were executed. Each run for Case 3 was set up with the following PSO parameters: 20 particles, 20 iterations, with exploitation rate of 1.0, and exploration rate of 0.5. The same parameters were used for Case 4, except that the number of iterations was set to 40. The results for Case 3 are shown in Table 5.

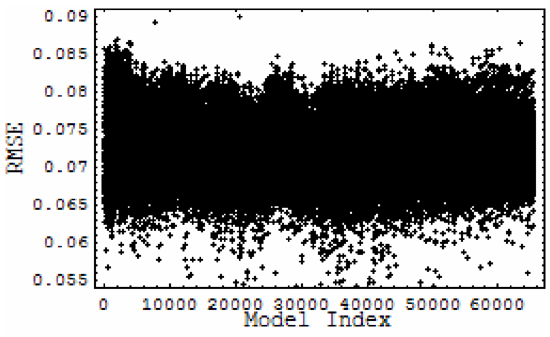

Fig. 3. Fitness values for Case 4.

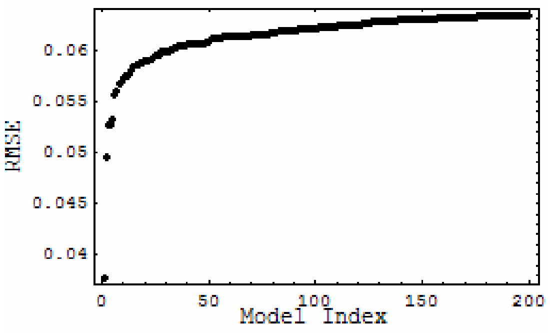

Fig. 4. Fitness values for the top 200 models of Case 3, sorted from best to worst (left to right).

Table 5. Performance of Discrete PSO for Case 3.

Ranking of Fitness value (%)	Number of Solutions with this rank or less	Percent of PSO runs
0.1	16	24
0.2	33	37
0.3	49	54
0.4	66	71
0.5	82	76
0.6	98	85
0.7	115	90
0.8	131	93
0.9	147	94
1.0	164	98

The first entry in the table shows that 24% of the PSO runs found at least one of the top 0.1% best fitness values (16 out of 16380 possible values). The last entry shows that 98% of the PSO runs found at least one of the top 1% best fitness values. Usually, several "hits" were found in each category. While, sometimes a hit would be found after the first iteration, other times it was found at the last iteration. The overall mean iteration count for finding at least one model with a fitness value among the top 1% of the best models was 10. This corresponds to an average of 200 neural net evaluations, out of a total 16380 possible (or 1.2%).

The corresponding results for Case 4 are shown in Table 6. In this case, the overall mean iteration count for finding a model with a fitness in the top 1% of the top models, was 0.7%, 440 neural net evaluations out of a total possible 65520.

Table 6. Performance of Discrete PSO for Case 4.

Ranking of Fitness value (%)	Number of Solutions with this rank or less	Percent of PSO runs
0.1	66	29
0.2	131	59
0.3	197	75
0.4	262	88
0.5	328	92
0.6	393	98
0.7	459	100
0.8	524	100
0.9	590	100
1.0	655	100

4 Conclusions

We have developed a discrete version of Particle Swarm Optimization and used it to select structural design elements to automatically construct feed-forward multivariate neural networks. We applied the methodology to develop nonlinear time series models for predicting day-ahead electricity prices. We found that PSO works well for this application and is able to consistently find useful, near-optimum models, requiring a relatively small number of neural network evaluations.

References:

[1] T. Kohonen, *Self-Organizing Maps*, Springer, Berlin, 1997.

[2] G. Leng, G. Prasad, and T.M. McGinnity, An on-line algorithm for creating self-organizing neural networks, *Neural Networks*, 17, 2004, pp.487-493.

[3] B. Hammer, A. Micheli, A. Sperduti, and M. Strickert, Recursive self-organizing network models *Neural Networks,* 17, 2004, pp. 1061-1085.

[4] S. Haykin, *Neural Networks: A Comprehensive Foundation,* Second Edition, Prentice Hall, New Jersey, 1999

[5] C-T. Chen and W-D. Chang, A Feedforward Neural Network with Function Shape Auto-tuning, *Neural Networks*, Vol. 9, No. 4, 1996, pp.627-641.

[6] S-K. Oh and W. Pedrycz, A new approach to self-organizing multi-layer fuzzy polynomial neural networks based on genetic optimization, *Advanced engineering Infomatics*, 18, 2004, pp. 29-39.

[7] I. Pitas, C. Kotropoulos, N. Nikolaidis, and A. Bors, Robust and Adaptive Techniques in Self-Organizing Neural Networks, *Nonlinear Analysis, Theory & Applications*, Vol. 30, No. 7, 1997, pp 4517-4528.

[8] A. Ghodi and D. Schuurmans, Automatic basis selection techniques for RBF networks, *Neural Networks*, 16, 2003, pp. 809-816.

[9] D. B. Fogel, L. J. Fogel, and V. W. Porto, Evolving Neural Networks, *Biol. Cybern.,* 66, 1990, pp. 487-493.

[10] A. N. Kalos, Automated Heuristic Growing of Neural Networks for Nonlinear Time Series Models, *IEEE International Joint Conference On Neural Networks*, Montreal, Canada, 2005, in press.

[11] N. Garcia-Pedrajas, C. Hervas-Martinez, and J. Munoz-Perz, Multi-objective cooperative coevolution of artificial neural networks (multi-objective cooperative networks), *Neural Networks,* 15, 2004, pp. 1259-1278.

[12] *Mathematica* 5.0 by Wolfram Research, http://www.wolfram.com. The Neural Networks package is an add-on component to the standard *Mathematica* software and must be purchased separately.

[13] J. Kennedy and R. C. Eberhart, Particle Swarm Optimization, *IEEE International Conference on Neural Networks*, Perth, Australia, IEEE Service Center, Piscataway, NJ, 1995.

[14] J. Kennedy, R.C. Eberhart, and Y. Shi, *Swarm Intelligence*, San Francisco: Morgan Kaufmann, CA, 2001.

[15] S. R. Katare, A. N. Kalos, D. H. West, A Hybrid Swarm Optimizer for Efficient Parameter Estimation, *Proceedings of the 2004 Congress on Evolutionary Computation, CEC2004*, Portland, Oregon, June 19-23, 2004, pp. 309-315.

[16] Kennedy, J. and R. C. Eberhart, A discrete binary version of the particle swarm algorithm, *International Conference on Systems, Man, and Cybernetics*, 1997.

[17] M. Kearns, A bound on error of cross validation using the approximation and estimation rates, with consequences for the training-test split, *Advances in Neural Information Processing Systems,* Vol. 8, Cambridge, MA, 1996, pp. 183-189.

[18] D.W. Marquardt, Journal of the Society for Industrial and Applied Mathematics, vol. 11, 1963, pp. 431-441.

[19] R. Fletcher, *Practical Methods of Optimization*, John Wiley & Sons, Chippenham, Great Britain, 1987.

[20] P. J. Werbos, *Beyond regression: New tools for prediction and analysis in the behavioral sciences*, Ph.D. Thesis, Harvard University, Cambridge, MA, 1974.

[21] J. Sjoberg and L. Ljung, Overtraining, Regularization, and Searching for Minimum with Application to Neural Nets, *Int. J. Control*, vol. 62, no. 6, 1995, pp. 1391–1407.

[22] J. Sjoberg and M. Viberg, Separable Non-linear Least-squares minimization—Possible Improvements for Neural Net Fitting, *IEEE Workshop in Neural Networks for Signal Processing*, Amelia Island Plantation, Florida, Sep. 24–26, 1997, pp. 345–354.

[23] Electric Reliability Council of Texas (ERCOT), http://www.ercot.com/.

[24] New York Mercantile Exchange (NYMEX), http://www.nymex.com/.

[25] G. Box, G. Jenkins, and G. Reinsel, *Time Series Analysis - Forecasting and Control*, Third Edition., Pearson Education, Inc, 1994.