Advancements and Implementation of Polynomial-Based Learning Machines for Data Processing and System Modeling

by

Michael Sylvester Pukish III

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
May 4, 2014

Keywords: Artificial neural networks, computational intelligence, machine learning, universal approximation

Copyright 2014 by Michael Sylvester Pukish III

Approved by

Bogdan Wilamowski, Chair, Professor of Electrical and Computer Engineering
Fa Foster Dai, Co-chair, Professor of Electrical and Computer Engineering
Thaddeus Roppel, Associate Professor of Electrical and Computer Engineering
Vitaly J. Vodyanoy, Professor of Physiology
N. Hari Narayanan, Professor of Computer Science and Software Engineering

Abstract

At the present time, the need in all disciplines for efficient and powerful algorithms for the handling of large and complex datasets is certainly at its highest. Extremely large multi-dimensional datasets are commonplace in archival climatology and weather prediction, image processing, biology, genetics, industrial electronics, financial analysis and forecasting, telecommunications, cyber security, and throughout the social sciences. In addition to the size and high dimensionality of the data, agile real-time systems are needed to process such information for interpolation and extrapolation implementations applied toward control systems, data streaming and filtering, and simulation and modeling.

In the interest of analysis and manipulation of the “big data” associated with such disciplines and tasks, certain techniques have come and gone over time, leading to a current subset of prevalent Computational Intelligence (CI) techniques. Throughout the fields of computer science and electrical engineering, these particular techniques have risen to their present popularity largely due to their existing familiarity and positive track record among researchers and engineers. Such techniques include fuzzy systems, Artificial Neural Networks (ANN), Radial Basis Function (RBF) networks, Support Vector Machines (SVM), Gaussian Processes (GP), and Evolutionary Computation (EC) (of which Genetic Algorithms (GA) are a predominant subset). Specific variants of some of these methods include Support Vector Regression (SVR) and a currently popular subset of RBF-based neural networks known as Extreme Learning Machines (ELM). Both of those variants and some of the more general techniques will be highlighted further in this work.

Historically, Polynomial-Based Learning Machines (PLM) had been used for the same classes of problems mentioned thus far. However, unwieldy kernel functions (in the form of large, high-order polynomials) and relatively limited

computer speed and capacity had limited the use of PLMs to comparatively small problems with low dimensionality and simple functional relationships among inputs and outputs. Thus, polynomial-based solutions within CI have, for the most part, drifted out of vogue for at least two decades.

This work attempts to reinvigorate the interest and viability of PLMs for use throughout all applications of CI by introducing enhancements for their implementation. It will be shown that once certain algorithms are applied to the generation, "training", and functional operation of PLMs, PLMs compete on par with the predominant methods currently in use, and in many cases perform with superior efficiency, compute time, and accuracy. Functional enhancements will be explained, and seven variants of a new generation of PLMs will be compared alongside the predominant CI techniques, through experimentation with a variety of problem types ranging from real-time industrial applications to approximation of benchmark "big data" sets.

Acknowledgements

I am and always will remain thankful to the institution of Auburn University, and to the people therein, particularly in the Department of Electrical and Computer Engineering, and in the Samuel Ginn College of Engineering in general. I would firstly like to thank the members of my Dissertation Committee for their patience, guidance, and expertise in making this research and opportunity possible. Foremost, I would like to thank my Chair, Professor Bogdan Wilamowski, for his persistent dedication to mentoring myself and all students who are either formally in his group, or who merely come to knock on his door. Professor Wilamowski is known for his unmatchable persistence in encouraging, instructively coercing, tirelessly and always effectively teaching, and relentlessly doing at least as much work as his students, night or day, whether he is presently local or traveling continents away. I would also like to specifically thank my Co-chair, Professor Foster Dai, for providing support, kind and patient guidance, and invaluable project opportunities continuing from the first days of my graduate studies in August, 2009. I would lastly like to thank my fellow colleagues and graduate students who I have had the pleasure of knowing and working alongside. All have helped me at some point, with knowledge in our field and most importantly with camaraderie, friendship, and humor. These fellow students include but are not limited to (in alphabetical order) Tim Brown, Joseph Cali, Parameshwaran Gnanachchelvi, Stephan Henning, Zachary Hubbard, Philip Reiner, Jordan Richardson, Christopher Wilson, and Xing Wu. I would also like to thank all colleagues of the Foster Dai research group.

Table of Contents

Abstract ii

Acknowledgements iv

List of Tables x

List of Figures xii

List of Abbreviations xvi

Chapter 1 Background of Polynomial Networks 1

Introduction 1

1.2 Existing Polynomial Networks 3

1.2.1 Development of the Group Method of Data Handling (GMDH) 3

1.2.2 Evolution of the GMDH Model 9

1.2.3 Divergence from the GMDH Model 11

1.2.4 Functional Link Networks – Complete Polynomials 13

1.2.5 Summary of the State of the Art for Polynomial Systems 16

Chapter 2 Implementation of Competing Methods 17

2.1 The Single-Layer Feed-Forward Neural Network (SLFN) 17

2.1.1 Competitive Considerations of Neural Networks 19

2.2 The Takagi-Sugeno-Kang Fuzzy System 20

2.2.1 Development of a Novel Data-Driven N-Dimensional TSK Fuzzy System 23

2.2.1.1 Fast Recursive N-dimensional Interpolation 24

2.2.1.2 Completion of an N-dimensional TSK Fuzzy Engine 27

2.2.2 Radial Basis Function Learning Machines 28

2.2.2.1 Review of RBF Networks 28

2.2.2.2 The Extreme Learning Machine RBF Variants 30

2.2.2.3 Support Vector Regression with RBF Kernels 31

Chapter 3 Computational Strategies to Improve Polynomial Network Performance 33

3.1 Efficient Generation of Monomial Polynomial Terms 33

3.2 Statistical Smoothing and Pruning of Monomial Coefficients 36

3.2.1 A Training Strategy to Stabilize and Isolate Noise-Responsive Coefficients 37

3.2.2 Forward-Computed Statistical Methods – Overview 38

3.2.3 Forward-Computed Statistical Methods – Iterative Numerical Detail 40

3.3 Exploration of Enhanced Regression Techniques 43

3.3.1 An Iterative Generalized Least Squares Regression Technique 44

3.3.2 An Iterative Ridge Regression Technique 47

3.3.2.1 Ridge Regression – Theory 49

3.3.2.2 Ridge Regression – Implementation 51

3.3.3 Iterative Regression Hybrids Using OLS, GLS, and Ridge Methods 53

3.3.3.1 The GLS + Ridge Regression Minimum-Variance Hybrid 54

3.3.3.2 The GLS + Ridge Regression Full Optimization Hybrid 55

3.4 Automated N-Dimensional Radial Clustering 56

Chapter 4 Proposed Polynomial-Based Learning Machines: Seven Variants within Three Species 59

4.1 PolyNet Species – The Initial Next-Generation Polynomial Learning Machine 59

4.2 The PolyStat Species – Utilizing Statistical Pruning of Monomial Terms 60

4.3 The PolyPaP Species – A Probe-and-Prune Methodology 62

4.4 Summary of Polynomial-Based Learning Machine Variants 66

Chapter 5 Experimental Results 67

5.1 Test Methodology 67

5.2 Experiments with Industrial Electronics Problems 68

5.2.1 Voltage Control for a Czuk DC-DC Converter 68

5.2.2 3-D Reverse Kinematics Control 69

5.2.3 Results for Two Industrial Electronics Problems 71

5.2.3.1 Training and Validation Times for IE Problems 71

5.2.3.2 Training and Validation Accuracy for IE Problems 74

5.2.3.3 Tabulation of IE Dataset Results 77

5.3 Experiments with Real-World Repository Datasets 79

5.3.1 Training and Testing Times for Real-World Datasets 81

5.3.2 Training and Validation Accuracy for Real-World Datasets 83

5.3.3 Tabulation of Real-World Dataset Results 86

Chapter 6 Conclusions and Future Work 90

6.1 Evaluation of PLM Variants and Competing Methods with an Original Figure of Merit Scheme 90

6.2 Summary Statement 94

6.3 Future Work 94

6.3.1 Improved Coefficient Term Analysis and Pruning 95

6.3.2 Process Pipelining 96

6.3.3 Development of a Universal Polynomial Spline Learning Machine 96

6.3.4 Using Video Card GPUs for Machine Learning Computation 97

Reference Pages 98

References 99

Appendices 107

7.1 Appendix A – Exploration of Chebychev Transform Methods 107

7.1.1 Background, and Two Chebychev Transform Implementations 107

7.1.2 Chebychev Techniques – Experimental Results 110

7.2 Appendix B – MATLAB Code: Unique Polynomial Term Generation 114

7.3 Appendix C – MATLAB Code: Statistical Processing of Monomial Term Weights 114

7.4 Appendix D – MATLAB Code: Iterative GLS Regression 118

7.4 1 1-D 3rd-order Test – GLS_1D.m excerpts 118

7.4.2 General GLS Code for Multiple Dimension Data Regression – GLS_reg.m excerpts 120

7.5 Appendix E – MATLAB Code: Iterative Ridge Regression 122

7.5.1 Computation of an Initial Minimum-variance λ – OLSridge_reg.m Excerpt 122

7.5.2 Iterative λ Optimization Process – OLSridge_reg.m excerpt 122

7.6 Appendix F – MATLAB Code: Hybrid Regression Techniques Including RR 123

7.6.1 Iterative GLS + RR Minimum-Variance Regression – GLSminvar_reg.m excerpts 123

7.6.2 Iterative GLS + RR Full Optimization Regression – GLSridge_reg.m Excerpts 124

7.7 Appendix G – MATLAB Code: Two Types of PLM Implementations 126

7.7.1 The PolyNet Variant – PolyNet.m excerpts 126

7.7.2 The PolyPaP Variants – PlyPaPGLSR.m excerpts 127

7.8 Appendix H – MATLAB Code: An N-Dimensional Data-Driven TSK Fuzzy System 129

7.8.1 Efficient Recursive Interpolation – recurinter.m excerpts 129

7.8 2 The TSK Fuzzy System Engine 131

7.9 Appendix I – MATLAB code: Fast, Forward-Computing N-Dimensional Radial Clustering 134

7.10 Appendix J – Selected Publications by This Author 136

List of Tables

Table I Total Number of Product Terms per Polynomial Order and Number of Inputs 5

Table II Number of Monomial Terms per Dimensions and Inputs 6

Table III The Poly-Gen Algorithm for Generation of Unique Monomial Terms 34

Table IV Example of the Recursive Poly-Gen Function for a 3-input, order-2 Case . 35

Table V Computational Steps for Statistical Processing of Monomial Term Weights 39

Table VI Feature Set of Polynomial-Based Learning Variants 66

TABLE VII Algorithm Efficiencies: Processing Times and Network Size IE Problem . 77

TABLE VIII Algorithm Accuracy: Average Training and Testing RMSEs per IE Problem 79

Table IX Benchmark Datasets: Specifications for 70/30 k-fold Testing 80

Table X Optimal Parameter Settings per Dataset for SVR and PLM Variants 80

Table XI Average Processing Times and Network Size per Datasets 1-3 86

Table XII Average Processing Times and Network Size per Datasets 4-5 86

Table XIII Average Processing Times and Network Size per Datasets 6-7 87

Table XIV Average Training and Testing RMSEs per Datasets 1-4 89

Table XV Average Training and Testing RMSEs per Datasets 5-7 89

Table XVI FOM_{RT}: Real-Time Figure of Merit of Each Algorithm per Dataset 92

Table XVII FOM_{OL}: Offline Figure of Merit of Each Algorithm per Dataset 93

Table XVIII Overall Real-time and Offline FOMs for All Algorithms Tested 94

Table XIX Results for Chebychev Transform Testing, no noise case (RED=worst, GREEN=best) 113

Table XX Results for Chebychev Transform Testing, 5% training noise (RED=worst, GREEN=best) 114

List of Figures

Figure 1 Diagram of a generalized functional link network ... 2

Figure 2 GMDH network for the solution of a 3-input, 3rd (or 4th) order function ... 7

Figure 3 2-input, 2nd-order computational node of the original GMDH architecture . 8

Figure 4 The Banfer and Nelles local model network [35]: The outputs $\hat{y}_i$ of the local polynomial networks (LM_i) are weighted with their associated domain function values (Φ_i) and superposed. ... 12

Figure 5 Polynomial network of degree 2 for 3 input variables ... 15

Figure 6 A 4-input, 5-neuron SLP with 4 neurons in the hidden node ... 18

Figure 7 Typical Iterative ANN Training: "hot spot" issue ... 20

Figure 8 Example of a Trapezoidal Fuzzy Membership Function Operation ... 22

Figure 9 TSK Product Encoding Output: Comparison of membership functions ... 22

Figure 10 Comparison of TSK and ANN (node-based) Output Quality ... 23

Figure 11 (a) View of random generated peaks() values, (b) Overlay with interpolated values ... 26

Figure 12 (a) Original function, 1000 data points, (b) 5x5 grid, (c) 10x10 grid, (d) 20x20 grid ... 27

Figure 13 A typical RBF network containing *H* neurons and *D* inputs ... 29

Figure 14 3-input 3rd-order case: Unique monomial terms on sliding diagonal ... 34

Figure 15 Stabilization of final normalized coefficient means as training proceeds 41

Figure 16 Tracking monomial term coefficient variation over successive training iterations ... 42

Figure 17 1-D, 3rd-order case of bad data, 40% outlaying trend ... 47

Figure 18 Iterative GLS performance vs. OLS: (a) GLS (red) beats OLS (blue), (b) Stabilization of GLS MSE of the transform errors ... 47

Figure 19 Curve-fitting of points with polynomials of increasingly higher order 48

Figure 20 Ridge Regression [83]: The variance-bias tradeoff, and performance vs. OLS 50

Figure 21 Ridge regression teaser: (PolyRidge) vs. OLS regression (PolyNet)......... 53

Figure 22 Iterative Ridge Regression: Both λ (Left) and training RMSE (Right) shrink monotonically towards optimal values during processing 53

Figure 23 Effect of GLS-RR minimum-variance method: minimal difference in training and validation RMSE curves 55

Figure 24 Fast, forward radial clustering of the Matlab peaks() function: (a) 7 cluster centers, top view, (b) 7 cluster centers, 3-D view, (c) 110 cluster centers, top view, (d) 110 cluster centers, 3-D view................. 58

Figure 25 Flowchart for the PolyNet PLM variant........ 60

Figure 26 Flowchart for the PolyStat family of PLM variants 62

Figure 27 Flowchart for the Probe phase of the PolyPaP family of PLM variants..... 65

Figure 28 Flowchart for the Solve/Prune phase of the PolyPaP family of PLM variants 65

Figure 29 A Czuk up-down DC-DC converter circuit 69

Figure 30 The Czuk DCDC Converter: Non-linear transient responses, 0 to 30ms . 69

Figure 31 Czuk Converter: non-linear steady state relationships between load conductance and duty-cycle vs. output voltage........ 70

Figure 32 3-D reverse-kinematics: Resultant arm tip distribution in free-space of randomly generated angle positions 70

Figure 33 (a) Reverse-kinematics problem: input angles are mapped to Cartesian coordinates (x, y, z), (b) x-position vs. two input angles 71

Figure 34 Total training times for all algorithms up to 200 nodes: DC-DC problem 72

Figure 35 Total training times for all algorithms up to 400 nodes: 3-D Kinematics problem........ 73

Figure 36 Validation times per network size vs. nodes for all algorithms: DC-DC problem........ 74

Figure 37 Validation times per network size vs. nodes for all algorithms: 3-D kinematics problem ... 74

Figure 38 Training error for all algorithms up to 200 nodes: DC-DC problem ... 75

Figure 39 Validation error for all algorithms up to 200 nodes: DC-DC problem ... 76

Figure 40 Training error for all algorithms up to 400 nodes: Kinematics problem 76

Figure 41 Validation error for all algorithms up to 400 nodes: Kinematics problem ... 77

Figure 42 Total training times for all algorithms up to 600 nodes: Boston Housing 81

Figure 43 Total training times for all algorithms up to 400 nodes: Machine CPU ... 82

Figure 44 Validation times per network size vs. nodes for all algorithms: Boston Housing ... 82

Figure 45 Validation times per network size vs. nodes for all algorithms: Machine CPU ... 83

Figure 46 Training error for all algorithms up to 600 nodes: Boston Housing ... 84

Figure 47 Validation error for all algorithms up to 600 nodes: Boston Housing ... 84

Figure 48 Training error for all algorithms up to 400 nodes: Machine CPU ... 85

Figure 49 Validation error for all algorithms up to 400 nodes: Machine CPU ... 85

Figure 50 Runge's phenomenon and refit with Chebychev nodes [94] ... 107

Figure 51 Polynomial Network, Chebychev Transform Method 1 – Training: encoding of input vectors, plus regeneration of desired outputs, Validation: straightforward processing with encoded weights and terms ... 109

Figure 52 Polynomial Network, Chebychev Transform Method 2 – Training: encoding of input vectors, Validation: Chebychev encoding of inputs and processing with encoded weights and terms ... 110

Figure 53 Validation for the Chebychev Transform Experiments, 9409 points (97x97) ... 111

Figure 54 9x9 training point grid: (left) standard spacing captures function, (right) Chebychev spacing fails to sample function at critical points ... 111

Figure 55 Training and Validation RMSE Curves (81 training points): (a) no Chebychev spacing, (b) Method 1, (c) Method 2 112

List of Abbreviations

ABFC	Adaptive Basis Function Construction
AICC	Corrected Akaike Information Criterion
ANN	Artificial Neural Network
BLUE	Best Linear Unbiased Estimator
CHSP	Complete Homogeneous set of Symmetric Polynomials
CI	Computational Intelligence
CI-ELM	Convex Incremental Extreme Learning Machine
CPU	Central Processing Unit
DC-DC	Direct-Current to Direct Current Converter
EBP	Error Back-Propagation
EC	Evolutionary Computation
EI-ELM	Enhanced Incremental Extreme Learning Machine
ELM	Extreme Learning Machine
FGLS	Feasible Generalized Least Squares
FLN	Functional Link Network
FOM	Figure of Merit
FPNN	Fuzzy Polynomial Neural Network
FS	Fuzzy System
GA	Genetic Algorithm

gFPNN	Genetically Optimized Fuzzy Polynomial Neural Network
GLS	Generalized Least Squares
GLS-RR	Generalized Least Squares – Ridge Regression (regression hybrid)
GMDH	Group Method of Data Handling
GP	Gaussian Process
GPU	Graphics Processing Unit
IE	Industrial Electronics
I-ELM	Incremental Extreme Learning Machine
LIBSVM	Support Vector Machine Library
LM	Levenberg-Marquardt (algorithm)
LUT	Look-Up Table
ML	Machine Learning
MSE	Mean-Squared Error
NBN	Neuron-By-Neuron (neural network training algorithm)
OLS	Ordinary Least Squares
PANN	Polynomial Artificial Neural Network
PLM	Polynomial-based Learning Machine
RAM	Random-Access Memory
RBF	Radial Basis Function
RMSE	Root-Mean-Squared Error
RR	Ridge Regression
SLP	Single-Layer Perceptron
SOPN	Self-Organizing Polynomial Network

STD	Standard Deviation
SVD	Singular Value Decomposition
SVM	Support Vector Machine
SVR	Support Vector Regression
TSK	Takagi-Sugeno-Kang (algorithm)
UA	Universal Approximation

Chapter 1
Background of Polynomial Networks

1.1 Introduction

The field of Computational Intelligence (CI) currently includes several prominent areas: Artificial Neural Networks (ANN), Fuzzy Systems (FS), Radial Basis Function (RBF) networks, Support Vector Regression (SVR), Gaussian Process (GP) methods, and evolutionary computation (EC) techniques. A major interest in these areas is in their ability to approximate non-linear functions with relative efficiency, computational speed, and accuracy. As a result of these characteristics, CI systems are most prominently used as control systems across such varied areas from motor actuation and control [1]–[4], to power systems [5]–[8], an on to an array of complex problem solving such as fault detection [9]–[11] and even water quality prediction [12].

Along the road to this current array of choices, conceptually and architecturally simple ideas have been introduced in previous decades in the form of Functional Link Networks (FLN). These networks, explored by researchers such as Pao [13], feature a single-layer architecture of an unlimited number of non-linear function-generating nodes, and a single summation output node. Single-layer network methods such as RBF, SVR, etc., with different node functions for each node, are actually subsets of FLN. In terms of the application of such a network to Machine Learning (ML), such a network can train a set of multi-dimensional input patterns to an equal number of desired output patterns using an appropriate single-layer method, such as linear regression [14]. Figure 1 shows such a general FLN with multi-dimensional inputs, **Xm**, and a potentially unlimited number of non-linear sub-function generators, **Fn**, which can theoretically approximate a complex non-linear function, **Z.** Note that multiple outputs, **Z**, could be approximated with

the inclusion of multiple parallel output summation stages. Each would be trained separately to the same patterns fed to the same or subset of same inputs, **Xm**, as necessary to produce multi-dimensional outputs.

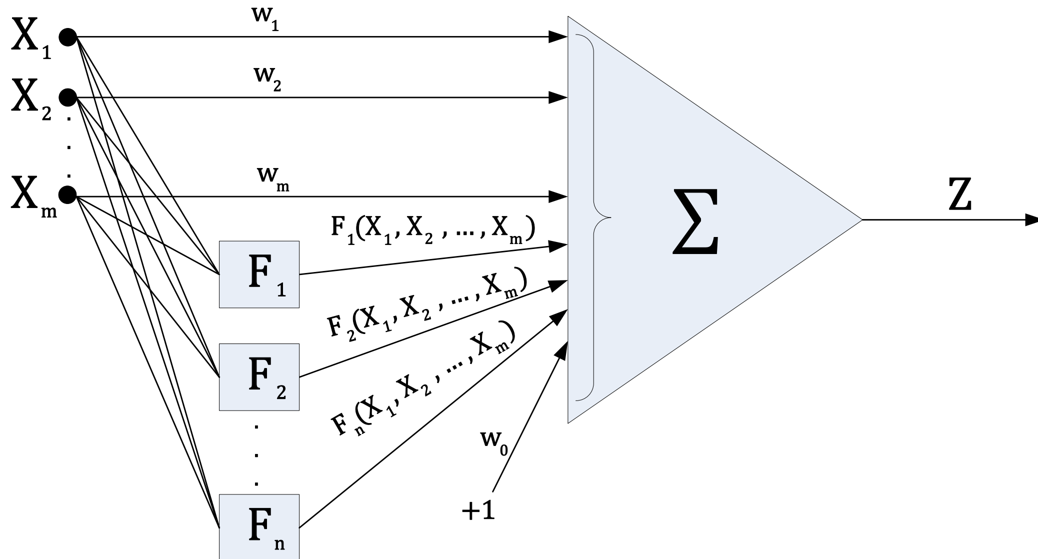


Figure 1 Diagram of a generalized functional link network

Of course, the important questions to ask are:

- What types of non-linear functions lend themselves to overall good performance as part of a universally approximating FLN?
- How can such functions be efficiently generated?

The goal of this work is to address these questions and to present promising results obtained with a novel implementation of polynomial artificial neural networks. Herein, a new algorithm for implementing a single-layer polynomial network is introduced that can be rapidly generated and trained to approximate multi-dimensional non-linear functions of significant complexity. Novel applications of existing improvements to polynomial networks are also introduced which greatly increase the accuracy of the polynomial network such that comparison with other universal approximating (UA) algorithms is either comparable or superior for certain performance parameters.

1.2 Existing Polynomial Networks

1.2.1 Development of the Group Method of Data Handling (GMDH)

CI methods and other similar algorithms generally propose a single type or family of related functions as computational units to be replicated within particular network architecture. The replication of computational units lends advantages in both software and hardware implementation of such networks in terms of reuse, scalability, etc. If we think of tasks such as function approximation (interpolation), function prediction (extrapolation), etc., as being variants of curve fitting, one can see that the choice of computational unit will determine, to a large extent, the ability and efficiency of a network to properly represent non-linearities. Many choices exist in the literature for these computational units.

Ivakhnenko first proposed a formal implementation of a polynomial–based computational engine in 1971 [15]. The potential of polynomial-based networks to approximate highly complex nonlinear functions is apparent with the examination of the Taylor Series expansion in a single dimension (in this case) [16]:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^n(a)}{n!}(x-a)^n \tag{1}$$

$$= f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f^{(2)}(a)}{2!}(x-a)^2 + \cdots \tag{2}$$

Many familiar nonlinear functions are adequately approximated with the Taylor Series [16]:

$$\begin{aligned} e^x &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots \; for\ all\ x \\ \sqrt{1+x} &= 1 + \frac{1}{2}x - \frac{1}{8}x^2 + \frac{1}{16}x^3 - \frac{5}{128}x^4 + \cdots \; for|x| \le 1 \\ \sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots \; for\ all\ x \\ \cosh x &= 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots \; for\ all\ x \\ \tanh x &= x - \frac{1}{3}x^3 + \frac{2}{15}x^5 - \frac{17}{315}x^7 + \cdots \; for\ |x| < \frac{\pi}{2} \end{aligned} \tag{3}$$

For the one-dimensional cases in (3), all approximations are simply polynomials with monomial terms of unique degree, each with a scalar coefficient. This is the case for multi-dimensional non-linear function approximations as well.

The difficulty in using polynomials as basis functions for learning networks, recognized by Ivakhnenko and many others, lies in the questions of how to efficiently generate such polynomial product terms (monomials) of the correct exponential degree or order, as well as how to solve for the coefficients of these terms. The first problem to recognize is the potential impact of the sheer number of monomials on computer memory. To illustrate the memory problem, let us first consider a three-dimensional (three-input) case for which both a second and third order solution are assumed. For three variables, x, y, and z, the full second-order polynomial captures all monomial terms of degree-2 and lower, and is yielded by the expansion of $(x + y + z + 1)^2$. Similarly, the third-order expression is given by $(x + y + z + 1)^3$. Typically in code or in hardware, barring any special algorithm, all product terms are generated straightforwardly by applying the multiplications indicated by the exponent. Equation (4) explicitly shows the product operations generated by such a straightforward approach for the three-input, second-order case:

$$(x + y + z + 1)^2 = x^2 + xy + xz + x + xy + y^2 + yz + y + xz + yz + z^2 + z + x + y + z + 1, \textit{16 product terms} \tag{4}$$

After combining terms and rearranging, the three-input second-order polynomial becomes:

$$(x + y + z + 1)^2 = x^2 + y^2 + z^2 + 2xy + 2xz + 2yz + 2x + 2y + 2z + 1, \textit{10 unique terms} \tag{5}$$

Note that for the three-input, third-order case, the number of product operations begins to dramatically increase, yielding 64 such terms (not shown). Combining and rearranging to obtain only unique unrepeated terms, and omitting coefficients:

$$(x + y + z + 1)^3 \rightarrow x^3 + y^3 + z^3 + x^2y + x^2z + xy^2 + y^2z + xz^2 + yz^2 + xyz + x^2 + y^2 + z^2 + xy + xz + yz + x + y + z + 1, \textit{20 unique terms} \tag{6}$$

Without any reduction of duplicates, the total number of monomial product terms created by a simple multiplicative algorithm as depicted by (4) is given by:

$$(k+1)^o$$
$$where \quad \begin{array}{l} o \equiv maximum\ order\ of\ polynomial \\ k \equiv number\ of\ dimensions/inputs \end{array} \tag{7}$$

Table I illustrates how the number of resultant multiplicative terms can quickly overwhelm computer memory as both the maximum polynomial order and the number of input dimensions increase.

Table I
Total Number of Product Terms per Polynomial Order and Number of Inputs

k \ o	2	3	4	5	6	7	8
2	9	27	81	243	729	2187	6561
3	16	64	256	1024	4096	16384	65536
4	25	125	625	3125	15625	78125	4E+05
5	36	216	1296	7776	46656	3E+05	2E+06
6	49	343	2401	16807	1E+05	8E+05	6E+06
7	64	512	4096	32768	3E+05	2E+06	2E+07
8	81	729	6561	59049	5E+05	5E+06	4E+07

The number of unique terms in a complete polynomial of a particular max and lower order, such as that of (5) and (6), is given by:

$$\#\ of\ terms = \frac{(o+k)!}{o!\,k!} \tag{8}$$

For computational and storage efficiency, it is essential to avoid accumulation of unnecessary terms during the generation process. Table II shows the results of this expansion for multiple combinations of max order and number of inputs, but with duplicate terms removed.

Table II
Number of Monomial Terms per Dimensions and Inputs

k \ o	2	3	4	5	6	7	8
2	6	10	15	21	28	36	45
3	10	20	35	56	84	120	165
4	15	35	70	126	210	330	495
5	21	56	126	252	462	792	1287
6	28	84	210	462	924	1716	3003
7	36	120	330	792	1716	3432	6435
8	45	165	495	1287	3003	6435	12870

In recognition of the difficulty of the problem of the generation of polynomial terms of a particular maximum order, Ivakhnenko and colleagues developed the Group Method of Data Handling (GMDH) [15] which remains the foundational method of polynomial term generation among researchers to this day. The idea is to create a single node function, in the original case a two-input second-order polynomial, which when cascaded in a multi-layer architecture, produces permutations of monomial product terms which comprise *an estimation* of higher order terms necessary to approximate various functions. The single node function specified by Ivakhnenko is a second-order polynomial transform expressed as:

$$Y = A_2(X) = a_0 + a_1 x_1^2 + a_2 x_2^2 + a_3 x_1 + a_4 x_2 + a_5 \qquad (9)$$

Note that each node requires the solution of six coefficients. A GMDH representation of a polynomial network which would be necessary to approximate the three-input, third-order polynomial expressed in (6) is shown in Figure 2.

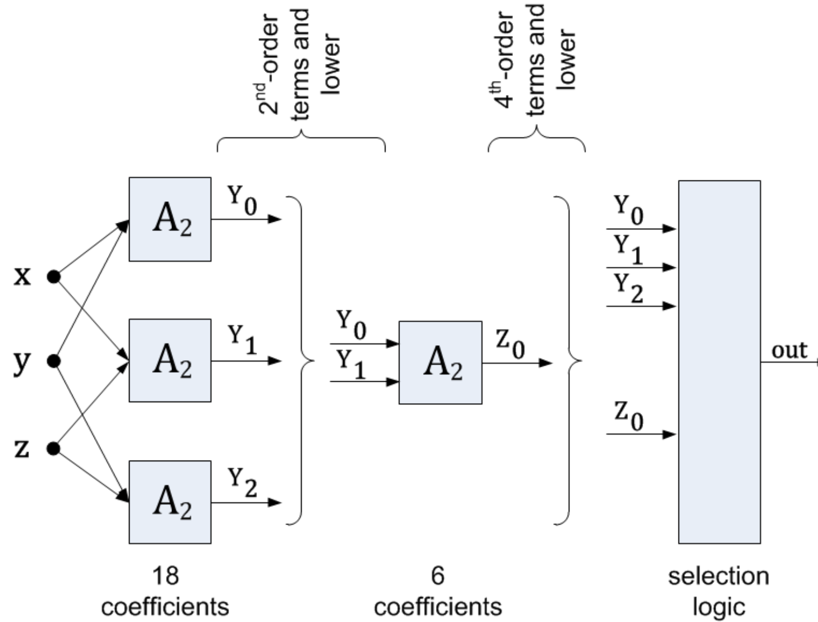


Figure 2 GMDH network for the solution of a 3-input, 3rd (or 4th) order function

Each A_2 node in Figure 2 must create the six input product terms and solve for the six associated coefficients, usually by linear regression, in order to produce an intermediate polynomial value (Y_0, Z_0, etc.). Figure 3 details the computational function of each GMDH node.

Though the GMDH network simplifies some of the required process by avoiding the explicit specification and generation of each polynomial term, several drawbacks become apparent. Network "training" necessary for solving for an optimized set of input and desired output relations, cannot be straightforward. Methods such as one-step linear regression, though used for each A_2 node, are not adequate for multi-layer feedforward networks. As will be discussed, due to the inherent multi-layered architecture, many complex algorithms are necessary to grow and prune GMDH networks, and to iteratively train optimal final sets of network coefficients in order to obtain reasonable results. Following the computation through the cascaded architecture, it becomes apparent that many redundant monomial terms are created through inner products, and many excess coefficients are created as well. Examining the intermediate polynomial terms of only the first layer of Figure 2 yields (term coefficients ignored):

$$Required\ Polynomial\ Terms:$$
$$x^3 + y^3 + z^3 + x^2y + x^2z + xy^2 + y^2z + xz^2 + yz^2 + xyz + x^2 + y^2 + z^2 + xy + xz + yz + x + y + z + C$$

$$\begin{gathered} Y_0 \rightarrow x^2 + y^2 + xy + x + y + c_0 \\ Y_1 \rightarrow x^2 + z^2 + xz + x + z + c_1, redundant\ terms:\ x^2, x, c_1 \\ Y_2 \rightarrow y^2 + z^2 + yz + y + z + c_2, redundant\ terms:\ y^2, z^2, y, z, c_2 \end{gathered} \tag{10}$$

After the processing of the first layer, eight non-unique terms are created, and the network has not yet produced the required third-order terms. Though the second layer of the GMDH network produces the necessary third-order terms, many excess or unnecessary terms are represented by the generated outputs:

$$\begin{gathered} Z_0 \rightarrow required\ terms\ generated:\ x^3, y^3, z^3, x^2y, x^2z, xy^2, y^2z, xz^2, \\ yz^2, xyz \\ number\ of\ excess\ terms:\ 23 \end{gathered} \tag{11}$$

The original three-input, third-order polynomial expression contains 20 terms and their coefficients. In contrast, the necessary GMDH solution, built with second-order, two-input nodes, ultimately requires 24 coefficients, and generates combined product terms which comprise the equivalent of 31 excess terms beyond those required in (6).

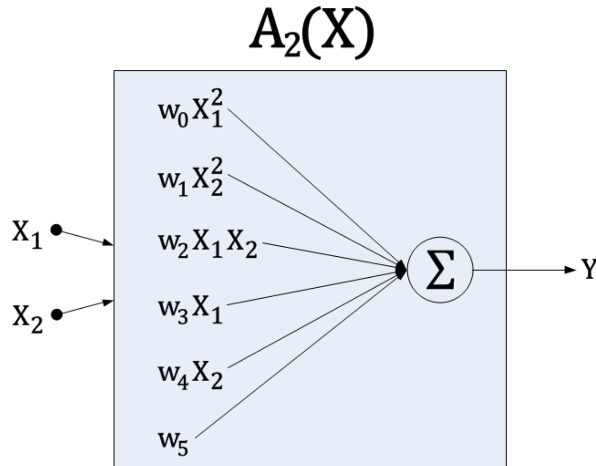


Figure 3 2-input, 2nd-order computational node of the original GMDH architecture

1.2.2 Evolution of the GMDH Model

Since the introduction of the GMDH method, various polynomial network implementations have competed successfully against the predominant perceptron-based feedforward networks. Stinchcombe and White proved that general nonlinear functions in the hidden layer, and not necessarily sigmoid functions, are adequate for the function of universal approximation [17]. Chen and Manry then verified that in fact, polynomial basis functions can be used to model multi-layer perceptron networks with good results as long as the degree of the model was sufficient to represent the training data [18]. They note most importantly that polynomial basis functions are more easily implemented in hardware than explicit sigmoid activation functions. That said, it should be noted that many hardware applications of neural networks utilize sufficient approximations of sigmoidal functions such as the Elliott function [19]. Though these improve the speed and ease of some computation, many implementations include both exponentials and division operations, potentially leading to solutions that are more complex than polynomial sum and product operations in some cases.

Almost all existing polynomial networks have been implemented with some variation on the GMDH model. Yang and Huang [20] added a third-order basis function to the GMDH model in addition to the second-order basis function. An elaborate algorithm was implemented, the Self-Organizing Polynomial Network (SOPN), which generates multiple cascading network layers which instantiate either the second or third-order node function, and which solves for intermediate values at each layer using standard Ordinary Least Squares (OLS) regression. Though OLS applied at each layer is computationally efficient compared to more elaborate training schemes, the confinement of the computation domain to within the boundary of each layer leads to arbitrary network construction which is potentially inefficient during validation operations.

Oh and Pedrycz [21] also add the option of third-order node activation functions, and similar to Yang and Huang, utilize OLS regression to develop intermediate solutions at each layer boundary. Their text goes further in analyzing

several drawbacks to GMDH. Chiefly, they note that pruning algorithms are mandatory to avoid networks which suffer from over-fitting to particular solutions. Additionally, they note that for deploying GMDH against low-dimensional datasets, the problem of over-fitted solutions becomes acute.

Jekabsons and Lavendels [22] confirm the same issues for GMDH applied to low-dimensional problems, and highlight the susceptibility of GMDH to local minima problems, similar to those found with use of ANNs. They produce data showing that for the second-order node function, problems of dimensionality less than or equal to four are likely to produce non-optimal results compared with other methods. They also point out the acute variance deficiencies of GMDH networks when deployed against noisy data sets, especially evident during k-fold testing methods. In a later publication [23], Jekabsons introduces the Adaptive Basis Function Construction (ABFC), a hill-climbing technique [24] which tries to grow a single-layer polynomial network by adding monomial terms one at a time. Per each new term, outputs for the entire network are computed via OLS regression and evaluated against desired output values by the Akaike Information Criterion (Corrected) (AICC) [25]. However, Jekabsons indicates several drawbacks to the method: Solution sets are susceptible to local minima and maxima errors. Due to the inherent and unknown correlation of monomials of a particular polynomial, the addition or extraction of single monomials at a time often results in under-damped or over-damped system responses as higher order terms are added. Such results do not present immediately with the ordered addition or extraction of terms. Jekabsons compensates with arbitrarily adding 2 or more terms at a time, but eventually confirms that this does not definitively overcome this issue. Computationally costly methods are employed, such as building several networks in parallel in response to the same training data, then averaging the final model based on AICC evaluations.

Nikolav and Iba developed yet another multi-layer GMDH variant which adds whole layers to the network, evaluates RMSE at the addition of each layer, then uses complex Genetic Algorithm (GA) techniques to prune apparently detrimental network connections [26]. While movement beyond standard linear regression techniques promises to more accurately groom the network for better

generalization performance during validation, repeated terms are still inferred in the network due to the cascaded GMDH model. Additionally, the computational complexity for high-dimensional data is prohibitive with GA methods since in general, all variants of the system must be fully computed before genetic selection can take effect.

Fuzzy Polynomial Neural Networks (FPNN) are a purported self-organizing neural network family theorized in earlier literature by Oh et al. [26–28]. This family uses a fuzzy front-end to gate node inputs in the first GMDH layer, thereby promising better function localization in the network performance. The original FPNNs required the designer to iteratively adjust certain fuzzy parameters, and to manually provide many of the GMDH network attributes. Still based on the GMDH model, an upgraded variant was offered in 2006 – the Genetically optimized FPNN (gFPNN) [30]. This variant uses GA techniques to select what is hoped to be an optimized GMDH network architecture. However, the resultant system architecture is extremely complex, and still requires offline specification of many design parameters (fuzzy membership functions, input assignments per node, total number of nodes retained after each generation, max width and depth of the GMDH network, etc.) based on analysis of the training data. The authors warn that the gFPNN is as yet prone to generalization errors if the proper design parameters are not specified. Furthermore, the final gFPNN system proposed is only tested with a single one-dimensional case (chaotic Mackey-Glass time series [31]) with only modest results compared to earlier FPNN variants, and is not compared to common benchmark problems alongside other well-known methods. Roh and Pedrycz revisit the gFPNN in a later work [32] whereby Information Granulation (IG) techniques [33] and C-means clustering [34] are used to automate some of the required analysis of the training data. An even more structurally complex system results that still requires tuning by the designer, though performs apparently well for three cases presented.

1.2.3 Divergence from the GMDH Model

Banfer and Nelles [35] depart from the GMDH model and propose a single-layer network structure which partitions the input and output space of a target

dataset into localized polynomial networks that are optimized to solve each resultant segment of the full solution. The outputs of each localized network are then superposed to afford the complete network solution. Figure 4 shows the diagram of their particular system:

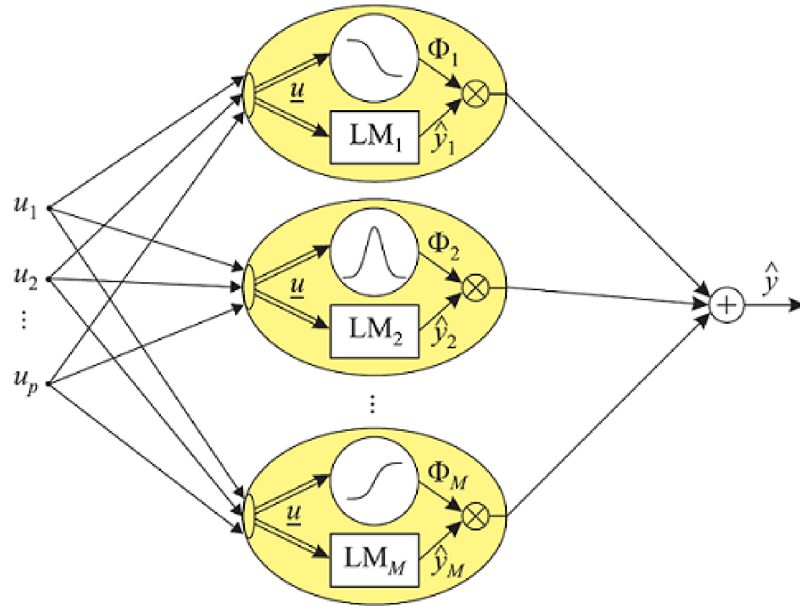


Figure 4 The Banfer and Nelles local model network [35]: The outputs $\hat{y}_i$ of the local polynomial networks (LM_i) are weighted with their associated domain function values (Φ_i) and superposed.

In this system, the outputs of localized polynomial transfer functions are essentially gated according to the current input domain by "validity functions" (Φ_i), and all resultant outputs $\hat{y}_i$ are finally added to yield a complete network solution. The authors assume that the "validity function" centers and standard deviations are given per the dataset under processing. This is akin to manually specifying membership functions in a fuzzy system, and therefore requires complex offline analysis of data, especially in multi-dimensional cases. Then per each local component, a stepwise technique is employed to grow that component's polynomial function by one increasing order at a time. An unspecified stepwise regression technique is used to compare output errors, and to subsequently add higher-order monomials to the current local network, or to split the local network into another which spans the same domain. The algorithm is tested on only one 9-input case, and comparisons are made only against one earlier variant proposed by the same

authors [36]. As previously noted by Jekabsons et al., Banfer and Nelles corroborate the inability to locate optimal solutions by evaluating the growth of polynomial functions by one term at a time. They acknowledge that local models can still generate excess terms which deteriorate the composite network's generalization ability.

1.2.4 Functional Link Networks – Complete Polynomials

For all CI methods, selection of the computational node functions is important, followed by the determination of network weights by "training", or by more direct computation if possible (such as by various forms of linear or non-linear regression). In the case of polynomial networks, the node functions are the individual monomial terms expressed as products of input variables and without coefficients. If one considers that for the realm of machine learning, such coefficients could be discovered via an appropriate training method, then it is easily seen that a FLN adapted with polynomial basis functions in the hidden layer achieves a simple polynomial network. This is viable, provided there is a way to generate all necessary polynomial terms in an efficient way. The coefficients of the composite polynomial expressed by the network become the trainable weights. Following from (5) and (6), all monomials are expressed from the degree of the maximum exponent downward to '1' as the constant term of degree-0. For the same inputs x, y, and z, the Complete Homogeneous set of Symmetric Polynomials (CHSP) [37], $h_k(x, y, z)$, consists of the unique products without coefficients of the expression, $(x + y + z)^k$, where k is the degree of the polynomial. For example, the full result for k = 2 would be:

$$h_2(x, y, z) = x^2 + y^2 + z^2 + xy + xz + yz \tag{12}$$

Note that the degree of every product term is 2. The function in (12) can be expressed formally as:

$$h_2(X_1, X_2, X_3) = \sum_{1 \leq j \leq k \leq 3} X_j X_k \tag{13}$$

Similarly, the third-order expression, h_3 , for a three-input CHSP would be expressed as:

$$h_3(X_1, X_2, X_3) = \sum_{1 \le j \le k \le l \le 3} X_j X_k X_l \tag{14}$$

The general form of (14) for any number of input variables and for any degree, *k*, is:

$$h_k(X_1, X_2, \dots, X_n) = \sum_{1 \le i_1 \le i_2 \le \cdots \le i_k \le n} X_{i_1} X_{i_2} \dots X_{i_k} \tag{15}$$

An identity for the zero-order CHSP is introduced:

$$h_0(X_1, X_2, \dots, X_n) = 1 \tag{16}$$

For present purposes, we would like to define the definitive set of all homogeneous symmetric polynomial terms using the concepts of (15) and (16) above. For this definitive set, the sum of all monomial terms of degree *k* and lower is expressed:

$$\begin{aligned} K_k = h_k(X_1, X_2, \dots, X_n) + h_{k-1}(X_1, X_2, \dots, X_n) + \cdots \\ + h_0(X_1, X_2, \dots, X_n), where\ k \ge 0 \end{aligned} \tag{17}$$

For variables *x*, *y*, and *z*, and for maximum polynomial degree of 2, the complete set of CHSPs of descending order can be defined as K_2:

$$\begin{aligned} K_2(x, y, z) &= h_2(x, y, z) + h_1(x, y, z) + h_0(x, y, z) \\ &= x^2 + y^2 + z^2 + xy + xz + yz + x + y + z + 1 \end{aligned} \tag{18}$$

For comparison, for a two-input case, and for maximum polynomial degree of 3:

$$\begin{aligned} K_3(x, y) &= h_3(x, y) + h_2(x, y) + h_1(x, y) + h_0(x, y) \\ &= x^3 + y^3 + x^2y + y^2x + x^2 + y^2 + xy + x + y + 1 \end{aligned} \tag{19}$$

Equation (18) introduces the function, K_2 , which expresses the sum of all monomial terms of degrees 2 and lower for the given input variables. Thus, for any degree of a polynomial, a function exists to construct all degree product terms of a polynomial for a max degree and lower. For clarification, equation (19) expresses the same function for two input variables of max degree 3. The network diagram of Figure 5 shows the polynomial neural network necessary to realize the 3-input, degree-2 function of (18).

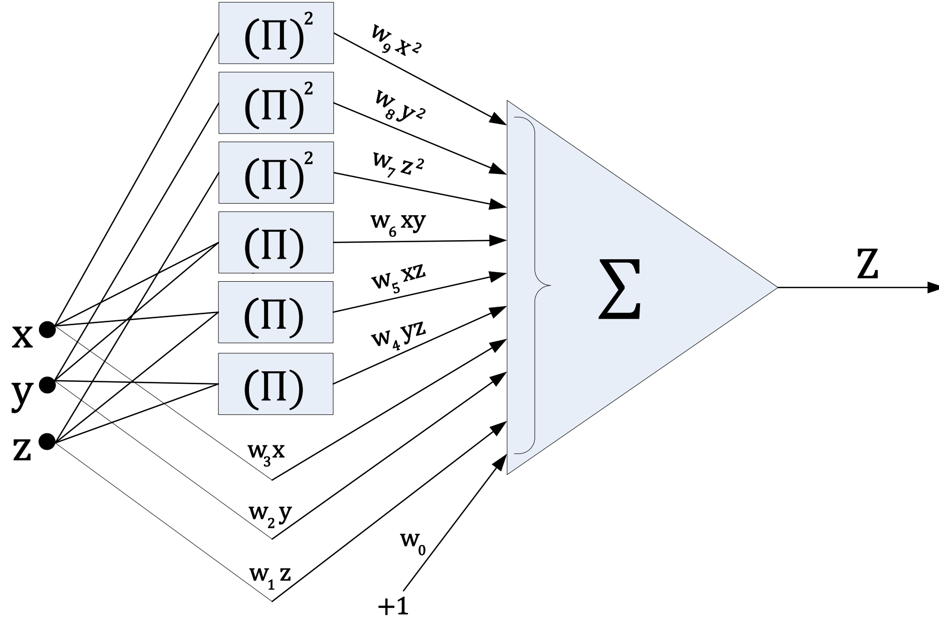


Figure 5 Polynomial network of degree 2 for 3 input variables

It is noted at this point that such a single-layer network can be trained in one matrix operation with techniques such as linear regression [14], where the following quantities of (20) and single ordinary least squares (OLS) matrix operation of (21) correspond to Figure 5:

$$\hat{X} = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \dots & \dots & \dots \\ x_{np} & y_{np} & z_{np} \end{bmatrix}, \quad \hat{P} = \begin{bmatrix} 1 & x_1 & y_1 & z_1 & x_1 y_1 & \dots & z_1^2 \\ 1 & x_2 & y_2 & z_2 & x_2 y_2 & \dots & z_2^2 \\ 1 & \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & x_{np} & y_{np} & z_{np} & x_{np} y_{np} & \dots & z_{np}^2 \end{bmatrix}$$

$$\hat{W} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \dots \\ w_N \end{bmatrix}, \quad \hat{Y} = \begin{bmatrix} o_1 \\ o_2 \\ \dots \\ o_{np} \end{bmatrix} \tag{20}$$

$$where\ np \equiv \#\ of\ patterns$$
$$N \equiv \#\ of\ monomial\ terms$$

$$\hat{W} = \left(\hat{P}^T \hat{P}\right)^{-1} \hat{P}^T \hat{Y} \tag{21}$$

1.2.5 Summary of the State of the Art for Polynomial Systems

At this point, it is noted that polynomial-based CI systems do appear in the literature, but are neither predominant nor significant in comparison with the use and performance of other methods. Key drawbacks exhibited by all such polynomial-based methods reviewed include:

- Deficient term generation and storage – No known systems make use of algorithms for direct generation of only the monomials necessary to produce polynomials of arbitrarily high order. Additionally, all systems studied compute and store all terms, including excess terms. This leads to computer memory limitations for problems of high dimensionality and/or large numbers of training patterns.
- Generalization error – Excess monomials are generated either directly or equivalently, leading to significant validation or testing bias that is not foreshadowed by performance during training.
- Network complexity – Methods which avoid explicit and economical generation of unique terms lead to complex network connectivity, which in turn necessitates complex training and pruning algorithms to improve generalization results.
- Training deficiencies – Some methods reviewed either rely too heavily on standard OLS regression, which does not necessarily optimize polynomial term coefficients for validation performance. Other methods are extremely complex in response to inefficient multi-layer network structure, and require initial data analysis and hands-on determination of run-time parameters.

The goals of this work aim to address these drawbacks, and to develop a family of polynomial-based learning systems which not only improve the state of the art, but which compete strongly with other popular CI methodologies.

Chapter 2
Implementation of Competing Methods

The PLM variants introduced later in this work were tested against several methods prominently in-use throughout CI. A basic introduction to each of these methods is included in this Chapter. The proposed PLM variants are ideal for operating offline upon highly non-linear multi-dimensional data, and in real-time, within highly non-linear applications requiring computational solutions within reliable run times. This quality is shared among several prominent CI algorithms which have been selected for competition against the PLMs including Artificial Neural Networks, Fuzzy Systems, and Radial Basis Function systems such as the Extreme Learning Machine variants, and Support Vector Regression with RBF kernels.

2.1 The Single-Layer Feed-Forward Neural Network (SLFN)

Artificial neural networks (ANN) are commonly applied to various problems in industrial fields, such as motor actuation and control [38], [39], fault detection and prediction [11], and robotics [40]–[42]. They are capable of providing models for highly nonlinear and noisy data that are difficult to process with classical parametric techniques. Three choices common with ANN design are network architecture (size and topology), activation (node) function, and training.

Though there are many different kinds of ANN architectures, Single Layer Perceptrons (SLP) are neural network SLFNs which are both simplest to train, and are historically the most thoroughly investigated in comparative studies with other CI methods. One practical reason for the prominence of SLPs in research is the lack of a well-known training algorithm which is both accurate and efficient, and can also train bridged architectures. Variants of the Levenberg-Marquardt (LM) [43]

algorithm (supplied in the MATLAB Neural Network Toolbox [44]) and its derivatives are widespread, however it cannot handle bridged networks. For these reasons, a SLP is chosen for comparisons in this study. An example of a SLP is pictured in Figure 6.

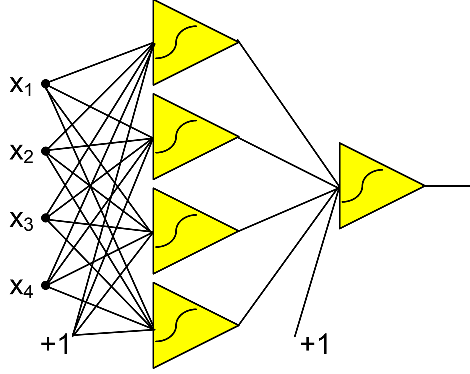


Figure 6 A 4-input, 5-neuron SLP with 4 neurons in the hidden node

The most traditional implementation of an artificial neuron as a computational node is with either a unipolar or bipolar sigmoidal "soft" activation function based on the hyperbolic *tanh* function. The unipolar case is as follows:

$$f(net) = \frac{\tanh(gain \times net) + 1}{2}$$

$$where: \; net = \sum_{i=0}^{K_n} w_{n_i} x_{n_i} \tag{22}$$

$$\begin{aligned} i &\equiv index\ of\ input\ connection\ per\ neuron \\ n &\equiv index\ of\ neuron \\ K_n &\equiv total\ \#\ of\ connections\ per\ neuron \end{aligned}$$

The training algorithm used in the comparative studies herein is the Neuron-By-Neuron (NBN) algorithm, an LM derivative developed by Wilamowski et al. [45]–[47]. This training algorithm has been shown to possess many advantages over other known LM variants, including efficient matrix computation, reduced memory usage, and superior convergence accuracy [46]. NBN employs an enhanced second-order gradient method to optimize network solutions. Regression is employed similar to the PLMs, however instead of directly solving for network weights, an

updating factor is iteratively computed for an initially randomly generated set of network weights. All LM variants, including NBN, update network weights according to the following equation [48]:

$$\Delta \boldsymbol{w} = (\boldsymbol{J}^{\boldsymbol{T}}\boldsymbol{J} + \mu \boldsymbol{I})^{-1}\boldsymbol{J}^{\boldsymbol{T}}\boldsymbol{e} \tag{23}$$

where $\boldsymbol{w}$ is the weight vector, $\boldsymbol{I}$ is the identity matrix, and μ is the combination coefficient. The Jacobian matrix, $\boldsymbol{J}$, with dimensions $(P \times M) \times N$, and the error vector, $\boldsymbol{e}$, with dimensions $(P \times M) \times 1$, are defined as:

$$\boldsymbol{J} = \begin{bmatrix} \frac{\partial e_{11}}{\partial w_1} & \frac{\partial e_{11}}{\partial w_2} & \cdots & \frac{\partial e_{11}}{\partial w_N} \\ \frac{\partial e_{12}}{\partial w_1} & \frac{\partial e_{12}}{\partial w_2} & \cdots & \frac{\partial e_{12}}{\partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{1M}}{\partial w_1} & \frac{\partial e_{1M}}{\partial w_2} & \cdots & \frac{\partial e_{1M}}{\partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{P1}}{\partial w_1} & \frac{\partial e_{P1}}{\partial w_2} & \cdots & \frac{\partial e_{P1}}{\partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{PM}}{\partial w_1} & \frac{\partial e_{PM}}{\partial w_2} & \cdots & \frac{\partial e_{PM}}{\partial w_N} \end{bmatrix} \quad \boldsymbol{e} = \begin{bmatrix} e_{11} \\ e_{12} \\ \cdots \\ e_{1M} \\ \cdots \\ e_{P1} \\ \cdots \\ e_{PM} \end{bmatrix} \tag{24}$$

where $\boldsymbol{P}$ is the number of training patterns, $\boldsymbol{M}$ is the number of outputs, and $\boldsymbol{N}$ is the number of weighted connections. Elements in the error vector, $\boldsymbol{e}$, are calculated by:

$$e_{pm} = d_{pm} - o_{pm} \tag{25}$$

where d_{pm} and o_{pm} are the desired and actual output respectively, at network output m, when training pattern p.

2.1.1 Competitive Considerations of Neural Networks

Neural networks, along with other node-based computational systems, are generally capable of finding better solutions than other methods. However, the performance of neural networks relies heavily on the training algorithm involved. Because LM derivatives, and earlier first-order training methods such as Error Back-

Propagation (EBP) and its derivatives [49]–[51], all rely on randomized production of network weights at various stages, ANNs are particularly susceptible to converging to local minima and maxima as first or second order gradients are computed. Also, due to the flat tail of the *tanh*() function extending out to +/- infinity, many computer and hardware learning implementations have trouble when function inputs that are too far out on the margins result in stagnant error computations that encounter the digital quantization limit of the host processor. This is known as the flat-spot problem [52]. Thus, multiple iterative trials are necessary to generate a family of solutions, which must then be searched for the best solution. An example of this can be seen in Figure 7. In summary, ANNs offer good probability of convergence to some solution, but never guarantee convergence to an optimal solution.

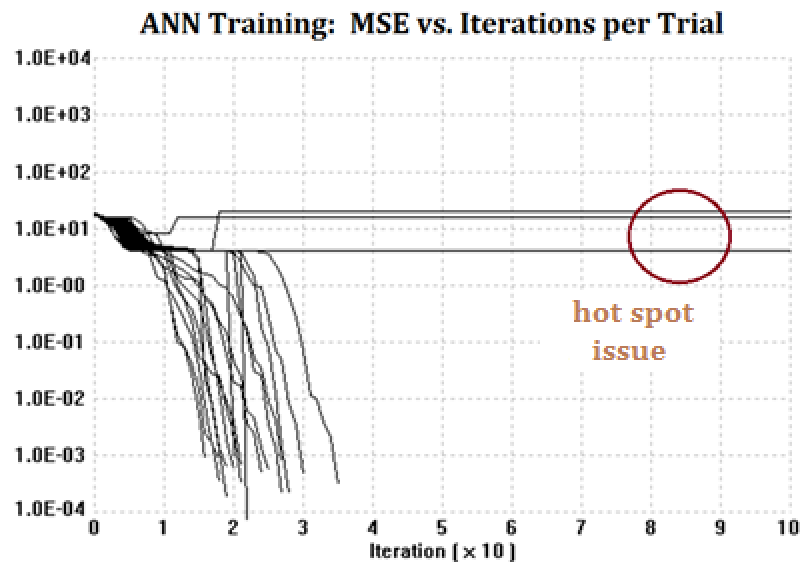


Figure 7 Typical Iterative ANN Training: "hot spot" issue

2.2 The Takagi-Sugeno-Kang Fuzzy System

Fuzzy systems are at least as popular as neural networks for solving similar types of approximation and prediction problems. Traditionally, fuzzy systems do not require training with sample data. However, particularly in those cases they require hands-on front-end development by the designer in order to approach

acceptable solutions [53]. Two foundational designs predominate for fuzzy systems – The Mamdani inference [54], and the Takagi-Sugeno-Kang (TSK) inference [55]. For the comparative studies in this work, the TSK approach has been selected for its architectural simplicity and since it tends to yield more accurate output mapping than the Mamdani model for the same problems with equal input resolution [56].

In applying fuzzy systems to dataset problems, each input dimension must be evaluated in order to properly select the resolution and/or precise location of sample points, which are then translated to in-situ Look-Up Tables (LUT) for real-time deployment. Care must be taken for designs that intend to handle high dimensionality, since the fuzzy output table storage increases by $O(n^D)$, where D is the number of input dimensions. The development of the input data translators, called membership functions, is often done by the designer without explicit or complete data available. For data of high dimensionality, the design and resolution of the membership functions can become highly subjective [57]. One such example of fuzzification of an input temperature variable converted to a scaled subjective value is seen in Figure 8. The underlying basis for the output value is certainly numeric and amounts to transformation of input quantities to a sum of weighted neighboring factors included in the membership function. The specific type of membership function pictured in Figure 8 is the trapezoidal. The slope of the sides of the trapezoid is maps directly to weighting factors which will be applied to interpolate an input value that lies between stored breakpoints on the input axis. Other membership functions include Gaussian and triangular. Triangular membership functions have been shown to give better output resolution than other methods for the same input resolution. For this reason, triangular functions will be used in this study. A comparison of different membership function outputs for the same function at the same output resolution is seen in Figure 9.

Following the fuzzy design process, the resulting architecture is sufficiently simple, essentially an input-to-output LUT, such that fuzzy systems are often preferred over other methods for hardware implementation. The validation times for real-time fuzzy systems are extremely reliable, further lending value to use of these systems in clock-based designs, since equivalent computations are done for

each input-to-output operation. It should be noted that the validation times for fuzzy systems are not necessarily minimal compared to other methods. Propagation delay in hardware systems merely becomes an address spin, followed by multiplicative interpolation of a finite number of values extracted from the fuzzy table LUT.

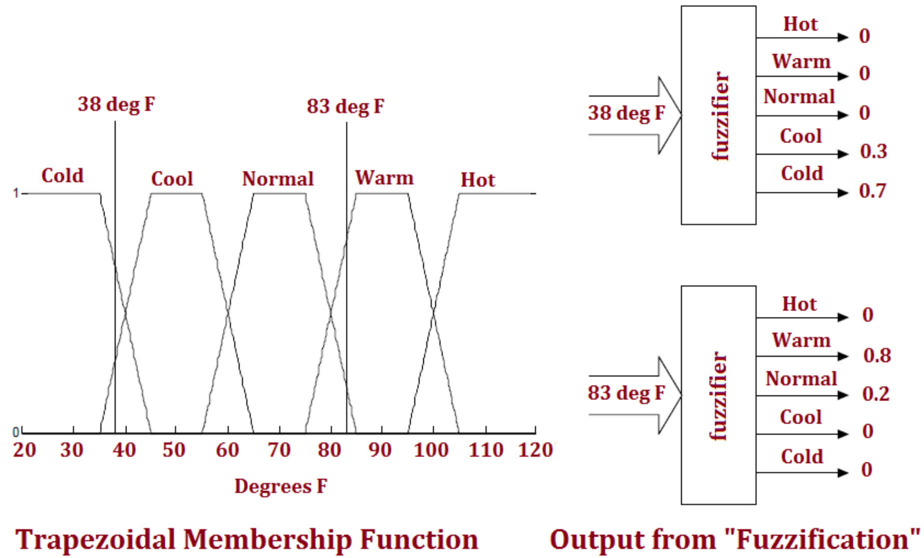


Figure 8 Example of a Trapezoidal Fuzzy Membership Function Operation

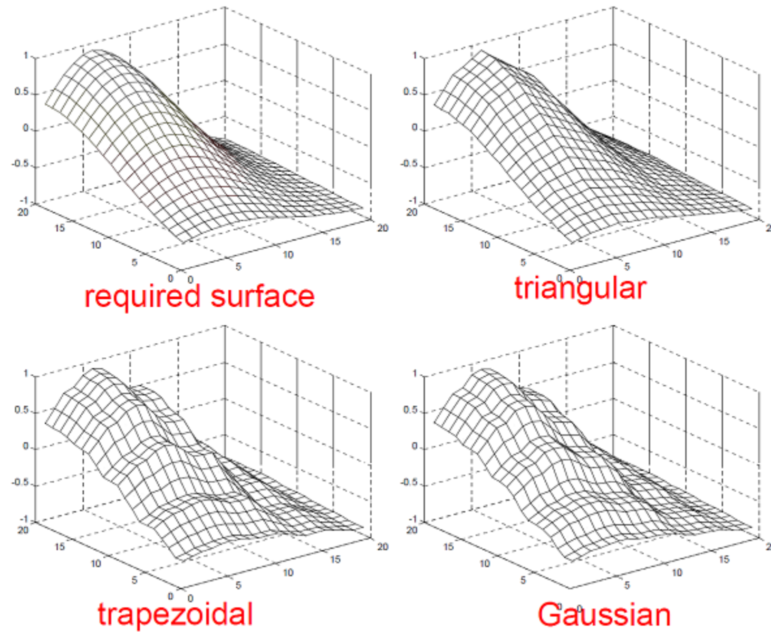


Figure 9 TSK Product Encoding Output: Comparison of membership functions

In general, node-based computation systems such as PLMs, ANNs, etc., provide smoother, more accurate non-linear mapping of functions than fuzzy systems [58]. Though output accuracy can be increased by increasing the resolution and of breakpoints in the membership functions, this could lead to stability problems known as "hunting" [59] in feedback applications of fuzzy systems. An example of a TSK system output to that of an ANN for the same problem is shown in Figure 10.

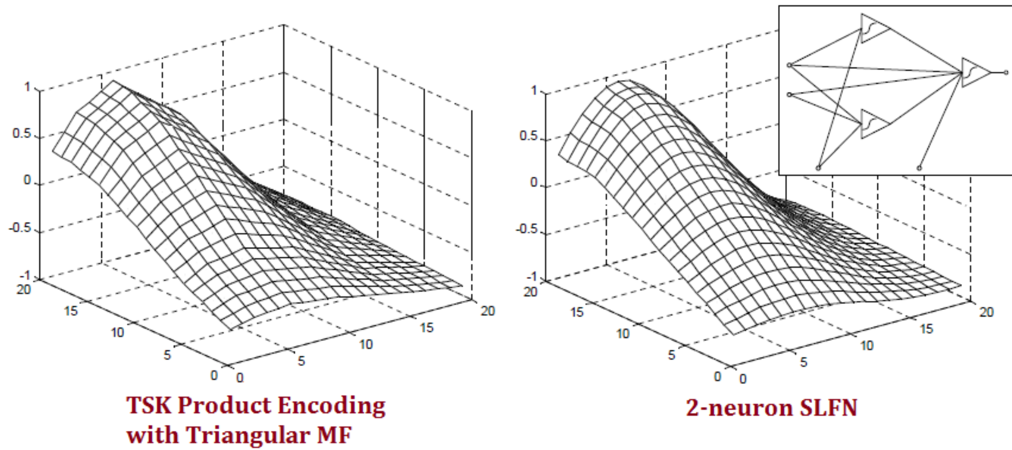


Figure 10 Comparison of TSK and ANN (node-based) Output Quality

2.2.1 Development of a Novel Data-Driven N-Dimensional TSK Fuzzy System

For proper comparison with PLMs and with the other algorithms studied, an N-dimensional TSK fuzzy system was developed. Instead of manual design of membership functions, a recursive algorithm was developed to compute regularly-spaced interpolation values for the fuzzy output table, given any random distribution of n-dimensional training data over the input range. Since all dataset inputs and outputs are normalized (see Section 5.1 Test Methodology) in keeping with current testing standards in the field of CI, the job of automatically generating optimized interpolated values is made easier. Following generation of the fuzzy output table, the TSK product method is coded to forward-compute n-dimensional output values from stored output tables.

2.2.1.1 Fast Recursive N-dimensional Interpolation

An original, efficient recursive algorithm was developed which accepts randomized multi-dimensional data points as input, as well as a set of target interpolation points (regularized or arbitrary), and returns the target points populated with optimized interpolation values. The algorithm is used for the purpose of generating the required fuzzy output table; however its uses could be extended to any process which requires regularization of multidimensional data, or merely interpolation at specific arbitrary points. The MATLAB code for this algorithm is included in Section 7.8.1 of Appendix H.

A summary of the operation of this process follows.

Starting with the complete set of training data as input, and with a complete set of arbitrary interpolation target points:

1. Initialization Stage: Per each target interpolation point vector, the algorithm begins in interior-point mode. A 1-D span is set which will be used along each dimension at a time. This strategy is aided as all input vector values are standardized, that is, they are scaled to the range [-1:1]. Stage 2 is called recursively. If Stage 2 returns with less than the minimum candidate points, the 1-D search span is increased. If the 1-D search span exceeds a maximum value, the fore-aft requirement cannot be met; the target point is on an edge, so the radius is reset to a minimum value and the mode becomes boundary point mode, and Stage 2 is called recursively. If Stage 2 returns with candidate points, Stage 3 is called and the current target vector receives an output interpolation value. If there are more target vectors, Stage 2 is called with that vector. If not, algorithm returns the complete set of interpolated target points.
2. Candidate Search Stage: Given the full training data point set and a single interpolation target coordinate, the algorithm searches along one dimension at a time. In the interior point mode, the algorithm searches along one dimension for at least one point fore, and one aft of the target point. If this condition cannot be met after searching, the output subset is

set to NULL as a signal to Stage 1, and the level returns to Stage 1. In the boundary point mode, the fore-aft requirement is not enforced. If the last dimension has not yet been processed, and if the subset of candidate points is non-empty, the search dimension is incremented, and the subset, current mode, current radius, and current target point is sent recursively to another call to Stage 2. If the last dimension has been searched, the final subset of candidate points is returned.

3. Value Interpolation: For a valid set of candidate points, the output interpolation value is computed without a square root operation according to the following sequence and returned to Stage 1:

Given: $\hat{I} = (x_1, x_2, \ldots x_D) \equiv target\ vector$

$$\hat{Y} = \begin{bmatrix} y_{11} & y_{21} & \cdots & y_{D1} \\ y_{12} & y_{22} & \cdots & y_{D2} \\ \cdots & \cdots & \cdots & \cdots \\ y_{1N} & y_{2N} & \cdots & y_{DN} \end{bmatrix} \equiv subset\ coordinates$$

$$\hat{Z} = \begin{bmatrix} z_1 \\ z_2 \\ \cdots \\ z_N \end{bmatrix} \equiv subset\ values \qquad (26)$$

$$where\ D \equiv \#\ input\ dimensions$$
$$N \equiv \#\ candidate\ vectors$$

$$\hat{R} = \begin{bmatrix} \sum_{i=1}^{D} \left(\hat{I}_i - \hat{Y}_{i1}\right)^2 \\ \sum_{i=1}^{D} \left(\hat{I}_i - \hat{Y}_{i2}\right)^2 \\ \cdots \\ \sum_{i=1}^{D} \left(\hat{I}_i - \hat{Y}_{iN}\right)^2 \end{bmatrix} \equiv row\ sum\ squared\ values \qquad (27)$$

$$SSQ = \sum_{i=1}^{N} \hat{R}_i \equiv sum\ of\ all\ elements\ of\ \hat{R} \qquad (28)$$

$$Output = \frac{\hat{R}^T \hat{Z}}{SSQ \times (N-1)} \equiv interpolated\ value \qquad (29)$$

To demonstrate the effectiveness of the interpolator, the highly non-linear MATLAB peaks() function is used to generate 1000 randomly distributed points over a 3-D space. The proposed function is used to interpolate regularized (in this case) target points for potential population of an output fuzzy table. The randomized points and 25 (5 breakpoints along each dimension) interpolated point locations are seen in Figure 11. Figure 12 demonstrates a comparison as the original function is interpolated with increasing resolution as the interpolated point density increases from 5 to 20 breakpoints along each dimension.

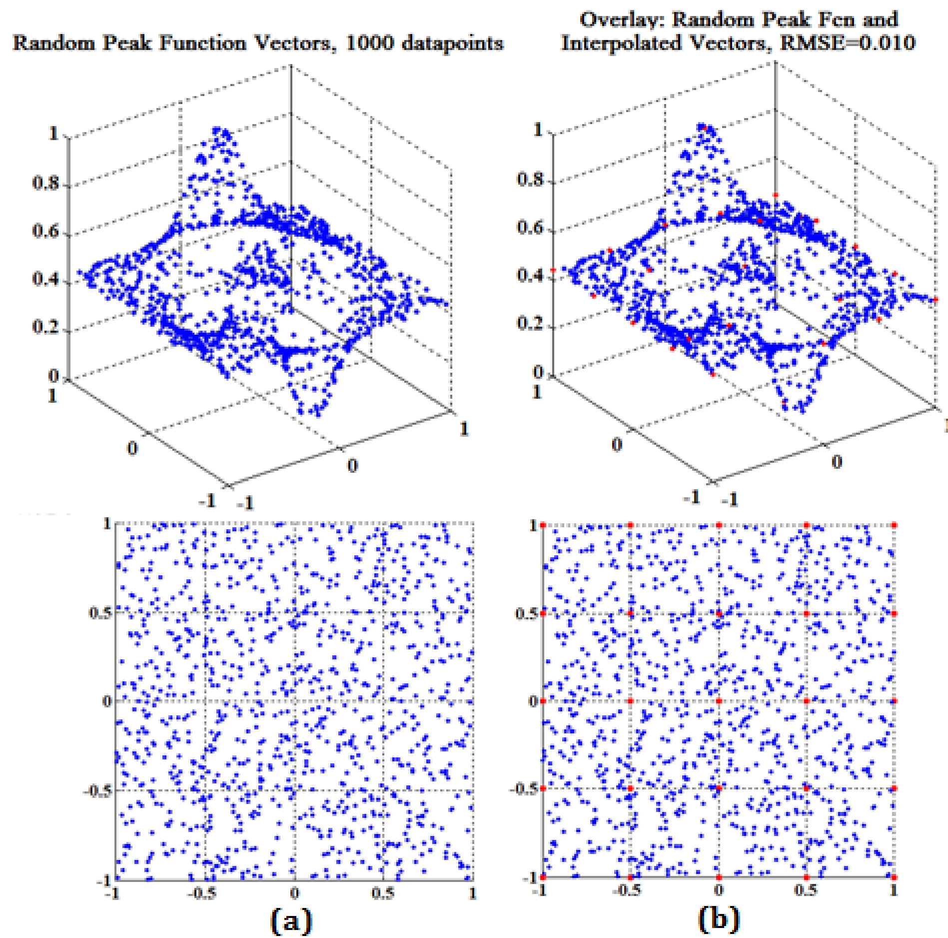


Figure 11 (a) View of random generated peaks() values, (b) Overlay with interpolated values

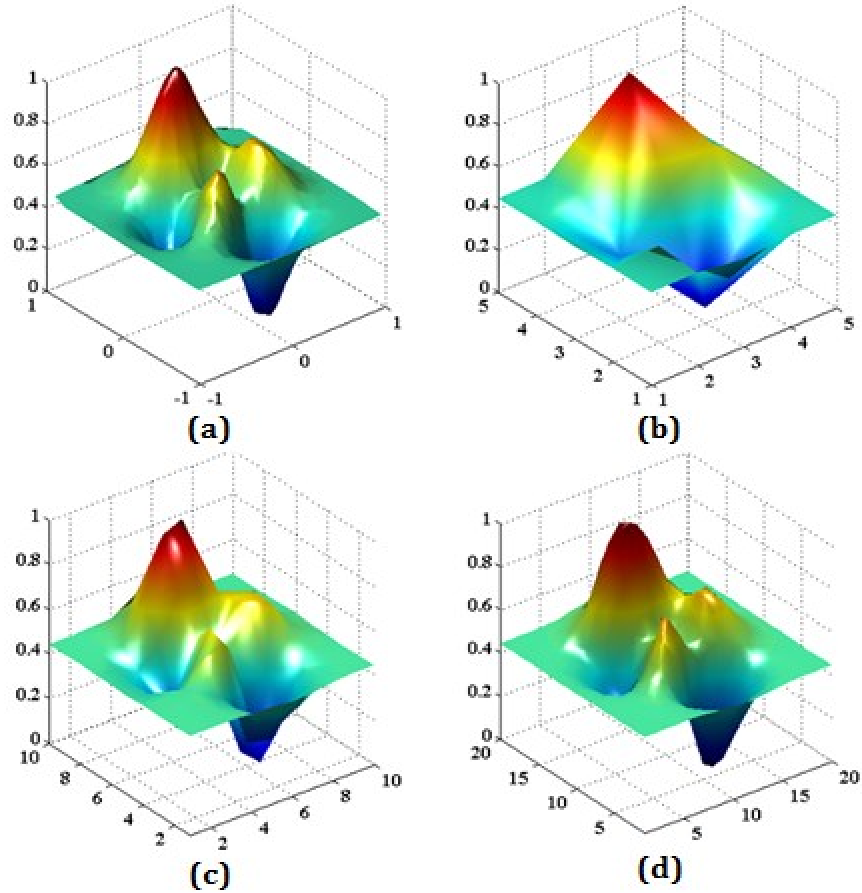


Figure 12 (a) Original function, 1000 data points, (b) 5x5 grid, (c) 10x10 grid, (d) 20x20 grid

2.2.1.2 Completion of an N-dimensional TSK Fuzzy Engine

The TSK fuzzy system implemented for this study proceeds with forward-computed product encoding, which addresses points to output within the fuzzy output table created from the interpolation software, and which builds the required weighted product terms using the common formula:

$$Output = \frac{\sum_{k=1}^{2^N} w_k z_k}{\sum_{k=1}^{2^N} w_k} \tag{30}$$

$$\begin{aligned} &where\ N \equiv \#\ membership\ functions \\ &w \equiv the\ inverse - proportional\ weighting\ factors \\ &\quad computedfor\ each\ value \\ &z \equiv fuzzy\ table\ output\ values \end{aligned}$$

For all cases, the denominator of (30) reduces to 1 and may be ignored. The weighting factors, $\boldsymbol{w}_k$, are all inverse-proportional in distance from the target point to the fore and aft fuzzy table terms along each dimension, and sum to 1 in each case.

Original software was created to complete the implementation of an N-dimensional fuzzy engine for comparison studies. The system receives input training datasets and maximum tolerance values, and subsequently uses the methods described previously to generate TSK solutions of increasingly higher resolution by specifying an equal number of regularly distributed fuzzy table breakpoints along each dimension. Each solution evaluates training and validation error and runtime results, and reports these values to other coordinating software. The associated MATLAB code for the TSK fuzzy engine can be viewed in Section 7.8.2 of Appendix H.

2.2.2 Radial Basis Function Learning Machines

Currently in the literature, much attention is garnered by SLFNs constructed with Radial Basis Function (RBF) nodes in the hidden layer. Architecturally, RBF systems resemble both PLMs and ANNs. However, just as training paradigms identify ANNs from other learning machines, specific RBF methods diverge from other CI methods largely but not solely based on computational differences in the training phase. The most prominent RBF-based algorithms at the time of this writing will be compared in this study.

2.2.2.1 Review of RBF Networks

Figure 13 represents a typical RBF network structure. In reference to the Figure:

- p is the index from 1 to P, where P is the total number of input patterns.
- H is the total number of hidden-layer RBF nodes.
- d is the index from 1 to D, where D is the number of dimensions in the input.
- Input patterns, x, of dimension, D, are described as $x_p = [x_{p,1}, x_{p,2}, \dots, x_{p,D}]$.

- β_h are the multiplicative weighting factors per RBF node output.
- o_p are the network outputs per input pattern.

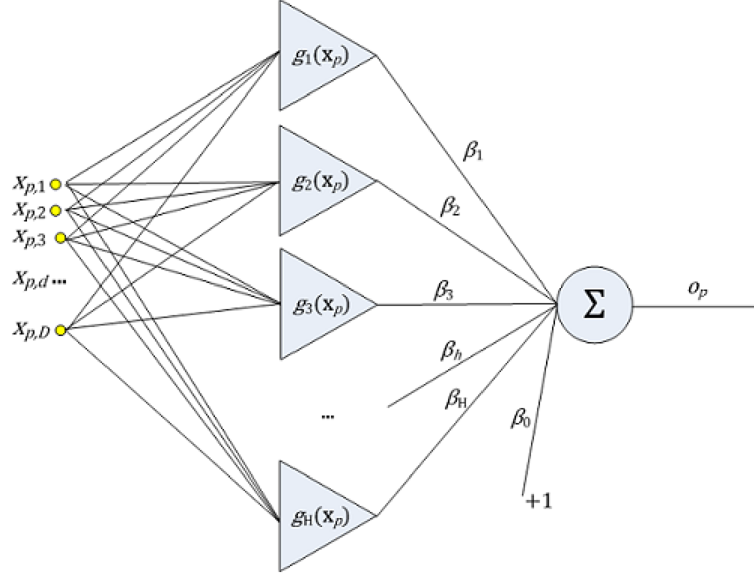


Figure 13 A typical RBF network containing H neurons and D inputs

A RBF SLFN, like the PLM, is comprised of a single hidden layer of H computational nodes, an output summing node, and D input ports. Each of the H nodes contains a kernel function, $g_n(x)$, which again similar to the PLM, are parametrically nonidentical to the kernel functions of other nodes in the same network. For all RBF variants in this work, the node kernel functions will be defined as Gaussian functions of the form:

$$g_n = \exp\left(-\frac{\left|\left|x_p - c_n\right|\right|^2}{\sigma_n^2}\right) \tag{31}$$

$$\begin{aligned} where:\ c_n &\equiv center\ of\ the\ n^{th}\ RBF\ unit \\ \sigma_n &\equiv width\ of\ the\ n^{th}\ RBF\ unit \\ \left|\left|\cdot\right|\right| &\equiv the\ Euclidean\ Norm\ computation \end{aligned}$$

The output summing function per input pattern then becomes:

$$o_p = f(x_p) = \sum_{n=1}^{H} \beta_n g_n(x_p) \tag{32}$$

From (31) and (32), it is seen that per a particular RBF network architecture, three parameters can be adjusted for the purpose of RBF network optimization. The three parameters are the weights, centers, and radii given by β_n, c_n, and σ_n, respectively. Any training method (LM, regression, etc.) can be chosen in any combination for this optimization task.

2.2.2.2 The Extreme Learning Machine RBF Variants

Huang et al. originally developed the Extreme Learning Machine (ELM) algorithm in [60]. The authors propose a RBF system which seeks to optimize network weights and biases after first randomizing the initial sets of those quantities. Final output weights are then solved for using a matrix pseudo-inversion technique. ELM was expanded by Huang et al. into an incrementally constructive algorithm (I-ELM) in [61]. Following this, two additional algorithms were introduced by the same authors in further attempts to improve upon I-ELM. The Convex Incremental Extreme Learning Machine (CI-ELM) and Enhanced random-search-based Incremental Extreme Learning Machine (EI-ELM) are introduced in [62] and [63] respectively. All of these algorithms are deployed in comparative publications using Gaussian RBF artificial neurons as the kernel function as in (31).

For all ELM variants, new nodes are added incrementally with randomly generated centers and radii. CI-ELM uses an equation to minimize in-process training error by adjusting all of the output weights in the existing network each time a new neuron is added. In the case of EI-ELM, a parameter, k, is introduced which specifies the number of new randomized nodes added for each training epoch. As a final sequence of each such epoch, errors are computed for each of the k added neurons, and the one with minimum resultant error is selected and added to the network. Following the indices established in Section 2.2.2.1, the general algorithm for the I-ELM is shown below as a basis for all the ELM variants. The algorithm is extracted directly from Huang et al. [61]:

Given a training set $\{(x_p, y_p) \mid x_p \in \Re^D,\ y_p \in \Re, p = [1..P]\}$, an activation function *g(x)* (31), a maximum node number *H*, and an expected learning accuracy ε:

1. **Initialize:** Let the number of nodes, *n=0*, and residual error, *E = y*.
2. **Learning:** While(*n < H*) and (RMSE > *ε*)
   a. *n*++
   b. Assign random center c_n and a width σ_n within an acceptable range for the new hidden node.
   c. Based on the random activation function and the error, calculate the output weight β_n for the node:
   $$\beta_n = \frac{\sum_{p=1}^{P} e_p g_n(x_p)}{\sum_{p=1}^{P} g_n(x_p)^2} \tag{33}$$
   d. Calculate the residual error after adding the new hidden node:
   $$E = E - \beta_n * g_n(x) \tag{34}$$
   where x and E are the vectors containing all of the input patterns and errors for each pattern respectively

   End while loop
3. Output is calculated using equation (32).

This algorithm will yield a network with a single hidden layer of RBF nodes connected with weighting terms, β, to a summing output node. It was proven by Huang et al. that this network is ideal as a universal approximator. The algorithm allows for fast training times and in the case of I-ELM, there is only one calculation to make per iteration. The computation of β in matrix form can be done efficiently in most environments. However, the drawback to all ELM variants is that only one of the three RBF neuron parameters is optimized. The ELM variants have been previously implemented in MATLAB code by this research group and documented elsewhere.

2.2.2.3 Support Vector Regression with RBF Kernels

Support Vector Regression (SVR) is a complex method of machine learning that retains selected training patterns for in-situ processing in addition to

incorporating a variety of network architectures and computational kernels. The theory was initially introduced by Vapnik [64], and most mature implementations are informed by Smola and Scholkopf [65]. SVR systems with RBF kernels have been cited in many recent comparative works, including those of Huang et al., and are therefore included among the methods studied in this work. When implemented correctly, they achieve excellent training and validation error compared to other methods. Though accuracy is often superior to most other methodologies, SVR requires offline optimization of runtime parameters by the designer. Essentially, a superior result is always possible but not guaranteed. SVR is described extensively in the included references, and is therefore not detailed in this work.

The SVR with RBF kernel networks tested in this study were implemented with the widely-used LIBSVM C/C++ code library by Chang and Lin [66], and were compiled for the Microsoft Windows 7 operating system. For consistency, the SVR libraries were executed within a MATLAB wrapper environment, also widely-used, and developed by Weston et al. as part of the Spider machine learning environment [67].

Chapter 3

Computational Strategies to Improve Polynomial Network Performance

This Chapter features new methods applied to a single-layer model of a polynomial network. Several strategies have been studied, and theory and implementation of each relevant strategy will be described herein. In the course of research, many methods were explored. Only those methods which appear to provide significant advantage, in light of the drawbacks to current schemes highlighted at the end of the previous Chapter, will be discussed. One method was extensively explored but did not make the final cut for affording effective improvement. The results of that study can be examined in Appendix A – Exploration of Chebychev Transform Methods.

3.1 Efficient Generation of Monomial Polynomial Terms

For the proposed polynomial networks, it is first and foremost desirable to generate the individual monomial terms in such a way as to avoid unnecessary scalar multipliers and repeated terms. We also want to be able to adapt the general case for theoretically unlimited multi-dimensional input data and monomial term order. An analogous observation is pictured for the 3-input, 3rd-order case of Figure 14. In this case, the necessary 3rd and lower-order terms are unique if only the upper diagonal of the initial 2-D multiplication matrix are used, followed by the terms encompassed by the downward-sliding diagonal for the multiplication of the initial 2-D plane by each dimension. The same principle can be extended for additional dimensions/inputs, and for higher orders. The best algorithm will generate the intended terms directly without extra or repeated terms.

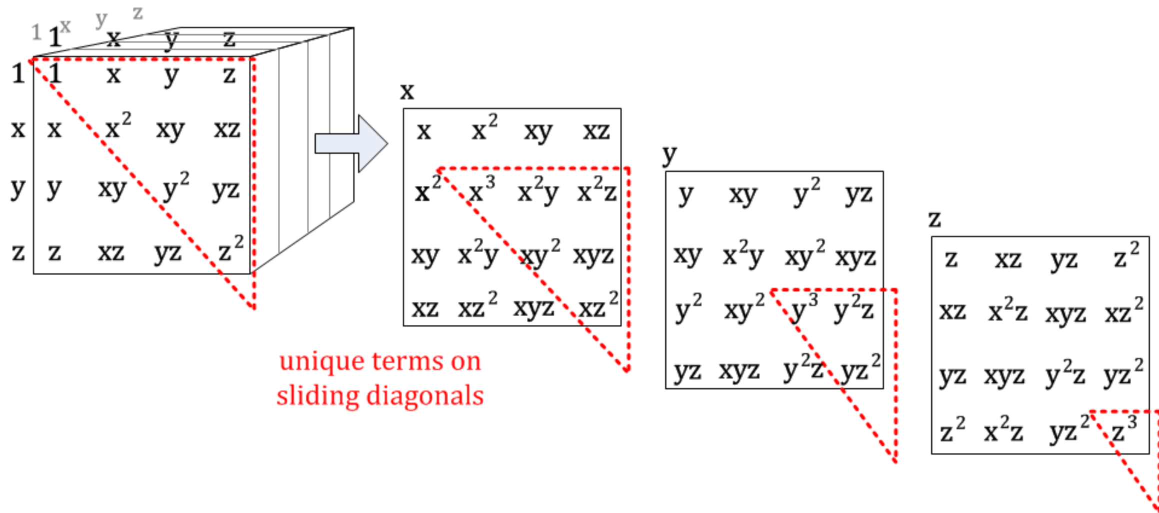


Figure 14 3-input 3rd-order case: Unique monomial terms on sliding diagonal

As pictured in Figure 14, we can conceive of arranging the 1st order terms of each input variable into an array, $\hat{A}$, with a leading 1 as first term, such as $\hat{A}$ = [1,*x*,*y*,*z*]. The indices for the elements of $\hat{A}$ from left to right would be (1, 2, 3, 4). An algorithm is introduced in Table III which, per given input array and max polynomial order, generates sets of indices representing the position of elements of $\hat{A}$ as product terms comprising all monomial terms of the intended polynomial.

Table III
The Poly-Gen Algorithm for Generation of Unique Monomial Terms

```
indices = POLY-GEN(Â,maxord)
    1   nd ← length of Â
    2   Nmax ← Ø
    3   for I ← 1 to maxord
    4       Nmax ← {Nmax, nd}
    5   indices = Find-Idx(Nmax)            // recursion
    6   return indices

indices = FIND-IDX(Nmax)
    1   len ← length of Nmax
    2   if len = 1
    3       then indices = [1:Nmax(0)]^T
    4       return indices
    5   idx ← FIND-IDX(Nmax[1: len – 1)])   // recursion
    6   [rows, cols] ← size of idx
    7   indices ← Ø
    8   for row ← 1 to rows
```

```
9         for j ← idx[row,cols] to len
10            new ← {idx[row,1:cols], j}
11            indices ← append new to last row of indices
12  return indices
```

Two advantages to this algorithm are:

- Per each max order, no terms are repeated.
- No products are yet formed. Only the indices of the ordered input variables are generated and stored for later operation.

Table IV displays an example of recursive polynomial factor index generation for a 3-input, max-order-2 case where the indices correspond to the array: [1,*x*,*y*,*z*]. The MATLAB code which generates the polynomial term indices is included in 7.2 Appendix B – MATLAB Code: Unique Polynomial Term Generation.

Table IV
Example of the Recursive Poly-Gen Function for a 3-input, order-2 Case

Phase I
(advance across)

1			**→ 2**			**→ 1**		
Nmax	len	idx	Nmax	len	idx	Nmax	len	idx
[4 4]	2	--	[4]	1	$\begin{bmatrix}1\\2\\3\\4\end{bmatrix}$	[4 4]	2	$\begin{bmatrix}1\\2\\3\\4\end{bmatrix}$

Phase II
(advance down)

new	**indices**	**new**	**indices**	**new**	**indices**
idx_row=1	**j=1**	**idx_row=2**	**j=2**	**idx_row=3**	**j=3**
[1 1]	[1 1]	[2 2]	$\begin{bmatrix}1 & 1\\1 & 2\\1 & 3\\1 & 4\\2 & 2\end{bmatrix}$	[3 3]	$\begin{bmatrix}1 & 1\\1 & 2\\1 & 3\\1 & 4\\2 & 2\\2 & 3\\2 & 4\\3 & 3\end{bmatrix}$

idx_row=1 **j=2**

$[1 \quad 2]$ $\begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$

idx_row=1 **j=3**

$[1 \quad 3]$ $\begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{bmatrix}$

idx_row=1 **j=4**

$[1 \quad 4]$ $\begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \end{bmatrix}$

idx_row=2 **j=3**

$[2 \quad 3]$ $\begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 2 & 2 \\ 2 & 3 \end{bmatrix}$

idx_row=2 **j=4**

$[2 \quad 4]$ $\begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 2 & 2 \\ 2 & 3 \\ 2 & 4 \end{bmatrix}$

idx_row=3 **j=4**

$[3 \quad 4]$ $\begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 2 & 2 \\ 2 & 3 \\ 2 & 4 \\ 3 & 3 \\ 3 & 4 \end{bmatrix}$

idx_row=4 **j=4**

$[4 \quad 4]$ $\begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 2 & 2 \\ 2 & 3 \\ 2 & 4 \\ 3 & 3 \\ 3 & 4 \\ 4 & 4 \end{bmatrix}$

terms

1
x
y
z
$\mathbf{x^2}$
xy
xz
$\mathbf{y^2}$
yz
$\mathbf{z^2}$

3.2 Statistical Smoothing and Pruning of Monomial Coefficients

As previously discussed, most methods in the literature which attempt to improve network performance, apart from enhanced regression methods, focus on heuristic approaches which rely on direct relationships either between the training patterns and output training error computations, or between the inclusion or exclusion of polynomial terms and similar output error computations. In all such cases studied, the evaluation of a particular monomial term is done following from and independently of some regression computation. In this section, an iterative strategy is introduced which uses incremental statistical computations to both stabilize monomial coefficients in the face of noisy and or spurious training data, and to eliminate one or more monomial terms based on the variation of their coefficients, borne out during iterative network training techniques.

3.2.1 A Training Strategy to Stabilize and Isolate Noise-Responsive Coefficients

In most real-world applications, noise is present in the training data and must be dealt with to achieve reasonable network performance, regardless of the method. Additionally, systems which optimize to training data usually perform significantly worse in response to new data during validation. This is commonly known as generalization error [68]. For polynomial-based and other node-based supervised learning methods using standard linear (or other) regression techniques, training data are often evaluated and streamlined via clustering, filtering, or other schemes as part of an additional front-end process. Then the surviving training vectors are put through to the regression technique at hand. Instead of using front-end schemes which attempt to deal with the training data directly, the strategy discussed herein iteratively runs randomly-selected subsets of the training data through the regression engine, and computes increasingly stable mean coefficient values. At the same time, incremental standard deviation (STD) computations are performed which are later used to flag noise-responsive coefficients for removal from the final network.

Following once again from the OLS regression equation of (21), it is noted that the full set of initial training vectors, $\hat{X}$, can be iteratively broken into randomly selected subsets, and fed through the regression process (OLS in this case), yielding multiple corresponding polynomial equations, $\hat{P}_k$, and solution weight sets, $\hat{W}_k$:

$$\widehat{W}_0 = \left(\hat{P}_0{}^T\hat{P}_0\right)^{-1}\hat{P}_0{}^T\hat{Y},\ \ \widehat{W}_1 = \left(\hat{P}_1{}^T\hat{P}_1\right)^{-1}\hat{P}_1{}^T\hat{Y},\ \dots,\ \ \widehat{W}_K = \left(\hat{P}_K{}^T\hat{P}_K\right)^{-1}\hat{P}_K{}^T\hat{Y} \quad (35)$$
$$where K \equiv the\ final\ iteration\ step$$

It is expected that during the training process, $\hat{W}_k$ will vary during successive regression computations due to the alternate absence or presence of critical patterns in the training subsets, as well as due to noise among what would ideally be equivalent training patterns. A running mean can be numerically computed for each weight as processing continues, which essentially promises to approach a more stable average value for that particular weight:

$$\hat{\hat{W}}_K = \begin{bmatrix} (w_{0_1} + w_{0_2} + \cdots + w_{0_K})/K \\ (w_{1_1} + w_{1_2} + \cdots + w_{1_K})/K \\ \cdots \\ (w_{N_1} + w_{N_2} + \cdots + w_{N_K})/K \end{bmatrix} = \begin{bmatrix} \overline{w}_0 \\ \overline{w}_1 \\ \cdots \\ \overline{w}_N \end{bmatrix} \tag{36}$$

Simultaneous to the weight stabilization, a set of more involved computations can be employed to track the average magnitude of the variation of each coefficient from iteration to iteration. At the completion of such an iterative training process, this criterion can be used to make decisions about which monomials may be over-responsive to both noise in critical inputs and more generally, to the presence of erroneous training patterns. The following section will detail the computation steps but in general, we seek an expression to evaluate the magnitude of the perturbation of the network monomial weights in response to iterative processing of random subsets of the training data. A typical standard deviation expression for such a process is defined:

$$STDEV_{\hat{W}_K} = \begin{bmatrix} \sigma_{w_{0_K}} \\ \sigma_{w_{1_K}} \\ \cdots \\ \sigma_{w_{N_K}} \end{bmatrix} \tag{37}$$

Once these values are known per monomial weight, computational decisions can be made to excise either the "noisiest" monomial term, or "noisiest" terms above a designated threshold, based on the relative magnitude of these values. It is proposed that such a method, focusing on the response of network terms to training data, is potentially more efficient and accurate than the many complex and arbitrary filtering and clustering methods proposed as front-end processors of the training data themselves.

3.2.2 Forward-Computed Statistical Methods – Overview

This section formalizes the statistical computation included in some of the polynomial-based algorithm variants in this study. These numerical methods are designed to forward-compute current key values per each training iteration, thereby

reducing memory and processing overhead possible with the large matrices involved. The associated computations discussed herein are used for:

- smoothing monomial weights towards more stable average values
- flagging "noisy" or otherwise susceptible monomial terms
- tracking stopping criteria for the iterative processes involved

The essential MATLAB code implementation of all such statistical processes is included in 7.3 Appendix C – MATLAB Code: Statistical Processing of Monomial Term

Weights. A summary of the computational steps follows in Table V:

Table V
Computational Steps for Statistical Processing of Monomial Term Weights

			Purpose		
Computational Step (iterative)	**Code Variable**	**Uses Equation**	**Stabilize Values**	**Evaluate Term Noise**	**Stopping Criteria**
Collect/compute monomial weights	wwtemp	(OLS, etc.)	X	X	X
Compute running means of weights	wwmeans	(38)	X	X	X
Compute normalized weights per current maximum absolute	wwscaled	(39)		X	X
Compute running means of normalized weights	wwscmeans	(40)		X	X
Compute sum of iteration Δ ∀ wwscmeans (1st order)	scmeansn_1	(41)		X	X
Compute iteration Δ ∀ scmeansnn_1 (2nd order)	scmeanslope	(42)			X
Compute running stdev of means of normalized weights	wwscstds	(43)		X	
Option 1: STOP, remove "noisiest" term, REPEAT					
Option 2: CONTINUE, compute threshold criteria, remove terms, REPEAT					
Computational Step (post-iterative)					
Compute mean of final wwscstds	meanstd	(44)		X	
Compute stdev of final wwscstds	stdstd	(45)		X	
Compute threshold of term removal	maxscstd	(46)		X	

3.2.3 Forward-Computed Statistical Methods – Iterative Numerical Detail

For the iterative training process described, wherein randomly selected subsets of the training data are successively processed, the following computations are performed. For all equations in this section, subscripts for an individual member of $\hat{W}$, such as w_0, will be omitted as in, w, to simplify readability.

Initial values for monomial term weights, $\hat{W}_0$, are computed as in (35) per the particular regression method associated with the current polynomial network variant in use. A straightforward running mean is computed per monomial term weight. Such formulae can be found in numerous sources, such as in Press et al.'s Numerical Recipes series [69]:

$$\bar{w}_k = \frac{(k-1)\bar{w}_{k-1} + w_k}{k}, \qquad where\text{: } w \in \widehat{W} \quad k \equiv current\ iteration \quad \bar{w} \equiv mean\ value\ of\ weight \tag{38}$$

The maximum absolute value of each weight is updated at each iteration and is used to forward-compute a normalized version of that weight. Herein, a "dot" will be used to designate a normalized value. Thus, normalized weights become:

$$\dot{w}_k = \frac{w_k}{maxabs(w)}, \quad and\ \ \widehat{\dot{W}}_k \equiv complete\ set\ of\ current\ weights \tag{39}$$

The running means of the normalized weights are now computed as in (38):

$$\bar{\dot{w}} = \frac{(k-1)\bar{\dot{w}}_{k-1} + \dot{w}_k}{k} \tag{40}$$

The sum of the absolute values of the slopes, $\bar{\dot{w}}_k - \bar{\dot{w}}_{k-1}$, is forward computed as an incremental measure of the quiescence of the entire set of normalized means in the course of the iteration process. This is similar to a 1st-order gradient:

$$S_k = \sum_{i=1}^{N} \left| \bar{\dot{w}}_{i_k} - \bar{\dot{w}}_{i_{k-1}} \right|, \qquad where\ N \equiv \#\ of\ weights \tag{41}$$

The tracking of the sequential change in the running scaled means is further smoothed by averaging, similar to (38) and (40). The effect of this is similar to a 2nd-order gradient:

$$\bar{S}_k = \frac{\bar{S}_{k-1} + S_k}{k} \tag{42}$$

The current value of $\bar{S}_k$ is evaluated against an arbitrary maximum value set at run time, and in this way, (41) and (42) are used solely as stopping criteria for the iterative training algorithm.

The plot of Figure 15 shows how the mean values of term coefficients stabilize as regression iterations proceed successively on randomized subsets of the training data. Each subset contains approximately 75% of the original training set. Each data point marks the average of the absolute value of the sum of the change of the coefficient means from one iteration to the next as computed by (42):

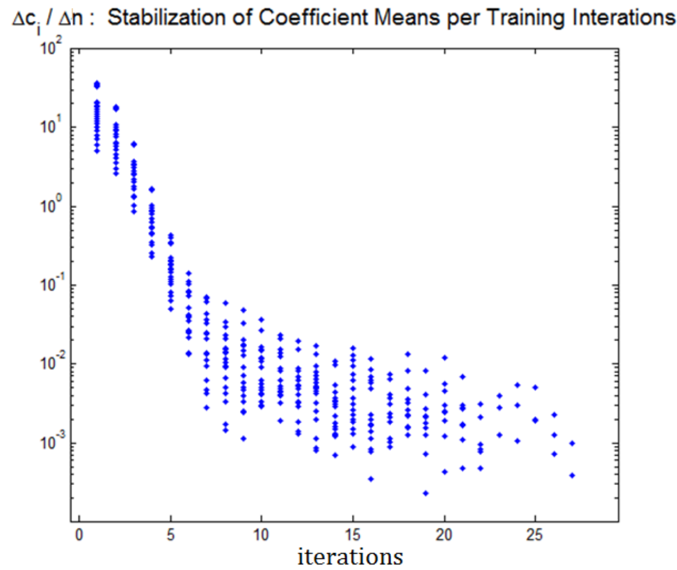


Figure 15 Stabilization of final normalized coefficient means as training proceeds

For this particular case, a threshold arbitrarily set at 10^{-2} would cause the algorithm to stop at 25 iterations or fewer for all cases plotted.

Computation continues to measure the relatively stability of monomial weights in a current network. As mentioned thus far, running means have been normalized earlier in the process. This is necessary to avoid a biased numerical

decision that would tend to penalize a larger magnitude optimized monomial term coefficient versus a smaller optimized term coefficient. Per monomial term, the running standard deviation of the normalized means of the iterative weight values is computed following a numerical formula adapted from an original algorithm by Donald Knuth [70], but taking advantage of quantities that have already been normalized, thus insuring a uniform range of output values in the range [0:1] for all monomial weights. For an individual weight in $\hat{W}$, the formula is:

$$STD_{w_k} = \sigma_{\bar{w}_k} = \sqrt{\left(\frac{\sigma^2_{\bar{w}_{k-1}}}{k} + (\bar{w}_k - \bar{w}_{k-1})^2\right) \times (k-1)} \tag{43}$$

The formula has been tested alongside the non-incremental MATLAB std() function, and it is accurate to less than 1×10^{-14} absolute error on a 64-bit processor running 64-bit MATLAB. The output of the function in (43) was observed as in Figure 16. In the plot shown, each column of data points represents normalized standard deviations computed for the full set of monomial term weights active during a particular training epoch.

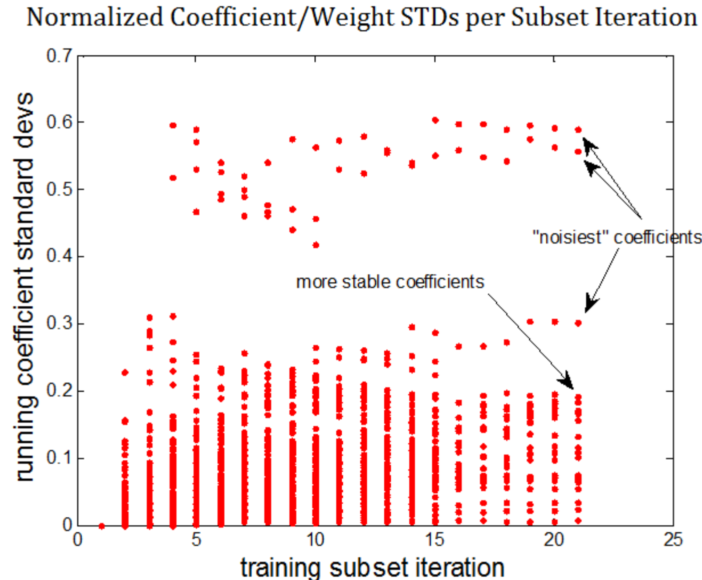


Figure 16 Tracking monomial term coefficient variation over successive training iterations

As training subset iterations proceed, one can see that values are indeed normalized, and that "noisy" terms can easily be identified at the end of a training epoch.

At this point, two options arise in the statistical processing:

- Option 1 – The most noise-responsive monomial term can be extracted from the set, and training can continue with remaining terms applied to additional epochs.
- Option 2 – Threshold criteria can be computed at the end of an iterative epoch, and multiple monomial terms can be excised, after which remaining terms continue through to the next epoch.

In the case of Option 2, two more computations remain. Once the iterative subset training completes, a simple mean of all final STDs computed by (43) can be computed:

$$\overline{STD}_{\widehat{W}_K} = \bar{\sigma}_{\bar{\widehat{W}}_K} = (\sigma_{\bar{\widehat{w}}_{0_K}} + \sigma_{\bar{\widehat{w}}_{1_K}} + \cdots + \sigma_{\bar{\widehat{w}}_{N_K}})/N \\ where\ N \equiv \#\ of\ monomial\ terms \tag{44}$$

Additionally, a simple (non-iterative) STD of all elements of STD_{WK} can be computed:

$$STDSTD_{\widehat{W}_K} = \sigma_{\sigma_{\bar{\widehat{W}}_K}} = STD\left(\sigma_{\bar{\widehat{w}}_{0_K}}, \sigma_{\bar{\widehat{w}}_{1_k}}, \dots, \sigma_{\bar{\widehat{w}}_{N_K}}\right) \tag{45}$$

A parameter, *stdscale*, can be arbitrarily set at run-time, and with the quantities computed in (44) and (45), monomial terms can be evaluated by their respective STD_{wK} from (43), relative to those of other terms as depicted in Figure 16. The final threshold criterion for inclusion/exclusion is expressed as:

$$maxSTD = \overline{STD}_{\widehat{W}_K} + \left(stdscale \times STDSTD_{\widehat{W}_K}\right) \\ evaluate\!:\ STD_{w_K} > maxSTD \tag{46}$$

3.3 Exploration of Enhanced Regression Techniques

As previously mentioned, the predominant method used in the literature to solve single-layer polynomial networks, or single-layer segments of those networks, is standard OLS, as depicted in (21). Several enhanced regression techniques were

explored, and two in particular were found to display significant improvement in network accuracy under certain conditions. Generalized Least Squares (GLS) and Ridge Regression (RR) are discussed in this section. Finally, hybrid implementations of both techniques were explored, and successful variations are discussed as well.

3.3.1 An Iterative Generalized Least Squares Regression Technique

In the world of supervised learning techniques in CI, a primary goal is finding and employing training methods which circumvent "bad data" or more specifically, noisy data and extreme "outliers" in the training data. Bad data are a significant problem especially for single-layer polynomial networks solved via OLS, since polynomial networks are already extremely susceptible to over-fitting solutions. As mentioned previously, most existing methods attempt to pre-process the training data with unsupervised culling techniques such as clustering, hidden-Markov chain analysis [44] [45], or other filtering techniques. Herein, an iterative regression technique using the Generalized Least Squares (GLS) method is implemented.

In the field of statistics, the Gauss-Markov theorem [73] states that the Best Linear Unbiased Estimator (BLUE) of the coefficients of a linear regression model is given by the OLS estimator but only under certain conditions. Applied to machine learning, "best" indicates that the variance between training and validation output results is minimized. The conditions are that errors among the different input dimensions must have expectation zero (zero bias about a true mean), must be uncorrelated from dimension to dimension, and must be "homoscedastic" (equal variance among each input dimension) [74]. Matrix algebra and geometric proofs of the theorem are offered by Borghers [75] and Ruud [76] respectively.

Consider at this point, the real-world dataset, "Abalone", available from the UC Irvine Machine Learning Repository [77]. This dataset attempts to predict the age of abalone from physical measurements. The equal variance restriction for using OLS as a BLUE applies to the errors present in the different input dimensions or "attributes". Two such attributes for the Abalone dataset are "viscera weight after bleeding", given in grams, and sex of the animal, tabulated as "male, female, or

infant [indeterminate]". It is highly unlikely that errors acquired in the collection and tabulation of those separate attributes have equal variance, though those errors may indeed be uncorrelated. Additionally, the condition that the expectation of attribute errors be zero (perfectly uniform around the correct mean value) is also unlikely for such varied attributes tabulated in real-world conditions. Hence, there is reason to search for regression techniques other than OLS which promise better results for multi-dimensional real-world datasets.

GLS regression, discovered by Aitken [78] and described in detail by Kuan [79], promises several key advantages including handling of correlated errors, and handling of "heteroscedastic" data; Errors along different dimensions of training data can have different variance and non-zero expectation (non-uniform error bias). Keeping in mind the OLS regression equation of (21), the basic GLS matrix equation is introduced:

$$\widehat{W}_{GLS} = \left(\hat{P}^T \widehat{\Omega}_{GLS} \hat{P}\right)^{-1} \hat{P}^T \widehat{\Omega}_{GLS} \hat{Y}$$
$$where\ \widehat{\Omega}_{GLS} \equiv covariance\ matrix\ of\ output\ errors \tag{47}$$

Kuan notes that for Ω_{GLS}, "The covariance structure of the errors must [usually] be known up to a multiplicative constant." However, Kuan purports a two-step method: Compute initial error components in Ω using OLS, compute the initial Feasible Generalized Least Squares (FGLS) estimator, $\hat{W}_{FGLS0}$, and then use an iterative method to apply GLS and successively recompute errors to update Ω. Following Kuan's procedure:

Compute initial errors:

$$\widehat{W}_{OLS} = \left(\hat{P}^T \hat{P}\right)^{-1} \hat{P}^T \hat{Y}\ , \qquad \hat{\mu}_{OLS} = \hat{Y} - \hat{P}\widehat{W}_{OLS} \tag{48}$$

Construct initial OLS covariance matrix:

$$\widehat{\Omega}_{OLS} = diag\left(\mu^2_{OLS_1}, \mu^2_{OLS_2}, \dots, \mu^2_{OLS_{np}}\right),$$
$$where\ np \equiv \#\ of\ training\ patterns \tag{49}$$

Compute initial FGLS estimator:

$$\widehat{W}_{FGLS_0} = \left(\hat{P}^T \widehat{\Omega}_{OLS} \hat{P}\right)^{-1} \hat{P}^T \widehat{\Omega}_{OLS} \hat{Y} \tag{50}$$

Compute errors, next Ω_{GLS}, and REPEAT:

$$\hat{\mu}_{GLS_0} = \hat{Y} - \hat{P}\widehat{W}_{FGLS_0}, \;\; \widehat{\Omega}_{GLS_0} = diag\left(\mu^2_{GLS\,0}, \mu^2_{GLS_1}, \dots, \mu^2_{GLS_{np}}\right) \tag{51}$$
$$\widehat{W}_{FGLS_1} = \left(\hat{P}^T \widehat{\Omega}_{GLS_0} \hat{P}\right)^{-1} \hat{P}^T \widehat{\Omega}_{GLS_0} \hat{Y}\,, \qquad etc.$$

The MSE can be computed from errors at each iteration and can be evaluated against previously computed MSE to serve as a stopping criterion as the algorithm approaches convergence:

$$MSE_{GLS_i} = \frac{\hat{\mu}_{GLS_i} \widehat{\Omega}_{GLS_i} \hat{\mu}^T_{GLS_i}}{np} \tag{52}$$

The MATLAB code used to directly implement the iterative GLS procedure is available in 7.4 Appendix D – MATLAB Code: Iterative GLS Regression.

The effect of the iterative GLS regression technique on a 1-D, 3rd-order example is dramatic compared to standard OLS regression performance. Following from the Gauss-Markov constraints on optimum OLS performance, an example of "bad data" which would handily violate those constraints would be marked by:

- a large percentage of outliers relative to the ideal function
- all points would exhibit at least some random noise
- some data points exhibit non-zero expectation (non-uniform bias)
- some data points exhibit uncorrelated variance (bimodal trend present)

Figure 17 shows an example of such a data distribution, with 40% of the data points laying in a trend far outside the original function.

The comparative results of iterative GLS regression versus standard OLS regression are stark, and are plotted in Figure 18. Two-hundred points are generated for the ideal function, noise is added with a normal distribution, then 40% of the data points are randomly selected and subjected to a second uncorrelated, biased trend. The "realbest' line in Figure 18 represents the best GLS solution evaluated by comparing the transform MSE of (52) against the original uncorrupted function data. The "'BEST' GLS" line represents the best GLS solution found after the transform MSE is evaluated against an arbitrary stopping threshold. Certainly, GLS regression promises higher accuracy for real-world, noisy data. The

MATLAB code producing these results for the 1-D test case is also available in 7.4 Appendix D – MATLAB Code: Iterative GLS Regression.

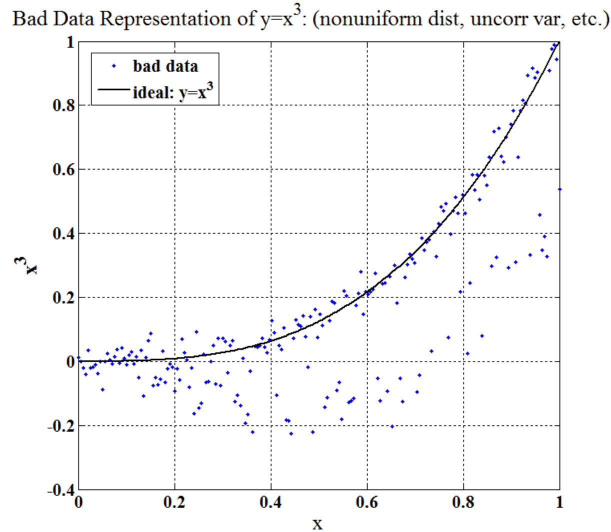


Figure 17 1-D, 3rd-order case of bad data, 40% outlaying trend

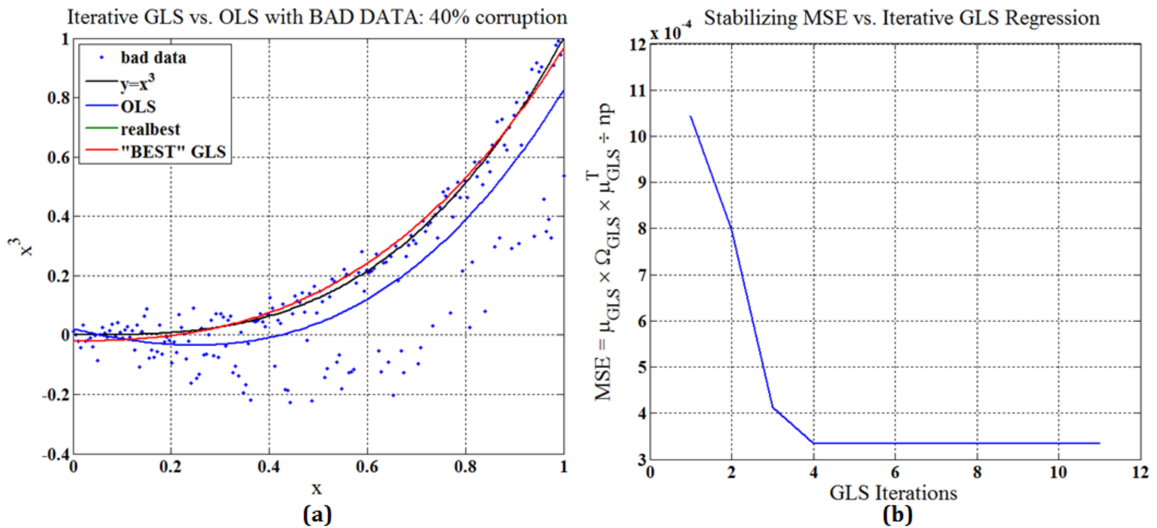


Figure 18 Iterative GLS performance vs. OLS: (a) GLS (red) beats OLS (blue), (b) Stabilization of GLS MSE of the transform errors

3.3.2 An Iterative Ridge Regression Technique

With polynomial-based networks, the presence of excess monomials of relatively high order leads to over-fitting or over-constraint of outputs in the

validation phase. This issue, discussed at length by Moody [68] and many others, is known across CI methodologies as generalization error. This is illustrated for a one-dimensional case in Figure 19, wherein a finite set of data points is fit by increasingly higher order polynomials. Though all polynomial solutions pictured from approximately 7th-order upward approximate the particular points rather well, one observes that the higher order solutions will not translate to an accurate model for interpolation of new values.

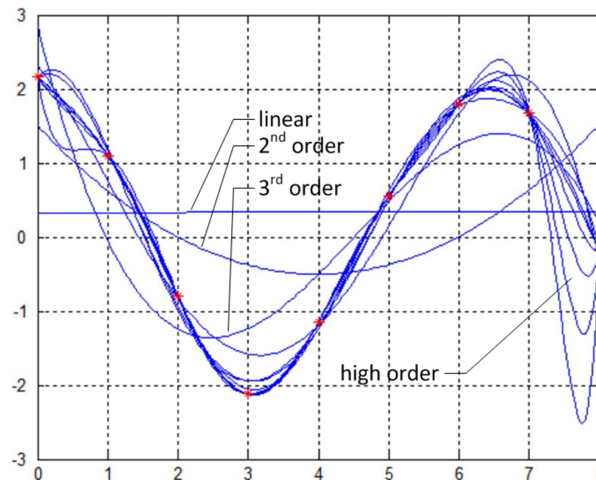


Figure 19 Curve-fitting of points with polynomials of increasingly higher order

With any polynomial network scheme, solutions may be discovered in the training phases which exhibit multicollinearity [80]. In short, some polynomial terms may have a nearly linear relationship to each other. Not only does this lead to solutions with excess terms, but the collinear terms tend be over-sensitive to noise in the data and to the introduction of new patterns. Standard regression techniques such as OLS will not automatically suppress multicollinearity. Breheny and others have shown that in fact, ridge regression is especially suited to improving computation with multicollinear data [81].

3.3.2.1 Ridge Regression – Theory

Ridge Regression (RR), developed by Hoerl and Kennard [82], purports advantages for multi-dimensional computation. The basic formulation is similar to both OLS and GLS:

$$\widehat{W}_{ridge} = \left(\hat{P}^T\hat{P} + \lambda I\right)^{-1}\hat{P}^T\hat{Y}$$
$$where\ I \equiv the\ p \times p\ identity\ matrix\ (p \equiv \#of\ polynomial\ terms) \qquad (53)$$
$$\lambda \equiv constant$$

It is instructive to note that this formulation is also similar to the Levenberg-Marquardt algorithm for non-linear least squares problems [43]. Hoerl and Kennard have observed that the function of the resulting diagonal matrix, λI, is to "regularize" or shrink all coefficient magnitudes, $\hat{W}_{ridge}$, rendering a model that is less sensitive to new patterns introduced during validation. This results in much less variance between training and validation errors. Hoerl and Kennard pose several key theorems. First, let us define the MSE of expected model outputs versus training outputs, and alternately versus validation outputs. Additionally, we define the expression for the variance between these two error measurements:

$$MSE_{\hat{Y}} = \left|\left|\hat{Y} - \hat{P}\widehat{W}^T\right|\right|^2 \equiv MSE\ of\ network\ training\ errors \qquad (54)$$

$$MSE_{\hat{Y}^*} = \left|\left|\hat{Y}^* - \hat{P}^*\widehat{W}^T\right|\right|^2 \equiv MSE\ of\ network\ validation\ errors \qquad (55)$$

$$VAR(MSE_{\hat{Y}}, MSE_{\hat{Y}^*}) \equiv variance\ between\ training\ and\ output\ errors \qquad (56)$$

Findings of Hoerl and Kennard [82] are presented in the parlance of CI:

- $Theorem\ 4.1\!:\ as\ \lambda \to \infty, VAR(MSE_{\hat{Y}}, MSE_{\hat{Y}^*}) \to 0$
  The total variance of validation versus training errors is a continuous, monotonically decreasing function of increasing λ.
- $Theorem\ 4.2\!:\ as\ \lambda \to \infty, MSE_{\hat{Y}} \to \infty\ \ and\ \ MSE_{\hat{Y}^*} \to \infty$
  The squared training and validation bias (error) is a continuous, monotonically increasing function of increasing λ.

- $Theorem$ 4.3: (Existence Theorem) $\exists\lambda > 0, \ni MSE_{\hat{Y}}(\lambda) < MSE_{\hat{Y}}(0)$
  There always exists a λ such that the MSE for ridge regression is less than the MSE for OLS.

The results of these theorems are visualized in Figure 20 relative to OLS. From this we can infer that an improved MSE is always achievable using ridge regression, and it is numerically computable given that both variance and the square of the bias are monotonic.

Breheny presents a theorem which affords an additional benefit [81]:

- $Theorem$: $\forall\, \hat{P},\ \left(\hat{P}^T\hat{P} + \lambda I\right)$ is always invertible, thus there is always a unique solution of $\widehat{W}_{ridge}$.

This can be used to kick a regression process that is stuck on a singular matrix inversion problem out of insolubility, simply by adding λI to the inversion term where λ is very small.

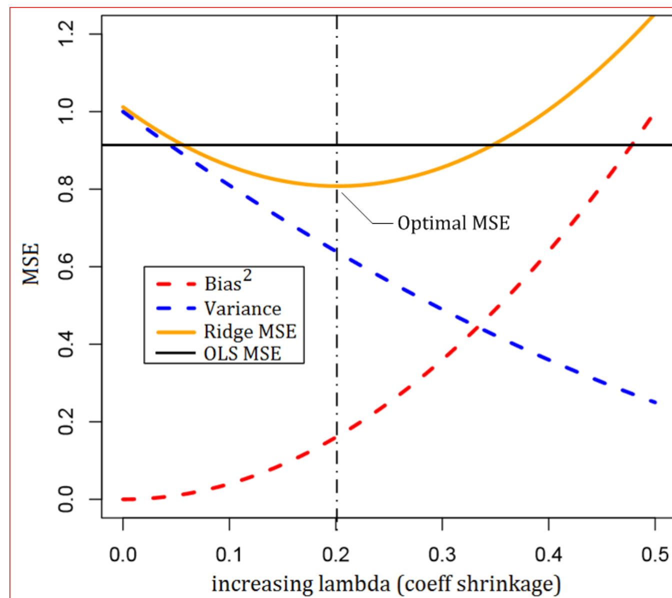


Figure 20 Ridge Regression [83]: The variance-bias tradeoff, and performance vs. OLS

3.3.2.2 Ridge Regression – Implementation

Celov et al. propose a Newton-Raphson/Fisher-scoring numerical method to compute an optimized λ, or "regularization parameter" [84]. In our context, "optimized" equivalently means that this process will produce a minimum-variance λ. This is useful for several reasons as will be discussed. Chiefly, it computes an initial value for λ from which an optimized MSE can be found using Hoerl's Theorems 4.1 and 4.2; though the minimum-variance λ will also coincide with a non-ideal bias, one can utilize the monotonicity of the proportional λ-bias relationship to subsequently search for a smaller λ that yields an optimized MSE. First, in developing algorithms which solve ridge regression, Shedden and other researchers have defined the "effective degrees of freedom" of a ridge regression problem as [85]:

$$df = trace[\hat{P}\left(\hat{P}^T\hat{P} + \lambda I\right)^{-1}\hat{P}^T] \tag{57}$$

Trace refers to a standard operation in linear algebra [86]. It is observed that λ and *df* cannot be solved for independently. Celov et al. use Singular Value Decomposition (SVD) [87] in their formulation, assuming a given df:

$$let\ [u\ s\ v] = SVD\left(\hat{P}\right),\ \ d_i \equiv i^{th} diagonal\ entry\ of\ 's' \tag{58}$$

$$df = \sum_{i=1}^{p} \frac{d_i^2}{d_i^2 + \lambda}\ \ ,\ \ p \equiv rows\ of\ \hat{P} \tag{59}$$

$$h(\lambda) = \sum_{i=1}^{p} \frac{d_i^2}{d_i^2 + \lambda} - df = 0\,, \qquad \frac{\partial h}{\partial \lambda} = -\sum_{i=1}^{p} \frac{d_i^2}{(d_i^2 + \lambda)^2} \tag{60}$$

$$h(\lambda) \cong h(\lambda_0) + (\lambda - \lambda_0)\frac{\partial h}{\partial \lambda}|_{\lambda=\lambda_0} = 0 \tag{61}$$

$$\lambda = \lambda_0 - \left[\frac{\partial h}{\partial \lambda}|_{\lambda=\lambda_0}\right]^{-1} h(\lambda_0) \tag{62}$$

iterative summation: (63)

$$1.\ assume\ d_i^2 = 1, \qquad \lambda_0 = \frac{p - df}{df} \tag{64}$$

$$2.\ \lambda_{j+1} = \lambda_j + \left[\sum_{i=1}^{p} \frac{d_i^2}{\left(d_i^2 + \lambda_j\right)^2}\right]^{-1} \left[\sum_{i=1}^{p} \frac{d_i^2}{d_i^2 + \lambda_j} - df\right] \tag{65}$$

Though (57) implies a computable solution, the above method still leaves the choice of *df* as an open issue. Many researchers have suggested using various quantities such as the total number of training patterns. This author has tried multiple quantities by trial and error, and a serendipitous choice for *df*, applying RR to single-layer polynomial networks, appears to be the degree of the polynomial, i.e., the degree of the maximum-degree monomial. The results of this choice will be seen in further sections. The MATLAB code implementing the numerical method of (58)-(65), and the code completing the optimization of λ, are given in 7.5 Appendix E – MATLAB Code: Iterative Ridge Regression.

As an initial proof-of-concept, the RR technique was applied to a single-layer polynomial network solution computed for a highly non-linear 2-input, 2500-point dataset, DCDCgldd (detailed later). This implementation of the iterative ridge regression process yields the training and validation results seen in Figure 21. The standard OLS algorithm (PolyNet) produces a monotonically descending RMSE trend line in training. But the generalization problem, inherent with polynomial and other node-based solutions, is pronounced in validation where the RMSE line diverges upward from that of the training line after the network is grown to greater than approximately 100 nodes. In contrast, the RR implementation (PolyRidge) yields a very similar shaped RMSE trend in both training and validation, and sustains a better optimal RMSE in validation.

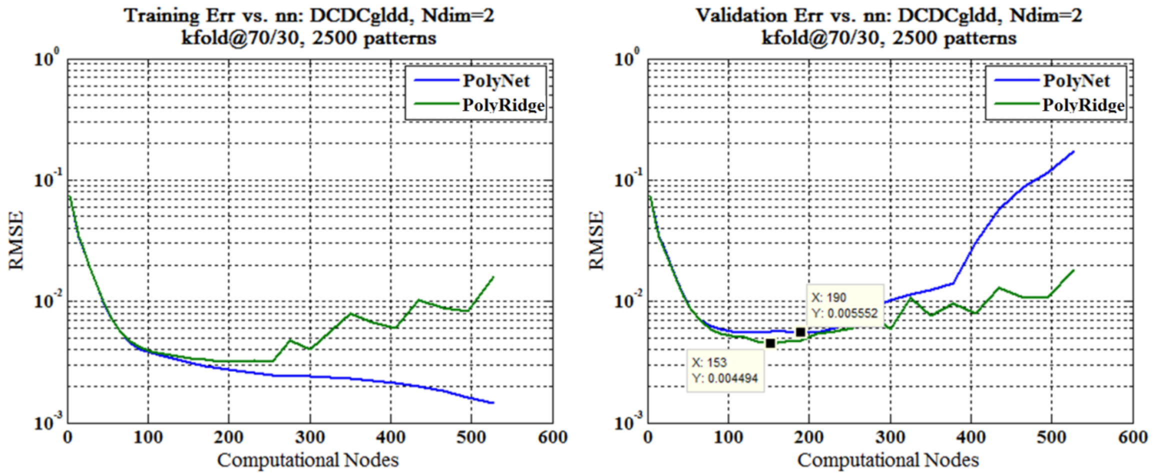


Figure 21 Ridge regression teaser: (PolyRidge) vs. OLS regression (PolyNet)

The shrinking of both λ and of the resulting training RMSE during the iterative process are plotted in Figure 22.

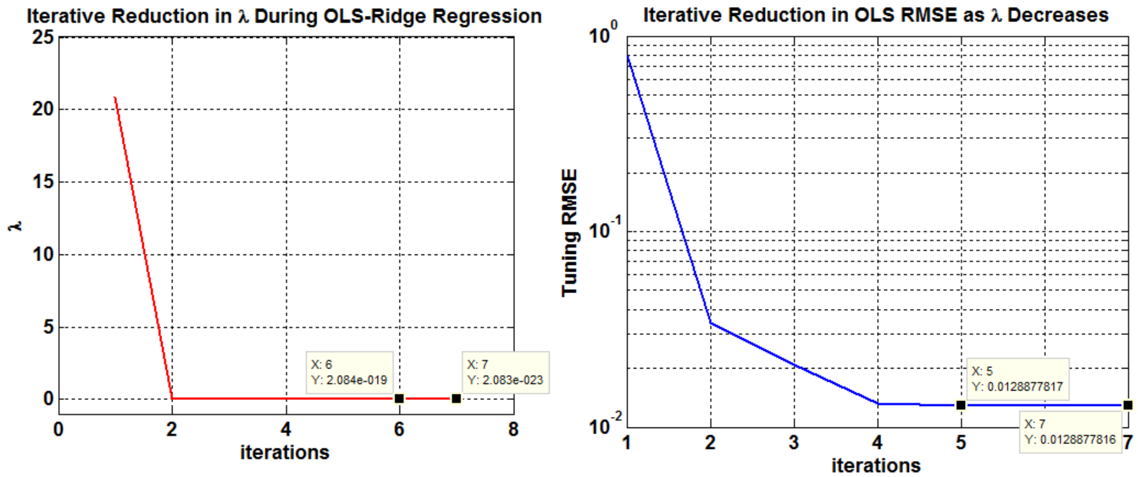


Figure 22 Iterative Ridge Regression: Both λ (Left) and training RMSE (Right) shrink monotonically towards optimal values during processing

3.3.3 Iterative Regression Hybrids Using OLS, GLS, and Ridge Methods

In the course of the research for this work, several hybrid regression techniques were explored among OLS, GLS, and RR methods. Multiple combinations of these basic types were possible, and many were tested including:

- OLS + RR: 1) RR → compute min-variance solution, 2) OLS → optimize
- OLS + RR: 1) OLS → compute initial solution, 2) RR → optimize

- GLS + RR: 1) RR → compute min-variance solution, 2) GLS → optimize
- GLS + RR: 1) GLS → compute initial solution, 2) RR→ optimize
- GLS + RR: embedded → iteratively compute single optimization step combining compute GLS and RR together (multiple implementations possible)
- etc.

Winners and losers were borne out in the course of extensive initial testing not included in this study. Only the most promising contenders were retained for inclusion in several polynomial-based network variants, discussed in the following Chapter. This section introduces these hybrid regression algorithms.

3.3.3.1 The GLS + Ridge Regression Minimum-Variance Hybrid

A special result was discovered for one hybrid combination of GLS and RR methods. By combining the two techniques, it was observed that an optimal minimum-variance RMSE trend line is attainable relative to using either RR alone, or in other combinations of RR with both OLS and GLS. The combination of methods is trivial:

1. Compute the min-variance λ_{minvar} for given training inputs, max polynomial order, and number of training patterns (See Appendix E: 6.5.1 7.5.1 Computation of an Initial Minimum-variance λ – OLSridge_reg.m excerpt).
2. Compute iterative GLS regression as before, incorporating the equation:

$$\widehat{W}_{minvar} = \left(\hat{P}^T\widehat{\Omega}_{GLS}\hat{P} + \lambda_{minvar}I\right)^{-1}\hat{P}^T\widehat{\Omega}_{GLS}\hat{Y} \quad (66)$$

The MATLAB code for this regression hybrid can be seen in Appendix F: 7.6.1 Iterative GLS + RR Minimum-Variance Regression – GLSminvar_reg.m excerpts. The effect of this technique is seen below in Figure 23. Though the terrible bias renders the method unusable as a final step of a regression process aimed at accuracy, the near-equivalence of the training and validation RMSE curves is remarkable. Again using the 2-input, 2500-point non-linear dataset, DCDCgldd, the PolyNet algorithm (simple one-step OLS regression) exhibits expected

generalization error in validation. But the minimum-variance GLS-RR combination results in almost no generalization error for the same training and validation patterns. Furthermore, though the overall bias is untenable, the GLS-RR method identifies during the training phase what could potentially be an optimal set of monomial terms for this dataset. No such indications are present in this case or as usual for Polynet or other similar OLS techniques. This points to the use of the GLS-RR method as a potential "optimal model probe" which could be deployed as the first phase of a multiple-step training algorithm. This "probe" effect has significant potential not just for polynomial-based techniques, but for any node-based CI technique which is normally susceptible to large generalization errors and training inefficiencies.

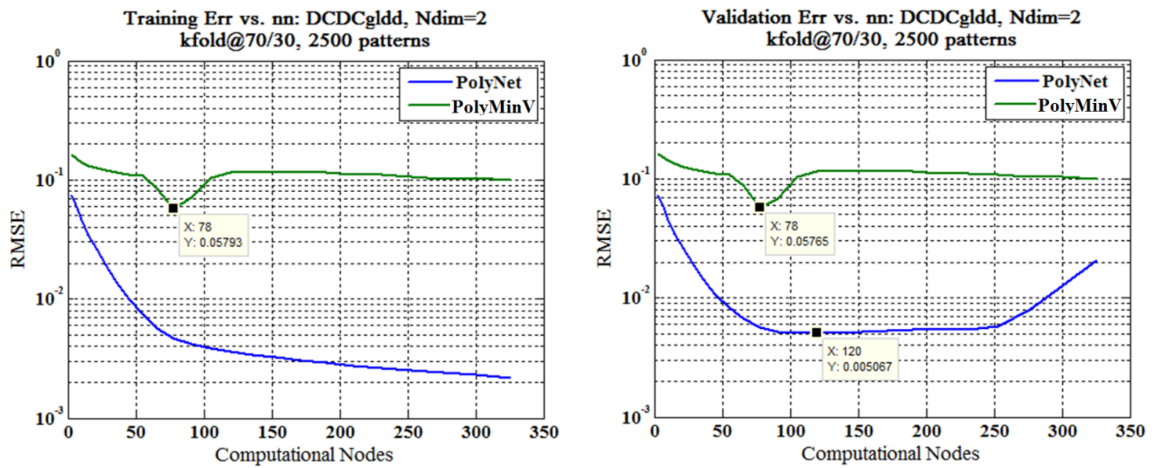


Figure 23 Effect of GLS-RR minimum-variance method: minimal difference in training and validation RMSE curves

3.3.3.2 The GLS + Ridge Regression Full Optimization Hybrid

This hybrid regression method is similar to the previous GLS + RR method of Section 3.3.3.1. However, the process is carried out to attain a minimum RMSE, as with the standard RR algorithm of Section 3.3.2.2. Additionally, the order of operations of this hybrid regression technique differs from that of the λ_{minvar} process. In this hybrid method:

1. A full iterative GLS regression is performed as in Section 3.3.1.

2. The resultant numerator $(\hat{P}^T \Omega_{GLS_{final}} \hat{Y})$ and denominator $(\hat{P}^T \hat{\Omega}_{GLS_{final}} \hat{P})$ from 1 are retained as constant matrices.
3. The iterative RR method of Section 3.3.2 is performed on these constant matrices.

The MATLAB code of this hybrid method can be seen in Appendix F –
7.6.2 Iterative GLS + RR Full Optimization Regression – GLSridge_reg.m excerpts. Results of the use of this method, incorporated into two different polynomial-based network variants, are evaluated alongside other variants in Chapter 5, Experimental Results.

3.4 Automated N-Dimensional Radial Clustering

Due to the introduction of several compute-intensive iterative methods herein, a solution was sought which can efficiently parse a multi-dimensional data space into roughly uniformly distributed clusters from which sample patterns can be extracted for training. For large datasets numbering in the thousands, or even tens of thousands of patterns, iterative regression techniques can have prohibitive run times. For experiments herein where GLS regression techniques were used, a rapid, simple clustering method was adapted from an original algorithm by B. Wilamowski [88] to forward-process an input training set, extracting no more than approximately 300 sample patterns from evenly-distributed clusters. Note that this is a clustering method employed not for traditional pattern classification, but for segmentation of a multi-dimensional data space into roughly equivalent and adjacent groupings. For an input training set with *np* total patterns:

1. Initialization: The first cluster is established as the first pattern: $C_1=\{P_1\}$. A threshold radius, r, is established. The first cluster center is established as the first pattern: $c_1 = P_1$. The cluster pattern count, nc, is set to 1.

2. For the remaining patterns in the training set, $P_n \rightarrow P_2$ through P_{np}, if $||c_1 - P_n|| < r$, P_n joins this cluster set as $C_1=\{P_1, P_n\}$, and the current cluster center is updated incrementally as the following:

$$c_1 = \frac{P_2 + c_1 * nc}{nc + 1} \tag{67}$$

Otherwise, if $||c_1 - P_n|| \geq r$, P_n is compared against the next cluster, c_k.

3. If P_n has been compared against all cluster centers, c_k and has not become a member of any, a new cluster is formed: $C_2 = \{P_n\}$. The corresponding new center is established: $c_2 = P_n$. The cluster pattern count is incremented: $nc = nc + 1$.

4. Cluster formation continues until the final pattern, P_{np}, has been evaluated.

The result of this algorithm can be seen in Figure 24. In this example, 5,000 uniformly distributed points are generated with the Matlab peaks() function, and fast, forward-computed radial clustering is used to form first 7 total clusters, then 110. Since the only goal of this clustering method is to adequately parse the data space into sectors sufficiently distributed around that space, complete accuracy is not necessary for point density, number of cluster members, etc. In fact, this method is computationally greedy, and cluster boundaries may overlap. This does not violate the goal of distributing training pattern sampling uniformly throughout the data space. As shown in Figure 24, clusters become more uniformly distributed as the total number of centers increases.

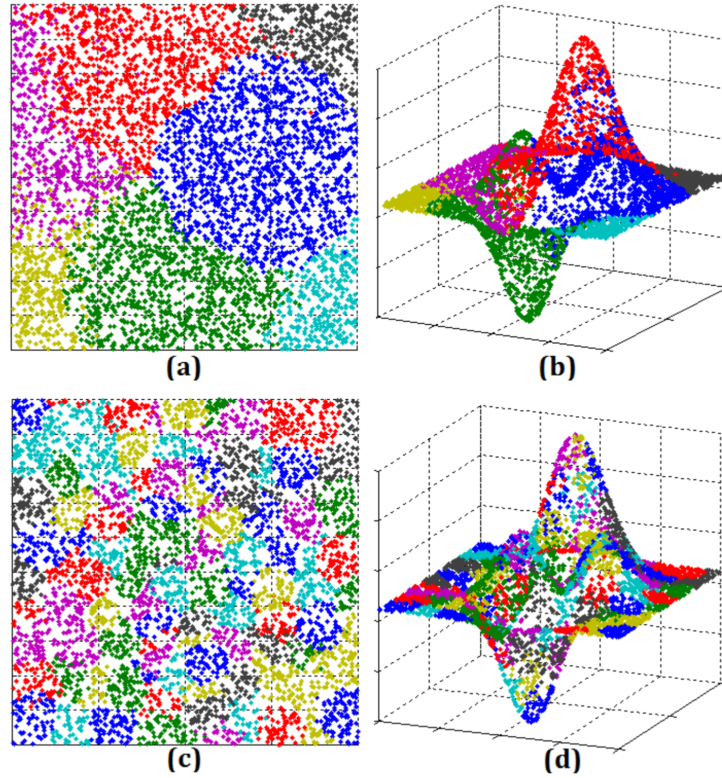


Figure 24 Fast, forward radial clustering of the Matlab peaks() function: (a) 7 cluster centers, top view, (b) 7 cluster centers, 3-D view, (c) 110 cluster centers, top view, (d) 110 cluster centers, 3-D view

The MATLAB code for this clustering method can be seen in

7.9 Appendix I – MATLAB code: Fast, Forward-Computing N-Dimensional Radial Clustering. As mentioned, all polynomial network variants which use iterative GLS regression employ this clustering to reduce training set sizes considerably.

Chapter 4
Proposed Polynomial-Based Learning Machines: Seven Variants within Three Species

Many combinations of the methods introduced in Chapter 3 were implemented in the course of extensive initial testing not included in this work. Three predominant species of algorithms survived initial testing. Within those species, seven combinations of techniques discussed have been retained as promising PLM variants which are explored for the remainder of this study. All seven variants are single-layer feed-forward networks following the topology and node computation of the FLN model discussed in Section 1.1.1. Additionally, the seven variants use the efficient monomial term generation algorithm of Section 3.1 in the course of solving for intermediate and final network parameters. However, the application of the statistical and regression methods of Chapter 3 differs significantly among the variants. Throughout this Chapter, "epoch" is defined as a training sequence which results in the computation of a final set of monomial coefficients for a particular network at a particular polynomial degree.

4.1 PolyNet Species – The Initial Next-Generation Polynomial Learning Machine

PolyNet is the most basic variant developed, and is the only member of the PolyNet species. It operates according to the following steps.

For given training pattern inputs and outputs, and for increasing maximum polynomial degree:

1. Generate monomial term indices and compute polynomial products according to Section 3.1.
2. Perform one-step OLS regression, as in (21), to obtain monomial coefficients.

3. RETAIN the resultant nonzero final monomial coefficients and their term indices for the current training's epoch.
4. STOP if either the maximum degree, or the max number of complete polynomial terms, computed by (8), has been reached. ELSE REPEAT 1 through 3 for the next highest maximum polynomial degree.

A flowchart for the logic of PolyNet is seen in Figure 25. The MATLAB code for the PolyNet variant can be examined in Appendix G – 7.7.1 The PolyNet Variant – PolyNet.m excerpts.

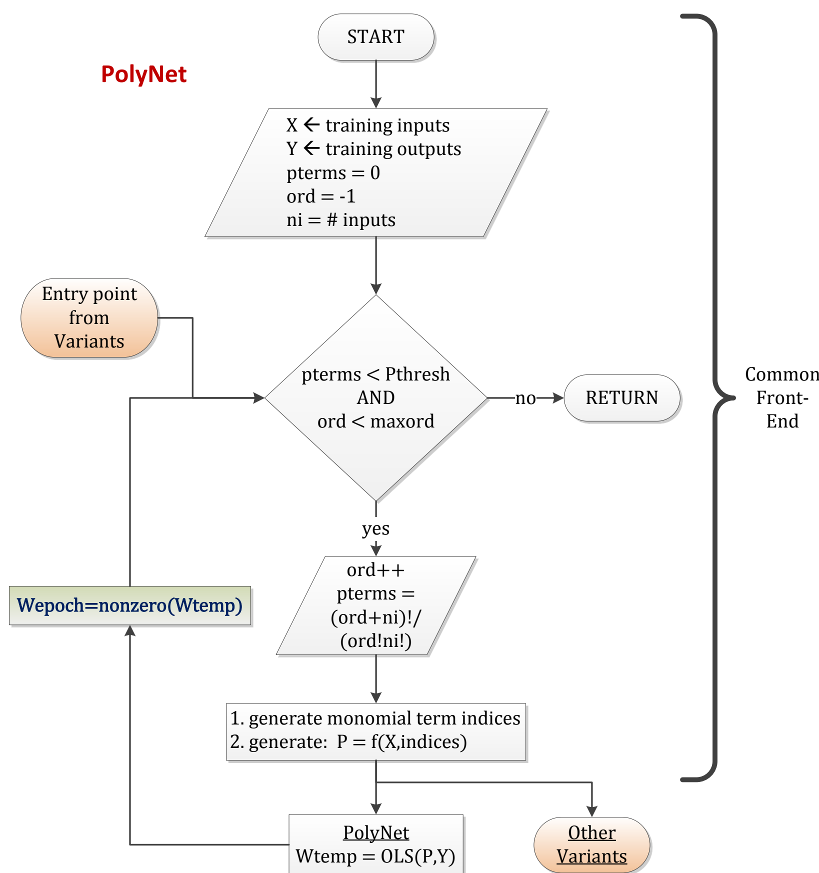


Figure 25 Flowchart for the PolyNet PLM variant

4.2 The PolyStat Species – Utilizing Statistical Pruning of Monomial Terms

Four of the final variants make use of the forward-computed statistical tabulation outlined in Section 3.2 in order to trim or prune monomial network terms following iterative regression steps. Additionally, these four variants can

each be considered extensions of PolyNet, essentially using that algorithm as a common front-end processor as depicted in Figure 25. The four variants differ in their use of the different regression algorithms introduced in Section 3.3. The PolyStat variants are:

- PolyStat – statistical term pruning plus standard OLS regression
- PlyGLS_P – same as PolyStat with iterative GLS regression instead of OLS
- PlyOLSR_P – same as PolyStat with iterative RR instead of OLS
- PlyGLSR_P – same as PolyStat with iterative hybrid GLS+RR instead of OLS

The PolyStat family of variants operates according to the following steps.

For given training pattern inputs and outputs, and for increasing maximum polynomial degree:

1. Generate monomial term indices and compute polynomial products according to Section 3.1.
2. Randomize the order of the training patterns, and extract a subset of these training patterns.
3. Iteratively per training pattern subset: Perform either one-step OLS regression, or iterative regression (GLS, OLSR, or GLSR, described in Section 3.3) to obtain a set of temporary monomial coefficients.
4. Compute running means and standard deviations of each coefficient per iteration.
5. REPEAT 2 through 4 until EITHER the sum of the running gradients of the coefficient means stabilizes below a threshold, OR until the maximum number of iterations is reached.
6. COMPUTE a threshold for the maximum-allowed STD per individual coefficient terms.
7. REMOVE monomial terms whose coefficient STDs are above the threshold computed in 6.
8. IF terms were removed in 7, CONTINUE steps 3 through 7 with the pruned set of monomial terms.

9. ELSE, RETAIN the resultant nonzero final monomial coefficients and their term indices for the current training's epoch.
10. STOP if either the maximum degree or the max number of complete polynomial terms, computed by (8), has been reached. ELSE REPEAT 1 through 9 for the next highest polynomial degree.

A flowchart for the PolyStat variants is seen in Figure 26. The MATLAB code for the all PolyStat variants is nearly identical to that seen in Appendix C – Option 2 Example – PolyStat.m excerpts. As mentioned previously, the variants differ only in the regression method employed.

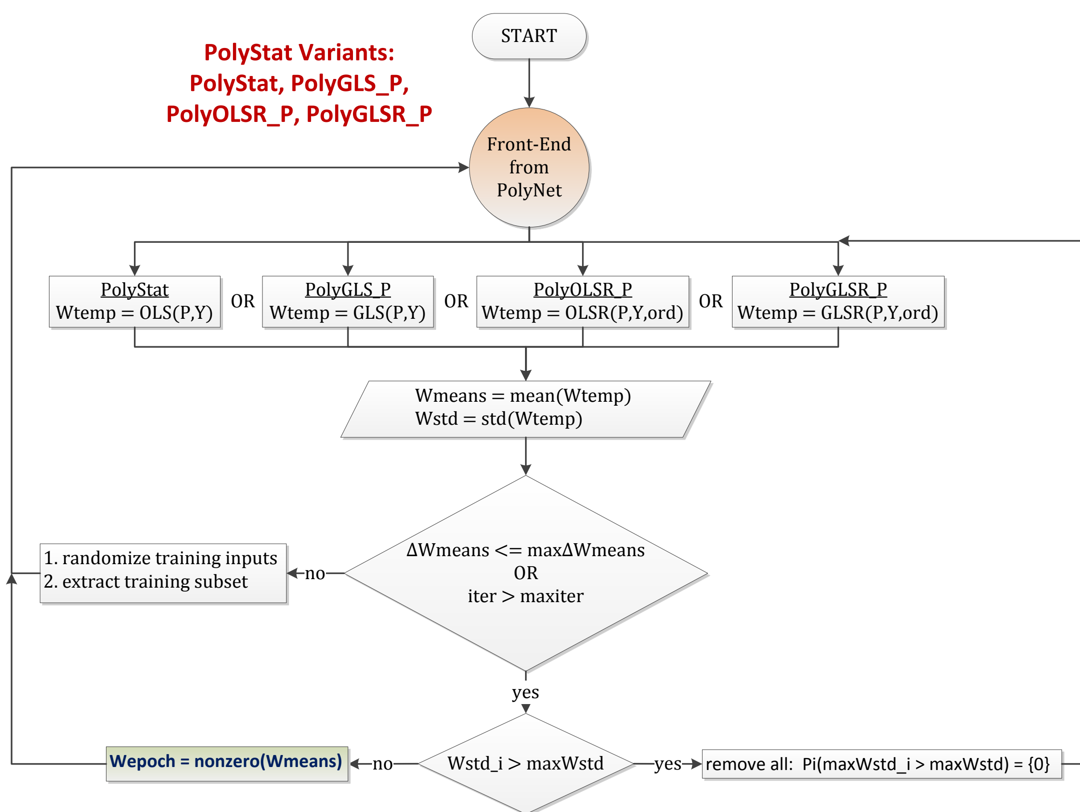


Figure 26 Flowchart for the PolyStat family of PLM variants

4.3 The PolyPaP Species – A Probe-and-Prune Methodology

Two of the final variants incorporate two regression stages. In the first stage, the minimum-variance GLS regression of Section 3.3.3.1 is applied as a probe to seek

the optimal polynomial degree and monomial term set for a particular solution in an attempt to defeat generalization issues in later stages. In the second stage, all monomial terms are regenerated at once at the optimal order discovered in the first stage. Then, one of two regression methods is applied to iteratively seek a final monomial set which is optimized for minimum RMSE applied to the training patterns. Additionally during the second stage, running statistics are accumulated as with the PolyStat variants. However, at the end of a training epoch, the final statistical quantities are used to identify only the single monomial term whose coefficient exhibited the greatest excursion during training iterations. This single term is removed from the solution set, and training proceeds until no terms remain in the set. The two PolyPaP variants differ in their use of two regression algorithms introduced in Section 3.3. The PolyPaP variants are:

- PlyPaPGLS – optimal polynomial degree probe, plus GLS regression and statistical term pruning
- PlyPaPGLSR – same as PlyPaPGLS with iterative hybrid GLS+RR regression

The PolyPaP family of variants operates according to the following steps.

For given training pattern inputs and outputs:

Part I – Minimum-Variance Probe

1. Per current polynomial degree, generate monomial term indices and compute polynomial products according to Section 3.1.
2. Perform the iterative minimum-variance GLS regression.
3. COMPUTE errors: If a current lowest MSE is computed, RETAIN the current order as ProbeOrder.
4. STOP if either the maximum degree or the max number of complete polynomial terms, computed by (8), has been reached. ELSE REPEAT 1 through 3 for the next highest polynomial degree.

Part II – Iterative Regression and Pruning

1. Generate all monomial terms at the polynomial degree equal to ProbeOrder found in Part I.

2. Randomize the order of the training patterns, and extract a subset of these training patterns.
3. Iteratively per training pattern subset: Perform either iterative GLS regression, or full iterative GLS+RR regression (described in Section 3.3) to obtain a set of temporary monomial coefficients.
4. Compute running means and standard deviations of each coefficient per iteration.
5. REPEAT 2 through 4 until EITHER the sum of the running gradients of the coefficient means stabilizes below a threshold, OR until the maximum number of iterations is reached.
6. REMOVE the single monomial term whose coefficient STD is the maximum. Pterms = pterms – 1. RETAIN the resultant nonzero final monomial coefficients and their term indices for this epoch.
7. STOP if pterms < 1. ELSE REPEAT 2 through 6 with the reduced monomial term set.

A flowchart for the minimum-variance probe phase of the PolyPaP variants is seen in Figure 27. The flowchart for the optimization and pruning phase is seen in Figure 28. The MATLAB code for the all PolyPaP variants appears in Section 7.7.2 of Appendix G – PlyPaPGLSR.m excerpts. As mentioned previously, the variants differ only in the regression method employed for the second phase.

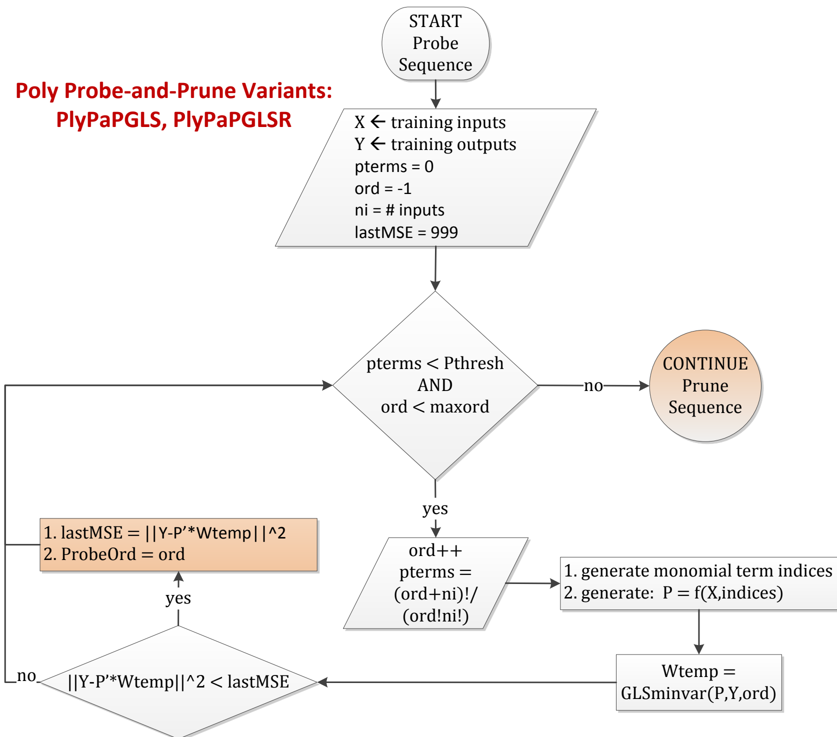


Figure 27 Flowchart for the Probe phase of the PolyPaP family of PLM variants

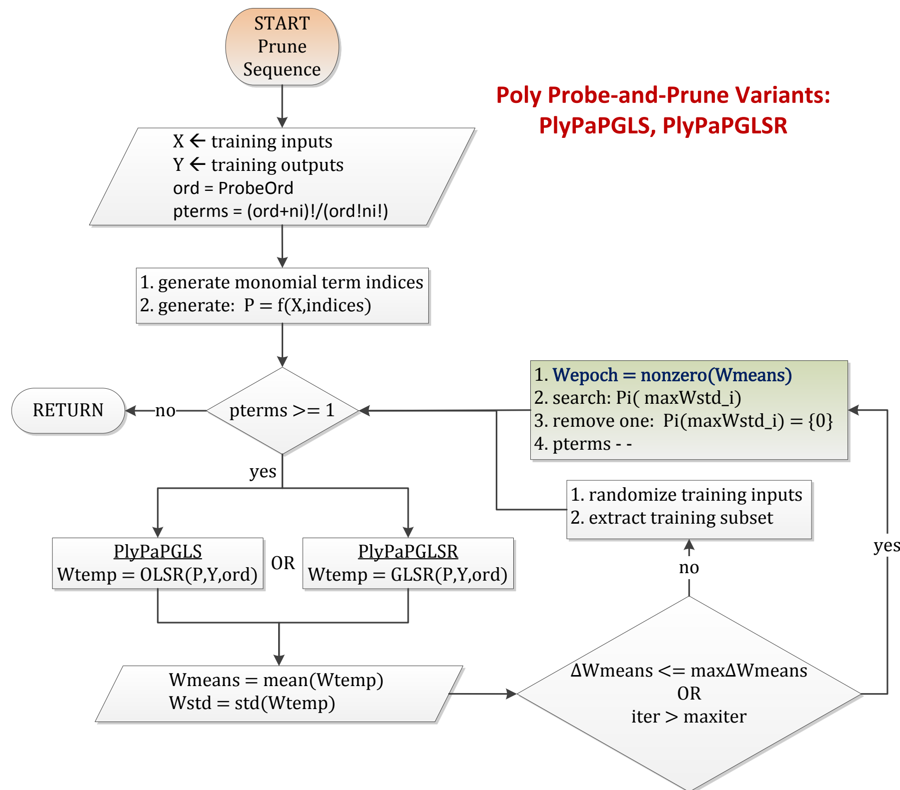


Figure 28 Flowchart for the Solve/Prune phase of the PolyPaP family of PLM variants

4.4 Summary of Polynomial-Based Learning Machine Variants

The main differences of the variants presented are summarized as a feature list in Table VI. These variants comprise the original methods of this study which will be deployed against other prominent learning machine processes found in the current literature.

Table VI
Feature Set of Polynomial-Based Learning Variants

	Poly	Min-Var	Regression Type				Statistical	STD(ΔCoeff) Term Pruning		
Species	Variant	Probing	OLS	GLS	OLSR	GLSR	Smoothing	== 0	Threshold	Single
PolyNet	PolyNet		X					X		
PolyStat	PolyStat		X				X	X	X	
	PlyGLS_P			X			X	X	X	
	PlyOLSR_P				X		X	X	X	
	PlyGLSR_P					X	X	X	X	
PolyPaP	PlyPaPGLS	X		X			X	X		X
	PlyPaPGLSR	X				X	X	X		X

Chapter 5
Experimental Results

The particular testing methodologies, adopted from other current rigorous studies, are discussed herein. Finally, results and findings from multiple experiments with two types of benchmark problems are discussed. All competing algorithms are exercised against nine total datasets within the two types.

5.1 Test Methodology

The proposed PLM variants and all competing CI methods were implemented according to theory and code specified by the original authors. The runtime parameters for the all methods were set based on the best performance observed during initial trials against the particular problems tested, and are noted within the results of the following sections.

The testing environment for all experiments consists of running 64-bit MATLAB(vR2013b) implementations of all algorithms over a 64-bit CentOS Linux operating system. All experiments were run in MATLAB's single-core mode on an Intel Core i7-3770 CPU @ 3.4GHz, with 8GB of RAM.

For all datasets in this study, both training and validation pattern vector input values were normalized over the range [-1:1]. Pattern vector outputs were normalized over the range [0:1]. For industrial electronics datasets using generated data, further processing was used to adequately simulate noise conditions in training. This will be discussed in detail in the next Section.

For all experiments, a 70/30 k-fold process was run for 20 separate trials per CI algorithm, and the final parametric results were averaged over the 20 trials. As implied by the 70/30 k-fold, the entire dataset is first randomized, and then 70% of the total dataset for each experiment is chosen for training, while a specific 30% of

training vectors are withheld for validation only. The next "fold" in the process withholds a different 30% for validation, and so on. The process repeats until all data patterns have been cycled through the validation process. All algorithms see the exact same permutations of the training and validation sets per each of the 20 trials of each experiment. All time metrics are measured in seconds, and all error metrics are evaluated with the standard Root-Mean-Squared Error (RMSE):

$$RMSE = \sqrt{\frac{\sum_{p=1}^{np} e_p^2}{np}} \tag{68}$$
$$\textit{where}: np \equiv \#\ \textit{of patterns computed}$$
$$e \equiv \textit{absolute error of output vs. desired}$$

5.2 Experiments with Industrial Electronics Problems

Many applications of CI methods are found in the area of industrial electronics. Two such problems are used for experiments herein: 1) output voltage control for a DC-DC converter under variable load, and 2) resolution of angular pose data from a 2-segment robotic arm to 3-D spatial location. For these problems, 2,500 uniformly distributed patterns were randomly generated over the normalized range [-1:1] for the function inputs. For training only, Gaussian distributed noise is added to the outputs, normalized in the range [0:1], with a 5% STD to simulate imperfect real-world data. For validation/testing, no noise is added to the simulated output data. This comprises a rigorous test of the algorithms' generalization abilities.

5.2.1 Voltage Control for a Czuk DC-DC Converter

An up-down Czuk DC-DC converter [89] is pictured in Figure 29. An example of individual component transient responses is pictured in Figure 30, where the circuit reaches stability after approximately 30ms. A voltage control system can be realized which targets a stable output voltage over C2 based on output load and duty-cycle of the switching pulse. 2,500 uniformly distributed data points were generated, representing load-conductance/duty-cycle pairs as constrained by

simulation of the circuit. Figure 31 shows the steady-state control surface of the problem parameters in relation to the desired output voltage.

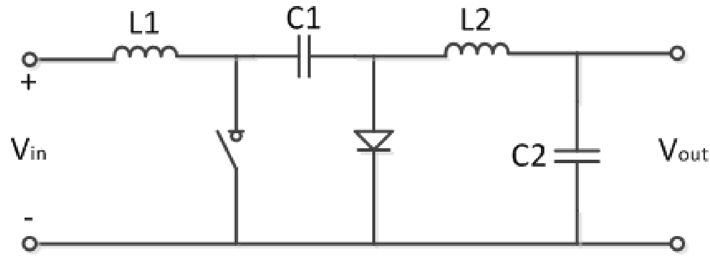


Figure 29 A Czuk up-down DC-DC converter circuit

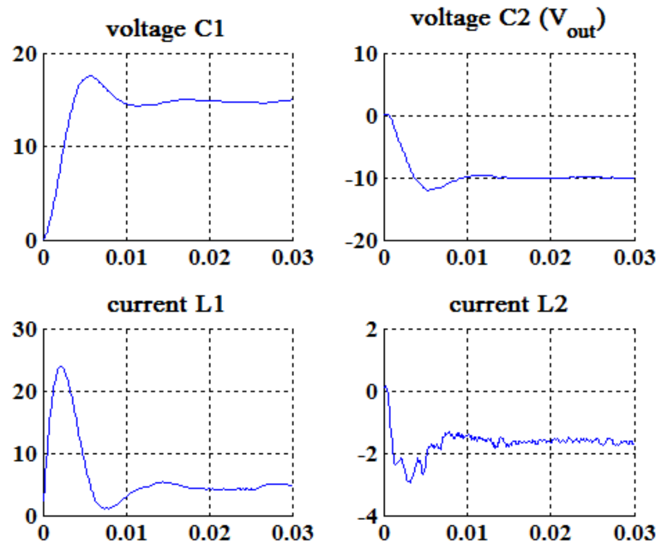


Figure 30 The Czuk DCDC Converter: Non-linear transient responses, 0 to 30ms

5.2.2 3-D Reverse Kinematics Control

A 3-D reverse-kinematics dataset was created. This emulates a control system that must transform the angles measured from the stepper motors on pivot points of a two-segment robotic arm to the resultant Cartesian location of the tip of the arm assembly as depicted in Figure 33(a). 2,500 points were generated from randomly distributed input angles according to the following three equations:

$$\begin{aligned} x &= R1\cos\alpha + R2\cos(\alpha+\beta) \\ y &= (R1\sin\alpha + R2\sin(\alpha+\beta)) \times \cos\phi \\ z &= (R1\sin\alpha + R2\sin(\alpha+\beta)) \times \sin\phi \end{aligned} \tag{69}$$

The distribution of points occupies a sphere of all possible locations of the tip of the arm assembly in Cartesian space, and can be seen in Figure 32. The highly non-linear nature of the resulting functions can be seen in the 2-D mappings of two such angles to one Cartesian dimension as in Figure 33(b). For the experiments herein, the 3-angle *y* component solution is exercised.

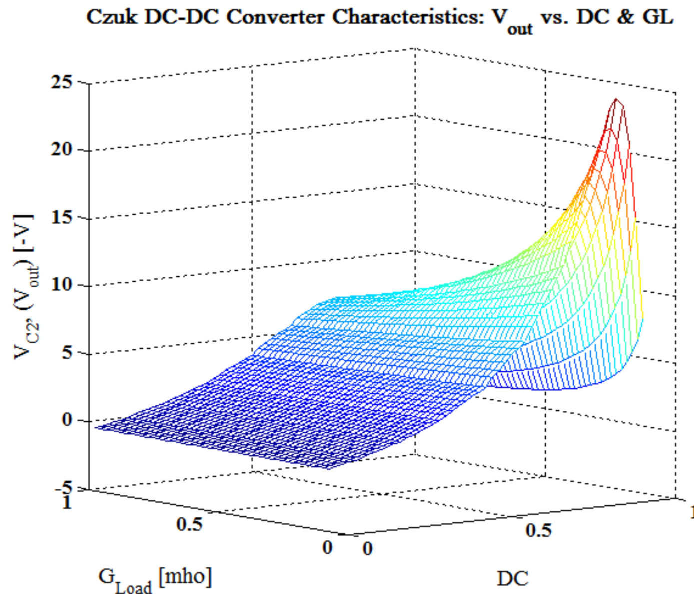


Figure 31 Czuk Converter: non-linear steady state relationships between load conductance and duty-cycle vs. output voltage

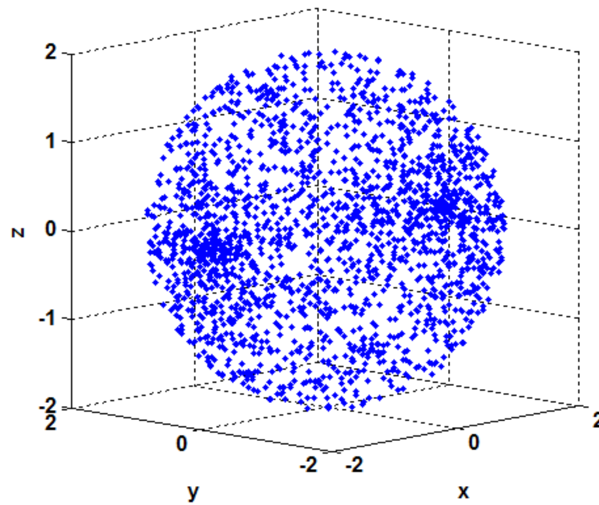


Figure 32 3-D reverse-kinematics: Resultant arm tip distribution in free-space of randomly generated angle positions

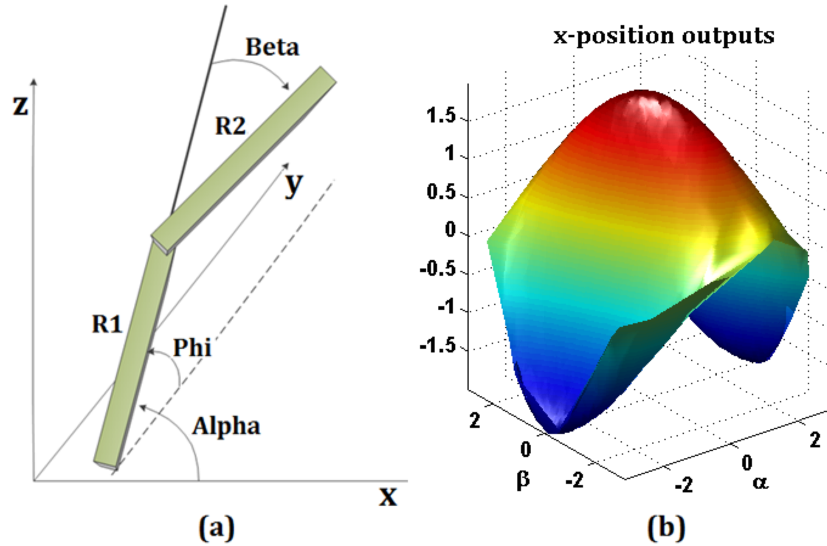


Figure 33 (a) Reverse-kinematics problem: input angles are mapped to Cartesian coordinates (x, y, z), (b) x-position vs. two input angles

5.2.3 Results for Two Industrial Electronics Problems

All CI algorithms discussed herein were run on the Czuk DC-DC and 3-D reverse-kinematics datasets. For the I-ELM variants, an impact factor of 2.7583 was set and centers were chosen randomly from the input domain. For the SVR algorithm, γ was set to 2^{-1}, $\epsilon = 2^{-3}$, and C was set to 1. For the four PLM variants that utilize threshold-based statistical pruning of terms based on calculated noise in the term coefficients, the *stdscale* parameter must be set. This represents a threshold of the number of normalized standard deviations above or below the mean STD of all term coefficients at which terms will be excised. Those factors are noted within the tabulated results.

5.2.3.1 Training and Validation Times for IE Problems

Figure 34 and Figure 35 display the captured run times for all algorithms operating on the two IE problems. The original PolyNet algorithm has the shortest training times of all algorithms tested across all comparable network sizes. Due to the multiple iterative processes contained within the other PLM variants, their training times can be orders of magnitude above the majority. Algorithms which contain both statistical term pruning and compound regression techniques, such as

PlyPaPGLS, require significantly longer run times. In these plots and others to follow, the PLM variants which utilize term pruning by threshold can exhibit non-monotonicity when compared with other algorithms evaluated per increasing node count. This occurs since monomial terms are generated according to increasing polynomial order, yet subsequent evaluation of coefficient noise during iterative training may dictate trimming more terms at later generation stages compared to earlier ones. Such plot traces are registered with data markers rather than with continuous lines. Examples of such occurrences can be seen with PlyGLS_P in Figure 34, and with PolyStat, PlyGLS_P, and PlyOLSR_P in Figure 35.

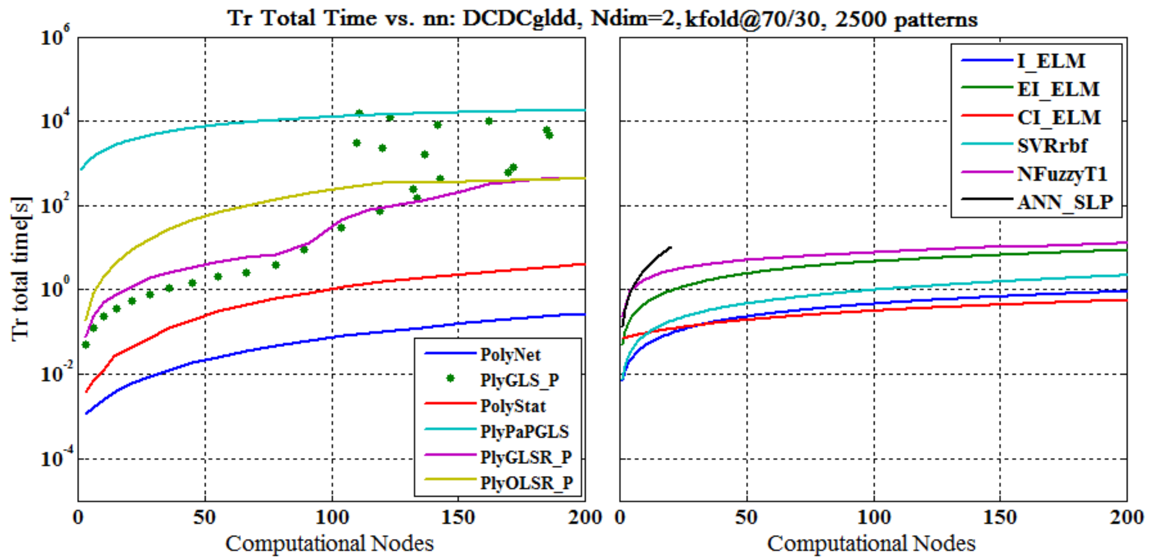


Figure 34 Total training times for all algorithms up to 200 nodes: DC-DC problem

Since the 3-D kinematics dataset exhibits both an additional dimension and a greater degree of non-linearity than the DC-DC problem, all training times rise for all algorithms as expected. However, the original PolyNet algorithm with one-step OLS regression remains relatively unchanged. Finally, it is noted that the ANN-SLP network rises sharply in training time compared to other methods as expected, due to the costly matrix computations inherent with LM-based neural network methods.

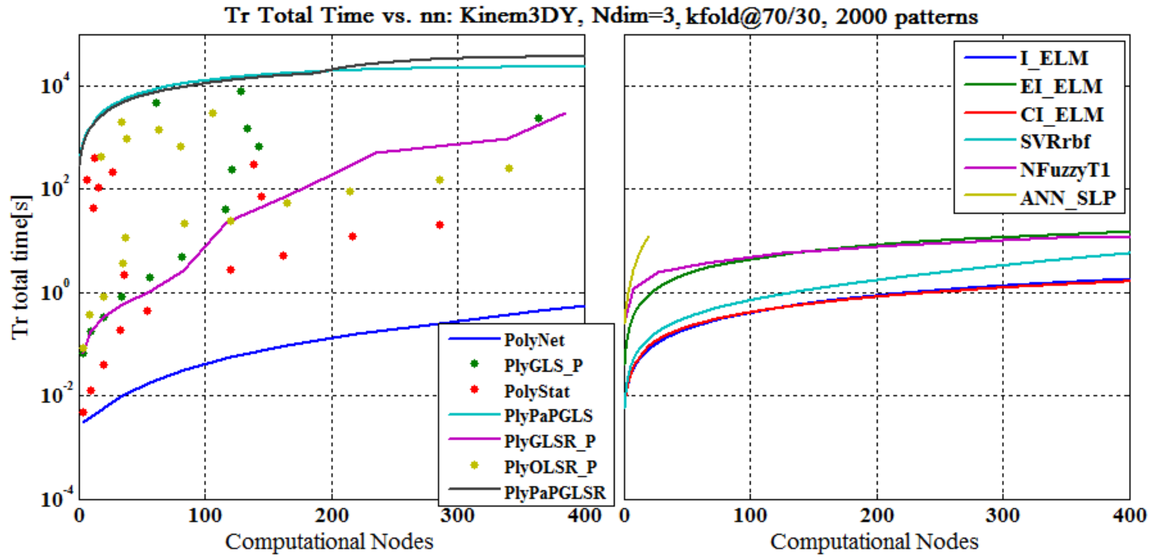


Figure 35 Total training times for all algorithms up to 400 nodes: 3-D Kinematics problem

For some applications, certainly for real-time uses or for any tasks in which compute time of the network is critical, validation time is more important than training time. One expects the stable, optimized network to require much less compute time per output computation. For all experiments herein, validation time is tabulated not as a running total as with training time, but is measured instead per each optimized network node size following each training cycle. This more accurately indicates performance of each algorithm under deployed conditions.

Such validation time results are seen in Figure 36 and Figure 37 for the DC-DC and Kinematics datasets, respectively. It is most notable that all PLM variants produce final optimized networks which are almost two orders of magnitude more efficient than all other methods except SVR-RBF. SVR-RBF also shares an advantage with only the N-D fuzzy system in that run time across a range of network sizes is relatively stable. This is useful particularly for hardware implementations where compute times are highly dependent upon a stable number of clock cycles. Among the PLM variants, all enhanced methods produce slightly more efficient optimized networks than the original PolyNet. This is likely due to the pruning of higher order terms from the final polynomial product engine, whereas all high-order terms are retained with the original algorithm. This is generally the case, but is not guaranteed for the solution of every dataset.

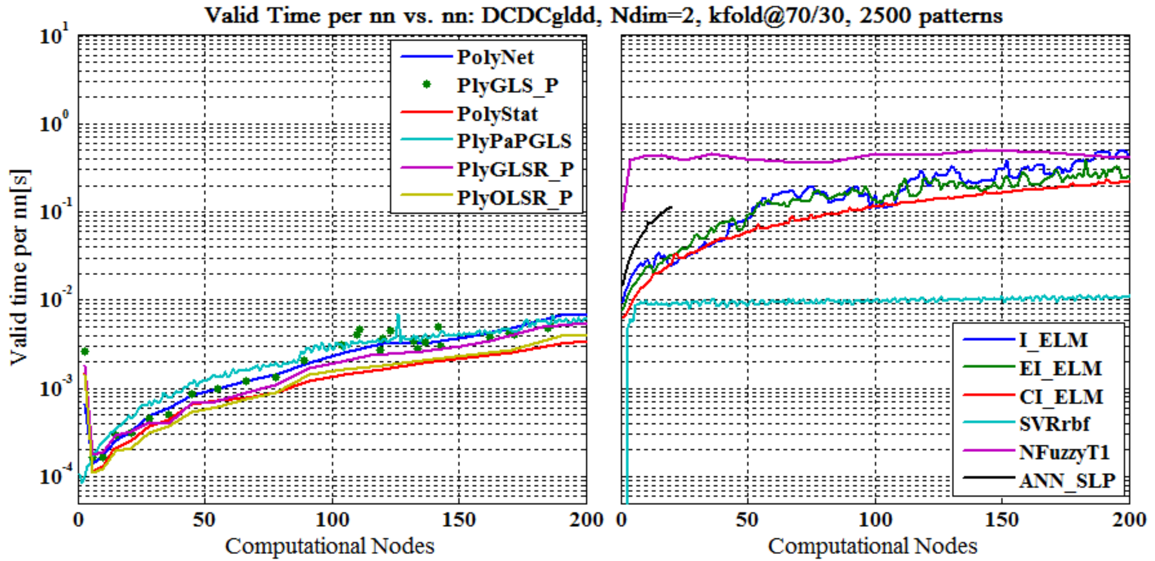


Figure 36 Validation times per network size vs. nodes for all algorithms: DC-DC problem

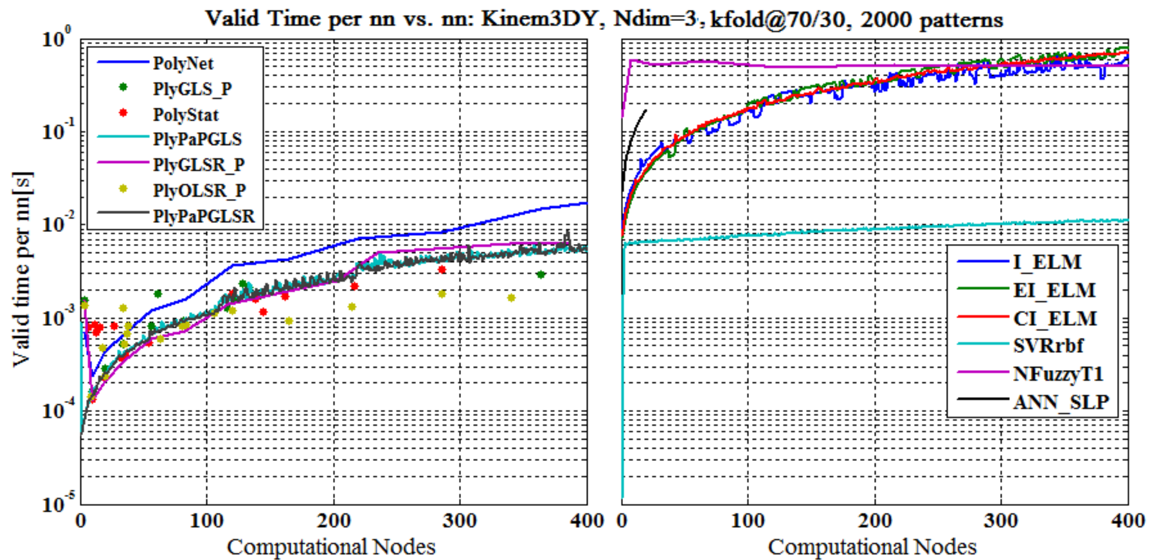


Figure 37 Validation times per network size vs. nodes for all algorithms: 3-D kinematics problem

5.2.3.2 Training and Validation Accuracy for IE Problems

For algorithms which exhibit good generalization performance (which may itself vary versus different datasets) training error is usually a decent indication of validation accuracy. In general, it is expected that validation accuracy will lag behind training accuracy due to the inability of the training processes to directly

infer new patterns encountered during validation. However, for the two IE problems, it is important to recall that significant noise was added to the trained output values which accounts for higher than usual training error in these cases. As such, most algorithms showed a decrease in performance error from training to validation.

Comparing Figure 38 through Figure 41, we see that the ANN-SLP algorithm achieved the best overall accuracy for both problems. However, three PLM variants – PolyNet, PolyStat, and PlyOLSR_P – achieved validation errors almost as low for both. These variants also converged to optimal performance with reasonably economical networks totaling less than 100 nodes each. Two PLM variants, PlyGLS_P and PlyPaPGLS, exhibit good generalization, but extremely poor error bias overall for both IE problems. It is possible that the GLS component of these algorithms tracks the normal noise added to the training outputs as the predominant trend in the data. Thus, where one counts on GLS to do particularly well in the presence of truly random and asymmetric noise, it is possible that it is duped to track noise with a symmetric normal distribution as the predominant trend in the data.

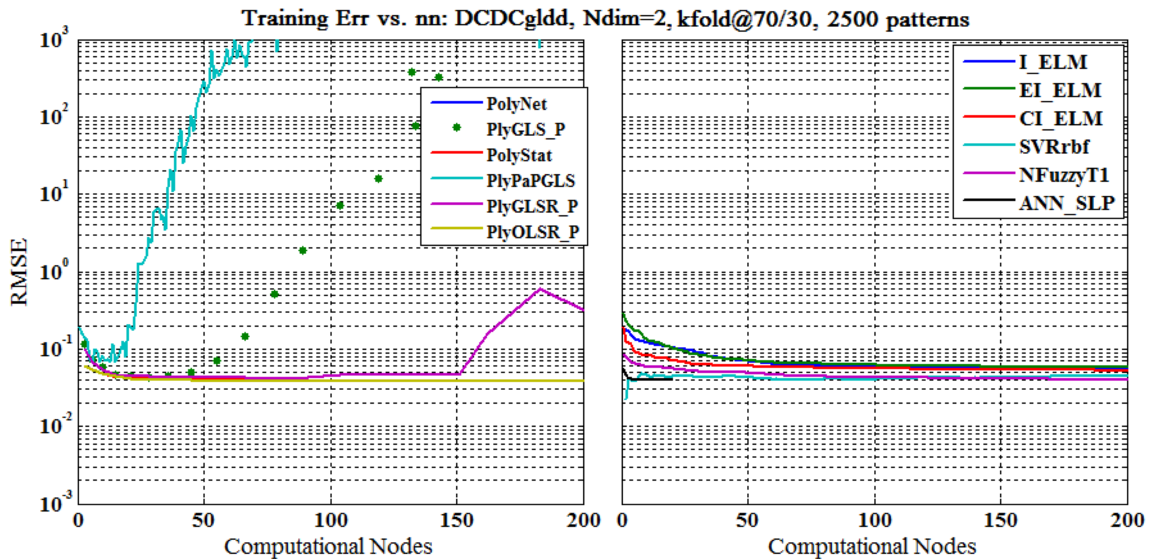


Figure 38 Training error for all algorithms up to 200 nodes: DC-DC problem

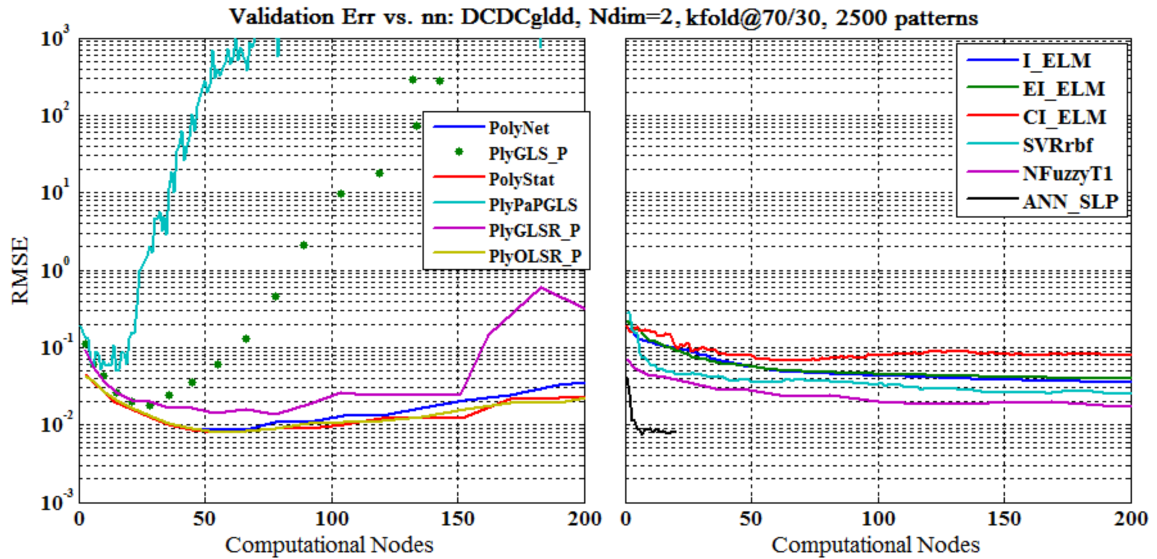


Figure 39 Validation error for all algorithms up to 200 nodes: DC-DC problem

Particularly for the Kinematics problem, the PLM variants incorporating RR have almost identical traces between training and validation, attesting to the exceptional generalization enhancement afforded by the technique. Though validation error for these variants is unimpressive for these datasets, further experimentation will yield more promising qualities.

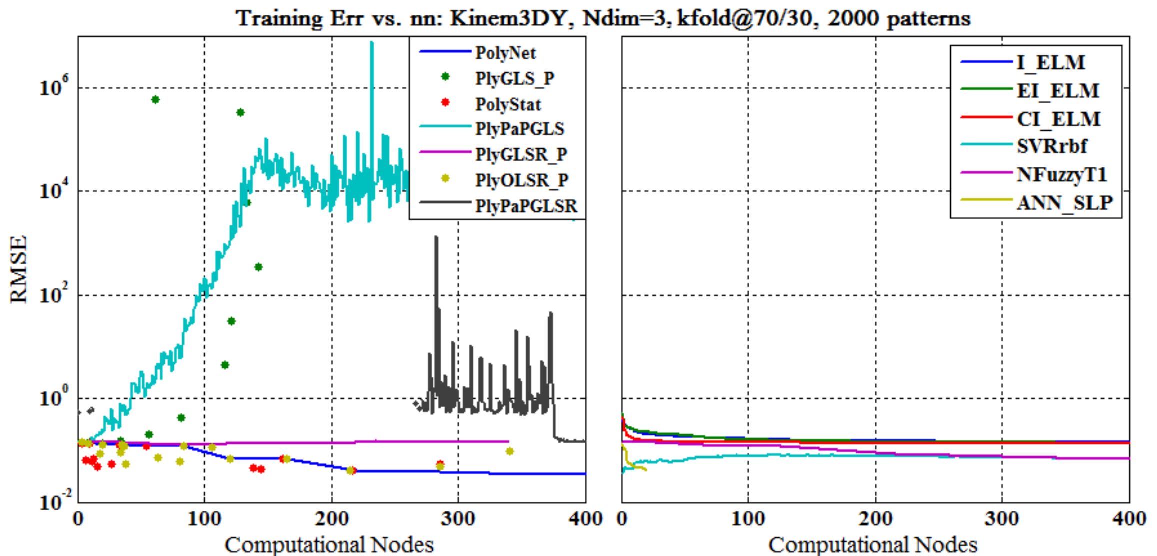


Figure 40 Training error for all algorithms up to 400 nodes: Kinematics problem

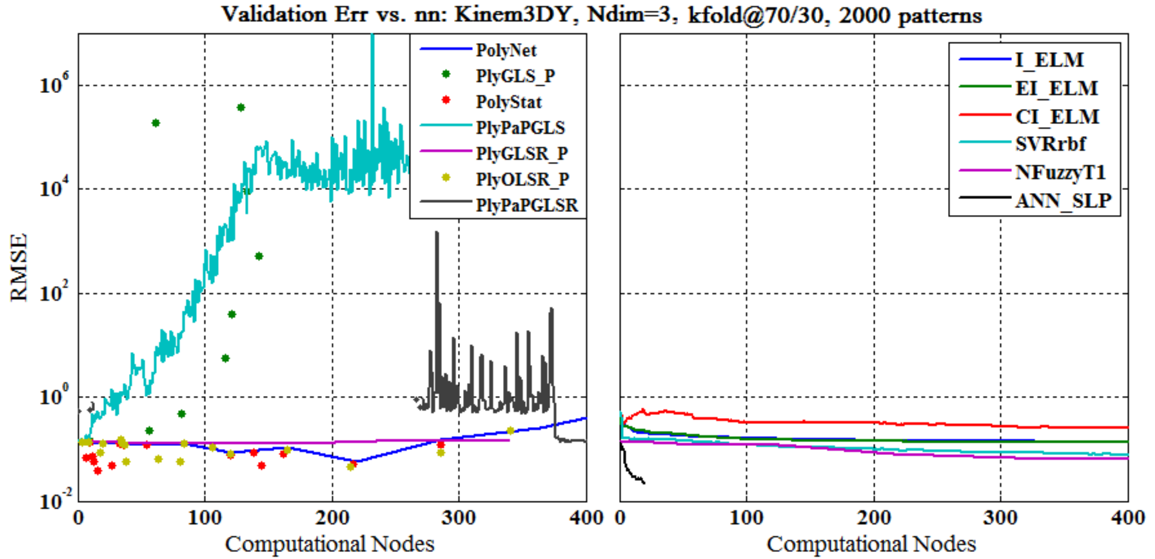


Figure 41 Validation error for all algorithms up to 400 nodes: Kinematics problem

5.2.3.3 Tabulation of IE Dataset Results

Table VII displays the relevant efficiency metrics for the thirteen algorithms compared against the IE datasets. Training and validation/testing times are in seconds, and the final optimized network sizes are expressed as the number of computational nodes. In the case of the TSK FS, the number of nodes denotes the number of values in the optimized fuzzy output table. Winners are noted in green bold, and honorable mentions are noted in green italics. Significant last-place finishers are highlighted in red.

TABLE VII
Algorithm Efficiencies: Processing Times and Network Size IE Problem

Algorithms	Czuk DCDC Training [s]	Czuk DCDC Testing [s]	Czuk DCDC Nodes [#]	3-D Kinematics Training [s]	3-D Kinematics Testing [s]	3-D Kinematics Nodes [#]
ANNSLP	1.9185	0.0480	**7**	12.3101	0.1480	**20**
TSK Fuzzy	21.877	0.3905	353	387.71	0.4637	11201
I-ELM	4.1230	1.4662	995	3.9960	1.5369	993
EI-ELM	41.168	1.7415	998	32.440	1.3298	996
CI-ELM	2.7211	1.1590	999	3.8145	1.5443	996
SVR-RBF	31.023	0.0176	972	46.624	0.0166	999
PolyNet	**0.0361**	0.0012	62	**0.0450**	0.0024	85
PolyStat	1.0022	*0.0008*	55	173.62	0.0009	86
PlyOLSRP	1.3737	0.0010	63	227.23	0.0011	151
PlyGLSP	0.8615	*0.0005*	28	0.3376	*0.0003*	**20**
PlyGLSRP	4.8895	0.0014	70	2.6381	*0.0007*	83

	Czuk DCDC			3-D Kinematics		
Algorithms	Training [s]	Testing [s]	Nodes [#]	Training [s]	Testing [s]	Nodes [#]
PlyPaPGLS	2,313.3	**0.0003**	455	955.98	**0.0001**	451
PlyPaPGLSR	78,772	0.0128	**3**	36,028	0.0077	72

In terms of computational efficiency for training the network, the original PolyNet PLM is the most efficient, beating the closest rivals by two orders of magnitude. This is valuable for applications that require rapid turnaround, or for in-situ network retraining in cases where solutions are sought that must adapt to new trends in incoming data.

In terms of efficiency for the testing of the deployed networks, the PLM variants are arguably best. Though they require more nodes than the ANN, the real-time computation cycles of the *tanh* kernel function of the ANN exceed that which is necessary for the simple product and sum operations of the PLM node functions. As such, the testing times of the final PLM networks attest to their optimal efficiency.

The final and perhaps most important metric is network accuracy, particularly in the deployed stage, depicted by the testing results of TABLE VIII. The single-layer ANN, trained with the NBN algorithm, remains the winner for both datasets tested. The PolyStat and PlyOLSR_P PLM variants performed almost as well as the ANN for the DCDC problem. For the higher-dimensionality and non-linearity of the 3-D kinematics problem, PolyNet achieved an honorable second-place among all contenders. It is again noted that in most cases, testing RMSE was better than training RMSE due to the addition of Gaussian noise to the training outputs in the simulated system. The training RMSEs were computed against pristine training pattern outputs without noise, yielding higher training errors than expected compared to real-world data conditions.

TABLE VIII
Algorithm Accuracy: Average Training and Testing RMSEs per IE Problem

	Czuk DCDC		3-D Kinematics	
Algorithms	Training	Testing	Training	Testing
ANN-SLP	0.0401	**0.0079**	0.0421	**0.0217**
TSK Fuzzy	**0.0387**	0.0170	0.0397	0.0390
I-ELM	0.0493	0.0292	0.1321	0.1288
EI-ELM	0.0495	0.0300	0.1306	0.1264
CI-ELM	0.0483	0.0274	0.1327	0.1289
SVR-RBF	0.0421	0.0170	0.0528	0.0432
PolyNet	*0.0393*	0.0087	**0.0272**	0.0362
PolyStat	*0.0397**	*0.0082*	0.0482†	0.0393
PlyOLSRP	*0.0397***	*0.0081*	0.0456†	0.0471
PlyGLSP	0.0438***	0.0188	0.1357***	0.1297
PlyGLSRP	0.0421††	0.0148	0.1313††	0.1267
PlyPaPGLS	0.0670	0.0509	0.1312	0.1306
PlyPaPGLSR	0.0945	0.0868	0.1411	0.1356

** setscale = 3.3*
*** setscale = 3.6*
† setscale = 1.75
**** setscale = 2.0*
†† setscale = 2.5

5.3 Experiments with Real-World Repository Datasets

Multi-dimensional complex datasets were obtained from the University of California at Irvine repository of machine learning databases [77] for the following experiments. The sources of the data vary widely, and include but are not limited to biological sciences, retail marketing, computer analytics, and manufacturing. The datasets selected are often a mix of continuous and multi-modal discrete data, are highly non-linear, and express many input dimensions. Generally, these datasets are too complex to analyze or model with closed-form methods, and as such are regularly utilized as benchmark datasets in the current literature. Table IX introduces key specifications of the seven datasets used. Once again, each algorithm is run for 20 trials of a 70/30 k-fold validation process. During each trial, each algorithm is fed the same randomized training and testing set as all other algorithms.

Table IX
Benchmark Datasets: Specifications for 70/30 k-fold Testing

Dataset	# Training Vectors	# Testing Vectors	Dimensionality
1 Abalone	2924	1253	8
2 Auto Price	112	47	15
3 Boston Housing	355	151	13
4 California Housing	14448	6192	8
5 Delta Ailerons	4991	2138	5
6 Delta Elevators	6662	2855	6
7 Machine CPU	147	62	6

Optimal results are selected for each algorithm on the basis of minimum testing RMSE obtained for a converging solution. The PLM and I-ELM variants are allowed to compute up to 500 terms/nodes on the way toward convergence. The SVR-RBF algorithm is allowed to compute up to 1,000 nodes. The ANN-SLP is allowed to compute up to 20 nodes per solution network due to the comparatively long training times for the NBN training algorithm. The N-D Fuzzy system computes up to 100,000 stored output table values on its way toward an optimal solution.

For all datasets, the I-ELM variants used an impact factor of 2.7583, and initial RBF centers were chosen randomly from the input domain. For the SVR algorithm and for the four PLM variants that use threshold-based pruning of terms, optimal parameters were discovered and set as indicated in Table X.

Table X
Optimal Parameter Settings per Dataset for SVR and PLM Variants

	SVR		PolyStat	PlyOLSRP	PlyGLSP	PlyGLSRP
Dataset	C	γ	stdscale			
Abalone	2^4	2^{-6}	2.5	2.75	2.0	2.5
Auto Price	2^8	2^{-5}	3.0	2.75	2.75	2.75
Boston Housing	2^4	2^{-3}	2.0	2.5	2.0	2.5
California Housing	2^3	2^1	2.5	2.5	2.0	2.5
Delta Ailerons	2^3	2^{-3}	1.75	2.5	2.0	2.0
Delta Elevators	2^0	2^{-2}	2.0	2.75	2.0	2.5
Machine CPU	2^6	2^{-4}	2.5	2.75	2.75	2.5

Key metrics are plotted for two of the seven datasets. The Boston Housing and Machine CPU datasets provide certain insights into the attributes of the seven PLM variants. The Boston Housing dataset features particularly high input dimensionality (13) which will exercise the algorithms' ability to approach

convergence under such circumstances. The Machine CPU dataset expresses moderate dimensionality, but also mixes bimodal and continuous data and high variance within certain input dimensions.

5.3.1 Training and Testing Times for Real-World Datasets

The training time plots of Figure 42 and Figure 43 exhibit a non-monotonicity for the original PolyNet variant. Though no statistical term pruning is included in this method, terms with zero-value coefficients following one-step linear regression are certainly omitted. Due to this, it is possible to see where the algorithm progress starts to yield excess terms as polynomial order increases, yet new coefficients resolve to zero. As with the IE problems, the original PolyNet is unbeatable for training speed. Also as before, the more complex PLM variants take orders of magnitude longer to solve for equivalent sized networks.

We also note that one of the "probe and prune" algorithms, PlyPaPGLS, computes terms only up to the order found to indicate minimal RMSE during the initial hybrid GLS-RR ridge regression phase. Ideally, overall training time should be reduced as the process truncates unnecessary computation.

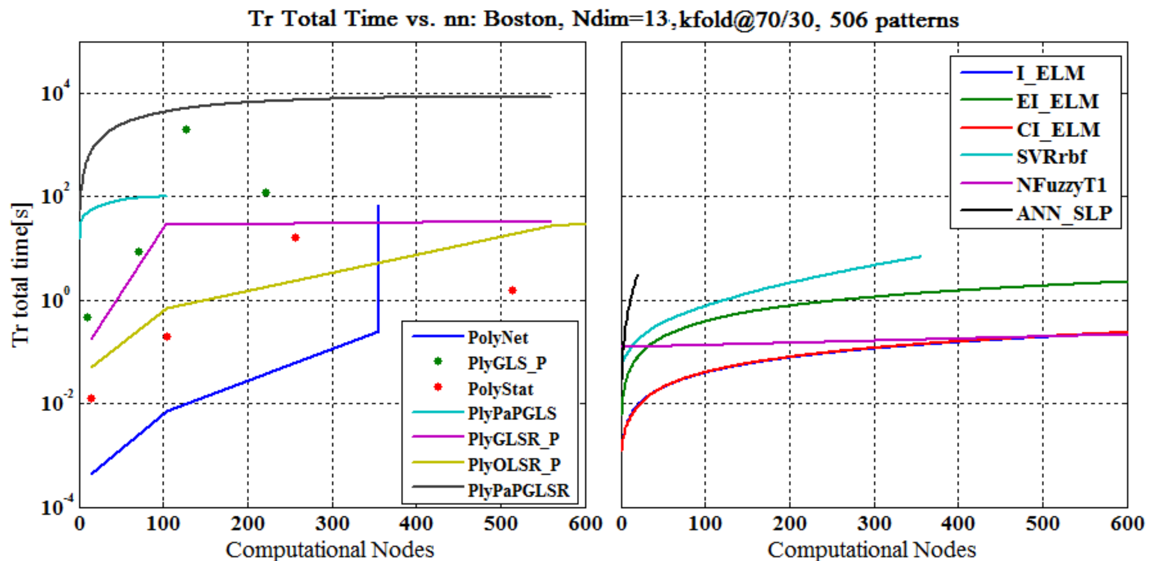


Figure 42 Total training times for all algorithms up to 600 nodes: Boston Housing

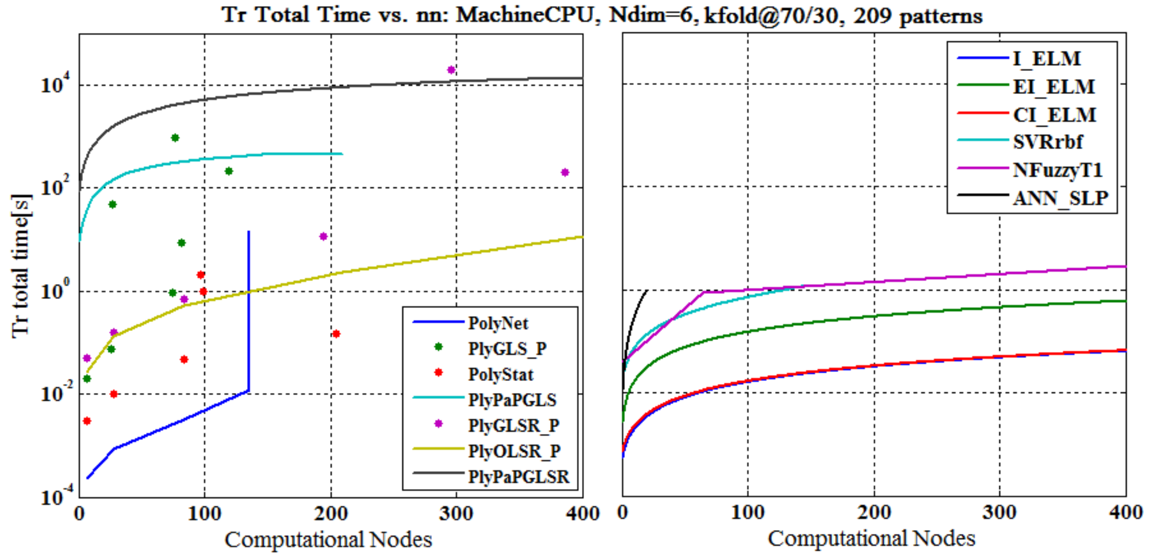


Figure 43 Total training times for all algorithms up to 400 nodes: Machine CPU

All PLM variants again show superior validation time performance for the optimized networks, as seen in Figure 44 and Figure 45. One algorithm, SVR-RBF, produces a more efficient result than PolyNet for networks roughly larger than 200 terms for the Boston Housing dataset. All PLM variants produce more efficient equal-sized networks for the Machine CPU dataset.

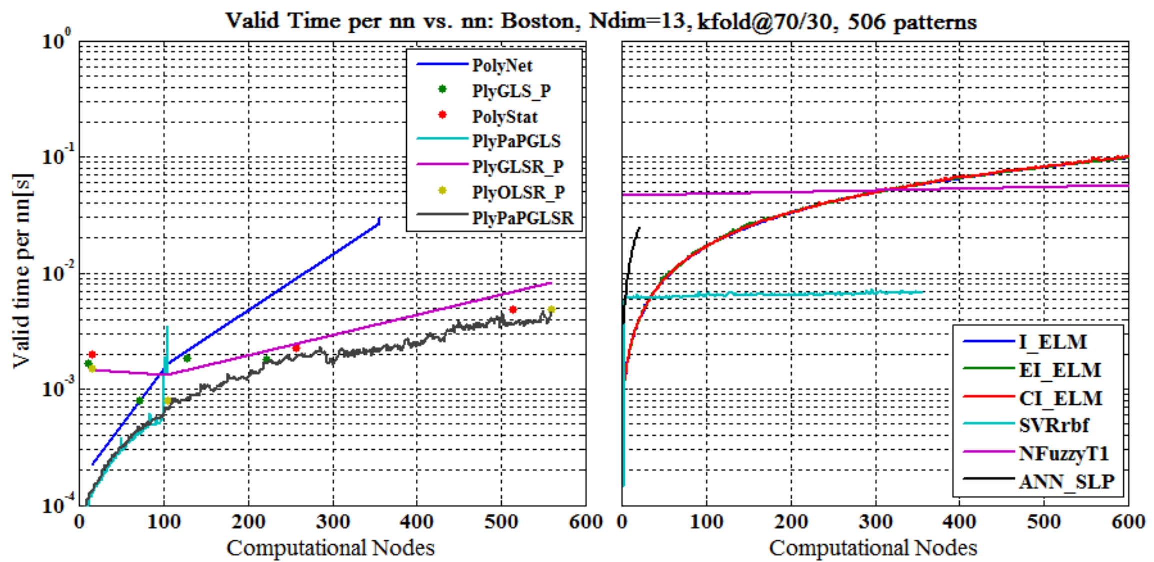


Figure 44 Validation times per network size vs. nodes for all algorithms: Boston Housing

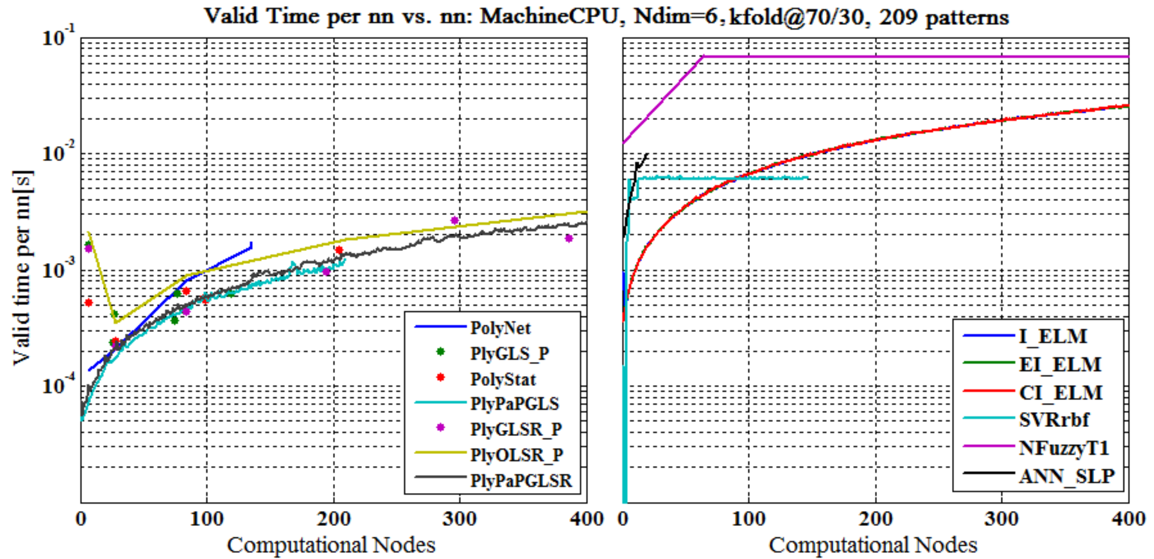


Figure 45 Validation times per network size vs. nodes for all algorithms: Machine CPU

5.3.2 Training and Validation Accuracy for Real-World Datasets

The excellent generalization abilities of all but two of the algorithms are evident while examining the training and validation results of Figure 46 and Figure 47 for the Boston Housing dataset. PolyNet and ANN-SLP exhibit large variance between training and validation error for this dataset. For the single epochs plotted, SVR-RBF and PlyPaPGLSR produce the best testing error. For the 20 x 70/30 k-fold trials, the PlyPaPGLSR algorithm wins more decisively as will be examined.

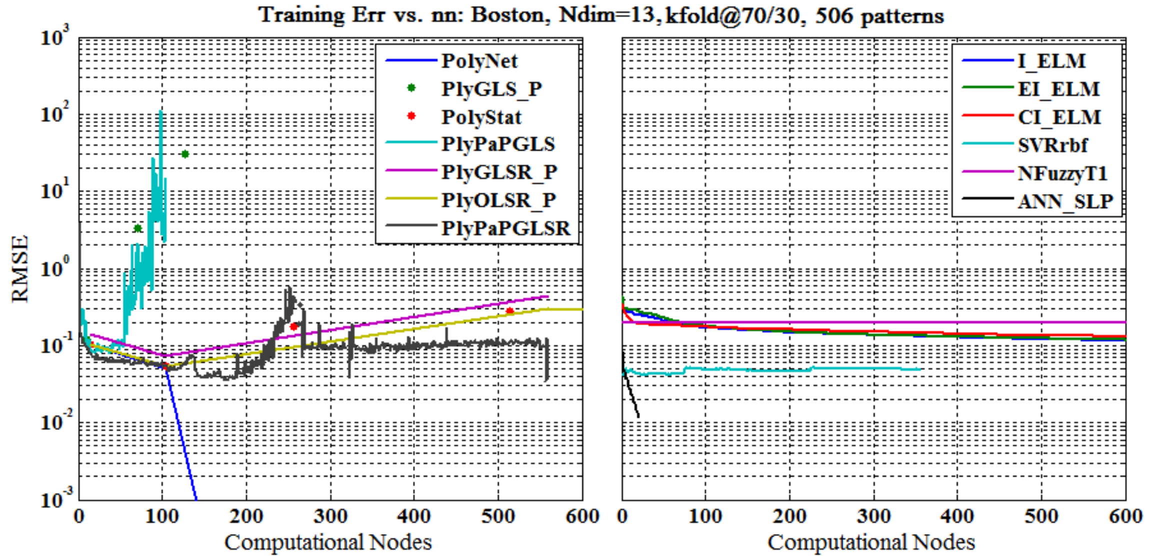


Figure 46 Training error for all algorithms up to 600 nodes: Boston Housing

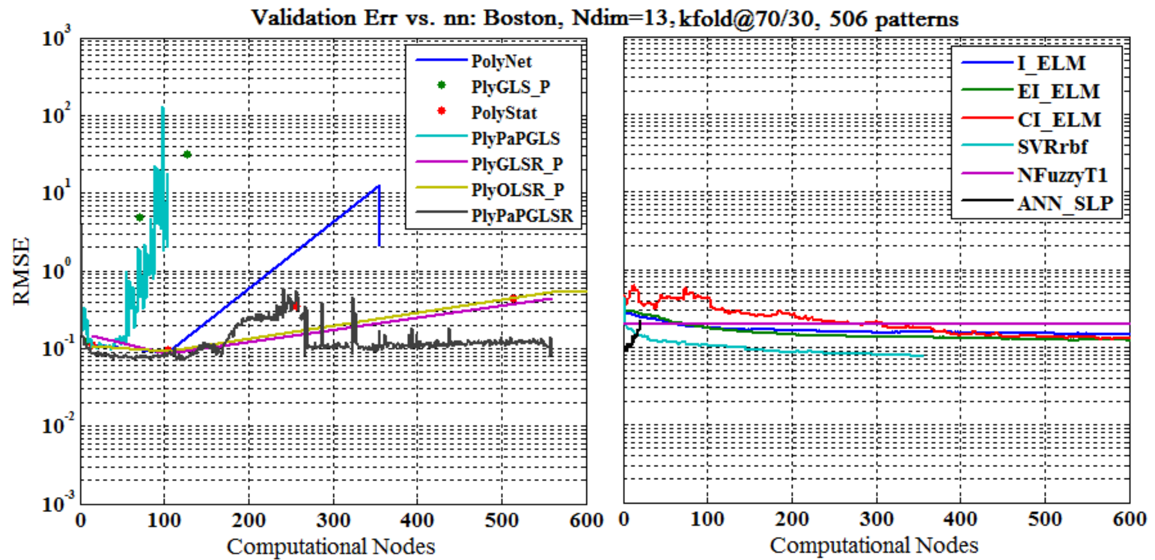


Figure 47 Validation error for all algorithms up to 600 nodes: Boston Housing

Very similar results are seen for the training and validation error plots of Figure 48 and Figure 49. The PlyPaPGLSR algorithm is again the winner by a small but definitive margin, though the optimal solution is not computed until the network grows to over 300 terms. Though the generalization performance of PolyNet and PlyPaPGLS is again poor, those and the remaining PLM variants find a much smaller competitively accurate network in less than 30 nodes for this dataset,

as seen in Figure 49. Given this, it is important to realize that lowest validation RMSE is not necessarily the most important metric for all applications, as a more concise network with slightly reduced error performance may be desirable in many cases.

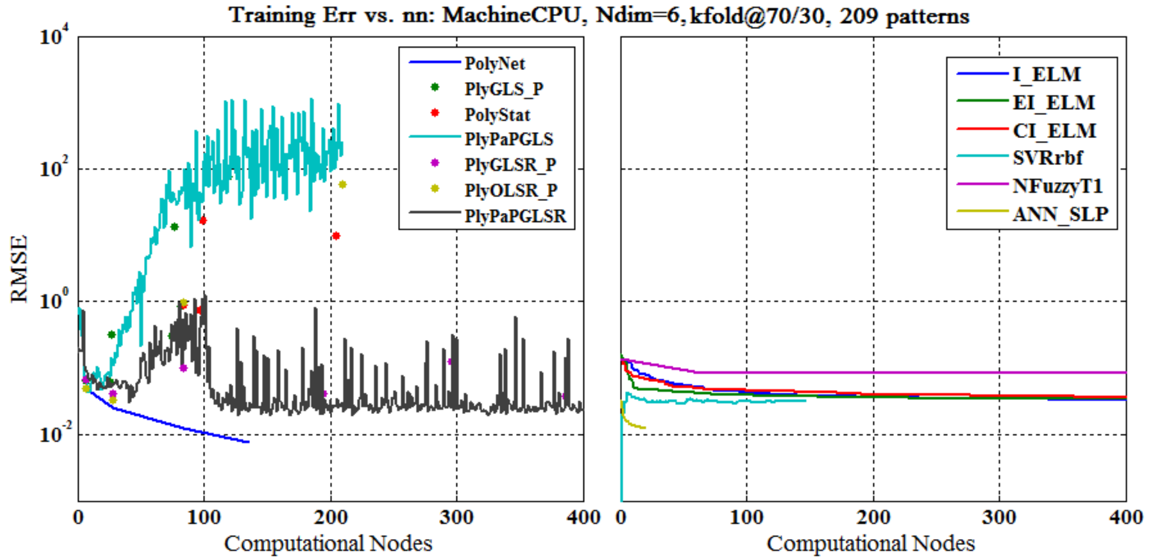


Figure 48 Training error for all algorithms up to 400 nodes: Machine CPU

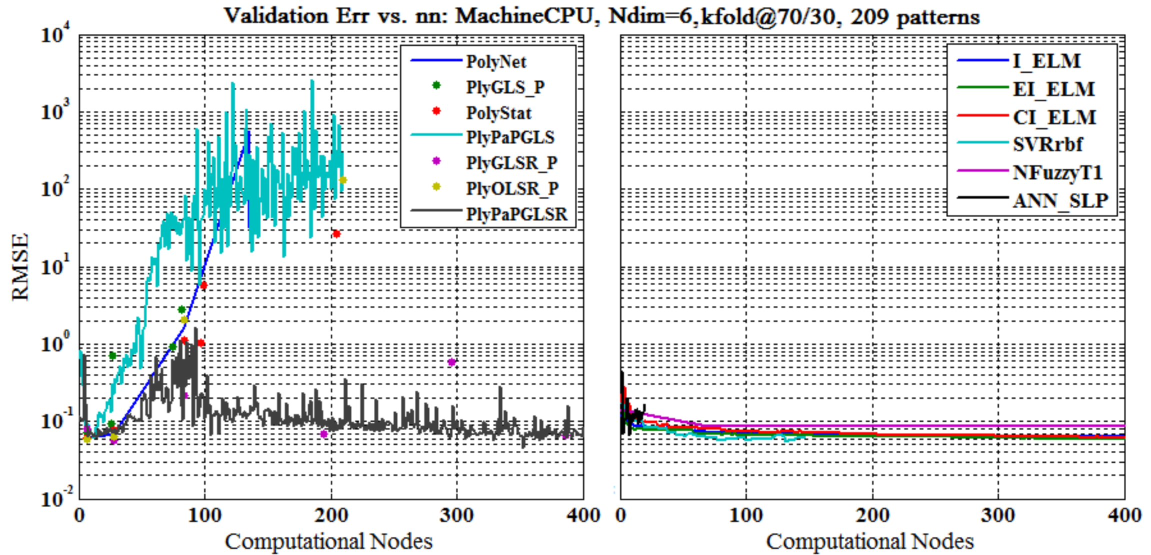


Figure 49 Validation error for all algorithms up to 400 nodes: Machine CPU

5.3.3 Tabulation of Real-World Dataset Results

All seven repository datasets were run using the 20 x 70/30 k-fold process as previously described. Training and validation runtimes and optimized network node count are displayed for all algorithms versus all datasets in Table XI through Table XIII. Per each dataset, the winners are highlighted in green bold, and numeric values close to these ("runner-ups") are highlighted in green italics. Results which show particular inefficiency are highlighted in red. Overall, the PLM variants make a strong showing for the cases of training and validation efficiency. The original PolyNet variant is consistently at or near first place of all algorithms studied in training time. In general, one or more PLM variants place at or near first in validation times for every dataset. Additionally, all PLMs generally score well in final optimized network size, though the ANN-SLP algorithm is in first place in this category for almost every dataset.

Table XI
Average Processing Times and Network Size per Datasets 1-3

	Abalone			Auto Price			Boston Housing		
Algorithms	Training [s]	Testing [s]	Nodes [#]	Training [s]	Testing [s]	Nodes [#]	Training [s]	Testing [s]	Nodes [#]
ANNSLP	0.4787	0.0330	**2**	0.0089	0.0028	**1**	0.2306	0.0076	**5**
TSK Fuzzy	1,868.4	2.0228	65,536	896.32	3.5710	32,768	212.23	1.0103	8,192
I-ELM	0.5990	0.2843	500	0.0203	0.0106	500	0.0822	0.0326	500
EI-ELM	5.9603	0.2903	500	0.1777	0.0105	500	0.6910	0.0330	500
CI-ELM	0.6098	0.2791	500	0.0204	0.0105	500	0.2237	0.0328	500
SVR-RBF	0.2659	0.4454	1,000	0.0294	0.0061	112	0.0461	0.0064	156
PolyNet	**0.0056**	0.0015	18	**0.0005**	**0.0003**	16	**0.0070**	0.0016	100
PolyStat	0.0505	0.0024	13	0.0065	*0.0007*	16	0.1602	*0.0009*	104
PlyOLSRP	8.9766	0.0029	*9*	0.0506	0.0014	16	0.3011	0.0019	13
PlyGLSP	2.3405	0.0018	*9*	0.0398	0.0020	16	0.8112	0.0013	105
PlyGLSRP	62.803	0.0046	163	0.2733	0.0023	16	29.651	0.0010	104
PlyPaPGLS	738.72	**0.0005**	144	0.8337	**0.0003**	*6*	63.706	**0.0002**	80
PlyPaPGLSR	4986.8	0.0022	71	122.72	**0.0003**	85	5522.1	0.0011	391

Table XII
Average Processing Times and Network Size per Datasets 4-5

	California Housing			Delta Ailerons		
Algorithms	Training [s]	Testing [s]	Nodes [#]	Training [s]	Testing [s]	Nodes [#]
ANNSLP	70.936	0.8275	17	18.604	0.2560	**15**
TSK Fuzzy	7,154.6	13.082	65,536	3,752.12	1.7204	100,000

Algorithms	California Housing Training [s]	California Housing Testing [s]	California Housing Nodes [#]	Delta Ailerons Training [s]	Delta Ailerons Testing [s]	Delta Ailerons Nodes [#]
I-ELM	1.8631	1.2970	500	0.5511	0.4466	500
EI-ELM	14.554	1.2650	500	5.3020	0.4852	500
CI-ELM	1.4608	1.2820	500	0.5453	0.4750	500
SVR-RBF	6.0684	0.0777	400	0.2499	0.0203	807
PolyNet	**0.1183**	0.0316	39	**0.0563**	0.0134	56
PolyStat	2.2587	0.0122	45	0.8330	0.0028	56
PlyOLSRP	*0.1222*	0.0023	**9**	0.2936	**0.0007**	21
PlyGLSP	408.34	0.0028	45	174.89	0.0021	60
PlyGLSRP	60.580	0.0120	162	18.705	0.0073	103
PlyPaPGLS	3,591	**0.0011**	30	14,854	*0.0008*	217
PlyPaPGLSR	3,644.4	0.0024	19	53,289	0.0079	24

Table XIII
Average Processing Times and Network Size per Datasets 6-7

Algorithms	Delta Elevators Training [s]	Delta Elevators Testing [s]	Delta Elevators Nodes [#]	Machine CPU Training [s]	Machine CPU Testing [s]	Machine CPU Nodes [#]
ANNSLP	4.5820	0.1413	**5**	0.1944	0.0047	**8**
TSK Fuzzy	485.14	2.8707	15,625	181.46	0.0650	12,388
I-ELM	0.8651	0.6349	500	0.2549	0.0129	500
EI-ELM	8.5861	0.6363	500	0.3078	0.0128	500
CI-ELM	0.8789	0.5946	500	0.0353	0.0129	500
SVR-RBF	1.0234	0.0506	1,000	0.0265	0.0060	133
PolyNet	**0.1395**	0.0385	84	**0.0003**	**0.0001**	**7**
PolyStat	2.1873	0.0062	84	0.0032	*0.0005*	**7**
PlyOLSRP	0.3279	0.0012	28	0.0379	0.0015	10
PlyGLSP	340.28	0.0034	84	0.0320	0.0020	**9**
PlyGLSRP	46.526	0.0100	134	31.368	0.0010	137
PlyPaPGLS	936.88	**0.0006**	*8*	51.495	**0.0001**	196
PlyPaPGLSR	1,303.6	**0.0007**	*9*	14,140	0.0037	314

The tabulated results for training and validation error for all experiments run against all repository datasets are listed in Table XIV and Table XV. Though the ANN-SLP algorithm prevails for four of the seven datasets overall, several of the PLM variants have validation errors almost as low for three of those datasets (Abalone, Delta Ailerons, and Delta Elevators). In these cases, there is negligible difference in the final validation errors among PolyNet, PolyStat, PlyGLSP, and PlyGLSRP. For this reason, since the PolyNet algorithm is significantly simpler and more computationally efficient than the others, it is possible to say that the PolyNet algorithm is the best overall choice in the case of the particular datasets run.

The PlyPaPGLSR variant prevails in the remaining three datasets (Auto Price, Boston Housing, Machine CPU) not won by ANN-SLP or SVR-RBF. It is notable that two of these datasets exhibit particularly high dimensionality (Auto Price (15) and Boston Housing (13)). It is arguable that the enhanced generalization ability and incremental term pruning afforded by PlyPaPGLSR is ideal for high dimensional data. Additionally, this variant takes up the slack particularly where the remaining PLM variants did not place relatively near ANN-SLP. In fact, viewing the results overall, it appears that four of the earlier variants taken together – PolyNet, PolyStat, PlyGLSP, and PlyGLSRP – form a perfect performance compliment to PlyPaPGLSR. If an experimenter is unsure which algorithm to run for a particular dataset, it appears that excellent coverage could be had by running both PolyNet (ignoring the other similarly scoring but more computationally complex variants) and PlyPaPGLSR.

All observations regarding the performance of the new PLM variants tested versus other prominent algorithms can be summarized as the following:

- The PLM variants score near or better than other algorithms for a variety of datasets, and are usually more computationally efficient for final network implementation.
- Two variants, PlyOLSRP and PlyPaPGLS, are negligible in performance compared alongside other PLM variants, and are therefore expendable.
- In certain dataset experiments, the PolyNet algorithm appears to match the accuracy of other prominent ML methods such as ANN-SLP and SVR-RBF. For these same experiments, three other variants, PlyGLSP, PlyGLSRP, and PolyStat, score almost identically or not significantly better than the PolyNet algorithm. Additionally, those three variants are significantly more compute intensive during training. For these reasons, PolyStat, PlyGLSP, and PlyGLSRP can be considered redundant, and are therefore expendable.
- For dataset cases where the PolyNet variant performs significantly worse than other methods, such as for datasets with high dimensionality, the PlyPaPGLSR algorithm happens to perform exceptionally well.

- Taken altogether, two PLM variants, PolyNet and PlyPaPGLSR, arguably provide full coverage when both are applied to any variety of datasets.

Table XIV
Average Training and Testing RMSEs per Datasets 1-4

	Abalone		Auto Price		Boston Housing		California Housing	
Algorithms	Training	Testing	Training	Testing	Training	Testing	Training	Testing
ANN-SLP	**0.0731**	**0.0734**	0.0618	0.0879	**0.0409**	0.0898	0.1085	**0.1109**
TSK Fuzzy	0.0883	0.0892	0.1678	0.1696	0.1617	0.1641	0.1648	0.1653
I-ELM	0.0922	0.0938	0.1184	0.1222	0.1359	0.1261	0.1649	0.1691
EI-ELM	0.0924	0.0829	0.1107	0.1139	0.1130	0.1077	0.1669	0.1503
CI-ELM	0.0837	0.0845	0.1168	0.1197	0.1162	0.1423	0.1648	0.1756
SVR-RBF	0.0756	0.0786	**0.0395**	0.0935	0.0468	0.0925	**0.0845**	0.1413
PolyNet	0.0772	0.0789	0.0728	0.0918	0.0530	0.0890	0.1342	0.1392
PolyStat	0.0782	0.0788	0.0729	0.0917	0.0551	0.0940	0.1317	0.1409
PlyOLSRP	0.0894	0.0919	0.0822	0.0977	0.1205	0.1263	0.2018	0.2017
PlyGLSP	0.0785	0.0788	0.0730	0.0923	0.0549	0.0926	0.1304	0.1406
PlyGLSRP	0.0842	0.0848	0.0908	0.0986	0.0739	0.0855	0.1487	0.1488
PlyPaPGLS	0.1150	0.0865	0.0782	0.0983	0.0826	0.0959	0.1641	0.1495
PlyPaPGLSR	0.0804	0.0827	0.0547	**0.0707**	0.0554	**0.0717**	0.1498	0.1519

Table XV
Average Training and Testing RMSEs per Datasets 5-7

	Delta Ailerons		Delta Elevators		Machine CPU	
Algorithms	Training	Testing	Training	Testing	Training	Testing
ANN-SLP	**0.0355**	**0.0378**	0.0518	**0.0527**	**0.0144**	0.0673
TSK Fuzzy	0.0382	0.0405	0.0556	0.0565	0.0725	0.0793
I-ELM	0.0518	0.0521	0.0698	0.0632	0.0363	0.0674
EI-ELM	0.0519	0.0516	0.0659	0.0575	0.0358	0.0554
CI-ELM	0.0423	0.0555	0.0555	0.0566	0.0400	0.0675
SVR-RBF	0.0376	0.0467	**0.0377**	0.0603	0.0316	0.0539
PolyNet	0.0375	0.0387	0.0521	0.0530	0.0494	0.0606
PolyStat	0.0375	0.0386	0.0521	0.0531	0.0499	0.0606
PlyOLSRP	0.0406	0.0410	0.0555	0.0559	0.0616	0.0725
PlyGLSP	0.0375	0.0387	0.0522	0.0530	0.0490	0.0589
PlyGLSRP	0.0395	0.0397	0.0551	0.0554	0.0421	0.0585
PlyPaPGLS	0.0394	0.0407	0.0539	0.0546	0.0922	0.0611
PlyPaPGLSR	0.0391	0.0408	0.0547	0.0556	0.0289	**0.0449**

Chapter 6
Conclusions and Future Work

Following experimentation with a variety of datasets, it is apparent that results for all such algorithms studied can be similarly varied. The selection of one learning machine over another is in practice a highly subjective decision based on the particular intended application. Still, an attempt will be made to summarize the work herein, and to quantify the overall results with an original figure of merit scheme for ML algorithms.

6.1 Evaluation of PLM Variants and Competing Methods with an Original Figure of Merit Scheme

In order to address final comparisons among all algorithms tested, an original Figure of Merit (FOM) scheme is proposed. It is acknowledged that the imposition of such a FOM is itself a highly subjective exercise. This attempt is deemed appropriate for the dataset problems presented throughout this study.

A set of equations is proposed which taken together, appropriately penalize the key metrics on a logarithmic scale. Such quantities can be computed directly from resultant experimental data and individually resolve to a range [1:0], with 1 representing perfect performance for the parameter, and 0 representing utter failure. Five such equations are seen in (70) below. The figure of merit component for training time is defined as fom_{Ttime}. Likewise, the component for validation time is fom_{Vtime}. The component for the final number of nodes for an optimized network is $fom_{\#nodes}$. The component for validation error is fom_{Verr}. Validation error is usually considered more significant than training error by itself. However, the generalization ability of an algorithm is significant throughout this field and throughout this work. As such, a final component for generalization ability is defined as fom_{Gen}, and contains the ratio of best validation error over best training

error for the same resultant network as: $\frac{RMSE_V}{RMSE_T}$. In this way, five critical metrics are expressible as normalized quantities.

$$fom_{Ttime} = \begin{cases} 1, & t < 1ms \\ \frac{1}{12}\log\left(\frac{t + 1e9}{t}\right), & t \geq 1ms \end{cases}$$

$$fom_{Vtime} = \begin{cases} 1, & t < 0.1ms \\ \frac{1}{6}\log\left(\frac{t + 100}{t}\right), & t \geq 0.1ms \end{cases}$$

$$fom_{Verr} = \begin{cases} 1, & t < 0.001 \\ \frac{1}{5}\log\left(\frac{RMSE_V + 100}{RMSE_V}\right), & t \geq 0.001 \end{cases} \tag{70}$$

$$fom_{\#nodes} = \begin{cases} 1, & t \leq 10 \\ \frac{1}{3}\log\left(\frac{n + 10000}{n}\right), & t > 10 \end{cases}$$

$$fom_{Gen} = \begin{cases} 1, & RMSE_V \leq RMSE_T \\ \frac{1}{1.0414}\log\left(\frac{\frac{RMSE_V}{RMSE_T} + 10}{\frac{RMSE_V}{RMSE_T}}\right), & RMSE_V > RMSE_T \end{cases}$$

From this basis of equations, one can further derive separate overall FOMs for different applications. For this study, two such FOMs are proposed which apply different weighting ratios to the five components of (70), resulting in final quantities which express an overall figure of merit also in the range [1:0]. Equation (71) introduces a real-time FOM as FOM_{RT}. Such a rating considers final validation error as the most important metric, but also allots sufficient weighting factors to the time metrics for applications where training and validation times are more critical. Similarly, a FOM is introduced in (72) for “offline” implementations, where final validation error is weighted more heavily, and time metrics are deemphasized.

$$\begin{aligned} FOM_{RT} = (0.5)fom_{Verr} + (0.1)fom_{GEN} + (0.15)fom_{Ttime} \\ + (0.15)fom_{Vtime} + (0.1)fom_{\#nodes} \end{aligned} \tag{71}$$

$$FOM_{OL} = (0.75)fom_{Verr} + (0.1)fom_{GEN} + (0.05)fom_{Ttime} + (0.05)fom_{Vtime} + (0.05)fom_{\#nodes} \quad (72)$$

The real-time FOM, FOM_{RT}, was computed for each algorithm based on its performance against each dataset individually. The results are shown in Table XVI below. Winners and runner-ups are highlighted in green bold, and problematic finishers are highlighted in red. Based on these scorings, for applications where compute times are crucial, the PolyNet, PolyStat, and PlyGLSP algorithms score prominently for both IE and general repository dataset problems. In contrast, the FS algorithm scores significantly poorly for large datasets with greater than 3 dimensions. Though the PlyPaPGLSR algorithm scored well in error performance relative to other algorithms, its success is offset by its exceptionally long training times. As such, it might be rejected for hardware or time-critical applications.

Table XVI
FOM_{RT}: Real-Time Figure of Merit of Each Algorithm per Dataset

	ANN-SLP	Fuzzy	I-ELM	EI-ELM	CI-ELM	SVR-RBF	PolyNet
DCDC	0.8022	0.6820	0.6391	**0.6235**	0.6466	0.6998	**0.8331**
3-D Kinematics	**0.7259**	0.5887	**0.5743**	**0.5653**	**0.5745**	0.6574	**0.7475**
Abalone	0.7168	**0.5209**	0.6252	0.6185	0.6302	0.6472	**0.7620**
Auto-Price	0.7446	**0.4927**	0.6673	0.6587	0.6685	0.6740	**0.7797**
Boston Housing	0.6996	**0.5230**	0.6473	0.6424	0.6289	0.6731	**0.7120**
California Housing	0.6286	**0.4679**	0.5768	0.5720	0.5752	0.5937	0.6763
Delta Ailerons	0.6955	**0.5504**	0.6468	0.6341	0.6335	0.6749	0.7401
Delta Elevators	0.7027	**0.5491**	0.6323	0.6239	0.6370	0.6341	0.7048
Machine CPU	0.6946	**0.5793**	0.6558	0.6698	0.6698	0.7076	**0.8176**

	PolyStat	PlyGLS_P	PlyOLSR_P	PlyGLSR_P	PlyPaPGLS	PlyPaPGLSR
DCDC	**0.8234**	0.8043	**0.8189**	0.7802	0.6842	0.6547
3-D Kinematics	**0.7204**	**0.7357**	0.6993	0.6949	0.6600	0.6168
Abalone	**0.7506**	0.7169	0.7365	0.6652	0.6763	0.6615
Auto-Price	**0.7580**	0.7375	0.7355	0.7263	0.7448	0.7003
Boston Housing	0.6984	**0.7215**	0.6861	0.6865	0.6992	0.6421
California Housing	0.6669	**0.7095**	0.6545	0.6310	0.6590	0.6557
Delta Ailerons	0.7426	**0.7759**	0.7158	0.7064	0.6812	0.6807
Delta Elevators	0.7096	**0.7519**	0.6890	0.6798	**0.7315**	0.7273
Machine CPU	**0.7910**	0.7589	0.7646	0.6924	0.7177	0.6405

The offline FOM, FOM_{OL}, was computed for each algorithm based on its performance against each dataset individually. The results are shown in Table XVII

below. In this case, where final process accuracy is weighted much more heavily than compute times, PolyNet and PolyStat once again emerge as strong contenders. Also in this scenario, the ANN-SLP algorithm is properly rewarded for its superior accuracy. The other PLM variants score occasional win or runner-up berths, but none are as consistent as PolyNet and ANN-SLP. Again, the PlyPaPGLSR variant is somewhat neutralized by its prohibitive training times. Also, the FS algorithm is a last-place finisher for almost all datasets.

Table XVII

FOM_{OL}: Offline Figure of Merit of Each Algorithm per Dataset

	ANN-SLP	Fuzzy	I-ELM	EI-ELM	CI-ELM	SVR-RBF	PolyNet
DCDC	0.8294	0.7421	0.6979	0.6913	0.7037	0.7457	0.8301
3-D Kinematics	0.7510	0.6622	0.6011	0.5991	0.6012	0.6842	0.7219
Abalone	0.6878	0.5962	0.6353	0.6397	0.6424	0.6495	0.6974
Auto-Price	0.6794	0.5546	0.6355	0.6363	0.6372	0.6349	0.6907
Boston Housing	0.6530	0.5675	0.6281	0.6345	0.6108	0.6385	0.6583
California Housing	0.6358	0.5475	0.5891	0.5941	0.5857	0.5923	0.6379
Delta Ailerons	0.7119	0.6446	0.6726	0.6689	0.6583	0.6811	0.7232
Delta Elevators	0.6996	0.6294	0.6581	0.6601	0.6647	0.6479	0.6949
Machine CPU	0.6501	0.6208	0.6475	0.6665	0.6543	0.6812	0.7262

	PolyStat	PlyGLS_P	PlyOLSR_P	PlyGLSR_P	PlyPaPGLS	PlyPaPGLSR
DCDC	0.8302	0.7836	0.8292	0.7854	0.6866	0.6588
3-D Kinematics	0.7156	0.6638	0.6973	0.6479	0.6308	0.6188
Abalone	0.6948	0.6759	0.6909	0.6566	0.6597	0.6581
Auto-Price	0.6836	0.6749	0.6756	0.6730	0.6768	0.6728
Boston Housing	0.6506	0.6603	0.6475	0.6609	0.6598	0.6491
California Housing	0.6330	0.6344	0.6288	0.6169	0.6301	0.6289
Delta Ailerons	0.7242	0.7351	0.7149	0.7100	0.6978	0.7025
Delta Elevators	0.6964	0.7109	0.6897	0.6835	0.7075	0.7051
Machine CPU	0.7176	0.6986	0.7104	0.6768	0.6904	0.6682

Equations (71) and (72) can be applied more generally in an attempt to compare the average performance of each algorithm against all others for all experiments in this study. Each individual component of (70) was computed as an average value of each associated metric across all nine datasets in this study. Those average component values were then applied to the FOM_{RT} and FOM_{OL} equations to attempt to derive an overall comparative performance evaluation of each algorithm. Table XVIII shows the results averaged from all dataset results. For the real-time

evaluation, PolyNet, PolyStat, and PlyGLSP emerge as the clear winners. The FS suffers in the overall evaluation as expected. It should be noted that in practice, fuzzy systems are still among the easiest and most straightforward to implement in hardware applications and therefore should not be underestimated. For the overall offline scores, the field is much less separated since all algorithms tested exhibit excellent accuracy performance. Even with the smaller margins, PolyNet, PolyStat, and PlyGLSP once again prevail. The ANN-SLP is rewarded in this evaluation as well.

Table XVIII
Overall Real-time and Offline FOMs for All Algorithms Tested

	ANN-SLP	Fuzzy	I-ELM	EI-ELM	CI-ELM	SVR-RBF	PolyNet
FOMrt	0.7123	0.5505	0.6294	0.6232	0.6293	0.6624	0.7526
FOMol	0.6998	0.6183	0.6406	0.6434	0.6398	0.6617	0.7090

	PolyStat	PlyGLS_P	PlyOLSR_P	PlyGLSR_P	PlyPaPGLS	PlyPaPGLSR
FOMrt	0.7401	0.7458	0.7222	0.6958	0.6949	0.6644
FOMol	0.7051	0.6930	0.6983	0.6790	0.6710	0.6625

6.2 Summary Statement

In summary, this author feels that the case has been sufficiently made for the viability of renewed interest and research in polynomial based learning machines throughout computational intelligence. The performance of the algorithms developed in this work is near or better than that of several prominent methods in use today for several accepted benchmark problems. Additionally, new methods introduced for efficient generation of polynomial terms places PLMs at the top of the field for computational performance. It is hoped that other researchers will recognize these findings and proceed with ongoing development of new classes of polynomial based learning machines that are optimal for ongoing and future applications throughout machine learning and data mining fields.

6.3 Future Work

Much work remains to be done to make PLMs even more viable for deployment throughout science, industry, and business. Long runtimes for some of

the variants presented may betray exceptional performance in terms of convergence, accuracy, and generalization. Additionally, the response of the PLMs presented and of the existing methods tested is somewhat unpredictable from dataset to dataset. There is room for improvement in both direct development of polynomial based learning machine algorithms, and in the understanding and classification of particular datasets for use with PLMs.

6.3.1 Improved Coefficient Term Analysis and Pruning

The PLM variants that utilize term pruning based on noise detected in the coefficients can be greatly improved. Term pruning according to thresholds of coefficient noise is promising, but currently requires initial manual trials in order to discover the optimal setting of the *setscale* threshold. This deviates from the desire to deploy "run-and-done" algorithms that are completely operationally autonomous. In contrast, the PLM variants that use the incremental "probe-and-prune" method are operationally autonomous, but require exceptionally long runtimes due to the application of three distinct iterative processes.

The probe phase appears to work occasionally as a way to initially discern a maximum term order for the processes that follow. This method can be further explored, essentially probing each dimension of training data separately in order to define and limit monomial degrees for those variables, and to further improve the generalization transparency of the network looking from training to validation. Instead of either setting a noise threshold for coefficient variation during iterative training (PolyStat family), or creating all terms at once and removing noisy performers one-at–a-time in reverse (Probe-and-Prune family), clustering methods could be employed to track which coefficients move together in response to incoming training vectors. Continuing in this regard, coefficient groups that cluster together can be evaluated for term order, perhaps favoring and flagging lower-order terms in the grouping. In this way, "noisy" and/or high-order terms can be identified appropriately. Subsequent decision processes can then excise and replace these terms, observing and responding to intermediate errors.

6.3.2 Process Pipelining

Not just for PLMs, but for all learning machines, methods of data visualization and "cold-start" analysis can be explored to discern the ideal type of learning machine to deploy for each dataset. Recall that the PolyNet and PlyPaPGLSR variants complement each other for performance on various datasets. There are opportunities to develop analysis methods that can pipeline datasets or data streams to one process or the other in an automated fashion. Discrete or modal data are arguably distinguishable from continuous data and can be tabulated before selection of a particular algorithm. Datasets that express high variance in one or more parameters can be classified as well.

6.3.3 Development of a Universal Polynomial Spline Learning Machine

The PLMs introduced in this study generally spawn all generated monomial terms at the coordinate origin of the multi-dimensional space that is represented in the dataset. In order to cancel unwanted features in the *interpolated* output space of such polynomial-based systems, more, higher-order monomial terms must be heaped on at the origin of the system in order to produce desired cancellations. Additionally, such polynomial-based systems do poorly at *extrapolating* new output values outside of the trained data boundaries as polynomial functions explode beyond those boundaries. This makes such polynomial systems initially inappropriate for predictive systems. More fundamentally, the absence of localization leads to the requirement of higher order terms in the solution than is suggested by the actual local gradients of the data.

In the study of polynomial kernels, the overfitting problem is well-known and is addressed in various ways by other historical methods. Learning machines such as RBF-based and sigmoidal neural networks have the advantage that computational nodes can serve to gate each other, essentially localizing the effect of one node or group of nodes in the learning space of the dataset. Following the idea of Banfer and Nelles [35], a hybrid polynomial based network can be created that comprises a superset of much smaller, lower-order subset polynomial networks of

the kind introduced in this study. However, this hybrid network could have a layer of gating functions which localize the range of each subset network over the data space. Each subset network would still be a single-layer network, trainable by the methods introduced throughout this study. New algorithms must be developed which probe the data space and locate local minima and maxima, flagging them as subset network coordinate centers. The combined network, though conceptually more complex, would result in a much smaller network in terms of node terms. Additionally, each subset could be trained by the simplest OLS method used by PolyNet, for example.

6.3.4 Using Video Card GPUs for Machine Learning Computation

There is mounting use in the machine learning community of video card Graphics Processing Unit (GPU) hardware for machine learning processes [90][91]. The built-in matrix computation ability of GPUs is ideal for rapid and accurate computation of the matrix operations encountered during machine learning processes. Such matrix processing power would be ideal for all of the iterative regression processes presented in this study.

Reference Pages

References

[1] Zhengyu Lin, Jiabin Wang, and D. Howe, "A Learning Feed-Forward Current Controller for Linear Reciprocating Vapor Compressors," *IEEE Trans. Ind. Electron.*, vol. 58, no. 8, pp. 3383–3390, Aug. 2011.

[2] A. Bhattacharya and C. Chakraborty, "A Shunt Active Power Filter With Enhanced Performance Using ANN-Based Predictive and Adaptive Controllers," *Ind. Electron. IEEE Trans. On*, vol. 58, no. 2, pp. 421 –428, Feb. 2011.

[3] V. N. Ghate and S. V. Dudul, "Cascade Neural-Network-Based Fault Classifier for Three-Phase Induction Motor," *IEEE Trans. Ind. Electron.*, vol. 58, no. 5, pp. 1555–1563, May 2011.

[4] T. Orlowska-Kowalska and M. Kaminski, "FPGA Implementation of the Multilayer Neural Network for the Speed Estimation of the Two-Mass Drive System," *IEEE Trans. Ind. Inform.*, vol. 7, no. 3, pp. 436–445, Aug. 2011.

[5] Rong-Jong Wai and Chun-Yu Lin, "Dual Active Low-Frequency Ripple Control for Clean-Energy Power-Conditioning Mechanism," *IEEE Trans. Ind. Electron.*, vol. 58, no. 11, pp. 5172–5185, Nov. 2011.

[6] Yu Chen, Xuejun Pei, Songsong Nie, and Yong Kang, "Monitoring and Diagnosis for the DC–DC Converter Using the Magnetic Near Field Waveform," *IEEE Trans. Ind. Electron.*, vol. 58, no. 5, pp. 1634–1647, May 2011.

[7] G. W. Chang, Cheng-I Chen, and Yu-Feng Teng, "Radial-Basis-Function-Based Neural Network for Harmonic Detection," *IEEE Trans. Ind. Electron.*, vol. 57, no. 6, pp. 2171–2179, Jun. 2010.

[8] M. Charkhgard and M. Farrokhi, "State-of-Charge Estimation for Lithium-Ion Batteries Using Neural Networks and EKF," *IEEE Trans. Ind. Electron.*, vol. 57, no. 12, pp. 4178–4187, Dec. 2010.

[9] C. He, C. Liu, Y. Li, and J. Tao, "Intelligent gear fault detection based on relevance vector machine with variance radial basis function kernel," in *2010 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*, 2010, pp. 785 –789.

[10] L. J. Mpanza and T. Marwala, “Artificial neural network and rough set for HV bushings condition monitoring,” in *2011 15th IEEE International Conference on Intelligent Engineering Systems (INES)*, 2011, pp. 109 –113.

[11] V. Machado, A. Neto, and J. D. de Melo, “A Neural Network Multiagent Architecture Applied to Industrial Networks for Dynamic Allocation of Control Strategies Using Standard Function Blocks,” *IEEE Trans. Ind. Electron.*, vol. 57, no. 5, pp. 1823–1834, May 2010.

[12] H.-G. Han, Q. Chen, and J.-F. Qiao, “An efficient self-organizing RBF neural network for water quality prediction,” *Neural Netw.*, vol. 24, no. 7, pp. 717–725, Sep. 2011.

[13] Y.-H. Pao, *Adaptive pattern recognition and neural networks*. Addison-Wesley, 1989.

[14] M. Manic and B. Wilamowski, “Robust neural network training using partial gradient probing,” in *IEEE International Conference on Industrial Informatics, 2003. INDIN 2003. Proceedings*, 2003, pp. 175 – 180.

[15] A. G. Ivakhnenko, “Polynomial Theory of Complex Systems,” *IEEE Trans. Syst. Man Cybern.*, vol. SMC-1, no. 4, pp. 364 –378, Oct. 1971.

[16] E. W. Weisstein, “Taylor Series -- from Wolfram MathWorld.” [Online]. Available: http://mathworld.wolfram.com/TaylorSeries.html. [Accessed: 29-Sep-2012].

[17] M. Stinchcombe and H. White, “Universal approximation using feedforward networks with non-sigmoid hidden layer activation functions,” in *, International Joint Conference on Neural Networks, 1989. IJCNN*, 1989, pp. 613 – 617 vol.1.

[18] M.-S. Chen and M. T. Manry, “Conventional modeling of the multilayer perceptron using polynomial basis functions,” *IEEE Trans. Neural Netw.*, vol. 4, no. 1, pp. 164 –166, Jan. 1993.

[19] D. L. Elliott, “A Better Activation Function for Artificial Neural Networks,” 1993.

[20] H.-T. Yang and Y.-C. Huang, “Intelligent decision support for diagnosis of incipient transformer faults using self-organizing polynomial networks,” *IEEE Trans. Power Syst.*, vol. 13, no. 3, pp. 946–952, 1998.

[21] S.-K. Oh, W. Pedrycz, and B.-J. Park, “Polynomial neural networks architecture: analysis and design,” *Comput. Electr. Eng.*, vol. 29, no. 6, pp. 703–725, Aug. 2003.

[22] G. Jēkabsons and J. Lavendels, “A Heuristic Approach for Surrogate Modelling of Electro-Technical Systems,” in *publication.editionName*, 2008, pp. 62–67.

[23] G. Jekabsons, “Adaptive Basis Function Construction: An Approach for Adaptive Building of Sparse Polynomial Regression Models,” in *Machine Learning*, Y. Zhang, Ed. InTech, 2010, pp. 127–156.

[24] S. J. Russell, P. Norvig, and E. Davis, *Artificial intelligence: a modern approach*. Upper Saddle River, NJ: Prentice Hall, 2010.

[25] H. Akaike, “A new look at the statistical model identification,” *IEEE Trans. Autom. Control*, vol. 19, no. 6, pp. 716–723, 1974.

[26] N. Y. Nikolaev and H. Iba, *Adaptive Learning of Polynomial Networks: Genetic Programming, Backpropagation and Bayesian Methods*. Springer, 2006.

[27] S.-K. Oh and W. Pedrycz, “Self-organizing polynomial neural networks based on PNs or FPNs: Analysis and design,” *Fuzzy Sets Syst*, vol. 144, no. 2, pp. 365–366, 2004.

[28] S.-K. Oh and W. Pedrycz, “Fuzzy Polynomial Neuron-Based Self-Organizing Neural Networks,” *Int. J. Gen. Syst.*, vol. 32, no. 3, pp. 237–250, 2003.

[29] S.-K. Oh, W. Pedrycz, and T.-C. Ahn, “Self-organizing neural networks with fuzzy polynomial neurons,” *Appl. Soft Comput.*, vol. 2, no. 1, pp. 1–10, Aug. 2002.

[30] S.-K. Oh, W. Pedrycz, and H.-S. Park, “A New Approach to the Development of Genetically Optimized Multilayer Fuzzy Polynomial Neural Networks,” *IEEE Trans. Ind. Electron.*, vol. 53, no. 4, pp. 1309–1321, 2006.

[31] M. C. Mackey and L. Glass, “Oscillation and chaos in physiological control systems,” *Science*, vol. 197, no. 4300, pp. 287–289, Jul. 1977.

[32] S.-B. Roh, W. Pedrycz, and S.-K. Oh, “Genetic Optimization of Fuzzy Polynomial Neural Networks,” *IEEE Trans. Ind. Electron.*, vol. 54, no. 4, pp. 2219–2238, 2007.

[33] L. A. Zadeh, “Toward a theory of fuzzy information granulation and its centrality in human reasoning and fuzzy logic,” *Fuzzy Sets Syst.*, vol. 90, no. 2, pp. 111–127, Sep. 1997.

[34] J. C. Bezdek, R. Ehrlich, and W. Full, “FCM: The fuzzy c-means clustering algorithm,” *Comput. Geosci.*, vol. 10, no. 2–3, pp. 191–203, 1984.

[35] O. Banfer and O. Nelles, “Polynomial model tree (POLYMOT) - A new training algorithm for local model networks with higher degree polynomials,” in *IEEE*

International Conference on Control and Automation, 2009. ICCA 2009, 2009, pp. 1571–1576.

[36] O. Nelles, S. Sinsel, and R. Isermann, "Local basis function networks for identification of a turbocharger," in *Control '96, UKACC International Conference on (Conf. Publ. No. 427)*, 1996, vol. 1, pp. 7–12 vol.1.

[37] E. W. Weisstein, "Symmetric Polynomial -- from Wolfram MathWorld." [Online]. Available: http://mathworld.wolfram.com/SymmetricPolynomial.html. [Accessed: 02-Oct-2012].

[38] Q. N. Le and J.-W. Jeon, "Neural-Network-Based Low-Speed-Damping Controller for Stepper Motor With an FPGA," *IEEE Trans. Ind. Electron.*, vol. 57, no. 9, pp. 3167–3180, 2010.

[39] Changliang Xia, Chen Guo, and Tingna Shi, "A Neural-Network-Identifier and Fuzzy-Controller-Based Algorithm for Dynamic Decoupling Control of Permanent-Magnet Spherical Motor," *IEEE Trans. Ind. Electron.*, vol. 57, no. 8, pp. 2868–2878, Aug. 2010.

[40] Ching-Chih Tsai, Hsu-Chih Huang, and Shui-Chun Lin, "Adaptive Neural Network Control of a Self-Balancing Two-Wheeled Scooter," *IEEE Trans. Ind. Electron.*, vol. 57, no. 4, pp. 1420–1428, Apr. 2010.

[41] Chia-Feng Juang, Yu-Cheng Chang, and Che-Meng Hsiao, "Evolving Gaits of a Hexapod Robot by Recurrent Neural Networks With Symbiotic Species-Based Particle Swarm Optimization," *IEEE Trans. Ind. Electron.*, vol. 58, no. 7, pp. 3110–3119, Jul. 2011.

[42] M. O. Efe, "Neural Network Assisted Computationally Simple PID Control of a Quadrotor UAV," *Ind. Inform. IEEE Trans. On*, vol. 7, no. 2, pp. 354 –361, May 2011.

[43] K. Levenberg, "A method for the solution of certain problems in least squares," *Q. Appplied Math.*, vol. 2, pp. 164–168, 1944.

[44] *MATLAB Neural Network Toolbox*. MathWorks.

[45] B. M. Wilamowski, N. J. Cotton, O. Kaynak, and G. Dundar, "Computing Gradient Vector and Jacobian Matrix in Arbitrarily Connected Neural Networks," *Ind. Electron. IEEE Trans. On*, vol. 55, no. 10, pp. 3784 –3790, Oct. 2008.

[46] B. M. Wilamowski and H. Yu, "Improved Computation for Levenberg - Marquardt Training," *Neural Netw. IEEE Trans. On*, vol. 21, no. 6, pp. 930 –937, Jun. 2010.

[47] B. M. Wilamowski and H. Yu, "Neural Network Learning Without Backpropagation," *Neural Netw. IEEE Trans. On*, vol. 21, no. 11, pp. 1793 –1803, Nov. 2010.

[48] M. T. Hagan and M. B. Menhaj, "Training feedforward networks with the Marquardt algorithm," *Neural Netw. IEEE Trans. On*, vol. 5, no. 6, pp. 989 –993, Nov. 1994.

[49] S. E. Fahlman, "Faster-learning variations on Back-propagation: An empirical study," vol. pp, pp. 38–51, 1988.

[50] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Publ. Online 09 Oct. 1986 Doi101038323533a0*, vol. 323, no. 6088, pp. 533–536, Oct. 1986.

[51] *Stuttgart Neural Network Simulator*. University of Tubingen.

[52] B. M. Wilamowski and L. Torvik, "Modification of gradient computation in the back-propagation algorithm," in *Proc. Artif. Neural Netw. Eng.*, St. Louis, MO, 1993, pp. 175–180.

[53] B. M. Wilamowski, "Challenges in applications of computational intelligence in industrial electronics," in *Industrial Electronics (ISIE), 2010 IEEE International Symposium on*, 2010, pp. 15 –22.

[54] E. H. Mamdani, "Advances in the linguistic synthesis of fuzzy controllers," *Int. J. Man-Mach. Stud.*, vol. 8, no. 6, pp. 669–678, Nov. 1976.

[55] B. Rezaee and M. H. F. Zarandi, "Data-driven Fuzzy Modeling for Takagi-Sugeno-Kang Fuzzy System," *Inf Sci*, vol. 180, no. 2, pp. 241–255, Jan. 2010.

[56] T. T. Xie, H. Yu, and B. M. Wilamowski, "Comparison of Fuzzy and Neural Systems for. Implementation of Nonlinear Control Surfaces," in *Human - Computer Systems Interaction*, vol. II, Berlin, Heidelberg: Springer-Verlag, 2012, pp. 313–324.

[57] M. R. Civanlar and H. J. Trussell, "Constructing membership functions using statistical data," *Fuzzy Sets Syst.*, vol. 18, no. 1, pp. 1–13, Jan. 1986.

[58] B. M. Wilamowski, "Neural Networks and Fuzzy Systems for Nonlinear Applications," in *Intelligent Engineering Systems, 2007. INES 2007. 11th International Conference on*, 2007, pp. 13 –19.

[59] T. Xie, H. Yu, and B. Wilamowski, "Replacing fuzzy systems with neural networks," in *Human System Interactions (HSI), 2010 3rd Conference on*, 2010, pp. 189 –193.

[60] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme learning machine: Theory and applications," *Neurocomputing*, vol. 70, no. 1–3, pp. 489–501, Dec. 2006.

[61] G.-B. Huang, L. Chen, and C.-K. Siew, "Universal approximation using incremental constructive feedforward networks with random hidden nodes," *IEEE Trans. Neural Netw.*, vol. 17, no. 4, pp. 879 – 892, Jul. 2006.

[62] G.-B. Huang and L. Chen, "Convex incremental extreme learning machine," *Neurocomputing*, vol. 70, no. 16–18, pp. 3056–3062, Oct. 2007.

[63] G.-B. Huang and L. Chen, "Enhanced random search based incremental extreme learning machine," *Neurocomputing*, vol. 71, no. 16–18, pp. 3460–3468, Oct. 2008.

[64] V. Vapnik, *The Nature of Statistical Learning Theory*, 1st ed. Wiley Interscience, 1998.

[65] A. J. Smola and B. Schölkopf, "A tutorial on support vector regression," *Stat. Comput.*, vol. 14, no. 3, pp. 199–222, Aug. 2004.

[66] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, pp. 27:1–27:27, 2011.

[67] Jason Weston, Andre Elisseeff, Gokhan Bakir, and Fabian Sinz, *The Spider*. Tubingen, Germany: MPI for Biological Cybernetics, 2006.

[68] John E. Moody, "The Effective Number of Parameters: An Analysis of Generalization and Regularization in Nonlinear Learning Systems," in *Proceedings of the 1991 NIPS Conference*, San Mateo, CA: Morgan Kaufmann Publishers, 1992, pp. 847–854.

[69] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C (2Nd Ed.): The Art of Scientific Computing*. New York, NY, USA: Cambridge University Press, 1992.

[70] D. E. Knuth, in *Seminumerical Algorithms*, 3rd ed., vol. 2, Boston, MA: Addison-Wesley, 1997, p. 232.

[71] J. L. Marroquin, E. A. Santana, and S. Botello, "Hidden Markov measure field models for image segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 25, no. 11, pp. 1380–1387, Nov. 2003.

[72] X. Xie and R. J. Evans, "Multiple target tracking and multiple frequency line tracking using hidden Markov models," *IEEE Trans. Signal Process.*, vol. 39, no. 12, pp. 2659 –2676, Dec. 1991.

[73] R. L. Plackett, “Some Theorems in Least Squares,” *Biometrika*, vol. 37, no. 1/2, pp. 149–157, Jun. 1950.

[74] T. L. Lai, H. Robbins, and C. Z. Wei, “Strong Consistency of Least Squares Estimates in Multiple Regression,” *Proc. Natl. Acad. Sci. U. S. A.*, vol. 75, no. 7, pp. 3034–3036, Jul. 1978.

[75] E. Borghers and P. Wessa, “Statistics-Econometrics-Forecasting.” Office for Research Development and Education, 2012.

[76] Paul A. Ruud, “Proof of the Gauss-Markov Theorem.” Econometrics Laboratory, University of California, Berkeley, 1995.

[77] A. Frank and A. Asuncion, *UCI Machine Learning Repository*. University of California, Irvine, School of Information and Computer Sciences, 2010.

[78] A. Aitken, “On least squares and linear combination of observations,” *Proc R Soc Edinb*, vol. 55, pp. 42–48, 1934.

[79] Chung-Ming Kuan, “Generalized Least Squares Theory,” in *Statistics: Concepts and Methods*, 2nd ed., Taipei, China: Huatai Publisher, 2004, pp. 77–107.

[80] D. E. Farrar and R. R. Glauber, “Multicollinearity in Regression Analysis: The Problem Revisited,” *Rev. Econ. Stat.*, vol. 49, no. 1, pp. 92–107, Feb. 1967.

[81] Patrick Breheny, “Ridge Regression,” presented at the BST 764: Applied Statistical Modeling, University of Kentucky, Lexington, KY, 01-Sep-2011.

[82] A. E. Hoerl and R. W. Kennard, “Ridge Regression: Biased Estimation for Nonorthogonal Problems,” *Technometrics*, vol. 12, no. 1, pp. 55–67, Feb. 1970.

[83] Joseph E. Cavanaugh, “Penalized (Ridge) Regression and the LASSO,” presented at the 171:290 Model Selection, Department of Biostatistics, University of Iowa, 27-Nov-2012.

[84] Dmitrij Celov, “How to calculate regularization parameter in ridge regression given degrees of freedom and input matrix?,” *Stack Exchange*, 15-Mar-2011. [Online]. Available: http://stats.stackexchange.com/questions/8309/how-to-calculate-regularization-parameter-in-ridge-regression-given-degrees-of-f. [Accessed: 31-Dec-2013].

[85] Kerby Shedden, “Prediction,” presented at the Statistics 600, Department of Statistics, University of Michigan, 07-Nov-2011.

[86] E. W. Weisstein, “Matrix Trace -- from Wolfram MathWorld.” [Online]. Available: http://mathworld.wolfram.com/MatrixTrace.html. [Accessed: 01-Jan-2014].

[87] E. W. Weisstein, "Singular Value Decomposition -- from Wolfram MathWorld." [Online]. Available: http://mathworld.wolfram.com/SingularValueDecomposition.html. [Accessed: 01-Jan-2014].

[88] B. M. Wilamowski, "B. Wilamowski on Fast Forward Spherical N-Dimensional Clustering: private conversation," Oct-2010.

[89] Baranowski Jerzy and Czajkowski Grzegorz, "Special Purpose Analog Circuits," in *Electronic Circuits Part II Analogue and pulse nonlinear*, Poland: WNT, 2004.

[90] "NVIDIA Newsroom - Releases - Researchers Deploy GPUs to Build World's Largest Artificial Neural Network." [Online]. Available: http://nvidianews.nvidia.com/Releases/Researchers-Deploy-GPUs-to-Build-World-s-Largest-Artificial-Neural-Network-9c7.aspx. [Accessed: 16-Apr-2014].

[91] T. P. Morgan, "Netflix Speeds Machine Learning With Amazon GPUs," *EnterpriseTech*. [Online]. Available: http://www.enterprisetech.com/2014/02/11/netflix-speeds-machine-learning-amazon-gpus/. [Accessed: 16-Apr-2014].

[92] B. W. Oskar Xaver Schlömilch, *Zeitschrift für Mathematik und Physik...* 1901.

[93] G. Dahlquist, A. Bjorck, and Mathematics, *Numerical Methods*. Dover Publications, 2003.

[94] "Runge's phenomenon," *Wikipedia, the free encyclopedia*. 24-Sep-2012.

[95] R. L. Burden and J. D. Faires, *Numerical Analysis*, 7th ed. Brooks Cole, 2000.

[96] T.-T. Lee and J.-T. Jeng, "The Chebyshev-polynomials-based unified model neural networks for function approximation," *IEEE Trans. Syst. Man Cybern. Part B Cybern.*, vol. 28, no. 6, pp. 925 –935, Dec. 1998.

[97] K. Levenberg, "A method for the solution of certain problems in least squares," *Q. Appplied Math.*, vol. 2, pp. 164–168, 1944.

Appendices

7.1 Appendix A – Exploration of Chebychev Transform Methods

7.1.1 Background, and Two Chebychev Transform Implementations

It is well-known that polynomial interpolations of functions produce oscillation at the edges of the interval known as Runge's phenomenon [92]. The error between the generating function and the interpolating polynomial of order *n* is given by [93]:

$$f(x) - P_n(x) = \frac{f^{n+1}\xi}{(n+1)!}\prod_{i=1}^{n+1}(x - x_i) \tag{73}$$

$$for\ some\ \xi\ in\ [-1,1]$$

The effect worsens for higher degrees of polynomials as seen in Figure 50 below:

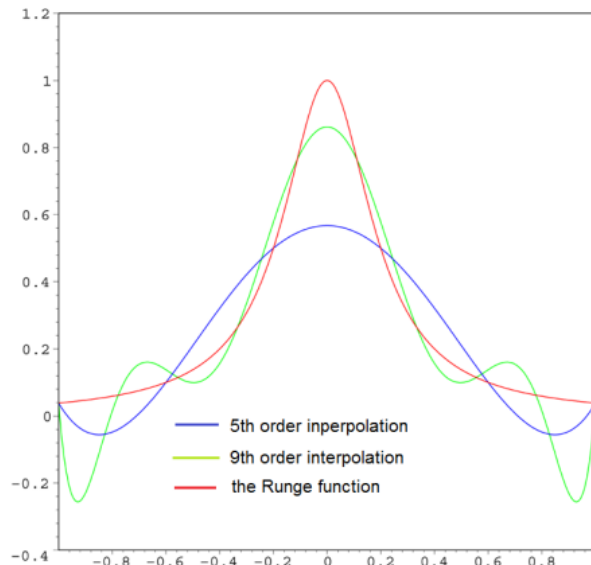


Figure 50 Runge's phenomenon and refit with Chebychev nodes [94]

The effect is pronounced when the inputs are spaced evenly over the input interval. It is also well-known that transforming the input spacing to that of Chebychev node spacing can minimize the error [95]. The affine transform spacings of inputs x_i over an arbitrary interval $[a, b]$ are given by:

$$\tilde{x}_i = \frac{1}{2}(a+b) + \frac{1}{2}(b-a)\cos\left(\frac{2i-1}{2n}\pi\right) \tag{74}$$

Other authors have proposed Chebychev-based polynomial neural networks as universal approximators. Lee and Jeng simulated a feed-forward neural network with 40 hidden computational nodes using two layers to implement a one-dimensional Chebychev polynomial network [96]. The 1st hidden layer contained monomial term activation functions to create the polynomial product terms, and a 2nd hidden layer was used to explicitly compute an approximate transform of the resulting terms into Chebychev spacing. In other words, the method proposed by Lee and Jeng requires either two separate stages of training, or requires an efficient second-order training algorithm (such as Levenberg-Marquardt [97]). Though their results were promising (SSE=0.2115 for 20 training epochs, max polynomial order = 19), a more efficient method was sought for the current polynomial-based networks under study.

Two implementations of input Chebychev transform spacing were explored. The transform for both versions is straightforwardly computed for any real values by the simple MATLAB routine below:

```
%% rescaling for Chebyshev nodes
% Runge's phenomenon
% Chebyshev nodes
a=min(min(x)); b=max(max(x));
y=0.5*(a+b)-0.5*(b-a)*cos((x-a)./(b-a)*pi);
```

Alternately, the transform can be more easily computed for input data that is pre-normalized to [-1:1] as in:

```
function y=chebConv(x)
%% rescaling for Chebyshev nodes
% Runge's phenomenon
% Chebyshev nodes for data normalized over [-1 1]
y = -cos( ((x+1)./(2))*pi );
```

The diagram of Figure 51 displays the Method 1 approach. It shows how a simple Chebychev transform of input domain data can lead to network coefficients (weights) and monomial terms that essentially contain the complete Chebychev encoding – the same network weights and terms can be used for processing input validation data without any further transformation. However, it is noted that this method requires explicit knowledge of the input-output relationship of the function to be trained. The desired outputs ("d.o.") of the original training data must be generated by a known transform function.

Method 1:

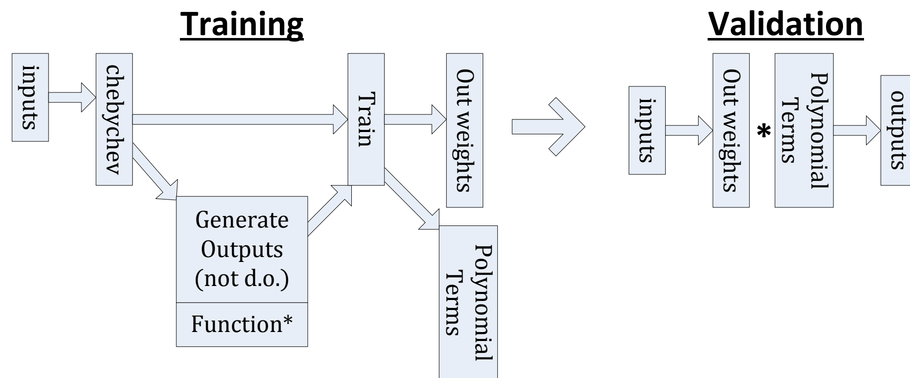
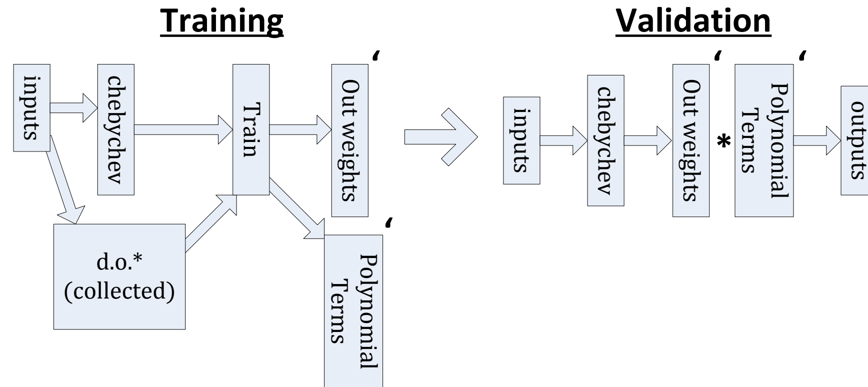


Figure 51 Polynomial Network, Chebychev Transform Method 1 – Training: encoding of input vectors, plus regeneration of desired outputs, Validation: straightforward processing with encoded weights and terms

Method 1 will not allow the use Chebychev transform techniques on real-world data when the relationships between the inputs and the outputs are unknown. For those cases, a second method was implemented.

For Method 2, both training and validation inputs require the same Chebychev transform, but training can proceed without reconstruction of the underlying input-output function. System weights and terms are produced which process equivalently during training and validation, at the cost of an additional Chebychev transform which must be applied to validation inputs. Figure 52 illustrates Method 2:

Method 2:



*** This method does not require specific knowledge of the original function, but results in a transformed solution that may or may not work as well as the non-Chebychev case**

Figure 52 Polynomial Network, Chebychev Transform Method 2 – Training: encoding of input vectors, Validation: Chebychev encoding of inputs and processing with encoded weights and terms

7.1.2 Chebychev Techniques – Experimental Results

The following 2-input non-linear control surface was selected as the experimental function to approximate:

$$z = \frac{5}{1 + 0.25(x-1)^2 + 0.04(y-2)^2} + \frac{2}{1 + 0.25(x+1)^2 + 0.25(y-1)^2} \quad (75)$$

The validation surface is shown in Figure 53. Initial experimentation showed that results with Chebychev methods vary depending on the presence of training vectors near critical output function details, as illustrated in Figure 54. For this reason, several different resolutions of regularly-spaced training data were tried. Additionally, it was noted that transform method response was sensitive to noise in the training output data, as is encountered in real-world situations. Therefore trials were run which include these variations. All experiments were run using the initial polynomial network variant, PolyNet, discussed elsewhere in this study.

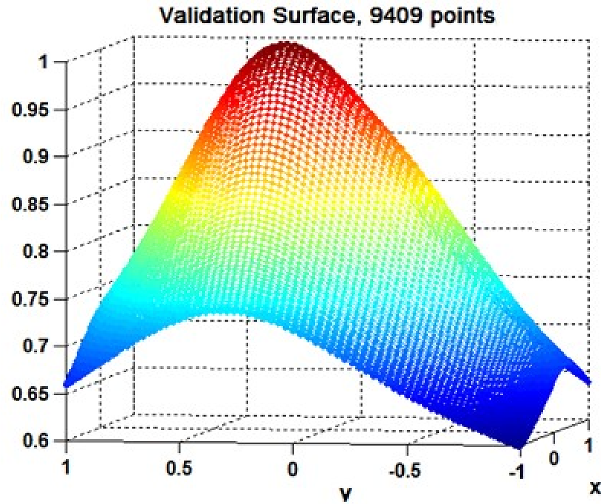


Figure 53 Validation for the Chebychev Transform Experiments, 9409 points (97x97)

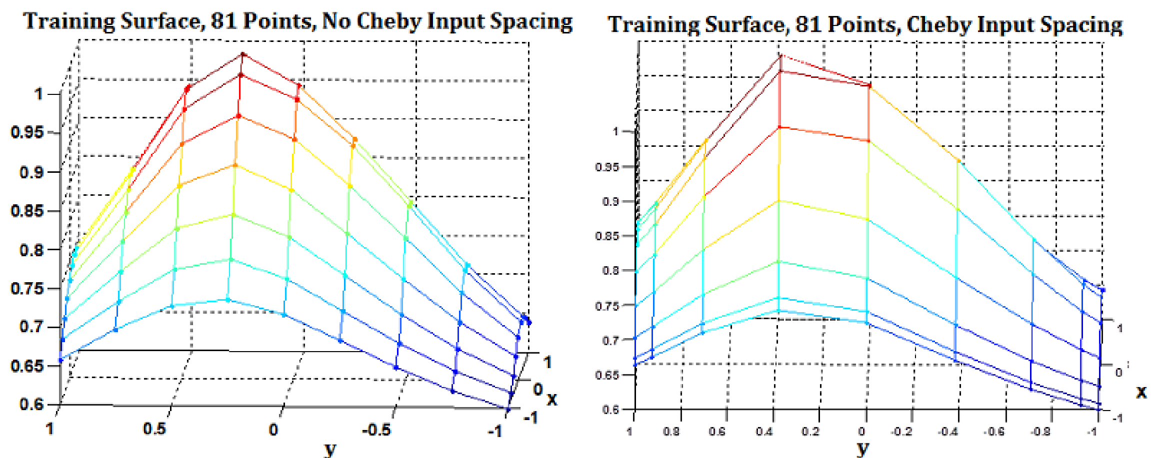


Figure 54 9x9 training point grid: (left) standard spacing captures function, (right) Chebychev spacing fails to sample function at critical points

Experiments were run with various training point resolutions. For each resolution, three cases were run: no Chebychev transform, Method 1, Method 2. The output training and validation RMSE curves are shown in Figure 55 for the 81-point (9x9) training resolution. Note that these results look initially promising for the Chebychev methods. As expected, the training curves are almost identical for all three variants, and are also all monotonic. As usual, the validation RMSE curve for the non-Chebychev variant (standard PolyNet) is non-monotonic, with best performance at max-order 7, with 36 monomials. Method 1 yields an almost

monotonic validation curve, and Method 2 does achieve a monotonic validation RMSE curve which nearly matches the training curve in shape. However, these results vary greatly depending on the resolution and location of training vectors in the input space.

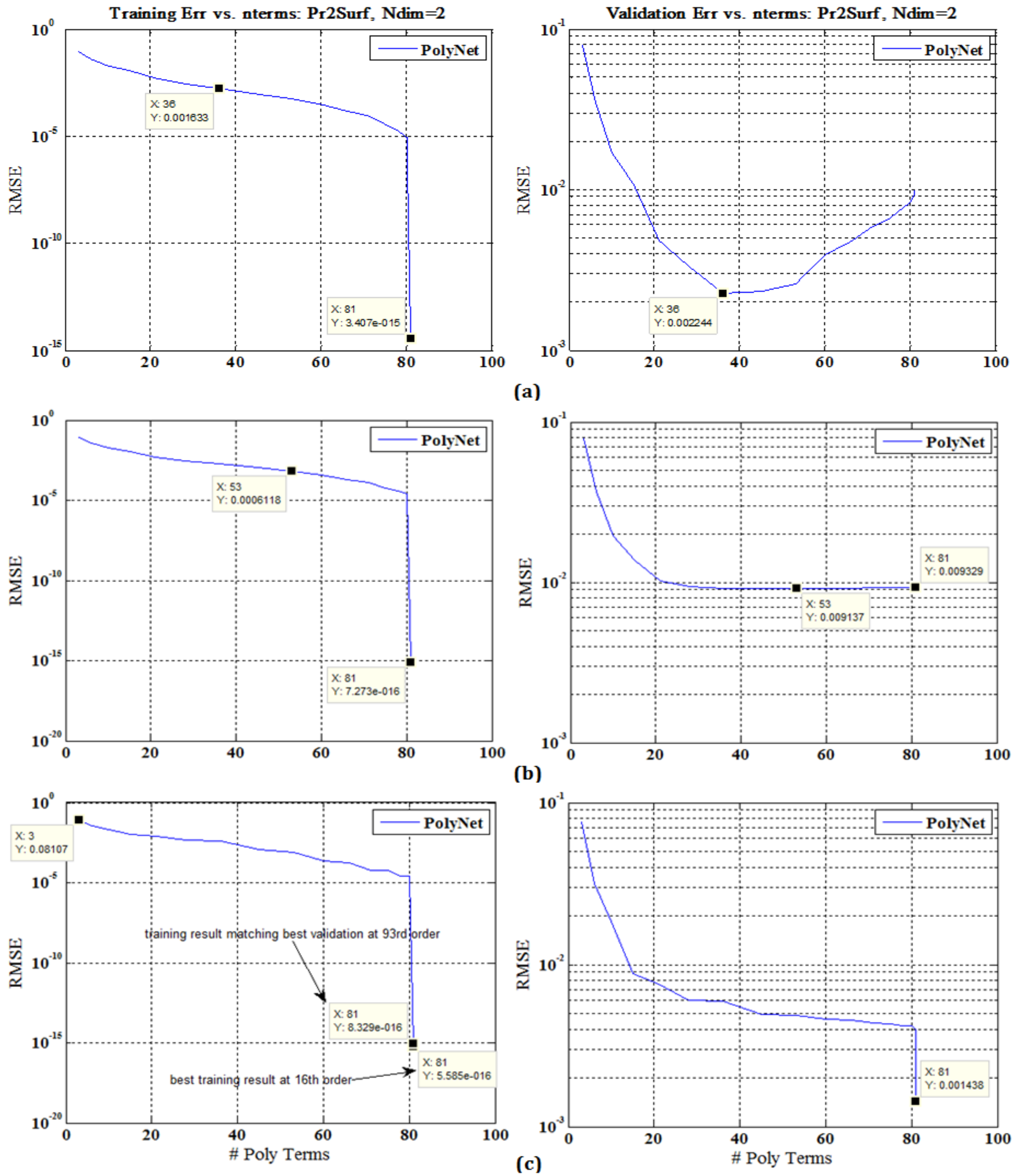


Figure 55 Training and Validation RMSE Curves (81 training points): (a) no Chebychev spacing, (b) Method 1, (c) Method 2

Testing was eventually performed with several different training input resolutions. The final results point to a less enthusiastic conclusion for the use of Chebychev input transforms. As seen in Table XIX, neither Chebychev transform method outperforms the no-transform standard for all four input resolutions depicted. In fact, Method 1, though promising in training, does not prevail in any validation case. Method 2 appears to prevail only when the training resolution is sparse compared to validation density.

Table XIX Results for Chebychev Transform Testing, no noise case
(RED=worst, GREEN=best)

Training Resolution	Training RMSE			Validation RMSE (9409 points)		
	Chebychev Method			Chebychev Method		
	none	1	2	none	1	2
5x5	1.49e-16	1.73e-16	2.11e-16	31.57e-3	35.47e-3	22.38e-3
9x9	9.38e-16	4.34e-16	5.59e-16	2.244e-3	9.137e-3	1.438e-3
17x17	6.35e-14	1.33e-14	6.47e-16	0.153e-3	2.248e-3	1.147e-3
33x33	5.81e-10	7.00e-10	1.03e-3	0.127e-3	0.358e-3	0.980e-3

Lastly, the same experiments were repeated by including uniformly distributed random noise on the training outputs to a max deviation of 5% of full output magnitude (0.05 on a 0:1 scale). Validation outputs were kept exact. The results of Table XX show even less favorable results for the Chebychev methods in the presence of noise in the training data. Method 2 shows promise only for the sparsest training point resolution relative to validation density. The standard non-Chebychev regression in PolyNet prevails more decisively against the two variants.

In summary, no advantage was discovered in using Chebychev transform methods on the input spacing of the system. It is assumed that even less advantage would be apparent as the methods are applied to multi-dimensional data problems.

Table XX Results for Chebychev Transform Testing, 5% training noise (RED=worst, GREEN=best)

Training Spacing	Training RMSE			Validation RMSE		
	Chebychev Method			Chebychev Method		
	none	1	2	none	1	2
5x5	1.66e-16	2.16e-16	1.65e-16	43.86e-3	35.66e-3	22.26e-3
9x9	1.68e-13	1.11e-15	1.28e-15	6.397e-3	14.01e-3	8.697e-3
17x17	7.97e-3	6.54e-3	6.47e-16	3.989e-3	4.773e-3	6.097e-3
33x33	10.78e-3	11.1e-3	11.2e-3	2.648e-3	2.944e-3	3.474e-3

7.2 Appendix B – MATLAB Code: Unique Polynomial Term Generation

```
function index=Get_In_pol(Nmax)
%% sorting and then using  index=findinx(Nmax)
[N,I]=sort(Nmax) ;  % sorting from smallest to largest
temp=findinx(N);
index=ones(size(temp));   % calculating indices
for i=1:length(I),    %puting everything back in the original order
    index(:,I(i))=temp(:,i);
end;
return;

function index=findinx(Nmax)    % only for ordered
%% finding indices so multidim case is posible in one loop
len=length(Nmax);
if len==1,      index=(1:Nmax(1))';     return;   end;
ind=findinx(Nmax(1:(len-1)));    %using recursion
% ind = [1:Nmax(1)]'
[x,y]=size(ind);
index=[];
for i=1:x,
    for j=ind(i,y):Nmax(len)
        add=[ind(i,:),j];
        index=[index;add];
    end;
end;
return;
```

7.3 Appendix C – MATLAB Code: Statistical Processing of Monomial Term Weights

Option 1 Example – PlyPaPGLS.m excerpts

These variants remove one monomial term at a time following iterative evaluation of noise-responsiveness of term coefficients.

```
function ['...'] = PlyPaPGLS('...')

trins = otrins ;         % training pattern inputs
trouts = otrouts ;       % training pattern outputs

[np,nd]=size(trins);
agu=[ones(np,1),trins];  %agumented input space  (+1 is added in the
column)

['...']

maxmeansslope = 1e-2 ;
trpct = 0.75 ;        % percent of training data to pull into the
iterative training

Nmax=(nd+1)*ones(1,PrOrder);% GENERATE all monomial term indices up to
a max order
ind=Get_In_pol(Nmax) ;   % get **all** poly indices right away
[Nt,Nd]=size(ind) ;      % get indices size for current round

tridx = 1 ;       % initial training index value to store output vectors
while (Nt >= 1)
    % ENTER the iterative OLS (or other) regression phase:
    wwmeans = 0 ;     % initialize running mean
    wwabsmax = [] ;    % initial running max abs value of coeffs
    wwscstds = 0 ;
    wwscmeans = 0 ;
    prevwwscaled = 0 ;    % initialize k-1 normalized weights
    scmeanslope = 0 ;
    for k = 1:100  % drill down to the essential output weights for
each instance
        agusubset = agu(1:trlength,:);% take subset of training
patterns
        troutsbst = trouts(1:trlength,:) ;

        %% BUILDING polynomial terms...
        Jsub = [] ;  % DON'T FORGET the zero-order term..
        for jj=1:Nt
            Pr=1;
            for i=1:Nd
                Pr=Pr.*agusubset(:,ind(jj,i));
            end
            Jsub = [Jsub Pr];
        end

        wwtemp = OLS_reg(Jsub,troutsbst);% Finding weights, OLS (this
case)
        wwmeans = (wwtemp + wwmeans.*(k-1))./k;% compute running means
of raw coefficients
        wwabsmax = max(abs([wwabsmax wwtemp]),[],2) + 1e-16 ; % compute
running max-abs of raw coefficients
```

```
        wwscaled = wwtemp./wwabsmax ; % NORMALIZE raw coefficients per
current max-abs
        prevscmeans = wwscmeans ; % capture preceding normalized coeff
means
        wwscmeans = (wwscaled + wwscmeans.*(k-1))./k ;  % compute
running means of scaled coefficients
        scmeansn_1 = sum(abs(wwscmeans - prevscmeans)) ; % compute sum
of 1st order gradients from last scaled means to current
        scmeanslope = abs(scmeanslope + scmeansn_1)/k ;  % compute
running mean of all scmeansn-1

        % Compute incremental STDs of normalized mean coefficients per
monomial term
        wwscstds = sqrt( (((wwscstds.^2)./k + ((wwscmeans -
prevscmeans).^2)).*(k-1)) ) ;

        if (avscmeansslope <= maxmeansslope)  % STOPPING criterion
            break
        end

        shuffle = randperm(np) ;    %  set up for next iteration...
        agu = agu(shuffle,:) ;
        trouts = trouts(shuffle,:) ;
    end

    wwtemp = wwmeans ;    % assign last wwmeans as final coeffs

    ['...']

    tridx = tridx + 1 ;  % increment the trial index for output vectors
    % &%&%&%&%&%&%&%&%&%&%&%&%&%&%&%&%&%&%&%&%&%&%&%&%&
    % TRIM the noisiest term from the poly set, run again
    [maxwwscstd maxstdidx] = max(wwscstds);% find the current
"noisiest" coefficient term
    ind(maxstdidx,:) = [] ;   % ...remove it
    Nt = Nt - 1 ;             % ...reduce total poly term count by 1
end
```

Option 2 Example – PolyStat.m excerpts

These variants remove one one or more monomial terms at a time following iterative threshold-based evaluation of noise-responsiveness of term coefficients.

```
function ["…"] = PolyStat("…")

maxmeansslope = 1e-2 ; % arbitrary max Delta to stop iterative training
trpct = 0.75 ;        % percent of training data to pull into the
iterative training
stdscale = 2.75       % multiplication factor for max STD threshold

trins = otrins ;      % training pattern inputs
trouts = otrouts ;    % training pattern outputs

[…]
```

```
ind = "..."  % GENERATE polynomial term indices..
reducing = true ;   % set flag to indicate we are still in the term-
pruning mode
while (reducing == true)

    % ENTER the iterative OLS (or other) regression phase:
    wwmeans = 0 ;    % initialize running mean
    wwabsmax = [] ;   % initial running max abs value of coeffs
    wwscstds = 0 ;
    wwscmeans = 0 ;
    scmeanslope = 0 ;

    for k = 1:50    % drill down to the essential output weights
        [Nt,Nd]=size(ind) ;        % get indices size for current round
        agusubset = agu(1:trlength,:);% take subset of training
patterns
        troutsbst = trouts(1:trlength,:) ;

        %% BUILDING POLY TERMS
        Jsub = [] ;  % DON'T FORGET the zero-order term..
        for jj=1:Nt
            Pr=1;
            for i=1:Nd
                Pr=Pr.*agusubset(:,ind(jj,i));
            end
            Jsub = [Jsub Pr];
        end
        wwtemp = lin_reg(Jsub,troutsbst) ;  % Finding weights, OLS
method
        prevmeans = wwmeans ; % capture preceding raw coefficient means
        wwmeans = (wwtemp + wwmeans.*(k-1))./k ;% compute running means
of raw coefficients
        wwabsmax = max(abs([wwabsmax wwtemp]),[],2) + 1e-16 ; % compute
running max-abs of raw coefficients

        wwscaled = wwtemp./wwabsmax ; % NORMALIZE raw coefficients per
current max-abs
        prevscmeans = wwscmeans ; % capture preceding normalized ???
        wwscmeans = (wwscaled + wwscmeans.*(k-1))./k ;  % compute
running means of scaled coefficients
        scmeansn_1 = sum(abs(wwscmeans - prevscmeans)) ; % compute sum
of 1st order gradients from last scaled means to current
        scmeanslope = abs(scmeanslope + scmeansn_1)/k ;  % compute
running mean of all scmeansn-1

        % Compute incremental STDs of normalized mean coefficients per
monomial term
        wwscstds = sqrt( (((wwscstds.^2)./k + ((wwscmeans -
prevscmeans).^2)).*(k-1)) ) ;

        if (scmeanslope <= maxmeansslope)  % STOPPING criterion
            break
        end

        shuffle = randperm(np) ;   %  set up for next iteration...
        agu = agu(shuffle,:) ;
        trouts = trouts(shuffle,:) ;
```

```
    end

    meanstd = mean(wwscstds(find(wwscstds),:)) ; % only figure non-
zeros into this..
    stdstd = std(wwscstds(find(wwscstds),:),1) ; % only figure non-
zeros into this..
    maxscstd = meanstd + (stdscale * stdstd) ;    % MAX stdstd threshold
criterion

    termthresh = (wwscstds > maxscstd) ;     %  PRUNE terms...
    if ( (sum(termthresh) > 0) && (sum(termthresh) < Nt) )
        wwtemp = wwtemp.*(~termthresh);% code terms over threshold with
zeros
        nonzeroidx = find(wwtemp) ;     % kick out zero-coefficient
terms
        ind = ind(nonzeroidx,:) ;       % REDUCE POLYNOMIAL TERMS !!!
    else
        reducing = false ;              % we're done reducing, kick out
        break
    end
end

wwtemp = wwmeans ;    % assign last wwmeans as final coeffs
```

7.4 Appendix D – MATLAB Code: Iterative GLS Regression

7.4 1 1-D 3rd-order Test – GLS_1D.m excerpts

```
function GLS_1D(npts,pctbad)
% INPUTS:
%           npts:           number of data points
%           pctbad[whole#]: percentage of data points with a terrible
outlaying
%                               value and a terrible bias

x = linspace(0,1,npts) ;

yideal = x.^3 ;

% first step: create normally distributed noise in all data:
y1 = x.^3 + (0.04).*randn([1,npts]) ;

% next step: make it worse by creating randomly distributed large
magnitude
%   errors with a non-symmetric bias:
replacex = sort(randi(npts,1,ceil((pctbad/100)*npts))) ;
y2 = y1 ;
for i=replacex
    y2(i) = y1(i) - x(i)*0.65*(1 - 0.2*abs(randn)) ;
end

figure(9); clf
plot(x,y2,'.',x,yideal,'k') ; hold on
```

```
h_title = title('Bad Data (nonuniform dist, uncorr var, awful trend,
etc.)') ;
h_xlabel = xlabel('x'); h_ylabel = ylabel('x^2');
legend('bad data','y=x^3','Location','northwest')

figure(10); clf     % this plot is for all proper 1st order tests
plot(x,y2,'.',x,yideal,'k', 'LineWidth',2) ; hold on

augx3 = [ones(1,npts)' x' (x.^2)' (x.^3)'] ;     % prep 3rd order
overfit
X3tX3 = augx3'*augx3 ;                            % 3rd order
W22 = augx3\y2' ;                                 % 3nd order
Y22 = augx3*W22 ;   % 2nd order overfit

figure(10)
plot(x,Y22,'LineWidth',2) ; hold all

% compute a baseline error btw desired function (y==x) and our full
"bad
%   data" set:
ebase = sqrt(sum((yideal - y2).^2)/npts) ;
display(['Baseline, inherent RMSE, bad data vs. ideal: '
num2str(ebase)]) ;

% compute the "real" error btw desired fcn and our current regression
%   method, which is equivalent to OLS:
ereal_OLS = sqrt(sum((yideal - Y22').^2)/npts) ;
display(['Real error, RMSE OLS(ours) vs. ideal: ' num2str(ereal_OLS)])
;

% compute the "dataset" error btw given (bad) data and our current
regression
%   method, which is equivalent to OLS:
edata_OLS = sqrt(sum((y2 - Y22').^2)/npts) ;
display(['Data error, RMSE OLS(ours) vs. bad data: '
num2str(edata_OLS)]) ;

% NEW METHOD: "GENERALIZED LEAST SQUARES"
errs = (y2 - Y22') ;     % compute the unsquared errors of our first
OLS attempt
Omega_1 = (diag(errs.^(2))) ;
edata_OLS
old_rmseGLS = 999 ;
BESTrmseI = 999 ;
RMSEs = [] ;
OmegaErrs = 999 ;
minGLSerr = 999 ;
for i = 1:50
    [a b] = size(augx3) ;
    lambdaI = (0.25/b).*eye(b) ;
    Bfgls = (augx3'*Omega_1*augx3 + lambdaI)\(augx3'*Omega_1*y2') ;

    Ygls = augx3*Bfgls ;
    errs = (y2 - Ygls') ;
    xfrmErr = sum(errs*Omega_1*errs')/npts ;
    if (xfrmErr < minGLSerr)
        minGLSerr = xfrmErr ;
```

```
        Bfinal = Bfgls ;
    end
    OmegaErrs = [OmegaErrs xfrmErr] ;
    IDEALrmse = sqrt(sum((yideal - Ygls').^2)/npts) ;
    RMSEs = [RMSEs IDEALrmse] ;
    if (IDEALrmse < BESTrmseI)
        BESTrmseI = IDEALrmse ;
        BESTi = i ;
        Bgls = Bfgls ;
    end
    if (i == 2)
        RMSEat2 = IDEALrmse ;
        Bglsat2 = Bfgls
    end

    if ((mod(i,1)==0) && abs(OmegaErrs(end-1)-xfrmErr)<=1e-10)
        breaki = i
         break
     end
    Omega_1 = (diag(errs.^(2))) ;
end

RMSEat2
BESTrmseI
BESTi

figure(10)
plot(x,augx3*Bgls,'-.',x,augx3*Bfinal,'--','LineWidth',2); hold all
legend('bad data','y=x^3','OLS','realbest','"BEST"
GLS','Location','northwest')
h_title = title(['GLS vs. OLS with BAD DATA: ' num2str(pctbad) '%
corruption']) ;
h_xlabel = xlabel('x'); h_ylabel = ylabel('x^2');
figprefs;

OmegaErrs(1) = [] ;     % eliminate initial sum value
figure(22); clf
plot(OmegaErrs)
```

7.4.2 General GLS Code for Multiple Dimension Data Regression – GLS_reg.m excerpts

```
function ww=GLS_reg(J,out)
%% Generalized Least Squares linear regression
maxERR = 1e-7 ;
maxiter = 150 ;

[np,ni]=size(J);
[n1,n2]=size(out);
if (np ~= n1) || (n2 ~= 1)
    error('Matrix size in GLS_reg.m are wrong');
end

initlambdaI = 1e-15.*eye(ni) ;  % a simple strategy to get unstuck from
local minima..
```

```
% 1) compute the standard (OLS) linear regression to start the process:
Bols = J\out ;
if ( isnan(sum(Bols)) )
    Bols = lscov(J,out) ;  % if matrix singular, use MATLAB's OLS
end
% 2) compute intermediate Y* and errors, construct the FGLS diagonal
matrix,
%    Omega:
Yols = J*Bols ;          % standard OLS regression
OLSerrs = (out' - Yols') ;  % unsquared errors of our first OLS attempt
Omega_1 = (diag(OLSerrs.^2,0)) ;
% 3) begin iterative process to minimize transformed GLS error each
time:
lastErr = 999 ;
for i = 1:maxiter
    invprod = J'*Omega_1*J ;     % denominator...
    numprod = J'*Omega_1*out ;  % numerator...
    Bfgls = invprod\numprod ;

   % PROTECT against singular matrix potential...
    if ( isnan(sum(Bfgls)) )
        lambdaI = initlambdaI ;
        while ( isnan(sum(Bfgls)) )
            lambdaI = 10.*lambdaI ;
            Bfgls = (invprod+lambdaI)\(numprod) ;
            if ( lambdaI(1,1) > 1e4 )
                try
                    Bfgls = lscov(J,out,Omega_1,'orth') ;
                    if ( isnan(sum(Bfgls)) )
                        Bfgls = Bols ;  %
                    end
                catch
                    try
                        Bfgls = lscov(J,out,Omega_1) ;
                        if ( isnan(sum(Bfgls)) )
                            Bfgls = Bols ;  %
                        end
                    catch
                        Bfgls = Bols ;  % RESORT TO OLS if all else
fails
                    end
                end
            end
        end
    end

    Ygls = J*Bfgls ;                      % compute intermediate values
    errs = (out' - Ygls') ;                % compute intermediate errors
    xfrmErr = (errs*Omega_1*errs')/np  ;  % intermediate GLS TRANSFORM
MSE
    if (abs(lastErr-xfrmErr)<=maxERR)
        ww = Bfgls ;
        break
    end
    lastErr = xfrmErr ;
    Omega_1 = (diag(errs.^2,0)) ;   % compute next Omega if necessary
```

```
end

if (i==maxiter)
    'WARN max GLS iter'
    ww = Bfgls ;
end
```

7.5 Appendix E – MATLAB Code: Iterative Ridge Regression

7.5.1 Computation of an Initial Minimum-variance λ – OLSridge_reg.m excerpt

The following computes a minimum-variance λ for all variants of iterative ridge regression. This is the necessary first-step preceding an original iterative process. The code is adapted from Celov et al. [84].

```
%% &%&%&%&%&%&% AUXILLIARY FUNCTION &%&%&%&%&%&%&%&%&%&%&%&%
% Compute a minimum-variance lambda by Newton-Raphson/Fisher process
function [lamda] = calc_lamda(Xnormalised,df,p)

        %Finding SVD of data
        [u s v]=svd(Xnormalised);  % canned MATLAB function
        Di=diag(s) ;
        Dsq=Di.^2;

        %Newton-rapson method to solve for lamda
        lamdaPrev = (p-df)/df ;
        lamdaCur = 99 ;%random large value
        diff=lamdaCur-lamdaPrev;
        threshold = 1e-14 ;
        count = 0 ;
        while (diff>threshold) && (count < 51)
            count = count + 1 ;
            numerator=sum(Dsq ./(Dsq+lamdaPrev))-df ;
            denominator=sum(Dsq./((Dsq+lamdaPrev).^2)) ;
            lamdaCur=lamdaPrev+(numerator/denominator);
            diff=abs(lamdaCur-lamdaPrev) ;
            lamdaPrev=lamdaCur;
        end
        lamda=lamdaCur ;
return
```

7.5.2 Iterative λ Optimization Process – OLSridge_reg.m excerpt

The following computes an optimized (minimum RMSE) λ, balancing a compromise between ideal variance and ideal bias.

```
function ww=OLSridge_reg(J,out,df)
%% OLS-Ridge regression
% INPUTS:   J:      training vector inputs
%           out:    training vector outputs
%           df:     RR degrees of freedom (usually polynomial order)
```

```
[np,ni]=size(J);
[n1,n2]=size(out);
if (np ~= n1) || (n2 ~= 1)
    error('Matrix size is wrong');
end

muscale = 1.5 ;       % initial lambda scaling factor to hunt for ideal

lambda = calc_lamda(J,df,ni) ;       %  compute initial min-var lambda
prevSSE = np ;                      % initialize previous errors
prevlambda = lambda ;  % needed for odd condition of RMSE increase upon
first iteration
hessi = J'*J ;       % compute once, use repeatedly
regnum = J'*out ;    % compute once, use repeatedly
for count = 1:1000  % overkill, but just in case...
    ww = (hessi+(lambda.*eye(ni)))\regnum ;  %compute initial weights
    currSSE = sum((out' - (J*ww)').^2) ;   % compute current errors
    if (currSSE < prevSSE)       % condition for shrinkage
        prevlambda = lambda ;
        lambda = lambda - lambda/muscale ;   % shrink lambda
        prevSSE = currSSE ;
    elseif (currSSE > prevSSE)  % condition for overshoot...
        lambda = prevlambda ;
        muscale = muscale*2 ;
    else
        break                  % we hope we are done
    end
end
```

7.6 Appendix F – MATLAB Code: Hybrid Regression Techniques Including RR

7.6.1 Iterative GLS + RR Minimum-Variance Regression – GLSminvar_reg.m excerpts

```
function ww=GLSminvar_reg(J,out,df)
%% Generalized Least Squares linear regression which initially computes
% the minimum-variance Lambda (from Ridge Regression).  This produces
% training and validation RMSE results with the least variance between
the two.
% INPUTS:   J:      training vector inputs
%           out:    training vector outputs
%           df:     degrees of freedom (usually polynomial order)

maxERR = 1e-7 ;
maxiter = 100 ;

[np,ni]=size(J);
[n1,n2]=size(out);
if (np ~= n1) || (n2 ~= 1)
    error('Matrix size is wrong');
end

lambdaI = calc_lamda(J,df,ni).*eye(ni) ;  % SEE APPENDIX E
```

```
% 1) compute the standard (OLS) linear regression to start the process:
Bols = J\out ;
if ( isnan(sum(Bols)) )
    Bols = lscov(J,out) ;  %
end
% 2) compute intermediate Y* and errors, construct the FGLS diagonal
matrix,
%    Omega:
Yols = J*Bols ;           % standard linear regression
OLSerrs = (out' - Yols') ;  % unsquared errors of our first OLS attempt
Omega_1 = (diag(OLSerrs.^2,0)) ;
% 3) begin iterative process to minimize transformed GLS error each
time:
lastErr = 999 ;
for i = 1:maxiter
    Bfgls = (J'*Omega_1*J+lambdaI)\(J'*Omega_1*out) ;
    if ( isnan(sum(Bfgls)) )         % trap singular matrix conditions
        Bfgls = (J'*J+lambdaI)\(J'*out) ;
    end
    Ygls = J*Bfgls ;                     % compute intermediate values
    errs = (out' - Ygls') ;               % compute intermediate errors
    xfrmErr = (errs*Omega_1*errs')/np  ;  % intermediate transform MSE
    if (abs(lastErr-xfrmErr)<=maxERR)
        ww = Bfgls ;
        break
    end
    lastErr = xfrmErr ;
    Omega_1 = (diag(errs.^2,0)) ;   % compute next Omega if necessary
end
```

7.6.2 Iterative GLS + RR Full Optimization Regression – GLSridge_reg.m excerpts

```
function ww=GLSridge_reg(J,out,df)
%% Generalized Least Squares regression with ridge regression component
maxERR = 1e-7 ;
maxiter = 200 ;

[np,ni]=size(J);
[n1,n2]=size(out);

% NOTE: this just uses the lambdaI technique to avoid singularity, no
ridge yet..
initlambdaI = 1e-15.*eye(ni) ;   % simple method to escape singular
matrix problem

% 1) compute the standard (OLS) linear regression to start the process:
Bols = J\out ;
if ( isnan(sum(Bols)) )
    Bols = lscov(J,out) ;  %
end
% 2) compute intermediate Y* and errors, construct the FGLS diagonal
matrix,
% Omega:
Yols = J*Bols ;           % standard linear regression
OLSerrs = (out' - Yols') ;  % unsquared errors of our first OLS attempt
```

```
Omega_1 = (diag(OLSerrs.^2,0)) ;
% 3) begin iterative process to minimize transformed GLS error each
time:
lastErr = 999 ;
for i = 1:maxiter
    invprod = J'*Omega_1*J ;
    numprod = J'*Omega_1*out ;
    Bfgls = invprod\numprod ;
    if ( isnan(sum(Bfgls)) )
        lambdaI = initlambdaI ;
        while ( isnan(sum(Bfgls)) )
            lambdaI = 10.*lambdaI ;
            Bfgls = (invprod+lambdaI)\(numprod) ;
            if ( lambdaI(1,1) > 1e4 )
                try
                    Bfgls = lscov(J,out,Omega_1,'orth') ;
                    if ( isnan(sum(Bfgls)) )
                        Bfgls = Bols ;  %
                    end
                catch
                    try
                        Bfgls = lscov(J,out,Omega_1) ;
                        if ( isnan(sum(Bfgls)) )
                            Bfgls = Bols ;  %
                        end
                    catch
                        Bfgls = Bols ;
                    end
                end
            end
        end
    end
    Ygls = J*Bfgls ;                        % compute intermediate values
    errs = (out' - Ygls') ;                  % compute intermediate errors
    xfrmErr = (errs*Omega_1*errs')/np  ;  % intermediate MSE
    if (abs(lastErr-xfrmErr)<=maxERR) % lowest yet !!!
        break
    end
    lastErr = xfrmErr ;
    Omega_1 = (diag(errs.^2,0)) ;   % compute next Omega if necessary
end

% 4) NOW begin iterative Lambda-tuning process
lambda = calc_lamda(J,df,ni) ;  % compute initial min-var Lambda
muscale = 1.5 ;        % initial lambda scaling factor to hunt for ideal
prevSSE = np ;                     % initialize previous errors
prevlambda = lambda ;  % needed for larger RMSE on first iteration
for count = 1:1000
    ww = (invprod+(lambda.*eye(ni)))\numprod ;  %compute initial
weights
    currSSE = sum((out' - (J*ww)').^2) ;   % compute current errors
    if (currSSE < prevSSE)        % condition for shrinkage
        prevlambda = lambda ;
        lambda = lambda - lambda/muscale ;   % shrink lambda
        prevSSE = currSSE ;
    elseif (currSSE > prevSSE)  % condition for overshoot...
        lambda = prevlambda ;
```

```
        muscale = muscale*2 ;
    else
        break                   % we hope we are done
    end
end
```

7.7 Appendix G – MATLAB Code: Two Types of PLM Implementations

7.7.1 The PolyNet Variant – PolyNet.m excerpts

```
function [ww,indfinal,npp,"..."] =
PolyNet(trins,trouts,maxord,Pthresh"...")

[np,nd]=size(trins);
agu=[ones(np,1),trins];  %agumented input space  (+1 is added in the
column)

pterms = 0 ;    % initialize number of monomials
ord = -1 ;       % initialize polynomial order
orders = [] ;    % initialize vector of orders per solution
[...]
rmseTr = [] ;   % per solution RMSEs
npp = [] ;       % initialize count of poly terms
while ((pterms <= Pthresh) && (ord < maxord))
    ord = ord + 1 ;
    pterms=factorial(ord+nd)/(factorial(ord)*factorial(nd) ) ;
    if (pterms <= Pthresh)
        Nmax=(nd+1)*ones(1,ord+1);
        ind=Get_In_pol(Nmax) ;
        [Nt,Nd]=size(ind);

        %% calculating polynomial terms
        J = [] ;
        for jj=1:Nt
            Pr=1;
            for i=1:Nd
                Pr=Pr.*agu(:,ind(jj,i));
            end
            J = [J Pr];
        end
        wwtemp = lin_reg(J,trouts) ;  % Finding weights
        nonzeroidx = find(wwtemp) ;

        % Now reduce all matrices considerably :) :)  :)
        ww{:,ord+1} = wwtemp(nonzeroidx) ;
        Jreduced{:,:,ord+1} = J(:,nonzeroidx) ;
        indfinal{:,:,ord+1} = ind(nonzeroidx,:) ;

        %% verifying with training points and getting errors
        Xappx = Jreduced{:,ord+1}*ww{:,ord+1} ;
        rmse = sqrt(sum(sum((trouts - Xappx).^2))/np) ;
        npp = [npp length(nonzeroidx)] ;
        orders = [orders ord+1] ;
        rmseTr = [rmseTr rmse] ;
```

```
    end
end
```

7.7.2 The PolyPaP Variants – PlyPaPGLSR.m excerpts

```
function [ww,indfinal,npp,maxorders,"..."] ...
          = PlyPaPGLSR(trins,trouts,maxord,Pthresh,"...")

[np,nd]=size(trins);
agu=[ones(np,1),trins];  %agumented input space  (+1 is added in the
column)

% &%&%&%&%&%&%&%&%  INITIAL STUFF FOR PROBE SEQUENCE .....
tic ;   % prepare to capture probing time ...
pterms = 0 ;    % initialize number of monomials
ord = -1 ;       % initialize polynomial order
rmsePr = 9999 ;   % Initialize "lowest" probe RMSE
while ((pterms <= Pthresh) && (ord < maxord))
    ord = ord + 1 ;
    pterms=factorial(ord+nd)/(factorial(ord)*factorial(nd) ) ;
    if (pterms <= Pthresh)
        Nmax=(nd+1)*ones(1,ord+1);
        ind=Get_In_pol(Nmax) ;
        [Nt,Nd]=size(ind);

        %% calculating weights
        J = [] ;  % DON'T FORGET the zero-order term..
        for jj=1:Nt
            Pr=1;
            for i=1:Nd
                Pr=Pr.*agu(:,ind(jj,i));
            end
            J = [J Pr];
        end

        wwtemp = GLSminvar_reg(J,trouts,Nd) ;   % Use min-var ridge to
probe min RMSE point..
        XappxPr = J*wwtemp ;
        rmse = sqrt(sum(sum((trouts - XappxPr).^2))/np) ;
        if (rmse < rmsePr)
            PrOrder = ord ;     % set optimized probe order
            rmsePr = rmse ;     % set new optimized probe RMSE
        end
    end
end
PrTime = toc ;       % capture training time for probe sequence
% &%&%&%&%&%&%&%&%&%&%&%&%& END PROBE SEQUENCE

maxmeansslope = 1e-2 ;
trpct = 0.75 ;        % percent of training data to pull into the
iterative training

trlength = ceil(trpct*np) ;     % establish number pf patterns to
include for each spin
maxorders = [] ;    % initialize vector of maxorders per solution
```

```
npp = [] ;        % initialize count of poly terms

Nmax=(nd+1)*ones(1,PrOrder); % GENERATE all terms at ProbeOrd order...
ind=Get_In_pol(Nmax) ;
[Nt,Nd]=size(ind) ;          % get indices size for current round

tridx = 1 ;                  % initial training index value to store
output vectors
while (Nt >= 1)
    tic ;    % start clocking time for this node's training
    % ENTER the iterative OLS regression phase:
    tstwwmeans = 0 ;     % initialize running mean
    tstwwabsmax = [] ;    % initial running max abs value of coeffs
    tstwwscstds = 0 ;
    tstwwscmeans = 0 ;
    prevwwscaled = 0 ;     % initialize k-1 normalized weights
    avscmeansslope = 0 ;
    for k = 1:100     % drill down to the essential output weights for
each instance
        agusubset = agu(1:trlength,:) ;      % take subset of training
patterns
        troutsbst = trouts(1:trlength,:) ;

        %% building monomial terms
        Jsub = [] ;
        for jj=1:Nt
            Pr=1;
            for i=1:Nd
                Pr=Pr.*agusubset(:,ind(jj,i));
            end
            Jsub = [Jsub Pr];
        end

        wwtemp = GLSridge_reg(Jsub,troutsbst,Nd) ;  % Finding weights,
OLS with iterative ridge regression
        tstwwmeans = (wwtemp + tstwwmeans.*(k-1))./k ;               %
compute running means of raw coefficients
        tstwwabsmax = max(abs([tstwwabsmax wwtemp]),[],2) + 1e-16 ; %
compute running max-abs of raw coefficients

        tstwwscaled = wwtemp./tstwwabsmax ;      % NORMALIZE raw
coefficients per current max-abs
        prevscmeans = tstwwscmeans ;             % capture preceding
normalized ???
        tstwwscmeans = (tstwwscaled + tstwwscmeans.*(k-1))./k ;
        scmeansnn_1 = sum(abs(tstwwscmeans - prevscmeans)) ;
        avscmeansslope = abs(avscmeansslope + scmeansnn_1)/k ;

        tstwwscstds = sqrt( (((tstwwscstds.^2)./k + ((tstwwscmeans -
prevscmeans).^2)).*(k-1)) ) ;

        if (avscmeansslope <= maxmeansslope)
            break
        end
        shuffle = randperm(np) ;
        agu = agu(shuffle,:) ;
        trouts = trouts(shuffle,:) ;
```

```
    end
    finalk = k

    wwtemp = tstwwmeans ;     % assign last wwmeans as final coeffs
(this is the way to go 7/3/13)
    ["..."]
    %% checking weights:  Keep only poly vectors with non-zero
coefficients
    nonzeroidx = find(wwtemp) ;      % necessary one more time...
    ["..."]
    tridx = tridx + 1 ;          % increment the trial index for output
vectors

    % &%&%&%&%&%&%&%&%&%&%&%&%&%&%&%&%&%&%&%&%&%&%&%&%&%&
    % TRIM the noisiest term from the poly set, run again
    [maxwwscstd maxstdidx] = max(tstwwscstds) ;  % find the current
"noisiest" coefficient term
    ind(maxstdidx,:) = [] ;                     % ...remove it
    Nt = Nt - 1 ;                               % ...reduce total poly
term count by 1
end
```

7.8 Appendix H – MATLAB Code: An N-Dimensional Data-Driven TSK Fuzzy System

7.8.1 Efficient Recursive Interpolation – recurinter.m excerpts

```
function [outputs] = recurinter(intrain,intpts,varargin)
% This is a program which recursively interpolates new output values
for
% any input set of points (one or many), given randomly spaced
% multi-dimensional input data as "training points".
%
% INPUTS:
%           intrain:    -- input "training points" of dataset
%           intpts:     -- set of point locations to interpolate, eg:
%                           [x1,y1,...,w1;
%                            x2,y2,...,w2;
%                            xnp,ynp,...,wnp]
%           varargin{1}=dim     -- current dimension to sort and thin
%           varagrin{2}=lastdim   -- last dimension to operate upon
%           varargin{3}=radius -- current 1-D "radius" for
identification
%                                       of candidate vectors
%           varargin{4}=mode -- '0'==interior point search
%                               '1'==boundary point search
% OUTPUT:
%           outputs:     -- the interpolated points with final 'Z'
values

inithalfspan = 0.00625 ;     % initial search radius for points to
interpolate

%% MAIN:  Recursive Interpolation
if (nargin==2)      % If initial invocation, wrapper tasks ensue...
```

```
    [irows,icols] = size(intpts) ;
    maxhfspan = 2*sqrt(icols) ;     % mark greatest span across
hypercube..
    outputs = [] ;      % initialize final set of outputs
    for row = 1:irows
        tempsets = [] ; % initialize final subset of training pts to be
averaged
        tmpvectors = 0 ;% initialize number of tempsets members
        halfspan = inithalfspan ;     % initial 1-D "radius"
        mode = 0 ;  % we start by assuming non-boundary point search

        while (tmpvectors < icols)  % require at least one datapoint
per dimension..
            halfspan = halfspan*1.1 ;
            if (halfspan > maxhfspan)
                mode = 1 ; % now we are at an "edge" and search as
boundary point
                halfspan = inithalfspan*1.1 ;
            end
            tempsets =
recurinter(intrain,intpts(row,:),1,icols,halfspan,mode) ;
            [tmpvectors, tmpcols] = size(tempsets) ;
        end
        outputs = [outputs; [intpts(row,:)
computeval(tempsets,intpts(row,:))]] ;
    end
    return ;
end

% vvvvv if this is a recursive call: vvvvv
dim = varargin{1} ;      % current dimension
lastdim = varargin{2} ; % last input dimension
radius = varargin{3} ;  % 1-D "radius"
mode = varargin{4} ;    % '0' for in-bound point search, '1' for out-
of-bounds search
if ( dim<=lastdim )
    nextidx = (intrain(:,dim)>=(intpts(dim)-
radius))&(intrain(:,dim)<=(intpts(dim)+radius)) ;
    subset = intrain(nextidx,:) ;
    dim = dim + 1 ;
    if (dim > lastdim)
        if (~isempty(subset) && (mode==0))
            for p = 1:lastdim
                testo = find(subset(:,p)>intpts(p), 1) ;
                if isempty(testo)
                    subset = [] ;
                    break
                end
                testo = find(subset(:,p)<intpts(p), 1);
                if isempty(testo)
                    subset = [] ;
                    break
                end
            end
        end
        outputs = subset ;
    else
```

```
        outputs = recurinter(subset,intpts,dim,lastdim,radius,mode) ;
    end
end
return  % recurinter()

%% COMPUTE final interpolated value for current testpoint
function ruleval = computeval(subset,tstpt)
% Support function for recurinter:  Computes the interpolated output
value
% once the complete subset of points for the current interpolation
point is
% identified.  Avoids square root computation.
%
% INPUTS:
%            subset:         -- the set of valid poitns for which the
%                               interpolated value is computed
%            tstpt:          -- the current point at which to compute
the
%                               value
% OUTPUT:
%            ruleval:        -- the interpolated output value for the
%                               current point location

[rows cols] = size(subset) ;
if (rows == 1)
    ruleval = subset(1,cols) ;
else
    rawdists2 = [] ;
    for row = 1:rows
        rawdists2 = [rawdists2; (tstpt - subset(row,1:(cols-1))).^2] ;
    end

    sumdists = sum(rawdists2,2) ;
    sumalldists = sum(sumdists).*ones(rows,1) ;

    Zprods = subset(:,end).*(sumalldists - sumdists) ;
    ruleval = sum(Zprods)/(sumalldists(1)*(rows-1)) ;

    if isnan(ruleval)
        ruleval = 0 ;
    end
end
return  % computeval()
```

7.8 2 The TSK Fuzzy System Engine

Training:

```
function [fuzztables,npp,"..."] ...
    = NFuzzyT1(trins,trouts,FTmax,F1Dmax)
% INPUTS:
%        trins:      the inputs to the dataset you would like to train
%        trouts:     the training outputs corresponding to the inputs
%        FTmax:      max allowed number of output fuzzy table values
%        F1Dmax:     max allowed number of table breakpoints along 1-D
```

```
trainset = [trins trouts] ;          % "training" set

[np,nd]=size(trins);
dim = nd ;
nF1D = 0 ;       % initialize number of breakpoints along 1 table
dimension
FTterms = 0 ;    % initialize total number of table elements

rmseTr = [] ;    % per solution RMSEs
npp = [] ;       % initialize count of fuzzy table elements
while ((FTterms <= FTmax) && (nF1D < F1Dmax))
    nF1D = nF1D + 1 ;
    FTterms = nF1D^nd
    if (FTterms <= FTmax)
        % vvvvv set up fuzzy table points, same coordinates for all
dimensions:
        % NOTE:  final set of coords will be
(#breakpoints)^(#dimensions), beware..
        span = 2/(nF1D-1) ;
        if (isinf(span))
            tblpts = [0] ;
        else
            tblpts = [-1:span:1];
        end
        dim = nd ;      % this initializes for each new fuzzy table..
        % create set of all multi-dim fuzzy table points:
        tblcoords = [] ; % initialize table of fuzzy output value
*locations*
        tblidx = [] ;
        expo = 0 ;       % initial power of 2 exponent
        while(dim > 0)
            vect = [] ;
            idxvect = [] ;
            for i = (0.00001):1/(nF1D^expo):nF1D
                vect = [vect tblpts(ceil(i))] ;
                idxvect = [idxvect ceil(i)] ;
            end
            expo = expo + 1 ;
            fullvect = [] ;
            fullidxvect = [] ;
            for j = 1:(nF1D^(dim-1))
                fullvect = [vect fullvect] ;
                fullidxvect = [idxvect fullidxvect] ;
            end
            dim = dim - 1 ;
            tblcoords = [fullvect' tblcoords] ;
            tblidx = [fullidxvect' tblidx] ;
        end

        tblidx = num2cell(tblidx) ;
        [tblrws,tblcols] = size(tblcoords) ;

        % compute the fuzzy output table values and assign to an
indexable matrix:
        intrpouts = recurinter(trainset,tblcoords) ;
```

```
        for p = 1:tblrws
            fuzztable(tblidx{p,:}) = intrpouts(p,end) ;
        end
        fuzztables{:,nF1D} = fuzztable ;

        % ***** "training" verification -- COMPUTE fuzzy outputs...
*****
        fuzzouts = nDfuzz_out(fuzztable,trins) ;

        rmse = sqrt(sum(sum((trouts - fuzzouts).^2))/np) ;
        npp = [npp FTterms] ;
        rmseTr = [rmseTr rmse] ;
    end
end
```

Fuzzy Table Output Generation:

```
function [fuzzouts] = nDfuzz_out(fuzztbl,validins)
% Author:  Michael S. Pukish
% Modification Date:  09/21/13
% Description:  Computes n-D fuzzy system outputs given a set of input
points
%   and the normalized n-D fuzzy output table.
%   INPUTS:
%       fuztbl:         -- the n-D fuzzy output table
%       validins:       -- row vector points to solve for
%   OUTPUTS:

tblres = size(fuzztbl,1) ;                  % capture resolution along one
axis
yint = (tblres+1)/2 ;                       % compute y-intercept for all
fuzzified axis conversions
yslope = (tblres-1)/2 ;

[vrows inlength] = size(validins) ;        % capture max count & length of
vectors

fuzzouts = [] ;                    % initialize fuzzy outputs
for vidx = 1:vrows
    vect = yslope.*(validins(vidx,:)) + yint ;    % capture/convert
current vector to solve
    floors = floor(vect) ;        % get integer bounds of each dimension
    ceils = ceil(vect) ;          %   ...

    % vvvv build up solution subset from fuzz table vvvv :
    setstr = 'subset=fuzztbl(' ;
    for m = 1:inlength
        if (( vect(m)>1 ) && ( vect(m)<tblres ))
            em = num2str(m) ;
            setstr = [setstr 'floors(' em '):ceils(' em '),'] ;
        elseif ( vect(m)<=1 )
            setstr = [setstr '1,'] ;
        else
            setstr = [setstr num2str(tblres) ','] ;
        end
```

```
    end
    setstr(end) = []; setstr = [setstr ');'] ;
    eval(setstr) ;               % resolves subset for current input
vector

    % now, build the products...
    if (( vect(1)>1 ) && ( vect(1)<tblres ))    % initialize the first-
D fuzzy distances..
        fzzfact = mod(vect(1),1) ;                % ...compute fuzzy
distance factor for current
    else
        fzzfact = 1 ;
    end
    fuzzprods = nonzeros([(1-fzzfact) fzzfact]) ;  % initialize the
product values
    for d = 2:inlength
        if (( vect(d)>1 ) && ( vect(d)<tblres ))
            fzzfact = mod(vect(d),1) ;
        else
            fzzfact = 1 ;
        end
        twonew = nonzeros([(1-fzzfact) fzzfact]) ;  % next dimension to
examine..
        tempprods = [] ;                        % initialize current 1-D
product sequence
        for i = 1:length(twonew)
            for e = 1:length(fuzzprods)
                tempprods = [tempprods twonew(i)*fuzzprods(e)] ;
            end
        end
        fuzzprods = tempprods ;
    end

    fuzzouts = [fuzzouts; fuzzprods*subset(:)] ;  % build columned
output vector
end
```

7.9 Appendix I – MATLAB code: Fast, Forward-Computing N-Dimensional Radial Clustering

```
function [outclusidx] = clstrsets(inptrns,maxrad,minclstrs,minclslen)
% Parses an input set of training patterns into multiple output
clusters.  Only
% indices of the original data set are reported, according to the input
pattern
% indices (one vector per row).
%
% INPUTS:   inptrns:    -- row vectors of given input pattern set
%           maxrad:     -- maximum (starting) radius of output clusers
%           minclstrs:  -- minimum number of output clusters
%           minclslen:  -- minimum length allowed per cluster
%
% OUTPUTS:  outclusidx: --

[n,m]=size(inptrns);  % determine dimensions of dataset array
```

```
for k = (maxrad*100):-1:1,     % vary test radii from marad downto 0.01
in steps of 0.01
    % np: number(s) of patterns per clusters, p: location(s) of cluster
centers
    r=k/100; np=[]; p=[]; p(1,:)=inptrns(1,:); np(1)=1; nc=1;  %
initialize values
    clstridx{1} = 1 ;
    for i=2:n, % for all patterns...
        nt=0;
        for j=1:nc,  % per each cluster in existence...
            % compute distance btw cluster center and present pattern,
compare
            dd=p(j,:)- inptrns(i,:); d=sqrt(dd*dd');
            if (d < r) % if distance of present pattern < current
threshold...
                % update weights for pressnt cluster's neuron
                p(j,:)=(inptrns(i,:)+p(j,:)*np(j))/(np(j)+1) ;
                clstridx{j} = [clstridx{j} i] ;
                np(j)=np(j)+1;  % increment pattern count for this
cluster...
                break;
            end;
            nt=nt+1;             % ... otherwise,
        end;
        if nt==nc,               % ... create a new cluster
            nc=nc+1; p(nc,:)=inptrns(i,:); np(nc)=1; % increment #
clusters,
            clstridx{nc} = i ;
        end;
    end;
    idxcnt = 1;
    for q=1:nc
        if (length(clstridx{idxcnt}) < minclslen)
            putsetidx = clstridx{idxcnt} ;
            clstridx(idxcnt) = [] ;
            p(idxcnt,:) = [] ;
            nc = nc - 1 ;
            clstridx = putback(clstridx,putsetidx,r,inptrns,p,nc) ;
        else
            idxcnt = idxcnt + 1 ;
        end
    end
    if (nc > minclstrs)
        break
    end
end;

outclusidx = clstridx ;
return

% helper function -- putback finds a new home for clusters that are too
% small
function [clsidxout] =
putback(clsidxin,pbset,initr,allpat,clsctrs,numclus)
pbsetlen = length(pbset) ;
```

```
for i = 1:pbsetlen
    currad = initr ;
    found = false ;
    while (~found)
        currad = currad + 0.01 ;
        for j = numclus:-1:1
            dd=clsctrs(j,:)- allpat(pbset(i),:); d=sqrt(dd*dd');
            if d<currad,  % if distance of present pattern < current
threshold...
                clsidxin{j} = [clsidxin{j} pbset(i)] ;
                found = true ;
                break;
            end;
        end
    end
end

clsidxout = clsidxin ;
return
```

7.10 Appendix J – Selected Publications by This Author

Journal Articles

D. Hunter, H. Yu, M. S. Pukish, J. Kolbusz, and B. M. Wilamowski, “Selection of Proper Neural Network Sizes and Architectures -- A Comparative Study,” Industrial Informatics, IEEE Transactions on, vol. 8, no. 2, pp. 228 –240, May 2012.

M. S. Pukish, P. Różycki, and B. M. Wilamowski, “Advances in Polynomial-Based Learning Machines for Industrial Electronics,” Industrial Informatics, IEEE Transactions On, May 2014. [submitted and under review]

Conference Papers

M. S. Pukish, S. Wang, and B. M. Wilamowski, “Segmentation of Cerebral Cortex MRI Images with Artificial Neural Network (ANN) Training,” in 2013 The 6th International Conference on Human System Interaction (HSI), 2013, pp. 320–327.

Michael S. Pukish, Philip Reiner, and Xing Wu, “Recent Advances in the Application of Real-Time Computational Intelligence to Industrial Electronics,” presented at the IECON2012, Montreal, Canada, 2012.

Pukish, Michael S., Gnanachchelvi, Parameshwaran, and Wu, Xing, “Recent Developments in Wireless Hardware Design, Modeling, and Analysis for Industrial Applications,” presented at the IEEE IECON2012, Montreal, Canada, 2012.

Parameshwaran Gnanachchelvi, Jiao Yu, and Michael Pukish, “Current Trends in In-vehicle Electrical Engineering Applications,” presented at the IEEE IECON2012, Montreal, Canada, 2012.

Xing Wu, Hao Yu, Tiantian Xie, and Michael S. Pukish, “Current Trends in Industrial Control,” presented at the IEEE IECON2012, Montreal, Canada, 2012.

Joseph Cali, Xueyeng Geng, Michael Pukish, and Jianjun Yu, “A 1 GHz DDFS for Stretch Processing Radar in 130nm CMOS process,” 2012.

Jianjun Yu, Joseph Cali, Feng Zhao, and Michael Pukish, “An X-Band Chirp Radar Transmitter with Direct Digital Synthesis in 0.13um BiCMOS,” 2012.

Yuan Yao, Jianjun Yu, Michael Pukish, Siyu Yang, and Zachary Hubbard, “A 12-bit 60MS/s Inter-leaved Pipeline A/D Converter with Op-amp Sharing Techniques,” 2012.