

## Is Nonparametric Learning Practical in Very High Dimensional Spaces?

**Gregory Z. Grudic**

Dept. of Elect. and Comp. Eng.  
University of British Columbia  
Vancouver, B.C., V6T 1Z4, Canada

gregg@ee.ubc.ca

<http://www.ee.ubc.ca/~gregg>

**Peter D. Lawrence**

Dept. of Elect. and Comp. Eng.  
University of British Columbia  
Vancouver, B.C., V6T 1Z4, Canada

peterl@ee.ubc.ca

<http://www.ee.ubc.ca/~peterl>

### Abstract

Many of the challenges faced by the field of Computational Intelligence in building intelligent agents, involve determining mappings between numerous and varied sensor inputs and complex and flexible action sequences. In applying nonparametric learning techniques to such problems we must therefore ask: "Is nonparametric learning practical in very high dimensional spaces?" Contemporary wisdom states that variable selection and a "greedy" choice of appropriate functional structures are essential ingredients for nonparametric learning algorithms. However, neither of these strategies is practical when learning problems have thousands of input variables, and tens of thousands of learning examples. We conclude that such nonparametric learning is practical by using a methodology which does not use either of these techniques. We propose a simple nonparametric learning algorithm to support our conclusion. The algorithm is evaluated first on 10 well known regression data sets, where it is shown to produce regression functions which are as good or better than published results on 9 of these data sets. The algorithm is further evaluated on 15 large, very high dimensional data sets (40,000 learning examples of 100, 200, 400, 800 and 1600 dimensional data) and is shown to construct effective regression functions despite the presence of noise in both inputs and outputs.

### 1 Introduction

Nonparametric learning algorithms are designed for learning problems where relatively little information is available *a priori* about the what type of model structure is required, or which set of variable inputs are important. Such problems are common in many domains of computational intelligence. As a simple example, consider the problem of autonomous navigation in a cluttered environment. As researchers we must decide which features (inputs) in the environment are important, and what kind of computational models are required.

Nonparametric learning techniques are of potential value to us because they are designed to define their own model structure, and to determine which set of inputs are most relevant to effectively predict the task output. There are a number of effective non-parametric learning techniques in the literature, including C4.5 [Quinlan, 1993], CART [Breiman *et al.*, 1984], and MARS [Friedman, 1991] (See [Michie *et al.*, 1994] for a more complete list).

Almost without exception, contemporary implementations of nonparametric learning algorithms have two common properties. First, some form of variable selection is done to determine which of all possible inputs best predict the output. And second, they utilize some type of "greedy" search procedure which attempts to pick the best model structure, from some set of possible structures. Such techniques are feasible when there are relatively few input variables ( $< 100$ ). However, neither of these two algorithmic properties are practical when large numbers of potential inputs are available. Consider the learning example where there are 100 potential inputs, and assume that not all of these inputs are necessary to predict the output. In order to pick the best 50 of these 100 inputs, one would need to construct  $\binom{100}{50} \cong 1 \times 10^{29}$  models, and then evaluate them all to see which set of 50 inputs best predict the output. Such an exhaustive search is currently not feasible, and is unlikely to be in the near future. Similarly, if we use some greedy search procedure to determine which of all possible model structures is the best, we would further increase our search space by the total number of possible model structures.

Certainly, the flexibility of nonparametric learning recommends it as a useful tool for constructing intelligent agents. This has been clearly demonstrated in numerous publications [Michie *et al.*, 1994]. However, in almost all applications of nonparametric learning, the number of inputs have been relatively low ( $< 100$ ), and the number of learning examples have been relatively few ( $< 20,000$ ). Thus, given that many challenges faced by the field of Computational Intelligence in building intelligent agents, involve determining mappings between numerous (thousands or more) and varied sensor inputs and complex and flexible action sequences requiring

tens of thousands of training examples, we must ask whether nonparametric learning can be usefully applied to such problems? Given our argument in the previous paragraph, the initial response to this question is no: the computational complexity associated with applying existing algorithms to such large problems makes them impractical. In fact, *most* currently published nonparametric learning algorithms, utilize search strategies which are impractical in high dimensional spaces. Even algorithms such as projection pursuit [Friedman and Stuetzle, 1981], which do not perform traditional exhaustive searches, build models using all input variables simultaneously, making them impractical for very high dimensional problems.

In an effort to develop methods to address the high dimensional mappings required for intelligent agents, we challenge the need for variable selection and greedy function search in nonparametric learning. Instead, we propose an algorithm which has two basic characteristics: first, very little computational effort is spent on variable selection; and second, the model selection strategy does not explode in computational complexity, as the dimension of the problem increases. Our intent is to show that a model having these properties is capable of building practical, effective nonparametric models, for both small low dimensional learning problems, as well as larger, very high dimensional problems.

The basic premise behind the proposed algorithm is that very high dimensional regression can be done using a finite number of low dimensional structural units, which are added one at a time to the regression function. These structural units are by necessity low dimensional, because high dimensional structures suffer from the *curse of dimensionality* (see [Friedman, 1994] for a detailed discussion). The inputs to new structural units can be the outputs of previously added units as well as input variables. *However, both the choice of which structural unit to add next, and which inputs will act on these structural units, must be done using minimal computational effort for the algorithm to be practical in high dimensional spaces.*

In the current paper, we implement and evaluate the simplest algorithmic structure which is consistent with the ideas outlined above. The proposed algorithm has a very simple structure, using only one type of structural unit, and variables are added to the model in a random order. The surprising conclusion of this paper is that this simple, in essence random structure, produces highly effective nonparametric models.

Much of the work in computational learning has been directed towards the classification problem. However, many real problems in AI involve controlling agents which manipulate the world using continuous values control signals. Hence, in this paper we have chosen to directly address the continuous valued regression problem, instead of the discrete valued classification problem. However, as discussed in [Friedman, 1994], classification problems can be solved using regression techniques in a number of ways. Therefore the high dimensional

regression techniques studied in this paper are, at least in theory, applicable to classification problems.

In the remainder of this paper we give a detailed description of the proposed algorithm, followed by an extensive numerical evaluation. Our evaluation is done in 3 parts. First we compare the performance of our algorithm to published results on 10 well known regression problems from the literature. This gives us a good measure of the performance of the proposed algorithm on some standard regression problems. Next we test the algorithm on 15 large very high dimensional, highly nonlinear regression problems. This allows us to test the efficacy of the algorithm on the type of problems we are most interested in addressing. Our final evaluation of the algorithm is on the 10-bit parity problem (or equivalently the 10 input XOR problem). This problem was chosen to show that the proposed algorithm is capable of solving problems which have an intrinsic dimension (10 inputs) greater than that of its highest dimensional functional unit (2 dimensional). A common criticism of algorithms which build functions using low dimensional structural units is that they cannot model high dimensional XOR functions. In this paper, we demonstrate that this is not true for the proposed algorithm.

## 2 The Proposed Learning Algorithm

Let  $R_L(x_1, \dots, x_N)$  be an  $N$  dimensional regression function constructed using an  $L$  level cascade of 2 dimensional functions:  $y = R_L(\mathbf{x})$  estimates the dependent variable  $y$  given independent variables  $\mathbf{x} = (x_1, \dots, x_N)$ . Then  $R_L(\mathbf{x})$  has the following form:

$$R_L(\mathbf{x}) = \alpha_1 \cdot g_1(x_{k_0}, x_{k_1}) + \alpha_2 \cdot g_2(g_1, x_{k_2}) + \dots + \alpha_L \cdot g_L(g_{L-1}, x_{k_L}) \quad (1)$$

where  $L$  defines the number of levels in the cascade or, equivalently, the number of 2 dimensional functions  $g_l(u, v)$  constructed;  $\alpha_l$  are real valued scaling factors; and, the subscripts  $k_0, \dots, k_L \in \{1, \dots, N\}$  serve to identify the independent variables  $(x_1, \dots, x_N)$ . For the purposes of this paper, the 2 dimensional functions  $g_l(u, v)$  are represented as 3rd order 2 dimensional polynomials:

$$g_l(u, v) = a_1 + a_2u + a_3v + a_4uv + a_5u^2 + a_6v^2 + a_7u^2v + a_8uv^2 + a_9u^3 + a_{10}v^3 \quad (2)$$

where the coefficients  $a_1, \dots, a_{10}$  are defined as each level is constructed. We postulate that this choice for representing  $g_l(u, v)$  is not unique (initial experiments indicate that higher order polynomials do not significantly affect results), however, as demonstrated in Section 3, it is an effective one.

We assume that  $M_L$  learning input/output examples  $(\mathbf{x}_i, y_i)$ , for  $i = 1, \dots, M_L$  have been divided into 2 sets: a training set  $\Gamma_T$  containing  $M_T$  input/output pairs, and a validation set  $\Gamma_V$  containing  $M_V$  input/output pairs. In Section 3 we describe how the training and validation sets are obtained. There are 3 preset learning parameters, symbolized

by  $depth_1$ ,  $depth_2$ , and  $\epsilon$ , which are used during the construction of the regression function. For all experimental results presented, these learning parameters are set to the same theoretically motivated and *data independent* values. The function of each learning parameter is explained in the following algorithmic description, and is further elaborated on below. A single cascade of  $g_l(\cdot)$  functions is constructed in a series of sections. When the current section  ${}^i R_j = \sum_{l=i}^j \alpha_l \cdot g_l(\cdot)$  of the cascade can no longer reduce mean squared error, learning outputs  $y$  are replaced by residual errors due to  ${}^i R_j$  and a new section is begun starting from the last error reducing function  $g_j(\cdot)$ . Stated in detail, the construction of the regression function,  $R_L(\mathbf{x})$  (equation (1)), proceeds as follows:

**STEP 1: Initialize algorithm:** Initialize the 2 counters  $i = 1$  and  $p = 1$ , where  $i$  and  $p$  are used as follows:  $i$  indexes the function  $g_i(\cdot)$  at the start of the current section;  $p$  is a subscript indicating that section  $p$  of the cascade will be constructed in step 2. Throughout this algorithm, the subscripts  $k_0, \dots, k_L$  (used to identify the independent variables) are randomly selected, one at a time, from the set  $\{1, \dots, N\}$ , *without* replacement. When this set is empty, it is re-initialized to  $\{1, \dots, N\}$ , and the process of selecting these subscripts continues.

**STEP 2: Construct new section:** Construct, in order, the functions  $g_i(\cdot), g_{i+1}(\cdot), \dots, g_j(\cdot)$ , where  $j - i > depth_1$ , as follows:

1. Obtain a bootstrap sample set  $\Gamma_T^B$  (i.e.  $M_T$  randomly chosen samples with replacement) from the training set  $\Gamma_T$ .
2. Given  $\Gamma_T^B$ , use Singular Value Decomposition to determine the coefficients  $a_1, \dots, a_{10}$  of equation (2).

The construction stops with function  $g_j(\cdot)$  when the change in cascade error on the validation set  $\Gamma_V$ , over the past  $depth_1 + 1$  level additions, is less than some small, positive real number  $\epsilon$ . This is defined as follows:

$$1 - \frac{MSE({}^i R_j)_{\Gamma_V}}{\frac{1}{depth_1} \sum_{q=j-1-depth_1}^{j-1} MSE({}^i R_q)_{\Gamma_V}} \leq \epsilon \quad (3)$$

where  ${}^i R_q = \sum_{l=i}^q \alpha_l \cdot g_l(\cdot)$  is the constructed cascade between levels  $i$  and  $q$ ; and  $MSE({}^i R_q)_{\Gamma_V}$  is the mean squared error of this cascade on the *validation* set  $\Gamma_V$ . With the addition of each level, the scaling factors  $\alpha_l$ , for  $l = i, \dots, j$ , are recomputed as follows:

$$\alpha_l = \frac{(MSE(g_l(\cdot))_{\Gamma_T})^{-1}}{\sum_{q=i}^j (MSE(g_q(\cdot))_{\Gamma_T})^{-1}} \quad (4)$$

where  $MSE(g_l(\cdot))_{\Gamma_T}$  is the mean squared error of the single function  $g_l(\cdot)$  on the training set  $\Gamma_T$ . Hence,  $\alpha_l$  is simply the inverse of the normalized mean squared error of level  $l$ , and its purpose is to give more weight to levels which contribute more significantly to error reduction.

**STEP 3: Prune section:** We now prune the current section back to the level which has the best mean squared error, as follows. Find the *minimum* level  $s \in \{i - 1, i, \dots, j\}$  which gives the least mean squared error of the cascade on the validation set  $\Gamma_V$ , in the following sense:

$$\text{choose } s < t \text{ s.t. } \forall t \in \{i, \dots, j\} \Rightarrow MSE({}^i R_s)_{\Gamma_V} \leq MSE({}^i R_t)_{\Gamma_V} \quad (5)$$

Delete all functions  $g_{s+1}(\cdot), g_{s+2}(\cdot), \dots, g_j(\cdot)$ . Set  $j = s$  and, using equation (4), recalculate the scaling factors  $\alpha_l$ . The current state of the cascade is now given by  $R_s(\mathbf{x}) = \sum_{l=i}^s \alpha_l \cdot g_l(\cdot)$ . Note that if  $s = i - 1$  then all the functions just constructed in step 2 are deleted.

**STEP 4: Update learning outputs:** Update the output values,  $y_i$ , of both the validation set  $\Gamma_V$  and training set  $\Gamma_T$  as follows:  $\forall(\mathbf{x}, y) \in \Gamma_V \cup \Gamma_T \Rightarrow y = y - \sum_{l=i}^s \alpha_l \cdot g_l(\cdot)$ . The sets  $\Gamma_V$  and  $\Gamma_T$  now contain the residual approximation errors after the cascade has been constructed to pruned level  $s$ . Let  $\rho_p$  be the variance of the output values,  $y_i$ , of the validation set  $\Gamma_V$  at the completion of section  $p$ . Thus  $\rho_p$  is equivalent to the mean squared error of the approximation  $R_s(\mathbf{x})$  on the validation set.  $\rho_p$  is used in the next step to establish a condition under which the construction of the cascade stops.

**STEP 5: Check stopping condition:** Go back to **STEP 2**, constructing at least  $depth_2 + 1$  sections, until the following condition is met:

$$1 - \frac{\rho_p}{\frac{1}{depth_2} \sum_{q=p-1-depth_2}^{p-1} \rho_q} \leq \epsilon \quad (6)$$

Therefore, the algorithm will terminate when the mean squared error fails to improve by a factor of  $\epsilon$ , over the last  $depth_2 + 1$  sections. Before executing **STEP 2**, set  $i = s + 1$  and increment the counter  $p$  to initialize the next section. Upon termination, set  $L = s$ , the total number of levels constructed.

As indicated above, the learning parameters  $depth_1$ ,  $depth_2$ , and  $\epsilon$ , are all used to define conditions under which the construction of the cascade stops. In particular,  $\epsilon$  defines the relative decrease in error, via the addition of further  $g_l(\cdot)$  functions, required in order for the cascade to continue to grow. Using  $\epsilon$ , the learning parameter  $depth_1$  defines how many levels of the cascade are used to determine the termination condition of **STEP 2**. Similarly  $depth_2$  and  $\epsilon$  are used in **STEP 5** to define how many times **STEP 2** is executed before the algorithm is terminated. Theoretically, the best regression function (with respect to least squared error) is obtained when  $\epsilon = 0$ , and  $depth_1$  and  $depth_2$  are very large. However, in order for the algorithm to terminate in a reasonable amount of time, for all of the results reported in this paper we fixed learning parameter values to:  $depth_1 = 25$ ,  $depth_2 = 6$ , and  $\epsilon = 0.0001$ . Hence the algorithm described above is completely automated, requiring no *a priori* parameter tuning.

Table 1: Low Dimensional Data Sets

Published Source	Data	Pub. Error	Prop. Alg. Error (100/10 runs)		
			best	ave.	s.d.
Breiman [Breiman, 1996]	Housing	11.6	1.95	9.4	8.4
	Friedman 1	6.1	1.77	2.88	0.43
	Friedman 2	22,100	16,733	19,603	1413
	Friedman 3	0.0242	0.014	0.0189	0.0024
Rasmussen [Rasmussen, 1996]	Auto Price	(0.29)	0.21	0.29	0.03
	Cpu	(0.17)	0.086	0.14	0.025
	House	(0.15)	0.128	0.145	0.007
	Mpg	(0.10)	0.088	0.100	0.004
	Servo	(0.15)	0.198	0.25	0.02
Jordan and Jacobs [Jordan and Jacobs, 1994]	For. Dyn.	(0.09)	0.0378	0.041	0.009

### 3 Results and Discussion

#### 3.1 Low Dimensional Data Sets

In this section we evaluate the efficacy of the proposed algorithm by applying it to 10 well known regression problems from the literature (Table 1).

For the data sets found in [Breiman, 1996] and [Rasmussen, 1996], the regression functions were constructed using 10 fold cross-validation: the Learning Data set was divided into 10 approximately equally distributed sets, and then 10 regression functions were constructed using, in turn, 9 of these sets as training sets, and the remaining set as a validation set. For each Test Data point (Test Data was *not* used during learning), the outputs of the 10 regression functions were averaged to produce the final approximation output, for which error results are reported. To test reproducibility, 100 independent approximations (independent with respect to random sequences of input variables and bootstrap samples as defined in Section 2) were generated using the Learning Data: Table 1 reports the best *Test Set* error, along with the average and standard deviation (s.d.) over the 100 independent runs.

For the data sets found in [Breiman, 1996], we followed the reported experimental setup. For each of the 4 data sets, 100 learning and 100 test sets were created randomly according to the guidelines given in [Breiman, 1996]. For each of the 100 learning sets a regression function was constructed (using 10 fold cross-validation as described above) and evaluated on the corresponding test set. Experimental results for the 100 experiments are given in Table 1. The previously published error for the [Breiman, 1996] data refers to the average bagged error reported.

For the data sets found in [Rasmussen, 1996], we report results for the largest learning sets only (we used 80 learning examples of auto price data, 104 of cpu data, 256 of housing data, 192 of mpg data, and 88 of servo data). For each of these 5 fixed data sets, the regression functions were constructed using 10 fold cross validation as described above. We generated 100 independent approximations in order to determine

what effect the stochastic aspect of the proposed algorithm has on its performance. As reported in Section 2, the ordering of the independent variables is random, and the construction of the 2 dimensional functions  $g_t(\cdot)$  is done using a (random) bootstrap sample of the training data. The best and average relative mean squared *test set* error, and its standard deviation, are reported in Table 1. From Table 1, it is evident that although there is some variation in error performance from run to run, the stochastic effect of the algorithm is mostly negligible. The previously published error given in Table 1 under the [Rasmussen, 1996] data sets is the *best* reported error (indicated by brackets) of the 5 algorithms evaluated, and was obtained from the graphs presented in the paper. For both the [Breiman, 1996] and [Rasmussen, 1996] data sets, the average learning time ranged from about 1 to 10 minutes per approximation (all learning times reported in this paper are for proposed algorithm running on a Pentium Pro 150 using LINUX). The average size of a single cascade was about 90 K-bytes.

Finally, for the forward dynamics data reported in [Jordan and Jacobs, 1994], our evaluation was done using the experimental setup described by Jordan and Jacobs for the on-line back-propagation algorithm: learning was done using 15,000 training examples, and learning stopped when the error could no longer be reduced on 5000 validation examples. The regression functions for this data consisted of a single cascade per output: due to the large data size, 10 fold cross validation was not necessary. In order to estimate the reproducibility of proposed algorithm on this data, we constructed 10 independent approximations. The best and average relative error on the validation set (in accordance with [Jordan and Jacobs, 1994]), and its standard deviation over these 10 independent experiments, is reported in Table 1. The previously published error, shown in Table 1, is the best relative error (on the validation data) of the 7 algorithms studied in [Jordan and Jacobs, 1994]. The forward dynamics data consists of 12 inputs and 4 outputs, which requires 4 of our regression functions (1 for each output). The algorithm required about 10 hours of com-

Table 2: High Dimensional Data Sets

Dim.	No Noise Data	Output Noise Data	Input/Output Noise Data
100	0.00139 s.d. 0.00006	0.1049 s.d. 0.0003	0.1048 s.d. 0.0004
200	0.0018 s.d. 0.0002	0.1064 s.d. 0.0003	0.1055 s.d. 0.0002
400	0.0009 s.d. 0.0002	0.1041 s.d. 0.0002	0.1064 s.d. 0.0003
800	0.0044 s.d. 0.0006	0.1111 s.d. 0.0003	0.1038 s.d. 0.0003
1600	0.0011 s.d. 0.0003	0.1087 s.d. 0.0003	0.1042 s.d. 0.0003

putation to build all 4 cascades, and the average size of each cascade was about 1 M-byte.

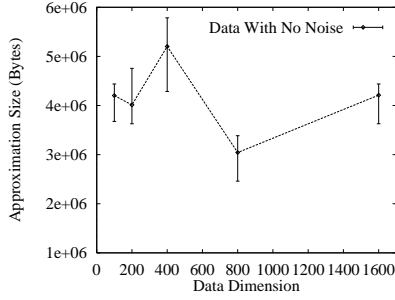


Figure 1: No Noise Data: Increase in Approximation Size with Dimension

From Table 1, it is evident that the proposed algorithm demonstrated as good or better error results on all but 1 (the servo data) of the data sets. *However*, this result should be interpreted with caution. The specific goal of the proposed algorithm is to build a regression function which best represents the learning data in the mean squared error sense. The regression function continues to grow in parameter size until the mean squared error can no longer be reduced: this is beneficial if one wants the “best” approximation, but detrimental if one wants a representation of fixed size. Most algorithms referred to in Table 1 are parametric and therefore of fixed size. Two exceptions are MARS and CART. However, even comparing these to the proposed algorithm should be done with caution: unlike our algorithm, both MARS and CART can be used to analyze the significance of various inputs and how they interact with one another.

### 3.2 High Dimensional Data

In this section, our goal is to study the proposed algorithm when applied to very high dimensional regression data, under various noise conditions. Since no large, high dimensional regression data examples were found in the literature, we applied the proposed algorithm to 15 artificial data sets ranging from 100 to 1600 input dimensions. For 5 of these data sets no noise was present, while the remaining 10 data sets had noise. For the first 5 of these “noisy” data sets, noise was present only in the output, while the remaining 5 data sets

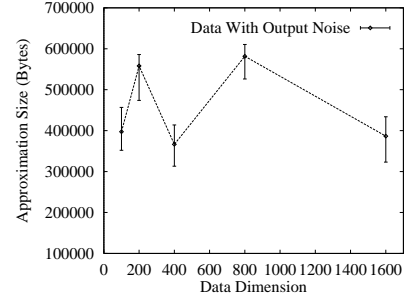


Figure 2: Output Noise Data: Increase in Approximation Size with Dimension

had both input and output noise. Input noise was generated by simply allowing only half of all inputs to contribute to the output (thus half of the inputs used in constructing the regression function had no effect on output). Output noise was generated using a normal distribution with the standard deviation selected to give a signal to noise ratio of 3 to 1, thus implying that the true underlying function accounts for 90% of the variance in learning and test data.

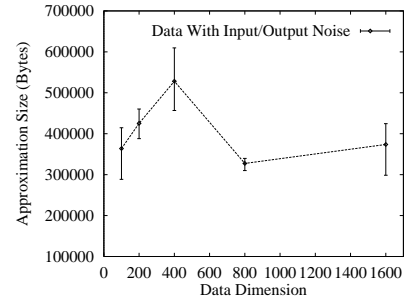


Figure 3: Input/Output Noise Data: Increase in Approximation Size with Dimension

The regression data was generated using the following function:

$$y = f(x_1, x_2, \dots, x_N) = \frac{\cos(\frac{1}{N} \sum_{i=1}^N \cos(r_i(x_i)))}{1.0 + \cos(\frac{1}{N} \sum_{i=1}^N \cos(s_i(x_i)))} \quad (7)$$

where  $N$  is the input dimension of the data, while  $r_i(\cdot)$  and

$s_i(\cdot)$  are continuous 1 dimensional functions which are non-linear and all different from one another. Equation (7) allows us to generate highly nonlinear data of any dimension, while at the same time controlling the *complexity* of the data via the appropriate selection of functions  $r_i(\cdot)$  and  $s_i(\cdot)$ . By *data complexity*, we are referring to the number of random sample points required to build sufficient non-parametric models of the data, with respect to least squared error. The more complex the generating function, the more data points are required for non-parametric modeling. For the purposes of this paper, we have chosen functions  $r_i(\cdot)$  and  $s_i(\cdot)$  such that 40,000 random sample points are sufficient in order to effectively model the data. For specific details on how  $r_i(\cdot)$  and  $s_i(\cdot)$  are constructed see <http://www.ee.ubc.ca/~gregg> for C code and documentation.

The simulation results for the 12 high dimensional data sets are presented in Table 2. For each data set there are 40,000 learning examples and 40,000 testing examples. Each learning set is further divided into 30,000 training examples and 10,000 validation examples: these are used to construct one cascade which forms the approximation for that data set. The 3 right most columns of Table 2 contain the test set average relative mean squared error and corresponding standard deviations over 10 independent runs. Relative mean squared error is defined, in the usual sense, as the mean squared error on the test data, divided by the variance of the test data. From Table 2 it is evident that the relative error is small when no noise is present. With the addition of noise the relative error approaches the theoretical limit (due to the 3 to 1 signal to noise ratio) of 0.1. Learning time for each data set with no noise was approximately 7 hours and produced cascades which were between 3 and 6 M-bytes in size. In contrast, learning time for each data set containing noise was about 1 hour and produced approximations of between 200 and 600 K-bytes in size. It is interesting to observe that the dimension of the data did not affect either learning time or the size of the approximation. This is clearly demonstrated in Figures 1 through 3, which show an increase in approximation size, as a function of the number of input variables. Note that the no noise data generated approximations which were much larger than those generated by data which had noise. This is because the proposed algorithm stops adding levels to the regression function when error can no longer be reduced on the validation set. This condition occurs much earlier when there is noise in the training data, hence generating much smaller regression functions.

### 3.3 10 Input XOR Problem

Because the proposed algorithm constructs approximations 2 dimensions at a time, one may be lead to conclude that it is not capable of solving XOR problems of 3 or more inputs, however, this is not the case. The proposed algorithm was able to solve the 10 input XOR problem in 10 minutes of learning time (the size of the resulting regression function

was 40 K-bytes). The theoretical basis for this surprising fact is the subject of ongoing theoretical study, however, we postulate that it is the direct result of our use of bootstrap samples during learning.

## 4 Conclusion

We have demonstrated that nonparametric learning is practical for very high dimensional learning problems. We did this by proposing a simple nonparametric learning methodology, which neither requires computationally expensive variable selection, nor an expensive search procedure to find the best model representation. An algorithm based on this methodology was successfully applied to several high dimensional learning problems, which no other currently published algorithm could effectively address. The size of the regression functions produced by the proposed algorithm depended on the learning data's complexity, and not on its dimension. Our current ongoing efforts are being directed to a more complete theoretical analysis of nonparametric algorithms having these properties, as well as the application of our methodology to real world agents. In addition, we are interested in exploring the efficacy of the proposed methodology as an analytical tool for exploring the relative importance of input variables.

## References

- [Breiman *et al.*, 1984] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth International Group, Belmont, California, 1984.
- [Breiman, 1996] L. Breiman. Bagging predictors. *Machine Learning*, 24:123, 1996.
- [Friedman and Stuetzle, 1981] J. Friedman and W. Stuetzle. Projection pursuit regression. *J. Amer. Statist. Assoc.*, 76(376):1–141, December 1981.
- [Friedman, 1991] J. H. Friedman. Multivariate adaptive regression splines. *Ann. Statist.*, 19:1–141, 1991.
- [Friedman, 1994] J. H. Friedman. An overview of predictive learning and function approximation. In V. Cherkassky, J. H. Friedman, and H. Wechsler, editors, *From Statistics to Neural Networks*, pages 1–61. Springer-Verlag, 1994.
- [Jordan and Jacobs, 1994] M. I. Jordan and R. A. Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural Computation*, 6:181–214, 1994.
- [Michie *et al.*, 1994] D. Michie, D. J. Siegelhalter, and C. C. Taylor. *Machine Learning, Neural and Statistical Classification*. Ellis Horwood, New York, NY, 1994.
- [Quinlan, 1993] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.
- [Rasmussen, 1996] C. A. Rasmussen. A practical monte carlo implementation of Bayesian learning. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *NIPS 8*. MIT Press, Cambridge MA, 1996.