Recurrent Neural Networks for Remaining Useful Life Estimation

Felix O. Heimes, BAE Systems

Abstract— **This paper presents an approach and solution to the IEEE 2008 Prognostics and Health Management conference challenge problem. The solution utilizes an advanced recurrent neural network architecture to estimate the remaining useful life of the system. The recurrent neural network is trained with back-propagation through time gradient calculations, an Extended Kalman Filter training method, and evolutionary algorithms to generate an accurate and compact algorithm. This solution placed second overall in the competition with a very small margin between the first and second place finishers.**

Index Terms—**Machine Learning, Prognostics, Recurrent Neural Networks, Remaining Useful Life**

I. INTRODUCTION

THIS paper describes the development of a data-driven algorithm to predict remaining useful life (RUL) of a complex system as it degrades from an unknown initial state to failure. Over two hundred multivariate time series data sets were provided as part of the IEEE 2008 PHM Conference challenge problem. Each data set is representative of a unique unit of the complex system that has unknown arbitrary initial wear and manufacturing variation. The data sets include three operational settings and twenty one sensor measurements that are contaminated with noise. "Training" and "test" sets are provided. The training sets contain examples of units that run until failure while the test sets end some time prior to failure. The test sets are used to evaluate the accuracy of solutions submitted.

The solution presented here was developed completely using proprietary machine learning software. No attempt was made to analyze the data to understand or extract underlying features that may have been present in the data. The primary algorithm used to model the system was a custom recurrent neural network architecture that was able to model the estimated remaining useful life of the system while at the same time filtering the data to minimize noise.

II. DATA OVERVIEW

The training set included operational data from 218 different units. In each data set, the unknown system was run for a variable number of cycles until failure. The lengths of the runs varied, with the minimum run length of 127 cycles and the maximum length of 356 cycles. The table below summarizes statistics about the length of each run.

TABLE 1 – LENGTH OF EACH RUN

Statistic	Run Length
Minimum	127
Maximum	356
Average	209
Standard Deviation	43.5

The plots in Figure 1 show two of the sensor measurements from the first sequence in the training data. The first sequence is 213 samples long. The top plot shows the first two signals as a function of sample index and the bottom plot shows signal 2 on the vertical axis and signal 1 on the horizontal axis. The bottom plot shows that the signals are clustered around six operating points and the variation around each operating point is very small compared to the magnitude of the signals. The top plot shows that the signals jump from operating point to operating point during each run. Given this characteristic in the data it is not possible to easily observe any trends in the data that might correlate to the degradation in the system. This attribute makes machine learning approaches excellent candidates for this problem, since the underlying physics would be difficult to understand directly.

III. CLASSIFIER TRAINING

An initial investigation was undertaken to determine the relative difficulty in detecting and modeling degradation in the system using the available machine learning tools. More specifically, could a Multi-Layer Perceptron (MLP) Neural Network accurately learn to classify the difference between a good system and a bad system? How large would the MLP structure need to be to accomplish the classification task? For what duration would the training algorithm need to execute before settling on a good solution?

The answers to these questions would provide insight into how difficult the problem would be to solve. If the problem appeared to be relatively difficult to solve, then more advanced data pre-processing techniques may need to be employed.

As a means to answer this question it was assumed that the first 30 samples in each sequence represent a healthy unit and the last 30 samples in each sequence represent a degraded unit. The data was then divided into two data sets: one representing a healthy unit and one representing a degraded unit.

Manuscript received July 21, 2008.

Felix O. Heimes is with BAE Systems, Electronics and Integrated Solutions, Johnson City, NY 13790 USA (e-mail: felix.o.heimes@baesystems.com).

9778-1-4244-1936-4/08/$25.00 © 2008 IEEE

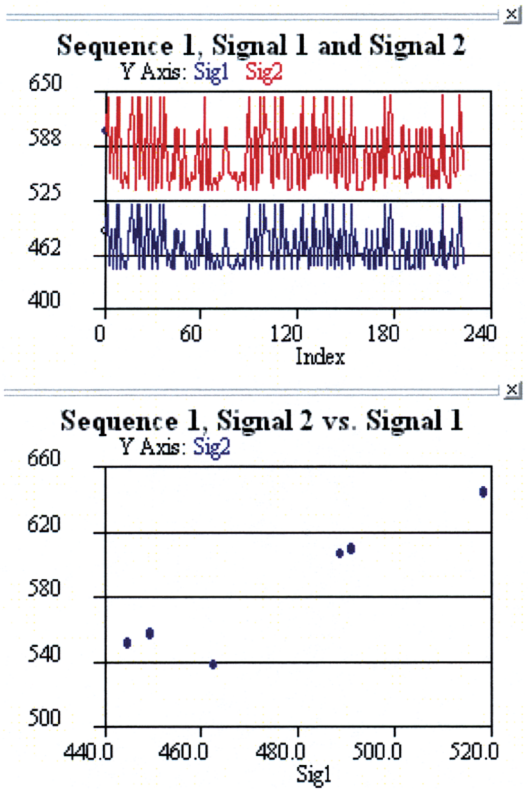

Fig. 1. – Typical input features for one data sequence. Signal 1 and Signal 2 versus sample index and Signal 1 versus Signal 2.

Proprietary Multi-Layer Perceptron Neural Network training software was used to train a classifier to distinguish between degraded and healthy units. This software tool utilizes an Extended Kalman Filter Training method that has been shown to be capable of learning complex models. The EKF training method was first introduced by Singhal and Wu [1] and the implementation used is for this effort is based on the work by Feldkamp and Puskorius [2].

Classifier training demonstrated that this classification task was easily solved by an MLP network. Experimentation showed that a three-layer network learned faster, created a more compact model, and was more accurate than a two-layer model.

A three layer network using all 24 inputs with three nodes in the first hidden layer and two nodes in the second hidden layer was able to achieve 99.1% classification accuracy after 5 seconds of training on an Intel 2-gigahertz processor running Windows XP. The training was performed on the entire training set of 218 sequences.

This experiment demonstrated that the raw inputs available in the data set provide enough information to indicate the health of the system and that an EKF trained MLP network is very capable of distinguishing between healthy and degraded units from the available data.

IV. APPROACH FOR RUL LEARNING

The classifier results were useful and interesting; however a classifier does not provide the required estimates of remaining useful life. For RUL estimation, a continuous variable representing the number of cycles remaining until failure was needed. An artificial signal representing RUL was created to train a function estimator neural network. This artificial signal was simply set to the number of cycles that remain before the end of the sequence. Considering the fact that the end of the run represents the last cycle before failure, the number of samples left before the end of the sequence should be an accurate representation of RUL.

V. MLP FUNCTION ESTIMATOR

The next step was to train an MLP function estimator to model the number of cycles remaining before the end of the run as a function of the 24 inputs. As before an MLP with two small hidden layers was trained using the EKF training algorithm and produced fairly good accuracy.

The plots in Figure 2 show the output of the MLP (in red) versus the target output (in blue) for three different sequences. The sequences shown in Figure 2 are sequence 1, sequence 2, and sequence 147. These sequences were selected since they represent an average length run, a short run, and a long run. (Note: The plots in figure 2 all show a vertical range of 0 to 250 for consistency. Sequence 147 appears to be limited to 250 in the plot, however this is just a result of limiting the plot range and the target output actually changes linearly from 320 to 0.)

There a number of interesting insights that can be derived by examining the MLP output shown in Figure 2. First we can see that the MLP network is able to approximately learn the number of remaining cycles. In all cases, the output curve starts our relatively flat and then degrades toward zero near the end of the run. These initial results were very encouraging and showed that a reasonable estimate of RUL could be derived.

Each plot seems to show a consistent characteristic. For the first third of the sequence, the MLP output is fairly flat and constant. In the last third of the sequence, the MLP output decreases almost linearly towards zero. Somewhere in the middle third of each plot, there is a knee in the curve as it transitions from a roughly constant value to a linearly decreasing curve. The knee in the curve shows when the failure first develops.

There is also a lot of noise in the MLP output that is attributed to sensor noise and needs to be reduced for better accuracy.

The slope of the curves in the second half of each sequence, as learned by the neural network, varies from run to run. The slope of the training target output is -1. The slope for sequence 1 is approximately equal to the target slope. The slope of sequence 2 is much steeper than the target output. Finally, the slope of sequence 147 is much more gradual than the target sequence.

A final observation on the data shows that the initial output for each sequence changes slightly from run to run. The MLP seems to have picked up on the initial manufacturing tolerance and initial wear in the system.

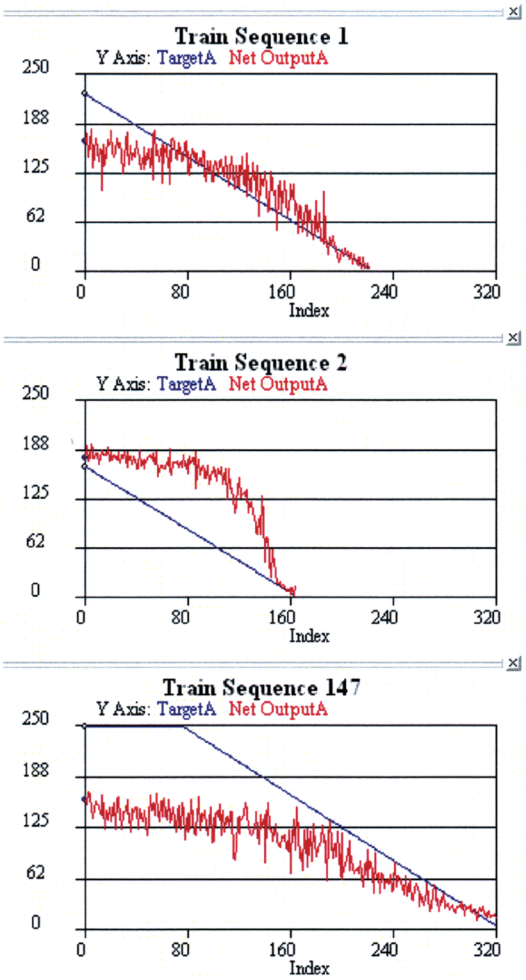

Fig. 2. Examples of MLP neural network output versus target output.

Observation on the characteristics of the trained network output motivated a change to the target output of the system such that the maximum target output is limited to a constant value for all sequences. Since the degradation in a system will generally not be noticeable until after the unit has been operated for some period of time and the initial failure had developed, it is probably unreasonable to estimate RUL until the system begins to degrade. How could one expect to predict how long a particular item will last when it is brand new and in a very healthy state? It is only after the unit has been operated for a time that degradation will show up and provide a relative indication of RUL. For this reason it seems reasonable to estimate the RUL when the system is new as a constant value.

The following reasoning was used to select the constant initial RUL value:

- The minimum run length is 127. There are no examples in which the run length is less than 127.
- The average run length is 209. On average the knee in the curve should occur around sample number 105.
- The MLP results shown previously produces an average steady state output near the beginning of each run of between 140 and 180.
- Negative RUL estimation errors are penalized less than positive errors.

Based on these observations, an initial RUL of around 120 to 130 cycles seemed reasonable. Some experimentation was done by training models whereby the maximum RUL was varied and the results were submitted to the website to evaluate which limit worked best. The final limit that was used for training was 130.

The plots in Figure 3 show the result of training an MLP neural network using a RUL target output that is limited to 130. A few observations on the training data:

- The outputs seem a little less noisy. Perhaps since we have better represented the true output, the network can better model it.
- The MLP steady state initial output is now about 120. This is much less than the prior initial steady state output as one would expect since the initial values of the training output are now much smaller.
- The knees in the curve have not changed by much and the slope near the end of the curves appears to be about the same as before.

These results show that near the end of the run when the system is close to failure the MLP network is fairly accurate at predicting the RUL. During the early and middle portions of the run the task of predicting RUL is much more difficult and the performance is not as good.

There are a couple of issues as to why an MLP neural network is not ideally suited to solve this problem:

- The data set is a temporal sequence and there is time dependence in the input/output mapping. The MLP network is a static mapping that only takes into account the current state of the system.
- The input data is known to be corrupted by noise. For any given sample there may be a large amount of noise in the input data that translates to significant noise in the output.

Incorporating time domain filtering in the modeling process is an obvious next step to an improved solution.

VI. FILTERING APPROACH (RECURRENT NEURAL NETWORK)

Filtering the data seems to be necessary to produce a reasonably accurate RUL estimate due to the large amount of noise and the temporal nature of the data. Some form of linear or non-linear time domain filter could be applied to the output of MLP network. However, the analytical derivation of such a filter would have to be done by trial and error, since the underlying physics of the system were not known to guide the filter design process.

A better approach would be a neural network solution that incorporates filtering within the model. The neural network training algorithm could then automatically derive a near optimum filter to match the desired output signal.

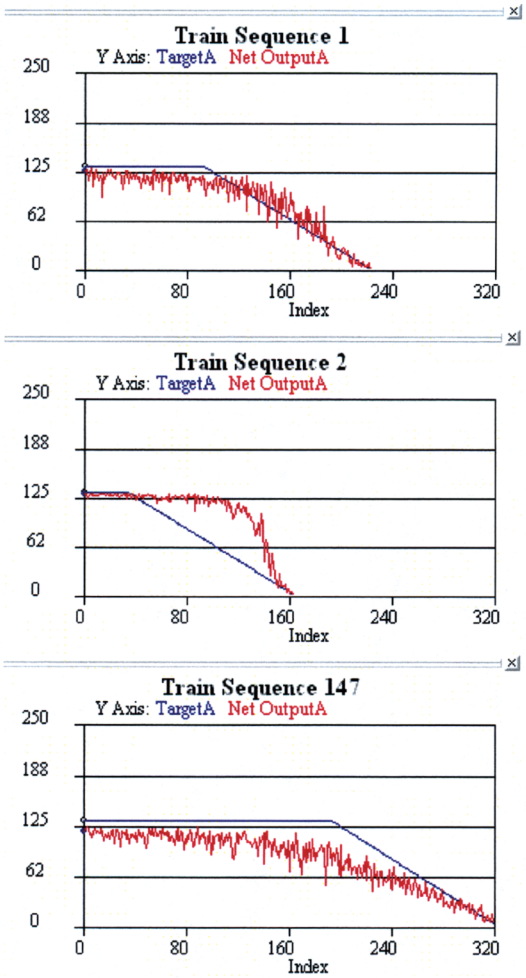

Fig. 3. Examples of MLP neural network output versus target output with target output limited to 130.

The most straight forward approach that is often used with neural networks when time domain modeling or filtering is needed is to present the neural network with multiple time delayed examples of each input parameter (a tapped delay line). This approach is straightforward and easy to implement since a standard static neural network and static derivative computations for learning can be used, however it has many problems. First of all it is difficult to know how many examples of each input are needed and how far back to go in time. Only a finite number of past states can be considered. Many trial and error iterations are needed to optimize the selection of the appropriate inputs.

Another major problem with this approach is that the network now has 2 or 3 (or more) times more inputs. The network now becomes much larger and has many more weights requiring more computational demand. An even worse problem is that the network is more likely to overfit the training data since it has many more adjustable weights. To overcome the overfitting problem, a more complex modeling approach might be needed such as Support Vector Machines (SVM) to ensure that the network will generalize well. However, SVMs are known to create extremely large models with possibly 10 to 100 times greater computational complexity than a neural network approach.

A better approach to handling non-linear time domain dynamic system modeling is to use a recurrent neural network. An RNN network utilizes internal memory and feedback and can learn complex non-linear dynamic mappings. A recurrent neural network can account for an arbitrary number of previous input and hidden states and can learn strongly hidden states [3].

Advanced proprietary recurrent neural network architectures and training algorithms specifically designed for modeling dynamic systems was utilized. These software tools were developed for problems such as modeling aircraft flight dynamics, turbine engine dynamics and other complex non-linear dynamic systems. It utilizes some innovative features within the network and has several adjustable internal features that can be adapted to best model a particular system.

One of the difficulties in developing a recurrent neural network training algorithm is in computing the gradients of the network weights. The gradients must be computed not only as a function of the output error, but they must also be computed backward in time to account to temporal variation in the data set. The training algorithm for our RNN utilizes the Truncated Back Propagation Through Time (BPTT) approach to compute the gradients.

As with the MLP network, the RNN training algorithm utilizes the Extended Kalman Filter training method to update the weights of the network. The EKF is particularly attractive for recurrent networks since it minimizes the number of training iterations and does not require all the training data to be utilized for each training iteration. Our RNN, BPTT, and EKF implementation is based on the many papers on this topic by Feldkamp and Puskorious as summarized in [4].

A final feature of the RNN training software is that it incorporates evolutionary algorithms to automate the normal iterative process that is performed optimizing a neural network model. This greatly reduces the amount of human effort that is needed and produces very accurate models. The evolutionary algorithm maintains a population of multiple solutions and adapts the population to improve the solutions. Our implementation for the RNN adapts the structure of the RNN, the number of hidden nodes in each layer, and a few other variables of the network. It also adapts two constants of the EKF training algorithm (R and Q).

The evolutionary approach utilizes a training data set and an independent validation data set. It trains neural networks using the training set and then evaluates the model on the validation data set to rank the results.

The training algorithm will attempt to fit the training data as accurately as possible for a given network architecture. A very tight fit to the training data may not be the best solution to the problem, however. A common problem with neural networks is that the model will "overfit" the training set and not generalize well to different data if there are extra input parameters or the network is too large. The DE therefore does not consider the training accuracy and instead ranks networks based on the validation data set only. Networks that produce higher accuracy on data independent of the training data will generally be better models and less likely to have overfit the

training set. As such the evolutionary process adapts a solution that models the underlying physical system with high accuracy and prevents overfitting the data.

VII. Differential Evolution

The evolutionary process that we employ is Differential Evolution (DE) [5]. DE has been shown to be an efficient method of optimizing real-valued multi-model functions and is conceptually simple and easy to use. DE is a parallel directed search method where the solution is coded in a vector of real-valued parameters. DE utilizes a population of potential solution vectors that are initialized with random values that cover the entire solution space. The population is evolved by evaluating the fitness of each individual in a new population to the existing population and replacing inferior individuals with better ones.

Each individual (parent) is modified from three individuals selected randomly out of the total population. The weighted difference (mutation factor) between two of the individuals is added to the third individual. Crossover is then performed by replacing a random number of elements in the parent with elements from the result of mutation. The probability of an element in the parent being replaced is controlled by a constant factor (crossover).

The DE algorithm used in this work also implements selection, whereby individuals used for the mutation and crossover are selected from the fitter elements in the population instead of from the entire population.

One attractive feature of DE is that modifies its search effort automatically based on the population. Initially the population covers the entire search space and the differences between vectors will be large. Therefore early in the evolution process the algorithm will make larger step changes in the search process and aggressively search the entire space. As the population converges the difference vectors become smaller and the algorithm automatically utilizes smaller step changes to converge on minima in the search space.

With the DE process, the user initializes the search parameters and then lets the training process run until an acceptable accuracy threshold is reached. The training processes will train hundreds of networks and converge towards a good solution. This eliminates the normal trial-and-error experimentation process that an engineer goes through to find a solution and reduces the time and cost to model systems.

VIII. Recurrent Neural Network Output

The output of a recurrent neural network trained on this data set is shown in Figure 4. The recurrent neural network matched the desired training data much more accurately then the static MLP. The output of the network is much less noisy then the MLP network, since the recurrent neural network is able to provide time domain filtering that the MLP is unable to provide.

It is also interesting to note that the recurrent neural network output is much more linear and consistent near the end of each sequence. For each sequence the recurrent neural network output reduces linearly as it approaches zero and for each sequence the rate of decay is consistent.

The recurrent neural network is able to detect when the system is beginning to fail and by providing feedback of past state information is able to learn a consistent response towards failure. Exactly how the network is able to do this is of-course unknown, but that is the benefit of the machine learning approach. The human developer does not need to understand all the underlying physics of the problem and can therefore develop a solution much quicker.

IX. Complexity of the Recurrent Neural Network

The relative complexity of this recurrent neural network is fairly small. The network utilizes all 24 inputs, has three layers of feed-forward connections, and recurrent connections. Several recurrent networks were trained over the course of the challenge problem and the number of weights typically ranged from 150 weights to 170 weights. In comparison, the MLP network with three layers of connections had 91 weights. The recurrent neural network represents an increase in complexity of the model by less than a factor of two, but the accuracy of the model significantly improved.

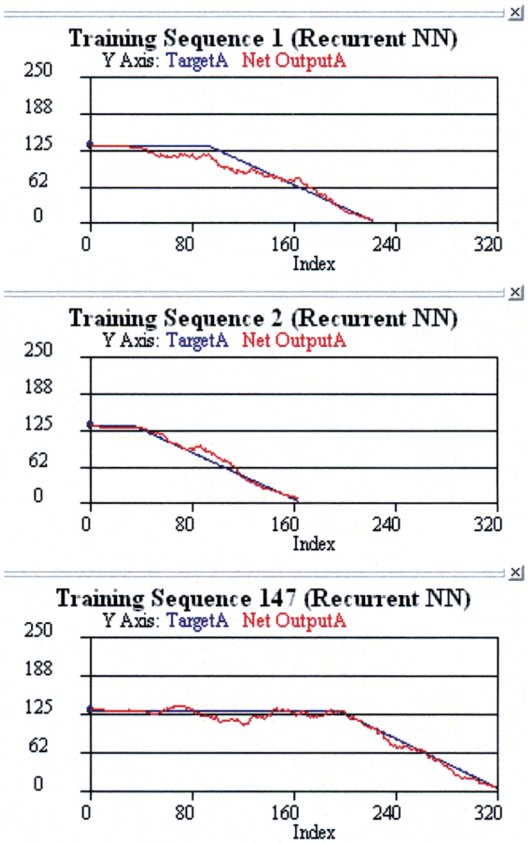

Fig. 4. Examples of recurrent neural network output versus target output.

X. COMBINATION OF MULTIPLE MODELS

Over the course of the competition several models were trained and submitted to the website. The three best models that were submitted achieved the following errors on the test set:

Model #	Error
Model 1	579
Model 2	566
Model 3	561

Figure 5 shows the output of each of the three models after sorting the outputs according to model 1. The vertical axis is the final estimate of remaining useful life for each sequence in the training data set. The horizontal axis is the sequence number after sorting (there are approximately 220 sequences in the training set). This plot shows that the three models while all providing a fairly accurate estimate of the test data set, have significant differences in certain areas. When the number of remaining cycles is small (less than 55) the models all correlate very well. The differences between the models are larger when the number of cycle to failure is greater. This is probably due to the fact that the target output is somewhat arbitrary early in the sequence and more accurate at the end.

Since three good models were available and they all had fairly different responses for some of the training and test examples a composite model was used for the final submission to the Challenge problem. The output of the three models was simply averaged and the average error was used as the final output. Averaging the three neural network outputs on the test set produced an accuracy of 519.8 and represented an improvement of 7% over any one individual model.

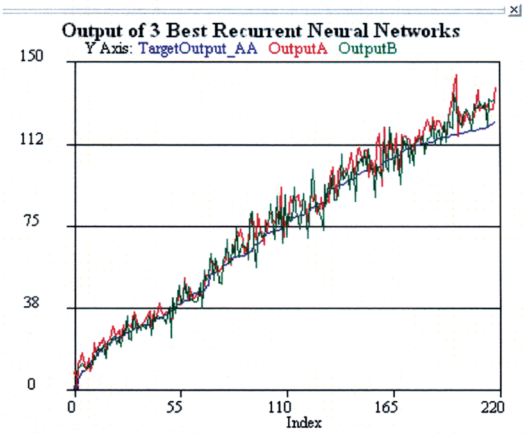

Fig. 5. Output of 3 best recurrent neural networks with data sorted from low to high for the first output.

REFERENCES

[1] S. Singhal and L. Wu, "Training multilayer perceptrons with the extended Kalman algorithm," in *Advances in Neural Information Processing Systems* 1, Denver 1988, D. S. Toretzky, Ed. San Mateo, CA: Morgan Kaufmann, 1989, pp. 130-140.

[2] G. V. Puskorius and L. A. feldkamp, "Decoupled extended Kalman filter training of feedforward layered networks," in *International Joint Conference on Neural Networks*, Seattle 1991, vol. I, pp. 771-777.

[3] R. J. Williams, "Adaptive state representation and estimation using recurrent connectionist networks," in *Neural Networks for Control*, W. T. Miller III, R. S. Sutton, and P. J. Werbos, Eds. Cambridge, MA: MIT Press, 1990 char 4. pp. 97-114.

[4] L. A. Feldkamp and G. V. Puskorius, "A Signal Processing Framework Based on Dynamic Neural Networks with Application to Problems in Adaptation, Filtering, and Classification," in *Proceedings of the IEEE*, vol. 86, No 11, November 1998.

[5] R. Storn, "On the Usage of Differential Evolution for Function Optimization", in *Biennial Conference of the North American Fuzzy Information Processing Society, 1996,* pp. 519 – 523.