A Priori Adaptation of a State Machine Sensor Network

Mike Murphy
CIS 731

8 May 2006

Abstract

A design and workstation implementation of a neural-network state transition apparatus for a state-machine sensor network is discussed, with an emphasis on the results of training and testing the backpropagation neural network. With a very large number of iterations, the neural network apparatus delivers good performance on both training and test data, provided that the behavior of the sensors for which the network is to be trained is known in advance. Lacking knowledge of the problem domain and the expected sensor behavior, the training and testing performance of the neural network is unsatisfactory. Furthermore, training the neural network to recognize changes resulting from Boolean combinations of several sensors is best accomplished by implementing additional neural networks for each sensor. These limitations raise concerns about the suitability of a neural network, as opposed to a system based on production rules, for solving the state machine generalization problem.

1 Introduction

Murphy and Post [2006] describes an environmental sensor network that employs a finite state machine algorithm on each component Berkeley mote, in order to determine if changes in sensor inputs are sufficiently significant to indicate an actual change in the surrounding environment. Discriminating between actual environmental changes and small spurious deviations (noise) in the sensor readings is crucial because these sensor networks generally rely on a small, finite supply of battery power. Sensor network nodes remain in a low-power "sleep" state unless an environmental event necessitates processing, transmitting, or logging observed data. Differentiating between actual environmental events and spurious non-events allows the nodes to remain asleep for a greater proportion of time, thereby conserving power.

A key issue with the implementation of this sensor network model is that the component that processes sensor inputs and decides if the environmental state has changed must be modified substantially for each new application. The sensor-specific encoding process must be changed whenever a new sensor is introduced, in order to adapt the existing state change algorithm to the new sensor's calibration curve. Thus, the existing Boolean transition function Ψ (formula 1) is not optimal for all applications, as it is constrained to operate only on the mean and standard deviation of the historical samples (μ_h and σ_h) and the mean of the latest sensor readings (μ).

$$\Psi(\mu_h, \sigma_h, \mu) \equiv (|\mu - \mu_h| \geq \alpha\, \sigma_h) \tag{1}$$

To complicate matters further, the parameters that define what constitutes an "interesting" environmental change are different for each new sensing application. In many cases, certain environmental changes are insignificant in the context of the domain being monitored. For the ephemeral stream detection application described in Murphy and Post [2006], an interesting change in soil moisture was one that was substantial enough to indicate the presence of rainfall runoff as opposed to condensation or drizzle. Other applications, however, will rely on other parameters to determine if the environment has in fact changed from the perspective of the application. For instance, a turbidity application that is being tested at Clemson University detects changes in the amount of suspended particulate matter in streams that eventually feed municipal water reservoirs. Certain critical turbidity values matter more than others when it comes to filtering drinking water [U.S. Environmental Protection Agency, 2001].

A major improvement to the current development paradigm for this sensor network model would be a faster method of adapting the sensor network state transition apparatus to new applications. Such a mechanism for rapid software adjustment would permit multiple new applications to be developed and tested within a short time frame. A neural network state transition algorithm was developed and tested as a candidate solution to this software adaptation problem.

A modification to the original state change algorithm from Murphy and Post [2006] (figure 1) was designed around a neural network with inputs for the current sensor readings and for saved state information (figure 2). This paper describes the design of the neural network state transition apparatus and its simulated performance in training and testing on the workstation. Limitations of this algorithm are discussed, along with an alternative algorithm, based on production rules, which might be a better solution candidate to the problem of rapid application adaptation.

2 Procedure

Although it would be theoretically ideal to be able to perform in-situ training of an adaptive neural calibration algorithm on an actual sensor network implemented in hardware, there are several practical issues with doing so. One of the major difficulties would be the substantial battery drain resulting from training or re-training the network. Another critical problem would be the code complexity and size of adding the learning algorithm code to the sensor network application: the program described in Murphy and Post [2006] is already starting to approach the code size limit of the mote's microcontroller flash memory.

To mitigate these issues, an *a priori* neural network adaptation algorithm was developed, making use of the widely understood backpropagation supervised training algorithm. This neural network was designed to take the latest sensor readings, coupled with saved state information for each sensor, as its input. A single output was employed to drive a step function to choose between the Boolean condition of no state change (0) and the condition of a state change (1). It is important to note that the time at which state information was last saved is unknown and nondeterministic, which precludes the use of a traditional recurrent network.

Figure 1: Modifications to the Original State Transition Apparatus

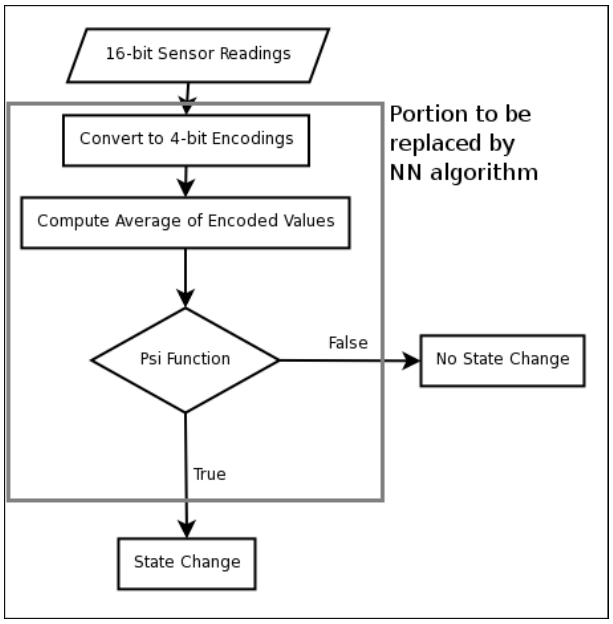
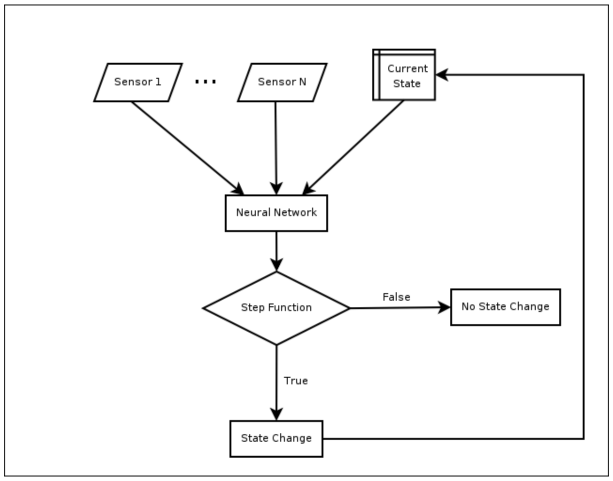

16-bit Sensor Readings
Convert to 4-bit Encodings
Portion to be replaced by NN algorithm
Compute Average of Encoded Values
Psi Function
False
No State Change
True
State Change

Figure 2: New State Transition Apparatus



2.1 Data Production Requirements

Training and test data sets for the neural network were produced by means of random function generation, utilizing turbidity sensor calibration data provided by Prof. Chris Post of the Clemson University Department of Forestry and Natural Resources [Post, 2006]. Due to uncertainties in the turbidity calibration data, it was determined that voltage values would be used for the training and testing, instead of turbidity units. The data set from Prof. Post suggested that the turbidity sensors experienced some random noise up to about 0.06 volts deviation from the desired values, with a measured output range from 0.68-3.06 volts.

A random curve generator was constructed using the C programming language, which would operate by selecting a random starting output value at time zero and then randomly moving to another value according to specified parameters. The chief feature of the curve generator, which allowed for more realistic emulation of environmental conditions, was the concept of "stasis," or a lack of change in the environment from time t to time $t+1$. Stasis was implemented by means of a probabilistic condition: if a randomly generated value exceeded the stasis threshold, then the function output value could change. However, if the stasis threshold was not reached, the output of the random function generator at time $t+1$ would be equal to that at time t.

When the probabilities did dictate non-static behavior, the random curve output value was changed by adding a random displacement to the current output value. Thus $y(t+1) = y(t) + \Delta t$, if the stasis threshold was exceeded. Δt was initially randomly selected to be positive or negative, and then kept the same sign with decreasing probability over the curve's "run length." The run length was specified by means of two parameters: a minimal value in iterations of t for which the sign of Δt would not change, and a maximal value after which the sign of Δt would be negated. Over the range of the run length minimum and maximum, the probability that the sign of Δt would change increased. The speed with which Δt changed, and the curve changed direction, was adjusted by means of a "smoothness" factor that was designed to reduce the number of sharp peaks in the graph. With these parameters, it was possible to generate a curve with certain desired characteristics such as the length of time the graph would travel in one direction before changing sign, and whether or not the graph would be able to reach its maximum and minimum output values (at which a sign change in Δy was forced immediately, producing a sharp point).

Since a certain amount of low-level noise was observed in the turbidity calibration data, a noise function was added to the random curve generator. This noise introduction occurred *after* the above random stasis and displacement code produced a desired output $y(t)$. The noise generator first made a probabilistic decision either to add noise or not to add noise, comparing against a user-specified noise probability threshold. If noise was to be added, then uniform random noise in the range $[0, n]$ was added, producing the final curve $y(t) = c(t) + n(t)$, where $c(t) = y(t-1) + \Delta y$ if the stasis factor was exceeded, or $c(t) = y(t-1)$ otherwise.

2.2 Data Production Procedure

Seven random curves were generated by means of the curve generator, each of which shared the same common parameters for all settings except for the stasis factor (table 1). The stasis factor was intentionally varied from highly static situations (e. g. core straight lines) to

Table 1: Common Curve Parameters

t_{max}	5000
y_{min}	0.68
y_{max}	3.06
Δy_{min}	0.001
Δy_{max}	0.100
Run length minimum	10
Run length maximum	1000
Probability of noise	0.5
Amount of noise	≤ 0.06
Smoothing	0.9

Table 2: Curve Differences

Curve Number	Stasis Factor	Number of State Changes
1	0.99	0
2	0.90	5
3	0.80	21
4	0.70	61
5	0.60	96
6	0.50	174
7	0.10	686

highly variable curves that would be unrealistic in an environmental setting (table 2). It was hypothesized that the neural network should be trained on the most variable curve (curve 7) and tested on the less variable data sets, since the network weights would have to be well-adjusted to achieve good training performance on the wildly variable data. As a control to this hypothesis, a data set close to the middle of the stasis factors was also selected as a training set (curve 4).

In order to test the generalizability of this method to situations in which the sensor behavior was not known *a priori*, a wastewater treatment plant data set was obtained from a neural network data repository at the University of California Irvine [Poch et al., 1993]. Several input and output parameters were arbitrarily selected from this data set, and the included time series data were split into a training set consisting of 75% of the exemplars, and a test set containing the remaining points.

Several different normalizations were tested while the neural network processing and training code was debugged. The final selection was a normalization in the interval $[0, 1]$, which was accomplished via a linear transformation from each initial data space. For the random turbidity curve data, the minimum and maximum y values were used to describe the input range. In the case of the water treatment plant data, each element in the data space was normalized according to the maximum and minimum values of that particular element. Each normalization routine was implemented using the Python language.

2.3 Neural Network Implementation

Backpropagation code was adapted from code obtained from the Internet [Mohan, 1997]. Wide variations in training quality were observed, even with large numbers of iterations, during initial testing. It was determined that the initial random weight values had a large effect on the quality of the results. For this reason, a known suitable random seed was hard-coded into the algorithm.

If the current state of a sensor is plotted along the x-axis of a standard Euclidean plane, the saved state is plotted along the y-axis, and the state change threshold considers changes in both the increasing and decreasing directions, then the data space may be divided into 3 clusters. The state change class is divided over two non-adjacent clusters by the middle cluster, which corresponds to the non-changing class. Thus, from Mehrotra et al. [2000, p. 86], 2 hidden nodes are needed to separate the two classes. A 2-2-1 network was selected for this reason, for use with a single sensor. To test two combined sensors, a 4-4-1 topology was tried in hopes of avoiding a larger network.

Due to severe time constraints resulting from unexpected complexities in the neural network setup and the author's health issues, several proposed components of the project were not implemented. These included the nesC and TOSSIM frameworks (which were partially implemented but not completed), and the timer component, which was not directly relevant to the neural network. Additionally, a user-adjustable penalty function was proposed for biasing the network toward or away from making a state change. This bias utility was implemented, but it was subsequently removed during a debugging phase and was not re-implemented because it was deemed to be unnecessary.

3 Results

The random curve generator routine was run seven times for the seven different stasis factors specified. Figure 3 shows the most static of these curves, which is essentially a noisy straight line. In figure 4, the curve moves slowly from its initial position to a lower output value, yet still retaining a high degree of stasis. Figure 5 shows a more varied situation with lower stasis, which increases substantially in the fourth curve (figure 6). As stasis becomes less, the number and sharpness of peaks and valleys increases substantially in curves 5 and 6 (figures 7 and 8). Curve 7 (figure 9) shows extreme variability.

For the turbidity problem, the state change threshold was selected to be a deviation in either direction of 0.25 volts in the random curve output. In other words, the condition of a state change could be expressed by the Boolean predicate $|current - saved| > 0.25$. A summary of the number of state changes in each data set was provided in table 2.

The backpropagation training algorithm was run using set 7 as the training set, with sets 1-6 used for testing. Three separate experiments were conducted in which training was allowed to proceed for up to 25,000, 100,000, or 1,000,000 iterations. Training time varied from under ten minutes for the two smaller iteration limits, to well over an hour for the million-epoch training (table 3 and figure 10). In all cases, the training behavior started off smoothly for about the first 10,000 iterations (figure 11), then became unstable until about

Figure 3: Random Turbidity Curve 1

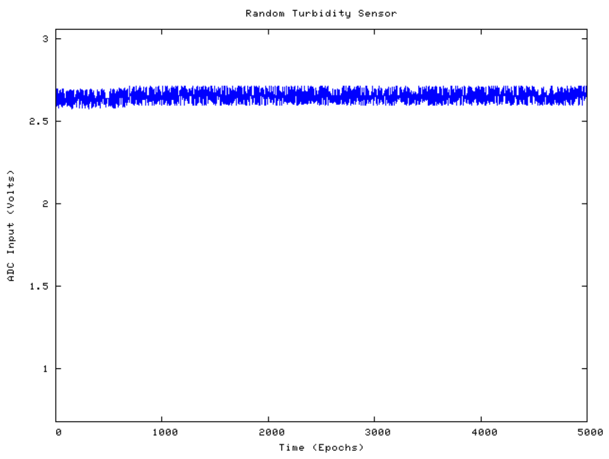


Figure 4: Random Turbidity Curve 2

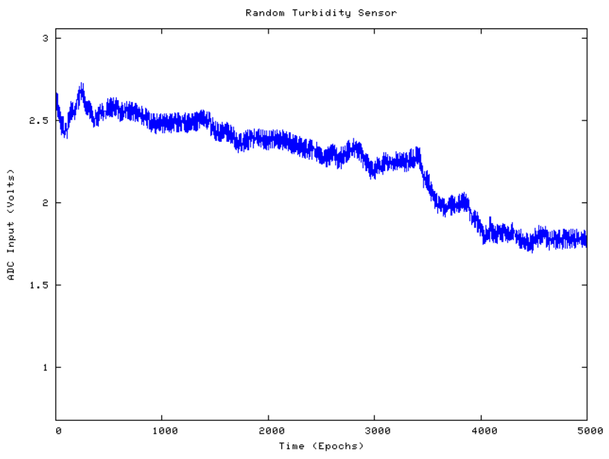


Figure 5: Random Turbidity Curve 3

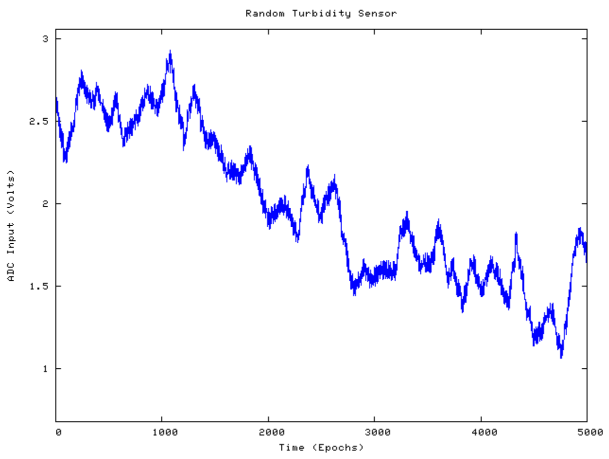


Figure 6: Random Turbidity Curve 4

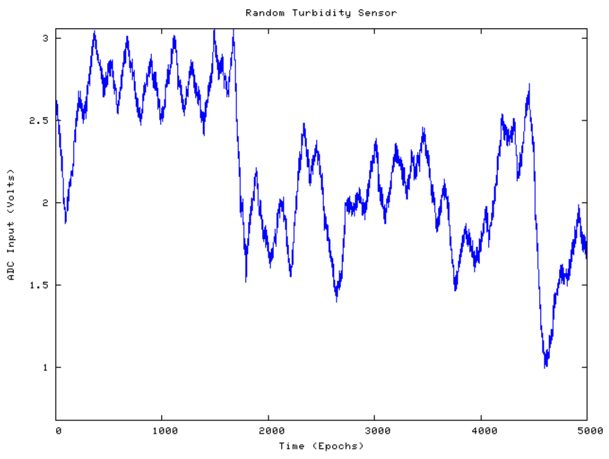


Figure 7: Random Turbidity Curve 5

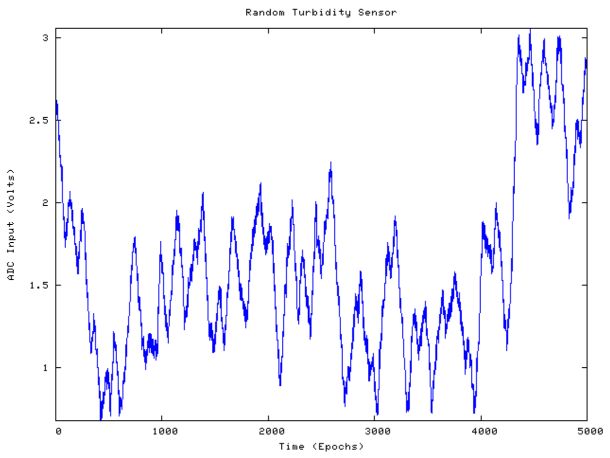


Figure 8: Random Turbidity Curve 6

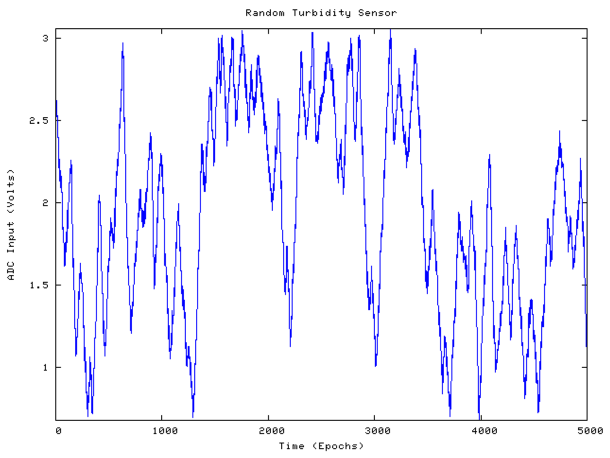


Figure 9: Random Turbidity Curve 7

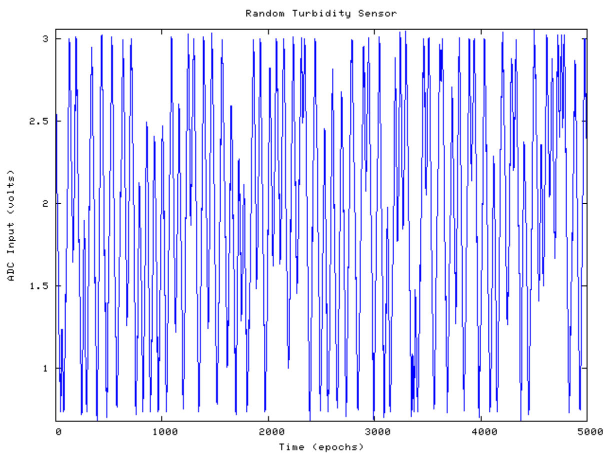


Table 3: Training Times (Curve 7)

Iterations	Time	Final MSE
25,000	00:02:09	0.07532
100,000	00:08:36	0.01239
1,000,000	01:26:47	0.00057

50,000 iterations (figure 12), then gradually converged toward lower error values until a steep decline between 400,000 and 500,000 iterations (figure 13). It should be noted that the hard-coded initial random seed and same training set for all three iteration limits means that the first two graphs show local subsets of the third.

After training, the number of misclassified points in the training set and test sets were calculated. These are summarized in figures 14, 15, and 16. Type I errors occur when the neural network indicates a state change that should not be made; these errors, as long as they occur in relatively small numbers, are not terribly concerning. Type II errors, however, are of concern because they indicate missed state changes and thus lost environmental data. Table 4 summarizes the number of missed state changes in each case, as a percentage of the total number of state changes in the data sets (see table 2). No state changes were missed with 1 million training epochs, and fewer changes were missed with 100,000 iterations than with 25,000.

As a control against the hypothesis that training was best conducted with the most variable data set, training was conducted with set 4, and the neural network was tested against all other sets. As shown in figure 17, the training error remained constant throughout the process (100,000 iterations were allowed). Figure 18 and table 5 show that Type II errors were high for all data sets except set 1.

For the data sets combined by the logical OR operator, training error (using a 4-4-1

Figure 10: Comparison of Training Times

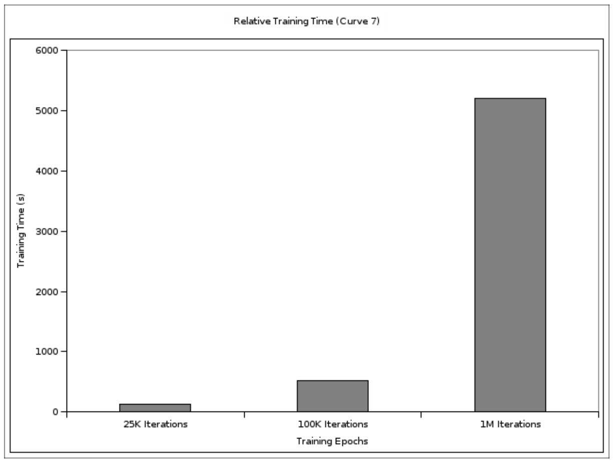


Figure 11: Training Error, Curve 7, 25K Iterations

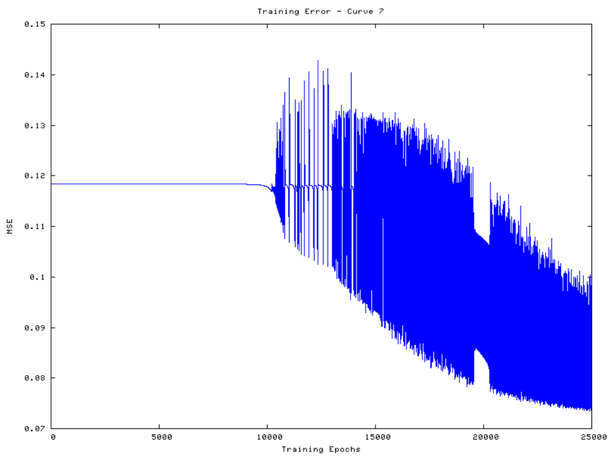


Table 4: Percentage of Missed Changes Training on Curve 7

Data Set	25K	100K	1M
Training (Set 7)	69.97%	7.87%	0.00%
Test Set 1	0.00%	0.00%	0.00%
Test Set 2	80.00%	60.00%	0.00%
Test Set 3	85.71%	9.52%	0.00%
Test Set 4	80.33%	60.39%	0.00%
Test Set 5	79.17%	18.75%	0.00%
Test Set 6	82.18%	12.64%	0.00%

Figure 12: Training Error, Curve 7, 100K Iterations

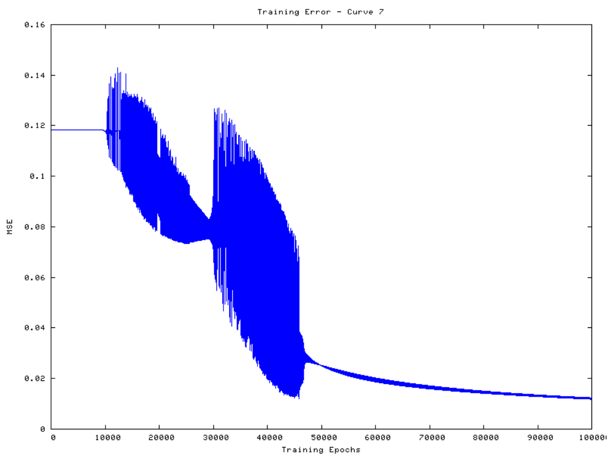


Figure 13: Training Error, Curve 7, 1M Iterations

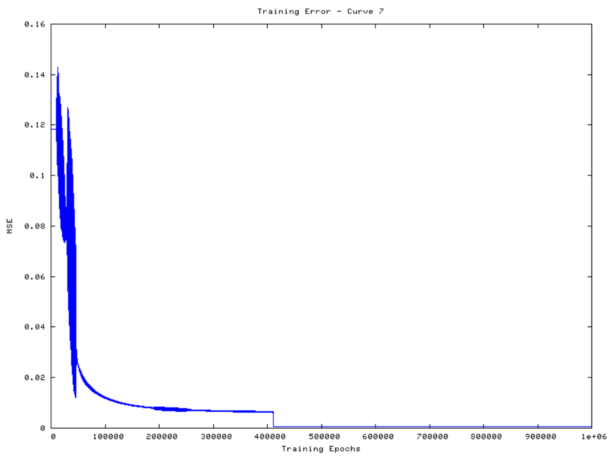


Table 5: Comparison Between Training on Curves 4 and 7, 100K Iterations

Data Set	7-results	4-results
Training (Set 7 / 4)	7.87%	100.00%
High-Stasis Test (1)	0.00%	0.00%
Test Set 2	60.00%	100.00%
Test Set 3	9.52%	100.00%
Test Set 4 / 5	16.39%	100.00%
Test Set 5 / 6	18.75%	100.00%
Test Set 6 / 7	12.64%	100.00%

Figure 14: Training and Testing Errors, Curve 7, 25K Iterations

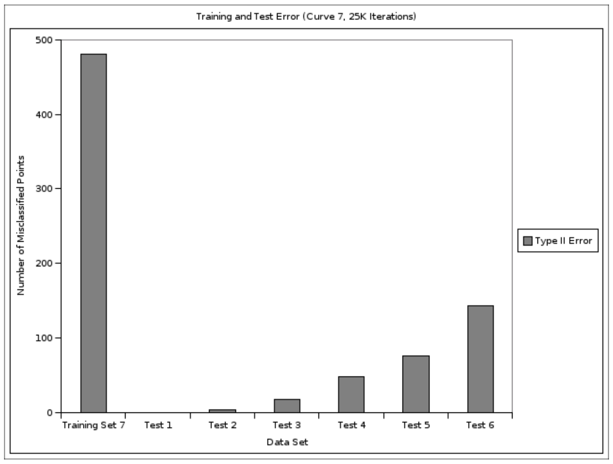


Figure 15: Training and Testing Errors, Curve 7, 100K Iterations

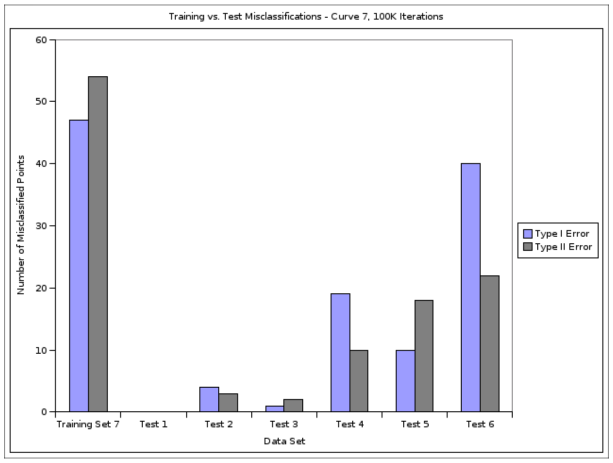


Figure 16: Training and Testing Errors, Curve 7, 1M Iterations

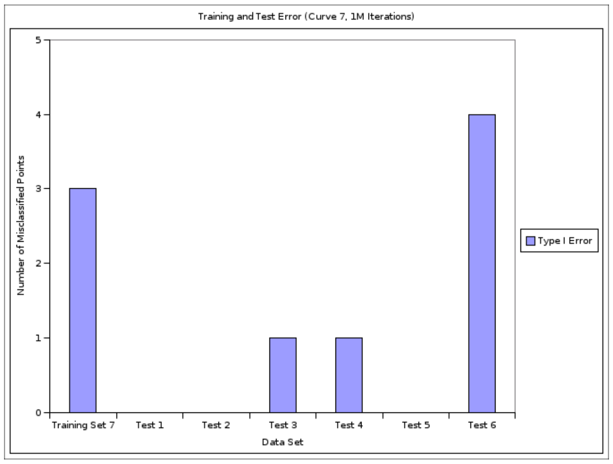


Figure 17: Training Error, Curve 4, 100K Iterations

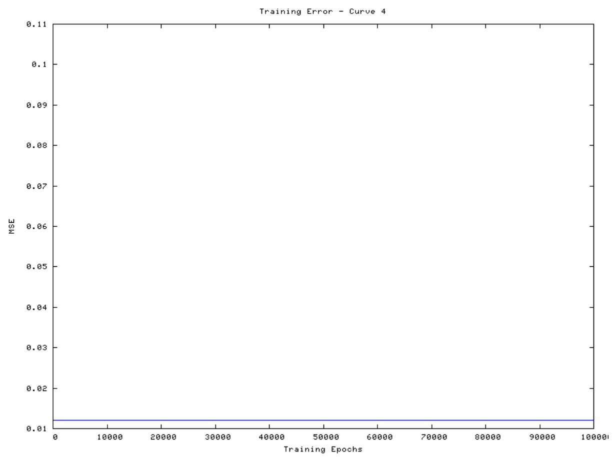


Figure 18: Training and Testing Errors - Curve 4

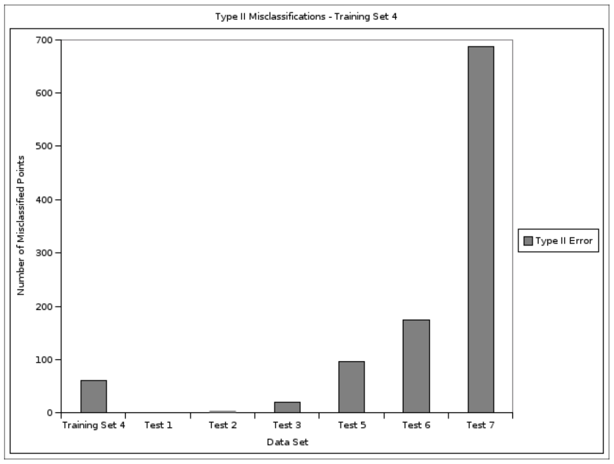


Table 6: Combined Data Set Results

Data Set	Missed State Changes	Final MSE
Training Set	61.02%	0.07951
Test Set 1+2	80.00%	0.00052
Test Set 4+5	75.00%	0.01763

network with a 100,000 iteration limit) indicated instability between about 40,000 and 60,000 epochs (figure 19). Type II error was quite high on all three sets (figure 20 and table 6).

The training behavior of the water treatment data is depicted in figure 21. For both the training set and test set, there were high levels of Type II error. Absolute Type II error was higher in the training set (figure 22), but relative Type II error was higher in the test set (table 7). A relatively high amount of Type I error was also noted in both sets, given the small size of each.

Table 7: Water Treatment Plant Results

Data Set	Missed State Changes	Final MSE
Training Set	76.52%	0.17795
Test Set	77.50%	0.25879

Figure 19: Training Error, Combined Curves 6-7

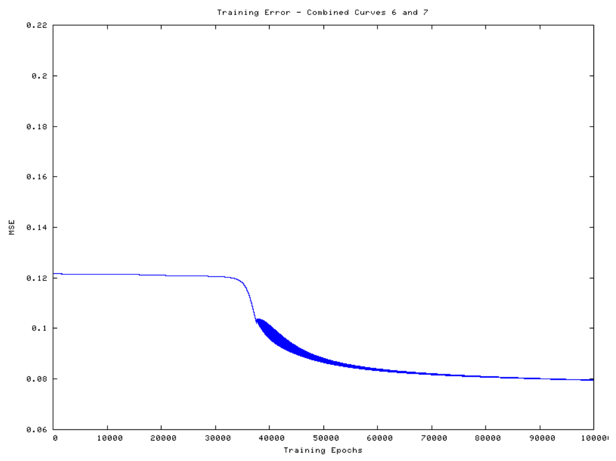


Figure 20: Training and Test Error on the Combined Data Sets

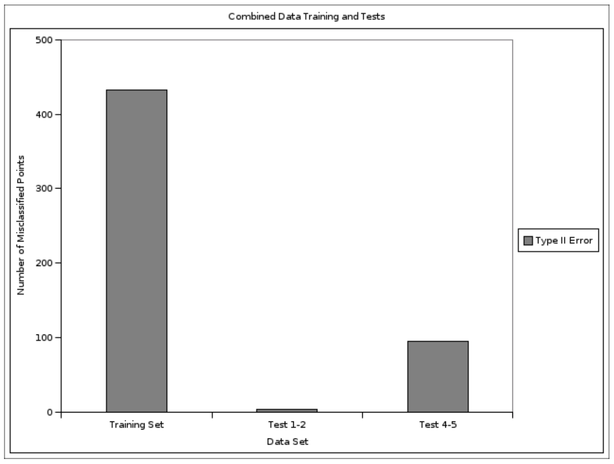


Figure 21: Training Error - Water Treatment Data

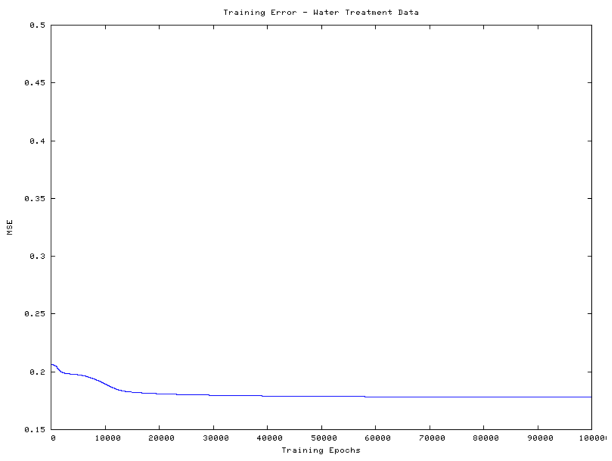


Figure 22: Training and Test Errors on the Water Treatment Data

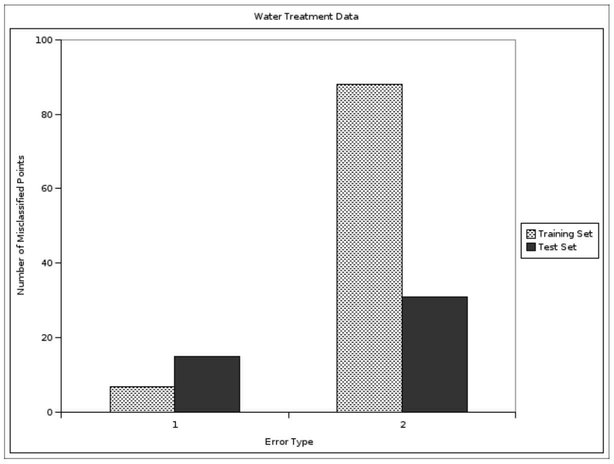


Figure 23: Multiply Instruction

```
load  [lhs], r0
load  [rhs], r1
mult  r0, r1, r2
store r2, [product]
```

Figure 24: Add Instruction

```
load  [lhs], r0
load  [rhs], r1
add   r0, r1, r2
store r2, [sum]
```

4 Architecture Issues

Two major issues arise when considering the operation of the neural network on a Berkeley mote class device. First, these devices generally use low-power RISC microcontrollers that lack floating-point arithmetic units. Second, if multiple sensors are to be used, it appears that a combination of individual neural networks with post-processing will be advantageous to a single larger network. Figures 23, 24, 25, and 26 provide simple generalizations of RISC load-store architecture instructions adapted from the Simple RISC Computer instruction set [Heuring and Jordan, 1997]. Let the function G give the number of RISC instructions needed to compute the Gaussian neural network node function.

A neural network with only one sensor and an absolute valued state change threshold has a problem space that contains three clusters: one where the state does not change, one where the new reading is substantially above the saved state, and one where the new reading is substantially below the saved state. If the logical OR of these two sensors is taken, and none of the threshold intervals overlap, there could be as many as 9 clusters in the resulting problem space. Thus, a 4-8-1 network may be needed, according to the formula in Mehrotra et al. [2000, p. 86].

Given the number of RISC instructions needed for each operation, and the G instructions needed for each Gaussian, a 4-8-1 network could require as many as $392 + 9G$ instructions to compute the output result and pass it through a final step function. Two 2-2-1 networks, each connected to a step function, with the step functions connected to a logical OR, would require only $144 + 6G$ instructions.

Considering that the initial sensor readings on the mote are encoded in integer variables, it would be possible to implement a state transition apparatus that used production rules as an alternative to the linked neural networks. Such a system would encode thresholds directly, using integer variables, and could operate using integer arithmetic. The behavior of a properly trained neural network and this type of production system should be identical.

Figure 25: Step Function

```
  load  [NN-result], r0
  brpl  r0, true
false:
  store 0, [step-result]
  br    end
true:
  store 1, [step-result]
end:
```

Figure 26: Logical OR Instruction

```
load  [lhs], r0
load  [rhs], r1
or    r0, r1, r2
store r2, [or-result]
```

5 Conclusions

The backpropagation network succeeded in learning the difference between state changes and environmental stasis, and the test results produced by this network were satisfactory. However, a large number of iterations were required to produce the desired results, and *a priori* domain expertise about the problem to be addressed by the sensor network was needed to achieve proper training of the neural network. In addition, training should be conducted on the most variable data set. Performance of a small network on combined data was unsatisfactory, as was the performance of the neural network on data where the underlying interrelationships were not known.

While the neural network state change apparatus is a solution to the original adaptation problem, it may not be the best candidate. Architecture concerns may favor a production rules system that can avoid floating-point arithmetic. However, there may be other aspects of the neural network system that remain to be explored, such as whether a neural network can learn certain combinations of production rules and thus result in fewer machine instructions than the other suggested candidate.

References

Vincent P. Heuring and Harry F. Jordan. *Computer Systems Design and Architecture.* Addison Wesley Longman, Inc., 1997.

Kishan Mehrotra, Chilukuri K. Mohan, and Sanjay Ranka. *Elements of Artificial Neural Networks.* MIT Press, 2000.

Chilukuri K. Mohan. Bp.c (sample code), 1997. URL `http://www.cis.syr.edu/ mohan/html/Bookfiles/bp.c`.

Michael A. Murphy and Christopher J. Post. A state machine sensor network for ephemeral stream detection. *To apppear in the International Journal of Distributed Sensor Networks*, 2006.

Manel Poch, Javier Bejar, and Ulises Cortes. Faults in a urban waste water treatment plant [sic]. Internet, Barcelona, Spain, June 1993. URL `ftp://ftp.ics.uci.edu/pub/machine-learning-databases/water-treatment/`.

Christopher J. Post. Turbidity sensor data. Personal Communication, 2006.

U.S. Environmental Protection Agency. Interim enhanced surface water treatment rule: A quick reference guide. Internet, May 2001. URL `http://www.epa.gov/OGWDW/mdbp/qrg_ieswtr.pdf`.