CS545: Assignment 8

An Application of Reinforcement Learning Techniques to Enable a Self-tuning Replication Server

Greg Carter

December 16, 2009

Contents

Contents ... 1
1. Introduction ... 1
2. RepServer Tuning Configurations ... 3
   2.1. SQT Cache Size ... 3
   2.2. Bulk Threshold ... 4
3. Monitoring RepServer Performance ... 5
4. Reinforcement Learning Mechanism Design ... 5
   4.1. Defining the Network ... 5
   4.2. Applying Reinforcement ... 7
   4.3. Choosing the Next Action ... 9
5. Extended Kalman Filter Training ... 9
6. Conclusion ... 11
References ... 12

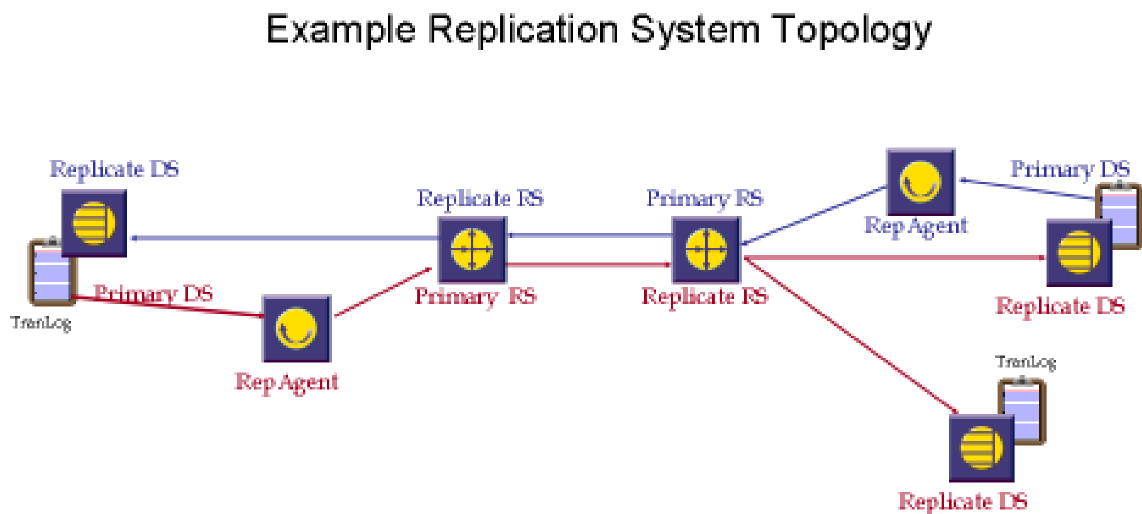
1. Introduction

Replication Server® is a software product manufactured and sold by Sybase, Inc. to address requirements in the information management world related to the movement of data between different geographical locations and between otherwise incompatible data management systems. Implementing a publish-and-subscribe model, RepServer runs as a multi-threaded, stand-alone process, exhibiting characteristics of both a server and a client with respect to the client-server design pattern.

As a server, RepServer accepts primary database operations in the form of Log Transfer Language (or LTL) from clients known as Replication Agents®. RepAgents are designed to monitor updates to a transaction log (typically, a relational database transaction log), select qualifying operations, package them in the form of LTL and hand them to RepServer. RepServer will validate the LTL and write it to a "stable queue", typically implemented upon a disk storage device. The database server whose transaction log is monitored by a RepAgent is known as a *primary data server*.

As a client, RepServer will read the LTL from the stable queue, determine subscribers for the operations identified in the LTL, reformat the operations into a form acceptable to the subscriber (typically the subscriber is another relational database server and so the LTL is reformatted into SQL) and hand the reformatted operations to the subscriber for execution. The subscriber is known as the *replicate data server*.

Figure 1: In this example replication system, there are two RepServer, two primary data server and three



replicate data servers. Data servers may act both as a primary and a replicate in a replication system for bi-directional replication of transactions. RepServers can be configured to pass transactions between each other to achieve a degree of load balancing.

Since the database operations are read from the database transaction log, written to the primary RepServer's stable queue, read from that queue and perhaps handed to other RepServer's where they are written to and read from still more stable queues before finally being applied to the replicate database, the process of replicating operations is obviously asynchronous; that is, operations are applied to the replicate database *after* they have been applied to the primary database. While this asynchronous mechanism, among other things, provides for guaranteed operation delivery even when the replicate database may be unavailable for a time, it also introduces a delay between the time an operation is committed at the primary and when it is committed at the replicate. For many applications, such as in financial market trading applications, this delay or latency can become costly if it gets very large (more than a few seconds). Consequently, a replication system and RepServer in particular, are constantly being scrutinized for opportunities to reduce latency.

In addition to the problem of latency, typical applications also must address the problem of throughput; that is, the number of operations or amount of data that can be replicated in a fixed amount of time. And it is often the case, that the same application must address both problems but at different times in the day. Taking the trading application for example, during the business day trade transactions are relatively small in number of bytes but must be moved quickly so that orders may be finalized and short-term market trends determined from the latest information. On the other hand, at night an accumulation of the day's completed transactions may need to be distributed out to field offices for long-term market analysis or off-site backup and disaster-recovery purposes. And these nightly update activities tend to be massive, typically measured in gigabytes per hour of replicated activity. In a poorly tuned

replication system, nightly update activities make not complete until well after the next day's business activities have started resulting in lost business opportunities.

As one might expect, RepServer offers dozens of configuration options, many of which are designed to tune performance in terms of decreasing latency or increasing throughput. But in many cases, the configuration settings that yield the lowest latency may harm throughput when replication volume is large, and visa versa when replication volume is low. And of course, there are many situations when low latency must be achieved along with high throughput so that the configuration of RepServer must be balanced between the two goals.

Today, customers must manually monitor the nature or profile of the transactions being replicated at various times in the day, and adjust RepServer configuration accordingly to achieve either low latency or high throughput or a combination of the two. But RepServer performance and the manner in which RepServer is configured to achieve best performance depends upon much more than RepServer's configuration. In addition, it depends upon the number and speed of the available processors, the type and availability of memory, the disk devices underlying the stables queues, the performance of the network supporting the replication system, the topology of the replication system including the number of primary and replicate databases supported by the RepServer, and the performance of the primary and replicate data servers and the RepAgents as well. Proper tuning for a given transaction profile in one environment may not work at all for a similar profile in a different environment. With so many variables, the task of configuring RepServer at any given moment to perform at its best has become a nearly impossible manual task. And evidence from both customer cases and controlled experiments[1] indicate that RepServer performance is a non-linear function of parameters both within and outside the control of RepServer.

In this paper we will examine the opportunities for using the techniques of unsupervised, reinforcement machine learning with a neural network implementation to relieve the RepServer administrator of the burden of tuning RepServer performance and have RepServer tune itself. After all, the most significant achievements in computing systems have come from having those systems manage their own complexity.[1]

2. RepServer Tuning Configurations

As noted, there are dozens of configuration options for RepServer, many of them related to performance tuning. An overview is available in RepServer's product documentation.[3] For the purposes of our discussion here, we will focus on just a couple of the more prominent configurations.

2.1. SQT Cache Size

Among the many modules[2] of which RepServer is comprised, one is the Stable Queue Manager/Transaction Interface or SQT for short. This module is charged with reconstituting and managing transactions[3] in memory as their component operations are read from the stable queue. When a client module requests a transaction from

[1] The proprietary nature of the data referenced here prevents its inclusion in this paper.

[2] RepServer was developed in the late '90s in C by a group of procedure-oriented engineers. However, the "modules" in RepServer may be viewed as collaborating classes from an object-oriented perspective, each with their own set of well-defined responsibilities, encapsulated state and in many cases, multiple instances. Some modules operate upon their own thread, some do not, still others may either be threaded or non-threaded depending upon the context of their use. The SQT module falls into the later category.

[3] A transaction is a group of one or more operations that either succeed or fail as a whole unit when applied to a database server.

SQT, SQT will give control of the transaction to the client. Once the client is finished with the transaction, it will notify SQT. When SQT determines that all requirements for the transaction have been satisfied (it has been successfully applied to all subscribing servers), then SQT will delete the transaction from memory and request that it be deleted from the stable queue. For the most part, all the operations of a transaction must be loaded into SQT Cache before it qualifies for access by a client module.

The memory into which SQT places and manages transactions is called SQT Cache. The size of SQT Cache plays an important role in RepServer performance, and it is perhaps, the most sensitive to changing transaction profiles. If SQT Cache is too small, fewer transactions can be loaded into cache for client access and so, fewer transactions are available at any given time to be applied to replicate database servers. Also, SQT Cache may be too small for the operations of any transaction to be fully loaded into cache in which case SQT will retain a transaction token in cache, and flush some the partially loaded transactions' cached operations to make room for more operations to be read. When a client requests access to a transaction whose operations have been flushed from cache, SQT must make a costly re-scan of the stable queue to reload the transaction's operations.

On the other hand, because of its design, if SQT Cache is too big, SQT will spend more time trying to fill its cache and less time answering requests from clients for access to transactions. As a result, the clients become starved for transactions and latency suffers.[4] While this starvation problem is well known, for a number of reasons it is not convenient to redesign SQT to resolve the problem in the foreseeable future.

SQT Cache is one candidate configuration for RepServer to "self-tune".

2.2. Bulk Threshold

Most data servers today admit several interface options for applying data manipulations including language commands, dynamically prepared SQL and its "cousin", array updates, and perhaps a proprietary bulk load interface. Each interface option has its advantages and disadvantages. Typically the language option is the most flexible but the least fast. Dynamically prepared SQL and array update interfaces are fairly flexible and fast but incur some overhead at statement preparation time and during various prepared statement cache management operations. For data servers that support a proprietary bulk load interface, these are typically the least flexible but the fastest of all but also require a good bit of overhead up front to set up.

RepServer offers the opportunity to take best advantage of the available data manipulation interfaces given the profile of transactions it is dealing with currently and the interfaces supported by the replicate data server. However, today all of the effort for identifying and taking the opportunity rests in the hands of RepServer administrators who are not always on-hand to make adjustments and even if they are, may not be monitoring RepServer or may not understand the implications of what the monitoring is telling them. For example, in order to take best advantage of a proprietary bulk load interface, the transaction must contain at least enough successive INSERT statements of sufficient length in terms of amount of data being inserted with each statement, to justify the overhead of setting up a bulk load operation.

[4] Throughput tends not to suffer so much in these cases since when SQT realizes that there are starving clients, it "feeds" them all at once, opening a huge hole in cache which it then spends its time trying to fill, ignoring the clients once again.

As with SQT Cache size, the threshold for starting a bulk operation is set to a default value, but it is rarely the case that the default value is the optimum value for the given RepServer and the environment into which it has been deployed.

Bulk threshold is another candidate configuration for RepServer to “self-tune”.

3. Monitoring RepServer Performance

RepServer employs a feature known as Monitors & Counters. M&C is an internal mechanism used to monitor and count various activities within RepServer during its processing cycles. There are literally hundreds of things that are monitored and counted such as the number of LTL commands received, and written the stable queue, the number and type of commands per transaction, the number of transactions, the time spent writing to the queue, the time spent reading from the SQT cache, the number of bytes per command, amount of bytes the average transaction takes up in SQT cache, the amount of time taken to apply a transaction in the replicate database, etc.

Counters that we will be interested in specifically for this self-tuning effort will be counters associated with throughput and latency. Actually, there are a number of such counters which consider throughput and latency at various points internally within RepServer. For the sake of simplicity we’ll consider two generic counters named *throughput* and *latency*. It is also helpful to know that by default, counters accumulate over a specified observation period, are recorded and then reset to begin the new observation period, and that observation periods are all of an equal amount of time.

4. Reinforcement Learning Mechanism Design

To perform the process of self-tuning its performance, RepServer must have a mechanism to review its state in terms of performance and suggest changes to its configuration that may yield a state with better performance. We will explore the possibility of using machine reinforcement learning as that mechanism. The reinforcement learning mechanism will be use a neural network to learn a Q function and the neural network will use Moller’s Scaled Conjugate Gradient algorithm to train.[5] The remaining discussion will assume a general design as presented by Chuck Anderson to his CS545 class as part of assignment 7 during the fall 2009 session at Colorado State University.[4] Also, while the implementation of this mechanism in RepServer will likely be done with C of C++, the examples in this paper will be written in R.

4.1. Defining the Network

The following will be used as a definition for the neural network:

```
makeNN <- function(ni,nh,no,stateRanges) {
  ## stateRanges is 2 x ni, as if two extreme samples
  V <- matrix(0.1*(runif((ni+1)*nh)-0.5), ni+1,nh)
  W <- matrix(0.1*(runif((nh+1)*no)-0.5), nh+1,no)
  standardizeF <- makeStandardizeF(stateRanges)
  list(V=V, W=W, standardizeF=standardizeF, lambda=NULL)
}
```

Here, *ni* is the number of inputs, *nh* is the number of hidden units, *no* is the number of outputs, and *stateRanges* is the pairwise range within which our possible states reside. Taking the each in turn:

We have two inputs since there are two components to our state, throughput and latency. For these we will refer to the counters *throughput* and *latency* and they will be collected at the end of each observation period. However, in the interest of collecting sufficient samples of state to obtain a true gradient, we will not update the network at the end of each observation period. The number of observation periods for which to collect samples before updating the network, *numSamples*, may be imperially obtained under different conditions (i.e. different varieties of transaction profiles) in a controlled environment and then the RepServer Administrator may choose the condition most like the one at hand in order to set this number for their RepServer.

Based upon observations from other deployments of neural networks[5], the optimal number of hidden units seems most influenced by the number of inputs and outputs. The more inputs and outputs the network has, the more hidden units seem to be in order. Again, the optimal number of hidden units may be imperially obtained in a controlled environment and possibly delivered unconfigurable if experiments agree with observations that with a fixed number of inputs and outputs, an optimal number does not depend upon other factors.

The outputs of the network are recommendations to increase, decrease or let alone, the settings for the configurations for which self-tuning is desired. In this case, we have two such configurations, SQT Cache Size and Bulk Threshold. It should be noted that there is a bit of a relationship between the two in that SQT Cache Size should not be so small that the number of operations indicated by Bulk Threshold can not be cached before a bulk operation can be started. Otherwise, the transaction will be flushed from cache resulting in it be rescanned from the stable queue to support the bulk operation. The upshot is that the bulk operation will likely be starved for operations from the queue as they are being rescanned.

Given that relationship, having one network output suggestions for both configurations seems reasonable. However, it is not completely clear that better results may be obtained if each configuration is treated with a separate Q function. And further, managing actions for two configurations seems to imply greater complexity in implementation (for example, with indicator variables) so for the sake of simplicity in discussion, we will confine our self-tuning mechanism to one neural network implementing one Q function outputting actions for one configuration, but the shared Q function option should be investigated further. Additionally, we will design the Q function to simply indicate a recommendation of increase, decrease or no change, and leave the amount by which the configuration should be increased or decreased to mechanisms outside of the Q function.

A configuration with three possible actions suggests three outputs from the network: increase, decrease, no change.

[5] These observations were made in the fall of 2009 in the CS545DL course offered by Colorado State University.

The range of states may be represented by a 2×2 matrix providing the maximums and minimums of our inputs. The values given in Table 1 have been determined from observing typical ranges at customer installations of RepServer.

	Minimum	**Maximum**
***throughput* (GB/minute)**	0.00001	1.25
***latency* (seconds)**	0.1	600

Table 1: A matrix of maximum and minimum values for the network inputs, *throughput* and *latency*, which forms the network construction's *stateRanges* parameter.

Network training is performed using the Scaled Conjugate Gradient algorithm proposed by Moller for its speed as compared to a "steepest descent" approach, for example.[6] It is adapted from Moller's original to support unsupervised learning.

4.2. Applying Reinforcement

Reinforcement will appear in the form of -1, 0 or 1 depending upon whether the current state is worse, the same or better than the previous state. Given the two inputs, *throughput* and *latency*, to determine state, we may form the matrix in Table 2 to help define the reinforcement that should be applied under any given condition.

Reinforcement applied	***latency* decreased**	***latency* stayed the same**	***latency* increased**
***throughput* increased**	1	1	?
***throughput* stayed the same**	1	0	?
***throughput* decreased**	?	?	-1

Table 2: Reinforcements to be applied given the current and previous state. "?" indicates a lack of information to make a good assessment.

In considering only *throughput* and *latency* to determine the nature of the state change, there is a difficulty in that proper context is not available to always make a good determination. For example, if *throughput* decreases, it may be due to a decrease in replicated activity at the primary database, and no amount of adjusting RepServer configuration can change that. To allow for this possibility, we will need to expand the inputs to the network to include proper context for determining whether increases, decreases or no change in *throughput* or *latency* is good or bad or neither. In particular, to establish context for the example provided, reduced activity at the

primary, we would want to include operations per second or counter *opsPerSec* in the network input. This *opsPerSec* counter counts the number of LTL operations received per second. Using this input, the reinforcement table expands by one dimension which we represent in the following tables.

Reinforcement applied for increased ***opsPerSec***	***latency* decreased**	***latency* stayed the same**	***latency* increased**
***throughput* increased**	1	1	?
***throughput* stayed the same**	1	0	-1
***throughput* decreased**	?	?	-1

Table 3: Reinforcements to be applied given the current and previous state when opsPerSec has increased.

Reinforcement applied for unchanged ***opsPerSec***	***latency* decreased**	***latency* stayed the same**	***latency* increased**
***throughput* increased**	1	1	?
***throughput* stayed the same**	1	0	-1
***throughput* decreased**	?	-1	-1

Table 4: Reinforcements to be applied given the current and previous state when opsPerSec has not changed.

Reinforcement applied for decrease in ***opsPerSec***	***latency* decreased**	***latency* stayed the same**	***latency* increased**
***throughput* increased**	1	1	0
***throughput* stayed the same**	1	0	-1
***throughput* decreased**	1	0	-1

Table 5: Reinforcements to be applied given the current and previous state when opsPerSec has not changed.

And again, it becomes apparent that yet more inputs are needed to make a good decision. Particularly, the size of an operation in terms of bytes, *opBytes*, can affect the decision. For example, suppose *opsPerSec* remains constant but *opBytes* increases. Then at a minimum we would prefer to see *latency* stay the same and *throughput* increase. That state should get 0 reinforcement as it does not add or detract from the previous state when *opBytes* was smaller.

Clearly, a number of more inputs are required in order to assess whether state is improving or not between samples. In addition, we may find this assessment less complex if we were to only consider either *throughput* or *latency* as our performance measure rather than the two together. For the sake of continuing the discussion, we will assume that we keep *throughput* and *latency* as our performance measures and that we have added enough context through the addition of other inputs such as *opsPerSec* and *opBytes* to make reasonable decisions about reinforcement. Adjustments to the *stateRange* parameter for network generation would need to be updated accordingly.

4.3. Choosing the Next Action

To choose the next action, observations of other neural network Q function deployments mentioned earlier suggest that there can be great value in occasionally choosing a random action, particularly when the goal changes. In this case, while the goal of high throughput and low latency does not change, the determination of when that goal has been reached will. That is to say, the highest throughput and lowest latency achievable under one set of circumstances, most notably, transaction profile, may be very different from the highest throughput and lowest latency achievable under a different transaction profile.

As the transaction profile changes, the Q function should be allowed to explore a random action assuming that the best state achievable under the old transaction profile is no longer achievable under the new transaction profile or perhaps a better state than that of the old transaction profile is achievable. And even if the transaction profile does not change, an occasional exploratory trip may prove fruitful. But if the random action and subsequent states yield no fruit after some number of network updates, we should allow for the best known state given the transaction profile to be restored so that RepServer does not "lose its way" in self-tuning to the point that it can not at least return to a good state. This suggests that transaction profiles and corresponding states along with the configuration used to achieve those states should be saved away for future reference. And this would be the case not only for recovery from an unfruitful exploratory trip, but also to initialize from a reboot of RepServer.

This also suggests that the probability of a random action needs to be based upon a change in transaction profile which then decays down to some minimal value. And further, there may need to be a mechanism whereby the RepServer administrator may disable the random actions when a satisfactory state has been achieved so that an occasional random action yielding no benefit will also not result in degraded performance at critical times during the business day.

5. Extended Kalman Filter Training

Investigating a similar solution to a similar problem, Won Seok Oh, et al propose using the Extended Kalman Filter (EKF) network training technique since it requires less training samples than gradient descent algorithms.[7] The network proposed here has someone of a different structure than that described by Chuck Anderson in his fall 2009

CS545 class at Colorado State University. Whereas Dr. Anderson's structure is comprised of two-layers, "hidden" and "output", the proposed in Oh's paper has a single layer. (See Figure 1.) In the two-layer design, the activation function is applied to its inputs in the hidden layer and then the network output, the weighted sum of the hidden layer outputs, are produced by the output layer. On the other hand, in the single-layer design, the sum of the weighted inputs are calculated first and then the activation function applied.

While it is not clear what differences the two network designs may yield in terms of performance or output quality, the EKF training technique does not appear to be dependent upon network design so that it may be substituted for SCG in the two-layer network originally assumed in this paper. Based on Oh's paper, the expectation would be that the technique is more suitable to real-time self tuning and so, perhaps better performing than a gradient descent, and producing good output with few sample and so perhaps more responsive to changes in the state.

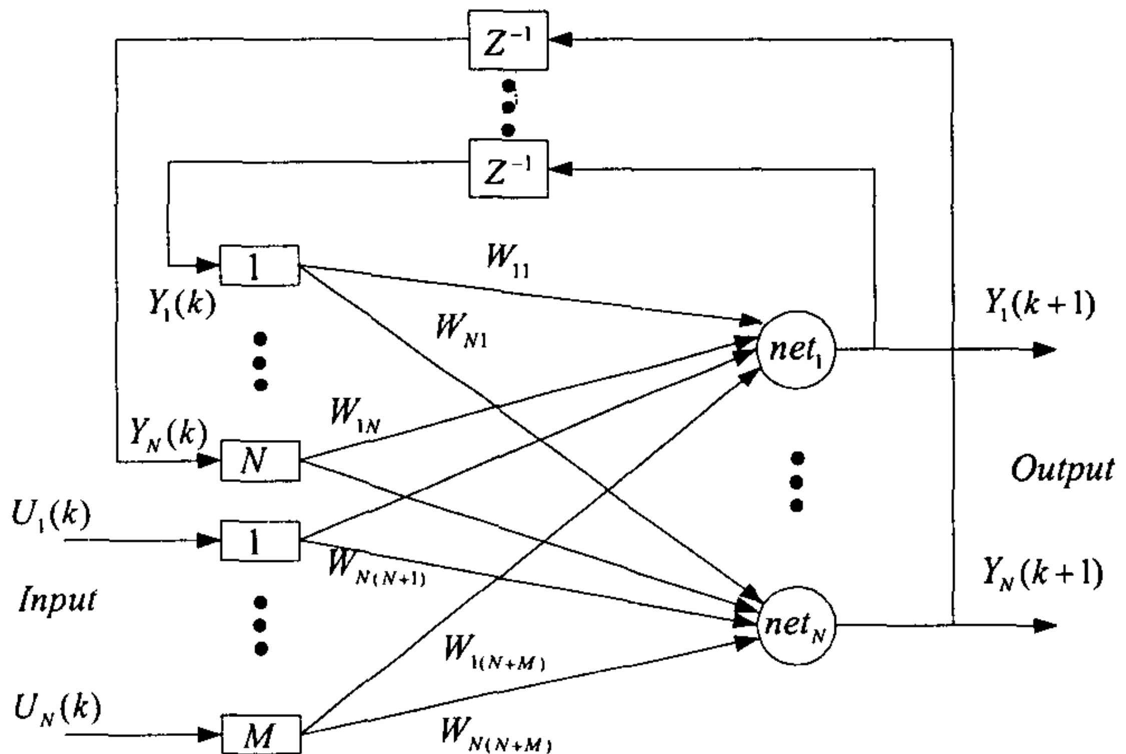


Figure 2: The single-layer neural network design presented by Won Seok Oh et al for solving a similar self-tuning problem.

6. Conclusion

A self-tuning solution to the problem of RepServer performance is highly desirable due to the difficulty in manually tuning RepServer particularly as its state is in constant flux, requiring constant configuration adjustments to maintain optimal performance. A reinforcement machine learning solution seems feasible. But depending upon the training algorithms used in the solution, there is a danger that the self-tuning mechanism itself and the requirements it places upon RepServer to operate may become a factor itself in the performance of RepServer. Therefore, a variety of network designs and training mechanisms should be evaluated not only for the quality of its tuning abilities, but also for its ability to produce quality results with minimal impact on the RepServer itself.

References

[1] "**Carter, Greg**." Rube Goldberg, Shifting Paradigms and Differentiation in Sybase DI Suite 2.0. Unpublished, Sybase, Inc. June 2007.

[2] SYBASE REPLICATION SERVER PERFORMANCE AND TUNING: Understanding and Achieving Optimal Performance with Sybase Replication Server. White Paper, Sybase, Inc. January 2007. http://www.sybase.com/files/White_Papers/Sybase_RepServer_Performance_Tuning_wp_022708.pdf.

[3] Sybase Replication Server. Product Documentation, Sybase, Inc., August 2009. http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.infocenter.help.rs.15.2/title.htm

[4] "**Anderson, Charles**." assignment7-code-start.R, gradientDescents.R, mlUtilities.R, and nnQ.R. Assignment 7 Starter Code, CS545, Department of Computer Science, Colorado State University. Fall 2009.

[5] "**Anderson, Charles**." TD Value Updates as Gradient Descent. Course Notes, CS545, Department of Computer Science, Colorado State University. Fall 2009.

[6] "**Moller, Martin F.**" A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning. Neural Networks, vol. 6, pp. 525-533, 1993.

[7] "**Oh, Won S., Bimal K. Bose, Kyu Min Cho and Hee Jun Kim**" Self Tuning Neural Network Controller for Induction Motor Drives. IEEE 2002 28th Annual Conference of the Industrial Electronics Society, Volume 1, Page(s):152 – 156, Nov. 2002.