Parallel Learning and Decision Making for a Smart Embedded Communications Platform

Karen Zita Haigh
Allan M. Mackay, Michael R. Cook, Li L. Lin
Raytheon BBN Technologies
10 Moulton St, Cambridge, MA 02138
khaigh@bbn.com

August 20, 2015

Abstract

Mobile ad hoc networks (MANETs) operate in highly dynamic environments where it is desirable to adaptively configure the radio and network stack to maintain consistent communications. Our goal is to automatically recognize conditions that affect communications quality, and learn to select a configuration that improves performance, even in highly-dynamic missions. We present our ML application, and the multi-threaded architecture that allows rapid decision making to proceed in parallel with a slower model-building loop. We present the challenges of managing shared memory so that the learning loop does not affect the faster decision making loop. We present results that show performance of our Strategy Optimizer, focusing on incremental in-mission learning.

1 Introduction

Mobile ad hoc networks (MANETs) operate in highly dynamic, potentially hostile environments. Current approaches to network configuration tend to be static, and therefore perform poorly. It is instead desirable to adaptively configure the radio and network stack to maintain consistent communications. Our goal is to automatically recognize conditions that affect communications quality, and select a configuration that improves performance, even in highly-dynamic missions. Our domain requires the ability for a decision maker to select a configuration in real-time, within the decision-making loop of the radio and IP stack. A human is unable to perform this dynamic configuration partly because of the rapid timescales involved, and partly because there are an exponential number of configurations.

Our domain also requires the ability to learn on the fly during a mission, for example to recognize changes to the RF environment, or to recognize when components have failed. The system must learn new models at runtime that accurately describe new communications environments. A real-time decision maker then uses these newly learned models to make decisions.

This paper describes our effort to place Machine Learning on the general purpose processor of an embedded communications system. We have chosen Support Vector Machines (SVMs) as the modeling approach [23, 24]. Our first step was to generate code that could run reliably and quickly on the platforms; we addressed this challenge by optimizing an existing SVM library for the embedded environment [6].

This paper focusses on our multi-threaded architecture that supports parallel model-building and decision-making. We have a rapid decision-making module that selects the configurations, and a slower learning loop that updates the models as it encounters new environmental conditions.

While the presentation and results in this paper are focused on a communications domain, the parallel ML and automatic configuration approach is relevant for any smart embedded system that has a heterogeneous suite of tools that it can use to change its behavior.

2 Embedded Communications Domain

Our target domain is a communications controller that automatically learns the relationships among configuration parameters of a mobile ad hoc network (MANET) to maintain near-optimal configurations automatically in highly dynamic environments. Consider a MANET with N nodes; each node has

- a set of observable parameters o that describe the environment,
- a set of controllable parameters c that it can use to change its behavior, and
- a metric m that provide feedback on how well it is doing.

Each control parameter has a known set of discrete values. We denote a *Strategy* as a combination of control parameters (CPs). The maximum number of strategies is $\Pi_{\forall c} v_c$, where v_c is the number of possible values for the controllable parameter c; if all n CPs are binary on/off, then there are 2^n strategies, well beyond the ability of a human to manage. The goal is to have each node choose its strategy s, as a combination of controllables c, to maximize performance of the metric m, by learning a model f that predicts performance of the metric from the observables and strategy: $m = f(o, s)$. The mathematics of this domain is described in more detail elsewhere [7, 8].

Observables: In this domain, observables are general descriptors of the signal environment. We generally use abstracted features rather than raw data; for example, observables might range from -1 to 1 (strongly “is not” to strongly “is”) relative to expectations. We capture these statistics at all levels of the IP stack, including data such as saturation, signal-to-noise ratio, error rates, gaussianness, repetitiveness, similarity to own communications signal, link and retransmission statistics, and neighborhood size.

Controllables: Controllables can include any parameter in the IP stack or on the radio, such as those settable through a Management Information Base (MIB). We can also consider choice of algorithm, modeled as a binary on/off. We assume a heterogeneous radio that the learner can configure arbitrarily, including techniques in the RF hardware, the FPGAs and the IP stack. For example,

- Antenna techniques such as beam forming and nulling,
- RF front end techniques such as analog tunable filters or frequency-division multiplexing,
- PHY layer parameters such as transmit power or notch filters,
- MAC layer parameters such as dynamic spectrum access, frame size, carrier sense threshold, reliability mode, unicast/broadcast, timers, contention window algorithm (linear, exponential, etc), neighbor aggregation algorithm
- Network layer parameters such as neighbor discovery algorithm, thresholds, and timers,
- Application layer parameters such as compression (e.g., jpg 1 vs 10) or method (e.g., audio vs video)

Metrics: Metrics can include measures of *effectiveness* such as Quality of Service, throuphput, latency, bit error rate, or message error rate, as well as measures of *cost* such as power, overhead, or probability of detection.

2.1 Target Platforms

We integrated into two existing embedded systems for communications, each with pre-established hardware and runtime environments. These are legacy systems on which we are deploying upgraded software capabilities. Both platforms have general-purpose CPUs with no specialized hardware acceleration. We consider this an embedded system because it is dedicated to a specific set of capabilities, has limited hardware resources, limited operating system capabilities, and requires an external device to build and download its runtime software. Our embedded platforms are:

ARMv7: ARMv7 rev 2 (v7l) at 800 MHz, 256 kB cache, vintage 2005. Linux 2.6.38.8, G++ 4.3.3, 2009.

PPC440: IBM PPC440GX [1] at 533MHz, 256 kB cache, vintage 1999. Green Hills Integrity RTOS, version 5.0.6. We use the Green Hills Multi compiler, version 4.2.3, which (usually) follows the ISO 14882:1998 standard for C++.

For comparison, we also show timing results on a modern Linux server:

Linux: 16 processor Intel Xeon CPU E5-2665 at 2.40GHz, 20480 kB cache, vintage 2012. Ubuntu Linux, version 3.5.0-54-generic. g++ Ubuntu/Linaro 4.6.3-1ubuntu5, running with `-std=c++98`.

3 Related Work

Adaptive techniques have been applied to improving network performance with some success. The term *Cognitive Radio* was first introduced by Mitola [13], and the concept *Cognitive Networks* extends this idea to include a broader sense of network-wide performance [5, 21]. A cognitive network must *identify and forecast network conditions*, including the communications environment and network performance, *adapt* to constantly changing conditions, and *learn* from prior experiences so that it doesn't make the same mistakes. A cognitive network should be able to adapt to continuous changes rapidly, accurately, and automatically.

Cognitive network approaches are similar to cross-layer optimization, in that both attempt to optimize parameters across multiple layers in the IP stack. However, most cross-layer approaches handle independent goals. Not only is this independence suboptimal because it is non-convex [12], it may also lead to adaptation loops [11]. A cognitive network selects actions that consider the objectives jointly.

The most notable drawback of most cross-layer optimization approaches is that they are *hand built* models of the interactions among parameters. Even assuming that the original model is correct, this approach is not maintainable as protocols are redesigned, new control parameters are exposed, or the objective function changes. A cognitive network *learns* from its experiences to improve performance over time. Machine Learning (ML) techniques automatically learn how parameters interact with each other and with the domain. Rieser [18] and Rondeau [19] used genetic algorithms to tune parameters and design waveforms. The experiments show no data about how fast it works and moreover the learning appears to operate offline; Rieser states explicitly that it "may not be well suited for the dynamic environment where rapidly deployable communications systems are used." Newman et al [15, 16] similarly use genetic algorithms to optimize parameters in a simulated network; they also show no time results. Montana et al [14] used a genetic algorithms approach for parameter configuration in a wireline network that can find the 95% optimal solution in "under 10 minutes." Chen et al [2] use Markov Decision Processes (MDPs) and Reinforcement Learning to identify complex relationships among parameters. After 200 samples of 10-second intervals (approximately 30 minutes), the system converges to an optimal policy. Fu and van der Schaar [4] also use MDPs to make optimization decisions. They acknowledge that it may be difficult to handle-real time scenarios, and therefore decompose the problem. The solution for these semi-independent goals takes about 100 iterations to converge.

To our knowledge, we are the only group to have demonstrated a cognitive network that learns and makes decisions for a rapidly changing environment. Using the ML design of Haigh [8], ADROIT [22] used Artificial Neural Networks to model communications effectiveness of different strategies. We were the first to demonstrate ML in an on-line (real-time) real-world (not simulation) MANET. Each node in our distributed approach learned a different model of global performance based on local observations (i.e., no shared information), thus meeting MANET requirements for rapid local learning and decision making. This paper presents our continuing work to improve accuracy, speed, and breadth of the learning and decision making. A key difference is that our current system supports *parallel* decision making and learning; i.e., where the node can use a previously-learned model to make rapid decisions, while the node is building a new one in a slower loop.

4 Machine Learning and Regression

Our goal is to have each node choose its strategy s, as a combination of controllables c, to maximize performance of the metric m, by learning a model f that predicts performance of the metric from the observables and strategy: $m = f(o, s)$. Support Vector Machines [23, 24] are ideally suited to learning this regression function from attributes to performance. The regression function is commonly written as:

$$m = \mathrm{SVM}(o, s) = f(o, s) = f(x) =< w, x > +b$$

where x are the attributes (combined o and s), w is a set of weights (similar to a slope) and b is the offset of the regression function.

When the original problem is not linear, we transform the feature space into a high-dimensional space that allows us to solve the new problem linearly. In general, the non-linear mapping is unknown beforehand, and therefore we perform the transformation using a *kernel*, $\phi(x_i, x)$, where x_i is an instance in the training data that was selected as a support vector, x is the instance we are trying to predict, and where ϕ is a vector representing the actual non-linear mapping. In this work, we use the *Pearson Universal Kernel [23]* because it has been shown to work well in a wide variety of domains, and was consistently the most accurate in our target communications domain:

$$\phi(x_i, x) = \frac{1}{\left[1 + \left(\frac{2}{\sigma}\sqrt{||x_i - x||^2}\sqrt{(2^{(1/\omega)} - 1)}\right)^2\right]^\omega} \tag{1}$$

ω describes the shape of the curve; as $\omega = 1$, it resembles a Lorentzian, and as it approaches infinity, it becomes equal to a Gaussian peak. σ controls the half-width at half-maxima.

The regression function then becomes:

$$\begin{aligned} m &= \mathrm{SVM}(o, s) \\ &= \sum_{i=1}^{n} (\alpha_i - \alpha_i^*)\, \phi(x_i, x) + b \end{aligned} \tag{2}$$

where n is the number of training instances that become support vectors, α_i and α_i^* are Lagrange multipliers computed when the model is built (see Üstün et al [23]).

There are many available implementations SVMs, e.g., [10, 25]. After evaluating these on 19 benchmark datasets [6], we selected the SVM implementation found in Weka [9], with Ũstün's Pearson VII Universal Kernel (Puk) [23] and Shevade's Sequential Minimal Optimization (SMO) algorithm [20] to compute the maximum-margin hyperplanes. We performed a series of optimizations on data structures, compiler constructs, and algorithmic restructuring, which ensured that we could learn an SVM model on our target platform within the timing requirements of our domain [6].

This paper describes the changes we made to Weka to support multi-threaded processing, in which a Rapid Response Engine (RRE) makes real-time decisions, while a Long Term Response Engine (LTRE) updates the learned models on the fly.

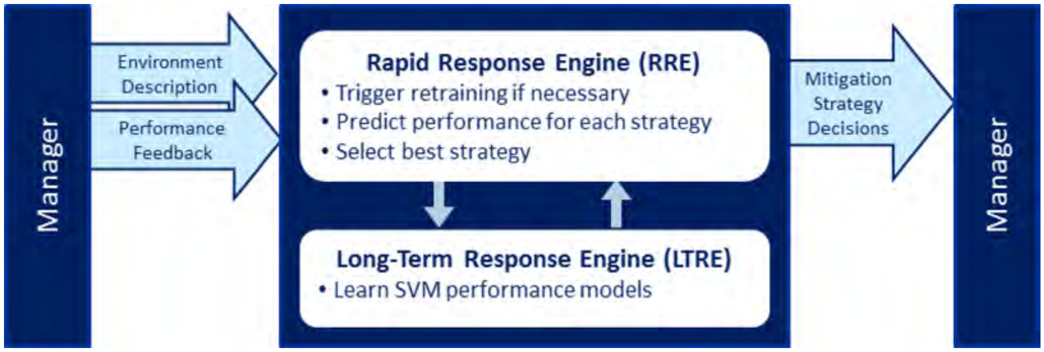


Figure 1. The BBN Strategy Optimizer comprises a Rapid Response Engine (RRE) that makes strategy decisions, and a Long Term Response Engine (LTRE) that learns the models of how strategies perform.

5 Architecture and Component Algorithms

The Strategy Optimizer must choose an effective set of controllable parameters c to maintain quality communications given the interference environment described by the observables o. Figure 1 illustrates the architecture of the BBN Strategy Optimizer. The Rapid Response Engine (RRE) must make strategy decisions within the timescales necessary for the domain. When invoked for retraining, the Long Term Response Engine (LTRE) learns new models, and then uploads them to the RRE.

There are three key datastructures in the Strategy Optimizer. An *Instance* is the representation of each observation of the RF environment, including observables o, controllables c and the observed performance metric m. An *Attribute* contains metadata about each observable and controllable, including its maximum & minimum values. The *Dataset* is a wrapper class for all Instances and Attributes, plus cached computations.

5.1 Manager

The Manager receives environment descriptions from the signal processing modules or the communications stack. If observations arrive at different rates or times, e.g., FPGA data arrives at a different time from IP stack statistics, then the Manager collects all of the most recent data into an observation. This observation is held as an Instance, and contains all of the information that the RRE needs for its decisions: current observables, current performance metrics, and the controllables currently in place (possibly chosen at a previous time by the RRE).

The Manager also enacts the strategy decisions made by the RRE. For example, if the RRE decides to change the maximum number of retransmissions in the MAC layer, then the Manager sets that configuration value in the IP stack. Similarly, if the RRE decides to apply a notch filter to the signal, the Manager would enable that path in the FPGA.

5.2 Rapid Response Engine (RRE)

The RRE has two primary tasks:

- Select the best strategy based on SVM predictions, and
- If prediction error has exceeded a configured error threshold, instruct the LTRE to retrain on all collected data.

ALGORITHM 1: The RRE operates on every Instance I_t, and updates the SVM when LTRE finishes retraining. Figure 2 illustrates the functional flow.

Input: $\mathcal{S}$: Set of available strategies; recall that a strategy s is a unique combination of controllables c
Input: τ: Error threshold from configuration
Input: I_t: Instance from Manager at time t, containing vector $< \mathbf{o}_t, \mathbf{c}_t, m_t >$ of observables, controllables, and metric
Input: $newSVM$: Support Vector Machine built by LTRE, per Equation 2
Output: $\hat{s}$: Strategy to Manager
Semaphore: TransferSVM($newSVM$) from LTRE at initialization and during mission
Semaphore: Observation(I_t) from Manager during mission
Semaphore: Terminate from Manager at mission end
Semaphore: Retrain to LTRE during mission
Semaphore: Enact($\hat{s}$) to Manager during mission

```
repeat in parallel
    wait until TransferSVM(newSVM) semaphore
        SVM ← newSVM
    end
    wait until Observation(I_t) semaphore            // Must have received at least one SVM
        if |m_{t-1} − m_t| > τ then
            Trigger semaphore Retrain to LTRE
        end
        Normalize I_t
        foreach s ∈ S do                             // Predict performance of each strategy
            p_s = SVM(o_t, s)
        end
        ŝ = argmax s, p_s                            // Select best strategy
        Trigger semaphore Enact(ŝ) to Manager
    end
until semaphore Terminate
```

Algorithm 1 and Figure 2 show the operation of the RRE. Once the RRE receives the initial SVM model from the LTRE with the TransferSVM() semaphore, it operates on each new observation in a continuous loop, waiting for the Observation() semaphore.

It first asesses the quality of the current SVM model: if the predicted value of the metric at time $t-1$, m_{t-1}, is outside of a certain tolerance of the observed value at time t, $\hat{m}_t$, then the model needs to be retrained (Figure 3):

if $|m_{t-1} - \hat{m}_t| > \tau$, then
trigger retraining using Retrain semaphore

The RRE then predicts the performance of each of the available strategies in the current environment. It uses the previously-learned SVM model, $f = (o, s)$ of Equation 2, to make this prediction:

for each candidate strategy s in $\mathcal{S}$
$m_s = \text{SVM}(o, s)$

Finally, the RRE selects the single strategy with the best expected performance, and then sends Enact() semaphore to tell the Manager.

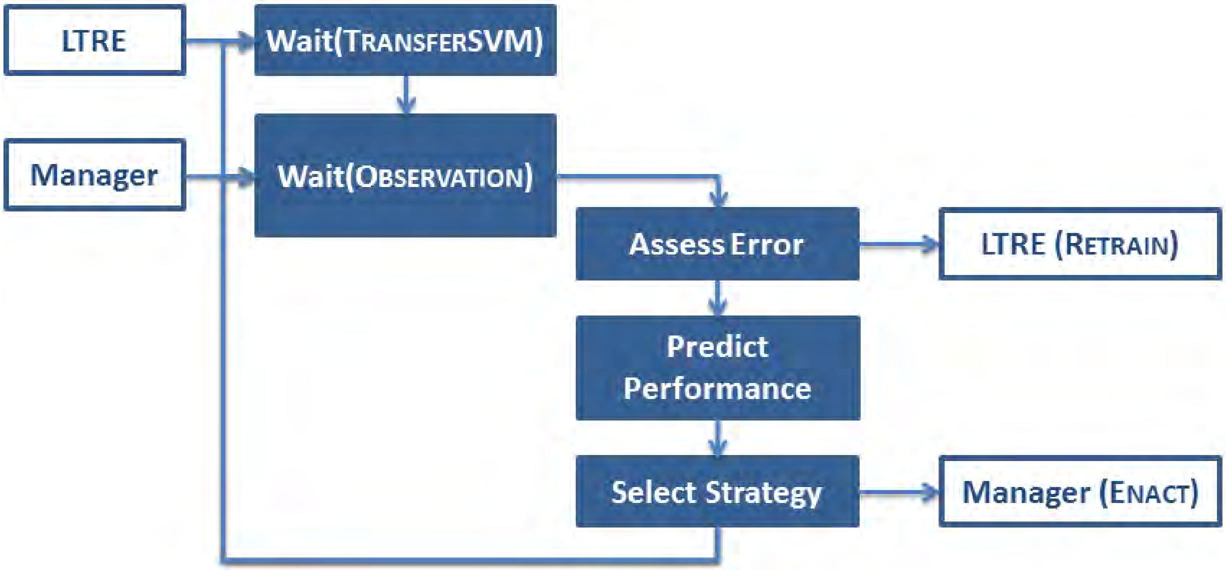


Figure 2. After initialization, the RRE operates in a continuous loop to select strategies. Algorithm 1 outlines the algorithm.

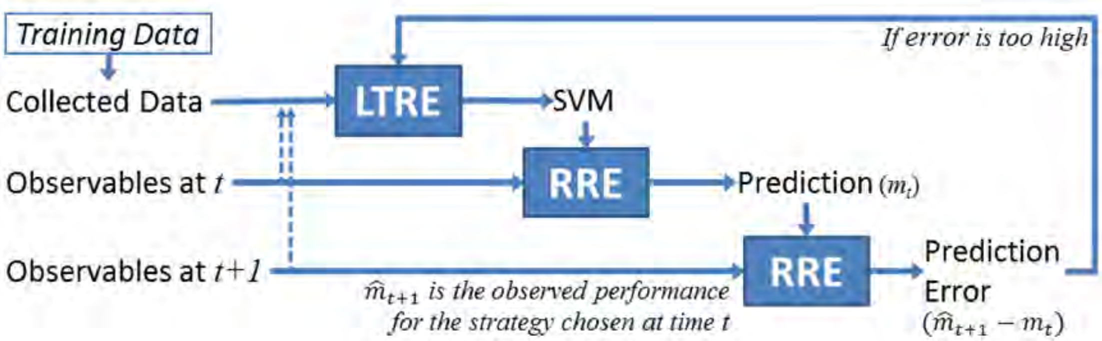


Figure 3. The RRE triggers retraining when the current SVM is predicting performance badly.

5.3 Long Term Response Engine (LTRE)

The LTRE updates the learned models on the fly during a mission. This capability enables the Strategy Optimizer to handle new conditions such as new interference sources, or when a technique is no longer operating effectively (e.g., a component has failed).

The LTRE builds the SVM model, f of Equation 2, from previous observations, and then updates the RRE. The LTRE therefore has two main tasks: to add normalized Instances to the Dataset, and to build the SVM model for the RRE. Algorithm 2 and Figure 4 show the algorithmic structure of the LTRE during the mission.

When the LTRE receives a RETRAIN semaphore from the RRE (or from the Manager during system initialization), it must first normalize the Dataset. If the maximum or minimum value for an Attribute has changed, the LTRE (re)normalizes the Dataset, and then (re)computes and caches the dot product and kernel value for every pair of Instances, per Equation 1. The LTRE then uses Shevade's optimization algorithm to compute the maximum-margin hyperplanes [9, 20, 17]. Finally, the LTRE prepares the learned SVM model for the RRE, and tells the RRE to update with the TRANSFERSVM() semaphore.

ALGORITHM 2: The LTRE updates the learned models during a mission so that the Strategy Optimizer can handle new interference conditions or recognize when techniques are no longer operating effectively. Figure 4 illustrates the functional flow.

Input: I_t: Instance from RRE at time t, containing vector $< \mathbf{o}_t, \mathbf{c}_t, m_t >$ of observables, controllables, and metric
Output: SVM: Support Machine for the RRE, per Equation 2, which includes the offset b, the Support Vectors (SVs), and their corresponding α values
Semaphore: OBSERVATION(I_t) from Manager during mission
Semaphore: RETRAIN from RRE during mission, from Manager during system initialization
Semaphore: TERMINATE from Manager at mission end

```
repeat in parallel
  wait until OBSERVATION(I_t) semaphore             // Process all pending Instances before Retrain
    if |Dataset| > MAXINSTANCES then                // Circular Buffer
      Remove oldest I from Dataset
    end
    Append I_t to Dataset
    foreach observable o_i ∈ o_t do
      if o_i > Attribute[i].max or o_i < Attribute[i].min then   // Flag for renormalization
        Set Renormalization flag for Attribute[i] to true
      end
    end
    foreach Instance I ∈ Dataset do                 // Cache items needed for Retraining
      Compute dotProd(I_t, I)
      Compute φ(I_t, I) using Equation 1
    end
  end
  wait until RETRAIN semaphore
    if ∃ Attribute with Renormalization flag set to true, then
      Renormalize Dataset                           // Renormalize
      foreach pair of Instances I_1, I_2 ∈ Dataset do   // Recompute Cached items
        Compute dotProd(I_1, I_2)
        Compute φ(I_1, I_2) using Equation 1
      end
    end
    Compute SVs using SMO [20]                      // Retrain
    SVM ←                                           // Copy SVs for RRE
        foreach sv ∈ SVs do
          Save offset b for RRE
          Save α^δ_sv = α_sv − α*_sv for RRE
          Make shared pointer to sv for RRE
        end
    Trigger semaphore TRANSFERSVM(SVM) to RRE
  end
until semaphore TERMINATE
```

When the LTRE receives a new Instance with the OBSERVATION() semaphore, it first checks whether the bounds have changed, and if so, flags the Attribute for renormalization. It then adds the Instance to the Dataset, and computes its dot products and kernel values to all other Instances. To live within the memory bounds on the embedded platform, the LTRE uses a circular buffer of Instances; when the bound is reached, the LTRE overwrites the oldest Instance. Note that the LTRE will add all pending Instances before starting a new retraining.

Note that the original implementation of Weka renormalizes the data, and then calls SMO. SMO computes

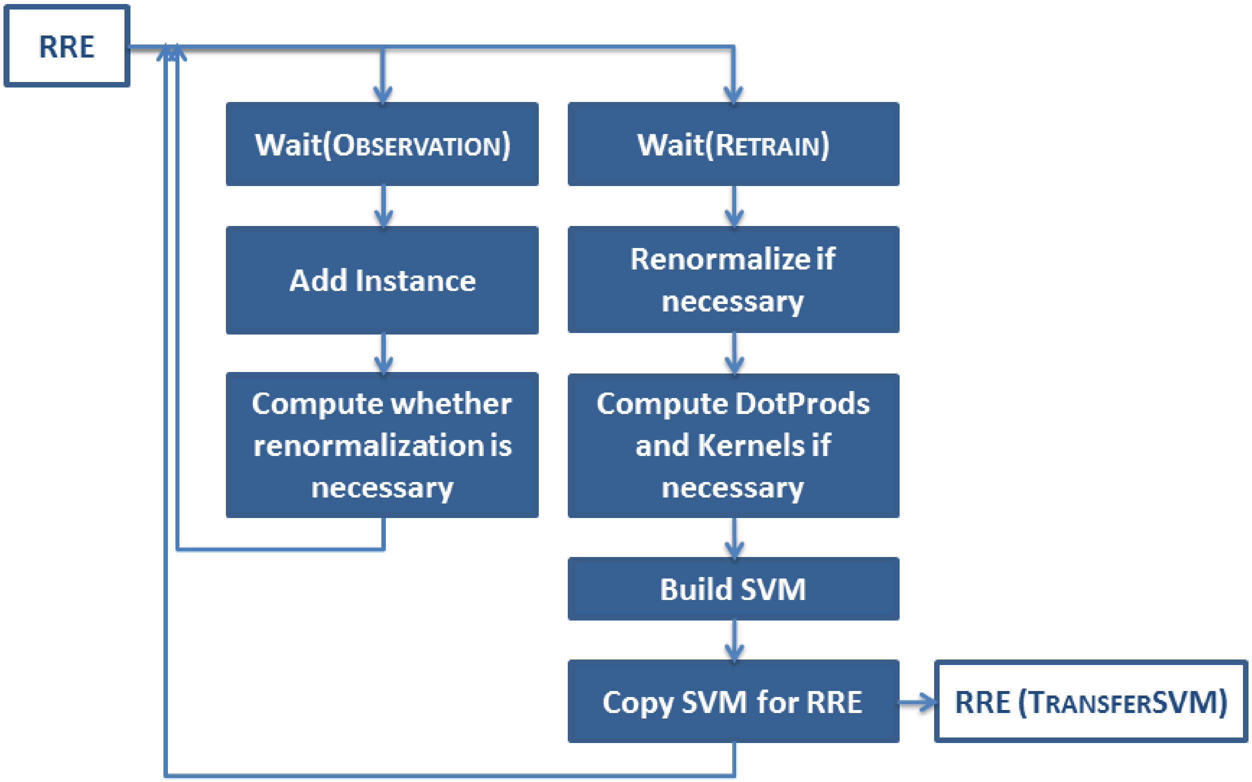


Figure 4. The LTRE manages the Dataset and rebuilds the learned models. Algorithm 2 outlines the algorithm.

all the dot products and ϕ kernel values internally. Because we may not need to renormalize our data at every retraining, we move this computation outside SMO.

6 Threading Model

There are three key threads in the Strategy Optimizer, corresponding to each of the components of Section 5. The Manager is the highest priority thread, as it must respond to incoming data immediately. It buffers the incoming data and queues messages for the RRE. The RRE is the second highest priority, and makes strategy decisions for current conditions. The LTRE is a lower priority thread, where the Strategy Optimizer learns models of how the strategies perform.

Figure 5 shows the sequence diagram of the pre-mission initialization, and the in-mission runtime loop. On startup, the LTRE thread makes its first SVM and notifies the RRE; the mission can start after this first SVM is ready. At runtime, the RRE makes an Instance for every observation that it receives. It then checks to see if the observed performance matches the predicted performance; if there was sufficient prediction error, the RRE triggers retraining in the LTRE. The RRE then predicts the performance for each available strategy in the current communications conditions, and selects the strategy with the highest expected performance for the next round.

We use a *Mailbox* to pass objects (actions or data) between threads. The mailbox is implemented as a max-depth queue.

- The Manager uses a depth-one queue to send data to the RRE, so that the RRE operates on only the most recent data.

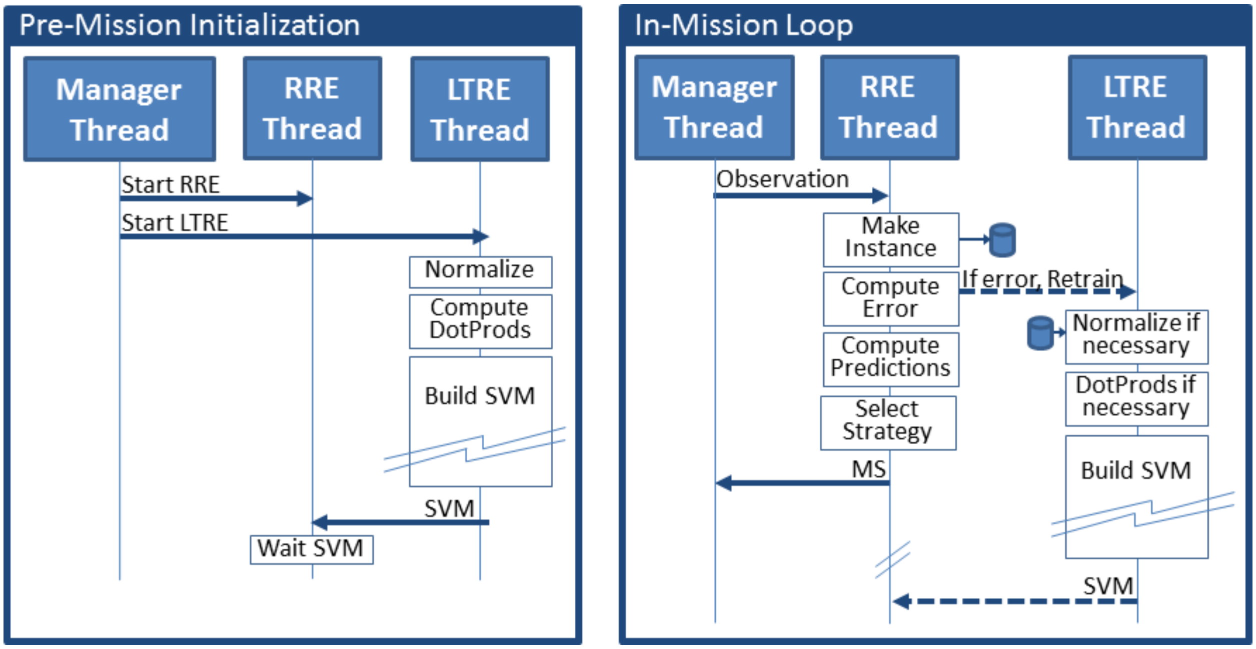


Figure 5. The LTRE builds its first SVM before the mission starts. During the mission the RRE processes each observation, and triggers retraining when necessary.

- The LTRE uses a depth-one queue to send updated SVM models to the RRE.
- The RRE uses a depth-d queue to send data to the LTRE because retraining a single SVM can span a temporal duration where the RRE receives many messages. Figure 6 illustrates a potential sequence of retraining interactions between the RRE and the LTRE. When the LTRE completes a retraining, it adds any pending Instances in the queue to the Dataset, then (if necessary) starts a new retraining.

7 Data Management

During initialization, the LTRE builds an SVM based on its initial training data; the RRE waits until this first SVM is ready. Then during the mission, the RRE processes every new observation while the LTRE waits for a retraining signal. If the RRE detects sufficient prediction error, then it triggers retraining; at this point the RRE and the LTRE are operating in parallel.

As it retrains, the LTRE must ensure that it does not alter any data supporting the RRE's existing SVM since the RRE thread is still using it to predict. While maintaining thread safety of this data, we must also minimize memory usage and data copy overhead.

7.1 Data Elements

The LTRE and RRE both use Instances to represent the observations of the communications environment. A Dataset is the wrapper class for all of the Instances. It maintains a set of Attributes, each of which contains meta-information about an observable or controllable, such as its maximum & minimum & mean values. A Dataset also contains derived calculations such as the pairwise dot products and kernel computations.

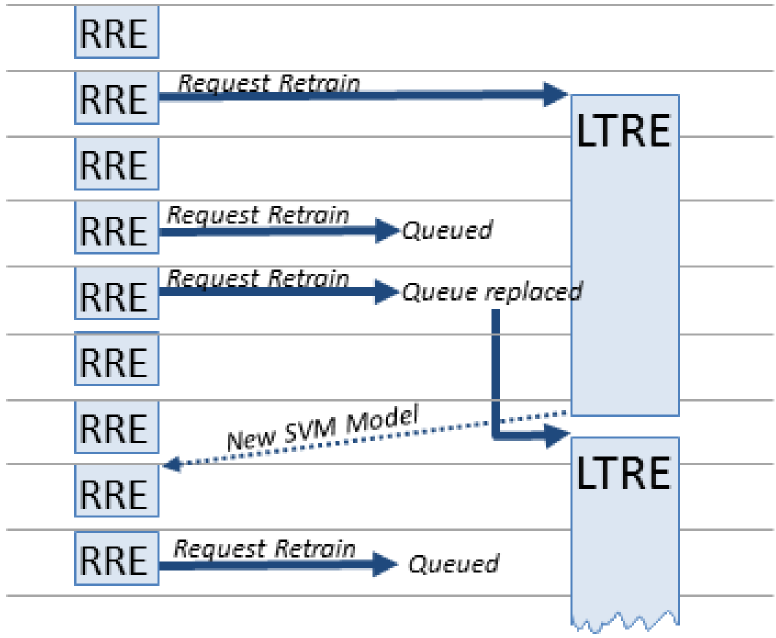


Figure 6. The LTRE operates on the most recent retraining request, and adds all pending observations to the Dataset.

7.1.1 Instance

The RRE uses the current Instance to predict performance of the available strategies. The LTRE uses the set of Instances to train the SVM, and selects a subset of these Instances to become Support Vectors. The key item in an Instance is a vector of values that describe each attribute from the observations, the observables o, the controllables c, and the metric m.

The RRE creates two independent copies of each incoming Instance: one for the RRE to use to predict metrics, and one for the LTRE to add to the Dataset for subsequent LTRE retraining.

The LTRE adds each Instance to its Dataset, and then when the RRE requests a retrain, the LTRE trains on the set of Instances in memory. When the LTRE finishes processing the training data, it makes references to the set of Instances that it selected as support vectors, and passes the references to the RRE (Section 7.3).

7.1.2 Dataset

The Dataset contains SVM-supporting data and data-related functionality. The LTRE and RRE each maintain separate datasets, though both may be referencing the same underlying Instance data, using shared pointers (Section 7.2). Both types of Dataset maintain Instance metadata such as a mapping of Attribute columns to Instance indices and Attribute value min/max/mean.

LTRE Dataset: The LTRE Dataset contains all the data required to build SVMs:

- Metadata about the Attributes (vector of Attribute).
- The set of raw training Instances (vector of Instance). This vector includes Instances that have been processed by the RRE but the LTRE has yet to incorporate into a new model by retraining. This vector is managed as a circular fixed-length buffer; when the vector is full, the LTRE overwrites the oldest Instance.

- The set of normalized instances (vector of Instance). This vector has a one-to-one correspondence with the set of raw Instances. Each raw value is normalized between 0.0 and 1.0.
- A triangular matrix that contains the cached dot products of every pair of normalized Instances. `dotProdMatrix`$(i, j) = \text{dotProd}(I_i, I_j)$, for all pairs of normalized Instances I_i and I_j.
- A triangular matrix of kernel computations. `kernelMatrix`$(i, j) = \phi(I_i, I_j)$ per Equation 1, for all pairs of normalized Instances I_i and I_j.

RRE Dataset: The RRE Dataset contains the current observation, and a snapshot of the Instances that the LTRE chose as SVs during the most recent retraining. Section 7.3 identifies the specific data items the LTRE passes the RRE and Section 7.2 describes the data sharing mechanisms. The RRE Dataset contains:

- The raw and normalized Instance most recently received from the Manager.
- Metadata about the Attributes (vector of Attribute). Most notably, this data includes the maximum and minimum values of the Attribute *at the time of the most recent retraining*, so that the RRE can normalize new observations according to the expected extremes.
- The set of normalized Instances the RRE uses as support vectors when making predictions. These were chosen by the LTRE in the most recent retraining, and the total number of SVs is a subset of the total number of Instances used to build the SVM, i.e., $|SVs| \leq |\text{Instances}|$. They may be shared references to underlying instance data that the LTRE is also referencing concurrently (see Section 7.2).

7.2 LTRE and RRE Data Sharing Mechanisms

To conserve memory and minimize data copying, LTRE and RRE share data where possible. The challenge is that the the LTRE's circular buffer of Instances may overwrite Instances that the RRE is using as SVs. Moreover, the LTRE may need to change the normalized Instances when the range of values for an Attribute increases.

The Strategy Optimizer therefore uses *shared pointers* to manage copies between the LTRE and RRE. The LTRE and RRE can then utilize the same Instance as long as the values do not change. When the LTRE must change a data item it may be sharing with the RRE, it copies the data and alters the copy, and drops its reference to the originally shared item. When the LTRE finishes retraining, it updates the list of SVs, and the RRE then updates its list of shared pointers, and releases its reference to previously shared items.

To ensure that such data items are not leaked when they are dropped, shared pointers use *reference counting*, which maintains a count of references to the item and releases the memory when the count goes to zero. We have the additional requirement that the reference counting be thread-safe since items are shared between the LTRE and RRE threads.

We use `boost::shared_ptr` for this functionality [3] because the 1998 standard for C++ has no native construct for reference-counted pointers to shared data. Regarding leak avoidance, the shared pointer documentation states, *"The shared ptr class template stores a pointer to a dynamically allocated object, typically with a C++ new-expression. The object pointed to is guaranteed to be deleted when the last shared ptr pointing to it is destroyed or reset."* With respect to thread safety of reference counts, *"shared ptr objects offer the same level of thread safety as built-in types. A shared ptr instance can be 'read' (accessed using only const operations) simultaneously by multiple threads. Different shared ptr instances can be 'written to' (accessed using mutable operations such as operator= or reset) simultaneously by multiple threads (even when these instances are copies, and share the same reference count underneath.)"* [3].

Section 6 describes the inter-thread queues we use to pass items between the RRE and LTRE threads. The thread queue handling loops for the LTRE and RRE pull items from the queue and apply them in the current thread.

7.3 Shared Data Items

Section 7.1 describes the data elements that the LTRE and RRE maintain. Section 7.2 outlines the mechanisms used by the LTRE and RRE to share data. This section identifies the data items that we pass between the LTRE and RRE during operation.

When the LTRE builds an SVM initially or on retrain, it must make a copy of the learned model for the RRE. Algorithm 2 shows the pseudocode for this step. It copies a set of model parameters to the RRE to use for predicting performance metric values:

- A timestamp indicating when the SVM was built (64-bit integer). The RRE uses the timestamp as part of error analysis to match predictions to the SVM incarnation in use, and to make sure that it doesn't request a retrain on an unused SVM.
- The list of Support Vectors (vector of shared pointer to Instance). This is the list of normalized Instances selected to be SVs, and then sorted for pipelining.
- Alpha delta values (vector of float). Set of ($\alpha_i^\delta = \alpha_i - \alpha_i^*$), sorted for pipelining, matching one-to-one with the SVs.
- Support Vector self-dot-products (vector of float). This vector is the diagonal of the Instance dot product matrix of each SV to itself
- Attribute metadata (vector of Attribute). See Section 7.1.2.
- Coefficients used by the SVM normalization filter for doing its linear transformation. (3 x int) b, x0 and x1.

7.4 System Operation

The LTRE, upon receiving a new Instance from the Manager, adds the Instance to its Dataset. When the LTRE next retrains, these new Instances are considered as candidate SVs. To place an upper bound on its memory use, the LTRE constrains its instance set to maximum size fixed at compile time. When a new Instance would exceed this maximum, the LTRE deletes the oldest Instance reference in its Dataset, and then adds the new one. The shared pointer implementation (Section 7.2) reduces the reference count to the Instance, and releases the memory if and only if the RRE is not using it. (The deleted Instance reference may be pointing to one of the SVs chosen in the previous retrain and currently in use by the RRE.)

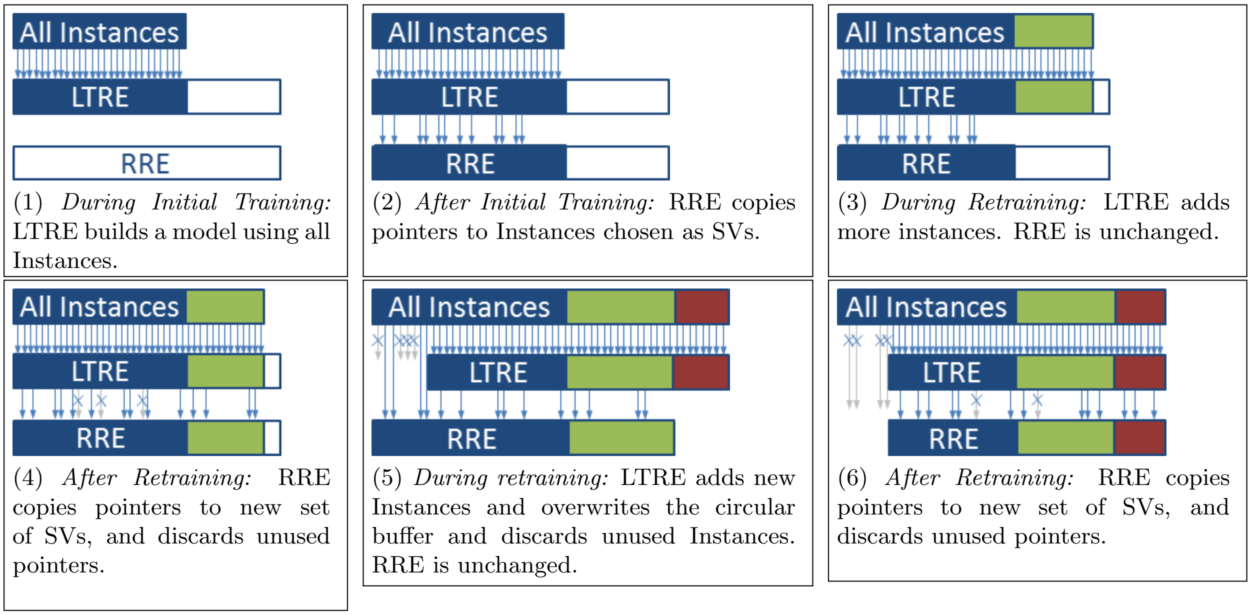
Figure 7 shows how the LTRE and RRE Instance sets can diverge over time due to circular buffering, but then re-align after the LTRE retrains. When the RRE receives the results of the retrain, it releases its references to the old Instances, which releases the underlying Instance data because the reference count has reached zero.

Figure 8 illustrates another cause of divergence between the LTRE and RRE Instance sets: renormalization. In this case, the LTRE releases references to its entire normalized Instance set (while the RRE retains its references) and creates a new set of normalized Instances. The LTRE and RRE Instance sets re-align after the LTRE retrain completes, with the RRE releasing the data associated with the old Instances.

8 Results

We performed our experiments using a suite of 18 different environmental conditions that varied the communications traffic requirements, mobility patterns and interference. There are 160 strategies, generated from legal combinations of 9 control parameters. We present the following results:

- Section 8.1: Compare dynamic Strategy Optimizer to a static system with one prechosen strategy.
- Section 8.2: Compare incremental learning to a system that starts with learned models, but then doesn't change.



(1) *During Initial Training:* LTRE builds a model using all Instances.

(2) *After Initial Training:* RRE copies pointers to Instances chosen as SVs.

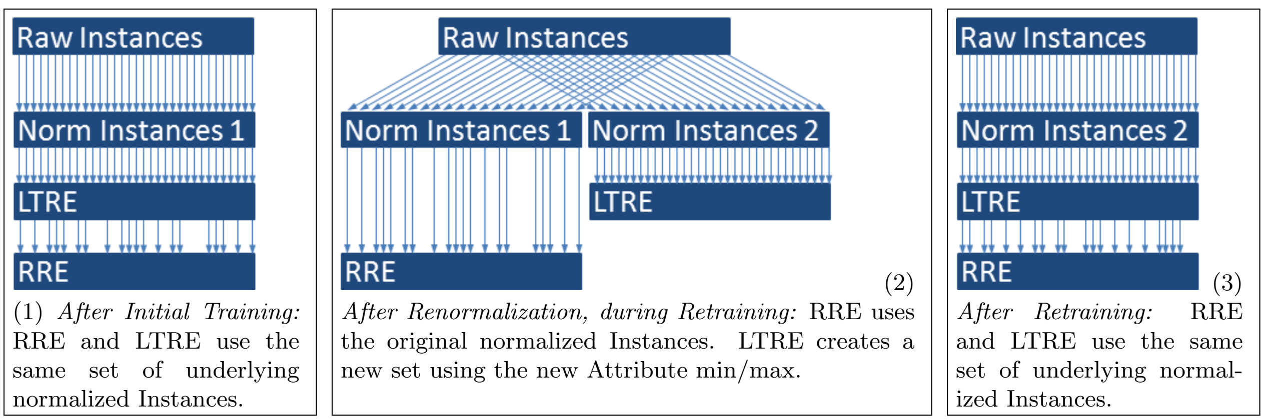
(3) *During Retraining:* LTRE adds more instances. RRE is unchanged.

(4) *After Retraining:* RRE copies pointers to new set of SVs, and discards unused pointers.

(5) *During retraining:* LTRE adds new Instances and overwrites the circular buffer and discards unused Instances. RRE is unchanged.

(6) *After Retraining:* RRE copies pointers to new set of SVs, and discards unused pointers.

Figure 7. At the end of each retraining (states 2, 4 and 6), RRE and LTRE operate on the same set of Instances. During retraining, RRE may be using a different set of pointers, because the LTRE's circular buffer has overwritten previous Instances (state 5), or because the LTRE has renormalized Instances (per Figure 8).



(1) *After Initial Training:* RRE and LTRE use the same set of underlying normalized Instances.

(2) *After Renormalization, during Retraining:* RRE uses the original normalized Instances. LTRE creates a new set using the new Attribute min/max.

(3) *After Retraining:* RRE and LTRE use the same set of underlying normalized Instances.

Figure 8. Renormalizing causes RRE and LTRE to get out of synch during retraining. After retraining, they resynchronize, and discard any unused Instances.

- Section 8.3: A detailed incremental learning example.
- Section 8.4: n-choose-k incremental learning, from no a priori data to full a priori data.
- Section 8.5: Parallel RRE decision making and LTRE incremental learning.

8.1 Comparison to a Static, Manually-Configured System

Compare system with one fixed strategy to system that switchs as conditions change.

The Strategy Optimizer performs significantly better when it can choose what CPs to use for each condition. Figure 9 shows two of the many examples in which our dynamic Strategy Optimizer performs better than a static system.

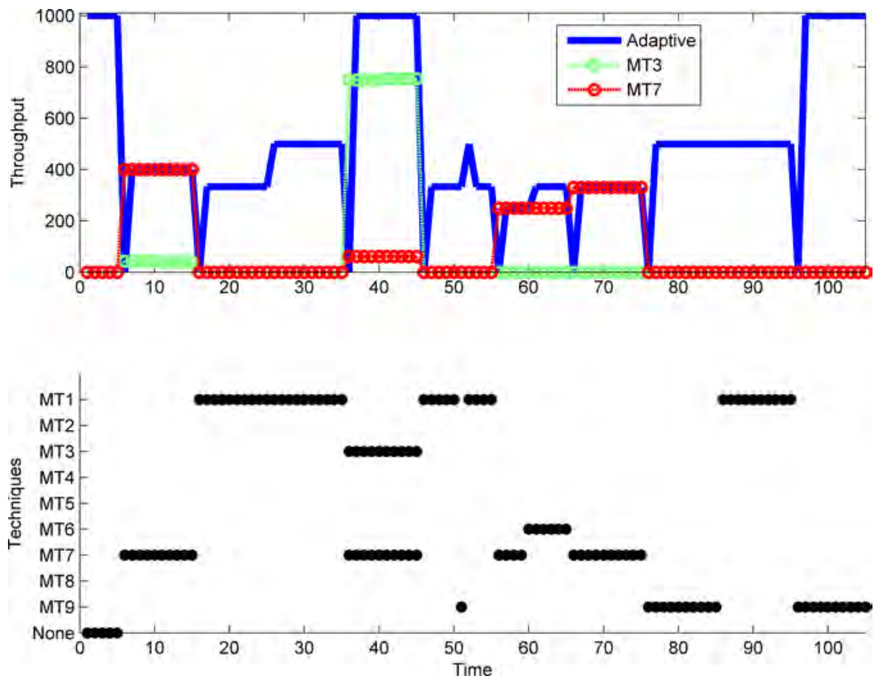


Figure 9. By adaptively choosing and combining different strategies as the environment changes, the Strategy Optimizer performs better than a system that only has one available. *Top graph shows observed performance. Bottom graph shows strategy chosen by the dynamic Strategy Optimizer, as a combination of CPs. X axis is time.*

8.2 Comparison to a Learning-Configured System with no Incremental Retraining

Both systems start with a learned model, but one system doesn't allow on-the-fly retrains.

Incremental Learning is Strategy Optimizer's critical ability to start from limited knowledge and incrementally improve performance during a simulation. Figure 10 shows a trace where the Strategy Optimizer was configured to do no retraining. The simulation encountered four similar RF conditions, but was not trained on any of these. The Strategy Optimizer achieves an average performance of 274.9 across all 40 timestamps, achieving optimal performance from time 21 to 31. On average, however, it only achieves 31% of what it could have achieved for the sequence. In Figure 11, the Strategy Optimizer retrains when prediction error is 100% (either the prediction is double the observation, or the observation is double the prediction). Only three retrainings are required to obtain an average performance of 820.8. Compared to optimal performance, the Strategy Optimizer achieves 94% of what it could have achieved for the sequence.

8.3 Detailed Trace with Incremental Learning

Partial training; RRE detects error, LTRE fully retrains, then RRE finishes.

Figure 12 and Figure 13 show a set of detailed results from an incremental learning trace involving 18 conditions. Condition #4 and all other similar conditions were not present in the training data. The Strategy Optimizer was configured to retrain the SVM models when prediction error was 50%. In the large rectangle, Figure 12 highlights time units 30-40, where the Strategy Optimizer encounters condition #4 for the first time. Figure 13 shows the details of throughput predictions for five of the 160 strategies. At time unit 30, the Strategy Optimizer chooses the strategy that uses CP6 and CP3; unfortunately, performance

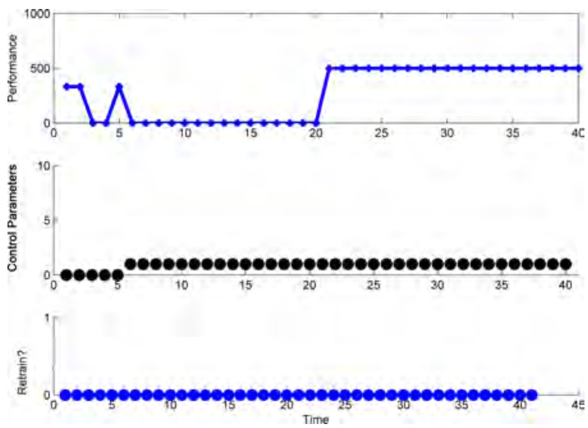


Figure 10. Performance averages 274.9 when the Strategy Optimizer is not allowed to retrain its models when it encounters new conditions. This is 31% of optimal.

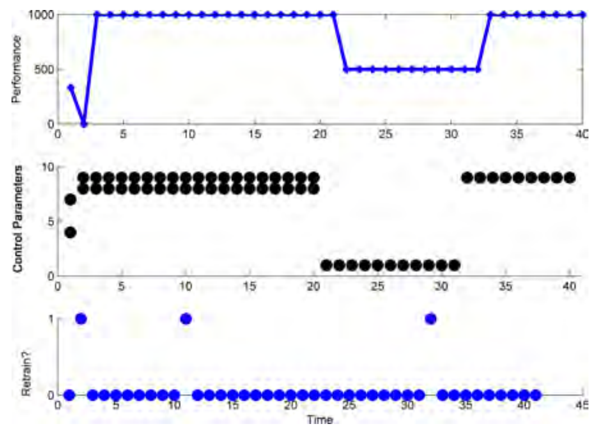


Figure 11. Performance averages 820.8 when the Strategy Optimizer is allowed to retrain in-mission; only three retrainings occur. This is 94% of optimal.

Top graph is observed performance, middle graph is chosen strategy, bottom graph is binary indicator of retraining.

is zero, and prediction error is very high. The Strategy Optimizer retrains its models, but at time unit 31, it makes a similar prediction. The training data very strongly indicates that CP6+3 will perform well: there are no examples in the training data in which it performs poorly. Essentially, this one observation of zero performance is treated as an outlier. Two more examples of zero performance for CP6+3 cause the Strategy Optimizer to switch strategies at time unit 33, and choose CP6+5. This time, only one example causes the Strategy Optimizer to switch. At time unit 34, it chooses CP5, and gets significantly improved performance. If the mission configuration set a minimum performance, the Strategy Optimizer could stop switching strategies at this point. By time unit 38, the Strategy Optimizer has found the optimal strategy, which achieves a Throughput of 750.0.

In the smaller rectangles, Figure 12 shows the long-term effect of the experience with condition #4. For *all* eight of the other similar conditions, the Strategy Optimizer gets excellent performance. Overall performance averages 93.1% of optimal across the simulation. The Strategy Optimizer is able to generalize the characteristics from condition #4 to the other cases. Note that it does not always choose the same strategy, and that the optimal performance is not always 750.0.

8.4 Aggregate Incremental Learning

All combinations of partial learning traces.

Figure 14 shows the results of an n-choose-k ablation study for incremental learning. We tested the Strategy Optimizer against $n = 9$ environmental conditions. In each experiment, the initial training data contains with a different subset k of these conditions, corresponding to the x axis. Each point therefore shows the performance of an experiment in which the Strategy Optimizer performs incremental learning as needed.[1]

The chart shows that performance compared to optimal is 99.3% when the Strategy Optimizer trains on all available training data, and that performance gracefully degrades when the Strategy Optimizer is trained with fewer conditions. When the Strategy Optimizer trains on nothing[2], then performance compared to optimal is 68.4%.

[1]Section 8.3 would correspond to one point on Figure 14.

[2]Training on nothing means that there are only two legal instances, loosely corresponding to when no detected energy yields no expected performance: (NaN, NaN, ..., NaN → NaN) and (0, 0, ..., 0 → 0).

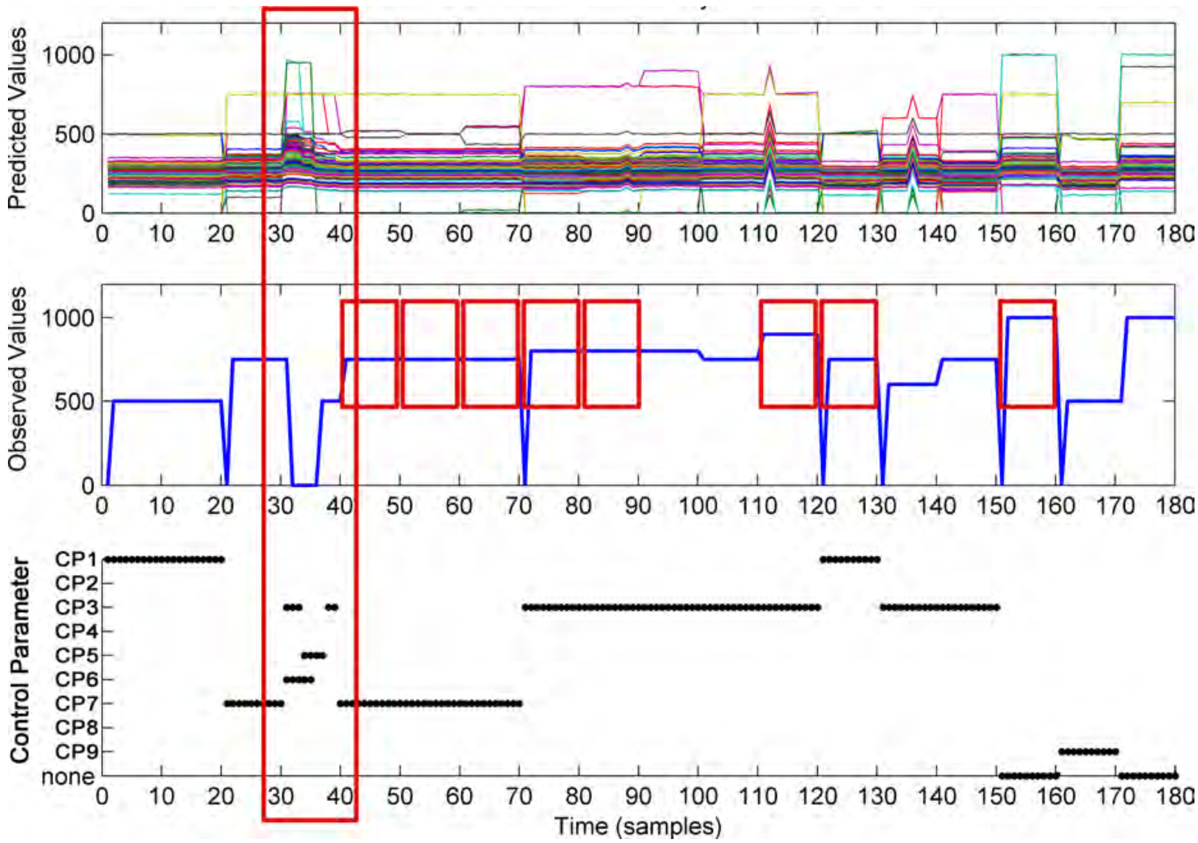


Figure 12. The Strategy Optimizer generalizes from its experience with condition #4 (large overlay rectangle) to perform well on other similar new emitters (smaller rectangles). *Top graph shows predicted values for all 160 strategies; note that optimal performance varies with condition. Middle graph shows observed value at time t. Lower graph shows chosen strategy, as a combination of CPs. X-axis is time.*

8.5 Timing Performance and Threading

Parallel RRE decision making and LTRE incremental learning.

Table 1 shows the timing results for one communications scenario on our Linux server and our two embedded platforms. The RRE compute time is linear in the number of strategies considered, while LTRE is approximately linear in the number of Instances used for training. In addition, when the underlying data is relatively redundant, it is faster to train the model, and fewer Instances are selected as SVs for the RRE.

	Linux	**ARMv7**	**PPC440**
RRE	11ms	303ms	8,201ms
LTRE retrain only	11ms	334ms	8,854ms
LTRE renorm+retrain	64ms	1,292ms	17,385ms

Table 1. Renormalizing the dataset requires floating-point divides, and takes a significant amount of time compared to the RRE and the LTRE when no renormalization is necessary.

Figure 15 and Figure 16 show the impact of sharing the CPU between the two threads. As the LTRE is given more CPU, the LTRE latency drops while the RRE latency increases. Table 2 shows the raw data underlying this plot. The shape of the curve is the same for PPC440. Note that the PPC440 platform uses a Real-Time Operating System, and thus we can directly control the CPU usage.

Time	CP3	CP5	CP6+3	CP6+5	CP7	Observed
30	751.4	751.2	969.3	948.4	749.5	0.0
31	751.3	751.2	953.4	949.3	749.6	0.0
32	751.2	752.0	950.8	948.4	749.0	0.0
33	750.6	750.9	402.8	949.6	749.1	0.0
34	750.1	750.3	376.4	414.9	748.8	500.0
35	750.9	749.1	376.6	414.9	748.9	500.0
36	752.1	501.2	373.4	378.4	750.7	500.0
37	749.4	501.2	372.2	377.5	749.3	500.0
38	502.5	502.9	336.3	375.0	750.1	750.0
39	501.9	501.8	335.6	374.2	749.1	750.0

Figure 13. Only a small number of training exemplars causes the Strategy Optimizer to choose different strategies. *Each "CP" column shows the predicted performance values for that strategy. A box indicates the strategy chosen by the Strategy Optimizer at time t. The "Observed" column shows the actual value from test harness at time $t+1$, and used by the Strategy Optimizer to decide whether to retrain the model.*

ARMv7			
%CPU for LTRE	**RRE latency (ms)**	**# RREs per LTRE**	**LTRE latency (ms)**
0%	303	N/A	N/A
1%	306	110	36,740
5%	319	21	7,014
10%	337	10	3,340
20%	379	5	1,670
25%	404	4	1,336
50%	606	2	668
PPC440			
%CPU for LTRE	**RRE latency (ms)**	**# RREs per LTRE**	**LTRE latency (ms)**
0%	8,200	N/A	N/A
1%	8,283	107	946,242
5%	8,632	21	181,602
10%	9,111	10	86,022
20%	10,250	5	38,232
25%	10,933	4	28,674
50%	16,400	2	9,558

Table 2. When the LTRE has 1% of the CPU, the RRE makes approximately 110 decisions while the LTRE is retraining a new model. As the LTRE is given more CPU, LTRE latency drops while RRE latency increases.

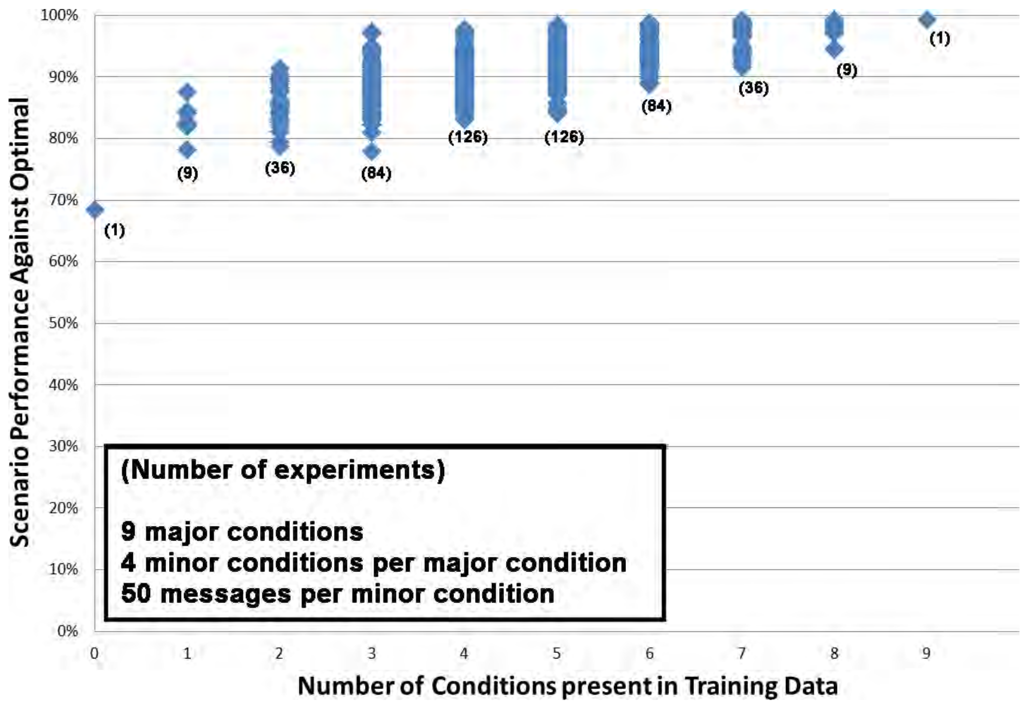


Figure 14. An n-choose-k ablation trial shows that the performance gracefully degrades as the Strategy Optimizer's a priori training data contains fewer conditions. *Each point corresponds to the performance of one experiment; the x-axis is k showing how many conditions were present in the training dataset, and the y-axis is the Performance-against-optimal value. In each column of points, we annotate that there are (X) experiments for 9-choose-k.*

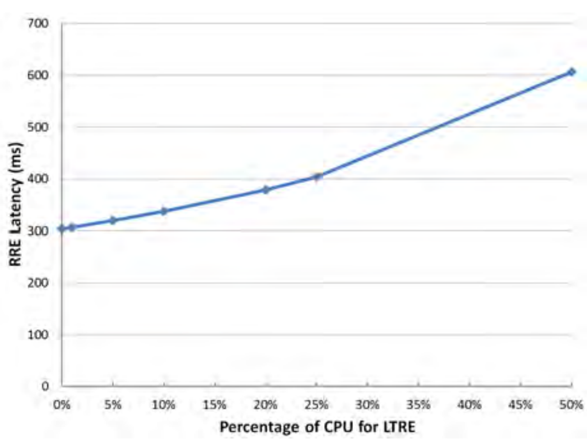


Figure 15. On ARMv7, RRE takes 303ms to compute when has full use of the processor. As the LTRE uses more CPU, the RRE has more latency to produce a decision. Table 2 shows the underlying data.

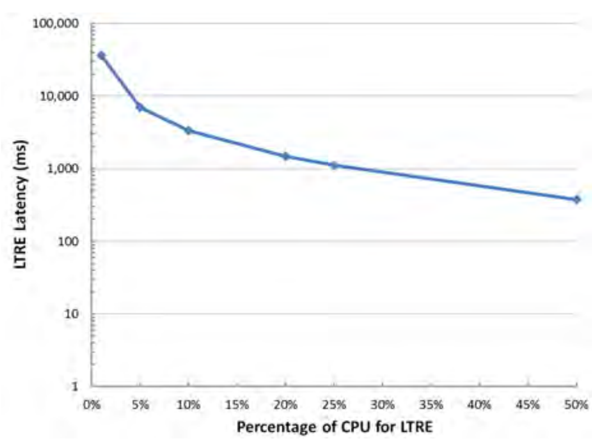


Figure 16. On ARMv7, LTRE takes 334ms to compute when it does not need to renormalize values and it has full use of the processor. As the LTRE uses less CPU, the LTRE has more latency to retrain the model.

9 Conclusion

This paper described our effort to place Machine Learning on an embedded communications system. Our Strategy Optimizer has a rapid decision-making loop that selects a configuration in real-time to optimize performance of the network as conditions change. The Strategy Optimizer also has a slower learning loop that updates the prediction models as the system encounters novel conditions.

In this paper, we focused on the threading model and data interchange mechanisms required for the two loops to operate in parallel. We presented results showing the accuracy of the decisions, and showing how the thread parallelism affects the latency of decisions.

Our multi-threaded structure, and the semantically-agnostic nature of the learner (i.e., it uses no labels), ensure that our approach is relevant for any embedded system that wishes to incorporate Machine Learning to make rapid decisions in a complex environment.

Acknowledgments

This work represents collaboration with BAE Systems (Merrimack, NH). We thank with gratitude John Tranquilli (BAE), Amber Dolan (BAE), Bruce Fette (formerly DARPA), and Vincent Sabio (formerly DARPA).

The views, opinions, and/or findings contained in this article/presentation are those of the author(s)/presenter(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

References

[1] Applied Micro Circuits Corporation. *PPC440 (PPC440G5) Processor User's Manual*, 1.0 edition, August 2011. http://www.amcc.com.

[2] Haifeng Chen, Guofei Jiang, Hui Zhang, and Kenji Yoshihira. Boosting the performance of computing systems through adaptive configuration tuning. In *Proc. ACM Symposium on Applied Computing (SAC)*, pages 1045–1049. (New York, NY: ACM Press), 2009.

[3] Greg Colvin, Beman Dawes, Darin Adler, and Peter Dimov. Boost c++ libraries: shared_ptr class template, 1999. http://www.boost.org/doc/libs/1_56_0/libs/smart_ptr/shared_ptr.htm.

[4] Fangwen Fu and Mihaela van der Schaar. Decomposition principles and online learning in cross-layer optimization for delay-sensitive applications. *IEEE Transactions on Signal Processing*, 58(3):1401–1415, 2010. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5290102.

[5] Karen Zita Haigh. AI technologies for tactical edge networks. In *MobiHoc 2011 Workshop on Tactical Mobile Ad Hoc Networking*, Paris, France, 2011. (New York, NY: ACM Press). Keynote address. http://www.cs.cmu.edu/∼khaigh/papers/Haigh-MobiHoc2011.pdf.

[6] Karen Zita Haigh, Allan M. Mackay, Michael R. Cook, and Li L. Lin. Machine learning for embedded systems: A case study. Technical Report BBN REPORT 8571, BBN Technologies, Cambridge, MA, March 2015. http://www.cs.cmu.edu/∼khaigh/papers/2015-HaighTechReport-Embedded.pdf.

[7] Karen Zita Haigh, Olu Olofinboba, and Choon Yik Tang. Designing an implementable user-oriented objective function for MANETs. In *IEEE International Conference On Networking, Sensing and Control*, pages 693–698, London, U.K., April 2007. (New York, NY: IEEE Press). http://www.cs.cmu.edu/∼khaigh/papers/Haigh07-ICNSC.pdf.

[8] Karen Zita Haigh, Srivatsan Varadarajan, and Choon Yik Tang. Automatic learning-based MANET cross-layer parameter configuration. In *Workshop on Wireless Ad hoc and Sensor Networks (WWASN2006)*, Lisbon, Portugal, 2006. (New York, NY: ACM Press). http://www.cs.cmu.edu/∼khaigh/papers/haigh06a-configuration.pdf.

[9] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: An update. *SIGKDD Explorations*, 11(1), 2009. http://www.cs.waikato.ac.nz/∼eibe/pubs/weka_ update.pdf.

[10] Ovidiu Ivanciuc. Support vector machine software, 2005. http://www. support-vector-machines.org/SVM_soft.html.

[11] V. Kawadia and P. R. Kumar. A cautionary perspective on cross-layer design. *IEEE Wireless Communication*, 12(1):3–11, 2005. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1404568.

[12] Xiaojun Lin, Ness B. Shroff, and R. Srikant. A tutorial on cross-layer optimization in wireless networks. *IEEE Journal on Selected Areas in Communications*, 24(8):1452–1463, 2006. http://www.ifp.illinois.edu/ srikant/Papers/linshrsri06.pdf.

[13] Joseph Mitola III. *Cognitive Radio: An Integrated Agent Architecture for Software Defined Radio*. PhD thesis, Kista, Sweden, 2000. http://www.diva-portal.org/smash/get/diva2:8730/FULLTEXT01.pdf.

[14] D. Montana, T. Hussain, and T. Saxena. Adaptive reconfiguration of data networks using genetic algorithms. In *Proc. Genetic and Evolutionary Computation Conference (GECCO)*, pages 1141–1159. (New York, NY: ACM Press), 2002. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.152.8686.

[15] T. R. Newman, R. Rajbanshi, A. M. Wyglinski, J. B. Evans, and G. J. Minden. Population adaptation for genetic algorithm-based cognitive radios. In *Proc. of ICST Conference on Cognitive Radio Oriented Wireless Networks and Communications (CROWNCOM)*, pages 279–284. (New York, NY: IEEE Press), May 2007. http://www.ittc.ku.edu/∼newman/publications/Conferences/crowncom2007_newman_rajbanshi_wyglinski_evans_minden.pdf.

[16] Timothy Newman, Joseph Evans, and Alexander Wyglinski. Reconfiguration, adaptation and optimization. In Alexander M. Wyglinski, Maziar Nekovee, and Thomas Hou, editors, *Cognitive Radio Communications and Networks: Principles and Practice*. (New York, NY: Elsevier/North Holland), 2009.

[17] John C Platt. Fast training of Support Vector Machines using Sequential Minimal Optimization. In B. Schoelkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*. (Cambridge, MA: MIT Press), 1998. http://research.microsoft.com/en-us/um/people/jplatt/smo-book.pdf.

[18] C. J. Rieser. *Biologically Inspired Cognitive Radio Engine Model Utilizing Distributed Genetic Algorithms for Secure and Robust Wireless Communications and Networking*. PhD thesis, Virginia Tech, Blacksburg, VA, 2004. http://scholar.lib.vt.edu/theses/available/etd-10142004-023653/.

[19] Thomas W. Rondeau. *Application of Artificial Intelligence to Wireless Communications*. PhD thesis, Department of Electrical Engineering, Virginia Polytechnic Institute, Blacksburg, VA, September 2007. http://scholar.lib.vt.edu/theses/available/etd-10052007-081332/.

[20] S. K. Shevade, S. S. Keerthi, C. Bhattacharyya, and K. R. K. Murthy. Improvements to the smo algorithm for svm regression. *IEEE Transactions on Neural Networks*, 11(5), Sept 2000. http://www.keerthis.com/smoreg_ieee_shevade_00.pdf.

[21] Ryan W. Thomas, Daniel H. Friend, Luiz A. DaSilva, and Allen B. MacKenzie. Cognitive networks: Adaptation and learning to achieve end-to-end performance objectives. *IEEE Communications Magazine*, pages 51–57, December 2006. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4050101.

[22] Gregory D. Troxel, Armando Caro, Isidro Castineyra, Nick Goffee, Karen Zita Haigh, Talib Hussain, Vikas Kawadia, Paul G. Rubel, and David Wiggins. Cognitive adaptation for teams in ADROIT. In *IEEE Global Communications Conference*, pages 4868–4872, Washington, DC, November 2007. (New York, NY: IEEE Press). Invited. http://www.cs.cmu.edu/∼khaigh/ papers/troxel07-globecom.pdf.

[23] B. Üstün, W. J. Melssen, and L. M. C. Buydens. Facilitating the application of Support Vector Regression by using a universal Pearson VII function based kernel. *Chemometrics and Intelligent Laboratory Systems*, 81(1):29–40, March 2006. http://www.sciencedirect.com/science/article/pii/S0169743905001474.

[24] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory.* (New York, NY: Springer), 1995.

[25] Wikipedia. Support vector machine, 2014. http://en.wikipedia.org/wiki/ Support_vector_machine, downloaded October 2014.