# Reinforcement Learning Approach to Managing Distributed Data Processing Tasks in Wireless Sensing Networks

Jeshua Bratman, Daniel Fabbri, Andy Zimmerman

## 1 Introduction

As data processing capabilities and techniques continue to rapidly improve across disciplines, the modern engineering community has become increasingly reliant on sensor data to provide an accurate assessment of system behavior and performance [8, 12, 13]. However, because of the high cost of installing and maintaining data cables in large engineered systems, it is often impractical to install sensing transducers in sufficient numbers. As such, wireless sensing systems, which can be deployed at less than one-tenth of the cost of traditional tethered systems, are being explored as a new interface between sensing transducer and data repository.

In addition their low costs, wireless sensing networks (WSNs) have also shown great promise because of their ability to process sensor data locally at each wireless node. Local data processing is especially advantageous when confronted with the huge amounts of data commonly associated with dense networks of sensors: transmitting only processed results (instead of raw sensor data) can drastically reduce the amount of data needing to be communicated. As such, many different architectures have been developed for embedded data processing using wireless sensors [4, 9, 10].

Recently, the WSN community has begun investigating increasingly parallel methods of in-network processing. For example, data aggregation and fusion techniques [11], query processing [14], and explicitly parallel architectures [21] have all been adopted in an attempt to create a framework for the autonomous, in-network processing of large tracts of sensor data. With these advances in mind, one of the key challenges yet to be overcome is that within a wireless environment many system resources (such as processing power and wireless bandwidth) required to perform complex computational tasks are available only in a limited manner. As such, in networks where multiple computational tasks may need to be executed simultaneously, it is important to devise an autonomous method of distributing these scarce system resources across multiple computational objectives. The goal of this study is to apply reinforcement learning (RL) techniques to create offline-trainable online-learning agents which can successfully allocate a WSN's scarce resources across a set of simultaneous computational objectives.

## 2 Problem Statement and Background

Consider a WSN consisting of a set of wireless nodes. Over its lifetime, this WSN may be required to complete some tasks that need to be processed repeatedly (*i.e.*, once every hour), and other tasks that may occur unexpectedly (*i.e.*, are triggered by some external event). Let us assume that each of the incoming tasks can be parallelized across the wireless nodes, and will gain some level of speed advantage in relation to the number of wireless sensing units (WSU) it is allowed to utilize for computation. Conversely, each task will require some amount of wireless communication that will also grow in relation to the number of WSUs used for processing. To complicate matters, each node in a WSN will have a small probability of becoming unresponsive due to sensor or communication failure. If this happens to a node currently working on a computing task, all work that node has processed will be lost (as well as the work of other nodes depending on it).

From an application perspective, there will always be some level of utility associated with the consumption of a WSN's scarce resources (such as time required to finish each job, wireless bandwidth usage, power consumption, etc.). This utility is dependent on the purpose and condition of a WSN, and may change over the network's lifetime. For example, if the WSN is low on battery power, increased utility will be placed on limiting power consumption. In this study, we assume that this utility value is observable to each WSU at any given time.

The problem of optimally allocating processing capabilities across a finite number of computational objectives (often referred to as task scheduling) has been studied for a very long time and from a wide variety of viewpoints [2, 5, 7]. This type of task scheduling problem has even been investigated within the wireless sensing community [6]. However, due to the unknown computational and communication demands of each computing task and the unreliable nature of the wireless environment, it remains a difficult (and yet unresolved) problem.

Recently, Zimmerman et al. [22] published a methodology for market-based task scheduling within ad-hoc WSNs, placing an emphasis on embedded data processing by focusing on optimal communication and data flow, network reliability, and computational speed and efficiency. When evaluating this resource allocation algorithm on a physical network of wireless sensor prototypes, it was found that it allows a set of multiple computational tasks to be completed while generating as much network utility as if an optimal number of sensors had been assigned *a priori* to each computational task at the outset of computation. However, one of the drawbacks of this allocation method is that it is reliant on the WSN having some level of *a priori* knowledge about the tasks that it will be charged with completing. Specifically, each computational task that the market-based resource allocation algorithm assigns processors to must first be analyzed in simulation with respect to its computational and communication requirements.

As such, it would be advantageous if new computational tasks could be exposed to the network without the need for any additional knowledge to be infused into the system. We propose a new RL-based method to allow for both off-line training as well as on-line learning to avoid the drawbacks of the market-based approach. By applying RL techniques to repeating allocation tasks, good allocation methodologies can be learned over time as the resource allocation techniques are repeatedly applied. As such, new computational objectives could be introduced to a given WSN without the need for extensive *a priori* data processing or even reprogramming of the wireless sensing network.

RL techniques have been successfully applied in some similar on-line applications such as job-shop scheduling [20], power management in WSNs [15], and process scheduling on clusters [19]. Although all these problems have significant differences to the task management on data-processing WSNs, many aspects studied will be applicable to the problem. Probably the most similar work was by Shah and Kumar [15], applying RL to resource management on wireless sensing networks. In this work, each node was an RL agent learning to optimize its power consumption, communication, and task management of non-parallel tasks.

## 3 Formulation as Reinforcement Learning Problem

This problem can be formulated as a reinforcement learning problem in several ways. Perhaps the most straightforward is to think of an agent controlling the entire network and assigning nodes to jobs as new jobs appear and complete. Advantages of formulation view include: 1) can be modeled as a markov decision process (MDP), 2) easy to analyze learned policies. Disadvantages include: 1) delayed reward, 2) large state space, 3) difficulty handling new tasks.

We can formulate a WSN with an agent controlling node assignments as a MDP (that we will call $\mathbf{M}$) if we make some simplifying assumptions. $\mathbf{M}$ can be described by the 4-tuple $\mathbf{M} := (S, A, P, R)$ where $S$ is a finite set of states, $A$ is a finite set of actions, $P$ are transition

probabilities, and $R$ is a reward function. One simplification decision we make is to model the WSN as a set of homogeneous nodes. On real networks, this is not the case because some nodes have higher communication costs due to their physical location.

- State in this formulation relates to the status of each computing job seen by the WSN. Although the number of jubs $J$ may change over time, we can upper bound it to create a finite state space. For each job number $i \in \{1, 2, \ldots, J\}$ we will associate five values: task type $k_i$, number of nodes assigned to this task $n_i$, total communication cost accrued thus far $c_i$, amount of time spent computing $t_i$, and the task progress $p_i$ (in reality we can only estimate this). Additionally, the state includes the number of unassigned nodes $u$, and the time of the world $\tau$. The state is therefore the set $S = \{k_1, n_1, c_1, t_1, p_1, \ldots, k_J, n_J, c_J, t_J, p_J, u, \tau\}$.
- Actions are the assignment of a node to a job numbered $A = \{1, \ldots, J\}$.
- Transition probabilities $P$ describe the probability of reaching a new state $s_{t+1} = s'$ from $s_t = s$ when taking action $a$. Mathematically, this is defined as $P^a_{ss'} = Pr\{s_{t+1} = s | s_t = s, a_t = a\}$. These probabilities are very difficult to calculate explicitly, but can be derived empirically through simulation.
- Reward signals can be difficult to choose. In many RL applications, reward is taken to be the designer's utility, but it is known that this may not be the best reward signal. We will base our utility and rewards on the speed and communication costs of executing tasks compared to executing them serially on a single node. One issue with this is that the performance of the entire network is not known until all jobs complete, making such evaluation difficult in a continuous (non-episodic) environment, although still possible.

If we make some simplifying assumptions about the WSN, the Markov property holds: the transition probabilities depend only on the current state and not on past actions if the status of each job $j_i$ is completely specified by $k_i, n_i, c_i, t_i$, and $p_i$.

### 3.1 Learning Algorithms

RL algorithms attempt to find an optimal policy $\pi^* : S \rightarrow A$. If the MDP model is known it is possible to solve for $\pi^*$ through dynamic programming algorithms, such as value iteration, to find the optimum value function $V^* = \max_{a \in A} \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^*(s')]$. However, the transition probabilities in $\mathbf{M}$ are difficult to calculate (and impossible in a real-world scenario) so we must learn $V^*$, or the analogous state-value function $Q^*$, through experience. Bread-and-butter RL techniques include Monte-Carlo (MC) estimation and Temporal Difference (TD) learning. Additionally, there are many non-value-function oriented approaches for estimating $\pi^*$ from experience including policy gradient, model learning, and many others. For this project we only use on-line state-value function learning methods including TD and TD($\lambda$) because they are well suited for on-line learning [16], and we hypothesize that this problem could be solved with these simple algorithms given carefully chosen features and rewards.

We implemented three algorithms: tabular TD(0) Q-learning, tabular TD(0) Sarsa-learning, and TD($\lambda$) Sarsa with linearly approximated Q function.

**Tabular TD(0) Q-learning and Sarsa** These two algorithms are one-step state-value function learning algorithms for control. Q-learning is an off-policy algorithm meaning it updates the state-action value function ($Q$) independent of the current policy with the following update rule:

$$\delta_t = r_t + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)$$
$$Q_{t+1}(s_t, a_t) \leftarrow Q_t(s_t, a_t) + \alpha \delta_t$$

Where $\alpha$ is a learning rate parameter, $\gamma$ reward discount factor, $r_t$ is reward at time $t$ and $\delta$ is the TD error. Unlike Q-learning, Sarsa is on-policy, meaning it updates $Q$ using the policy currently being followed:

$$\delta_t = r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)$$
$$Q_{t+1}(s_t, a_t) \leftarrow Q_t(s_t, a_t) + \alpha\delta_t$$

Both of these algorithms were implemented using hash tables to represent $Q$. Actions were chosen using an $\epsilon$-greedy policy.

**Linear Approximation Sarsa($\lambda$)** TD($\lambda$) is a combination of one-step TD and Monte Carlo learning. Instead of performing only a single step backup like TD(0) or updating all states in $\pi$ like Monte Carlo, TD($\lambda$) instead performs an average $n$-step TD backup based on parameter $\lambda$. This can be implemented efficiently on-line using an eligibility trace function to keep track of how recently states have been visited. TD($\lambda$) backup can be used for on-policy control with an algorithm analogous to Sarsa called Sarsa($\lambda$). The following is the $Q$ function update rule for Sarsa($\lambda$):

$$\delta_t = r_{t+1} + \gamma * Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)$$
$$Q_{t+1}(s, a) \leftarrow Q_t(s_t, a_t) + \alpha\delta_t e_t(s, a),\ \forall s \in S, \forall a \in A$$

where

$$e_t(s, a) = \begin{cases} \sigma_t \lambda e_{t-1}(s, a) + 1 & s = s_t, a = a_t \\ \sigma_t \lambda e_{t-1}(s, a) & otherwise \end{cases} \quad \forall s \in S, \forall a \in A$$

Using a tabular $Q$ function, Sarsa($\lambda$) still has all same problems with generalization as the one-step TD algorithms. To allow for greater generalization between states as well as continuous state variables, $Q$ can be represented as a continuous valued function. Neural networks and simple linear function approximation are widely used in the RL community to much success [1, 16]. We implemented a linear $Q$ function for Sarsa($\lambda$). Consider approximating $Q_a$ with a linear function parametrized by $\vec{\theta}_a$. If $\mathcal{F}_s$ is a set of features extracted from a state $s$, the approximate state-value function is given by $Q(s, a) = \mathcal{F}_s\vec{\theta}_a = \sum_{f \in \mathcal{F}_s} f\theta_{a,f}$. Notice that $\nabla_{\vec{\theta_a}}\ Q(s, a) = \mathcal{F}_s$ so gradient descent is extremely inexpensive and gives us a simple update rule for the weights $\vec{\theta}_a$. The TD error $\delta$ will not change, and we can update $\vec{\theta}_a$ by moving an $\alpha$-distance down the gradient in the eligibility trace $e_a$ for each $\theta_a$.

$$\vec{\theta}_{a,t+1} = \vec{\theta}_{a,t} + \alpha\delta_t\vec{e}_{a,t}$$
$$\vec{e}_{a,t} = \gamma\lambda\vec{e}_{a,t-1} + \nabla_{\vec{\theta_a}}\ Q(s, a) = \gamma\lambda\vec{e}_{a,t-1} + \mathcal{F}_s$$

Notice that we are using a separate function for each action. This is because it is difficult to justify a single $Q$ function linear in the numeric value of the action, although indicator functions could be used. The biggest difficulty in successfully applying this algorithm is extracting linear features $\mathcal{F}$ from a state. For linear functions, features are often chosen to be binary values extracted from the state space through tile coding or other methods [16].

### 3.2 Feature Extraction and Generalization

The main motivation for applying RL to this problem is the possibility of creating a learning agent which can manage tasks effectively in a dynamic system and handle new task types

by improving its own policy online. It is most likely hopeless to blindly use the algorithms described above without putting much thought into feature extraction. This problem of generalization is common in RL.

To demonstrate the problem, imagine that a tabular Q-learning agent has learned a good $Q$ function for observations containing task numbers $1 \dots K$ when it is suddenly given an observation containing task type $K + 1$. This agent will need to learn how to manage this task completely from scratch. Ideally, we would like the agent to first associate new states with the most similar state it has experienced before and then learn from there. There are are several approaches to solving this problem. One possibility is to aggregate similar states into bins [15] or create similarity trees over the space of state-actions [3]. Another option is to to extend the agent's value function by copying verbatim the row of the value function associated with the most similar other task type [18].

Feature extraction can be used both to help generalize and to reduce the state-space from the perspective of the agent. If we select features by intelligently combining aspects of the underlying state, we can often turn an intractable problem into something manageable. For this project, we put some work into feature extraction and also approach generalization through a multi-agent formulation.

## 4 Roughly Simulating a WSN Controlled by Agent

To accurately simulate a WSN, we would need to model processing nodes completing real computing tasks over time. This type of simulation is extremely slow so we decided to model the processing time and communication costs of theoretical tasks mathematically for our initial investigation. Such a mathematical model does not need to represent any particular computational task accurately as long as it roughly represents the dynamics of communication/processing time tradoffs in a WSN.

Our simulator specifies an environment with several parameters: number of nodes $N$, number of jobs $J$, and number of different types of tasks $K$. The simulator then creates $K$ job types. Each job $j$ of type $k$ has $W_k$ units of work that must be completed; let $k_j$ be the job type of job $j$. We use a logarithmic curve $W_k(x) = \nu_k \times ln(x) + 1$ to determine the amount of work $x$ nodes can complete per second for a job of type $k$; a logarithmic curve creates a model so the per node speedup decreases as more nodes are assigned to a job. We use an exponential curve $C_k(x) = \exp\{\frac{x}{\eta_k}\} - \exp\{\frac{1}{\eta_k}\}$ to determine the amount of communication per second for a job type based on the number of nodes assigned; an exponential curve creates a model where the amount of communication increases quickly as more nodes are added to a job. Note that when no nodes are assigned to a job, no work or communication is done and therefore $W_k(0) = C_k(0) = 0$. The coefficients $\nu_k$ and $\eta_k$ are parameters which determine the trade-off between processing time and communication as more nodes are added to a job of type $k$. In order to evaluate a WSN, we use a conversion ratio $r = 0.1$ to express a situation in which each second of computational speedup is ten times more valuable than each byte communicated. It is important to note that the agent does not know the value of the curves, but must learn them over time. Utility achieved by a WSN can then be measured after every job is completed.

$$utility = \frac{total\ work}{time + r \times total\ communication}$$

where $total\ work = \sum_{j=1\dots J} W_k \mid k_j = k$, $time$ is the number of seconds to complete the episode, and $total\ communication = \sum_{j=1\dots J} \sum_{t=1}^{time} C_k(x_t) \mid k_j = k$, is the sum of the bytes communicated over each job's lifetime. If we wish to model a continuing WSN we need to be able to measure the utility over a certain time period. To do this, we can evaluate a single

job completed as it compares to a baseline (serial case) combined with some evaluation of the performance achieved by all jobs finishing over the time period the job was active.

### 4.1 Episodic Simplification

Our experiments simulated the WSN in an episodic manner where all jobs appear at time 0 instead of modeling a long-lived agent in dynamically changing WSN. This is a big simplification, and was meant only as a first step. This decision makes the problem a little less sequential in nature because all that needs to be learned is an optimal assignment of nodes at the beginning of the episode based on the combination of task types. However, it is a good first step for the scope of this project.

For the episodic simulator, an agent handles the assignment of nodes from the beginning of the episode until all jobs are complete. On each episode, the number of jobs stays constant, but the type of each job is random. An agent can then take the action of assigning a node to a job or keeping a node idle. Once all nodes are assigned to work on a job or stay idle, work is completed on a per second basis. Whenever a job is completed, all nodes assigned to the job are freed.

### 4.2 Monolithic Agent

Our first experiments used a monolithic agent controlling the WSN as described earlier. This agent manages the node assignment of the entire network over an episode. Using the whole state $s \in S$ as an observation to this agent would be infeasible, so we put some work into choosing intelligent features from this state. If we provide the agent with a subset of the state information the problem is no longer an MDP and becomes a learning problem in a partially observable MDP (POMDP). TD algorithms provably converge in MDPs but not in POMDPs. However, in practice these algorithms can still perform well with partial observability given good features. As seen in Section 5, experiments using this monolithic model proved unsuccessful despite various methods of feature extraction.

### 4.3 Multi-Agent Bidding

Due to the limited success of the monolithic agent, we decided to take a new approach inspired by Shah and Kumar's work [15] optimizing non-parallel computational tasks and power management in WSNs. Their work modeled each node as a separate Q-learning agent. Due to the parallel nature of our tasks we instead modeled each separate job as an agent. Instead of directly controlling nodes, each agent has a numeric bid that it can change through actions. Nodes assign themselves to the job with the highest bid if any bid was above some threshold. Instead of waiting for all nodes to be assigned to begin computation on a job, each node gets the chance to assign itself once and then the computation proceeds one time step. After any node changes its assignment or finishes its computational work, each agent gets a chance to revise its bid. In the episodic case, all agents are rewarded with the utility earned when every job completes. In the continuous case (which was not implemented for this project), each agent could be rewarded when its job completes. When a new job appears (or at the beginning of a new episode for the episodic case) a new agent is created by copying an agent which had previously been assigned to the same task type. If this is a never-before-seen new task type then we copy a random agent (in reality we may be able to copy an agent from the most similar task if we have such a measure).

For consistency, we will describe the new MDP associated with this formulation. The multi-agent setting does not preclude description as a MDP from the point of view of any one of these agents if we modify the state-space. The other agents can be thought of as part of the system dynamics controlling the transition probabilities. Recall $\mathbf{M} = (S, A, P, R)$.

- State is as before with the addition of a current bid $b_i$ associated with each job (or agent, since there is one agent per job): $S = \{k_1, n_1, c_1, t_1, p_1, b_1, \ldots, k_J, n_J, c_J, t_J, p_J, b_J, u, \tau\}$.
- Actions are: increase bid, decrease bid, do nothing.

Advantages of this model are many: 1) the policy for each individual agent is much simpler as the agent needs only to learn how to change its bid given any state, 2) selecting intelligent features for a single agent is more intuitive, 3) generalization between task types is no longer a problem since each job is its own agent.

Just as in the monolithic agent case, we will introduce partial observability by extracting features in order to simplify the problem for learning agents. For this task we will supply the agent with the following feature observations:

- agent's current bid
- relation of this bid to the other agents' bids
- whether current number of nodes is less-than, the same, or greater than the best previous number for the current combination of tasks.

We found that this last observation is especially valuable and could possibly be used to improve the monolithic agent (we did not have time to test this hypothesis during this study).

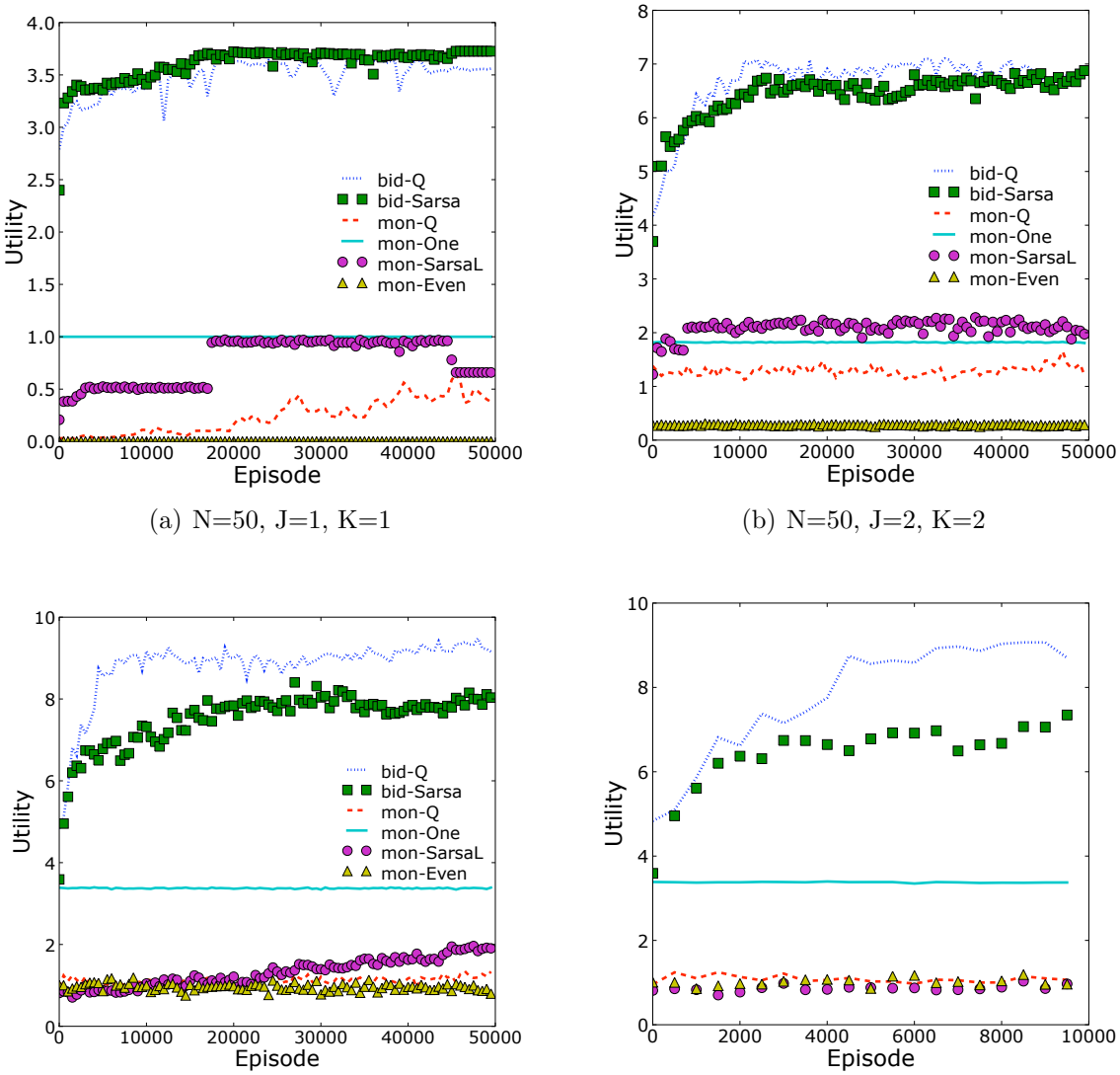
## 5 Results

We first evaluated our RL algorithms on a simulated WSN with 50 Nodes ($N = 50$). Experiments were performed with both the monolithic agent and multi-agent bidding formulation on the same problems. Figure 1 shows 6 different agents working on the same set of $J$ tasks over 50,000 episodes (each plot is averaged over 5 trials). The difficulty of each task type $K$ is generated randomly using the $\nu_k$ and $\eta_k$ parameters and then held constant for each episode and trial. All tasks are such that the optimal number of nodes to assign is somewhere in the range $[1, 50]$, but each task is significantly different. The agents used are as follows: 1) mon-Even is a fixed policy of assigning an even number of nodes to each task. This is never a good policy because too much communication cost is accrued. 2) mon-One assigns a single node to every task. This is the baseline performance. 3) mon-Q is Q-learning monolithic agent. 4) mon-SarsaL is Sarsa($\lambda$) monolithic agent where the state space values are quantized into groups. 5) bid-Sarsa is one-step Sarsa bidding agents. 6) bid-Q is Q-learning bidding agents. We did not have a chance to run Sarsa($\lambda$) using the bidding simulator

As you can see from agents mon-SarsaL, and mon-Q, the monolithic agent was not successful. This result is not overly surprising. Despite some efforts at feature extraction, the state space was still too large. We hypothesize that better choices of features could make this monolithic agent concept feasible. On the other hand, the node-bidding agents perform very well. Both Sarsa and Q-Learning dramatically improve network utility and begin performing better than the baseline within a few episodes. This result is very encouraging because for this to be viable for a real WSN, the agents would need to be able to improve quickly to be worthwhile. We find that these agents learn to bid based on their features to reach the near-optimal number of nodes for each task type. The last plot in figure 1 shows the results of bidding Q-learning and Sarsa for just the first 10,000 episodes to demonstrate their rapid learning.

## 6 Verification in a Simulated WSN Environment

In order to validate the RL techniques developed in this study using an actual WSN framework, a WSN simulator is developed based on the *Narada* wireless sensor (Figure 2a), which was designed at the University of Michigan [17]. When run, this simulator first creates a set of $J$ computational tasks. In order to simulate a suite of computationally diverse tasks, each




(a) N=50, J=1, K=1

(b) N=50, J=2, K=2

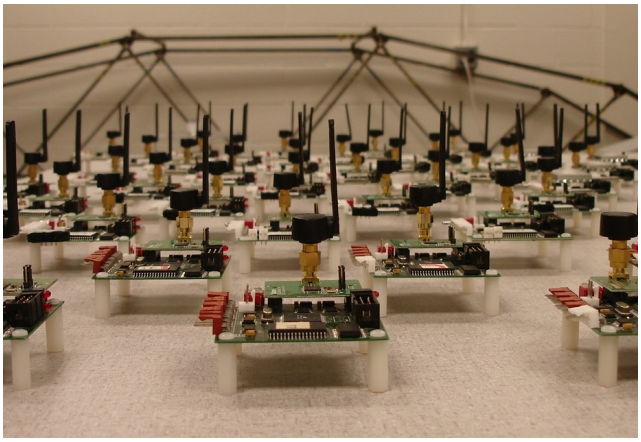
(c) N=50, J=4, K=2

(d) N=50, J=4, K=2

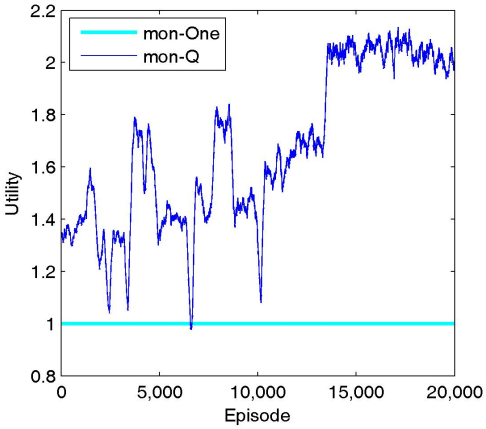
Figure 1: (a)-(c) Experiments for various tasks for 50,000 episodes averaged over 5 trials. Bidding agents are the only ones that consistently do better than the baseline. (d) First 10,000 episodes from figure (c). All agents had parameters chosen heuristically: $\gamma = 0.1$ and $\epsilon = 0.1$ which was set to 0 in the last 5,000 epsides. For Q-learning and Sarsa, $\alpha = 0.1$. For Sarsa($\lambda$), $\lambda = 0.6$ and $\alpha = 1/(10|\mathcal{F}|)$

of the $J$ tasks generated by the simulator is an instantiation of the $n$-Queens combinatorial optimization problem, where the complexity of each task varies as the value of $n$ changes. Once a set of $J$ tasks have been created, the simulator then utilizes the agent-based RL architecture developed herein to assign available WSUs to each computing task and reproduces the computational and communication demand that an actual network of *Narada* sensors would experience under the same set of tasks.

Because of the significant computational demand required to simulate a large network of sensors (it takes approximately 14 hours to simulate 10,000 episodes of the assignment of 50 WSUs to four tasks (25,50,75, and 100-Queens)), we were only able to evaluate a few select



(a) *Narada* Wireless Sensing Network




(b) N=50, J=1, K=1 (25-Queens)

Figure 2: Verification in a simulated WSN environment

test cases in this simulated environment. Nevertheless, early results have been encouraging. As seen in Figure 2b, the mon-Q framework is able to learn, within 20,000 episodes, to outperform the mon-One (baseline) case when assigning 50 WSUs to the 25-Queens problem given a utility function where each CPU second saved is 50 times more valuable than each byte communicated.

## 7 Conclusion

Over the course of our experiments we discovered that our initial formulation of WSN task management as a reinforcement learning problem had flaws, but by re-thinking this formulation we have we have demonstrated, at least in simulation, that this approach shows promise for real WSNs. Large state spaces present in such WSNs poses intimidating hurdles that can only be overcome by intelligent feature extraction. Future work includes more experimentation on the *Narada* WSN simulator, further study of feature extraction and selection, deeper investigation into choosing reward functions for the agents, learning on a more dynamic and realistic environment, and experimentation with more sophisticated RL algorithms including internal state and options.

## 8 Individual Efforts

All group members collaborated closely on this project and the report. Jeshua Bratman investigated related RL literature and implemented both the learning algorithms and the agent bidding simulator. Dan Fabbri designed mathematical processing/communication tradeoff model, and implemented the monolithic agent simulator. Andy Zimmerman developed the *Narada* simulation environment for WSN validation, and investigated the related WSN literature.

## References

[1] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-dynamic programming.* Athena Scientific, Belmont, MA, 1996.

[2] R. Cocchi, S. Shenker, D. Estrin, and L. Zhang. Pricing in computer networks: motivation, formulation, and example. *IEEE/ACM Transactions on Networking*, 1(6):614–627, 1993.

[3] Sertan Girgin, Faruk Polat, and Reda Alhajj. State similarity based approach for improving performance in rl. In *IJCAI'07: Proceedings of the 20th international joint conference on*

*Artifical intelligence*, pages 817–822, 2007.

[4] Y. Hashimoto, A. Masuda, and A. Sone. Prototype of sensor network with embedded local data processing. *Smart Structures and Materials Conference, San Diego, CA*, pages 245–252, 2005.

[5] Y. C. Ho, L. Servi, and R. Suri. A class of center-free resource allocation algorithms. *Large Scale Systems*, 1(1):51–62, 1980.

[6] J. Jin, W. H. Wang, and M. Palaniswami. Application-oriented flow control for wireless sensor networks. *3rd International Conference on Networking and Services, Athens, Greece*, pages 423–429, 2007.

[7] Y. F. Kao and J. H. Huang. Price-based resource allocation for wireless ad hoc networks with multi-rate capability and energy constraints. *Computer Communications*, 31:3613–3624, 2008.

[8] T. H. Loutas, J. Kalaitzoglou, G. Sotiriades, and V. Kostopoulous. A novel approach for continuous acoustic emission monitoring on rotating machinery without the use of slip ring. *Journal of Vibration and Acoustics (ASME)*, 130(6):1–6, 2008.

[9] J. P. Lynch. *Decentralization of wireless monitoring and control techniques for smart civil structures.* Ph.D. dissertation, John A. Blume Earthquake Engineering Center, Stanford University, Stanford, CA, 2002.

[10] J. P. Lynch, A. Sundararajan, K. H. Law, A. S. Kiremidjian, and E. Carryer. Embedding damage detection algorithms in a wireless sensing unit for operational power efficiency. *Smart Materials and Structures*, 13:800–810, 2004.

[11] T. Nagayama, B. F. Spencer, G. A. Agha, and K. A. Mechitov. *Model-based data aggregation for structural monitoring employing smart sensors.* 3rd International Conference on Networked Sensing Systems, Chicago, IL, 2006.

[12] Y. Q. Ni, H. F. Zhou, K. C. Chan, and J. M. Ko. Modal flexibility analysis of cable-stayed ting kau bridge for damage identification. *Computer-Aided Civil and Infrastructure Engineering*, 23(3):223–236, 2008.

[13] J. Parjajka and G. Bloschl. The value of modis snow cover data in validating and calibrating conceptual hydrologic models. *Journal of Hydrology*, 358(3-4):240–258, 2008.

[14] R. Rosemark and W. C. Lee. Decentralizing query processing in sensor networks. *2nd International Conference on Mobile and Ubiquitous Systems, San Diego, California*, pages 270–280, 2005.

[15] Kunal Shah and Mohan Kumar. Distributed independent reinforcement learning (dirl) approach to resource management in wireless sensor networks. *IEEE International Conference on Mobile Adhoc and Sensor Systems Conference*, 0:1–9, 2007.

[16] Richard S. Sutton and Andrew G. Barto. Reinforcement learning i: Introduction, 1998.

[17] R.A. Swartz, D. Jun, J.P. Lynch, Y. Wang, D. Shi, and M. Flynn. Design of a wireless sensor for scalable distributed in-network computation in a structural health monitoring system. In *IWSHM'05: Proceedings of the International Workshop on Structural Health Monitoring*, 2005.

[18] Matthew E. Taylor and Peter Stone. Behavior transfer for value-function-based reinforcement learning. In *The Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 53–59, July 2005.

[19] David Vengerov. A reinforcement learning framework for utility-based scheduling in resource-constrained systems. *Future Gener. Comput. Syst.*, 25(7):728–736, 2009.

[20] Wei Zhang and Thomas G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *In Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1114–1120, 1995.

[21] A. T. Zimmerman and J. P. Lynch. A parallel simulated annealing architecture for model updating in wireless sensor networks. *IEEE Sensors Journal*, 9(11):1503–1510, 2009.

[22] A. T. Zimmerman, J. P. Lynch, and F. T. Ferrese. Market-based computational task assignment within autonomous wireless sensor networks. *IEEE International Conference on Electro/Information Technology, Windsor, Canada*, pages 23–28, 2009.