Computer Science
Technical Report

Robust Reinforcement Learning Control with Static and Dynamic Stability[a]

R. Matthew Kretchmar, Peter M. Young, Charles W. Anderson,
Douglas C. Hittle, Michael L. Anderson, Christopher C. Delnero

Colorado State University

May 30, 2001

Technical Report CS-00-102

[a]From a thesis submitted to the Academic Faculty of Colorado State University in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science. This work was partially supported by the National Science Foundation through grants CMS-9804757 and 9732986.

Computer Science Department
Colorado State University
Fort Collins, CO 80523-1873

Phone: (970) 491-5792 Fax: (970) 491-2466
WWW: http://www.cs.colostate.edu

Robust Reinforcement Learning Control with Static and Dynamic Stability*

R. Matthew Kretchmar, Peter M. Young, Charles W. Anderson,
Douglas C. Hittle, Michael L. Anderson, Christopher C. Delnero

Colorado State University

May 30, 2001

Abstract

Robust control theory is used to design stable controllers in the presence of uncertainties. By replacing nonlinear and time-varying aspects of a neural network with uncertainties, a robust reinforcement learning procedure results that is guaranteed to remain stable even as the neural network is being trained. The behavior of this procedure is demonstrated and analyzed on two simple control tasks. For one task, reinforcement learning with and without robust constraints results in the same control performance, but at intermediate stages the system without robust constraints goes through a period of unstable behavior that is avoided when the robust constraints are included.

*From a thesis submitted to the Academic Faculty of Colorado State University in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science. This work was partially supported by the National Science Foundation through grants CMS-9804757 and 9732986.

1 Introduction

The design of a controller is based on a mathematical model that captures as much as possible all that is known about the plant to be controlled and that is representable in the chosen mathematical framework. The objective is not to design the best controller for the plant model, but for the real plant. Robust control theory achieves this goal by including in the model a set of uncertainties. When specifying the model in a Linear-Time-Invariant (LTI) framework, the nominal model of the system is LTI and "uncertainties" are added with gains that are guaranteed to bound the true gains of unknown, or known and nonlinear, parts of the plant. Robust control techniques are applied to the plant model augmented with uncertainties and candidate controllers to analyze the stability of the true system. This is a significant advance in practical control, but designing a controller that remains stable in the presence of uncertainties limits the aggressiveness of the resulting controller, resulting in suboptimal control performance.

In this article, we describe an approach for combining robust control techniques with a reinforcement learning algorithm to improve the performance of a robust controller while maintaining the guarantee of stability. Reinforcement learning is a class of algorithms for solving multi-step, sequential decision problems by finding a policy for choosing sequences of actions that optimize the sum of some performance criterion over time [27]. They avoid the unrealistic assumption of known state-transition probabilities that limits the practicality of dynamic programming techniques. Instead, reinforcement learning algorithms adapt by interacting with the plant itself, taking each state, action, and new state observation as a sample from the unknown state transition probability distribution.

A framework must be established with enough flexibility to allow the reinforcement learning controller to adapt to a good control strategy. This flexibility implies that there are numerous undesirable control strategies also available to the learning controller; the engineer must be willing to allow the controller to temporarily assume many of these poorer control strategies as it searches for the better ones. However, many of the undesirable strategies may produce instabilities. Thus, our objectives for the approach described here are twofold. The main objective that must always be satisfied is stable behavior. The second objective is to add a reinforcement learning component to the controller to optimize the controller behavior on the true plant, while never violating the main objective.

While the vast majority of controllers are LTI due to the tractable mathematics and extensive body of LTI research, a non-LTI controller is often able to achieve greater performance than an LTI controller, because it is not saddled with the limitations of LTI. Two classes of non-LTI controllers are particularly useful for control: nonlinear controllers and adaptive controllers. However, nonlinear and adaptive controllers are difficult, and often impossible, to study analytically. Thus, the guarantee of stable control inherent in LTI designs is sacrificed for non-LTI controllers.

Neural networks as controllers, or neuro-controllers, constitute much of the recent non-LTI control research. Because neural networks are both nonlinear and adaptive, they can realize superior control compared to LTI. However, most neuro-controllers are static in that they respond only to current input, so they may not offer any improvement over the dynamic nature of LTI designs. Little work has appeared on dynamic neuro-controllers. Stability analysis of neuro-controllers has been very limited, which greatly limits their use in real applications.

The stability issue for systems with neuro-controllers encompasses two aspects. *Static stability* is achieved when the system is proven stable provided that the neural network weights are constant. *Dynamic stability* implies that the system is stable even while the network weights are changing. Dynamic stability is required for networks which learn on-line in that it requires the system to be stable regardless of the sequence of weight values learned by the algorithm.

Our approach in designing a stable neuro-control scheme is to combine robust control techniques with reinforcement learning algorithms for training neural networks. We draw upon the reinforcement learning research literature to construct a learning algorithm and a neural network architecture that are suitable for application in a broad category of control tasks. Robust control provides the tools we require to guarantee the stability of the system.

Figure 1 depicts the high-level architecture of the proposed system. In this paper, we focus on tracking tasks. Let r be the reference input to be tracked by the plant output, y. The tracking error is the difference between the reference signal and the plant output: $e = r - y$. A *nominal* controller, K, operates on the tracking error to produce a control signal c. A neural network is added in parallel to the nominal controller which also acts on the tracking error to produce a control signal, which we call an action, a. The nominal control output and the neural network output are summed to arrive at the overall control signal: $u = c + a$. Again, the goal of the controller(s) is twofold. The first goal is to guarantee system stability. The second goal is to produce the control signals to cause the plant to closely track the reference input over time. Specifically, this latter performance goal is to learn a control function to minimize the mean squared tracking error over time. Note that the neural network does not replace the nominal controller; this approach has the advantage that the control performance of the system is improved during the learning process. If the neuro-controller were operating alone, its initial control performance would most likely be extremely poor. The neural network would

require substantial training time to return to the level of performance of the nominal LTI controller (and in fact may not even get there since it is a static, albeit nonlinear, controller). Instead, the neuro-controller starts with the performance of the nominal (dynamic LTI) controller and adds small adjustments to the control signal in an attempt to further improve control performance.

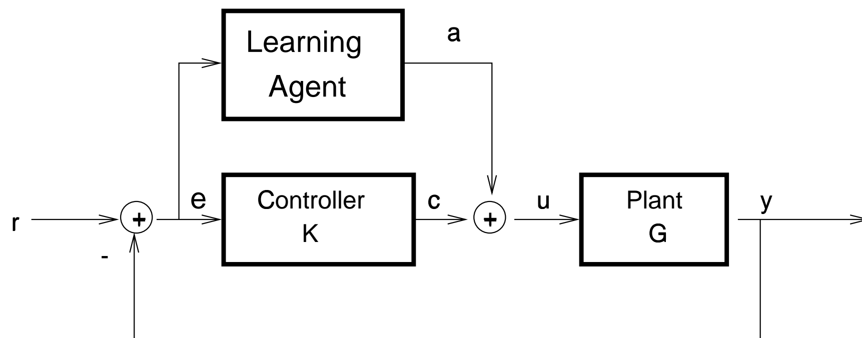

Figure 1: Nominal controller with neural network in parallel.

To solve the static stability problem, we must ensure that the neural network with a fixed set of weights implements a stable control scheme. Since exact stability analysis of the nonlinear neural network is intractable, we need to extract the LTI components from the neural network and represent the remaining parts as uncertainties. To accomplish this, we treat the nonlinear hidden units of the neural network as sector-bounded, nonlinear uncertainties. We use Integral Quadratic Constraint (IQC) analysis [15] to determine the stability of the system consisting of the plant, the nominal controller, and the neural network with given weight values. Others have analyzed the stability of neuro-controllers using other approaches. The most significant of these other static stability solutions is the NLq research of Suykens and DeMoor [28]. Our approach is similar in the treatment of the nonlinearity of the neural network, but we differ in how we arrive at the stability guarantees. Our approach is also graphical and thus amenable to inspection and change-and-test scenarios.

Along with the nonlinearity, the other powerful feature of using a neural network is its adaptability. In order to accommodate this adaptability, we must solve the dynamic stability problem—the system must be proven stable while the neural network is learning. As we did in the static stability case, we use a sector-bounded uncertainty to cover the neural network's nonlinear hidden layer. Additionally, we add uncertainty in the form of a slowly time-varying scalar to cover weight changes during learning. Again, we apply IQC-analysis to determine whether the network (with the weight uncertainty) forms a stable controller.

The most significant contribution of this article is a solution to the dynamic stability problem. We extend the techniques of robust control to transform the network weight learning problem into one of network weight uncertainty. With this key realization, a straightforward computation guarantees the stability of the network during training.

An additional contribution is the specific architecture amenable to the reinforcement learning control situation. The design of learning agents is the focus of much reinforcement learning literature. We build upon the early work of actor-critic designs as well as more recent designs involving Q-learning. Our dual network design features a computable policy (this is not available in Q-learning) which is necessary for robust analysis. The architecture also utilizes a discrete value function to mitigate difficulties specific to training in control situations.

The remainder of this article describes our approach and demonstrates its use on two simple control problems. Section 2 provides an overview of reinforcement learning and the actor-critic architecture. Section 3 summarizes our use of IQC to analyze the static and dynamic stability of a system with a neuro-controller. Section 4 describes the method and results of applying our robust reinforcement learning approach to two simple tracking tasks. We find that the stability constraints are necessary for the second task; a non-robust version of reinforcement learning converges on the same control behavior as the robust reinforcement learning algorithm, but at intermediate steps before convergence, unstable behavior appears. In Section 4 we summarize our conclusions and discuss current and future work.

2 Reinforcement Learning

2.1 Roots and Successes of Reinforcement Learning

In this section we review the most significant contributions of reinforcement learning with emphasis on those directly contributing to our work in robust neuro-control. Sutton and Barto's text, *Reinforcement Learning: An Introduction* presents a detailed historical account of reinforcement learning and its application to control [27]. From a historical perspective, Sutton and Barto identify two key research trends that led to the development of reinforcement learning: trial and error learning from psychology and dynamic programming methods from mathematics.

It is no surprise that the early researchers in reinforcement learning were motivated by observing animals (and people) *learning* to solve complicated tasks. Along these lines, a few psychologists are noted for developing formal theories of this "trial and error" learning. These theories served as spring boards for developing algorithmic and mathematical representations of artificial agents learning by the same means. Notably, Roger Thorndike's work in *operant conditioning* identified an animal's ability to form associations between an action and a positive or negative reward that follows [30]. The experimental results of many pioneer researchers helped to strengthen Thorndike's theories. Notably, the work of Skinner and Pavlov demonstrates "reinforcement learning" in action via experiments on rats and dogs respectively [23, 19].

The other historical trend in reinforcement learning arises from the "optimal control" work performed in the early 1950s. By "optimal control", we refer to the mathematical optimization of reinforcement signals. Today, this work falls into the category of dynamic programming and should not be confused with the optimal control techniques of modern control theory. Bellman [7] is credited with developing the techniques of dynamic programming to solve a class of deterministic "control problems" via a search procedure. By extending the work in dynamic programming to stochastic problems, Bellman and others formulated the early work in Markov decision processes.

Successful demonstrations of reinforcement learning applications on difficult and diverse control problems include the following. Crites and Barto successfully applied reinforcement learning to control elevator dispatching in large scale office buildings [8]. Their controller demonstrates better service performance than state-of-the-art, elevator-dispatching controllers. To further emphasize the wide range of reinforcement learning control, Singh and Bertsekas have outcompeted commercial controllers for cellular telephone channel assignment [22]. Our initial application to HVAC control shows promising results [1]. An earlier paper by Barto, Bradtke and Singh discussed theoretical similarities between reinforcement learning and optimal control; their paper used a race car example for demonstration [6]. Early applications of reinforcement learning include world-class checker players [21] and backgammon players [29]. Anderson lists several other applications which have emerged as benchmarks for reinforcement learning empirical studies [2].

2.2 Q-Learning and SARSA

Barto and others combined these two historical approaches in the field of reinforcement learning. The reinforcement learning agent interacts with an environment by observing states, s, and selecting actions, a. After each moment of interaction (observing s and choosing a), the agent receives a feedback signal, or reinforcement signal, R, from the environment. This is much like the trial-and-error approach from animal learning and psychology. The goal of reinforcement learning is to devise a control algorithm, called a *policy*, that selects optimal actions for each observed state. By optimal, we mean those actions which produce the highest reinforcements not only for the immediate action, but also for future actions not yet selected. The mathematical optimization techniques of Bellman are integrated into the reinforcement learning algorithm to arrive at a policy with optimal actions.

A key concept in reinforcement learning is the formation of the value function. The value function is the expected sum of future reinforcement signals that the agent receives and is associated with each state in the environment. Thus $V(s)$ is the value of starting in state s and selecting optimal actions in the future; $V(s)$ is the sum of reinforcement signals, R, that the agent receives from the environment.

A significant advance in the field of reinforcement learning is the Q-learning algorithm of Chris Watkins [31]. Watkins demonstrates how to associate the value function of the reinforcement learner with both the state and action of the system. With this key step, the value function can now be used to directly implement a policy without a model of the environment dynamics. His Q-learning approach neatly ties the theory into an algorithm which is both easy to implement and demonstrates excellent empirical results. Barto, et al., [5], describe this and other reinforcement learning algorithms as constituting a general Monte Carlo approach to dynamic programming for solving optimal control problems with Markov transition probability distributions [5].

To define the Q-learning algorithm, we start by representing a system to be controlled as consisting of a discrete state space, S, and a finite set of actions, A, that can be taken in all states. A *policy* is defined by the probability, $\pi(s_t, a)$, that action a will be taken in state s_t at time step t. Let the reinforcement resulting from applying action a_t while the system is in state s_t be $R(s_t, a_t)$. $Q_\pi(s_t, a_t)$ is the value function given state s_t and action a_t, assuming policy π governs action selection from then on. Thus, the desired value of $Q_\pi(s_t, a_t)$ is

$$Q_\pi(s_t, a_t) = \mathrm{E}_\pi \left\{ \sum_{k=0}^{T} \gamma^k R(s_{t+k}, a_{t+k}) \right\},$$

where γ is a discount factor between 0 and 1 that weights reinforcement received sooner more heavily than reinforcement received later. This expression can be rewritten as an immediate reinforcement plus a sum of future reinforcement:

$$\begin{aligned} Q_\pi(s_t, a_t) &= \mathrm{E}_\pi \left\{ R(s_t, a_t) + \sum_{k=1}^{T} \gamma^k R(s_{t+k}, a_{t+k}) \right\}, \\ &= \mathrm{E}_\pi \left\{ R(s_t, a_t) + \gamma \sum_{k=0}^{T-1} \gamma^k R(s_{t+k+1}, a_{t+k+1}) \right\}. \end{aligned}$$

In dynamic programming, policy evaluation is conducted by iteratively updating the value function until it converges on the desired sum. By substituting the estimated value function for the sum in the above equation, the iterative policy evaluation method from dynamic programming results in the following update to the current estimate of the value function:

$$\Delta Q_\pi(s_t, a_t) = \mathrm{E}_\pi \left\{ R(s_t, a_t) + \gamma Q_\pi(s_{t+1}, a_{t+1}) \right\} - Q_\pi(s_t, a_t),$$

where the expectation is taken over possible next states, s_{t+1}, given that the current state is s_t and action a_t was taken. This expectation requires a model of state transition probabilities. If such a model does not exist, a Monte Carlo approach can be used in which the expectation is replaced by a single sample and the value function is updated by a fraction of the difference:

$$\Delta Q_\pi(s_t, a_t) = \alpha_t \left[R(s_t, a_t) + \gamma Q_\pi(s_{t+1}, a_{t+1}) - Q_\pi(s_t, a_t) \right],$$

where $0 \le \alpha_t \le 1$. The term within brackets is often referred to as a *temporal-difference* error [25].

One dynamic programming algorithm for improving the action-selection policy is called value iteration. This method combines steps of policy evaluation with policy improvement. Assuming we want to minimize total reinforcement, which would be the case if the reinforcement is related to tracking error as it is in the experiments described later, the Monte Carlo version of value iteration for the Q function is

$$\Delta Q_\pi(s_t, a_t) = \alpha_t \left[R(s_t, a_t) + \gamma \min_{a' \in A} Q_\pi(s_{t+1}, a') - Q_\pi(s_t, a_t) \right]. \quad (1)$$

This is what has become known as the *Q-learning* algorithm. Watkins [31] proves that it does converge to the optimal value function, meaning that selecting the action, a, that minimizes $Q(s_t, a)$ for any state s_t will result in the optimal sum of reinforcement over time. The proof of convergence assumes that the sequence of step sizes α_t satisfies the stochastic approximation conditions $\sum \alpha_t = \infty$ and $\sum \alpha_t^2 < \infty$. It also assumes that every state and action are visited infinitely often.

The Q function implicitly defines the policy, π, defined as

$$\pi(s_t) = \operatorname*{argmin}_{a \in A} Q(s_t, a).$$

However, as Q is being learned, π will certainly not be an optimal policy. We must introduce a way of forcing a variety of actions from every state in order to learn sufficiently accurate Q values for the state-action pairs that are encountered.

One problem inherent in the Q-Learning algorithm is due to the use of two policies, one to generate behavior and another, resulting from the min operator in (1), to update the value function. Sutton defined the SARSA algorithm by removing the min operator, thus using the same policy for generating behavior and for training the value function [27]. In Section 4, we use SARSA as the reinforcement learning component of our experiments.

2.3 Architectures

A reinforcement learning algorithm must have some construct to store the value function it learns while interacting with the environment. These algorithms often use a function approximator to store the value function to lessen the curse of dimensionality due to the size of the state and action spaces. There have been many attempts to provide improved control of a reinforcement learner by adapting the function approximator which learns/stores the Q-value function. Anderson adds an effective extension to Q-learning by applying his "hidden restart" algorithm to the difficult pole balancer control task [3]. Moore's Parti-Game Algorithm [17] dynamically builds an approximator through on-line experience. Sutton [26] demonstrates the effectiveness of discrete local function approximators in solving many of the neuro-dynamic problems associated with reinforcement learning control tasks. We turn to Sutton's work with CMACs (Cerebellar Model Articular Controller) to solve some of the implementation problems for our learning agent. Anderson and Kretchmar have also proposed additional algorithms that adapt to form better approximation schemes such as the Temporal Neighborhoods Algorithm [12, 13].

Though not necessary, the policy implicitly represented by a Q-value function can be explicitly represented by a second function approximator, called the actor. This was the strategy followed by Jordan and Jacobs [11] and is very closely related to the actor-critic architecture of Barto, et al., [4] in their actor-critic architecture, and later by

In the work reported in this article, we were able to couch a reinforcement learning algorithm within the robust stability framework by choosing the actor-critic architecture. The actor implements a policy as a mapping from input to control signal, just as a regular feedback controller would. Thus, a system with a fixed, feedback controller and an actor can be analyzed if the actor can be represented in a robust framework. The critic guides the learning of the actor, but the critic is not part of the feedback path of the system. To train the critic, we used the SARSA algorithm. For the actor, we select a two-layer, feedforward neural network with hidden units having hyperbolic tangent activation functions and linear output units. This feedforward network explicitly implements a policy as a mathematical function and is thus amenable to the stability analysis detailed in the next section. The training algorithm for the critic and actor are detailed in Section 4.

3 Stability Analysis of Neural Network Control

3.1 Robust Stability

Control engineers design controllers for physical systems. These systems often possess dynamics that are difficult to measure and change over time. As a consequence, the control engineer never completely knows the precise dynamics of the system. However, modern control techniques rely upon mathematical models (derived from the physical system) as the basis for controller design. There is clearly the potential for problems arising from the differences between the mathematical model (where the design was carried out) and the physical system (where the controller will be implemented).

Robust control techniques address this issue by incorporating *uncertainty* into the mathematical model. Numerical optimization techniques are then applied to the model, but they are confined so as not to violate the uncertainty regions. When compared to the performance of pure optimization-based techniques, robust designs typically do not perform as well on the model (because the uncertainty keeps them from exploiting all the model dynamics). However, optimal control techniques may perform very poorly on the physical plant, whereas the performance of a well designed robust controller on the physical plant is similar to its performance on the model. We refer the interested reader to [24, 32, 9] for examples.

3.2 IQC Stability

Integral quadratic constraints (IQC) are a tool for verifying the stability of systems with uncertainty. In this section, we present a very brief summary of the IQC theory relevant to our problem. The interested reader is directed to [15, 16, 14] for a thorough treatment of IQCs.

Consider the feedback interconnection shown in Figure 2. The upper block, M, is a known Linear-Time-Invariant (LTI) system, and the lower block, Δ is a (block-diagonal) structured uncertainty.

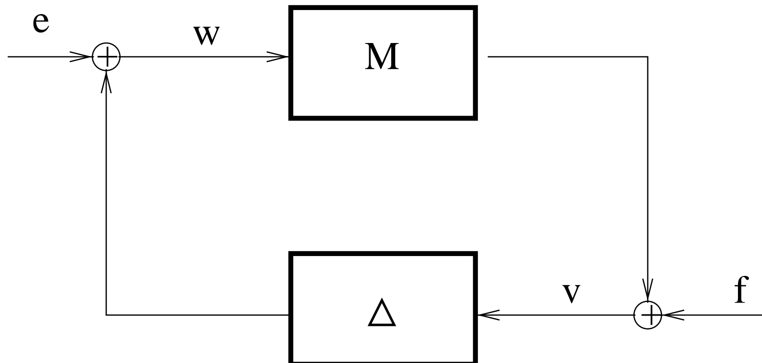

Figure 2: Feedback System

Definition 1 *An **Integral Quadratic Constraint** (IQC) is an inequality describing the relationship between two signals, w and v, characterized by a matrix Π as:*

$$\int_{-\infty}^{\infty} \left| \begin{array}{c} \hat{v}(j\omega) \\ \hat{w}(j\omega) \end{array} \right|^{*} \Pi(j\omega) \left| \begin{array}{c} \hat{v}(j\omega) \\ \hat{w}(j\omega) \end{array} \right| d\omega \geq 0 \tag{2}$$

where $\hat{v}$ and $\hat{w}$ are the Fourier Transforms of $v(t)$ and $w(t)$. We now summarize the main IQC stability theorem from [15].

Theorem 1 *Consider the interconnection system represented in Figure 2 and given by the equations*

$$v = Mw + f \tag{3}$$

$$w = \Delta(v) + e \tag{4}$$

Assume that:

- *The interconnection of M and Δ is well-posed. (i.e., the map from $(v, w) \rightarrow (e, f)$ has a causal inverse)*
- *The IQC defined by Π is satisfied.*
- *There exists an $\epsilon > 0$ such that*

$$\begin{vmatrix} M(j\omega) \\ I \end{vmatrix}^* \Pi(j\omega) \begin{vmatrix} M(j\omega) \\ I \end{vmatrix} \leq -\epsilon I \tag{5}$$

Then the feedback interconnection of M and Δ is stable.

The power of this IQC result lies in its generality and its computability. First we note that many system interconnections can be rearranged into the canonical form of Figure 2 (see [18] for an introduction to these techniques). Secondly, we note that many types of uncertainty descriptions can be well captured as IQCs, including norm bounds, rate bounds, both linear and nonlinear uncertainty, time-varying and time-invariant uncertainty, and both parametric and dynamic uncertainty. Hence this result can be applied in many situations, often without too much conservatism [15, 16]. Moreover, a library of IQCs for common uncertainties is available [14], and more complex IQCs can be built by combining the basic IQCs.

Finally, the computation involved to meet the requirements of the theorem is not difficult. The theorem requirements can be transformed into a Linear Matrix Inequality (LMI). As is well known, LMIs are convex optimization problems for which there exist fast, commercially available, polynomial time algorithms [10]. In fact there is now a beta-version of a Matlab IQC toolbox available at http://www.mit.edu/~cykao/home.html. This toolbox provides an implementation of an IQC library in Simulink, facilitating an easy-to-use graphical interface for setting up IQC problems. Moreover, the toolbox integrates an efficient LMI solver to provide a powerful comprehensive tool for IQC analysis. This toolbox was used for the calculations throughout this article.

3.3 Uncertainty for Neural Networks

In this section we develop our main theoretical results. We only consider the most common kind of neural network—a two-layer, feedforward network with hyperbolic tangent activation functions. First we present a method to determine the stability status of a control system with a fixed neural network, a network with all weights held constant. We also prove the correctness of this method: we guarantee that our static stability test identifies all unstable neuro-controllers. Secondly, we present an analytic technique for ensuring the stability of the neuro-controller while the weights are changing during the training process. We refer to this as dynamic stability. Again, we prove the correctness of this technique in order to provide a guarantee of the system's stability while the neural network is training.

It is critical to note that dynamic stability is not achieved by applying the static stability test to the system after each network weight change. Dynamic stability is fundamentally different than "point-wise" static stability. For example, suppose that we have a network with weights W_1. We apply our static stability techniques to prove that the neuro-controller implemented by W_1 provides a stable system. We then train the network on one sample and arrive at a new weight vector W_2. Again we can demonstrate that the static system given by W_2 is stable, and we proceed in this way to a general W_k, proving static stability at every fixed step. However, this does not prove that the time-varying system, which transitions from W_1 to W_2 and so on, is stable. We require the additional techniques of dynamic stability analysis in order to formulate a reinforcement learning algorithm that guarantees stability throughout the learning process. However, the static stability analysis is necessary for the development of the dynamic stability theorem; therefore, we begin with the static stability case.

Let us begin with the conversion of the nonlinear dynamics of the network's hidden layer into an uncertainty function. Consider a neural network with input vector $x = (x_1, ..., x_n)$ and output vector $a = (a_1, ..., a_m)$. For the experiments described in the next section, the input vector has two components, the error $e = r - y$ and a constant value of 1 to provide a bias weight. The network has h hidden units, input weight matrix $W_{h \times n}$, and output weight matrix $V_{m \times h}$, where the bias terms are included as fixed inputs. The hidden unit activation function is the commonly used hyperbolic tangent function, which produces the hidden unit outputs as vector $\Phi = (\phi_1, \phi_2, \ldots, \phi_h)$. The neural network computes its output by

$$\Phi = Wx, \tag{6}$$

$$a = V \tanh(\Phi). \tag{7}$$

With moderate rearrangement, we can rewrite the vector notation expression in (6,7) as

$$\begin{aligned}
\Phi &= Wx, \\
\gamma_j &= \begin{cases} \frac{\tanh(\phi_j)}{\phi_j}, & \text{if } \phi_j \neq 0; \\ 1, & \text{if } \phi_j = 0, \end{cases} \\
\Gamma &= \mathsf{diag}\{\gamma_j\}, \\
a &= V\Gamma\Phi.
\end{aligned} \tag{8}$$

The function, γ, computes the output of the hidden unit divided by the input of the hidden unit; this is the *gain* of the hyperbolic tangent hidden unit. Note that $\tanh$ is a sector bounded function (belonging to the sector [0,1]), as illustrated in Figure 3.

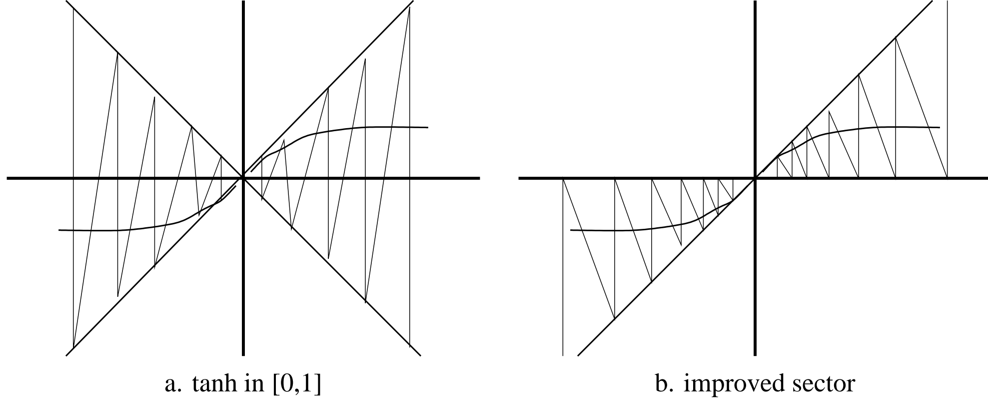

Figure 3: Sector bounds on $\tanh$

Equation 8 offers two critical insights. First, it is an exact reformulation of the neural network computation. We have not changed the functionality of the neural network by restating the computation in this equation form; this is still the applied version of the neuro-controller. Second, Equation 8 cleanly separates the nonlinearity of the neural network hidden layer from the remaining linear operations of the network. This equation is a multiplication of linear matrices (weights) and one nonlinear matrix, Γ. Our goal, then, is to replace the matrix Γ with an uncertainty function to arrive at a "testable" version of the neuro-controller (i.e., in a form suitable for IQC analysis).

First, we must find an appropriate IQC to cover the nonlinearity in the neural network hidden layer. From Equation 8, we see that all the nonlinearity is captured in a diagonal matrix, Γ. This matrix is composed of individual hidden unit gains, γ, distributed along the diagonal. These act as nonlinear gains via

$$y(t) \quad = \quad \gamma x(t) \quad = \left(\frac{\tanh(x(t))}{x(t)}\right) x(t) \quad = \quad \tanh(x(t)) \tag{9}$$

(for input signal $x(t)$ and output signal $y(t)$). In IQC terms, this nonlinearity is referred to as a *bounded odd slope nonlinearity*. There is an Integral Quadratic Constraint already configured to handle such a condition. The IQC nonlinearity, ψ, is characterized by an odd condition and a bounded slope, i.e., the input-output relationship of the block is $y(t) = \psi(x(t))$ where ψ is a static nonlinearity satisfying (see [14]):

$$\psi(-x) = -\psi(x), \tag{10}$$

$$\alpha(x_1 - x_2)^2 \leq (\psi(x_1) - \psi(x_2))(x_1 - x_2) \leq \beta(x_1 - x_2)^2. \tag{11}$$

For our specific network, we choose $\alpha = 0$ and $\beta = 1$. Note that each nonlinear hidden unit function ($\tanh(x)$) satisfies the odd condition, namely:

$$\tanh(-x) = -\tanh(x) \tag{12}$$

and furthermore the bounded slope condition

$$0 \ \leq \ (\tanh(x_1) - \tanh(x_2))(x_1 - x_2) \ \leq \ (x_1 - x_2)^2 \tag{13}$$

is equivalent to (assuming without loss of generality that $x_1 > x_2$)

$$0 \ \leq \ (\tanh(x_1) - \tanh(x_2)) \ \leq \ (x_1 - x_2) \tag{14}$$

which is clearly satisfied by the $\tanh$ function since it has bounded slope between 0 and 1 (see Figure 3). Hence the hidden unit function is covered by the IQCs describing the bounded odd slope nonlinearity (10,11).

We now need only construct an appropriately dimensioned diagonal matrix of these bounded odd slope nonlinearity IQCs and incorporate them into the system in place of the Γ matrix. In this way we form the testable version of the neuro-controller that will be used in the following Static Stability Procedure.

Before we state the Static Stability Procedure, we also address the IQC used to cover the other non-LTI feature of our neuro-controller. In addition to the nonlinear hidden units, we must also cover the time-varying weights that are adjusted during training. Again, we will forego the complication of designing our own IQC and, instead, select one from the pre-constructed library of IQCs. The *slowly time-varying real scalar IQC* allows for a linear gain block which is (slowly) time-varying, i.e., a block with input-output relationship $y(t) = \psi(t)x(t)$, where the gain $\psi(t)$ satisfies (see [15]):

$$|\psi(t)| \leq \beta, \tag{15}$$

$$|\dot{\psi}(t)| \leq \alpha, \tag{16}$$

where ψ is the non-LTI function. In our case ψ is used to cover a time varying weight update in our neuro-controller, which accounts for the change in the weight as the network learns . The key features are that ψ is bounded, time-varying, and the rate of change of ψ is bounded by some constant, α. We use the neural network learning rate to determine the bounding constant, α, and the algorithm checks for the largest allowable β for which we can still prove stability. This determines a safe neighborhood in which the network is allowed to learn.

Static Stability Procedure: We now construct two versions of the neuro-control system, an applied version and a testable version. The applied version contains the full, nonlinear neural network as it will be implemented. The testable version covers all non-LTI blocks with uncertainty suitable for IQC analysis, so that the applied version is now contained in the set of input-output maps that this defines. For the static stability procedure, we temporarily assume the network weights are held constant. The procedure consists of the following steps:

1. Design the nominal, robust LTI controller for the given plant model so that this nominal system is stable.
2. Add a feedforward, nonlinear neural network in parallel to the nominal controller. We refer to this as the applied version of the neuro-controller.
3. Recast the neural network into an LTI block plus the odd-slope IQC function described above to cover the nonlinear part of the neural network. We refer to this as the testable version of the neuro-controller.
4. Apply IQC-analysis. If a feasible solution to the IQC is found, the testable version of the neuro-control system is stable. If a feasible solution is not found, the system is not proven to be stable.

Dynamic Stability Procedure: We are now ready to state the dynamic stability procedure. The first three steps are the same as the static stability procedure.

1. Design the nominal, robust LTI controller for the given plant model so that this nominal system is stable.
2. Add a feedforward, nonlinear neural network in parallel to the nominal controller. We refer to this as the applied version of the neuro-controller.
3. Recast the neural network into an LTI block plus the odd-slope IQC function described above to cover the nonlinear part of the neural network. We refer to this as the testable version of the neuro-controller.

4. Introduce an additional IQC block, the slowly time-varying IQC, to the testable version, to cover the time-varying weights in the neural network.

5. Perform a search procedure and IQC analysis to find bounds on the perturbations of the current neural network weight values within which the system is stable. This defines a known "stable region" of weight values.

6. Train the neural network in the applied version of the system using reinforcement learning while bounding the rate of change of the neuro-controller's vector function by a constant. Continue training until any of the weights approach its bounds of the stable region, at which point repeat the previous step, then continue with this step.

In the next section, we demonstrate the application of the dynamic stability procedure and study its behavior, including the adaptation of the neural network's weights and the bounds of the weights' stable region.

4 An Algorithm for Dynamically Stable, Reinforcement Learning Control

In this section, the structure of the actor and critic parts of our reinforcement learning approach are first described. This is followed by the procedure by which the actor and critic are trained while guaranteeing stability of the system. Experiments are then described in which we apply this algorithm to two control tasks.

4.1 Architecture and Algorithm

Recall that the critic accepts a state and action as inputs and produces the value function for the state/action pair. Notice that the critic is not a direct part of the control system feedback loop and thus does not play a direct role in the stability analysis, but stability analysis does constrain the adaptation of the weights that is guided by the critic. For the experiments in this section, we implemented several different architectures for the critic and found that a simple table look-up mechanism (discrete and local) is the architecture that worked best in practice. The critic is trained to predict the expected sum of future reinforcements that will be observed, given the current state and action. In the following experiments, the reinforcement was simply defined to be the magnitude of the error between the reference signal and the plant output that we want to track the reference.

As described earlier, the actor neural network, whose output is added to the output of the nominal controller, is a standard two-layer, feedforward network with hyperbolic tangent nonlinearity in the hidden layer units and just linear activation functions in the output layer. The actor network is trained by first estimating the best action for the current state by comparing the critic's prediction for various actions and selecting the action with minimal prediction, because the critic predicts sums of future errors. The best action is taken as the target for the output of the actor network and its weights are updated according to the error backpropagation algorithm which performs a gradient descent in the squared error in the network's output following the common error backpropagation algorithm [20].

Figure 4 places the actor-critic network within the control framework. The actor network receives the tracking error e and produces a control signal, a, which is both added to the traditional control signal and is fed into the critic network. The critic network uses e (the state) and a (the action) to produce the Q-value which evaluates the state/action pair. The critic net, via a local search, is used to estimate the optimal action to update the weights in the actor network. Figure 5 depicts each component. The details for each component are presented here, some of which are repeated from

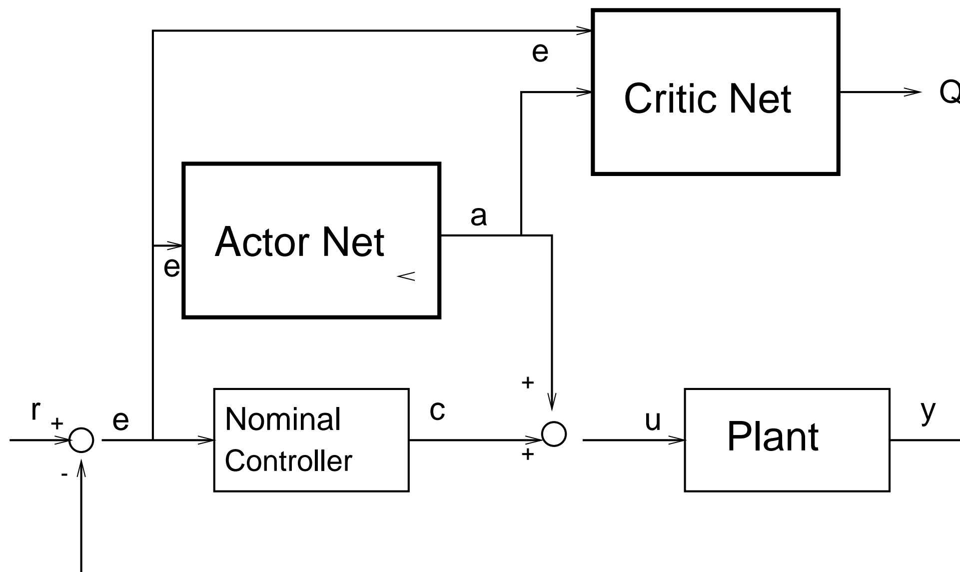

Figure 4: Learning in the Actor-Critic

the previous section.

Let n be the number of inputs to the actor network. For most tasks, this includes the tracking error and possibly additional plant state variables. Also included is an extra variable held constant at 1 for the bias input. Let m be the number of components in the output, a, of the actor network. This is the number of control signals needed for the plant

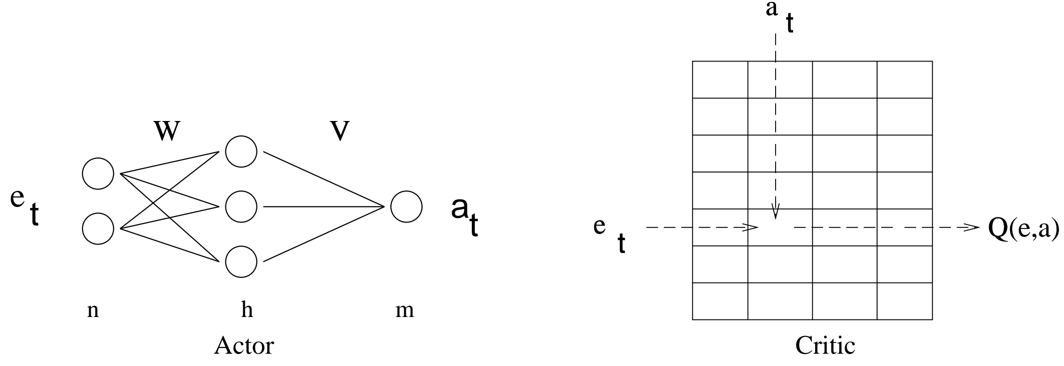

Figure 5: Network Architectures

input. Let h be the number of hidden units in the actor network. The best value for h is determined experimentally. The hidden layer weights are given by W, an $h \times n$ matrix, and the output weights are given by V, an $m \times h$ matrix. The input to the actor network is given by vector x, composed of the error, e, between the reference signal, r, and the plant output, y, and of a constant input that adds a bias term to the weighted sum of each hidden unit. Other relevant measurements of the system could be included in the input vector to the actor network, but for the simple experiments described here, the only variable input was e.

The critic receives inputs e and a. An index into the table of Q values stored in the critic is found by determining which e and a partition within which the current error and action values fall. The number of partitions for each input is determined experimentally.

We can now summarize the steps of our robust reinforcement learning algorithm. Here we focus on the reinforcement learning steps and the interaction of the nominal controller, plant, actor network, and critic. The stability analysis is simply referred to, as it is described in detail in the previous section. Variables are given a time step subscript. The time step is defined to increment by one as signals pass through the plant.

The definition of our algorithm starts with calculating the error between the reference input and the plant output.

$$e_t = r_t - y_t$$

Next, calculate the outputs of the hidden units, Φ_t, and of the output unit, which is the action, a_t:

$$\Phi_t = \tanh(W_t e_t)$$
$$a_t = \begin{cases} V_t \Phi_t, & \text{with probability } 1 - \epsilon_t; \\ V_t \Phi_t + a_{\text{rand}}, & \text{with probability } \epsilon_t, \text{ where } a_{\text{rand}} \text{ is a Gaussian} \\ & \quad \text{random variable with mean 0 and variance 0.05} \end{cases}$$

Repeat the following steps forever.

Apply the fixed, feedback control law, f, to input e_t, and sum the output of the fixed controller, c_t, and the neural network output, a_t, to get u_t. This combined control output is then applied to the plant to get the plant output y_{t+1} for the next time step through the plant function g.

$$c_t = f(e_t)$$
$$u_t = c_t + a_t$$
$$y_{t+1} = g(u_t)$$

Again calculate the error, e_{t+1}, and the hidden and output values of the neural network, Φ_{t+1} and a_{t+1}.

$$\begin{aligned}
e_{t+1} &= \mathrm{r}_{t+1} - y_{t+1} \\
\Phi_{t+1} &= \tanh(W_t e_{t+1}) \\
a_{t+1} &= \begin{cases} V_t \Phi_{t+1}, & \text{with probability } 1 - \epsilon_{t+1}; \\ V_t \Phi_{t+1} + a_{\text{rand}}, & \text{with probability } \epsilon_{t+1}, \text{ where } a_{\text{rand}} \text{ is a Gaussian} \\ & \quad \text{random variable with mean 0 and variance 0.05} \end{cases}
\end{aligned}$$

Now assign the reinforcement, R_{t+1}, for this time step. For the experiments presented in this paper, we simply define the reinforcement to be the absolute value of the error,

$$R_{t+1} = |e_{t+1}|.$$

At this point, we have all the information needed to update the policy stored in the neural network and the value function stored in the table represented by Q. Let Q_{index} be a function that maps the value function inputs, e_t and a_t, to the corresponding index into the Q table. To update the neural network, we first estimate the optimal action, a_t^*, at step t by minimizing the value of Q for several different action inputs in the neighborhood, A, of a_t. The neighborhood is defined as

$$A = \{a_i | a_i = a_{min} + i(a_{\max} - a_{\min})/n, i = 1, \dots, n, a_{\min} < a_t < a_{\max}\}$$

for which the estimate of the optimal action is given by

$$a_t^* = \operatorname*{argmin}_{a \in A} Q_{Q_{\text{index}}(e_t, a)}$$

Updates to the weights of the neural network are proportional to the difference between this estimated optimal action and the actual action:

$$\begin{aligned}
V_{t+1} &= V_t + \beta(a_t^* - a_t)\Phi_t^T \\
W_{t+1} &= W_t + \beta V^T (a_t^* - a_t) \cdot (1 - \Phi_t \cdot \Phi_t) e_t,
\end{aligned}$$

where $\cdot$ represents component-wise multiplication. We now update the value function, Q. The Q indices, q_t, for step t and for step $t+1$ are calculated first, then the Q value for step t is updated:

$$\begin{aligned}
q_t &= Q_{\text{index}}(e_t, a_t) \\
q_{t+1} &= Q_{\text{index}}(e_{t+1}, a_{t+1}) \\
Q_{q_t} &= Q_{q_t} + \alpha(R_{t+1} + \gamma Q_{q_{t+1}} - Q_{q_t})
\end{aligned}$$

Now we determine whether or not the new weight values, W_{t+1} and V_{t+1}, remain within the stable region S. If they are, S remains unchanged. Otherwise a new value for S is determined.

$$\begin{aligned}
\text{If } (W_{t+1}, V_{t+1}) \in S_t, \quad & \text{then } S_{t+1} = S_t, \\
& \text{else } W_{t+1} = W_t \\
& \qquad V_{t+1} = V_t \\
& \qquad S_{t+1} = \text{newbounds}(W_t, V_t)
\end{aligned}$$

Repeat above steps forever.

To calculate new bounds, S, do the following steps. First, collect all of the neural network weight values into one vector, N, and define an initial guess at allowed weight perturbations, P, as factors of the current weights. Define the initial guess to be proportional to the current weight values.

$$\begin{aligned}
N &= (W_t, V_t) = (n_1, n_2, \dots) \\
P &= \frac{N}{\sum_i n_i}
\end{aligned}$$

Now adjust these perturbation factors to estimate the largest factors for which the system remains stable. Let z_u and z_s be scalar multipliers of the perturbation factors for which the system is unstable and stable, respectively. Initialize them to 1.

$$z_u = 1$$
$$z_s = 1$$

Increase z_u until system is unstable:

If stable for $N \pm P \cdot N$,
then while stable for $N \pm z_u P \cdot N$, do
$$z_u = 2 z_u$$

Decrease z_s until system is stable:

If not stable for $N \pm P \cdot N$,
then while not stable for $N \pm z_s P \cdot N$ do
$$z_s = \frac{1}{2} z_s$$

Perform a finer search between z_s and z_u to increase z_s as much as possible:

While $\frac{z_u - z_s}{z_s} < 0.05$ do
$$z_m = \frac{z_u + z_s}{2}$$
If not stable for $N \pm z_m P \cdot N$
then $z_s = z_m$
else $z_u = z_m$

We can now define the new stable perturbations, which in turn define the set S of stable weight values.

$$P = z_s P = (p_1, p_2, \dots)$$
$$S = \left\{ [(1-p_1)n_1, (1+p_1)n_1] \times [(1-p_2)n_2, (1+p_2)n_2] \times \cdots \right\}$$

4.2 Experiments

We now demonstrate our robust reinforcement learning algorithm on two simple control tasks.. The first task is a simple first-order positioning control system. The second task adds second-order dynamics which are more characteristic of standard "physically realizable" control problems.

4.2.1 Task 1

A single reference signal, r, moves on the interval $[-1, 1]$ at random points in time. The plant output, y, must track r as closely as possible. The plant is a first order system and thus has one internal state variable, x. A control signal, u, is provided by the controller to position y closer to r. The dynamics of the discrete-time system are given by:

$$x_{t+1} = x_t + u_t \tag{17}$$
$$y_t = x_t \tag{18}$$

where t is the discrete time step for which we used 0.01 seconds. We implement a simple proportional controller (the control output is proportional to the size of the current error) with $K_p = 0.1$.

$$e_t = r_t - y_t \tag{19}$$
$$u_t = 0.1\, e_t \tag{20}$$

The critic's table for storing the values of $Q(e,a)$ is formed by partitioning the ranges of e and a into 25 intervals, resulting in a table of 625 entries. Three hidden units were found to work well for the actor network.

The critic network is a table look-up with input vector $[e,a]$ and the single value function output, $Q(e,a)$. The table has 25 partitions separating each input forming a 25x25 matrix. The actor network is a two-layer, feed forward neural network. The input is e. There are three $tanh$ hidden units, and one network output a. The entire network is then added to the control system. This arrangement is depicted in block diagram form in Figure 4.

For training, the reference input r is changed to a new value on the interval $[-1,1]$ stochastically with an average period of 20 time steps (every half second of simulated time). We trained for 2000 time steps at learning rates of $\alpha = 0.5$ and $\beta = 0.1$ for the critic and actor networks, respectively. Then we trained for an additional 2000 steps with learning rates of $\alpha = 0.1$ and $\beta = 0.01$. Recall that α is the learning rate of the critic network and β is the learning rate for the actor network. The values for these parameters were found by experimenting with a small number of different values.

Before presenting the results of this first experiment, we summarize how the IQC approach to stability is adapted to the system for Task 1. Our IQC analysis is based on Matlab and Simulink. Figure 6 depicts the Simulink diagram for the nominal control system in Task 1. We refer to this as the nominal system because there is no neuro-controller added to the system. The plant is represented by a rectangular block that implements a discrete-time state space system. The simple proportional controller is implemented by a triangular gain block. Another gain block provides the negative feedback path. The reference input is drawn from the left and the system output exits to the right.

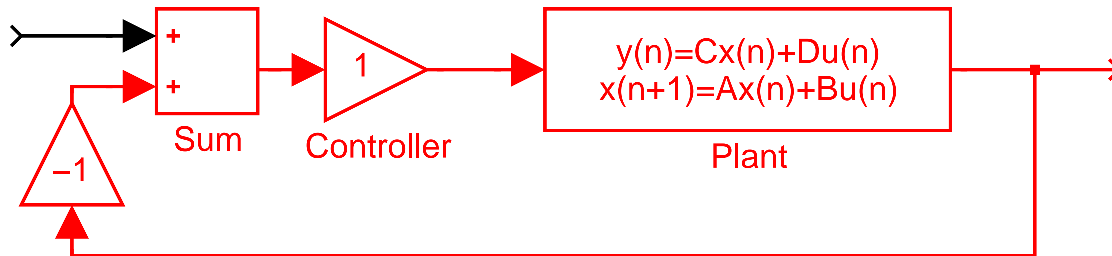

Figure 6: Task 1: Nominal System

Next, we add the neural network controller to the diagram. Figure 7 shows the complete version of the neuro-controller including the $tanh$ function. This diagram is suitable for conducting simulation studies in Matlab. However, this diagram cannot be used for stability analysis, because the neural network, with the nonlinear $tanh$ function, is not represented as LTI components and uncertain components. Constant gain matrices are used to implement the input side weights, W, and output side weights, V. For the static stability analysis in this section, we start with an actor net that is already fully trained. The static stability test will verify whether this particular neuro-controller implements a stable control system.

Notice that the neural network is in parallel with the existing proportional controller; the neuro-controller adds to the proportional controller signal. The other key feature of this diagram is the absence of the critic network; only the actor net is depicted here. Recall that the actor net is a direct part of the control system while the critic net does not directly affect the feedback/control loop of the system. The critic network only influences the direction of learning for the actor network. Since the critic network plays no role in the stability analysis, there is no reason to include the critic network in any Simulink diagrams.

To apply IQC analysis to this system, we replace the nonlinear $tanh$ function with the *odd-slope nonlinearity* discussed in the previous section, resulting in Figure 8. The *performance* block is another IQC block that triggers the analysis.

Again, we emphasize that there are *two* versions of the neuro-controller. In the first version, shown in Figure 7, the neural network includes all its nonlinearities. This is the actual neural network that will be used as a controller in the system. The second version of the system, shown in Figure 8, contains the neural network converted into the robustness analysis framework; we have extracted the LTI parts of the neural network and replaced the nonlinear $tanh$ hidden layer with an odd-slope uncertainty. This version of the neural network will never be implemented as a controller; the sole purpose of this version is to test stability. Because this version is LTI plus uncertainties, we can use the IQC-analysis tools to compute the stability margin of the system. Again, because the system with uncertainties *overestimates* the gain

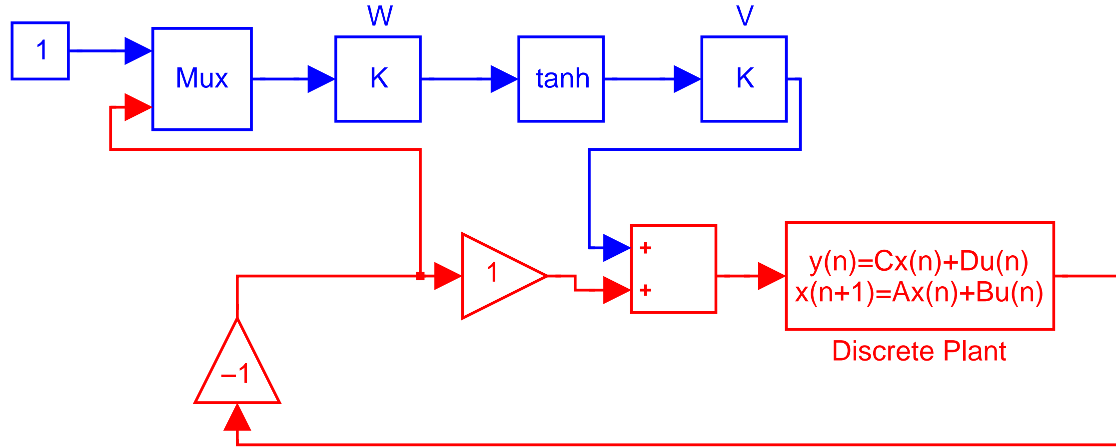

Figure 7: Task 1: With Neuro-Controller

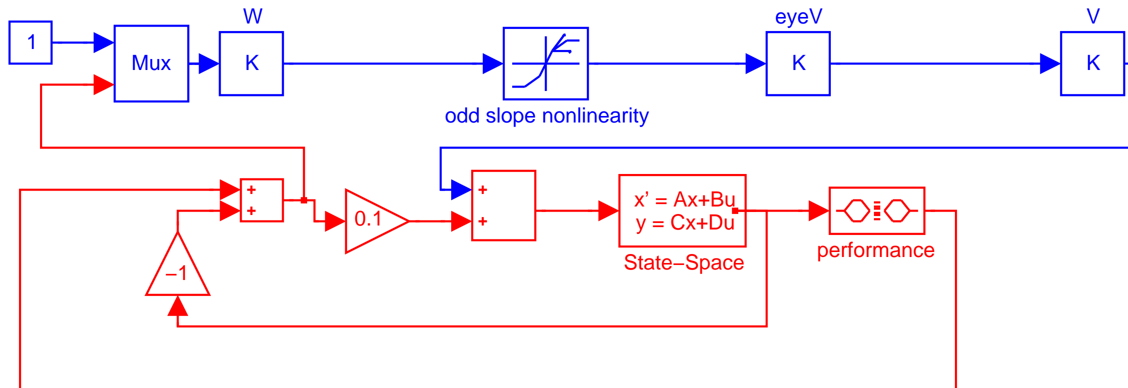

Figure 8: Task 1: With Neuro-Controller as LTI (IQC)

of the nonlinearity in the original system, a stability guarantee on the system with uncertainties also implies a stability guarantee on the original system.

When we run the IQC analysis on this system, a feasibility search for a matrix satisfying the IQC function is performed. If the search is feasible, the system is guaranteed stable; if the search is infeasible, the system is not guaranteed to be stable. We apply the IQC analysis to the Simulink diagram for Task 1 and find that the feasibility constraints are easily satisfied; the neuro-controller is guaranteed to be stable.

At this point, we have assured ourselves that the neuro-controller, after having completely learned its weight values during training, implements a stable control system. Thus we have achieved *static stability*. We have not, however, assured ourselves that the neuro-controller did not temporarily implement an unstable controller while the network weights were being adjusted during learning.

Now we impose limitations on the learning algorithm to ensure the network is stable according to dynamic stability analysis. Recall this algorithm alternates between a stability phase and a learning phase. In the stability phase, we use IQC-analysis to compute the maximum allowed perturbations for the actor network weights that still provide an overall stable neuro-control system. The learning phase uses these perturbation sizes as room to safely adjust the actor net weights.

To perform the stability phase, we add an additional source of uncertainty to our system. We use an STV (Slowly Time-Varying) IQC block to capture the weight change uncertainty. This diagram is shown in Figure 9. The matrices

dW and dV are the perturbation matrices. An increase or decrease in dW implies a corresponding increase or decrease in the uncertainty associated with W. Similarly we can increase or decrease dV to enact uncertainty changes to V.

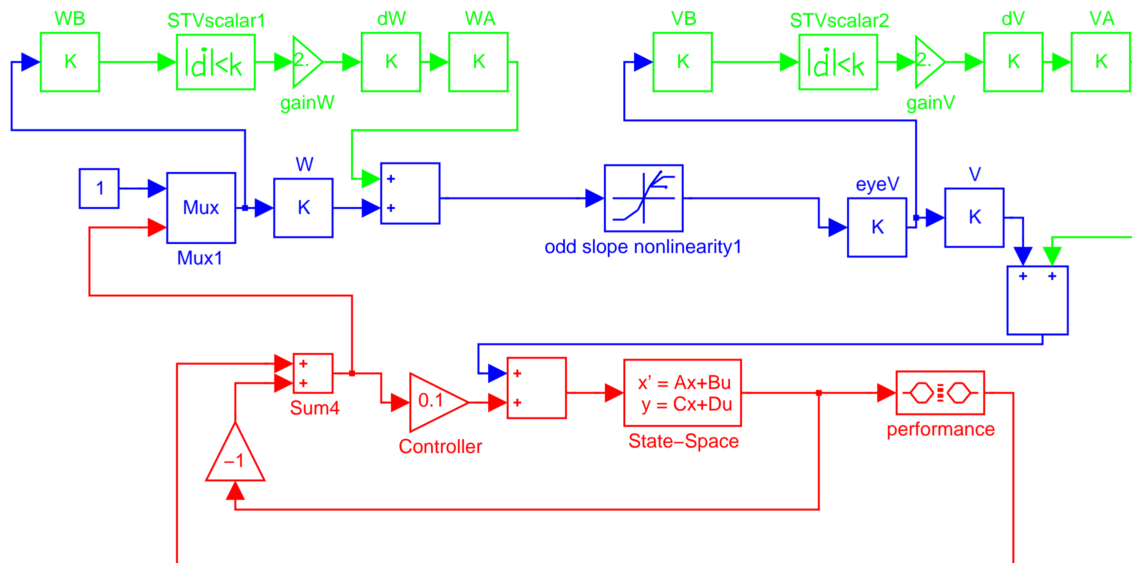

Figure 9: Task 1: Simulink Diagram for Dynamic IQC-analysis

The matrices WA, WB, VA, and VB are simply there for re-dimensioning the sizes of W and V; they have no affect on the uncertainty or norm calculations. In the diagram, dW and dV contain all the individual perturbations along the diagonal while W and V are not diagonal matrices. Thus, W_{hxn} and dW_{hnxhn} are not dimensionally compatible. By multiplying with WA and WB we fix this "dimensional incompatibility" without affecting any of the numeric computations. VA and VB are analogous matrices for V and dV.

If analysis of the system in Figure 9 shows it is stable, we are guaranteed that our control system is stable for the current neural network weight values. Furthermore, the system will remain stable if we change the neural network weight values as long as the new weight values do not exceed the range specified by the perturbation matrices, dW and dV. In the learning phase, we apply the reinforcement learning algorithm until one of the network weights approaches the range specified by the additives.

We now present the results of applying the robust reinforcement learning algorithm to Task 1. We trained the neural network controller as described in earlier in this section. We place the final neural network weight values (W and V) in the constant gain matrices of the Simulink diagram in Figure 7. We then simulate the control performance of the system. A time-series plot of the simulated system is shown in Figure 10. The top diagram shows the system with only the nominal, proportional controller, corresponding to the Simulink diagram in Figure 6. The bottom diagram shows the same system with both the proportional controller and the neuro-controller as specified in Figure 7. The reference input, r, shown as a dotted line, takes six step changes. The solid line is the plant output, y. The small-magnitude line is the combined output of the neural network and nominal controller, u.

The system was tested for a 10 second period (1000 discrete time steps with a sampling period of 0.01). We computed the sum of the squared tracking error (SSE) over the 10 second interval. For the nominal controller only, the $SSE = 33.20$. Adding the neuro-controller reduced the SSE to 11.73. Clearly, the reinforcement learning neuro-controller is able to improve the tracking performance dramatically. Note, however, with this simple first-order system it is not difficult to construct a better performing proportional controller. In fact, setting the constant of proportionality to 1 ($K_p = 1$) achieves minimal control error. We have purposely chosen a suboptimal controller so that the neuro-controller has room to learn to improve control performance.

To provide a better understanding of the nature of the actor-critic design and its behavior on Task 1, we include the following diagrams. Recall that the purpose of the critic net is to learn the value function (Q-values). The two inputs to the critic net are the system state (which is the current tracking error e) and the actor net's control signal (a). The critic net forms the Q-values, or value function, for these inputs; the value function is the expected sum of future squared

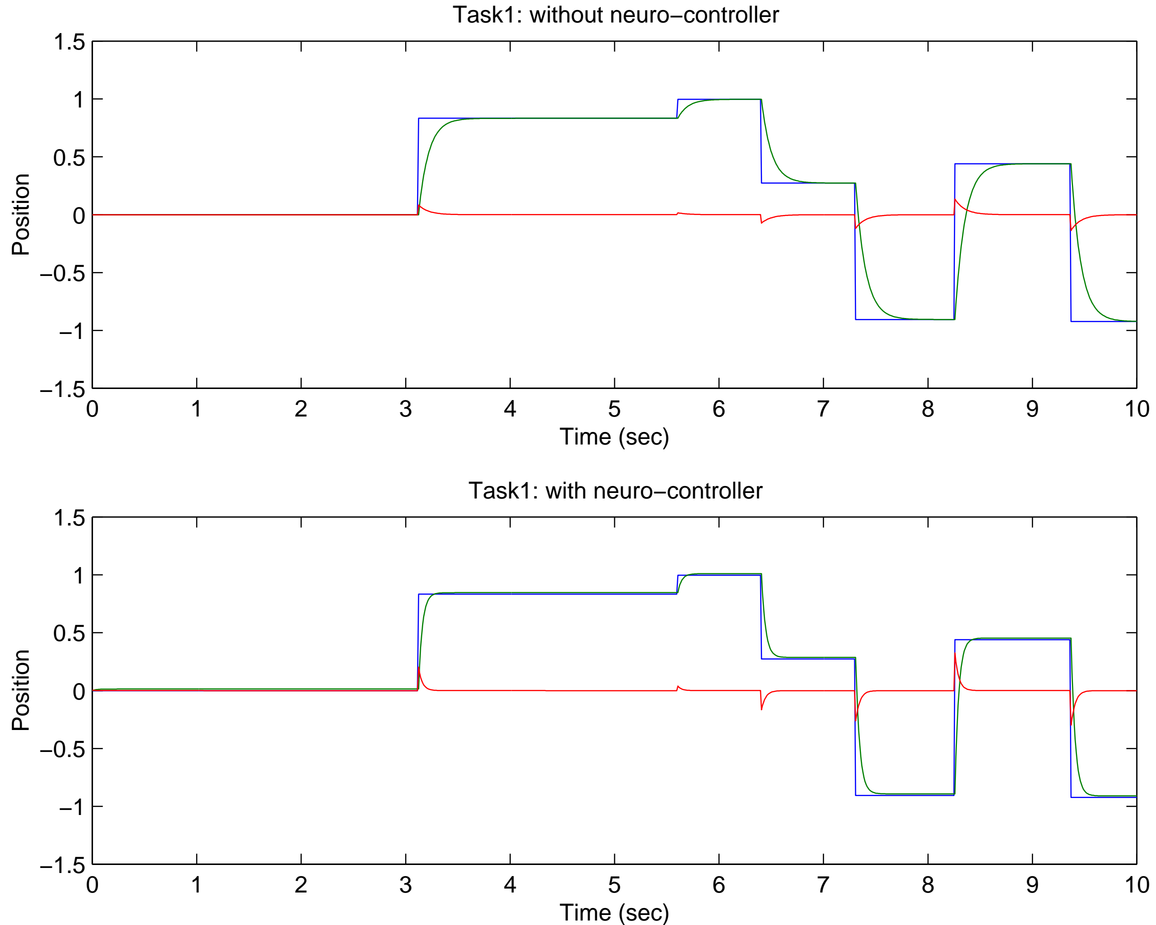

Figure 10: Task 1: Simulation Run

tracking errors. In Figure 11 we see the value function learned by the critic net. The tracking error e is on the x-axis while the actor network control action a forms the y-axis. For any given point (e, a) the height (z-axis) of the diagram represents the expected sum of future reinforcement, or tracking error magnitudes.

We can take "slices", or y-z planes, from the diagram by fixing the tracking error on the x-axis. Notice that for a fixed tracking error e, we vary a to see a "trough-like" shape in the value function. The low part of the trough indicates the minimum discounted sum squared error for the system. This low point corresponds to the control action that the actor net should ideally implement.

It is important to keep in mind that the critic network is an approximation to the true value function. The critic network improves its approximation through learning by sampling different pairs (e, a) and computing the resulting sum of future tracking errors. This "approximation" accounts for the bumpy surface in Figure 11.

The actor net's purpose is to implement the current policy. Given the input of the system state (e), the actor net produces a continuous-valued action (a) as output. In Figure 12 we see the function learned by the actor net. For negative tracking errors $(e < 0)$ the system has learned to output a strongly negative control signal. For positive tracking errors, the network produces a positive control signal. The effects of this control signal can be seen qualitatively by examining the output of the system in Figure 10.

The learning algorithm is a repetition of stability phases and learning phases. In the stability phases we estimate the maximum additives, dW and dV, which still retain system stability. In the learning phases, we adjust the neural network weights until one of the weights approaches its range specified by its corresponding additive. In this section, we present a visual depiction of the learning phase for an agent solving Task 1.

In order to present the information in a two-dimensional plot, we switch to a minimal actor net. Instead of the three

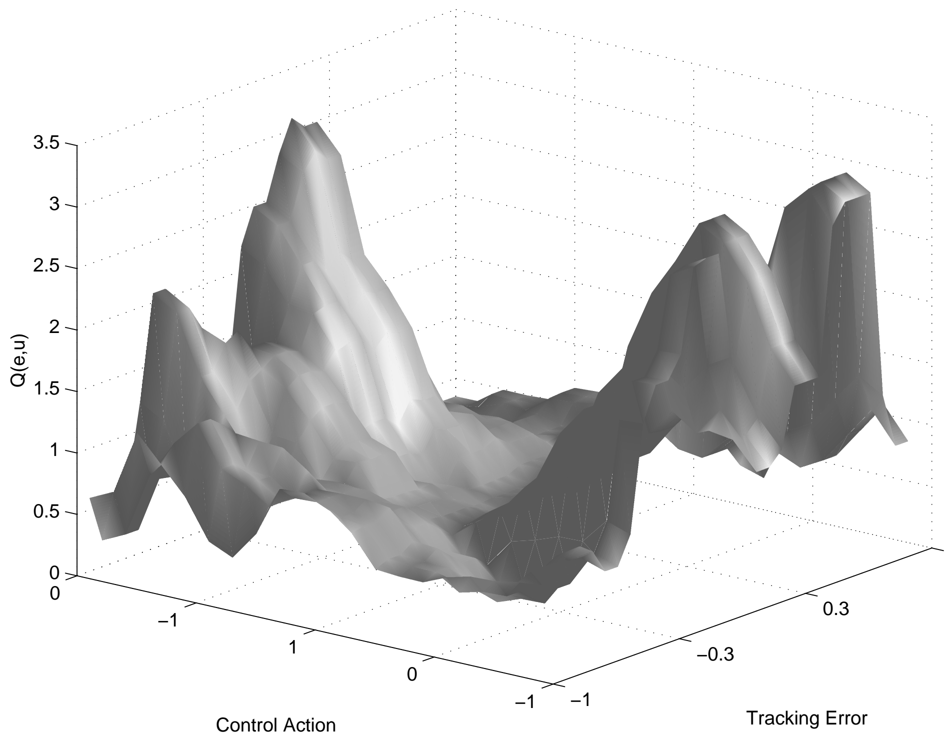

Figure 11: Task 1: Critic Net's Value Function

$tanh$ hidden units specified earlier, we use one hidden unit *for this subsection only*. Thus, the actor network has two inputs (the bias = 1 and the tracking error e), one $tanh$ hidden unit, and one output (a). This network will still be able to learn a relatively good control function. Refer back to Figure 12 to convince yourself that only one hidden $tanh$ unit is necessary to learn this control function; we found, in practice, that three hidden units often resulted in faster learning and slightly better control.

For this *reduced* actor net, we now have smaller weight matrices for the input weights W and the output weights V in the actor net. W is a 2x1 matrix and V is a 1x1 matrix, or scalar. Let W_1 refer to the first component of W, W_2 refer to the second component, and V simply refers to the lone element of the output matrix. The weight, W_1, is the weight associated with the bias input (let the bias be the first input to the network and let the system tracking error, e, be the second input). From a stability standpoint, W_1 is insignificant. Because the bias input is clamped at a constant value of 1, there really is no "magnification" from the input signal to the output. The W_1 weight is not on the input/output signal pathway and thus there is no contribution of W_1 to system stability. Essentially, we do not care how weight W_1 changes as it does not affect stability. However, both W_2 (associated with the input e) and V do affect the stability of the neuro-control system as these weights occupy the input/output signal pathway and thus affect the closed-loop energy gain of the system.

To visualize the neuro-dynamics of the actor net, we track the trajectories of the individual weights in the actor network as they change during learning. The weights W_2 and V form a two-dimensional picture of how the network changes during the learning process. Figure 13 depicts the two-dimensional weight space and the trajectory of these two weights during a typical training episode. The x-axis shows the second input weight W_2 while the y-axis represents the single output weight V. The trajectory begins with the black points and progresses to different shades of gray. Each point along the trajectory represents a weight pair (W_2,V) achieved at some point during the learning process.

The shades represent different phases of the learning algorithm. First, we start with a stability phase by computing, via IQC-analysis, the amount of uncertainty which can be added to the weights; the resulting perturbations, dW and dV, indicate how much learning we can perform and still remain stable. The black part of the trajectory represents the learning that occurred for the first values of dW and dV. The next portion of the trajectory corresponds to the first

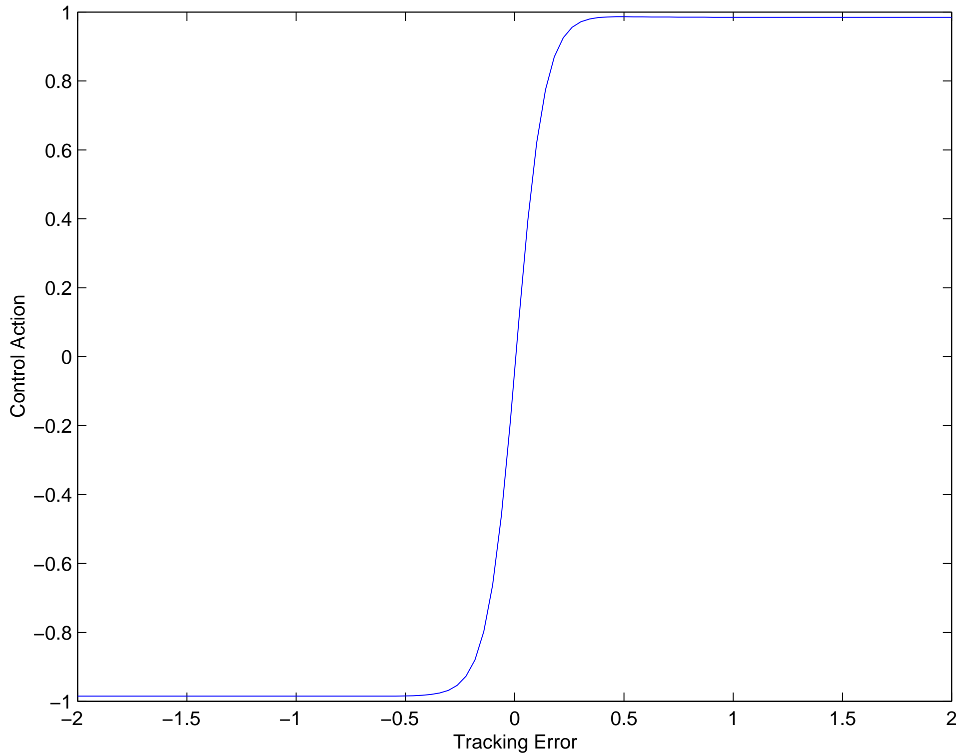

Figure 12: Task 1: Actor Net's Control Function

learning phase. After the first learning phase, we then perform another stability phase to compute new values for dW and dV. We then enter a second learning phase that proceeds until we attempt a weight update exceeding the allowed range. This process of alternating stability and learning phases repeats until we are satisfied that the neural network is fully trained. In the diagram of Figure 13 we see a total of five learning phases.

Recall that the terms dW and dV indicate the maximum uncertainty, or perturbation, we can introduce to the neural network weights and still be assured of stability. If W_2 is the current weight associated with the input e, we can increase or decrease this weight by dW and still have a stable system. $W_2 + dW$ and $W_2 - dW$ form the range, R_{W_2}, of "stable values" for the input actor weight W_2. These are the values of W_2 for which the overall control system is guaranteed to be stable. Similarly $V \pm dV$ form the stable range of output weight values. We depict these ranges as rectangular boxes in our two-dimensional trajectory plot. These boxes are shown in Figure 13.

Again, there are five different bounding boxes corresponding to the five different stability/learning phases. As can be seen from the black trajectory in this diagram, training progresses until the V weight reaches the edge of the blue bounding box. At this point we must cease our current reinforcement learning phase, because any additional weight changes will result in an unstable control system (technically, the system might still be stable but we are no longer *guaranteed* of the system's stability – the stability test is conservative in this respect). At this point, we recompute a new bounding box using a second stability phase; then we proceed with the second learning phase until the weights violate the new bounding box. In this way the stable reinforcement learning algorithm alternates between stability phases (computing bounding boxes) and learning phases (adjusting weights within the bounding boxes).

It is important to note that if the trajectory reaches the edge of a bounding box, we may still be able to continue to adjust the weight in that direction. Hitting a bounding box wall does not imply that we can no longer adjust the neural network weight(s) in that direction. Recall that the edges of the bounding box are computed *with respect to the network weight values at the time of the stability phase*; these initial weight values are the point along the trajectory in the exact center of the bounding box. This central point in the weight space is the value of the neural network weights at the beginning of this particular stability/learning phase. This central weight space point is the value of W_2 and V that are used to compute dW and dV. Given that our current network weight values are that central point, the bounding

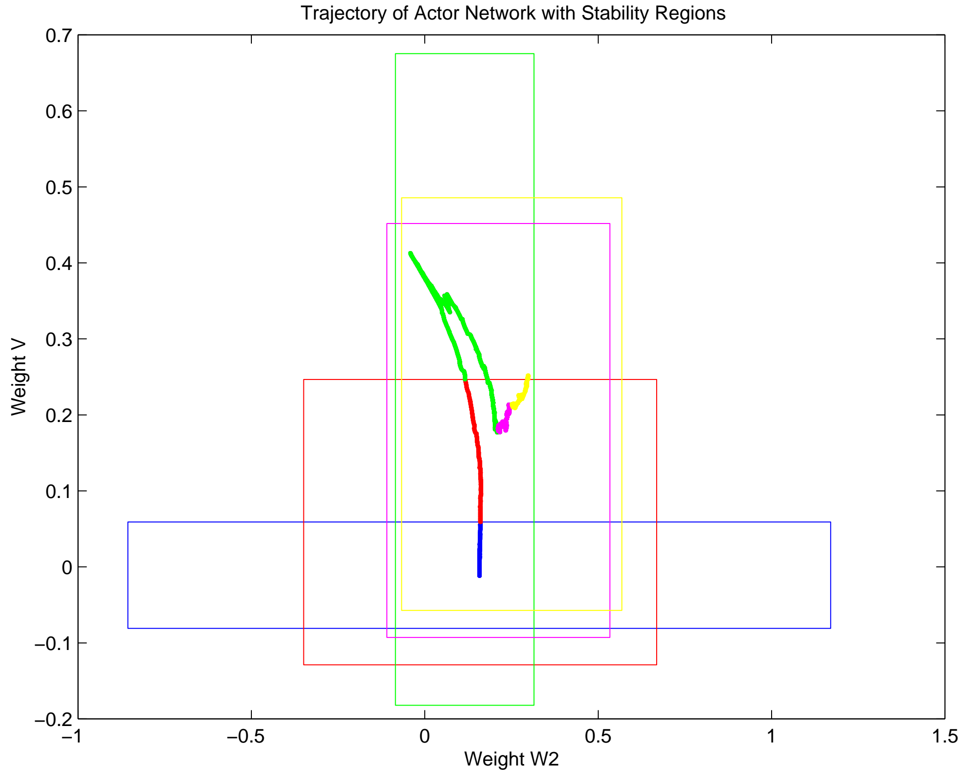

Figure 13: Task 1: Weight Update Trajectory with Bounding Boxes

box is the *limit* of weight changes that the network tolerates without forfeiting the stability guarantee. This is not to be confused with an *absolute* limit on the size of that network weight.

The third trajectory component reveals some interesting dynamics. This portion of the trajectory stops near the edge of the box (doesn't reach it), and then moves back toward the middle. Keep in mind that this trajectory represents the weight changes in the actor neural network. At the same time as the actor network is learning, the critic network is also learning and adjusting its weights; the critic network is busy forming the value function. It is during this phase in the training that the critic network has started to mature; the "trough" in the critic network has started to form. Because the critic network directs the weight changes for the actor network, the direction of weight changes in the actor network reverses. In the early part of the learning the critic network indicates that "upper left" is a desirable trajectory for weight changes in the actor network. By the time we encounter our third learning phases, the gradient in the critic network has changed to indicate that "upper-left" is now an undesirable direction for movement for the actor network. The actor network has "over-shot" its mark. If the actor network has higher learning rates than the critic network, then the actor network would have continued in that same "upper-left" trajectory, because the critic network would not have been able to learn quickly enough to direct the actor net back in the other direction.

Further dynamics are revealed in the last two phases. Here the actor network weights are not changing as rapidly as they did in the earlier learning phases. We are reaching the point of optimal tracking performance according to the critic network. The point of convergence of the actor network weights is a local optima in the value function of the critic network weights. We halt training at this point because the actor weights have ceased to move and the resulting control function improves performance (minimizes tracking error) over the nominal system.

4.3 Task 2: A Second-Order System

The second task, a second order mass/spring/dampener system, provides a more challenging and more realistic system in which to test our neuro-control techniques. Once again, a single reference input r moves stochastically on the interval $[-1, 1]$; the single output of the control system y must track r as closely as possible. However, there are now

friction, inertial, and spring forces acting on the system to make the task more difficult than Task 1. Figure 14 depicts the different components of the system. We use the same system and block diagram for Task 2 except that we must keep

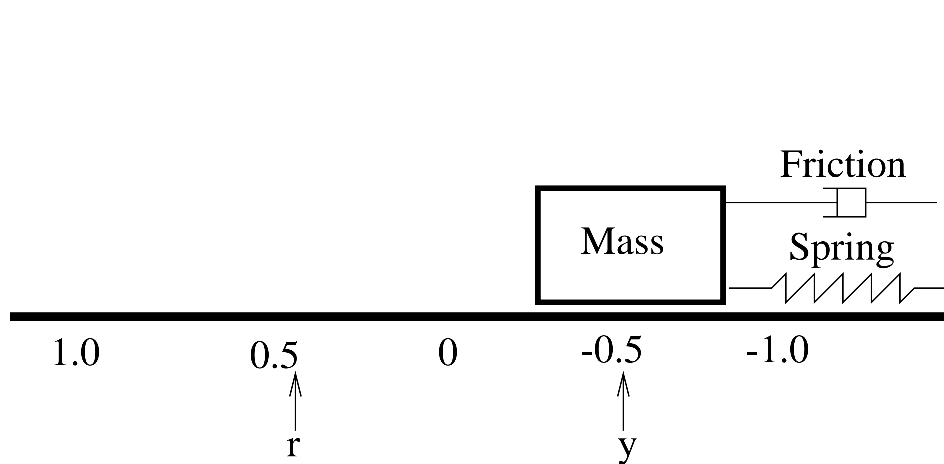

Figure 14: Task 2: Mass, Spring, Dampening System

in mind that the plant now has two internal states (position and velocity) and the controller also now has an internal state. The discrete-time update equations are given by:

$$e_t = r_t - y_t \tag{21}$$

$$u_t = K_p e_t + \int K_i e_t, \text{where } K_p = 0.01 \text{ and } K_i = 0.001 \tag{22}$$

$$x_{t+1}) = \begin{bmatrix} 1 & 0.05 \\ -0.05 & 0.9 \end{bmatrix} x_t + \begin{bmatrix} 0 \\ 1.0 \end{bmatrix} u_t \tag{23}$$

$$y_t = \begin{bmatrix} 1 & 0 \end{bmatrix} x_t \tag{24}$$

Here, the nominal controller is a PI controller with both a *proportional* term and an *integral* term. This controller is implemented with its own internal state variable. The more advanced controller is required in order to provide reasonable nominal control for a system with second-order dynamics as is the case with Task 2. The constant of proportionality, K_p, is 0.01, and the integral constant, K_i, is 0.001. Once again, we have purposely chosen a controller with suboptimal performance so that the neural network has significant margin for improvement.

The neural architecture for the learning agent for Task 2 is identical to that used in Task 1. In practice, three $tanh$ hidden units seemed to provide the fastest learning and best control performance.

Again, for training, the reference input r is changed to a new value on the interval $[-1, 1]$ stochastically with an average period of 20 time steps (every half second of simulated time). Due to the more difficult second-order dynamics, we increase the training time to 10,000 time steps at learning rates of $\alpha = 0.5$ and $\beta = 0.1$ for the critic and actor networks respectively. Then we train for an additional 10,000 steps with learning rates of $\alpha = 0.1$ and $\beta = 0.01$.

In Figure 15, we see the simulation run for this task. The top portion of the diagram depicts the nominal control system (with only the PI controller) while the bottom half shows the same system with both the PI controller and the neuro-controller acting together. The piecewise constant line is the reference input r and the other line is the position of the system. There is a second state variable, velocity, but it is not depicted. Importantly, the K_i and K_p parameters are *suboptimal* so that the neural network has opportunity to improve the control system. As is shown in Figure 15, the addition of the neuro-controller clearly does improve system tracking performance. The total squared tracking error for the nominal system is $SSE = 246.6$ while the total squared tracking error for the neuro-controller is $SSE = 76.3$.

With Task 1, we demonstrated the ability of the neuro-controller to improve control performance. With Task 2, we address the stability concerns of the control system. In Figure 16 we see the Simulink diagram for dynamic stability computations of Task 2 using IQC-analysis. This diagram is necessary for computing the maximum additives, dW

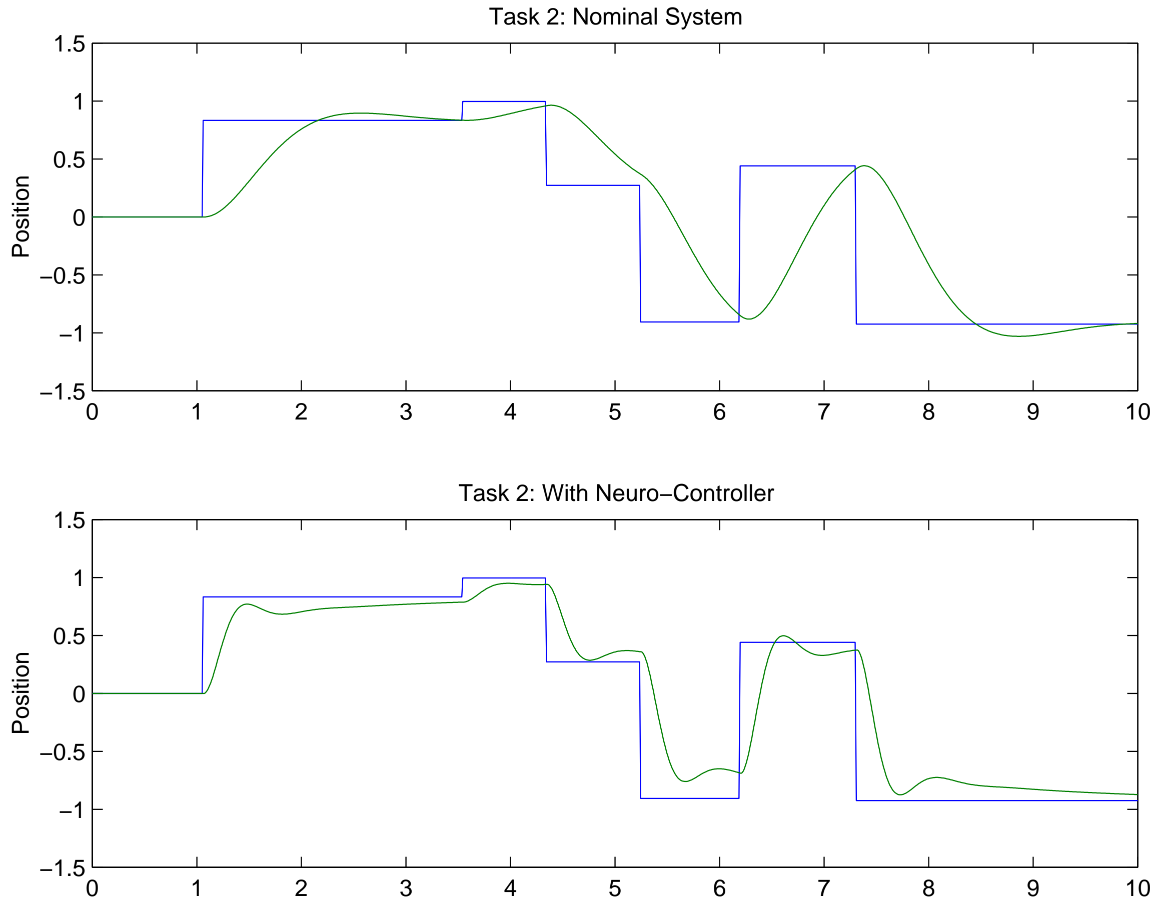

Figure 15: Task 2: Simulation Run

and dV, that can be appended to the actor neural network weights while still retaining stability. These additives are computed anew for each pass through the stability phase. Then, during the learning phase, the actor net is trained via reinforcement learning until one of the weight changes approaches the safety range denoted by the additives. Our IQC analysis is based on the IQC performance block, the IQC odd-slope nonlinearity block and the IQC slowly time-varying block. Using the IQC stability command, the optimizer finds a feasible solution to the constraint problem; thus the system is guaranteed to be stable. The final weights used to produce the simulation diagram in Figure 15 were learned using this IQC-analysis Simulink diagram.

We performed two different training scenarios with Task 2. The first training scenario involved the stable reinforcement learning algorithm with IQC-analysis. In the second training scenario, we trained with only reinforcement learning and no stability analysis. Both training scenarios resulted in similar control performance; they produced similar final weights for the actor network. The bottom half of Figure 15 depicts the stable training episode using IQC-analysis but the other scenario produces almost identical simulation diagrams. However, there is one important difference in the two scenarios. While both scenarios produce a stable controller as an end product (the final neural network weight values), only the stable IQC-analysis scenario retains stability *throughout* the training. The stand-alone reinforcement learning scenario actually produces unstable intermediate neuro-controllers during the learning process!

For the stand-alone reinforcement learning scenario (the one without the dynamic stability guarantees) we demonstrate the actor net's instability at one point during training. Figure 17 depicts a simulation run of Task 2 at an intermediate point during training. The widest-varying signals are the position and velocity variables. The reference signal is the piecewise constant signal, and the other small-magnitude signal is the control signal, u. Notice the scale of the y-axis compared to the stable control diagram of Figure 15. Clearly, the actor net is not implementing a good control

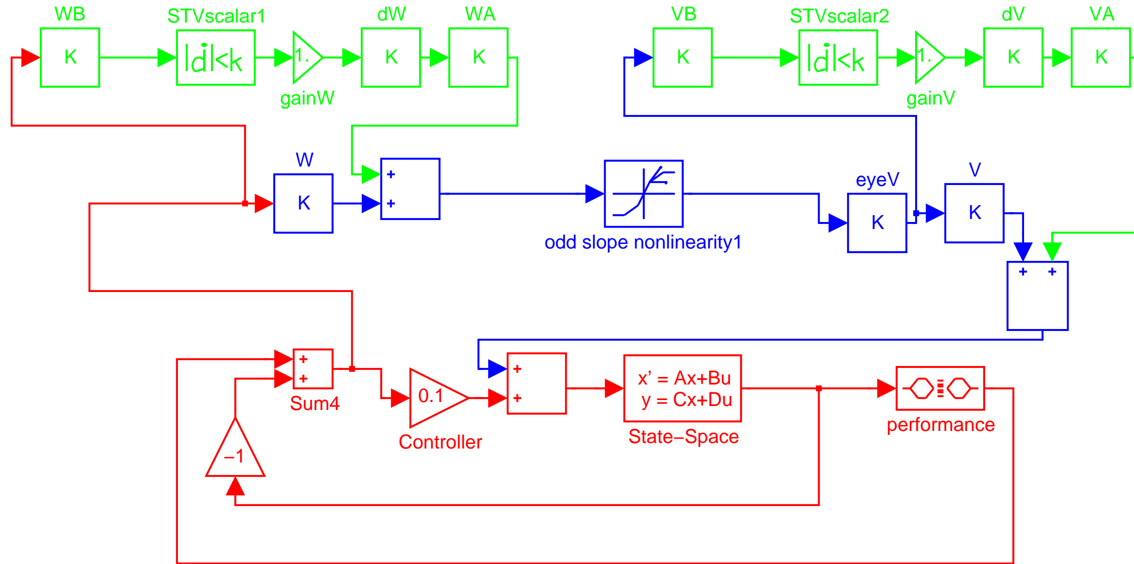

Figure 16: Task 2: Dynamic Stability with IQC-analysis

solution; the system has been placed into an unstable limit cycle, because of the actor network. This is exactly the type of scenario that we must avoid if neuro-controllers are to be useful in industrial control applications. To verify the instability of this system, we use these temporary actor network weights for a static stability test. IQC-analysis is unable to find a feasible solution. Both of these tests indicate that the system is indeed unstable. Again, we restate the requirement of stability guarantees both for the final network (static weights) and the network during training (dynamic weights). It is the stable reinforcement learning algorithm which uniquely provides these guarantees.

In summary, the purpose of Task 2 is to construct a control system with dynamics adequately simple to be amenable to introspection, but also adequately complex to introduce the possibility of learning/implementing unstable controllers. We see in this task that the restrictions imposed on weights from the the dynamic stability analysis are necessary to keep the neuro-control system stable during reinforcement learning.

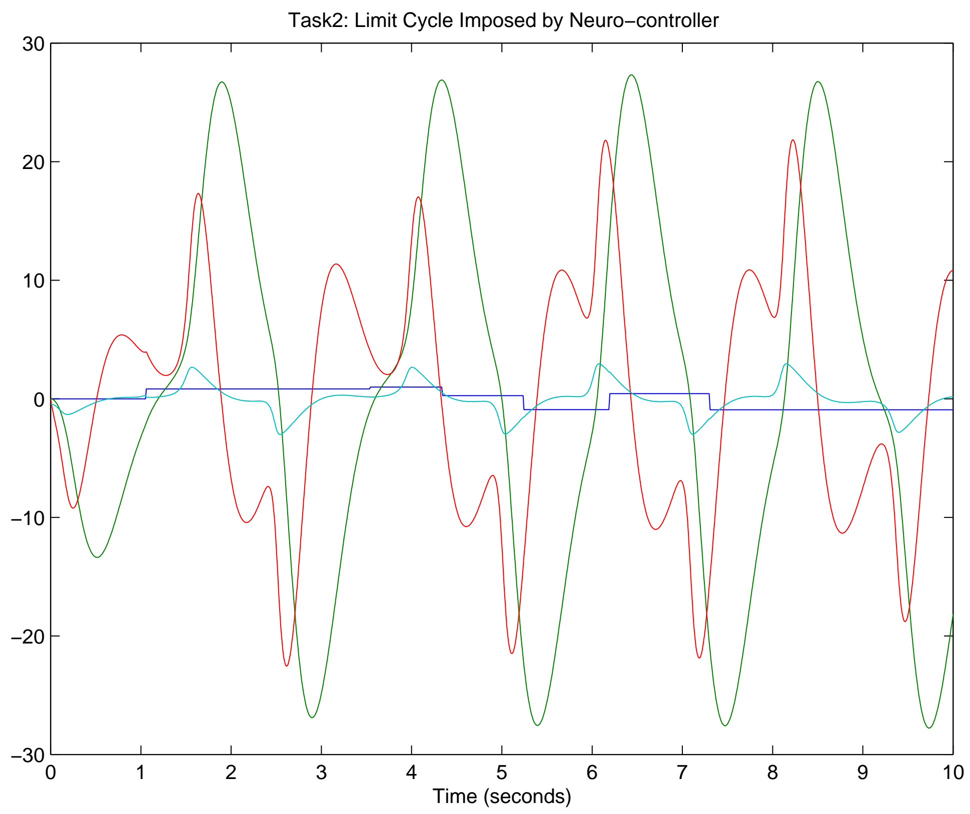

Figure 17: Task 2: Unstable Simulation Run

5 Conclusions

The primary objective of this work is an approach to robust control design and adaptation in which we combine reinforcement learning and robust control theory to implement a learning neuro-controller guaranteed to provide stable control. We discuss how robust control overcomes stability and performance problems in optimal control, due to differences in plant models and physical plants. However, robust control is often overly conservative and thus sacrifices some performance. Neuro-controllers are frequently able to achieve better control than robust designs, because they have nonlinear components and are adaptable on-line. However, neuro-control is not practical for real implementation, because the difficult dynamic analysis is intractable and stability cannot be assured.

We develop a *static stability* test to determine whether a neural network controller, with a specific fixed set of weights, implements a stable control system. While a few previous research efforts have achieved similar results to the static stability test, we also develop a *dynamic stability* test in which the neuro-controller is stable even while the neural network weights are changing during the learning process.

A secondary objective is to demonstrate that our robust reinforcement learning approach is practical to implement in real control situations. Our dynamic stability analysis leads directly to the stable reinforcement learning algorithm. Our algorithm is essentially a repetition of two phases. In the stability phase, we use IQC-analysis to compute the largest amount of weight uncertainty the neuro-controller can tolerate without being unstable. We then use the weight uncertainty in the reinforcement learning phase as a restricted region in which to change the neural network weights.

A non-trivial aspect of our second objective is to develop a suitable learning agent architecture. In this development, we rationalize our choice of the reinforcement learning algorithm, because it is well suited to the type of information available in the control environment. It performs the trial-and-error approach to discovering better controllers, and it naturally optimizes our performance criteria over time. We also design a high-level architecture based upon the actor-critic design in early reinforcement learning. This dual network approach allows the control agent to operate both like a reinforcement learner and also a controller. We applied our robust reinforcement learning algorithm to two tasks. Their simplicity permits a detailed examination of how the stable reinforcement learning algorithm operates.

In spite of the success we demonstrate here, the robust reinforcement learning controller is not without some drawbacks. First, more realistic control tasks with larger state spaces require correspondingly larger neural networks inside the controller. This increases the complexity of the neuro-controller and also increases the amount of training time required of the networks. In real life, the training time on a physical system could be prohibitively expensive as the system must be driven through all of its dynamics multiple times. Second, the robust neuro-controller may not provide control performance which is better than other "easier" design methods. This is likely to be the case in situations where the physical plant and plant model closely match each other or cases in which differences between the model and plant do not greatly affect the dynamics.

In our current work we are extending our robust reinforcement learning procedure to more difficult control problems. In one case we are developing a robust controller for a heating and cooling system, both in simulation and on a physical air duct. Once the robust controller is operating on the air duct, we will add our reinforcement learning algorithm. We expect an improvement in performance due to unknown quantities in the real air duct and also to the fact that nonlinear relationships among the measured variables are known to exist. We are also developing theoretical proofs of static and dynamic stability using IQCs that are specialized to our neural network architectures.

References

[1] C. Anderson, D. Hittle, A. Katz, and R.M. Kretchmar. Reinforcement learning combined with pi control for the control of a heating coil. *Journal of Artificial Intelligence in Engineering*, 1996.

[2] Charles W. Anderson. *Neural Networks for Control*, chapter Challenging Control Problems. Bradford, 1992.

[3] Charles W. Anderson. Q-learning with hidden-unit restarting. In *Advances in Neural Information Processing Systems 5*, pages 81–88, 1993.

[4] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:835–846, 1983. Reprinted in J. A. Anderson and E. Rosenfeld, *Neurocomputing: Foundations of Research*, MIT Press, Cambridge, MA, 1988.

[5] A. G. Barto, R. S. Sutton, and C. Watkins. Learning and sequential decision making. In M. Gabriel and J. Moore, editors, *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, pages 539–602. MIT Press, Cambridge, MA, 1990.

[6] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learting to act using real-time dynamic programming. *Artificial Intelligence: Special Issue on Computational Research on Interaction and Agency*, 72(1):81–138, 1996.

[7] R. E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[8] Robert H. Crites and Andrew G. Barto. Improving elevator performance using reinforcement learning. In *Advances in Neural Information Processing Systems 8*, 1996.

[9] John C. Doyle, Bruce A. Francis, and Allen R. Tannenbaum. *Feedback Control Theory*. Macmillan Publishing Company, 1992.

[10] Pascal Gahihet, Arkadi Nemirovski, Alan J. Laub, and Mahmoud Chilali. *LMI Control Toolbox*. MathWorks Inc., 1995.

[11] M. I. Jordan and R. A. Jacobs. Learning to control an unstable system with forward modeling. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 324–331. Morgan Kaufmann, San Mateo, CA, 1990.

[12] Robert M. Kretchmar and Charles W. Anderson. Comparison of cmacs and radial basis functions for local function approximation in reinforcement learning. In *ICNN'97: Proceedings of the International Conference on Neural Networks*. ICNN, 1997.

[13] Robert M. Kretchmar and Charles W. Anderson. Using temporal neighborhoods to adapt function approximator in reinforcement learning. In *IWANN'99: International Workshop on Artificial Neural Networks*. IWANN, 1999.

[14] Alexandre Megretski, Chung-Yao KAO, Ulf Jonsson, and Anders Rantzer. *A Guide to IQCβ: Software for Robustness Analysis*. MIT / Lund Institute of Technology, http://www.mit.edu/people/ameg/home.html, 1999.

[15] Alexandre Megretski and Anders Rantzer. System analysis via integral quadratic constraints. *IEEE Transactions on Automatic Control*, 42(6):819–830, June 1997.

[16] Alexandre Megretski and Anders Rantzer. System analysis via integral quadratic constraints: Part ii. Technical Report ISRN LUTFD2/TFRT–7559–SE, Lund Institute of Technology, September 1997.

[17] Andrew W. Moore. The parti-game algorithm for variable resolution reinforcement learning in multi-demensional state spaces. *Machine Learning*, 21, 1995.

[18] A. Packard and J. Doyle. The complex structured singular value. *Automatica*, 29(1):71–109, 1993.

[19] P. I. Pavlov. *Conditioned Reflexes*. Oxford University Press, 1927.

[20] D. Rumelhart, G. Hinton, and R. Williams. *Parallel Distributed Processing*, volume 1, chapter Learning internal representations by error propagation. Bradford Books, 1986.

[21] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, 3, 1959.

[22] Satinder Singh and Dimitri Bertsekas. Reinforcement learning for dynamic channel allocation in cellular telephone systems. In *Advances in Neural Information Processing Systems 8*, 1996.

[23] B. F. Skinner. *The Behavior of Organisms*. Appleton-Century, 1938.

[24] Sigurd Skogestad and Ian Postlethwaite. *Multivariable Feedback Control*. John Wiley and Sons, 1996.

[25] R. S. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3:9–44, 1988.

[26] Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, 1996.

[27] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

[28] Johan Suykens and Bart De Moor. Nlq theory: a neural control framework with global assymptotic stability criteria. *Neural Networks*, 10(4), 1997.

[29] G. J. Tesauro. Td-gammon, a self-teaching backgammon program, achieves master level play. *Neural Computation*, 6(2), 1994.

[30] E. L. Thorndike. *Animal Intelligence*. Hafner, 1911.

[31] C.J.C.H. Watkins. *Learning with Delayed Rewards*. PhD thesis, Cambridge University Psychology Department, Cambridge, England, 1989.

[32] Kemin Zhou and John C. Doyle. *Essentials of Robust Control*. Prentice Hall, 1998.