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

October 3, 2001

Abstract

Robust control theory is used to design stable controllers in the presence of uncertainties. This provides powerful closed-loop robustness guarantees, but can result in controllers that are conservative with regard to performance. Here we present an approach to learning a better controller through observing actual controlled behavior. A neural network is placed in parallel with the robust controller and is trained through reinforcement learning to optimize performance over time. By analyzing nonlinear and time-varying aspects of a neural network via uncertainty models, 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 control tasks. Results show that at intermediate stages the system without robust constraints goes through a period of unstable behavior that is avoided when the robust constraints are included.

1 Introduction

Typical controller design techniques are based on a mathematical model that captures as much as possible of what is known about the plant to be controlled, subject to it being representable and tractable in the chosen mathematical framework. Of course the ultimate objective is not to design the best controller for the plant model, but for the real plant. Robust control theory addresses 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 that are guaranteed to bound the 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 powerful tool for practical controller design, but designing a controller that remains stable in the presence of uncertainties limits the aggressiveness of the resulting controller, and can result 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 £nding a policy for choosing sequences of actions that optimize the sum of some performance criterion over time [18]. 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 ¤exibility to allow the reinforcement learning controller to adapt to a good control strategy. This ¤exibility 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, rather than merely degraded performance, and that is unacceptable. Thus, our objectives for the approach

*This work was partially supported by the National Science Foundation through grants CMS-9804757 and 9732986.

described here are twofold. The main objective that must always be satis£ed is stable behavior, both static and dynamic stability. *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. The second objective is for the reinforcement learning component to optimize the controller behavior on the true plant, while never violating the main objective.

To solve the static stability problem, we must ensure that the neural network with a £xed 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 [9] 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 alternative approaches. In particular, static stability solutions are afforded by the NLq research of Suykens and DeMoor [19], and also by various researchers using Lyapunov-based approaches [13, 4]. 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. The use of the IQC framework affords us great ¤exibility in specifying all that is known about the uncertainties, to deliver analysis tools that are as non-conservative as possible. Moreover, we are able to extend the tools developed using our approach to the dynamic stability problem (see below).

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 signi£cant contribution of this article is this 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 speci£c 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 which is necessary for robust analysis. The architecture also utilizes a discrete value function to mitigate dif£culties speci£c to training in control situations.

The remainder of this article describes our approach and demonstrates its use on two control problems. Section 2 provides an overview of reinforcement learning and the actor-critic architecture. Section 3 summarizes our use of IQC's 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 tracking tasks. We £nd 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 5 we summarize our conclusions and discuss current and future work.

2 Reinforcement Learning

A 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.

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. A signi£cant advance in the £eld of reinforcement learning is the Q-learning algorithm of Chris Watkins [21]. 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., [3], 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 [3].

To de£ne the Q-learning algorithm, we start by representing a system to be controlled as consisting of a discrete state space, S, and a £nite set of actions, A, that can be taken in all states. A *policy* is de£ned 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 the random variable $R(s_t, a_t)$. $Q_\pi(s_t, a_t)$ 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

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

where γ is a discount factor between 0 and 1 that weights reinforcement received sooner more heavily than reinforcement received later.

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

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

This is what has become known as the one-step *Q-learning* algorithm. Watkins [21] 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 satis£es the stochastic approximation conditions $\sum \alpha_t = \infty$ and $\sum \alpha_t^2 < \infty$. It also assumes that every state and action are visited in£nitely often.

The Q function implicitly de£nes the policy, π, de£ned 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 suf£ciently 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 de£ned the SARSA algorithm by removing the $\min$ operator, thus using the same policy for generating behavior and for training the value function [18]. In Section 4, we use SARSA as the reinforcement learning component of our experiments. Convergence of SARSA and related algorithms has been proved for tabular and linear approximators of the Q function [20], but not for nonlinear neural networks commonly used in practice and that we use in our experiments. However, our approach to stability analysis results in bounds on the neural network weight values guaranteeing that the weights do not diverge.

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 [7] and is very closely related to the actor-critic architecture of Barto, et al., [2] in their actor-critic architecture.

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 £xed, 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 (it's impact will be effectively absorbed into the time-varying uncertainty in the weight updates). 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 dif£cult 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 con£ned 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 [17, 22, 5] for examples.

3.2 IQC Stability

Integral Quadratic Constraints (IQC's) 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 [9, 10, 8] for a thorough treatment of IQCs.

First we present a very brief overview of the main concepts. This material is taken from [10], where the interested reader may £nd a more detailed exposition. Consider the feedback interconnection shown in Figure 1. The upper block, M, is a known Linear-Time-Invariant (LTI) system, and the lower block, Δ is a (block-diagonal) structured uncertainty. An *Integral Quadratic Constraint* (IQC) is an inequality describing the relationship between two signals,

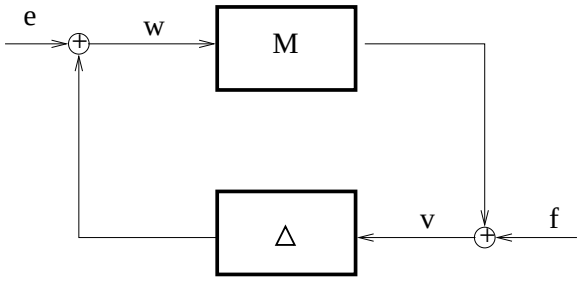


Figure 1: Feedback System

w and v, characterized by a Hermitian matrix function Π as:

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

where $\hat{v}$ and $\hat{w}$ are the Fourier Transforms of $v(t)$ and $w(t)$. The basic IQC stability theorem can be stated as follows.

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

$$v = Mw + f \tag{3}$$
$$w = \Delta(v) + e \tag{4}$$

Assume that:

- $M(s)$ *is a stable, proper, real-rational transfer matrix, and* Δ *is a bounded, causal operator.*

- *The interconnection of M and $\tau\Delta$ is well-posed for all $\tau \in [0,1]$. (i.e., the map from $(v,w) \to (e,f)$ has a causal inverse)*
- *The IQC de£ned by Π is satis£ed by $\tau\Delta$ for all $\tau \in [0,1]$.*
- *There exists an $\epsilon > 0$ such that for all ω:*

$$\left| \begin{array}{c} M(j\omega) \\ I \end{array} \right|^* \Pi(j\omega) \left| \begin{array}{c} M(j\omega) \\ I \end{array} \right| \leq -\epsilon I \tag{5}$$

Then the feedback interconnection of M and Δ is stable.

The power of this IQC result lies in both its generality and its computability. First we note that many system interconnections can be rearranged into the canonical form of Figure 1 (see [11] 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 [9, 10]. Moreover, a library of IQCs for common uncertainties is available [8], and more complex IQCs can be built by combining the basic IQCs.

Furthermore, the computation involved to meet the requirements of the theorem is tractable, since the theorem requirements can be transformed into a Linear Matrix Inequality (LMI) as follows. Suppose that we parameterize the IQC's that cover Δ, and hence are candidates to satisfy theorem 1, as:

$$\Pi(j\omega) = \sum_{i=1}^{n} p_i \Pi_i(j\omega) \tag{6}$$

where p_i are positive real parameters. Then we can bring in State Space realizations of M and Π_i to write the IQC components as:

$$\left| \begin{array}{c} M(j\omega) \\ I \end{array} \right|^* \Pi_i(j\omega) \left| \begin{array}{c} M(j\omega) \\ I \end{array} \right| \quad = \quad \left| \begin{array}{c} (j\omega I - A)^{-1}B \\ I \end{array} \right|^* P_i \left| \begin{array}{c} (j\omega I - A)^{-1}B \\ I \end{array} \right| \tag{7}$$

where A is a Hurwitz matrix and P_i are real symmetric matrices. It follows from the Kalman-Yacubovich-Popov (KYP) lemma [14] that this is equivalent to the existence of a symmetric matrix Q such that

$$\left[\begin{array}{cc} QA + A^TQ & QB \\ B^TQ & 0 \end{array} \right] + \sum_{i=1}^{n} p_i P_i \quad < \quad 0 \tag{8}$$

which is a £nite-dimensional LMI feasibility problem in the variables p_i and Q. As is well known, LMIs are convex optimization problems for which there exist fast, commercially available, polynomial time algorithms [6]. In fact there is now a beta-version of a Matlab IQC toolbox available at http://web.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 ef£cient 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 £xed neural network, i.e., a network with all weights held constant. This test guarantees to identify 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, the approach provides a guarantee of the system's stability while the neural network is training. Note however that these tests are not exact, and may potentially be conservative, i.e., it is possible to fail the test even if the controller is stabilizing. Of course, in the worst case this means we may be more cautious than necessary, but we are always guaranteed to be safe.

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 £xed 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_{hxn}, and output weight matrix V_{mxh}, where the bias terms are included as £xed 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

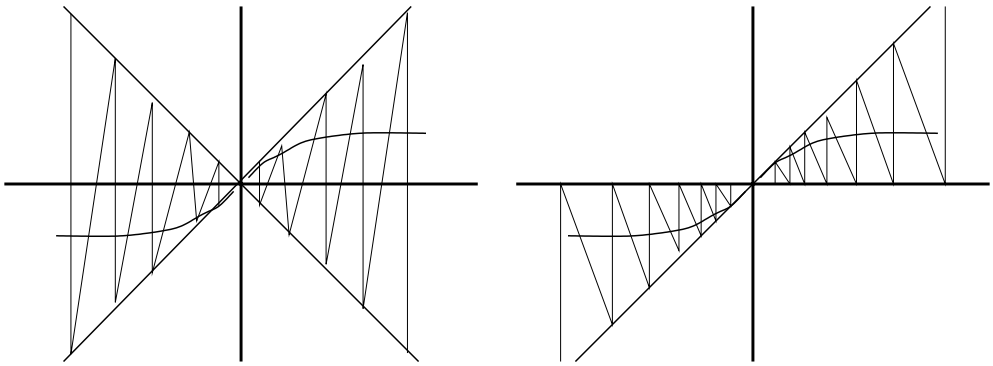
$$\Phi = Wx, \tag{9}$$

$$a = V \tanh(\Phi). \tag{10}$$

With moderate rearrangement, we can rewrite the vector notation expression in (9,10) as

$$\begin{aligned} \Phi &= Wx, \\ \gamma_j &= \begin{cases} \frac{\tanh(\phi_j)}{\phi_j}, & \text{if } \phi_j \neq 0; \\ 1, & \text{if } \phi_j = 0, \end{cases} \\ \Gamma &= \mathsf{diag}\{\gamma_j\}, \\ a &= V\Gamma\Phi. \end{aligned} \tag{11}$$

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 2.



a. tanh in [-1,1] b. improved sector, tanh in [0,1]

Figure 2: Sector bounds on $\tanh$

Equation 11 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 11 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 £nd an appropriate IQC to cover the nonlinearity in the neural network hidden layer. From Equation 11, 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

$$w(t) \;=\; \gamma v(t) \;=\left(\frac{\tanh(v(t))}{v(t)}\right) v(t) \;=\; \tanh(v(t)) \tag{12}$$

(for input signal $v(t)$ and output signal $w(t)$). In IQC terms, this nonlinearity is referred to as a *bounded odd slope nonlinearity*. There is an Integral Quadratic Constraint already con£gured 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 $w(t) = \psi(v(t))$ where ψ is a static nonlinearity satisfying (see [8]):

$$\psi(-v) = -\psi(v), \tag{13}$$

$$\alpha(v_1 - v_2)^2 \le (\psi(v_1) - \psi(v_2))(v_1 - v_2) \le \beta(v_1 - v_2)^2. \tag{14}$$

For our speci£c network, we choose $\alpha = 0$ and $\beta = 1$. Note that each nonlinear hidden unit function ($\tanh(v)$) satis£es the odd condition, namely:

$$\tanh(-v) = -\tanh(v) \tag{15}$$

and furthermore the bounded slope condition

$$0 \;\le\; (\tanh(v_1) - \tanh(v_2))(v_1 - v_2) \;\le\; (v_1 - v_2)^2 \tag{16}$$

is equivalent to (assuming without loss of generality that $v_1 > v_2$)

$$0 \;\le\; (\tanh(v_1) - \tanh(v_2)) \;\le\; (v_1 - v_2) \tag{17}$$

which is clearly satis£ed by the $\tanh$ function since it has bounded slope between 0 and 1 (see Figure 2). Hence the hidden unit function is covered by the IQCs describing the bounded odd slope nonlinearity (13,14) [8], specialized to our problem, namely:

$$\Pi(j\omega) = \begin{bmatrix} 0 & 1 + \frac{p}{j\omega+1} \\ 1 + \frac{p}{-j\omega+1} & -2\left(1 + \mathrm{Re}\left(\frac{p}{j\omega+1}\right)\right) \end{bmatrix} \tag{18}$$

with the additional constraint on the (otherwise free) parameter p that $|p| \le 1$ (which is trivially reformulated as another IQC constraint on p). Note that this is the actual IQC we used for analysis, and it is based on a scaling of $H(s) = \frac{1}{s+1}$, but one can attempt to get more accuracy (at the expense of increased computation) by using a more general scaling $H(s)$ (in fact it can be any transfer function whose $\mathcal{L}_1$ norm does not exceed one - see [9]).

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 have available a suitable IQC from [9]. 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 $w(t) = \psi(t)v(t)$, where the gain $\psi(t)$ satis£es (see [9]):

$$|\psi(t)| \le \beta, \tag{19}$$

$$|\dot{\psi}(t)| \le \alpha, \tag{20}$$

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. Having determined α and β, the corresponding IQC's specialized to our problem can be stated as follows:

$$\int_{-\infty}^{\infty} \left| \begin{array}{c} \hat{v}_{\rm ext}(j\omega) \\ \hat{w}_{\rm ext}(j\omega) \end{array} \right|^* \left[\begin{array}{cc} \beta^2 K_1 & M_1 \\ M_1^T & -K_1 \end{array} \right] \left| \begin{array}{c} \hat{v}_{\rm ext}(j\omega) \\ \hat{w}_{\rm ext}(j\omega) \end{array} \right| d\omega \geq 0 \tag{21}$$

and also

$$\int_{-\infty}^{\infty} \left| \begin{array}{c} \hat{y}(j\omega) \\ \hat{u}(j\omega) \end{array} \right|^* \left[\begin{array}{cc} \alpha^2 K_2 & M_2 \\ M_2^T & -K_2 \end{array} \right] \left| \begin{array}{c} \hat{y}(j\omega) \\ \hat{u}(j\omega) \end{array} \right| d\omega \geq 0 \tag{22}$$

where the free parameters K_1, K_2, M_1, M_2 are subject to the additional (IQC) constraints that K_1, K_2 are symmetric positive de£nite matrices, and M_1, M_2 are skew-symmetric matrices. The signals $v_{\rm ext}, w_{\rm ext}$ are de£ned in terms of v, w and an additional (free) signal u as:

$$\hat{v}_{\rm ext}(s) = \left[\begin{array}{c} \frac{\hat{v}(s)}{s+1} \\ \hat{v}(s) \end{array} \right] \qquad\qquad \hat{w}_{\rm ext}(s) = \left[\begin{array}{c} \frac{\hat{w}(s)}{s+1} + \frac{\hat{u}(s)}{s+1} \\ \hat{w}(s) \end{array} \right] \tag{23}$$

Note again that this is the actual IQC we used for analysis, but in fact there are free scaling parameters in this IQC which we have simply assigned as $\frac{1}{s+1}$. A more general statement of this IQC (with more general scalings) can be found in [8].

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 de£nes. For the static stability procedure, we assume the network weights are held constant (i.e., training has been completed). The procedure consists of the following steps:

1. Design the nominal, robust LTI controller for the given plant model so that this nominal closed-loop 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 in (18) to cover the nonlinear part of the neural network. We refer to this as the testable version of the neuro-controller.
4. Apply the IQC stability analysis result from theorem 1, with the computation tools summarized in equations (6-8), to reduce to a (convex) LMI feasibility problem. If a feasible solution to this problem is found, then the testable version of the neuro-control system is robustly stable, and hence the overall closed-loop 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, which provides a stability guarantee *during learning*. The £rst 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 closed-loop 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 in (18) 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 described in equations (21-23), to the testable version, to cover the time-varying weights in the neural network.

5. Commence training 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.

6. Apply the IQC stability analysis result from theorem 1, with the computation tools summarized in equations (6-8), to reduce to a (convex) LMI optimization problem to £nd bounds on the perturbations of the current neural network weight values within which the system is guaranteed to be stable. This de£nes a known "stable region" of weight values in which it is safe to learn at the speci£ed rate.

7. Continue training until any of the weights approaches its bounds for the corresponding the stable region, at which point repeat the previous step. Adjust learning rates/bounds as necessary and 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 £rst 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 a simple control task and a realistic task involving the control of a distillation column.

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 de£ned to be the magnitude of the error between the reference signal and the plant output that we wish to track the reference signal.

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 £rst 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 [15].

Figure 3 places the actor-critic network within the control framework. Note that the neural network sits in feedforward around the nominal controller, and hence in feedback around the plant. As such the network augments the action of the nominal feedback controller, so as to deliver the potential for enhanced performance, because of its capability for nonlinear control and adaptation. Note further that this architecture starts the learning controller from the nominal solution, which utilized our a-priori information about the plant. This means that zero output from the learning controller delivers the nominal performance, so that learning is intended to improve *from this point*. In addition to providing a stable control loop on startup, this serves to cut down on the required learning experience, and hence also to minimize the size of uncertainties required to cover the learning process in our robust stability analysis tools.

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 4 depicts each component. The details for each component are presented here, some of which are repeated from 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 input. Let h be the number of hidden units in the actor network. The best value for h is determined experimentally.

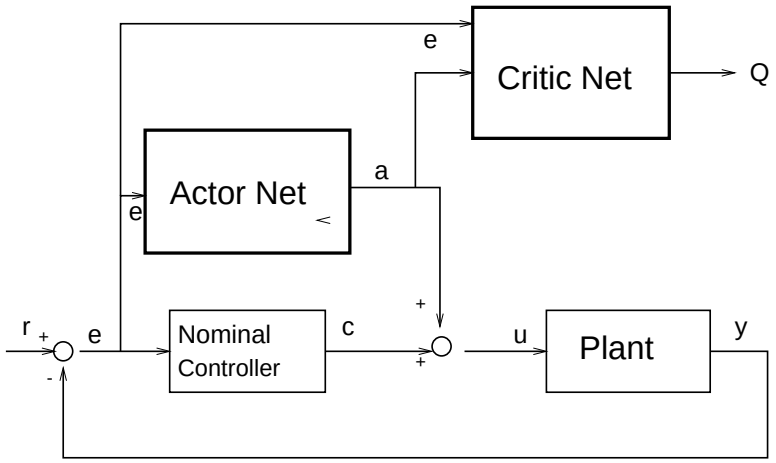


Figure 3: Learning in the Actor-Critic

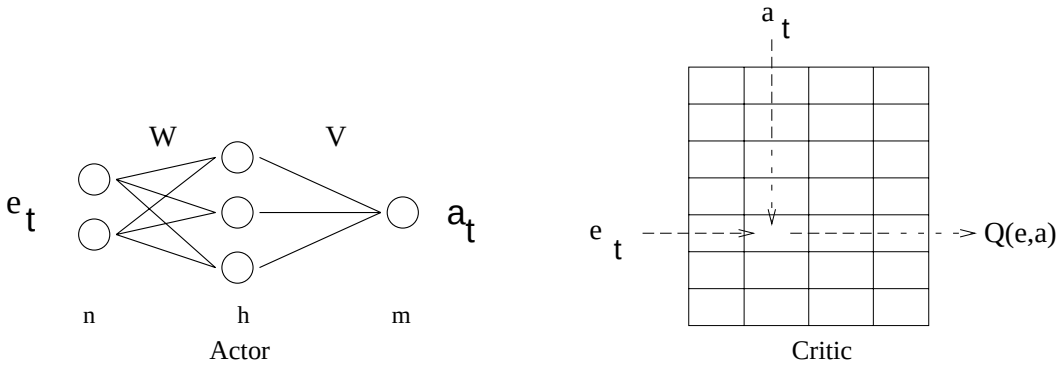


Figure 4: Network Architectures

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 de£ned to increment by one as signals pass through the plant.

The de£nition 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 £xed, feedback control law, f, to input e_t, and sum the output of the £xed 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.

$$\begin{aligned} c_t &= f(e_t) \\ u_t &= c_t + a_t \\ y_{t+1} &= g(u_t) \end{aligned}$$

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 de£ne 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 £rst 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 de£ned as

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

for which the estimate of the optimal action is given by

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

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 £rst, 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. Note that initial values for W and V are random variables from a Gaussian distribution with mean zero and variance of 0.1. The stable region S is always a rectangle in the multi-dimensional weight space and is initially centered at zero with size determined by an iterative expanding search involving small increases to the size and a corresponding IQC analysis to determine stability until a maximum size is reached or instability is determined. After calculating changes to V and W, if the new weight values fall within S, 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 de£ne an initial guess at allowed weight perturbations, P, as factors of the current weights. De£ne 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.

$$\begin{aligned} z_u &= 1 \\ z_s &= 1 \end{aligned}$$

Increase z_u until system is unstable:

$$\begin{aligned} &\text{If stable for } N \pm P \cdot N, \\ &\quad \text{then while stable for } N \pm z_u P \cdot N, \text{ do} \\ &\qquad z_u = 2 z_u \end{aligned}$$

Decrease z_s until system is stable:

$$\begin{aligned} &\text{If not stable for } N \pm P \cdot N, \\ &\quad \text{then while not stable for } N \pm z_s P \cdot N \text{ do} \\ &\qquad z_s = \frac{1}{2} z_s \end{aligned}$$

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

$$\begin{aligned} &\text{While } \frac{z_u - z_s}{z_s} < 0.05 \text{ do} \\ &\quad z_m = \frac{z_u + z_s}{2} \\ &\quad \text{If not stable for } N \pm z_m P \cdot N \\ &\qquad \text{then } z_s = z_m \\ &\qquad \text{else } \ z_u = z_m \end{aligned}$$

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

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

4.2 Experiments

We now demonstrate our robust reinforcement learning algorithm on one simple control task (for pedagogical reasons) and one more complex, realistic task. The £rst task is a very simple £rst-order positioning control system. The second task involves a challenging control problem on a distillation column [17]. The distillation column task is MIMO (multi-input, multi-output) with an ill conditioned plant, resulting in high sensitivity to modeling errors that make this task a dif£cult control problem.

4.2.1 Task 1: A Simple Position Control Problem

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 £rst 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:

$$x_{t+1} = x_t + u_t \tag{24}$$

$$y_t = x_t \tag{25}$$

where t is the discrete time step for which we used 0.01 seconds. We implement a simple proportional controller with $K_p = 0.1$.

$$e_t = r_t - y_t \tag{26}$$

$$u_t = 0.1\, e_t \tag{27}$$

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 3.

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 £rst experiment, we summarize how the IQC approach to stability is adapted to the system for Task 1. Our computation for IQC analysis is based on Matlab and Simulink. Figure 5 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.

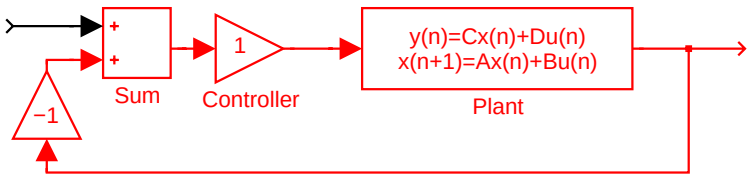


Figure 5: Task 1: Nominal System

Next, we add the neural network controller to the diagram. Figure 6 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

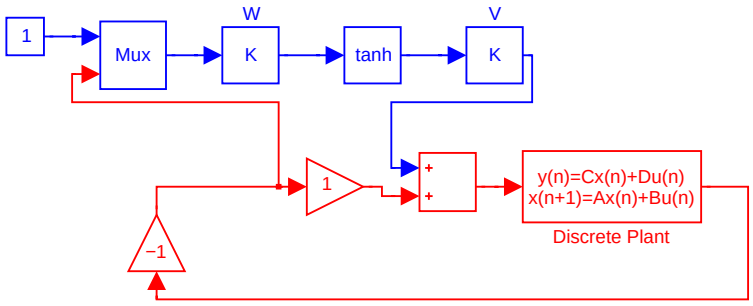


Figure 6: Task 1: With Neuro-Controller

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 (it merely in¤uences the (speed of) adaptation of the weights).

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 7. The *performance* block is another IQC block merely used to trigger the analysis for our problem.

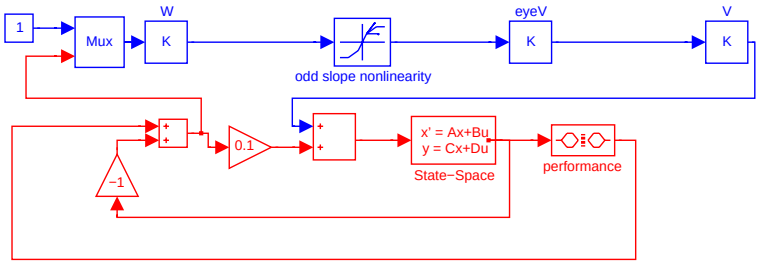


Figure 7: Task 1: With Neuro-Controller as LTI (IQC)

Again, we emphasize that there are *two* versions of the neuro-controller. In the £rst version, shown in Figure 6, 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 7, contains the neural network converted into the IQC robustness analysis framework. Applying our static stability procedure to the system depicted in Figure 7 involves solving a (convex) LMI feasibility problem (as described earlier). We £nd that the resulting feasibility constraints are easily satis£ed; the neuro-controller is guaranteed to be stable.

This approach provides a stability guarantee for the closed-loop system once the network has converged. 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 8. The matrices dW and dV are the perturbation matrices. Note that the matrices WA, WB, VA, and VB are simply there to cast the

uncertainty into standard block-diagonal form.

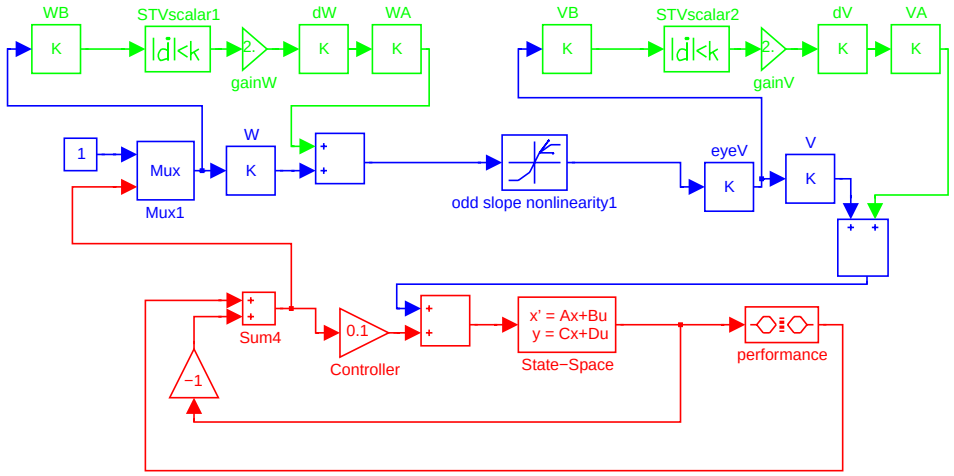


Figure 8: Task 1: Simulink Diagram for Dynamic IQC-analysis

If analysis of the system in Figure 8 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 as we change the neural network weight values, so long as the new weight values do not exceed the *range* (in both magnitude and learning rate) speci£ed 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 boundary of the safe range computed via IQC analysis.

We trained the neural network controller as described in earlier in this section, and a time-series plot of the simulated system from is shown in Figure 9. The top diagram shows the system with only the nominal, proportional controller, corresponding to the Simulink diagram in Figure 5. The bottom diagram shows the same system with both the proportional controller and the neuro-controller as speci£ed in Figure 6. 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. Of course, with this simple £rst-order system it is not dif£cult to construct a better performing (proportional) controller. We have purposely chosen a suboptimal controller for demonstration purposes 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 tracking errors.

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 10 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 9.

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 the boundary of its safe (stable) range computed in the stability phase. 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 $tanh$ hidden units speci£ed earlier, we use one hidden unit *for this subsection only*. Thus, the actor network has

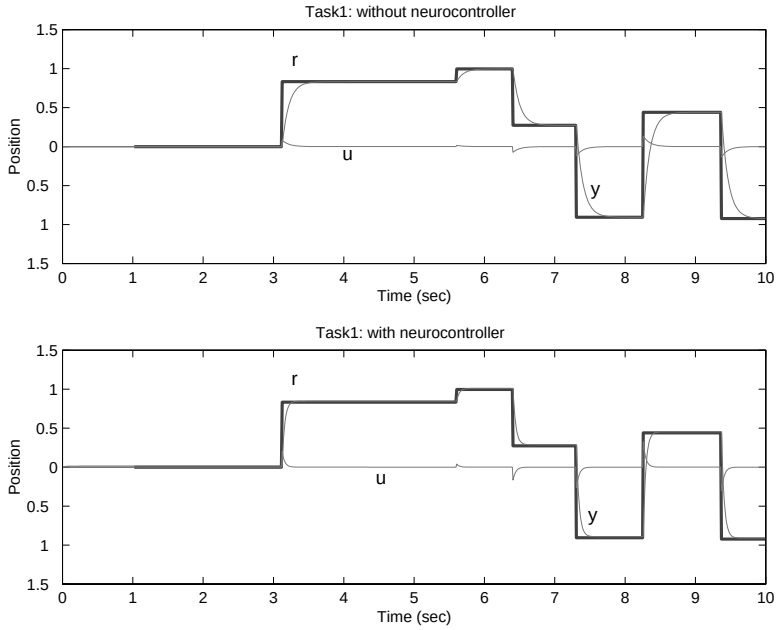


Figure 9: Task 1: Simulation Run

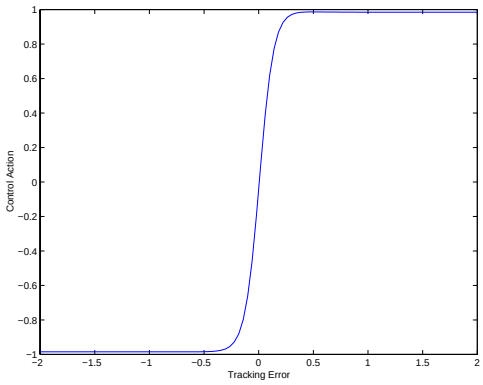


Figure 10: Task 1: Actor Net's Control Function

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 10 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 £rst 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 £rst input to the network and let the system tracking error, e, be the second input). From a stability standpoint, W_1 is insigni£cant. Because the bias input is clamped at a constant value of 1, there really is no "magni£cation" 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 11 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.

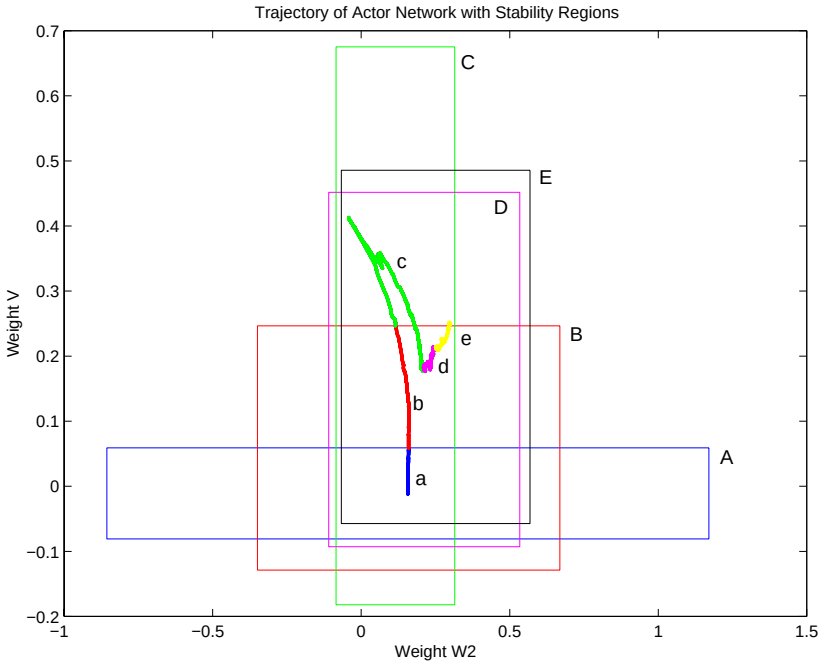


Figure 11: Task 1: Weight Update Trajectory with Bounding Boxes

The shades represent different phases of the learning algorithm, (a)-(e), with corresponding stability regions, A-E. 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. We depict these safe ranges as rectangular boxes in our two-dimensional trajectory plot shown in Figure 11, and the £rst is shown as box A. The black part (a) of the trajectory represents the learning that occurred for the £rst values of dW and dV. After the £rst learning phase, we then perform another stability phase to compute new values for dW and dV, shown by box B. We then enter a second learning phase (b) that proceeds until we attempt a weight update exceeding the allowed range. This process of alternating stability and learning phases repeats until we are satis£ed that the neural network is fully trained. In the diagram of Figure 11 we see a total of £ve learning phases ((a)-(e)).

Again, there are £ve different bounding boxes A-E corresponding to the £ve different stability/learning phases. As can be seen from the black trajectory (a) in this diagram, training progresses until the V weight reaches the edge of bounding box A. At this point we must cease our current reinforcement learning phase, because any additional weight changes might result in an unstable control system (actually, the system might still be stable but we are no longer *guaranteed* of the system's stability). At this point, we recompute a new bounding box B using a second stability phase; then we proceed with the second learning phase (b) until the weights almost 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. We may be able to compute a more accurate stability region, by adjusting the center and aspect ratio of the box (and then recomputing the maximum box size with these new parameters), or we may adjust the algorithm learning rate, or some other modi£cation (of course it may also be that this is truly a region

of instability and the algorithm simply prevents you from going there). In our case we adjusted learning rates to allow the critic more time to get better estimate of the value function, as well as recomputing the box with an updated center and aspect ratio.

The third trajectory component (c) reveals some interesting dynamics. This portion of the trajectory stops near the edge of box C (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 (c), 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 (d),(e). 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 optimum 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 MIMO System

The material in sections 4.3.1-4.3.3 is taken from [17]. We £rst brie¤y discuss the dynamics of the distillation column process from [16] and show why it is a dif£cult control problem. We then overview two designs presented for this system by Skogestad et al [16] (see also [17] for a tutorial overview). The £rst design is a decoupling controller, which delivers excellent nominal performance, but fails badly for small perturbations to the system. The second design is a robust controller which addresses the shortcomings of the decoupling controller with regard to perturbations, but necessarily sacri£ces performance to do it. Finally, in section 4.3.4 we apply our stable reinforcement learning controller on top of the robust controller; the reinforcement learner is able to regain some of the lost performance margin, while still retaining the desirable robustness properties.

4.3.1 Plant Dynamics

Figure 12 is Skogestad's depiction of the distillation column [17]. Without concerning ourselves with the rather involved chemistry, we brie¤y summarize the dynamics of the distillation column. The two output variables, y_1 and y_2, are the concentrations of chemical A and chemical B, respectively. We control these concentrations by adjusting two ¤ow parameters: u_1 = L ¤ow and u_2 = V ¤ow. The reference inputs, r_1 and r_2, and the outputs are scaled so that $r_1, r_2, y_1, y_2 \in [0, 1]$. The standard feedback con£guration in Figure 13 is considered, with the following 2×2 LTI model for the plant G:

$$G(s) = \frac{1}{75s+1} \begin{pmatrix} 87.8 & -86.4 \\ 108.2 & -109.6 \end{pmatrix} \tag{28}$$

Since we implement the neuro-controller using a digital system, we approximate Skogestad's continuous-time plant given above with the following discrete-time, state space system [17, 12]:

$$x(k+1) = Ax(k) + Bu(k) \tag{29}$$

$$y(k) = Cx(k) + Du(k) \tag{30}$$

where

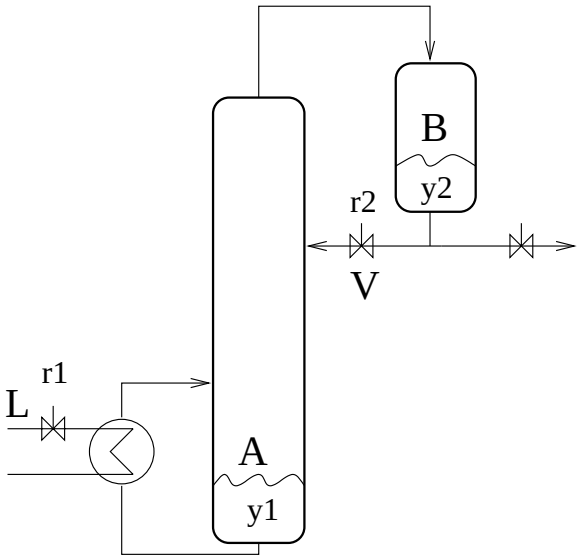


Figure 12: Distillation Column Process

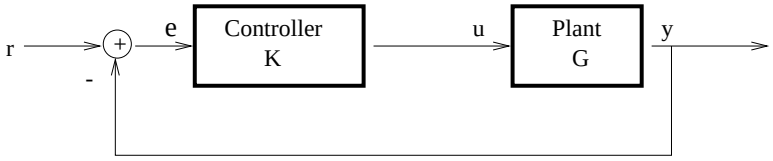


Figure 13: Distillation Column Process: Block Diagram

$$A = \begin{pmatrix} 0.99867 & 0 \\ 0 & 0.99867 \end{pmatrix} \quad B = \begin{pmatrix} -1.01315 & 0.99700 \\ -1.24855 & 1.26471 \end{pmatrix}$$
$$C = \begin{pmatrix} -0.11547 & 0 \\ 0 & -0.11547 \end{pmatrix} \quad D = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \tag{31}$$

The sampling interval, k, is one second. In order to see why this is a dif£cult control problem, Skogestad computes the singular value decomposition of the plant, G (ignoring the $\frac{1}{75s+1}$ term which imparts no directional information)

$$\begin{pmatrix} 87.8 & -86.4 \\ 108.2 & -109.6 \end{pmatrix} = \begin{pmatrix} 0.625 & -0.781 \\ 0.781 & 0.625 \end{pmatrix} \begin{pmatrix} 197.2 & 0 \\ 0 & 1.39 \end{pmatrix} \begin{pmatrix} 0.707 & -0.708 \\ -0.708 & -0.707 \end{pmatrix} \tag{32}$$

From the SVD, Skogestad points out that inputs aligned in opposite directions ($[0.707, -0.708]^T$) produce a large response in the outputs (indicated by singular value of 197.2). Conversely, inputs aligned in the same direction ($[-0.708, -0.707]^T$) produce a minimal response in the output (singular value = 1.39). This plant is *ill conditioned* in that it is highly sensitive to changes in individual inputs, but relatively insensitive to changes in both inputs.

To design a robust controller, we must incorporate uncertainty into the plant model so that the model covers the dynamics of the physical plant. Multiplicative uncertainty is incorporated to each input control path, u_1 and u_2, in the amount of $\pm$ 20% gain (which is a fairly typical uncertainty level for a practical distillation column). Figure 14 shows the system with 20% gain uncertainty on each input.

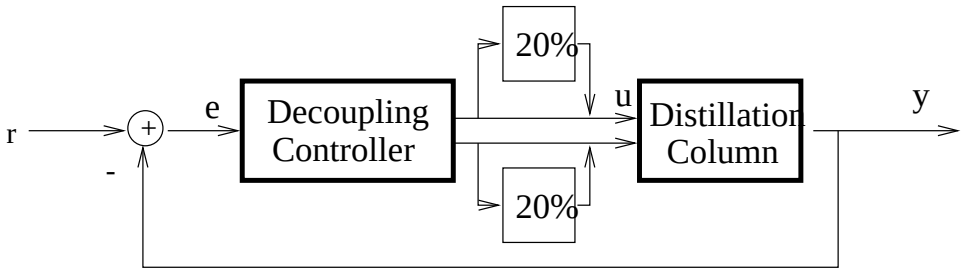


Figure 14: Distillation Column Model with Input Gain Uncertainty

4.3.2 Decoupling Controller

A decoupling controller simply inverts the directional component of plant in attempt to have input u_1 affect only output y_1 and input u_2 affect only output y_2. A cancellation controller is then implemented to swap the current plant dynamics for desired plant dynamics. This yield the following controller:

$$K_{decoup}(s) = \frac{0.7}{s} G^{-1}(s) = \frac{0.7(1+75s)}{s} \begin{pmatrix} 0.3994 & -0.3149 \\ 0.3943 & -0.3200 \end{pmatrix} \tag{33}$$

Figure 15 shows the nominal plant response to a step change in one of the inputs. As seen in the diagram, the output y_1, quickly rises to track the step change in the input $u_1 = 0 \rightarrow 1$. Output y_2 has been "decoupled" from input u_1 and thus remains constant at $y_2 = 0$. The performance of the decoupling controller on the nominal plant is excellent.

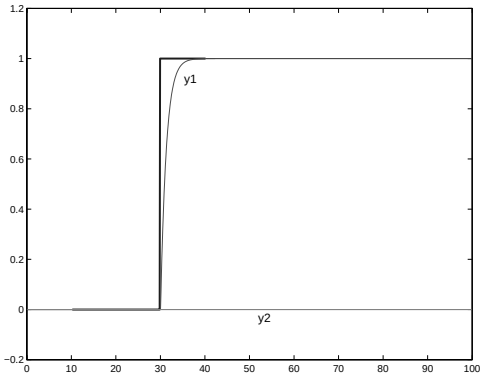


Figure 15: Step Response: LTI Model with Decoupling Controller

Figure 16 depicts the step response of the decoupling controller on a perturbed plant, where we amplify input u_1 by 20% while decreasing input u_2 by 20%. It is readily seen that the tracking performance of the decoupling controller on the perturbed plant is very poor. Output y_1 rises to 6.5 before decaying back to its desired value of 1.0, and even worse, output y_2 goes past 7.0 before dropping back to 0.

The poor performance of the decoupling controller on the real plant is a result of it being highly tuned to the dynamics of the LTI model. It exploits the model's dynamics in order to achieve high performance. Even though the dynamics of the perturbed plant are close (20% uncertainty on the inputs is not an unreasonable amount), the decoupling controller's performance on the perturbed plant is very poor, and it is clear that robust performance is not achieved.

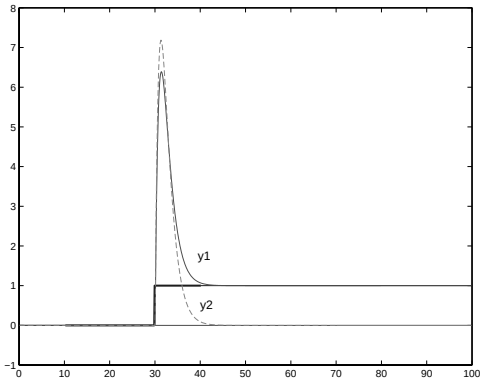


Figure 16: Step Response: Physical Plant with Decoupling Controller

4.3.3 Robust Controller

To address the problems encountered when the decoupling controller is implemented on the physical plant, Skogestad designs a robust controller [17] via μ-synthesis, using the Matlab μ-synthesis toolbox [1, 17]. The resulting robust controller is eighth order. Using μ analysis one can provide robust performance guarantees for this design, and indeed on simulating we see that good performance is achieved for the nominal model (Figure 17) and the perturbed plant (Figure 18).

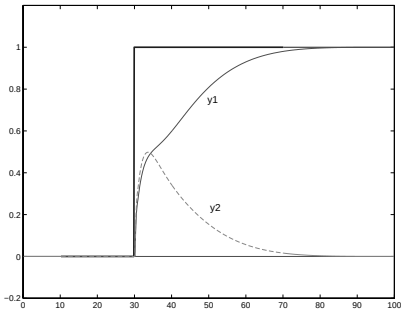


Figure 17: Step Response: LTI Model with Robust Controller

It is important to note that the robust controller does not match the performance of the decoupling controller on the nominal model. Once again, this is because the decoupling controller fully exploits the dynamics of the plant model to achieve this high performance. The robust controller is "prohibited" from exploiting these dynamics by the uncertainty built into the model. Thus, a robust controller will not perform as well as a controller (optimally) designed for one particular plant. However, the robust controller will perform fairly well for a general class of plants which possess similar dynamics. In summary, we sacri£ce a margin of performance for the *robustness* of a robust controller.

One of the criticisms of robust control is that the performance sacri£ce might be larger than necessary. A degree of conservativeness is built into the robustness design process in order to achieve the stability guarantees. The stable reinforcement learning controller of the next subsection attempts to regain some of this lost performance.

4.3.4 Stable Reinforcement Learning Controller

Next, we apply the stable reinforcement learning controller with the goal of regaining some of the performance lost in the robust control design. We add a neuro-controller to the existing robust controller to discover the non-LTI dynamics which exist in the physical plant but not the LTI model. The neuro-controller learns, via reinforcement learning,

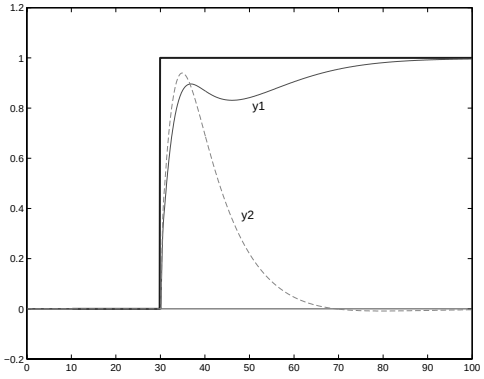


Figure 18: Step Response: Physical Plant with Robust Controller

while interacting with the simulated physical plant (i.e., the perturbed plant used in prior simulations). In effect, the reinforcement learner discovers more information about the actual dynamics of the physical plant and exploits this extra information not available to the robust controller.

In the previous example task, the state information of the system is small, having just one state variable (x the position). Furthermore, the dynamics of the £rst example task were simple enough that the neuro-controller could learn good control functions without using all the state information; only the tracking error was required (which is captured in the position state variable mentioned above). For the distillation column, the *state* of the system is quite large. To capture the full state of the system at any point in time we require the following:

- the two reference inputs: r_1 and r_2
- the internal state of the robust controller: 8 states
- the internal state of the plant: x_1 and x_2

There are a total of twelve state variables. To train on the full state information requires a actor net with 13 inputs (one extra input for the bias term) and a critic net with 14 inputs (two extra inputs for the "actions" of the actor net). These networks need an extraordinary amount of memory and training time to succeed in their neuro-control role. Consequently, we select a small subset of these states for use in our network. To the actor net, we use the two tracking errors (e_1, e_2 – which, again are essentially duplicates of the concentrations x_1, x_2) as the two inputs to the actor neural network. The actor network has two output units for the control signals $\hat{u}_1$ and $\hat{u}_2$. We select four hidden units for the actor network as this proved to be the most effective at learning the required control function quickly.

The critic net is a table look-up. It is a four dimensional table with inputs of the state (e_1, e_2) and the action ($\hat{u}_1, \hat{u}_2$). The resolution is ten for each dimension resulting in 10^4 "entries" in the table. The actor-critic network is trained for 500,000 samples (representing 500,000 seconds of elapsed simulation time) during which one of the two reference inputs (r_1, r_2) is ¤ipped $\{0 \rightarrow 1, 1 \rightarrow 0\}$ every 2,000 seconds. The learning rates are $\alpha = 0.01, \beta = 0.001$ for the critic and actor networks, respectively.

We perform two training runs. The £rst training run contains no robust stability constraints on the neuro-controller. Reinforcement learning is conducted without regard for bounding boxes and dynamic stability analysis. As a result, the actor network implements several unstable controllers during the non-robust training run. An example of such a controller is shown in Figure 19. While it may arrive at a satisfactory end-point (i.e, an improved control design after learning is completed), this phenomenon prohibits the use of reinforcement learning on-line for practical control applications.

In the second training run, we use the full stable reinforcement learning algorithm. After training is complete, the network improves the tracking performance as shown in Figure 20. Compare this result to the robust controller alone in Figure 18. Although the two graphs seem similar, the addition of the neuro-controller reduces the mean square tracking error from 0.286, achieved with just the robust controller, down to 0.243, achieved with the robust controller and the neuro-controller, a gain in tracking performance of approximately 15%. In Table 1, we summarize the tracking performance of various controllers by measuring the sum squared tracking error.

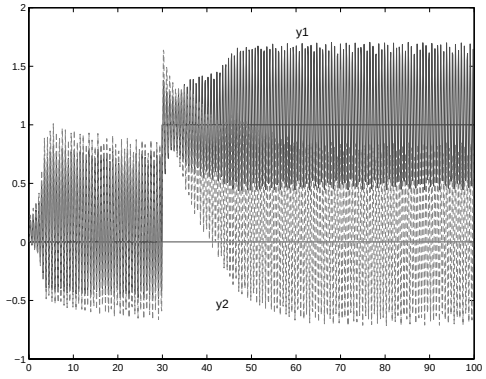


Figure 19: Perturbed Distillation Column with Unstable Neuro-controller

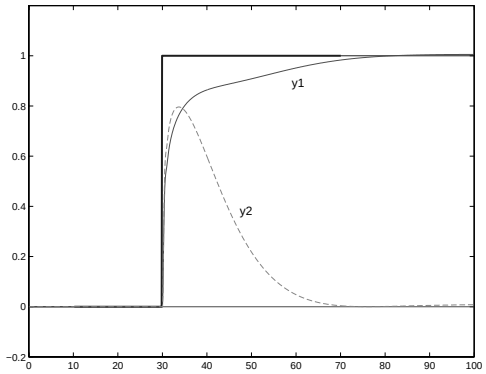


Figure 20: Perturbed Distillation Column with Stable Neuro-controller

In summary, the decoupling controller performs quite well on the plant model, but its performance on the physical plant is unacceptable. The robust controller does not perform nearly as well as the highly optimized decoupling controller on the nominal model. However, when applied to the "real plant", the robust controller halves the tracking error of the decoupling controller. Even more impressive than the reduction in tracking error is the signi£cantly better step response of the robust controller. Finally, we add the neuro-controller. By applying the stable reinforcement learning algorithm, the system retains stability and we are able to improve the tracking performance over the robust controller by 15%. It should be noted that we encounter considerable dif£culty in achieving this performance gain; much of the dif£culty is attributed to the massive learning experience and sensitive dependence on learning algorithm parameters. As we have already stated, this is a dif£cult control problem, on an ill conditioned MIMO plant.

Note also that our stable reinforcement algorithm converges to essentially the same performance improvement as the conventional reinforcement learning algorithm. The important difference is that our algorithm found a path through the space of stabilizing controllers to this same end point. Hence our algorithm could reasonably be implemented in a practical setting, where closed-loop stability must be maintained at all times.

	Plant Model	**"Real Plant"**
Decoupling Controller	$1.90\text{x}10^{-2}$	$6.46\text{x}10^{-1}$
Robust Controller	$2.57\text{x}10^{-1}$	$2.86\text{x}10^{-1}$
Neuro-Controller	Not Applicable	$2.43\text{x}10^{-1}$

Table 1: Sum Squared Tracking Error

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 addresses stability and performance problems in optimal control, due to differences in plant models and physical plants. However, robust control can be overly conservative, and thus sacri£ce some performance. Neuro-controllers may able to achieve better control than robust designs, especially when neuro-control is used in addition to (rather than instead of) robust control, as we do here. This is because neuro-controllers contain nonlinear components and are adaptable on-line. However, conventional neuro-control is not practical for real implementation, because the dif£cult dynamic analysis is intractable and stability cannot be assured.

We develop a *static stability* test to determine whether a neural network controller, with a speci£c £xed 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 provides stable control 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, in principle, 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 dif£cult control problems. In one case we are developing a robust controller for a Heating-Ventilating and Air-Conditioning (HVAC) system, both in simulation and on a physical experiment. We now have a MIMO robust controller operating on the experimental system, and are in the process of adding our reinforcement learning algorithm. We expect an improvement in performance due to unknown quantities in the real HVAC system and also to the fact that nonlinear relationships among the measured variables are known to exist. We are also further developing the theoretical proofs of static and dynamic stability using IQCs that are more specialized to our neural network architectures.

References

[1] Gary J. Balas, John C. Doyle, Keith Glover, Andy Packard, and Roy Smith. *μ-Analysis and Synthesis Toolbox*. The MathWorks Inc., 24 Prime Park Way Natick, MA 01760-1500, 1 edition, 1996.

[2] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike elements that can solve dif£cult 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.

[3] 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.

[4] Fu-Chuang Chen and Hassan K. Khalil. Adaptive control of a class of nonlinear discrete-time systems using neural networks. *IEEE Transactions on Automatic Control*, 40(5):791–801, May 1995.

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

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

[7] 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.

[8] 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.

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

[10] 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.

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

[12] Charles L. Phillips and Royce D. Harbor. *Feedback Control Systems*. Prentice Hall, 3 edition, 1996.

[13] Marios M. Polycarpou. Stable adaptive neural control scheme for nonlinear systems. *IEEE Transactions on Automatic Control*, 41(3):447–451, March 1996.

[14] Anders Rantzer. On the Kalman-Yacubovich-Popov lemma. *Systems & Control Letters*, 28:7–10, 1996.

[15] D. Rumelhart, G. Hinton, and R. Williams. Learning internal representations by error propagation. In D. Rumelhart and J. McClelland, editors, *Parallel Distributed Processing*, volume 1. Bradford Books, 1986.

[16] Sigurd Skogestad, Manfred Morari, and John Doyle. Robust control of ill-conditioned plants: High-purity distillation. *IEEE Transactions on Automatic Control*, 33(12):1092–1105, 1988.

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

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

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

[20] J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, May 1997.

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

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