From the CD-ROM Proceedings of the
2002 World Congress on Computational Intelligence
Honolulu, HI USA, 12–17 May 2002.

DDEKF Learning for Fast Nonlinear Adaptive Inverse Control

Gregory L. Plett and Hans Böttrich
Department of Electrical and Computer Engineering, University of Colorado at Colorado Springs,
1420 Austin Bluffs Pkwy., P.O. Box 7150, Colorado Springs, CO 80933–7150, USA

Abstract—Adaptive inverse control (AIC) uses three adaptive filters: plant model, controller and disturbance canceler. Methods are known for quick and efficient training of these filters if the plant is linear; however, known methods for nonlinear AIC learn very slowly. This paper modifies the standard nonlinear AIC learning methods (based on RTRL) to be based on DDEKF. Training becomes significantly faster.

I. INTRODUCTION

Adaptive inverse control (AIC), as reformulated by Widrow and Walach [1], is an automatic control-system design method that uses signal-processing methods and *learns* over time how to control a specific plant, whether it be linear or nonlinear [1]–[6]. For nonlinear systems, nonlinear adaptive filtering methods employing dynamic neural networks are used throughout. Invisible to the user, a three-part process is used internally. First, an adaptive plant model $\widehat{P}$ learns the dynamics of the plant. Secondly, an adaptive feedforward controller C learns to control the dynamics of the plant. Thirdly, an adaptive feedback disturbance canceler X learns to cancel disturbances that affect the plant. These processes may proceed concurrently. A block diagram of a nonlinear adaptive inverse control system is shown in Fig. 1.

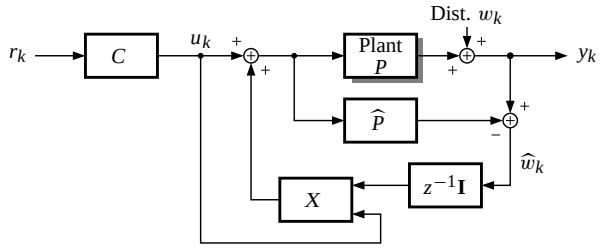

Fig. 1. Adaptive inverse control system block diagram.

Methods are known to quickly and efficiently train controllers and disturbance cancelers for linear SISO systems [1] and linear MIMO systems [7]. However, known methods for adaptive nonlinear control learn very slowly [2]. This paper presents some results regarding a training method for nonlinear adaptive inverse control that learns about an order of magnitude more quickly than previous methods.

II. DYNAMIC NEURAL-NETWORK FILTERING

Adaptive inverse control relies on adaptive digital filtering methods. Adaptive filters have an input x_k, an output y_k, and a "special input" called the desired response. The desired response d_k specifies the output we wish the filter to have. It is used to calculate an error signal e_k, which in turn is used to modify the internal parameters of the filter in such a way that the filter "learns" to perform a certain function.

Nonlinear AIC requires nonlinear adaptive filters. Here we use externally-recurrent layered neural networks to implement the filters. This layered structure makes it easy to compactly describe the network topology: $\mathcal{N}_{(\mathrm{a,b}):\alpha:\beta\ldots}$. This means: "The filter input is comprised of a tapped delay line with 'a' delayed copies of the exogenous input vector x_k, and 'b' delayed copies of the output vector y_k. Furthermore, there are 'α' neurons in the neural network's first layer of neurons, 'β' neurons in the second layer, and so on." Dynamics are introduced via the tapped delay lines at the input to the network, resulting in a *dynamic neural network.* A dynamic neural network of this type is called a *NARX* (Nonlinear AutoRegressive eXogenous input) filter. It is general enough to approximate any nonlinear dynamical system [8]. A nonlinear single-input-single-output (SISO) filter is created if x_k and y_k are scalars, and the network has a single output neuron. A nonlinear multi-input-multi-output (MIMO) filter is constructed by allowing x_k and y_k to be (column) vectors, and by augmenting the output layer with the correct number of additional neurons.

A. Adapting Dynamic Neural Networks

A feedforward neural network is one whose input contains no self-feedback of its previous outputs. Its weights may be adapted using the popular *backpropagation algorithm*, discovered independently by several researchers [9, 10], and popularized by Rumelhart [11]. A NARX filter, on the other hand, generally has self-feedback and must be adapted using a method such as *real-time recurrent learning* (RTRL) [12] or *backpropagation through time* (BPTT) [13]. Although compelling arguments may be made supporting either algorithm [14], we have chosen to use RTRL as the baseline algorithm in this paper since it is able to adapt filters used in an implementation of adaptive inverse control in real time.

RTRL works via the principle of gradient descent, whereby the weight change in the network ΔW is calculated as $\Delta W = -\eta d J_k^T / dW$, where J_k is a cost function to be minimized and the small positive constant η is called the *learning rate.* Often, $J_k = \mathbb{E}[e_k^T e_k]/2$ where $e_k = d_k - y_k$ is the network error computed as the desired response minus the actual neural-network

output. A stochastic approximation to J_k may be made as $J_k \approx e_k^T e_k/2$ which results in $dJ_k/dW = -e_k^T dy_k/dW$. For a *feedforward* neural network, $dy_k/dW = \partial y_k/\partial W$, which may be verified using the chain rule for total derivatives. The back-propagation algorithm is an elegant way of recursively computing $\partial J_k/\partial W$ by "back-propagating" the vector $-e_k$ from the output of the neural network back through the network to its first layer of neurons. The values which are computed are multiplied by η, and are used to adapt the neuron's weights.

For an externally-recurrent NARX filter, we must evaluate dy_k/dW differently. The filter computes a function of the form $y_k = f(x_k, x_{k-1}, \ldots, x_{k-n}, y_{k-1}, y_{k-2}, \ldots, y_{k-m}, W)$. Then, by the chain rule for total derivatives,

$$\frac{dy_k}{dW} = \frac{\partial y_k}{\partial W} + \sum_{i=0}^{n} \frac{\partial y_k}{\partial x_{k-i}} \frac{dx_{k-i}}{dW} + \sum_{i=1}^{m} \frac{\partial y_k}{\partial y_{k-i}} \frac{dy_{k-i}}{dW}. \quad (1)$$

The first term, $\partial y_k/\partial W$, is the direct effect of a change in the weights on y_k. The second term is zero, since dx_k/dW is zero for all k. The final term may be broken up into two parts. The first, $\partial y_k/\partial y_{k-i}$, is the effect on a change in a previous output on the current output. The second part, dy_{k-i}/dW, is simply a previously-calculated and stored value of dy_k/dW. When the system is "turned on," dy_i/dW are set to zero for $i = 0, -1, -2, \ldots$, and the rest of the terms are calculated recursively from that point on. The Jacobian partial derivatives may computed using the back-propagation algorithm [2].

A problem with first-order gradient descent learning such as RTRL is that it is very slow. Second-order techniques can provide speed improvements. *Extended-Kalman-Filter* (EKF) learning is one such second-order method [15], but may require operations on large matrices, so can be computationally demanding. *Decoupled-Extended-Kalman-Filter* (DEKF) learning breaks the neural network into g groups of neurons, where EKF learning is independently applied to each group [16], and is much faster since operations are performed on smaller matrices. It can be extended to the dynamic configuration as *Dynamic-Decoupled-Extended-Kalman-Filter* (DDEKF) learning [17]. For this paper, we use DDEKF where each group contains a single neuron. The algorithm is summarized in Alg. 1. The $H_i(k)$ matrix in the algorithm is the portion of the dy_k^T/dW matrix that applies to the weights in group i.[1]

III. NONLINEAR ADAPTIVE INVERSE CONTROL

A. System Identification using RTRL or DDEKF

The first step in performing adaptive inverse control is to make an adaptive model of the plant. The model should capture the dynamics of the plant well enough that a controller designed to control the plant model will also control the plant very well. This is a straightforward application of the adaptive filtering techniques in Section II.

Algorithm 1 DDEKF algorithm.

$$A(k) = \left[R(k) + \sum_{i=1}^{g} H_i(k)^T P_i(k) H_i(k) \right]^{-1}$$
$$K_i(k) = \eta(k) P_i(k) H_i(k) A(k)$$
$$W_i(k+1) = W_i(k) + K_i(k)(d(k) - y(k))$$
$$P_i(k+1) = P_i(k) - K_i(k) H_i(k)^T P_i(k) + Q_i(k),$$

where $R(k)$ is often set to I, $Q_i(k)$ is often set to a small constant times I, and $\eta(k)$ is the learning rate.

A method for adaptive plant modeling is depicted in Fig. 2. The plant is excited with the signal u_k, and the disturbed output y_k is measured. The plant model $\widehat{P}$ is also excited with u_k, and its output $\hat{y}_k$ is computed. The plant modeling error is the difference between the model output and the measured plant output: $e_k^{(mod)} = y_k - \hat{y}_k$. This modeling error is then used by the adaptation algorithm to update the weight values of the adaptive filter. It can be shown that the plant model is unbiased by the disturbance if the disturbance is zero-mean and if u_k and w_k are statistically independent [2].

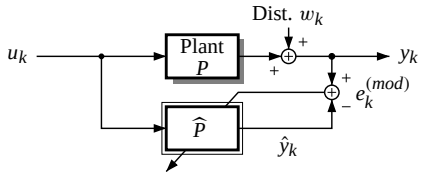

Fig. 2. Adaptive plant modeling.

B. Feedforward Control using BPTM

The second step is to adapt the feedforward controller C.[2] The goal is to make the dynamics of the controlled system PC approximate the fixed filter M as closely as possible, where M is a user-specified *reference model.* The reference-input signal r_k is filtered through M to create a desired response d_k for the plant output. The measured plant output is compared with the desired plant output to create a system error signal $e_k^{(sys)} = d_k - y_k$. We will adapt C to minimize the mean-squared system error.

To adapt C we regard the series combination of C and $\widehat{P}$ as a single adaptive filter. The desired response for this combined filter is d_k. Therefore, we can use d_k to *compute* weight updates for the conglomerate filter. *However, we only apply*

[1] Both RTRL and DDEKF employ a derivative-calculation method and a weight-update method. We see that the derivative calculation is the same for both learning rules (*e.g.,* Eq. (1)), but that the weight-update method is different.

[2] We shall restrict our development to apply only to stable plants. If the plant of interest is unstable, conventional feedback should be applied to stabilize it. Then the combination of the plant and its feedback stabilizer can be regarded as an equivalent stable plant.

0–7803–7278–6/02/$10.00 ©2002 IEEE

the weight updates to the weights in C; $\widehat{P}$ is still updated using the plant modeling error $e_k^{(mod)}$. Figure 3 shows the general framework to be used. We say that the system error is *back-propagated* through the plant model, and used to adapt the controller. For this reason, the algorithm is named "Back-Prop Through (Plant) Model" (BPTM).

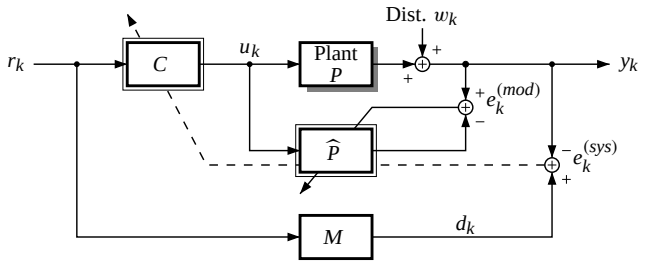

Fig. 3. Structure diagram illustrating the BPTM method.

If $g(\cdot)$ is the function implemented by the controller C, and $f(\cdot)$ is the function implemented by the plant model $\widehat{P}$, and W_C are the weights of the controller neural network,

$$u_k = g\big(u_{k-1}, u_{k-2} \ldots u_{k-m}, r_k, r_{k-1} \ldots r_{k-q}, W_C\big)$$
$$y_k \approx \hat{y}_k = f\big(\hat{y}_{k-1}, \hat{y}_{k-2} \ldots \hat{y}_{k-n}, u_k, u_{k-1} \ldots u_{k-p}\big),$$

Using gradient descent (*i.e.*, extending RTRL via BPTM) the update $\Delta(W_C)_k^T = -\eta e_k^T d\hat{y}_k/dW_C$. This can be found by the set of equations:

$$\frac{du_k}{dW_C} = \frac{\partial u_k}{\partial W_C} + \sum_{j=1}^{m} \left(\frac{\partial u_k}{\partial u_{k-j}}\right)\left(\frac{du_{k-j}}{dW_C}\right) \tag{2}$$

$$\frac{d\hat{y}_k}{dW_C} = \sum_{j=0}^{p} \left(\frac{\partial \hat{y}_k}{\partial u_{k-j}}\right)\left(\frac{du_{k-j}}{dW_C}\right) + \sum_{j=1}^{n} \left(\frac{\partial \hat{y}_k}{\partial \hat{y}_{k-j}}\right)\left(\frac{d\hat{y}_{k-j}}{dW_C}\right) \tag{3}$$

First, consider du_k/dW_C as expanded in (2). It is the same in form as (1) and is computed the same way. Second, consider $d\hat{y}_k/dW_C$ in (3). The first term in the first summation is the Jacobian $\partial\hat{y}_k/\partial u_{k-j}$, which may be computed via the back-propagation algorithm. The next term, du_{k-j}/dW_C is the current or a previously-computed and saved version of du_k/dW_C, computed via (2). The first term in the second summation, $\partial\hat{y}_k/\partial\hat{y}_{k-j}$ is another Jacobian. The final term, $d\hat{y}_{k-j}/dW_C$, is a previously-computed and saved version of $d\hat{y}_k/dW_C$.

A practical implementation is realized by compacting the notation into a collection of matrices. We define

$$dU_k \triangleq \left[\left(\frac{\partial u_k}{\partial W}\right)^T \left(\frac{\partial u_{k-1}}{\partial W}\right)^T \cdots \left(\frac{\partial u_{k-p}}{\partial W}\right)^T\right]^T$$

$$dH_k \triangleq \left[\left(\frac{\partial h(\cdot)}{\partial u_k}\right)^T \left(\frac{\partial h(\cdot)}{\partial u_{k-1}}\right)^T \cdots \left(\frac{\partial h(\cdot)}{\partial u_{k-r}}\right)^T\right]^T$$

Algorithm 2 BPTM algorithm to adapt C.

begin {Adapt C*}*
- Update du_k/dW:
  - Shift dU_k down N_i rows.
  - Backpropagate N_i unit vectors through C to form $\partial_U U_k$ and $\partial u_k/\partial W_C$. Each backpropagation produces one row of both matrices.
  - Compute top N_i rows of dU_k to be $\partial u_k/\partial W_C + (\partial_U U_k)(dU_k)$.
- Update dy_k/dW_C:
  - Backpropagate N_o unit vectors through $\widehat{P}$ to form $\partial_U Y_k$ and $\partial_Y Y_k$. Each backpropagation produces one row of both matrices.
  - Compute $d_W Y_k = (\partial_U Y_k)(dU_k) + (\partial_Y Y_k)(dYk)$.
  - Shift dY_k down N_o rows and save $d_W Y_k$ in the top N_o rows.
- Update Weights:
  - Compute $\Delta(W_C)_k^T = \eta e_k^T d_W Y_k$.
  - Adapt, enumerating weights in the same order as when computing $\partial_W U_k$.

end {Adapt C*}*

where N_i is the number of plant inputs and N_o is the number of plant outputs.

$$dY_k \triangleq \left[\left(\frac{\partial y_{k-1}}{\partial W}\right)^T \left(\frac{\partial y_{k-2}}{\partial W}\right)^T \cdots \left(\frac{\partial y_{k-n}}{\partial W}\right)^T\right]^T$$

$$\partial_U Y_k \triangleq \left[\left(\frac{\partial y_k}{\partial u_k}\right)^T \left(\frac{\partial y_k}{\partial u_{k-1}}\right)^T \cdots \left(\frac{\partial y_k}{\partial u_{k-p}}\right)^T\right]^T$$

$$\partial_Y Y_k \triangleq \left[\left(\frac{\partial y_k}{\partial y_{k-1}}\right)^T \left(\frac{\partial y_k}{\partial y_{k-2}}\right)^T \cdots \left(\frac{\partial y_k}{\partial y_{k-n}}\right)^T\right]^T$$

$$\partial_U U_k \triangleq \left[\left(\frac{\partial u_k}{\partial u_{k-1}}\right)^T \left(\frac{\partial u_k}{\partial u_{k-2}}\right)^T \cdots \left(\frac{\partial u_k}{\partial u_{k-p}}\right)^T\right]^T.$$

The algorithm is summarized in Alg. 2. Any programming language supporting matrix mathematics can very easily implement this algorithm. It works well.

To adapt the controller using DDEKF, the steps performed using RTRL must also be done to find $d\hat{y}_k/dW_C$. Then, let $H_i(k)$ be the portion of $d\hat{y}_k^T/dW_C$ associated with the weights in group i. Now, adapt W_C using Alg. 1. To distinguish between BPTM based on RTRL and DDEKF, the two methods are henceforth referred to as BPTM-RTRL and BPTM-DDEKF.

C. Disturbance Canceling using BPTM

As described so far, AIC is primarily a feedforward control mechanism (with slow feedback through the adaptation process), and does not make any attempt to reject disturbance. A fully integrated nonlinear MIMO adaptive inverse control system with disturbance canceling circuitry is shown in Fig. 4. The upper half of the figure shows how the plant is adaptively modeled and used to adapt a controller to perform model-reference based control. The lower half of the figure shows how disturbance is estimated (using a filter whose weights are a digital copy of the plant-model weights) and filtered to perform disturbance canceling.

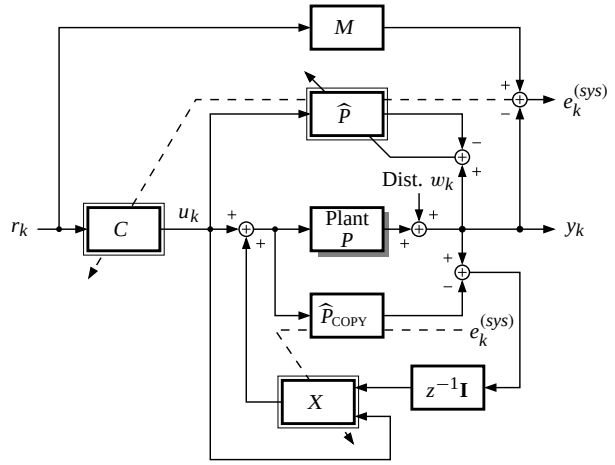

Fig. 4. An integrated nonlinear MIMO system.

Proceeding to develop an algorithm to adapt X, we consider that the system error is composed of three parts:

- One part of the system error is dependent on the input command vector $r_k, r_{k-1}, \ldots$ in C. This part of the system error is reduced by adapting C.
- Another part of the system error is dependent on the estimated disturbance vector $\hat{w}_k, \hat{w}_{k-1}, \ldots$ in X. This part of the system error is reduced by adapting X.
- The minimum-mean-squared-error. This part of the system error is independent of both the input command vector in C and the estimate disturbance vector in X. It is either irreducible (if the system dynamics prohibit improvement), or may be reduced by making the tapped-delay lines at the input to X or C larger. In any case, adaptation of the weights in X or C will not reduce the minimum-mean-squared-error.
- The fourth possible part of the system error is the part that is dependent on both the input command vector and the disturbance vector. However, by assumption, r_k and w_k are independent, so this part of the system error is zero.

If we use the BPTM algorithm and backpropagate the system error through the plant model (copy), and use it to adapt X as well, the disturbance canceler learns to reduce the component of the system error dependent on the estimated disturbance signal. The component of the system error due to unconverged C and minimum-mean-squared-error will not bias the disturbance canceler.

If $g(\cdot)$ is the function implemented by the disturbance canceler X, and $f(\cdot)$ is the function implemented by the plant model copy $\widehat{P}_{\text{COPY}}$, and W_X are the weights of the disturbance canceler neural network,

$$\tilde{u}_k = g\big(\tilde{u}_{k-1}, \tilde{u}_{k-2} \ldots \tilde{u}_{k-m}, \hat{w}_{k-1}, \hat{w}_{k-2} \ldots \hat{w}_{k-q}, u_k, u_{k-1} \ldots u_{k-r}, W_X\big)$$

$$y_k \approx \hat{y}_k = f\big(\hat{y}_{k-1}, \hat{y}_{k-2} \ldots \hat{y}_{k-n}, \bar{u}_k, \bar{u}_{k-1} \ldots \bar{u}_{k-p}\big),$$

where $\bar{u}_k = u_k + \tilde{u}_k$ and $\bar{u}_k$ is the input to the plant, u_k is the output of the controller, and $\tilde{u}_k$ is the output of the disturbance canceler.

BPTM using RTRL computes the disturbance-canceler weight update $\Delta(W_X)_k^T = -\eta e_k^T d\hat{y}_k/dW_X$. This can be found by the set of equations:

$$\begin{aligned}
\frac{d\bar{u}_k}{dW_X} &= \frac{du_k}{dW_X} + \frac{d\tilde{u}_k}{dW_X} = \frac{d\tilde{u}_k}{dW_X} \\
&= \frac{\partial \tilde{u}_k}{\partial W_X} + \sum_{j=1}^{m} \left(\frac{\partial \tilde{u}_k}{\partial \tilde{u}_{k-j}}\right)\left(\frac{d\bar{u}_{k-j}}{dW_X}\right) \\
\frac{d\hat{y}_k}{dW_X} &= \sum_{j=0}^{p} \left(\frac{\partial \hat{y}_k}{\partial \bar{u}_{k-j}}\right)\left(\frac{d\bar{u}_{k-j}}{dW_X}\right) \\
&\qquad + \sum_{j=1}^{n} \left(\frac{\partial \hat{y}_k}{\partial \hat{y}_{k-j}}\right)\left(\frac{d\hat{y}_{k-j}}{dW_X}\right).
\end{aligned}$$

We see that the method to adapt the disturbance-canceler weights is identical to the one to adapt the controller weights. So, either BPTM-RTRL or BPTM-DDEKF may be used.

IV. SIMULATION EXAMPLE

We present a simple example to demonstrate the relative efficiencies of DDEKF and RTRL as applied to the three adaptive processes in adaptive inverse control. The plant we wish to control is specified by the equations [18, 19]

$$\begin{aligned}
s_k &= \frac{s_{k-1}}{1+s_{k-1}^2} + \sin(u_{k-1}) \\
y_k &= s_k + w_k,
\end{aligned}$$

where the disturbance w_k is generated by filtering i.i.d. Gaussian random numbers with zero mean and standard deviation 0.01 through a one-pole filter with the pole at $z = 0.99$.

A. System Identification (in the absence of disturbance)

Two $\mathcal{N}_{(3,1):5,1}$ neural networks were trained to model the plant. Both started with the same set of initial weights, and used the RTRL and DDEKF algorithms, respectively. Training was performed in the absence of disturbance, although similar results are obtained if disturbance is present. Learning curves for both cases are shown in the top two plots of Fig. 5, where squared modeling error is plotted versus adaptive iteration. Convergence is qualitatively judged to occur when the learning curves flatten out. We see that the RTRL algorithm converges in approximately 1×10^7 iterations, and the DDEKF algorithm converges in approximately 1×10^5 iterations. Therefore, DDEKF is 100 times faster than RTRL in this example.

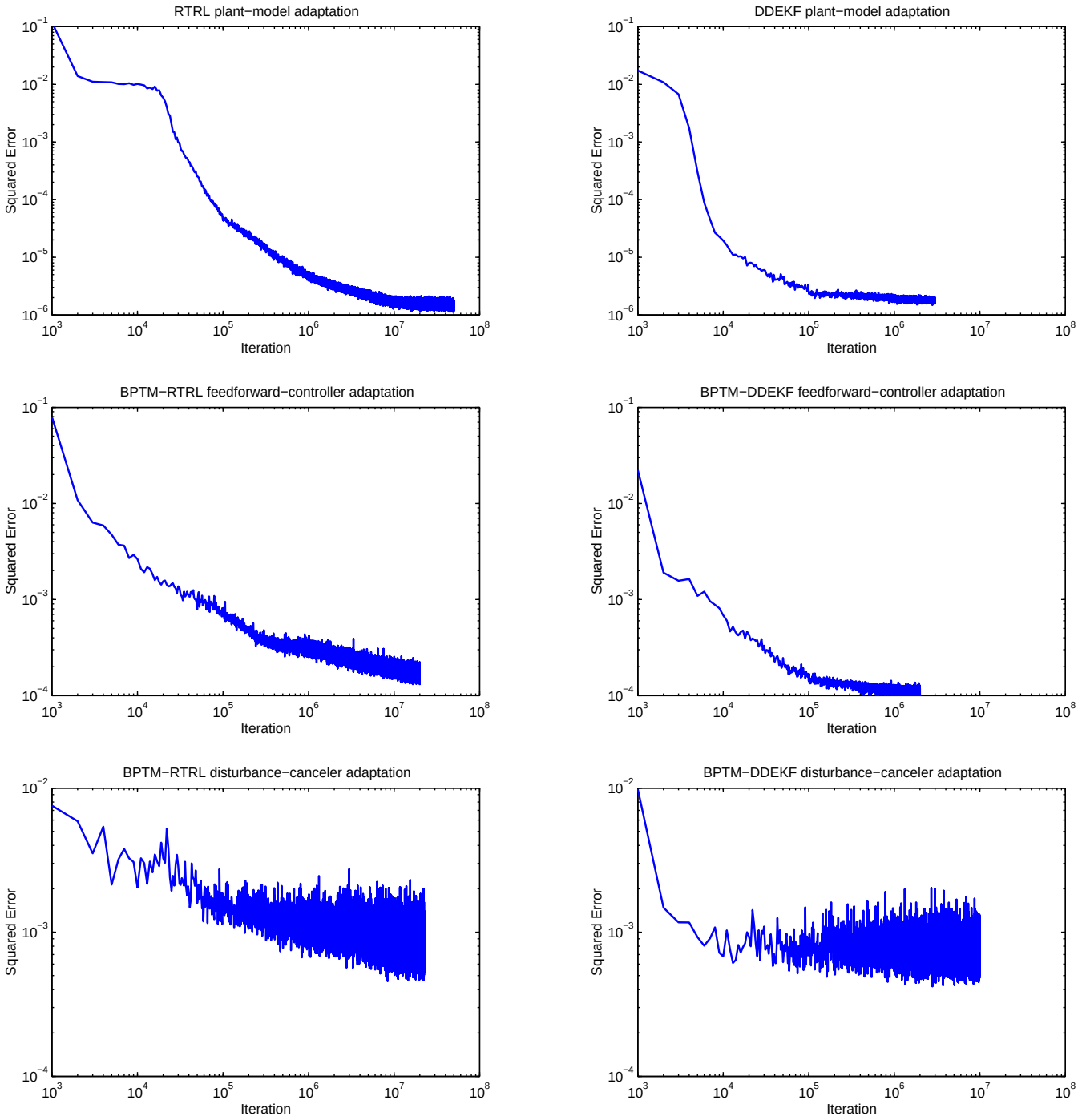

Fig. 5. Learning curves for plant-model adaptation (in the absence of disturbance), controller adaptation (in the absence of disturbance) and disturbance-canceler adaptation.

B. Feedforward Control (in the absence of disturbance)

Two $\mathcal{N}_{(5,3):10,1}$ networks were trained to perform feedforward control. Both started with the same set of initial weights and used the BPTM-RTRL and BPTM-DDEKF algorithms, respectively. Training was performed in the absence of disturbance, although similar results are obtained if disturbance is present. Learning curves for both cases are shown in the middle two plots of Fig. 5, where squared system error is plotted versus adaptive iteration. We see that BPTM-RTRL converges in approximately 1×10^7 iterations, and BPTM-DDEKF converges in approximately 1×10^5 iterations. Therefore, BPTM-DDEKF is 100 times faster than BPTM-RTRL in this example.

C. Disturbance Canceling

Two $\mathcal{N}_{([5,3],1):10,1}$ networks were trained to perform disturbance canceling. Both started with the same set of initial weights and used BPTM-RTRL and BPTM-DDEKF, respectively. The $[5, 3]$ notation indicates that five tap-delayed copies of w_k and three tap-delayed copies of u_k were used as primary network input; one feedback input of the previous disturbance-canceler output was also used as input. Learning curves for both cases are shown in the bottom two plots of Fig. 5, where squared system error is plotted versus adaptive iteration. We see that BPTM-RTRL converges in approximately 1×10^7 iterations, and BPTM-DDEKF converges in approximately 1×10^4 iterations. Therefore, BPTM-DDEKF is 1000 times faster than BPTM-RTRL in this example.

V. CONCLUSIONS

DDEKF learning is compared with RTRL learning for the purpose of adapting externally-recurrent neural networks for nonlinear adaptive inverse control. For the simple example presented, a speedup of two to three orders of magnitude is observed by using DDEKF. Other simulations not presented here indicate that at least an order of magnitude speedup is expected when using DDEKF. Complexity of DDEKF is not much higher than for RTRL, so we conclude that there is no compelling reason to use RTRL for the purpose of adapting filters for nonlinear adaptive inverse control. DDEKF appears to be the better choice.

References

[1] B. Widrow and E. Walach, *Adaptive Inverse Control*, Prentice Hall P T R, Upper Saddle River, NJ, 1996.

[2] G. L. Plett, *Adaptive Inverse Control of Plants with Disturbances*, PhD thesis, Stanford University, Stanford, CA 94305, May 1998.

[3] B. Widrow, G. L. Plett, E. Ferreira, and M. Lamego, "Adaptive inverse control based on nonlinear adaptive filtering", in *Proceedings of 5th IFAC Workshop on Algorithms and Architectures for Real-Time Control AARTC'98*, (Cancun, MX: April 1998), pp. 247–252, (invited paper).

[4] B. Widrow and G. L. Plett, "Nonlinear adaptive inverse control", in *Proceedings of the 36th IEEE Conference on Decision and Control*, (San Diego, CA: 10–12 December 1997), vol. 2, pp. 1032–1037.

[5] B. Widrow and G. L. Plett, "Adaptive inverse control based on linear and nonlinear adaptive filtering", in *Proceedings of International Workshop on Neural Networks for Identification, Control, Robotics and Signal/Image Processing*, (Venice, Italy: 21–23 August 1996), pp. 30–38.

[6] B. Widrow and G. L. Plett, "Adaptive inverse control based on linear and nonlinear adaptive filtering", in *Proceedings of the World Congress on Neural Networks*, (San Diego, CA: September 1996), pp. 620–27.

[7] G. L. Plett, "Efficient linear MIMO adaptive inverse control", in *Proceedings of 2001 IFAC Workshop on Adaptation and Learning in Control and Signal Processing*, (Cernobbio-Como, Italy: August 2001), pp. 89–94.

[8] H. T. Siegelmann, B. B. Horne, and C. L. Giles, "Computational capabilities of recurrent NARX neural networks", *IEEE Transactions on Systems, Man and Cybernetics—Part B: Cybernetics*, vol. 27, no. 2, pp. 208–215, April 1997.

[9] P. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, PhD thesis, Harvard University, Cambridge, MA, August 1974.

[10] D. B. Parker, "Learning logic", Tech. Rep. Invention Report S81–64, File 1, Office of Technology Licencing, Stanford University, October 1982.

[11] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation", in *Parallel Distributed Processing*, D. E. Rumelhart and J. L. McClelland, Eds., vol. 1, chapter 8. The MIT Press, Cambridge, MA, 1986.

[12] R. J. Williams and D. Zipser, "Experimental analysis of the real-time recurrent learning algorithm", *Connection Science*, vol. 1, no. 1, pp. 87–111, 1989.

[13] D. H. Nguyen and B. Widrow, "Neural networks for self-learning control systems", *IEEE Control Systems Magazine*, vol. 10, no. 3, pp. 18–23, April 1990.

[14] S. W. Piché, "Steepest descent algorithms for neural network controllers and filters", *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 198–212, March 1994.

[15] S. Singhal and L. Wu, "Training multilayer perceptrons with the extended kalman algorithm", in *Advances in Neural Information Processing Systems I* (Denver: 1988), D. S. Touretzky, Ed., San Mateo, CA, 1989, pp. 133–140, Morgan Kaufmann.

[16] G. V. Puskorius and L. A. Feldkamp, "Decoupled extended Kalman filter training of feedforward layered networks", in *Proceedings of the 1991 International Joint Conference on Neural Networks* (San Diego: 1990), New York, 1991, IEEE Neural Networks Society, vol. II, pp. 133–141.

[17] G. V. Puskorius and L. A. Feldkamp, "Recurrent network training with the decoupled extended Kalman filter algorithm", in *Science of Artificial Neural Networks* (Orlando Florida: 21–24 April 1992), New York, 1992, SPIE Proceedings Series, vol. 1710, part 2, pp. 461–473.

[18] M. Bilello, *Nonlinear Adaptive Inverse Control*, PhD thesis, Stanford University, Stanford, CA, April 1996.

[19] H. Böttrich, "Adaptive inverse control of nonlinear systems using recurrent neural networks", Master's thesis, University of Colorado at Colorado Springs, Colorado Springs, CO, May 2000, Technical Report EAS_ECE_2000_04.