Cascade Neural Networks with Node-Decoupled Extended Kalman Filtering

Michael C. Nechyba and Yangsheng Xu

The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Most neural networks used today rely on rigid, fixed-architecture networks and/or slow, gradient descent-based training algorithms (e. g. backpropagation). In this paper, we propose a new neural network learning architecture to counter these problems. Namely, we combine (1) flexible cascade neural networks, which dynamically adjust the size of the neural network as part of the learning process, and (2) node-decoupled extended Kalman filtering (NDEKF), a fast converging alternative to backpropagation. In this paper, we first summarize how learning proceeds in cascade neural networks. We then show how NDEKF fits seamlessly into the cascade learning framework, and how cascade learning addresses the poor local minima problem of NDEKF reported in [1]. We analyze the computational complexity of our approach and compare it to fixed-architecture training paradigms. Finally, we report learning results for continuous function approximation and dynamic system identification — results which show substantial improvement in learning speed and error convergence over other neural network training methods.

1. Introduction

In recent years, artificial neural networks have shown great promise in identifying complex nonlinear mappings from observed data and have found many applications in robotics and other nonlinear control problems. Despite significant progress in the application of neural networks to many real-world problems, however, the vast majority of neural network research still relies on *fixed-architecture* networks trained through *backpropagation* or some other slightly enhanced gradient descent algorithm.There are two main problems with this prevailing approach. First, the "appropriate" network architecture varies from application to application; yet, it is difficult to guess this architecture — the number of hidden units and number of layers — *a priori* for a specific application without some trial and error. Even within the same application, functional complexity requirements can vary widely, as is the case, for example, in modeling human control strategies from different individuals [2]. Second, backpropagation and other gradient descent techniques tend to converge rather slowly, often exhibit oscillatory behavior, and frequently convergence to poor local minima [3].

To address the problem of fixed architectures in neural networks, we look towards flexible cascade neural networks [4]. In cascade learning, the network topology is not fixed prior to learning, but rather adjusts dynamically as a function of learning, as hidden units are added to a minimal network one at a time.

To address the second problem — slow convergence with gradient-descent training algorithms — we look towards *extended Kalman filtering (EKF)*. What makes EKF algorithms attractive is that, unlike backpropagation, they explicitly account for the pairwise interdependence of the weights in the neural network during training. Singhal and Wu [5] were the first to show how the EKF algorithm can be used for neural network training. While converging to better local minima in many fewer epochs than backpropagation, their *global extended Kalman filtering (GEKF)* approach, carries a heavy computational toll. GEKF's computational complexity is $O(m^2)$, where m is the number of weights in the neural network. This is prohibitive, even for moderately sized neural networks, where the weights can easily number in the thousands.

To address this problem, Puskorius and Feldkamp [1], propose *node-decoupled extended Kalman filtering (NDEKF)*, which considers only the pairwise interdependence of weights feeding into the same node, rather than the interdependence of all the weights in the network. While this approach is computationally tractable through a significant reduction in the computational complexity, the authors report that NDEKF tends to converge to poor local minima, for network architectures not carefully selected to have little redundancy (i.e. few excess free parameters).

In this paper we show that combining cascade neural networks with NDEKF solves the problem of poor local minima reported in [1], and that the resulting learning architecture substantially outperforms other neural network training paradigms in learning speed and/or error convergence for learning tasks important in control problems. We first summarize how learning proceeds in cascade neural networks. We then show how NDEKF fits seamlessly into the cascade learning framework, and how cascade learning addresses the poor local minima problem of NDEKF. We analyze the computational complexity of our approach and compare it to fixed-architecture training paradigms. Finally, we report learning results for continuous function approximation and dynamic system identification.

2. Cascade neural networks

Below, we briefly summarize the cascade neural network training algorithm, as formulated in [4]. Further details, which are omitted here for space reasons, may be found in [2,4,6].

Initially, there are no hidden units in the network, only direct input-output connections which are trained first using the quickprop algorithm [3]. When no appreciable error reduction occurs, a first hidden unit is added to the network from a pool of *candidate* units, which are trained independently and in parallel with different random initial weights. Once installed, the hidden unit

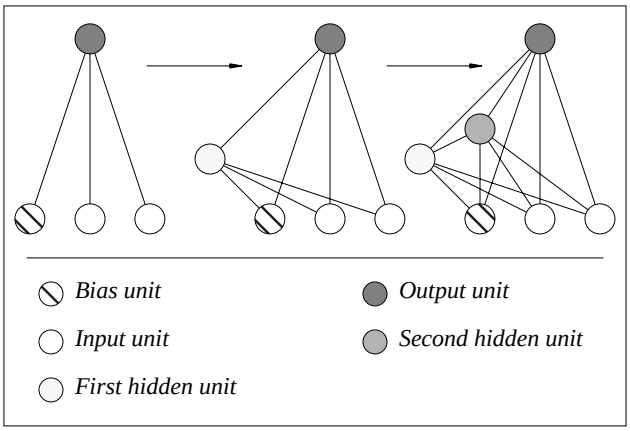

Fig. 1: The cascade learning architecture adds hidden units one at a time to an initially minimal network.

input weights are frozen, while the weights to the output units are retrained. This process is repeated with each additional hidden unit, which receives input connections from both the inputs and all previous hidden units, resulting in a cascaded structure. Figure 1, for example, illustrates how a two-input, single-output network grows as two hidden units are added. Thus, a cascade network with n_{in} inputs, n_h hidden units and n_o outputs, has n_w connections where,

$$n_w = n_{in}n_o + n_h(n_{in} + n_o) + (n_h - 1)n_h/2 \qquad \text{(Eq. 1)}$$

We can further relax *a priori* assumptions about functional form by allowing new hidden units to have variable activation functions [2, 7]. During candidate training, the algorithm will select for installment whichever candidate unit reduces the error for the training data the most. Typical alternatives to the sigmoidal activation function are the Gaussian function, Bessel functions, and sinusoidal functions of various frequency [6].

3. Node-decoupled extended Kalman filtering

While quickprop is an improvement over standard backpropagation it can still require many iterations until satisfactory convergence is reached [3, 5]. Thus, we modify standard cascade learning by replacing the quickprop algorithm with *node-decoupled extended Kalman filtering* (*NDEKF*), which has been shown to have better convergence properties and faster training times than gradient-descent techniques for fixed-architecture multi-layer feedforward networks [1].

3.1 Learning architecture

In *general extended Kalman filtering (GEKF)* [5], an $m \times m$ conditional error covariance matrix P, which stores the interdependence of each pair of m weights in a given neural network is explicitly generated. NDEKF reduces this computational and storage complexity by — as the name suggests — decoupling weights by node, so that we consider only the interdependence of weights feeding into the same unit (or node). This, of course, is a natural formulation for cascade learning, since we only train the input-side weights of one hidden unit and the output units at any one time; we can partition the m weights by unit into $n_o + 1$ groups — one group for the current hidden unit, n_o groups for the output units. In fact, by iteratively training one hidden unit at a time and then freezing that unit's weights, we minimize the potentially detrimental effect of the node-decoupling.

Denote ω_k^i as the input-side weight vector of length m_i at iteration k, for unit $i \in \{0, 1, \ldots, n_o\}$, where $i = 0$ corresponds to the current hidden unit being trained, and $i \in \{1, \ldots, n_o\}$ corresponds to the ith output unit, and

$$m_i = \begin{cases} n_{in} + n_h - 1 & i = 0 \\ n_{in} + n_h & i \in \{1, \ldots, n_o\} \end{cases} \qquad \text{(Eq. 2)}$$

The NDEKF weight-update recursion is then given by,

$$\omega_{k+1}^i = \omega_k^i + \{(\psi_k^i)^T(A_k\xi_k)\}\phi_k^i \qquad \text{(Eq. 3)}$$

where ξ_k is the n_o-dimensional error vector for the current training pattern, ψ_k^i is the n_o-dimensional vector of partial derivatives of the network's output unit signals with respect to the ith unit's net input, and

$$\phi_k^i = P_k^i\zeta_k^i \qquad \text{(Eq. 4)}$$

$$A_k = \left[I + \sum_{i=0}^{n_o} \{(\zeta_k^i)^T\phi_k^i\}[\psi_k^i(\psi_k^i)^T] \right]^{-1} \qquad \text{(Eq. 5)}$$

$$P_{k+1}^i = P_k^i - \{(\psi_k^i)^T(A_k\psi_k^i)\}[\phi_k^i(\phi_k^i)^T] + \eta_Q I \qquad \text{(Eq. 6)}$$

$$P_0^i = (1/\eta_P)I \qquad \text{(Eq. 7)}$$

where ζ_k^i is the m_i-dimensional input vector for the ith unit, and P_k^i is the $m_i \times m_i$ approximate conditional error covariance matrix for the ith unit. We include the parameter η_Q in (Eq. 6) to alleviate singularity problems for P_k^i [1]. In (Eq. 3) through (Eq. 6), {}'s, ()'s, and []'s evaluate to scalars, vectors and matrices, respectively.

The ψ_k^i vector is easy to compute within the cascade framework. Let O_i be the value of the *ith* output node, Γ_O be its corresponding activation function, net_{Oi} be its net activation, Γ_H be the activation function for the current hidden unit being trained, and net_H be its net activation. Then,

$$\partial O_i/\partial net_{Oj} = 0\,,\ \forall i \neq j \qquad \text{(Eq. 8)}$$

$$\partial O_i/\partial net_{Oi} = \Gamma'_O(net_{Oi}) \qquad \text{(Eq. 9)}$$

$$\partial O_i/\partial net_H = w \cdot \Gamma'_O(net_{Oi}) \cdot \Gamma'_H(net_H) \qquad \text{(Eq. 10)}$$

where w is the weight connecting the current hidden unit to the *ith* output unit.

Throughout this paper, we will use the short-hand notation in Table 1 to describe the various different learning techniques.

Table 1: Notation

Symbol	*Methodology*	*Training algorithm*
Fq	Fixed architecture[a]	quickprop
Cq	Cascade learning[b]	quickprop
Fk	Fixed architecture	NDEKF
Ck	Cascade learning	NDEKF

a. All weights are trained simultaneously.
b. Hidden units are added and trained one at a time.

3.2 Computational complexity

The computational complexity for cascade learning with NDEKF is given by,

$$O\left(n_o^3 + \sum_{i=0}^{n_o} m_i^2\right) \qquad \text{(Eq. 11)}$$

The $O(n_o^3)$ computational complexity caused by the matrix inversion in (Eq. 5) restricts this approach to applications where the number of outputs is relatively few. Below, we compare the computational complexity of our proposed learning architecture to two other regimes: (1) layered feedforward neural networks trained with backpropagation (pattern-wise update), and (2) NDEKF alone (i.e. used on fixed-architecture networks).

First, consider the computational cost (per training pattern) of training one candidate unit for a network with n_{in} input units, $(i-1)$ hidden units, and n_o output units($n_t = n_{in} + i$):

- cost(A^{-1}) $(n_o \times n_o$ symmetric matrix),
- $2n_t^2(n_o+1) + 4n_t n_o + n_o^3 + 4n_o^2 + 7n_o - 2$ multiplications,
- $(n_t^2(n_o+1) + n_t(9n_o+7) + 2n_o^2 + 4n_o - 16)/2$ additions,
- $3n_o + 1$ function evaluations

For comparison with backpropagation, we look at the computational cost (per training pattern) for a two-layered neural network with n_{in} input units, $n_H/2$ hidden units in both hidden layers, and n_o output units:

- $(5/4)n_H^2 + n_H(2n_{in}+1) + n_o(5/2 n_H + 1)$ multiplications,
- $n_H^2 + (3/2)n_H n_{in} + n_H(2n_o - 2)$ additions,
- $2(n_H + n_o)$ function evaluations

To arrive at a composite cost for each method, we weigh multiplications and additions by a factor of 1.0, and function evaluations by a factor of 5.0. In addition, we multiply the composite cost for the cascade/NDEKF method by $n_c = 8$, a typical number for the pool of candidate units and average the cost over all $i \in \{1, 2, \ldots, n_h\}$. Let γ_{Ck} denote the average computational cost per training pattern for our method, and let γ_{BP} denote the computational cost per training pattern for training the two-layered network with backpropagation. We are interested in the ratio,

$$\sigma = \gamma_{Ck}/\gamma_{BP} \qquad \text{(Eq. 12)}$$

for equivalently sized neural networks. By *equivalently sized*, we mean neural networks with approximately the same final number of weights, such that,

$$n_{in}n_o + n_h(n_{in}+n_o) + (n_h-1)n_h/2 \approx n_{in}(n_H/2) + n_H^2/4 + n_o(n_H/2) \qquad \text{(Eq. 13)}$$

In general, therefore, $n_h \neq n_H$. Figure 2, for example, plots σ for $n_{in} = \{1, 50, 100, \ldots, 450, 500\}$, $10 \le n_h \le 100$, and $n_o = 1$. We note that for $n_h > 20$, and $n_{in} < 400$, the ratio is upper bounded by $\sigma < 100$. In other words, if our approach reduces the number of epochs by a factor of 100 over standard backpropagation, our approach will be more efficient even for very large input spaces. Moreover, for small input spaces, a mere factor of 5 reduction in the number of epochs will result in increased computational efficiency.

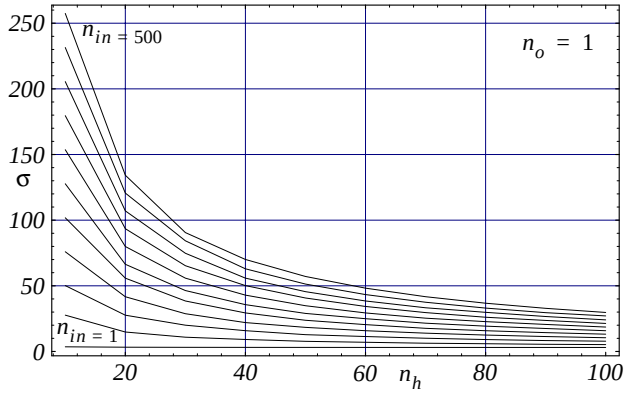

Fig. 2: Ratio of computational costs for various network sizes and one output unit. (Higher curves reflect ratios for larger number of inputs).

Second, we consider the difference in computational cost between our approach (*Ck*) and using NDEKF alone (*Fk*). Let γ_i^{Ck} denote the cost per epoch of training the *ith* hidden unit; let γ^{Ck} denote the total cost of training the *Ck* network (n_h final hidden units and n_c candidate units per hidden unit); let ε_i^{Ck} denote the number of epochs required to train the *ith* hidden unit; and let ε^{Ck} denote the total number of epochs. Also, let γ_ε^{Fk} denote the cost per epoch of training the *Fk* network; let γ^{Fk} denote the total cost of training the *Fk* network (n_h total hidden units); and let ε^{Fk} denote the total number of epochs for training the *Fk* network. Thus,

$$\gamma^{Ck} = \sum_{i=1}^{n_h} \varepsilon_i^{Ck}\gamma_i^{Ck} = n_c \sum_{i=1}^{n_h} \frac{1}{n_c}(\varepsilon_i^{Ck}\gamma_i^{Ck}) \qquad \text{(Eq. 14)}$$

$$\gamma^{Fk} = \varepsilon^{Fk}\gamma_\varepsilon^{Fk} \qquad \text{(Eq. 15)}$$

Now, we assume that,

$$\varepsilon_i^{Ck} \approx \varepsilon_j^{Ck},\ \forall i, j \qquad \text{(Eq. 16)}$$

so that (Eq. 14) becomes,

$$\gamma^{Ck} = n_c \frac{\varepsilon^{Ck}}{n_h} \sum_{i=1}^{n_h} \frac{1}{n_c}\gamma_i^{Ck} \qquad \text{(Eq. 17)}$$

Our experience justifies the approximation in (Eq. 16), which states that all hidden units require approximately the same number of epochs. Furthermore, neglecting differences in derivative calculations between methods *Ck* and *Fk*, we assume that,

$$\gamma_\varepsilon^{Fk} \approx \sum_{i=1}^{n_h} \frac{1}{n_c}\gamma_i^{Ck} \qquad \text{(Eq. 18)}$$

We can now get a relationship between ε^{Ck} and ε^{Fk} corresponding to equivalent costs between methods *Ck* and *Fk*. Setting (Eq. 14) and (Eq. 15) equal to each other and using approximations (Eq. 16) and (Eq. 18), we get that,

$$\varepsilon^{Ck} \approx (n_h/n_c)\varepsilon^{Fk} \qquad \text{(Eq. 19)}$$

In other words, using the cascade/NDEKF (*Ck*) algorithm, we can use approximately n_h/n_c as many epochs as for NDEKF alone (*Fk*) for the same computational cost.

4. Experiments

4.1 Problem descriptions

In this section, we present learning results for five different problems in continuous function approximation and dynamic system modeling. For the first problem (A), we want to approximate the following 3-to-2 *smooth*, continuous-valued mapping,

$$f_1(x, y, z) = z\sin(\pi y) + x \quad \text{(Eq. 20)}$$

$$f_2(x, y, z) = z^2 + \cos(\pi x y) - y^2 \quad \text{(Eq. 21)}$$

in the interval $-1 < x, y, z < 1$. The training set consists of 1000 random points; the cross validation set consists of an additional 1000 random points; and our test set consists of another 2000 random points.

Our second problem (B) is taken from [4]. We want to approximate the following 1-to-1 *nonsmooth*, continuous-valued mapping (see Figure 3),

$$f(x) = \frac{1}{a} \cdot \min\begin{bmatrix} 0.1x^2 \cdot \max(0.5\sin 2x,\ \sin x), \\ x \cdot \max(\sin x,\ \cos^2 x) \end{bmatrix} - b \quad \text{(Eq. 22)}$$

for $a = 34.55386$, and $b = 0.027099$. Our training, cross validation, and test sets are identical to those in [4] and consist of the following: (1) 4000 evenly spaced points are generated in the interval $0 \le x < 20$; (2) 968 of those points are randomly chosen for the training set; (3) 968 are randomly chosen from the remaining 3032 points for the cross validation set; and (4) the remaining 2064 points make up the test set.

Our third problem (C) is taken from [1]. We want to model the following dynamic system,

$$u(k+1) = f[u(k), u(k-1), u(k-2), x(k), x(k-1)] \quad \text{(Eq. 23)}$$

where,

$$f[x_1, x_2, x_3, x_4, x_5] = \frac{x_1 x_2 x_3 x_5 (x_3 - 1) + x_4}{1 + x_3^2 + x_2^2} \quad \text{(Eq. 24)}$$

and the input $x(k)$ is randomly generated in the interval $-1 < x(k) < 1$. We use a 2500-length sequence for training, another 2500-length sequence for cross validation, and another 5000-length sequence for testing.

Finally, our last two problems are taken once again from [4]. Here, we want to predict the chaotic Mackey-Glass time series [8], widely studied in the literature and described by,

$$\dot{x}(t) = \frac{a \cdot x(t-\tau)}{1 + x(t-\tau)^{10}} - b \cdot x(t) \quad \text{(Eq. 25)}$$

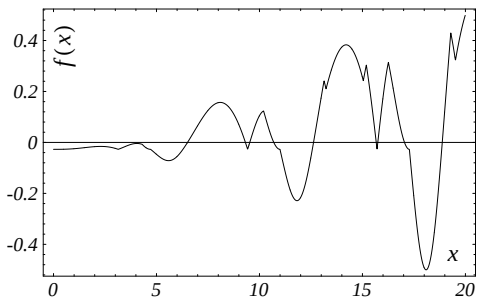

Fig. 3: Nonsmooth, continuous function for problem (B).

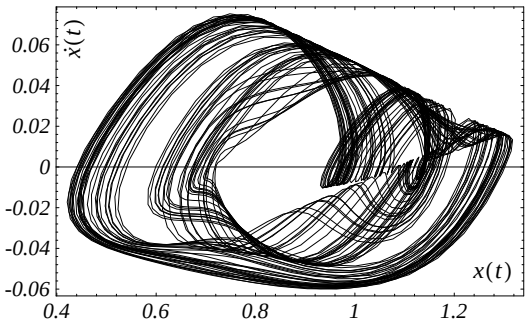

Fig. 4: Phase plot of the Mackey-Glass time series — (D), (E).

for $a = 0.2$, $b = 0.1$, and $\tau = 17$. While the Mackey-Glass differential equation has infinite degrees of freedom (due to the time delay τ), it's stationary trajectory lies on a low-dimensional attractor (as shown in Figure 4). We present

$$\{x(t-18), x(t-12), x(t-6), x(t)\} \quad \text{(Eq. 26)}$$

as the four inputs to the neural network, while the goal of this task is to predict $x(t+k)$ for $k \in \{6, 84\}$. We will refer to $k = 6$ as problem (D) and $k = 84$ as problem (E). Our training, cross validation, and test sets are once again identical to those in [4]. The training set consists of the 500 data points from time $t = 200$ to $t = 699$; the cross validation set consists of the 500 data points from time $t = 1000$ to $t = 1499$; and the test set consists of the 500 points from time $t = 5000$ to $t = 5499$.

For problems (A) and (C) above, we train over 25 trials to 15 hidden units for each method $\{Cq, Fk, Ck\}$. By fixing the network architecture prior to training for *Fk*, it is not possible to assign variable activation functions to each hidden unit; the space of all possible permutations of variable activation functions is too large to explore. Therefore, we try two different networks for method *Fk* — one with sigmoidal activation functions, and the other with sinusoidal activation functions. In previous work [6], we have shown that neural networks with sinusoidal activation functions perform approximately as well as those with variable activation functions. Both have been shown to outperform sigmoidal networks for continuous function approximation.

For problems (B), (D), and (E), taken from [4], we follow the same procedure in training with methods $\{Fk, Ck\}$, as Fahlman, *et. al.*, follow in training with methods $\{Fq, Cq\}$. In [4], *Cq* neural networks are allowed to grow to a maximum of 50 hidden units in 15 separate trials for each problem. For each trial, the best RMS error over the test set is recorded. Equivalently sized *Fq* networks are also trained, for up to 60,000 epochs per trial. The 60,000 figure is chosen to be approximately three times the maximum number of epochs required for any of the *Cq* training runs. Again, the best RMS error for the test set is recorded for each trial.

For the learning results in this paper involving NDEKF, we use the following parameter settings throughout:

$$\eta_Q = 0.0001,\ \eta_P = 0.01 \quad \text{(Eq. 27)}$$

In *Ck*, we upper-bound the number of epochs to 10 per hidden unit, while for *Fk*, we upper-bound the total number of epochs to 150. Finally, for the cascade methods $\{Cq, Ck\}$, we use eight candidate units, the same as in [4].

4.2 Results

Table 2 reports the average RMS error ($\bar{e}_{RMS}$) over the test sets for problems (A) through (E). We note that in all cases, our cascade/NDEKF (*Ck*) approach outperforms the other three methods. Figure 5 reports the percentage difference in $\bar{e}_{RMS}$ between our approach and competing training regimes.

Method *Fq* (fixed-architecture/quickprop) shows by far the worst performance, yet we use 120 times fewer epochs for *Ck* (approximately 500 for problems (B), (D), and (E)). Using Figure 2 as an approximate guide to the computational difference between *Ck* and *Fq*, we see that, for a 50-hidden unit network with relatively few inputs, a *Ck* epoch is no more than 10 times as computationally expensive as an *Fq* epoch. Hence, not only does our cascade/NDEKF approach generate better learning results, it is also more efficient than the fixed-architecture/quickprop approach.

Method *Fk* also performs worse than our *Ck* approach, despite allowing *Fk* to compute as much as twice as long as *Ck*. For problems (A) and (C), for example, the number of epochs required to train to 15 hidden units for *Ck* is approximately 140. Since we use eight candidate units, a roughly equivalent number of epochs in terms of computational cost for *Fk* is (from (Eq. 19)),

$$\varepsilon^{Fk} \approx \frac{n_c}{n_h}\varepsilon^{Ck} = \frac{8}{15}(140) \approx 75 \qquad \text{(Eq. 28)}$$

Yet, we allow *Fk* to compute twice that amount — 150 epochs.

One reason, *Fk* shows worse performance is its susceptibility to getting stuck in bad local minima. As the authors note in [1], "NDEKF at times requires a small amount of redundancy in the network in terms of the total number of nodes in order to avoid poor local minima for certain problems, which [they attribute] to high effective learning rates at the onset of training [1]." Consider, for example, Figure 6 below. While the minimum e_{RMS} for the *Fk* network is below that of the *Cq* network, its maximum e_{RMS} is much worse than either *Ck* or *Cq*. On the other hand, *Ck* avoids the bad local minima problem by iteratively training only a small number of weights in the network at once.

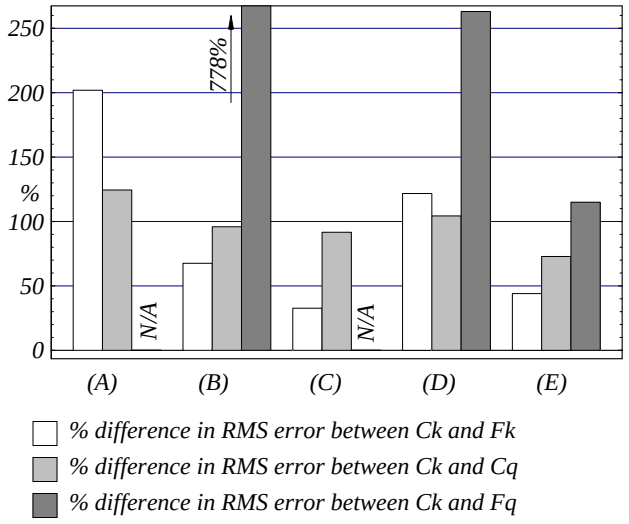

Fig. 5: Cascade/NDEKF significantly outperforms the other learning methods for each problem.

Table 2: Average RMS Error over test sets.[a]

	Ck ($\times 10^{-3}$)	*Fk* ($\times 10^{-3}$)[b]	*Cq* ($\times 10^{-3}$)	*Fq* ($\times 10^{-3}$)
(A)	42.1 (4.2)	127.1 (37.3)	94.5 (6.2)	N/A
(B)	7.4 (2.0)	12.4 (3.2)	14.5 (4.0)	65.0 (18.2)
(C)	15.6 (1.5)	20.7 (4.8)[c]	29.9 (2.0)	N/A
(D)	4.6 (0.6)	10.2 (4.0)	9.4 (2.7)	16.7 (2.2)
(E)	42.0 (5.9)	60.5 (3.1)	72.6 (16.3)	90.3 (8.3)

a. Standard deviations are in parentheses. Shaded cells are results taken from [4].
b. For the *Fk* results we report the better of the sinusoidal or sigmoidal networks (in all cases, the sinusoidal networks did better on average)
c. This is comparable to the result of 0.03 in [1] for a network with an equal number of parameters.

Finally, we look at the difference between the *Ck* and *Cq* methods. First, we note that *Cq* requires about 15 to 25 times as many epochs as does *Ck*. Since each *Cq* epoch is much less computationally expensive, however, *Cq* consumes only about 2/3 the time compared to *Ck* for the problems studied in this paper. On the other hand, *Ck* is able to achieve local minima comparable to *Cq*'s with fewer hidden units, and therefore requires significantly less time than *Cq* to reach the same average RMS error. Consider, for example, Figure 7. At the onset of training, $\bar{e}_{RMS}$ for *Ck* and *Cq* training is approximately equal (4% difference). As hidden units are added, however, we see that $\bar{e}_{RMS}$ diverges for the two training algorithms. Since each hidden unit receives input from all previous hidden units, the input-side weights of the hidden units become increasingly correlated. Figure 8, for example, plots

$$\rho = \sum_{i \neq j} |P_{ij}| / \sum_{i} |P_{ii}| \qquad \text{(Eq. 29)}$$

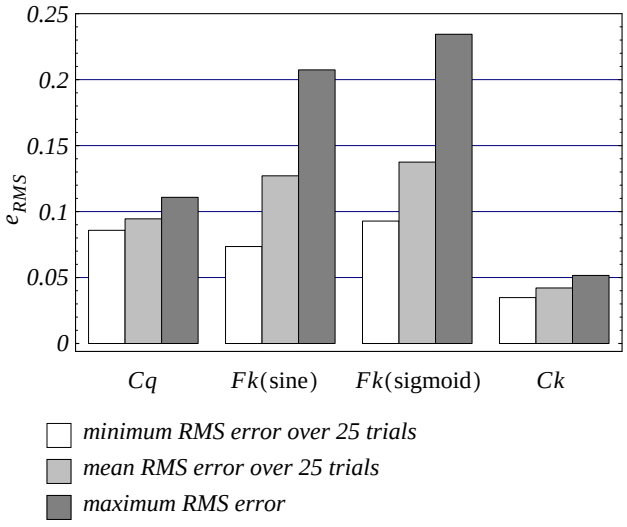

Fig. 6: *Fk* can get stuck in bad local minima, as witnessed by the large maximum RMS errors observed for problem (A).

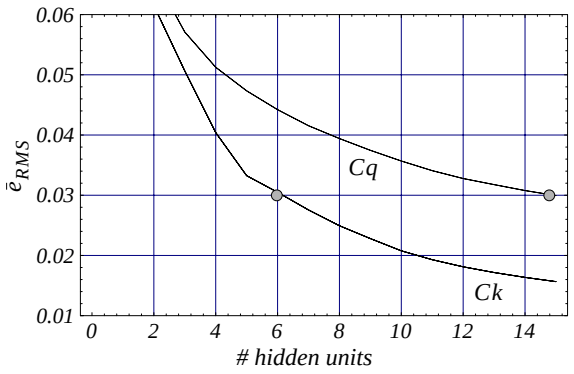

Fig. 7: *Ck* converges to approximately the same avg. RMS error with 6 hidden units (63 weights) as *Cq* does with 15 hidden units (216 weights) for problem (C).

(i.e. the ratio of off-diagonal terms to diagonal terms in the error covariance matrix *P*) for one trial in problem (C). By explicitly storing the interdependence of these weights in the conditional error covariance matrix, cascade/NDEKF copes better with this increasing correlation than does the cascade/quickprop algorithm.

4.3 Discussion

For the problems studied here, we see a significant improvement in learning times and error convergence with cascade/NDEKF over the other methods. Moreover, we see that incremental cascade learning and node-decoupled extended Kalman filtering complement each other well by compensating for each other's weakness. On the one hand, the idea of training one hidden unit at a time and adding hidden units in a cascading fashion offers a good alternative to the *ad hoc* selection of a network architecture. Quickprop and other gradient-descent techniques, however, become less efficient in optimizing increasingly correlated weights as the number of hidden units rises. This is where NDEKF can perform much better through the conditional error covariance matrix. On the other hand, NDEKF can easily become trapped in bad local minima if a network architecture is too redundant. Cascade learning accommodates this well by training only a small subset of all the weights at one time.

Second, throughout the paper we use identical parameter settings for our *Ck* method (Eq. 27). This stands in sharp contrast to the gradient-descent methods (i.e. *Cq* and *Fq)*, for which learning

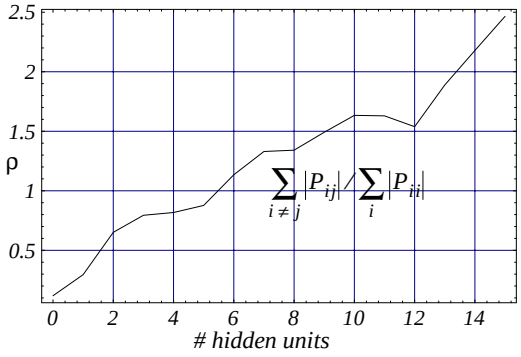

Fig. 8: Weights become increasingly correlated as hidden units are added to the network [problem (C)].

parameters were tuned for each particular problem in order to achieve good results [4]. The NDEKF weight-update recursion in (Eq. 3) can be thought of as an adaptive learning rate, which obviates the need for parameter tuning; thus, in our approach we need not waste time tuning parameters or network architectures.

Finally, while our approach performs well for the problems studied in this paper, it is clearly impractical for applications which have a large number of inputs and/or outputs. This tends to exclude vision-based tasks, where the input and/or output spaces are typically greater than 1000. Applications with inputs numbering in the low hundreds, however, are not excluded. For example, we are currently applying cascade/NDEKF learning successfully to modeling human control strategy, where the number of inputs typically range from 100 to 200.

5. Conclusion

In this paper, we have developed a new learning architecture for continuous-valued function approximation and dynamic system identification, which combines cascade neural networks and node-decoupled extended Kalman filtering. We show that incremental cascade learning and NDEKF complement each other well by compensating the other's weakness, and that the combination forms a powerful learning architecture, which records quicker convergence to better local minima than related neural-network training paradigms.

Acknowledgments

We thank Doug Baker and Scott Fahlman for generously giving us access to data and results for problems (B), (D) and (E).

References

[1] G. V. Puskorius and L. Feldkamp, "Decoupled Extended Kalman Filter Training of Feedforward Layered Networks," *Proc. Int. Joint Conf. on Neural Networks*, vol. 1, pp. 771-777, 1991.

[2] M. C. Nechyba and Y. Xu, "Human Control Strategy: Abstraction, Verification and Replication," to appear in *IEEE Control Systems Magazine*, October, 1997.

[3] S. E. Fahlman, "An Empirical Study of Learning Speed in Back-Propagation Networks," Technical Report, CMU-CS-TR-88-162, Carnegie Mellon University, 1988.

[4] S. E. Fahlman, L. D. Baker and J. A. Boyan, "The Cascade 2 Learning Architecture," Technical Report, CMU-CS-TR-96-184, Carnegie Mellon University, 1996.

[5] S. Singhal and L. Wu, "Training Multilayer Perceptrons with the Extended Kalman Algorithm," *Advances in Neural Information Processing Systems 1*, ed. Touretzky, D. S., Morgan Kaufmann Publishers, pp. 133-140, 1989.

[6] M. C. Nechyba and Y. Xu, "Towards Human Control Strategy Learning: Neural Network Approach with Variable Activation Functions, Technical Report, CMU-RI-TR-95-09, Carnegie Mellon University, 1995.

[7] M. C. Nechyba and Y. Xu, "Neural Network Approach to Control System Identification with Variable Activation Functions," *Proc. IEEE Int. Symp. Intelligent Control*, pp. 358-363, 1994.

[8] M. C. Mackey and L. Glass, "Oscillations and Chaos in Physiological Control Systems," *Science*, vol. 197, no. 4300, pp. 287-289, 1977.