CHAPTER 1

A CONSTRAINED BACKPROPAGATION (CPROP) APPROACH TO FUNCTION APPROXIMATION AND APPROXIMATE DYNAMIC PROGRAMMING

SILVIA FERRARI, KEITH RUDD, AND GIANLUCA DI MURO

Duke University, Durham, North Carolina

1.1 INTRODUCTION

The ability of preserving prior knowledge in an artificial neural network (ANN) while incrementally learning new information is important to many fields, including approximate dynamic programming (ADP), feedback control, and function approximation. Although ANNs exhibit excellent performance and generalization abilities when trained in batch mode, when they are trained incrementally with new data they tend to forget previous information due to a phenomenon known as interference. McCloskey and Cohen [26] and [32] were the first to suggest that a fundamental limitation of ANNs is that the process of learning a new set of patterns may suddenly and completely erase a network's knowledge of what it had already learned [12]. This phenomenon, known as catastrophic interference or catastrophic forgetting, seriously limits the applicability of ANNs to adaptive feedback control, and incremental function approximation. Natural cognitive systems learn most tasks incrementally, and need not re-learn prior patterns in order to retain them in their long-term memory

Reinforcement Learning and ADP for Feedback Control. By Frank L. Lewis and Derong Liu

(LTM) during their lifetime. Catastrophic interference in ANNs is caused by their very ability to generalize using a single set of shared connection weights, and a set of interconnected nonlinear basis functions. Therefore, the modular and sparse architectures that have been proposed so far for suppressing interference also limit a neural network's ability to approximate and generalize highly nonlinear functions. This chapter describes how constrained backpropagation (CPROP) can be used to preserves prior knowledge while training ANNs incrementally through ADP, and to solve differential equations, or approximate smooth nonlinear functions online.

1.2 BACKGROUND

Some of the most significant advances on preserving memories in ANNs to date have been made in the field of self-organizing networks and associative memories (e.g., [4, 13, 15, 17, 40]). In these networks, the neurons use competitive learning to recognize groups of similar input vectors and associate them with a particular output by allowing neurons that are physically near each other to respond to similar inputs. Interference has also been suppressed successfully in NN classifiers, using Learn++ algorithms that implement a weighted voting procedure to retain long-term episodic memory [30]. Although these methods are very important to pattern recognition and classification, they are not applicable to preserving functional knowledge in ANNs, as may be required for example by feedback control. Although ADP aims to improve existing ANN approximations of the control law and value function, it can greatly benefit from the ability to retain control knowledge in the long term, and, generally, from the elimination of interference. While associative memories in self-organizing networks resemble declarative memories for recalling episodes or facts, CPROP aims to establish *procedural memories*, which refer to cognitive and motor skills, such as the ability to ride a bike or fly an airplane [6].

The problem of interference in nonlinear differentiable NNs has been addressed along two main lines of research. One approach presents some LTM data together with short-term memory (STM) data, and has been proven effective for supervised radial-basis networks with compact support [16, 30, 38, 39]. While useful, this approach is not suited to ANN implementations that require LTM to be preserved reliably (e.g., control systems), nor to implementations that have stringent computational requirements due, for example, to high-dimensional input-output spaces, large training sets, or repeated incremental training over time, such as ADP. Another approach consists of partitioning the weights into two subsets, one that is used to preserve LTM by holding the weights' values constant, and one that is updated using the new STM data [18, 24]. Although effective in some applications, this approach cannot guarantee LTM preservation and may not suppress interference in nonlinear neural networks with global support (Section 1.3.3). Similarly to [18, 24], CPROP partitions the weights into STM and LTM subsets. But, in CPROP, both subsets updated based on new STM data at every epoch of the training algorithm as follows. While STM weights are updated to learn new STMs, the LTM weights are updated to preserve LTMs subject to the STM-weight changes.

1.3 CONSTRAINED BACKPROPAGATION (CPROP) APPROACH

Neural network training is typically formulated as an unconstrained optimization problem involving a scalar function $e : \mathbb{R}^N \to \mathbb{R}$, with respect to the network weights $\mathbf{w} \in \mathbb{R}^N$. The scalar function may consist of the the neural network output error, or of an indirect measure of performance, such as the cost-to-go in an ADP algorithm. By optimizing e, the training algorithm seeks to obtain a neural network representation of an unknown vector function $\mathbf{y} = \mathbf{h}(\mathbf{p})$, with input $\mathbf{p} \in \mathbb{R}^r$, and output $\mathbf{y} \in \mathbb{R}^m$. Assume the long-term memory (LTM) knowledge of the function to be approximated can be embedded into a functional relationship describing the network weights such as,

$$g(\mathbf{w}_L, \mathbf{w}_S) = \mathbf{0} \tag{1.1}$$

where $\mathbf{w}$ has been reorganized into a vector of LTM weights $\mathbf{w}_L \in \mathbb{R}^{N_L}$, and a matrix of STM weights $\mathbf{w}_S \in \mathbb{R}^{N_S}$, with $N_L + N_S = N$. As shown in Sections 1.3.2, the equality constraint (1.1) can be derived from the neural network equation and various forms of available prior knowledge. Then, training preserves the LTM expressed by (1.1) provided it is carried out according to the following constrained optimization problem:

$$\begin{aligned} &\text{minimize} \quad e(\mathbf{w}_L, \mathbf{w}_S) \\ &\text{subject to} \quad g(\mathbf{w}_L, \mathbf{w}_S) = \mathbf{0} \end{aligned} \tag{1.2}$$

The solution of a constrained optimization problem can be provided by the method of Lagrange multipliers or by direct elimination. If (1.1) satisfies the implicit function theorem, then it uniquely implies the function,

$$\mathbf{w}_L = \mathcal{C}(\mathbf{w}_S) \tag{1.3}$$

and the method of direct elimination can be applied by writing the error function as,

$$E(\mathbf{w}_S) = e(\mathcal{C}(\mathbf{w}_S), \mathbf{w}_S) \tag{1.4}$$

such that the value of $\mathbf{w}_S$ can be determined independently of $\mathbf{w}_L$. In this case, the solution of (1.2) is an extremum of (1.4) that obeys,

$$\nabla_{\mathbf{w}_S} E = \mathbf{0} \tag{1.5}$$

where, the gradient ∇ is defined as a column vector of partial derivatives taken with respect to every element of the subscript $\mathbf{w}_S$. Once the optimal value of $\mathbf{w}_S$ is determined, the optimal value of the weights $\mathbf{w}_L$ can be obtained from $\mathbf{w}_S$ using (1.3). If the equality constraint cannot be written as (1.3), the method of Lagrange multipliers can be used to solve (1.2).

Hereon, it is assumed that the equality constraint can be written in explicit form (1.3). Furthermore, since (1.3) can be very involved, its substitution in the error function is circumvented by seeking the extremum defined by the *adjoined error*

gradient, obtained by the chain rule,

$$\nabla_{\mathbf{w}_S} E := \left\{ \frac{\partial E}{\partial w_{S_i}} \right\} = \left\{ \frac{\partial e}{\partial w_{S_i}} + \frac{\partial e}{\partial \mathcal{C}} \frac{\partial \mathcal{C}}{\partial w_{S_i}} \right\} \tag{1.6}$$

where w_{S_i} is the i^{th} element of $\mathbf{w}_S$.

The constrained training approach is applicable to incremental training of neural networks for smooth function approximation under the following assumptions: (i) *a-priori* knowledge of the function is available locally in its domain (e.g., in the form of a batch training set, or a physical model); (ii) it can be expressed as an equality constraint on the neural network weights; (iii) it is desirable to preserve this prior knowledge during future training sessions; (iv) new functional information must be assimilated incrementally through domain exploration; and, (v) the new information is consistent with the prior knowledge (i.e., the function to be approximated is one-to-one and the information is noise-free). Then, constrained training can be implemented through the following steps: (I) Determine the LTM equality constraint (1.1) for the chosen ANN architecture; (II) Determine the neural network size, and label the LTM-STM weights $\mathbf{w}_S$ and $\mathbf{w}_L$; (III) Rewrite (1.1) in the explicit form (1.3); (IV) Compute the adjoined gradient (1.6) or Jacobian analytically; (V) Implement adjoined gradient or Jacobian in a chosen backpropagation-based algorithm. The CPROP equations utilized in steps (I)-(IV) are derived in the following subsections.

1.3.1 Neural Network Architecture and Procedural Memories

A feedforward, one-hidden-layer, sigmoidal architecture is chosen in this chapter because of its universal function approximation ability, and its broad applicability. The hidden layer can be represented by a diagonal operator with repeated sigmoids $\mathbf{\Phi}(\mathbf{n}) := [\sigma(n_1) \cdots \sigma(n_s)]^T$, where n_i denotes the i^{th} component of the input-to-node vector $\mathbf{n} \in \mathbb{R}^{s\times 1}$. The sigmoidal function $\sigma(n_i) : \mathbb{R} \to \mathbb{R}$ is assumed to be a bounded measurable function on $\mathbb{R}$ for which $\sigma(n_i) \to 1$ as $n_i \to \infty$, and $\sigma(n_i) \to 0$ as $n_i \to -\infty$. In this chapter, the sigmoid of choice is $\sigma(n_i) := (e^{n_i} - 1)/(e^{n_i} + 1)$. Then, the neural network input-output equation,

$$\hat{\mathbf{y}}(\mathbf{p}) = \mathbf{V}\mathbf{\Phi}(\mathbf{W}\mathbf{p} + \mathbf{d}) := \mathbf{V}\mathbf{\Phi}[\boldsymbol{\nu}(\mathbf{p})] \tag{1.7}$$

can be written in terms of linear *input-to-node operator*, $\boldsymbol{\nu} : \mathbb{R}^r \to \mathbb{R}^s$, which maps the input space into node space. Where, $\mathbf{d} \in \mathbb{R}^{s\times 1}$, $\mathbf{W} \in \mathbb{R}^{s\times r}$ and $\mathbf{V} \in \mathbb{R}^{m\times s}$, are the adjustable bias, and input and output ANN weights, respectively.

The *long-term memory (LTM)* of the ANN (1.7) is defined as the input-output and gradient information for the unknown function $\mathbf{h} : \mathbf{p} \to \mathbf{y}$ that must be preserved at all times, during one or more incremental-training session. It is assumed that $\mathbf{h}$ is many-to-one over a domain $\mathbf{p} \in \mathcal{P}$. The LTM may comprise sampled output and derivative information, or information about the functional form over a bounded subset $\mathcal{D} \subset \mathcal{P}$. The *short-term memory (STM)* of the ANN (1.7) is defined as the sequence of skills (e.g. control laws) or information that must be learned through one

or more training functions $\{e_k(\mathbf{w})\}_{k=1,2,...}$. For simplicity, it is assumed that the STM needs not be consolidated into LTM. CPROP utilizes algebraic neural network training [11], and the adjoined Jacobian described in the next subsection. By this approach, any backpropagation-based algorithm can be modified to retain the ANN's LTM during training, simply by redefining the error gradient.

1.3.2 Derivation of LTM Equality Constraints and Adjoined Error Gradient

Classical backpropagation-based algorithms minimize the scalar function e using an unconstrained optimization approach that is based on the gradient of e with respect to the ANN weights, $\nabla_{\mathbf{w}} e \in \mathbb{R}^N$. Since e often represents the ANN output error, they are commonly referred to as error backpropagation (EBP) algorithms. Examples of unconstrained optimization algorithms that have been utilized for ANN training are steepest descent, conjugate gradient, and Newton's method. In every case, the first- or second-order derivatives of e with respect to $\mathbf{w}$ are utilized to minimize e, and backpropagation refers to a convenient approach for computing these derivatives across the ANN hidden layer(s), e.g. $\mathbf{\Phi}$ in (1.7). Then, the definition of e determines the training style. In *supervised training* e is an error function representing the distance between the ANN output and the output data sampled from $\mathbf{h}$, and organized in a training set $\mathcal{T} = \{\mathbf{p}_k, \mathbf{y}_k\}_{k=1,2,...}$. Where, every sample satisfies the function to be approximated, i.e. $\mathbf{y}_k = \mathbf{h}(\mathbf{p}_k)$, for all k. In reinforcement learning and ADP e may be an indirect measure of performance, such as the value function, or the improved policy, in which case the weight update includes the temporal difference error. In *batch training* the information is presented all at once, by defining e as a sum over all training pairs. Whereas in *incremental training* the information is presented one sample at-a-time, or one subset at-a-time in batch mode. In every one of these instances, the chosen backpropagation-based algorithm can be constrained to preserve LTM by backpropagating a so-called adjoined gradient (or Jacobian, depending on the algorithm) that can be computed conveniently across the hidden layer using the approach described in this section.

The approach is illustrated for LTM that can be expressed by a training set of input-output samples and derivative information denoted by $\mathcal{T}_L = \{\mathbf{p}_\ell, \mathbf{y}_\ell, \boldsymbol{\chi}_\ell\}_{\ell=1,...,K}$, where $\mathbf{y}_\ell = \mathbf{h}(\mathbf{p}_\ell)$, $\boldsymbol{\chi}_\ell = \nabla_{\mathbf{p}} \mathbf{h}(\mathbf{p}_\ell)$, and $\mathbf{p}_\ell \in \mathcal{D}$ for all ℓ. If the functional form of $\mathbf{h}$ over $\mathcal{D}$ is known, then it can be sampled to produce $\mathcal{T}_L$. Whenever possible, derivative information should be incorporated in order to improve ANN generalization, and prevent overfitting. In general, the ANN performance may depend on a vector function $\boldsymbol{\epsilon}$, such as the ANN output error, or the gradient of the cost-to-go. Then, the STM scalar function to be minimized during training can be expressed by the quadratic form,

$$e(\mathbf{w}) = \frac{1}{q} \sum_{k=1}^{q} \boldsymbol{\epsilon}_k^T(\mathbf{w}) \boldsymbol{\epsilon}_k^T(\mathbf{w}) \tag{1.8}$$

where q is the number of STM samples available during the training session. In this chapter, the Levenberg-Marquardt (LM) is the EBP algorithm of choice, because of its excellent convergence and stability properties [21, 25].

In the classical, unconstrained case, the LM algorithm iteratively minimizes e with respect to $\mathbf{w}$, based on the ANN *unconstrained* Jacobian, $\mathbf{J} = \partial \boldsymbol{\epsilon}_k / \partial \mathbf{w}$ [20, 37]. An *input-to-node matrix* $\mathbf{N}$ can be defined by applying $\boldsymbol{\nu}$ to every one of the training samples used to define e in (1.8). For a training set $\mathcal{T}$ with q samples, let $\mathbf{N} := [\boldsymbol{\nu}(\mathbf{p}_1) \cdots \boldsymbol{\nu}(\mathbf{p}_q)] = \mathbf{WP} + \mathbf{D}$, where matrices $\mathbf{P} \in \mathbb{R}^{r \times q}$ and $\mathbf{D} \in \mathbb{R}^{s \times q}$ are defined as,

$$\mathbf{P} := [\mathbf{p}_1 \cdots \mathbf{p}_q], \quad \text{and} \quad \mathbf{D} := [\underbrace{\mathbf{d} \cdots \mathbf{d}}_{q}]. \tag{1.9}$$

The *unconstrained* higher-order derivatives of e, say of order i, can be computed as follows,

$$\mathbf{J}^i := \frac{\partial \boldsymbol{\epsilon}_k^i}{\partial \mathbf{w}^i} = [\mathbf{J}_{\mathbf{w}}^i \mid \mathbf{J}_{\mathbf{d}}^i \mid \mathbf{J}_{\mathbf{v}}^i]$$

where,

$$\mathbf{J}_{\mathbf{w}}^i = i\, \mathbf{S}^i \mathbf{W}_d^{i-1} \mathbf{V}_d + \mathbf{P} \mathbf{S}^{i+1} \mathbf{W}_d^i \mathbf{V}_d \tag{1.10}$$

$$\mathbf{J}_{\mathbf{d}}^i = \mathbf{S}^{i+1} \mathbf{W}_d^i \mathbf{V}_d \tag{1.11}$$

and

$$\mathbf{J}_{\mathbf{v}}^i = \mathbf{S}^i \mathbf{W}_d^i \tag{1.12}$$

Where, $\mathbf{S}^i(\mathbf{N}) \equiv \{\sigma^i(n_{ij})\}$, and $\sigma^i(\cdot)$ denotes the i^{th} derivative of the sigmoidal function, evaluated at every element of the input-to-node matrix $\mathbf{N}$.

In order to preserve the LTM, the equality constraints (1.1) are derived from the training set $\mathcal{T}_L$ using ANN algebraic training [11]. According to [11], a training set in the form of $\mathcal{T}_L$ can be matched exactly by an ANN with equation (1.7) provided there are $s_L = K$ (LTM) nodes in the hidden layer. An additional number of nodes, denoted by s_S, is added to allow the ANN to assimilate additional (STM) information during incremental training. s_S can be chosen by the user, or set equal to q based on [11], such that $s = s_L + s_S$. Subsequently the ANN weights associated with the LTM nodes, denoted by $\mathbf{W}_L$, $\mathbf{d}_L$, $\mathbf{V}_L$, are used to satisfy the LTM constraints at all times, and the ANN weights associated with the STM nodes, denoted by $\mathbf{W}_S$, $\mathbf{d}_S$, $\mathbf{V}_S$, are used to acquire the STM, via constrained LM. Then, the LTM training set is satisfied at all times provided the following set of algebraic equations is:

$$\mathbf{N} = \mathbf{WP} + \mathbf{D} \tag{1.13}$$

$$\mathbf{Y} = \mathbf{S}^0 \mathbf{V}^T \tag{1.14}$$

$$\mathbf{C} = \mathbf{S}^m \mathbf{V}_d \mathbf{W}_d \tag{1.15}$$

Where $\mathbf{P} = [\mathbf{p}_1 \cdots \mathbf{p}_K]$, $\mathbf{Y} = [\mathbf{y}_1 \cdots \mathbf{y}_K]$ and $\mathbf{C} = [\boldsymbol{\chi}_1 \cdots \boldsymbol{\chi}_K]$ are known and constant matrices containing the LTM input-output and derivative information obtained from $\mathcal{T}_L$. The above equality constraint, in the form (1.1), can be transformed into the explicit form (1.3) by re-writing (1.14) and (1.15) in terms of the STM and

LTM weights,

$$\mathbf{M} := \left[\begin{array}{c} \mathbf{Y} \\ \mathbf{C} \end{array} \right] = \left[\begin{array}{c} \mathbf{S}_L^0 \\ \mathbf{S}_L^1 \end{array} \right] \mathbf{V}_L + \left[\begin{array}{c} \mathbf{S}_S^0 \\ \mathbf{S}_S^1 \end{array} \right] \mathbf{V}_S := \mathbf{\Psi} \mathbf{V}_L + \mathbf{\Omega} \mathbf{V}_S \tag{1.16}$$

where $\mathbf{D}_{L,S} := [\mathbf{d}_{L,S} \cdots \mathbf{d}_{L,S}]$, and the shorthand notation $\mathbf{S}_{L,S}^i := \mathbf{S}^i(\mathbf{W}_{L,S}\mathbf{P} + \mathbf{D}_{L,S})$ is adopted for simplicity. Then, the explicit constraint (1.3) takes the form,

$$\mathbf{V}_L = \mathbf{\Psi}^{-1}[\mathbf{M} - \mathbf{\Omega}\mathbf{V}_S] \tag{1.17}$$

where $\mathbf{\Psi}$ is a known, constant and invertible matrix by design of $\mathcal{T}_L$. Now, let $\mathbf{V}'$ and $\mathbf{W}'$ denote sparse, block-diagonal matrices formed from $\mathbf{V}$ and $\mathbf{W}$, respectively. Then, the adjoined gradient is,

$$\breve{\nabla}_{\mathbf{V}_S} E = \nabla_{\mathbf{V}_S} e - \mathbf{S}_L^0 \mathbf{\Omega}^{-1} \left[\begin{array}{cc} \mathbf{P}\mathbf{S}_S^1 \mathbf{V}'_S & \mathbf{S}_S^1 \mathbf{V}'_S \quad \mathbf{S}_S^0 \\ (\mathbf{S}_S^1 \mathbf{V}'_S + \mathbf{P}\mathbf{S}_S^2 \mathbf{W}'_S \mathbf{V}'_S) & \mathbf{S}_S^2 \mathbf{W}'_S \mathbf{V}'_S \quad \mathbf{S}_S^1 \mathbf{W}'_S \end{array} \right] \tag{1.18}$$

Where, it can be seen that the adjoined gradient can be obtained by modifying the unconstrained gradient $\nabla_{\mathbf{V}_S} e$ by a term that contains distinct contributions from the LTM and STM weights and, thus, can be easily introduced and updated in any backpropagation-based algorithm.

By extending (1.6) to the Jacobian utilized by the LM algorithm, the adjoined Jacobian obtained subject to the LTM constraint (1.17) can be written as,

$$\begin{aligned} \breve{\mathbf{J}} &= \frac{\partial \boldsymbol{\epsilon}_k}{\partial \mathbf{w}_S} = \frac{\partial}{\partial \mathbf{w}_S} \left[\mathbf{S}_S^0 \mathbf{V}_S + \mathbf{S}_L^0 \mathbf{\Omega}^{-1} (\mathbf{M} - \mathbf{\Psi} \mathbf{V}_L) \right] && (1.19) \\ &= \mathbf{J} - \mathbf{S}_L^0 \mathbf{\Omega}^{-1} \left[\mathbf{H}_{\mathbf{W}_S} \mid \mathbf{H}_{\mathbf{d}_S} \mid \mathbf{H}_{\mathbf{V}_S} \right] && (1.20) \end{aligned}$$

where:

$$\begin{aligned} \mathbf{H}_{\mathbf{W}_S} &= \left[\begin{array}{c} \mathbf{P}\mathbf{S}_S^1 \mathbf{V}'_S \\ \mathbf{S}_S^1 \mathbf{V}'_S + \mathbf{P}\mathbf{S}^2 \mathbf{W}'_S \mathbf{V}'_S \end{array} \right] \\ \mathbf{H}_{\mathbf{d}_S} &= \left[\begin{array}{c} \mathbf{S}_S^1 \mathbf{V}'_S \\ \mathbf{S}_S^2 \mathbf{W}'_S \mathbf{V}'_S \end{array} \right] \\ \mathbf{H}_{\mathbf{V}_S} &= \left[\begin{array}{c} \mathbf{S}_S^0 \\ \mathbf{S}_S^1 \mathbf{W}'_S \end{array} \right] \end{aligned} \tag{1.21}$$

1.3.3 Example: Incremental Function Approximation

Since the most general interpretation of ANN training is function approximation, in this section CPROP is illustrated through a simple example, taken from [5], involving a scalar nonlinear function $h : p \to y$, to be approximated over a bounded domain $p \in \mathcal{P}$ by a one-layer sigmoidal neural network (1.7). Consider the nonlinear function plotted by a dashed line in Fig. 1.1 over a domain $\mathcal{P} = [0,\ 3\pi] \subset \mathbb{R}$. Suppose the shape of the function over a bounded subset $\mathcal{D} = [0,\ \pi] \subset \mathcal{P}$ is known a priori to be a sine function, and the LTM training set $\mathcal{T}_L$ is formed using the LTM samples shown in Fig. 1.1. Then, the LTM training set takes the form $\mathcal{T}_L = \{p_\ell, y_\ell, \partial y/\partial p|_\ell\}_{\ell=1,\ldots,K}$,

with $K = 4$, and $p_\ell \in \mathcal{P}$ for all ℓ. After learning $\mathcal{T}_L$ in batch mode, the ANN is re-trained using new STM data, $\mathcal{T}_S = \{p_k, y_k\}_{k=1,\ldots,q}$, where in this case $y_k \in \{\mathcal{P}/\mathcal{D}\}$, even though it is not a requirement for the algorithm. Thus, in this example, two subsets of sampled data are presented incrementally to the ANN, as could come about when the two subsets are too large to be presented at once, or when $\mathcal{T}_S$ becomes available at a later time, e.g., in on-line learning.

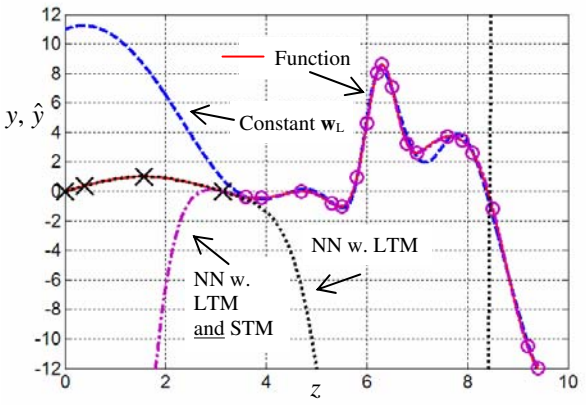


Figure 1.1 Function to be approximated incrementally and performance of existing algorithms (taken from [5]).

As illustrated in Fig. 1.1, when trained with existing algorithms, the neural network may forget the LTM while learning the STM data due to interference. Here, a sigmoidal ANN with 15 hidden nodes is trained to approximate $\mathcal{T}_{LTM}$ using the LM algorithm available from the MATLAB® Neural Network Toolbox [1]. At a later time, 18 new STM samples become available (Fig. 1.1), the NN is re-trained by the same LM algorithm using $\mathcal{T}_{STM}$. If training is conducted sequentially, without re-using the LTM data, the NN starts out with proper LTM (dotted line in Fig. 1.1), but then experiences catastrophic interference and, although it learns STM well, it forgets the LTM entirely in the process (dashed-dotted line in Fig. 1.1). Figure 1.1 also shows the performance of a NN that is trained with the incremental training method proposed by Mandziuk [24]. That is, LTM weights ($\mathbf{w}_L$) are held constant while learning from $\mathcal{T}_{STM}$. As shown by the dashed line in Fig. 1.1, when the NN is trained by this method, it still experiences interference and forgets the LTM. Instead, when the LM algorithm is constrained by implementing the above Jacobian and memory constraint, the LTM is preserved at every epoch, and the NN learns the STM without forgetting the LTM, as shown by Fig. 1.2.

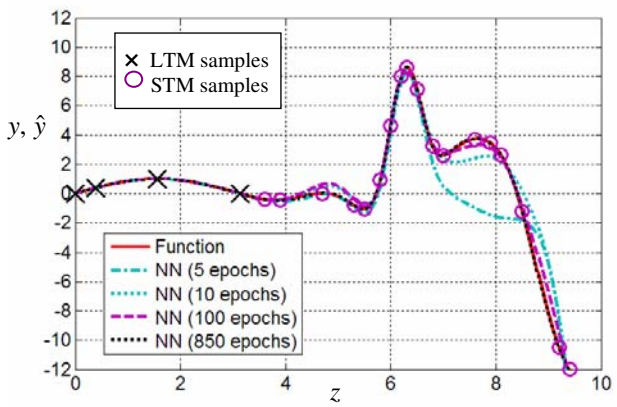


Figure 1.2 CPROP preserves LTM accurately at every epoch, until it properly learns the STM at approximately 850 epochs (taken from [5]).

1.4 SOLUTION OF PARTIAL DIFFERENTIAL EQUATIONS IN NONSTATIONARY ENVIRONMENTS

Neural networks are often implemented to provide a functional representations of numerical partial differential equation (PDE) solutions that are amenable to mathematical analysis and data assimilation algorithm. In many applications, a numerical algorithm, such as finite difference is first utilized to obtain the numerical solution in discrete form (e.g. look-up table), and then the solution is utilized to train a neural network using a conventional backpropagation algorithm. Methods have also been proposed to determine the PDE solution in one step, by training an ANN to minimize an error function formulate in terms of the differential operator. Different techniques have been developed to take into account the boundary conditions. One line of research [19] expresses the solution as the sum of two functions. One function is problem dependent and is designed by the user to satisfy the boundary conditions (BCs) with no adjustable parameters. The second function is an ANN trained to approximate the differential operator. However, this approach only is applicable to PDEs defined on orthogonal box domains, and cannot be extended to non-stationary environments in which the BCs change over time because part of the solution must be designed by the user off-line. A more recent study [19] overcomes some of these limitations, but adopts radial basis functions to correct the solution on the boundaries, making difficult to implement this technique in non-stationary environments, when the shape of the boundaries may change over time.

Another approach consists of embedding the BCs in the cost function [2, 35]. Although this approach has been shown effective in solving linear PDEs, in order to match BCs with high accuracy, which may be crucial in many engineering applications, it require many points on the integration domain boundaries, thereby increasing the computational cost dramatically. Also, it relies on evolutionary algorithms that

are computationally expensive and, typically, can only be implemented reliably off line in non-stationary environments. A methodology based on radial basis functions ANNs was developed in [14], by learning the adjustable parameters using a two-stage gradient-descent strategy. One advantage of this methodology is that the ANN architecture is adapted over time using a node insertion strategy. However, because it includes the BCs in the cost function, this methodology suffers from the same aforemention limitations, as [2]. By incorporating equality constraints of any form systematically in any nonlinear ANN training process, CPROP overcomes these existing limitations, and enables to solve the PDE adaptively in nonstationary environments.

1.4.1 CPROP Solution of Linear and Nonlinear PDEs

Consider the nonlinear PDE,

$$\mathcal{D}^k \left[u(\mathbf{y}) \right] = f(\mathbf{y}) \tag{1.22}$$

where $\mathcal{D}^k$ is a non-linear differential operator of order k, $\mathbf{y} \in \mathcal{I} \subset \mathbb{R}^r$, and $\mathcal{I} \in \mathbb{R}^r$ is a compact set with associated boundary conditions (BCs),

$$\mathcal{G}^j \left[u(\mathbf{y}) \right] = h(\mathbf{y}) \tag{1.23}$$

$\mathcal{G}$ is a linear operator of order $j < k$. The functions $\mathbf{y} \in \partial\mathcal{I} \subset \mathbb{R}^r$ and $f, h : \mathbb{R}^r \rightarrow \mathbb{R}$ are assumed to be continuous and known. Without loss of generality, assume that $\mathcal{D}^k = \mathcal{L}^{k_1} + \mathcal{H}^{k_2}$, where $k = \max\{k_1, k_2\}$, $\mathcal{L}^{k_1}$ is a linear differential operator of order k_1, and $\mathcal{H}^{k_2}$ is a nonlinear differential operator of order k_2 of the form,

$$\mathcal{H}^{k_2} = \sum_{m=1}^{r} \sum_{l=1}^{k_2} \sum_{r=1}^{R_l} c_{lmr} \, \frac{\partial^l u}{\partial y_m^l} \, u_r(\mathbf{y})$$

The solution $\hat{u}(\mathbf{y})$ is provided by the output of an ANN in the form (1.7), with scalar output ($m = 1$), and adjustable parameters $\mathbf{W}, \mathbf{d}, \mathbf{v}$. Then, the input-to-node operator is defined as,

$$\boldsymbol{\nu}\left(\mathbf{y}\right) = \mathbf{W}\,\mathbf{y} + \mathbf{d} \tag{1.24}$$

where, $\mathbf{W} \in \mathbb{R}^{s \times r}, \mathbf{d} \in \mathbb{R}^s$ are the input weights and bias, respectively, and the sigmoidal operator is, as before, defined as the nonlinear mapping $\boldsymbol{\Phi}(\mathbf{n}) : \mathbb{R}^s \rightarrow \mathbb{R}^s$ across the hidden (nonlinear) layer. Hence, the approximation of the solution of the PDE (1.22) is given by the ANN output provided it satisfies the relationship,

$$\hat{u}(\mathbf{y}) = \boldsymbol{\Phi}\left[\boldsymbol{\nu}(\mathbf{y})\right] \mathbf{v}^T \tag{1.25}$$

where $\mathbf{v} \in \mathbb{R}^{1 \times s}$ is a vector of output weights. It is then trivial to extend the approach to the case of multiple outputs' network because of their linearity with respect to the output weights. Substituting (1.25) into (1.22), the differential operator is applied to

the ANN,

$$\mathcal{D}^k \left\{ \mathbf{\Phi} \left[\boldsymbol{\nu}(\mathbf{y}) \right] \mathbf{v}^T \right\} = f(\mathbf{y}) \tag{1.26}$$

Since the aim is to approximate the solution of problem (1.22) on $\mathcal{D}$, the STM set is formed by input samples $\mathcal{T}_S = \{\mathbf{y}_k \in \mathcal{I}, k = 1, \ldots, q\}$ of the PDE solution that satisfies (1.22). Then, by applying the input-to-node operator (1.24) to every input sample, the following input-to-node matrix is obtained,

$$\mathbf{N} = \left[\mathbf{W}\mathbf{Y} + \mathbf{D} \right]^T \tag{1.27}$$

where, $\mathbf{Y} = [\mathbf{y}_1 \cdots \mathbf{y}_q]$ is an $r \times q$ matrix of input samples, and $\mathbf{N} \in \mathbb{R}^{q \times s}$. As before, the $q \times s$ matrix defined as $\mathbf{S}^0 = \mathbf{\Phi}(\mathbf{N})$ is utilized to derive the LTM constraints below, as well as to re-write the output of the ANN for all STM training samples in matrix form, i.e.:

$$\hat{u}(\mathbf{y})|_{\mathbf{y} \in \mathcal{T}_{\text{STM}}} = \mathbf{S}^0 \mathbf{v}^T \tag{1.28}$$

In order to solve (1.22), the ANN output must be differentiated with respect to its inputs up to the k^{th}-order derivative.

After some manipulations it is possible to extend the scalar equations presented in [19], adopting the operators previously introduced. The operators,

$$\mathbf{T} = \prod_{i=1}^{n} \mathbf{W}_i^{m_i} \quad \text{and} \tag{1.29}$$

$$\mathbf{R}_j = \begin{cases} \mathbf{W}_j^{m_j - 1} \prod_{i=1,\, i \neq j}^{n} \mathbf{W}_i^{m_i} & \text{if } m_j \geq 1 \\ \mathbf{0} \in \mathbb{R}^{s \times s} & \text{otherwise} \end{cases} \tag{1.30}$$

are introduced to obtain a more compact notation. Then, the derivative of the ANN evaluated at the samples in $\mathcal{T}_{\text{STM}}$ is given by,

$$\frac{\partial^{m_1}}{\partial y_1^{m_1}} \cdots \frac{\partial^{m_j}}{\partial y_j^{m_j}} \cdots \frac{\partial^{m_n}}{\partial y_n^{m_n}} \left[\hat{u}(\mathbf{y}) \right] |_{\mathbf{y} \in \mathcal{T}_{\text{STM}}} = \mathbf{S}^{\lambda}\, \mathbf{T}\, \mathbf{v}^T \tag{1.31}$$

where $\mathbf{S}^{\lambda}$ denotes the λ^{th} derivative of the σ function with respect to its scalar argument evaluated at the input-to-node matrix (1.27), and from hereon will be referred to as *transfer function matrix* of the λ^{th} order, where $\lambda = \sum_{i=1}^{n} m_i$, and $d^0\sigma/dr^0 = \sigma$. r diagonal matrices $\mathbf{W}_j \in \mathbb{R}^{s \times s}$, with $j = 1, 2, ..., r$ are defined such that the j^{th} component on the diagonal is given by the j^{th} input weight of the ANN.

We are now ready to compute the Jacobian of the error with respect to the ANN adjustable parameters required in order to train the ANN by backpropagation. Making use of equations (1.29)-(1.30), the Jacobian can be written as,

$$\mathbf{J} = [\, \mathbf{J}_{\mathbf{W}^1} \mid \cdots \mid \mathbf{J}_{\mathbf{W}^r} \mid \mathbf{J}_{\mathbf{d}} \mid \mathbf{J}_{\mathbf{v}}] \tag{1.32}$$

where,

$$\mathbf{J}_{\mathbf{W}^i} = (m_i\, \mathbf{S}^{\lambda}\mathbf{R}_i + \mathbf{Y}_i\mathbf{S}^{\lambda+1}\mathbf{T})\, \mathbf{V}\; i = 1, ..., r \tag{1.33}$$

$$\mathbf{J}_{\mathbf{d}} = \mathbf{S}^{\lambda+1}\mathbf{T}\, \mathbf{V} \tag{1.34}$$

$$\mathbf{J}_{\mathbf{v}} = \mathbf{S}^{\lambda}\mathbf{T} \tag{1.35}$$

and $\mathbf{Y}_i \in \mathbb{R}^{q\times q}, i = 1, 2, ..., r$ are diagonal matrices, defined as: $[\mathbf{Y}_i]_l^k = y_i^k \delta_{kl}$, where δ_{kl} is the Kronecker delta and y_i^k are the i^{th} components of the k^{th} samples' inputs, with no implied summation over indices. Similarly $\mathbf{V} \in \mathbb{R}^{s\times s}$ refers to a diagonal matrix assembled with the components of the $\mathbf{v}$ vector, or in components $[\mathbf{V}]_{mn} = v_m \delta_{mn}\; m = 1, ..., s$, and v_m is the m^{th} component of the output weights. Introducing the vector $\mathbf{f} := f(\mathbf{y})|_{\mathbf{y}\in\mathcal{T}_{\text{STM}}} \in \mathbb{R}^q$, the function to be minimized may be defined as,

$$e(\mathbf{w}) = \frac{1}{2}\boldsymbol{\epsilon}^T\boldsymbol{\epsilon} \tag{1.36}$$

where

$$\boldsymbol{\epsilon} := \mathcal{D}^k\left[\hat{u}(\mathbf{y})\right]|_{\mathbf{y}\in\mathcal{T}_{\text{STM}}} - \mathbf{f} \tag{1.37}$$

The above methodology can be extended to *nonlinear* PDEs by adopting a useful property of the Hadamard product. The Hadamard product of two matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{M\times r}$, also known as entry-wise product, is defined as,

$$(\mathbf{A}\circ\mathbf{B})_{ij} = a_{ij}b_{ij} \tag{1.38}$$

As shown in [3], the Hadamard product obeys the following property,

$$\frac{\partial\,(\mathbf{A}\circ\mathbf{B})}{\partial\alpha} = \frac{\partial\mathbf{A}}{\partial\alpha}\circ\mathbf{B} + \mathbf{A}\circ\frac{\partial\mathbf{B}}{\partial\alpha} \tag{1.39}$$

where, α is a scalar parameter, and $\mathbf{A}$ and $\mathbf{B}$ are matrices or matrix functions. In order to derive the Jacobian for the nonlinear part, we extend the property in (1.39) to the case of differentiation with respect to vectors. After some algebraic manipulations it may be shown that the following expression holds,

$$\frac{\partial\,(\mathbf{A}\circ\mathbf{B})}{\partial\mathbf{a}} = \frac{\partial\mathbf{A}}{\partial\mathbf{a}}\circ(\mathbf{B}\otimes\boldsymbol{\gamma}) + \frac{\partial\mathbf{B}}{\partial\mathbf{a}}\circ(\mathbf{A}\otimes\boldsymbol{\gamma}) \tag{1.40}$$

where in (1.40) $\mathbf{a} \in \mathbb{R}^l$ and $\boldsymbol{\gamma} \in \mathbb{R}^l$ are row-vectors, with $\boldsymbol{\gamma}$ defined as $\boldsymbol{\gamma} = [1\; 1 \cdots\; 1]$. The symbol $\otimes$ denotes the Kronecker product between tensors, and (1.40) may then be used to compute the adjoined Jacobian for a nonlinear PDE.

1.4.2 Example: PDE Solution on a Unit Circle

This example illustrates how an ANN trained via CPROP is capable to adapt and approximate a changing PDE solution in a non-stationary environment. The nonlinear PDE is forced by a known term $f_j(y_1, y_2)$ that is subject to change, and must be solved

over a unit circle centered at the origin,

$$\nabla^2 u + u\,\frac{\partial u}{\partial y_2} = f_j(y_1, y_2), \quad j = 1, 2, 3. \tag{1.41}$$

Where, three forcing terms, index by $j = 1, 2, 3$ are considered in this example. The above PDE solution must satisfy the boundary conditions (BCs),

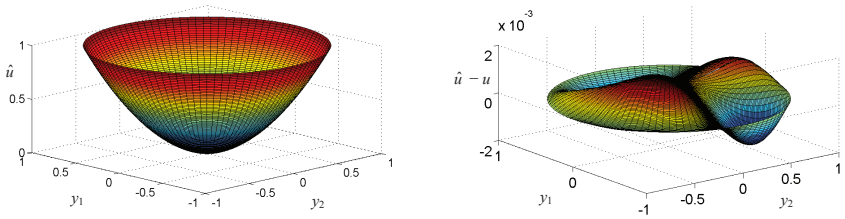
$$u(y_1, y_2) = 1 \tag{1.42}$$

on the unit circle defined as $\mathcal{I} := \{y_1, y_2 \,|\, y_1^2 + y_2^2 = 1\}$. Where, in (1.41), the solution $u(y_1, y_2)$ is denoted by u for brevity, and the functions $f_j(y_1, y_2)$, $j = 1, 2, 3$ are assumed to be continuous and known. Then, by defining a two-dimensional grid over $\mathcal{I}$, the STM training set $\mathcal{T}_{S_j} = \{\mathbf{y}_k \in \mathcal{I}, k = 1, \ldots, q\}$ is obtained for every j , and the values of the present (j) forcing function evaluated, and organized in the $q \times 1$ vector $\mathbf{f}_j := [f_j(\mathbf{y}_1) \cdots f_j(\mathbf{y}_q)]^T$. The STM training sets are used to adapt the ANN solution $\hat{u}(\mathbf{y})$ incrementally, by presenting each one as the forcing function changes from f_1 to f_2, and then from f_2 to f_3. These functions are chosen such that the above PDE has the following analytical solutions: $u = y_1^2 + y_2^2$ when $j = 1$, $u = 1.15(y_1^2 + y_2^2) - 0.15$ when $j = 2$, and $u = \exp[-(y_1^2 + y_2^2)] + 1.1(y_1^2 + y_2^2) - e^{-1} - 0.1$ when $j = 3$.

For the numerical solutions,a 100-points grid was used, with points chosen from 10 concentric circles, equidistant from the center, by dividing each circle into 10 equal sectors. In order to avoid a homogeneous radial distribution of points, which may lead to singularities, a positive, counter-clockwise $\theta = \pi/8$-swirl was imposed between circles during sampling. The ANN was trained with $q = 7$ STM samples, and contained $s_L = 25$ LTM nodes to be able to match the BCs on the unit circle (1.42) at all times. For every forcing term f_j, CPROP training is conducted for approximately 50 epochs, after which it is assumed that the environmental conditions have changed, and the ANN-PDE solution is adapted to the new term f_{j+1}. The final training has been run until satisfactory convergence has occurred and no further improvement seemed possible (after 450 epochs). For validation, the ANN solution is tested and plotted over a much denser grid obtained using 90 circles subdivided into twenty sectors. The results plotted in Figs. 1.3 - 1.5 show that the ANN output approximates every one of the three solutions with good accuracy, excellent generalization, and without any overfitting.

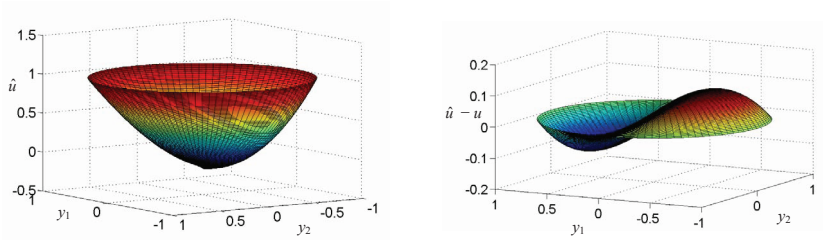
1.5 PRESERVING PRIOR KNOWLEDGE IN EXPLORATORY ADAPTIVE CRITIC DESIGNS

The advantages brought about by using classical control theory in conjunction with nonlinear NNs have long been recognized in the literature [6–10,22,23,27–29,31,34]. In particular, using classical controllers to obtain the starting neural control design has been shown to be a key step in the development of highly-effective adaptive neural controllers. One reason is that the starting design provides adequate performance while the adaptation compensates for nonlinearities and unmodeled dynamics.



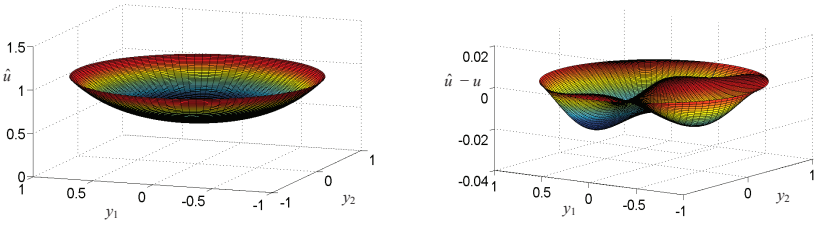
(a) (b)

Figure 1.3 Neural Network solution (a) and error surface (b) when $j = 1$.



(a) (b)

Figure 1.4 Neural Network solution (a) and error surface (b) when $j = 2$.



(a) (b)

Figure 1.5 Neural Network solution (a) and error surface (b) when $j = 3$.

Another reason is that many popular adaptation schemes, such as, adaptive critics, improve iteratively upon the existing approximation of the control law. Therefore, the classical controller provides a performance baseline for the adaptive neural controller. Typically, a classical controller is used to pre-train the neural controller in a supervised fashion, and the adaptation is carried out by reinforcement learning (RL), since the ideal control law for nonlinear and unmodeled dynamics is unknown. However, due to interference, the neural controller may rapidly forget the performance baseline provided by the classical controller during adaptation.

A popular approach for combining these two styles of learning, recently reviewed in [33], consists of computing the control as a weighted sum of a supervisor's control and an exploratory policy. While the supervisor provides the nominal control signal to the system to be controlled or plant, the exploratory policy or actor is updated by linearly interpolating between the RL weight update and a supervised learning (SL) weight update. Although this approach may not suppress interference, the supervisor overrides bad control signals from the actor and, in this fashion, guarantees a minimum performance baseline [33]. The main drawback of this approach is that the linear superposition of control policies may not lead to any performance improvement when the plant exhibits highly nonlinear dynamics. Similarly, computing the linear interpolation between the RL and SL weight updates may prove ineffective for learning highly nonlinear (and non-convex) control laws.

1.5.1 Derivation of LTM Constraints for Feedback Control

Consider a plant whose dynamics can be approximated by the nonlinear differential equation and output equation,

$$\dot{\mathbf{x}}(t) = \mathbf{f}[\mathbf{x}(t), \mathbf{p}_m(t), \mathbf{u}(t)] \tag{1.43}$$

$$\mathbf{y}(t) = \mathbf{h}[\mathbf{x}(t), \mathbf{u}(t)] \tag{1.44}$$

where $\mathbf{x} \in \mathcal{X} \subset \mathbb{R}^{n\times 1}$ is the state, and $\mathbf{u} \in \mathcal{U} \subset \mathbb{R}^{m\times 1}$ is the control. The differential equation structure and parameters, $\mathbf{p}_m$, are not always known *a priori*, and are subject to change during the lifetime of the plant. It is assumed that perfect knowledge of the state $\mathbf{x}$ is available at a present time t, based on error-free measurements of the output $\mathbf{y}$. Under restricted operating conditions the plant operates in a subset of the state space $\mathcal{X}_{LPV} \subset \mathcal{X}$, referred to as linear-parameter-varying or LPV regime, in which plant dynamics can be closely approximated by a linear-time-invariant (LTI) model near every equilibrium or operating point. Then, a finite set of p equilibria in $\mathcal{X}_{LPV}$ can be selected, and indexed by a corresponding *scheduling vector* $\boldsymbol{\zeta} \in \mathbf{R}^{z\times 1}$, which contains time-varying physical parameters that significantly influence plant dynamics. Let $\mathcal{Z} = \{\boldsymbol{\zeta}_1, \ldots, \boldsymbol{\zeta}_p\} \in \mathcal{X}_{LPV}$ denote a finite set of scheduling vectors, such that at every $\boldsymbol{\zeta}_j \in \mathcal{X}_{LPV}$ the plant can be represented by the transfer function,

$$\mathbf{H}_j(s) : \left\{ \begin{array}{rcl} \Delta\dot{\mathbf{x}} & = & \mathbf{A}_j\Delta\mathbf{x} + \mathbf{B}_j\Delta\mathbf{u} \\ \Delta\mathbf{y} & = & \mathbf{C}_j\Delta\mathbf{x} + \mathbf{D}_j\Delta\mathbf{u} \end{array} \right. . \tag{1.45}$$

Then, the set of transfer functions $\mathcal{H} = \{\mathbf{H}_1(s), \ldots, \mathbf{H}_p(s)\}$ can be used to design a gain-scheduled controller that interpolates between the LTI controllers derived from each transfer function in $\mathcal{H}$.

Several gain-scheduling techniques, including multivariable control, μ-synthesis, and H_∞ have been developed to design state-feedback linear control systems that interpolate among p transfer functions of the form,

$$K_j(s) : \left\{ \begin{array}{l} \dot{\mathbf{x}}_K = \mathbf{A}_{K_j}\mathbf{x}_K + \mathbf{B}_{K_j}\Delta\mathbf{x} \\ \mathbf{u}_K := \mathbf{C}_{K_j}\mathbf{x}_K + \mathbf{D}_{K_j}\Delta\mathbf{x} \end{array} \right. , \tag{1.46}$$

to control the plant in the LPV regime. Where, $\mathbf{x}_K \in \mathbf{R}^{\kappa\times 1}$ is the controller state. Several techniques, including convex interpolation, can be utilized to obtain a gain-scheduled controller from the set of LTI controllers $\mathcal{K} = \{K_1, \ldots, K_p\}$. In this chapter, the gain-scheduled controller is obtained by means of a recurrent neural network,

$$ANN : \left\{ \begin{array}{l} \dot{\mathbf{x}}_K = \mathbf{A}_K(\boldsymbol{\zeta})\mathbf{x}_K + \mathbf{B}_K(\boldsymbol{\zeta})\Delta\mathbf{x} \\ \mathbf{u}_N := \mathbf{v}\ \boldsymbol{\Phi}(\mathbf{W}\mathbf{x}_a) \end{array} \right. , \tag{1.47}$$

with input $\mathbf{x}_a := [\boldsymbol{\chi}^T \quad \boldsymbol{\zeta}^T]^T$. Where, $\boldsymbol{\chi} := [\Delta\mathbf{x}^T \quad \mathbf{x}_K^T]^T \in \mathbf{R}^{\nu\times 1}$, and it can be seen from (1.45) that $\mathbf{x}_K$ is a function of $\mathbf{u}_N$ because, once the controller is implemented, $\Delta\mathbf{u} = \mathbf{u}_N$. Hereon, the control input is assumed to be scalar to simplify the presentation.

Then, using algebraic training [6], it can be shown that given a set of LTI controllers $\mathcal{K} = \{K_1, \ldots, K_p\}$ there exists an ANN controller (1.47) with $l = p$ sigmoidal nonlinearities that is input-output equivalent to (1.46) at the equilibria $\{\boldsymbol{\zeta}_1, \ldots, \boldsymbol{\zeta}_p\} \in \mathcal{X}_{LPV}$, i.e., satisfies the *closed-loop* requirements,

$$u_N(\boldsymbol{\zeta}_j) = u_K(\boldsymbol{\zeta}_j), \quad \text{and} \quad \frac{\partial u_N}{\partial \dot{\boldsymbol{\chi}}}(\boldsymbol{\zeta}_j) = \frac{\partial u_K}{\partial \dot{\boldsymbol{\chi}}}(\boldsymbol{\zeta}_j), \quad \text{for} \quad j = 1, \ldots, p. \tag{1.48}$$

provided the ANN weights satisfy the algebraic equations,

$$\begin{array}{l} \mathbf{N} = \mathbf{W}_{\boldsymbol{\zeta}}\mathbf{Z} \\ \mathbf{S}\mathbf{v}^T = \mathbf{b} \\ \mathbf{D}\mathbf{V}\mathbf{W}_{\boldsymbol{\chi}} = \mathbf{M}_2 \end{array} \tag{1.49}$$

and provided the matrices $(\mathbf{I} - \mathbf{D}_j\mathbf{D}_{K_j})$ are invertible. The ANN inputs weights are partitioned into $\mathbf{W} = [\mathbf{W}_{\boldsymbol{\chi}} \quad \mathbf{W}_{\boldsymbol{\zeta}}]$ based on the corresponding inputs, $\mathbf{V} := \text{diag}(\mathbf{v})$, and $\mathbf{b} := \tilde{b}\ \boldsymbol{\imath}_p$. Where, $\boldsymbol{\imath}_p := 1_{p\times 1}$, $\tilde{b}$ is an arbitrary constant bias, and $\boldsymbol{\Phi}'(n) := [\sigma'(n_1)\cdots\sigma'(n_l)]^T$. The remaining matrices are defined in the Appendix. A proof is provided in [6], where the properties of the resulting ANN controller, including closed-loop stability, are also proven.

1.5.2 Constrained Adaptive Critic Design

The advantage of the ANN controller (1.47) over other gain-scheduled controllers is that it is nonlinear and adaptive. Therefore, an approximate dynamic programming (ADP) algorithm can be used to improve its performance online through incremental training. This may be necessary when the plant (1.43) operates outside of the LPV regime, i.e., $\mathbf{x} \in \{\mathcal{X}/\mathcal{X}_{LPV}\}$, and in the presence of parameter variations or unmodeled dynamics. In this case, the observed state $\hat{\mathbf{x}}$ differs from that estimated from the dynamic equation. Also, since the underlying assumptions are violated, the gain-scheduled controller's performance ceases to be optimal. However, if the weights of the ANN controller are updated via ADP using a conventional backpropagation algorithm, the algebraic equations in (1.49) may no longer be satisfied due to interference. Subsequently, when the plant returns to the LPV regime, the neural network controller will no longer meet the original LPV performance and stability guarantees. By viewing the algebraic equations (1.49) as the ANN LTM, it is possible to preserve the same guarantees of closed-loop performance and stability in $\mathcal{X}_{LPV}$, while the ANN controller is adapted incrementally online via ADP. In this case, the ADP policy-improvement routine and value-determination operation provide STMs to be learned incrementally over time, subject to the LTM constraints in (1.49).

The state-feedback control law is adapted by updating the weights of the recurrent neural network (1.47) through several iterations of the ADP algorithm that are indexed by k and take place over time. Then, letting the adaptive control law be defined as,

$$c_k[\mathbf{x}_a(t)] := \mathbf{v}^{[k]}\mathbf{\Phi}[\mathbf{W}^{[k]}\,\mathbf{x}_a(t)], \quad k = 0, 1, 2, \ldots \tag{1.50}$$

the weights can be determined by a constrained ADP approach that improves performance over time, while preserving the LTM. Where, $\mathbf{v}^{[k]}$ and $\mathbf{W}^{[k]}$ denote the values of $\mathbf{v}$ and $\mathbf{W}$ at the k^{th} iteration of the ADP algorithm. The same control objectives used to specify the LPV performance, such as, H_∞ and H_2 performance, and pole placement, are used to express the desired system performance in $\mathcal{X}$. When the plant dynamics are nonlinear, equivalent control objectives can be formulated by means of an integral cost function,

$$J = \lim_{t_f \to \infty} \left\{ \frac{1}{t_f} \int_{t_0}^{t_f} \sum_{i=1}^{q} \boldsymbol{\xi}_i^T(\tau)\boldsymbol{\xi}_i(\tau) d\tau \right\} \tag{1.51}$$

where each control objective defines a quadratic cost index $\boldsymbol{\xi}_i$, which is formulated in terms of the plant state and control vectors. For example, both H_2 performance and pole placement can be expressed by a cost index $\boldsymbol{\xi}_i(t) = \mathbf{M}_i^{1/2}[\mathbf{x}_a^T(t) \quad \mathbf{u}^T(t)]^T$, through a symmetric weighting matrix $\mathbf{M}_i$ of design parameters [36, Section 6.3]. Then, the value function,

$$V_k[\mathbf{x}_a(t), c_k] := \int_t^{t_f} \sum_{i=1}^{q} \left[\mathbf{x}_a^T(\tau)\; c_k^T[\mathbf{x}_a(\tau)]\right]\; \mathbf{M}_i^{[k]}\; \left[\mathbf{x}_a^T(\tau)\; c_k^T[\mathbf{x}_a(\tau)]\right]^T d\tau \tag{1.52}$$

can be updated through the ADP value-determination operation to learn the optimal cost-to-go for the actual system dynamics. Where $\mathbf{M}_i^{[k]}$ denotes the value of $\mathbf{M}_i$ at the k^{th} iteration of the ADP algorithm.

In the LPV regime, the values of the design matrices $\mathbf{M}_i$ are obtained using classical control techniques, such as implicit model following and dynamic compensation [36, Section 6.3], and (1.51) is optimized by the same values of $\mathbf{v}$ and $\mathbf{W}$ that satisfy (1.49). Therefore, at $k = 0$ all design parameters are set equal to these initial LPV values, obtained from (1.49), which hereon are denoted by $\mathbf{v}^{[0]}$, $\mathbf{W}^{[0]}$, and $\mathbf{M}_i^{[0]}$. Subsequently, if the plant leaves $\mathcal{X}_{LPV}$, or experiences parameter variations or unmodeled dynamics, the controller violates the optimality condition,

$$\nabla_{\mathbf{u}(t)}\mathcal{L}[\mathbf{x}(t),\mathbf{u}(t)] + \nabla_{\mathbf{u}(t)}V_k[\mathbf{f}[\mathbf{x}(t),\mathbf{p}_m(t),\mathbf{u}(t)],c_k]\} = \mathbf{0} \tag{1.53}$$

and sets off the ADP adaptation. The ADP adaptation optimizes the neural network control law (1.50) by cycling between the Policy-Improvement Routine and the Value-Determination Operation repeatedly over time, as illustrated in Fig. 1.6.

The unconstrained policy-improvement routine can be written as,

$$\begin{aligned} c_{k+1}[\mathbf{x}(t)] &= \arg\min_{\mathbf{u}(t)}\{\mathcal{L}[\mathbf{x}(t),\mathbf{u}(t)] + V_k[\mathbf{f}[\mathbf{x}(t),\mathbf{p}_m(t),\mathbf{u}(t)],c_k]\} &(1.54)\\ &=: \arg\min_{\mathbf{u}(t)}\{e_k(\mathbf{w})\}, \quad k = 0,1,2,\ldots. &(1.55)\end{aligned}$$

such that $V_k[\mathbf{x}(t),c_{k+1}] \leq V_k[\mathbf{x}(t),c_k]$, for $\forall\mathbf{x}(t) \in \mathcal{X}$. In order for the neural network controller to preserve its optimal performance in $\mathcal{X}_{LPV}$, a set of constraints obtained from (1.49) are adjoint leading to the following constrained optimization problem:

$$\begin{aligned} \text{minimize} \quad & E_k(\mathbf{w}) := \sum_{i=1}^{q}\hat{\boldsymbol{\xi}}_i^T(t)\hat{\boldsymbol{\xi}}_i(t) + V_k\left[\mathbf{f}\left[\hat{\mathbf{x}}(t),\mathbf{p}_m(t),c_k[\hat{\mathbf{x}}_a(t)]\right],c_k[\hat{\mathbf{x}}_a(t)]\right] \\ \text{subject to} \quad & \mathbf{g}(\mathbf{w}) := \begin{bmatrix} \mathbf{N} - \mathbf{W}_{\boldsymbol{\varsigma}}\mathbf{Z} \\ \mathbf{S}\mathbf{v}^T - \mathbf{b} \\ \mathbf{D}\mathbf{V}\mathbf{W}_{\boldsymbol{\chi}} - \mathbf{M}_2 \end{bmatrix} = \mathbf{0} \end{aligned} \tag{1.56}$$

Where, $\hat{\boldsymbol{\xi}}_i(t) := \mathbf{M}_i^{1/2}[\hat{\mathbf{x}}_a^T(t) \quad c_k^T[\hat{\mathbf{x}}_a(t)]]^T$ and $\hat{\mathbf{x}}_a(t)$ is the actual value of the augmented state observed at the present time t. The adjustable parameters $\mathbf{v}$ and $\mathbf{W}$ are rearranged into $\mathbf{w} \in \mathbf{R}^{N\times 1}$, which is the variable of the above constrained minimization problem. Where, $\mathbf{Z}$, $\mathbf{b}$, $\mathbf{M}_2$, and $\mathbf{R}_2$ are known constants defined in Section 1.5.1 the Appendix. The matrices $\mathbf{N}$, $\mathbf{S}$, $\mathbf{D}$, and $\mathbf{V}$ are functions of $\mathbf{w}$, as defined in Section 1.5.1. Finally, the parameters $\mathbf{M}_i$, $i = 1,\ldots,q$, in (1.52) are updated according to the value-determination operation,

$$V_{k+1}[\mathbf{x}(t),c_{k+1}] = \mathcal{L}[\mathbf{x}(t),\mathbf{u}(t)] + V_k[\mathbf{f}[\mathbf{x}(t),\mathbf{p}_m(t),\mathbf{u}(t)],c_{k+1}] \tag{1.57}$$

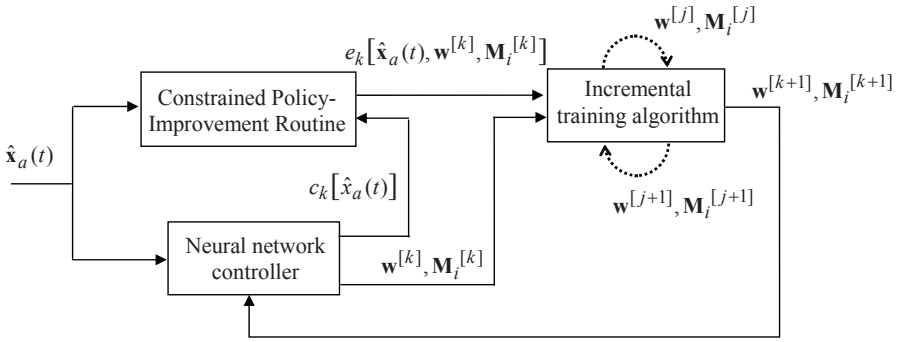


Figure 1.6 Algorithmic diagram of constrained Policy-Improvement Routine applied to the neural network controller (1.50).

Since in this case the equality constraint (1.56) may not satisfy the implicit function theorem, the method of Lagrange multipliers can be used to seek the solution of (1.56)-(1.56). In this method, reviewed in [36, pp. 36-41], the equality constraint (1.56) is adjoined to the function to be minimized (1.56) by defining an augmented error function

$$E_{a_k}(\mathbf{w}) \equiv e_k(\mathbf{w}) - \boldsymbol{\lambda}^T \, \mathbf{g}(\mathbf{w}) \tag{1.58}$$

The vector of Lagrange multipliers $\boldsymbol{\lambda}$ contains as many unknowns as there are equality constraints. As shown in [36, pp. 36-41], in the vicinity of an extremum of (1.58), $\boldsymbol{\lambda}$ takes the value,

$$\boldsymbol{\lambda}^* = -\left(\frac{\partial \mathbf{g}}{\partial \mathbf{w}}\right)^{-T} (\nabla_{\mathbf{w}} e_k)^T \tag{1.59}$$

where, $-T$ denotes the inverse transpose of a matrix. Then, the optimal values of $\mathbf{w}$ can be determined from,

$$\nabla_{\mathbf{w}} e_k - (\boldsymbol{\lambda}^*)^T \, \frac{\partial \mathbf{g}}{\partial \mathbf{w}} = \mathbf{0} \tag{1.60}$$

using a Newton-Raphson algorithm.

1.6 SUMMARY

Because of their ability to approximate nonlinear functions, and to generalize and learn from sampled data, adaptive ANNs have been able to solve a broad range of problems in the sciences and engineering, such as, differential equations, system identification, and control. But, as demonstrated by natural cognitive systems, it is only by learning tasks and information incrementally over time that intelligent systems can reach high levels of complexity and performance. The CPROP approach

described in this chapter provides a rigorous computational framework for procedural memory formation in incremental training. As shown through a few sample applications, CPROP allows ANNs to solve adapt problem solutions repeatedly, through multiple training sessions, while retaining a desired baseline performance at all times. CPROP offers a unified view of memory formation and retention in ANNs that does not rely on presenting the same information to the ANN repeatedly over time. Thus, similarly to natural cognitive systems, CPROP ANNs are capable of retaining a wide variety of long-term memories, including memory of input-output patterns (function approximation), knowledge of a system's behavior (system ID), boundary conditions (differential equations), and optimal motor and control skills, while learning new skills or short-term memories through new training sessions.

REFERENCES

1. *MATLAB® Neural Network Toolbox, User's Guide*. The MathWorks, 2005.
2. P. Aarts and P. V. D. Veer. Neural network method for solving partial differential equations. *Neural Processing Letters*, 14(3), 2001.
3. D. S. Bernstein. *Matrix Mathematics: Theory, Facts, and Formulas*. Princeton University Press, Princeton, NJ, 2009.
4. G. A. Carpenter. Neural network models for pattern recognition and associative memory. *Neural Networks*, 2(4), 1989.
5. Gianluca DiMuro and Silvia Ferrari. A constrained-optimization approach to training neural networks for smooth function approximation and system identification. In *Proceedings of the IEEE World Congress on Computational Intelligence*, pages 2354–2360, Hong Kong, China, June 2008.
6. S. Ferrari. Multiobjective algebraic synthesis of neural control systems by implicit model following. *IEEE Transactions on Neural Networks*, 20(3):406–419, 2009.
7. S. Ferrari and M. Jensenius. A constrained optimization approach to preserving prior knowledge during incremental training. *IEEE Transactions On Neural Networks*, 19(6), 2008.
8. S. Ferrari and R.F. Stengel. Classical/neural synthesis of nonlinear control systems. *Journal of Guidance, Control, and Dynamics*, 25(3):442–448, 2002.
9. S. Ferrari and R.F. Stengel. Model-based adaptive critic designs. In J. Si, A. Barto, and W. Powell, editors, *Learning and Approximate Dynamic Programming*. John Wiley and Sons, 2004.
10. S. Ferrari and R.F. Stengel. On-line adaptive critic flight control. *Journal of Guidance, Control, and Dynamics*, 27(5):777–786, 2004.
11. S. Ferrari and R.F. Stengel. Smooth function approximation using neural networks. *IEEE Trans. On Neural Networks*, 16(1):24–38, 2005.
12. R. M. French. Catastrophic forgetting in connectionist networks. *Trends in Cognitive Sciences*, 3(4), 1999.
13. J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proc. of the National Academy of Sciences USA*, 79, 1982.

14. L. Jianyu, L. Siwei, Q. Yingjian, and H. Yaping. Numerical solution of elliptic partial differential equation using radial basis function neural networks. *Neural Networks*, 16(5-6), 2003.

15. I. Kanter and H. Sompolinsky. Associative recall of memory without errors. *Physical Review A*, 35(1), 1987.

16. M. Kobayashi, A. Zamani, S. Ozawa, and S. Abe. Reducing computations in incremental learning for feedforward neural network with longterm memory. In *Proc. Int. Joint Conf. Neural Networks*. 1989.

17. T. Kohonen. *Self-Organization and Associative Memory*. Springer-Verlag, Berlin, Germany, 1987.

18. M. Kotani, K. Akazawa, S. Ozawa, and H. Matsumoto. Detection of leakage sound by using modular neural networks. In *Proc. Sixteenth Congress of the Int. Measurement Confederation*. 2000.

19. I.E. Lagaris, A. Likas, and D. I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Trans. On Neural Networks*, 9(5):987–1000, 1998.

20. Y. LeCun. A theoretical framework for backpropagation. In D. Touretzky, G. Hinton, and T. Sejnowski, editors, *Proceedings of the 1988 Connectionist Models Summer School*, pages 21–28, CMU, Pittsburgh, Pa, 1988. Morgan Kaufmann.

21. K. Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly Journal of Applied Mathmatics*, II(2):164–168, 1944.

22. F.-L. Lewis, S. Jagannathan, and A. Yesilderek. *Neural Network Control of Robot Manipulators and Nonlinear Systems*. Taylor and Francis, London, England, 1999.

23. S. Limanons and J. Si. Neural-network-based control design: An LMI approach. *IEEE Transactions on Neural Networks*, 9(6), 1998.

24. J. Mándziuk and L. Shastri. Incremental class learning - an approach to longlife and scalable learning. In *Proc. Int. Joint Conf. Neural Networks*. 1999.

25. D. W. Marquardt. An algorithm for least-squares estimation of non-linear parameters. *Journal of the Society of Industrial and Applied Mathematics*, 11(2):431–441, 1963.

26. M. McCloskey and N. Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. *The Psychology of Learning and Motivation*, 24(109-164), 1989.

27. K. S. Narendra and K. Parthasaranthy. Identification and control of dynamical systems using neural networks. *IEEE Transactions on Neural Networks*, 1:4–27, 1990.

28. J. Neidhoefer and K. Krishnakumar. Nonlinear control using neural approximators with linear control theory. In *Proc. of the AIAA Guidance, Navigation and Control Conference*, New Orleans, LA, 1997.

29. Y.-M. Park, M.-S. Choi, and K.Y. Lee. An optimal tracking neuro-controller for nonlinear dynamic systems. *IEEE Transactions on Neural Networks*, 7(5):1099–1110, 1996.

30. R. Polikar, L. Udpa, S. S. Udpa, and V. Honavar. Learn++: An incremental learning algorithm for supervised neural networks. *IEEE Transactions on Systems, Man, and Cybernetics - Part C*, 31(4), 2001.

31. M. M. Polycarpou. Stable adaptive neural control scheme for nonlinear systems. *IEEE Transactions on Automatic Control*, 14(3):4–27, 1996.

32. Ratcliff. Connectionist models of recognition memory: Constraints imposed by learning and forgetting functions. *Psychological Review*, pages 285–308, 1990.

33. M. T. Rosenstein and A. G. Barto. Supervised actor-critic reinforcement learning. In J. Si, A. Barto, and W. Powell, editors, *Learning and Approximate Dynamic Programming*. John Wiley and Sons, 2004.

34. R. Rysdyk and A. J. Calise. Robust nonlinear adaptive flight control for consistent handling qualities. *IEEE Transactions On Control Systems Technology*, 13(6):896–910, 2005.

35. Y. Shirvany, M. Hayati, and R. Moradian. Numerical solution of the nonlinear schrodinger equation by feedforward neural networks. *Communications in Nonlinear Science and Numerical Simulation*, 13(10), 2008.

36. R. F. Stengel. *Optimal Control and Estimation*. Dover Publications, Inc., 1986.

37. P. J. Werbos. *The roots of backpropagation: from ordered derivatives to neural networks and political forecasting*. Wiley-Interscience, New York, NY, USA, 1994.

38. H. Yamakawa, D . Masumoto, T. Kimoto, and S. Nagata. Active data selection and subsequent revision for sequential learning. Technical Report NC92/99, IEICE, 1993.

39. K. Yamauchi, N. Yamaguchi, and N. Ishii. Incremental learning methods with retrieving of interfered patterns. *IEEE Trans. Neural Networks*, 10(6), 1999.

40. A. Zhao. Global stability of bidirectional associative memory neural networks with distributed delays source. *Physics Letters*, 297(3-4), 2002.

Appendix: Algebraic ANN Control Matrices

$$\mathbf{M}_{1_j} := (I - D_j D_{K_j})^{-1} \begin{bmatrix} C_j & D_j C_{K_j} \end{bmatrix} \tag{A.1}$$

$$\mathbf{M}_{2_j} := \begin{bmatrix} 0 & C_{K_j} \end{bmatrix} + D_{K_j} \mathbf{M}_{1_j} \tag{A.2}$$

$$\mathbf{M}_2 := \left[\mathbf{M}_{2_1}^T \cdots \mathbf{M}_{2_p}^T \right]^T \tag{A.3}$$

$$Z := [\boldsymbol{\zeta}_1 \cdots \boldsymbol{\zeta}_p] \tag{A.4}$$

$$N := \left[n^1 \cdots n^p \right] \tag{A.5}$$

$$S := \left[\boldsymbol{\Phi}(n^1) \cdots \boldsymbol{\Phi}(n^p) \right]^T \tag{A.6}$$

$$D := \left[\boldsymbol{\Phi}'(n^1) \cdots \boldsymbol{\Phi}'(n^p) \right]^T \tag{A.7}$$