Neural Networks in Automotive Applications

Danil Prokhorov

Toyota Technical Center – a division of Toyota Motor Engineering and Manufacturing (TEMA), Ann Arbor, MI 48105, USA

Neural networks are making their ways into various commercial products across many industries. As in aerospace, in automotive industry they are not the main technology. Automotive engineers and researchers are certainly familiar with the buzzword, and some have even tried neural networks for their specific applications as models, virtual sensors, or controllers (see, e.g., [1] for a collection of relevant papers). In fact, a quick search reveals scores of recent papers on automotive applications of NN, fuzzy, evolutionary and other technologies of computational intelligence (CI); see, e.g., [2–4]. However, such technologies are mostly at the stage of research and not in the mainstream of product development yet. One of the reasons is "black-box" nature of neural networks. Other, perhaps more compelling reasons are business conservatism and existing/legacy applications (trying something new costs money and might be too risky) [5, 6].

NN technology which complements, rather than replace, the existing non-CI technology in applications will have better chances of wide acceptance (see, e.g., [8]). For example, NN is usually better at learning from data, while systems based on first principles may be better at modeling underlying physics. NN technology can also have greater chances of acceptance if it either has no alternative solution, or any other alternative is much worse in terms of the cost-benefit analysis. A successful experience with CI technologies at the Dow Chemical Company described in [7] is noteworthy.

Ford Motor Company is one of the pioneers in automotive NN research and development [9, 10]. Relevant Ford papers are referenced below and throughout this volume.

Growing emphasis on model based development is expected to help pushing mature elements of the NN technology into the mainstream. For example, a very accurate hardware-in-the-loop (HIL) system is developed by Toyota to facilitate development of advanced control algorithms for its HEV platforms [11]. As discussed in this chapter, some NN architectures and their training methods make possible an effective development process on high fidelity simulators for subsequent on-board (in-vehicle) deployment. While NN can be used both on-board and outside the vehicle, e.g., in a vehicle manufacturing process, only on-board applications usually impose stringent constraints on the NN system, especially in terms of available computational resources.

Here we provide a brief overview of NN technology suitable for automotive applications and discuss a selection of NN training methods. Other surveys are also available, targeting broader application base and other non-NN methods in general; see, e.g., [12].

Three main roles of neural network in automotive applications are distinguished and discussed: models (Sect. 1), virtual sensors (Sect. 2) and controllers (Sect. 3). Training of NN is discussed in Sect. 4, followed by a simple example illustrating importance of recurrent NN (Sect. 5). The issue of verification and validation is then briefly discussed in Sect. 6, concluding this chapter.

1 Models

Arguably the most popular way of using neural networks is shown in Fig. 1. NN receives inputs and produces outputs which are compared with target values of the outputs from the system/process to be modeled or identified. This arrangement is known as supervised training because the targets for NN training are always

D. Prokhorov: *Neural Networks in Automotive Applications*, Studies in Computational Intelligence (SCI) **132**, 101–123 (2008)
www.springerlink.com

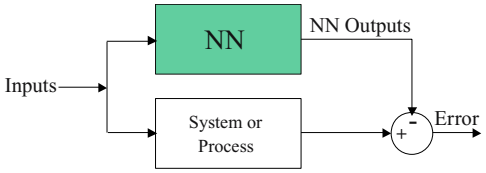

Fig. 1. A very popular arrangement for training NN to model another system or process including decision making is termed supervised training. The inputs to the NN and the system are not necessarily identical. The error between the NN outputs and the corresponding outputs of the system may be used to train the NN

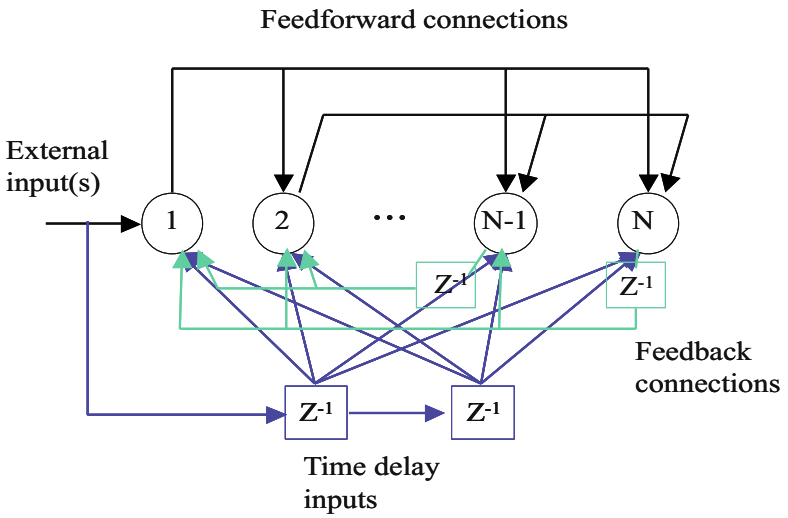

Fig. 2. Selected nodes in this network may be declared outputs. Any connectivity pattern, e.g., a popular layered architecture such as in multilayer perceptron (MLP), can be created by specifying the NN connectivity table. The order in which the nodes "fire," or get activated, also needs to be specified to preserve causality. Furthermore, explicit delays longer than one time step can also be included

provided by the system ("supervisor") to be modeled by NN. Figure 1 pertains to not only supervised modeling but also decision making, e.g., when it is required to train a NN classifier.

A general architecture of discrete-time NN is shown in Fig. 2. The neurons or nodes of the NN are labeled as 1 through N. The links or connections may have adjustable or fixed parameters, or NN weights. Some nodes in the NN serve as inputs of signals external to the NN, others serve as outputs from the NN to the external world. Each node can sum or multiply all the links feeding it. Then the node transforms the result through any of a variety of functions such as soft (sigmoidal) and hard thresholds, linear, quadratic, or trigonometric functions, Gaussians, etc.

The blocks Z^{-1} indicates one time step delay for the NN signals. A NN without delays is called feedforward NN. If the NN has delays but no feedback connections, it is called time delay NN. A NN with feedback is called recurrent NN (RNN).

A large variety of NN exists. The reader is referred to [13] for a comprehensive discussion about many of the NN architectures and their training algorithms.

Clearly, many problem specific issues must be addressed to achieve successful NN training. They include pre- and (sometimes) post-processing of the data, the use of training data sufficiently representative of the system to be modeled, architectural choices, the optimal accuracy achievable with the given NN architecture, etc.

For a NN model predicting next values of its inputs it is useful to verify whether iterative predictions of the model are meaningful. A model trained to predict its input for the next time step might have a very large error predicting the input two steps into the future. This is usually the sign of overfitting. A single-step prediction might be too simple a task, especially for a slowly changing time series. The model might quickly learn that predicting the next value to be the same as its current value is good enough; the iterative prediction test should quickly reveal this problem with the model.

The trick above is just one of many useful tricks in the area of NN technology. The reader is referred to [14] and others for more information [13, 15].

Automotive engine calibration is a good example for relatively simple application of NN models. Traditionally, look-up tables have been used within the engine control system. For instance, a table linking engine torque production (output) with engine controls (inputs), such as spark angle (advance or retard), intake/exhaust valve timing, etc. Usually the table is created by running many experiments with the engine on a test stand. In experiments the space of engine controls is explored (in some fashion), and steady state engine torque values are recorded. Clearly, the higher the dimensionality of the look-up table, and the finer the required resolution, the more time it takes to complete the look-up table.

The least efficient way is full factorial experimental design (see, e.g., [16]), where the number of necessary measurement increases exponentially with the number of the table inputs. A modern alternative is to use model-based optimization with design of experiment [17–20]. This methodology uses optimal experimental design plans (e.g., D- or V-optimal) to measure only a few predetermined points. A model is then fitted to the points, which enables the mapping interpolation in between the measurements. Such a model can then be used to optimize control strategy, often with significant computational savings (see, e.g., [21]).

In terms of models, a radial basis function (RBF) network [22, 23], a probabilistic NN which is implementationally simpler form of the RBF network [24], or MLP [25, 26] can all be used. So-called cluster weighted models (CWM) may be advantageous over RBF even in low-dimensional spaces [27–29]. CWM is capable of essentially perfect approximation of linear mappings because each cluster in CWM is paired with a linear output model. In contrast, RBF needs many more clusters to approximate even linear mappings to high accuracy (see Fig. 3).

More complex illustrations of NN models are available (see, e.g., [30, 31]). For example, [32] discusses how to use a NN model for an HEV battery diagnostics. The NN is trained to predict an HEV battery state-of-charge (SOC) for a healthy battery. The NN monitors the SOC evolution and signals about abnormally rapid discharges of the battery if the NN predictions deviate significantly from the observed SOC dynamics.

2 Virtual Sensors

A modern automobile has a large number of electronic and mechanical devices. Some of them are actuators (e.g., brakes), while others are sensors (e.g., speed gauge). For example, transmission oil temperature is measured using a dedicated sensor. A virtual or soft sensor for oil temperature would use existing signals from other available sensors (e.g., air temperature, transmission gear, engine speed) and an appropriate model to create a virtual signal, an estimate of oil temperature in the transmission. Accuracy of this virtual signal will naturally depend on both accuracy of the model parameters and accuracies of existing signals feeding the model. In addition, existing signals will need to be chosen with care, as the presence of irrelevant signals may complicate the virtual sensor design. The modern term “virtual sensor” appears to be used sometimes interchangeably with its older counterpart “observer.” Virtual sensor assumes less knowledge of the physical process, whereas observer assumes more of such knowledge. In other words, the observer model is often based on physical principles, and it is more transparent than that of virtual sensor. Virtual sensors are

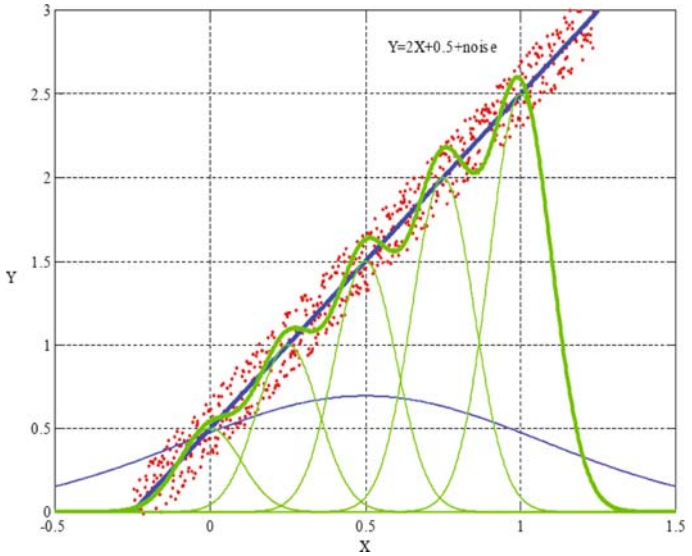

Fig. 3. A simple linear mapping $Y = 2X + 0.5$ subjected to a uniform noise (*red*) is to be approximated by CWM and RBF network. CMW needs only one cluster in the input space X and the associated mean-value linear model (the cluster and its model are shown in *blue*). In contrast, even with five clusters (*green*) the RBF network approximation (*thick green line*) is still not as good as the CWM approximation

often "black boxes" such as neural networks, and they are especially valuable when the underlying physics is too complex or uncertain while there is plenty of data to develop/train a virtual sensor.

In the automotive industry, significant resources are devoted to the development of continuous monitoring of on-board systems and components affecting tailpipe emissions of vehicles. Ideally, on-board sensors specialized to measuring the regulated constituents of the exhaust gases in the tailpipe (mainly hydrocarbons, nitrogen oxides (NO_x) and carbon monoxide) would check whether the vehicle is in compliance with government laws on pollution control. Given that such sensors are either unavailable or impractical, the on-board diagnostic system must rely on limited observations of system behavior and inferences based on those observations to determine whether the vehicle emissions are in compliance with the law. Our example is the diagnostics of engine combustion failures, known as misfire detection. This task must be performed with very high accuracy and under virtually all operating conditions (often the error rate of far less than 1% is necessary). Furthermore, the task requires the identification of the misfiring cylinder(s) of the engine quickly (on the order of seconds) to prevent any significant deterioration of the emission control system (catalytic converter). False alarm immunity becomes an important concern since on the order of one billion events must be monitored in the vehicle's lifetime.

The signals available to analyze combustion behavior are derived primarily from crankshaft position sensors. The typical position sensor is an encoder (toothed) wheel placed on the crankshaft of the engine prior to torque converter, transmission or any other engine load component. This wheel, together with its electronic equipment, provides a stream of accurately measured time intervals, with each interval being the time it takes for the wheel to rotate by one tooth. One can infer speed or acceleration of the crankshaft rotation by performing simple numerical manipulations with the time intervals. Each normal combustion event produces a slight acceleration of the crankshaft, whereas misfires exhibit acceleration deficits following a power stroke with little or no useful work (Fig. 4).

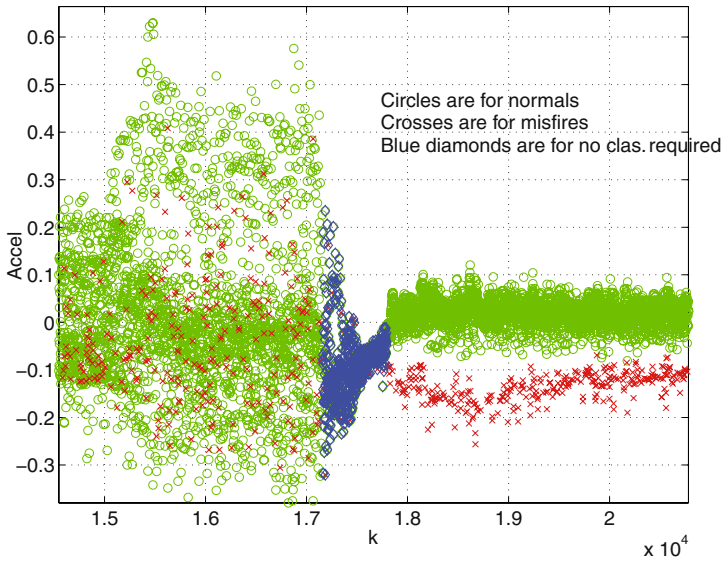

Fig. 4. A representative segment of the engine misfire data. Each cylinder firing is either normal (*circle*) or abnormal (misfire; *cross*). The region where misfire detection is not required is shown in the middle by *diamonds*. In terms of the crankshaft acceleration (y axis), sometimes misfires are easily separable from normal (the region between $k = 18{,}000$ and $k \sim 20{,}000$), and other times they are not (the region around $k = 16{,}000$)

The accurate detection of misfires is complicated by two main factors:

1. The engine exhibits normal accelerations and decelerations, in response to the driver input and from changing road conditions.
2. The crankshaft is a torsional oscillator with finite stiffness. Thus, the crankshaft is subject to torsional oscillations which may turn the signature of normal events into that of misfires, and vice versa.

Analyzing complex time series of acceleration patterns requires a powerful signal processing algorithm to infer the quality of the combustion events. It turns out that the best virtual sensor of misfires can be developed on the basis of a recurrent neural network (RNN) [33]; see Sect. 4 as well as [34] and [35] for training method details and examples. The RNN is trained on a very large data set (on the order of million of events) consisting of many recordings of driving sessions. It uses engine context variables (such as crankshaft speed and engine load) and crankshaft acceleration as its inputs, and it produces estimates of the binary signal (normal or misfire) for each combustion event. During each engine cycle, the network is run as many times as the number of cylinders in the engine. The reader is referred to [36] for illustrative misfire data sets used in a competition organized at the International Joint Conference on Neural Networks (IJCNN) in 2001. The misfire detection NN is currently in production.

The underlying principle of misfire detection (dependence of crankshaft torsional vibrations on engine operation modes) is also useful for other virtual sensing opportunities, e.g., engine torque estimation.

Concluding this section, we list a few representative applications of NN in the virtual sensor category:

- NN can be trained to estimate emissions from engines based on a number of easily measured engine variables, such as load, RPM, etc., [37–39], or in-cylinder pressure [40] (note that using a structure identification by genetic algorithms for NO_x estimation can result in performance better than that of a NN estimator; see [41] for details).

- Air flow rate estimating NN is described in [19], and air–fuel ratio (AFR) estimation with NN is developed in [42], as well as in [43] and [44].
- A special processor Vindax is developed by Axeon, Ltd. (http://www.axeon.com/), to support a variety of virtual sensing applications [45], e.g., a mass airflow virtual sensor [46].

3 Controllers

NN as controllers have been known for years; see, e.g., [47–51]. We discuss only a few popular schemes in this section, referring the reader to a useful overview in [52], as well as [42] and [8], for additional information.

Regardless of the specific schemes employed to adapt or train NN controllers, there is a common issue of linkage between the cause and its effect. Aside of the causes which we mostly do not control such as disturbances applied to *plant* (an object or system to be controlled), the NN controller outputs or *actions* in reinforcement learning literature [53] also affect the plant and influence a quality or performance functional. This is illustrated in Fig. 5.

To enable NN training/adaptation, the linkage between NN actions and the quality functional can be achieved through a model of the plant (and a model of the quality functional if necessary), or without a model. Model-free adaptive control is implemented sometimes with the help of a reinforcement learning module called *critic* [54, 55]. Applications of reinforcement learning and approximate dynamic programming to automotive control have been attempted and not without success (see, e.g., [56] and [57]).

Figure 6 shows a popular scheme known as model reference adaptive control. The goal of adaptive controller which can be implemented as a NN is to make the plant behave as if it were the reference model which specifies the desired behavior for the plant. Often the plant is subject to various disturbances such as plant parameter drift, measurement and actuator noise, etc. (not shown in the figure).

Shown by dashed lines in Fig. 6 is the plant model which may also be implemented as a NN (see Sect. 1). The control system is called *indirect* if the plant model is included, otherwise it is called *direct* adaptive control system.

We consider a process of indirect training NN controllers by an iterative method. Our goal is to improve the (ideal) performance measure I through training weights $\mathbf{W}$ of the controller

$$I(\mathbf{W}(i)) = E_{\mathbf{x}_0 \in \mathbf{X}} \left\{ \sum_{t=0}^{\infty} U(\mathbf{W}(i), \mathbf{x}(t), \mathbf{e}(t)) \right\}, \tag{1}$$

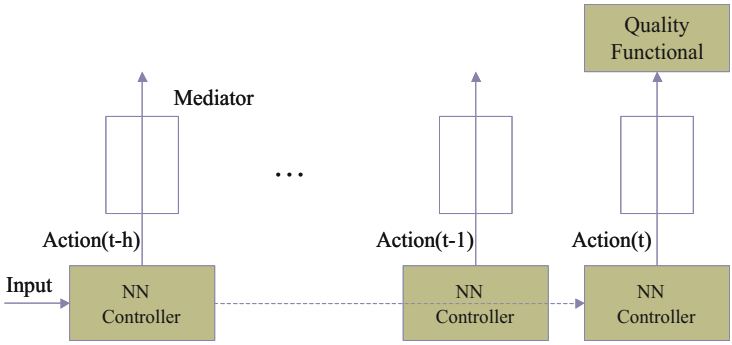

Fig. 5. The NN controller affects the quality functional through a mediator such as a plant. Older values of controls or actions (prior to t) may still have an effect on the plant as dynamic system with feedback, which necessitates the use of dynamic or temporal derivatives discussed in Sect. 4. For NN adaptation or training, the mediator may need to be complemented by its model and/or an adaptive critic

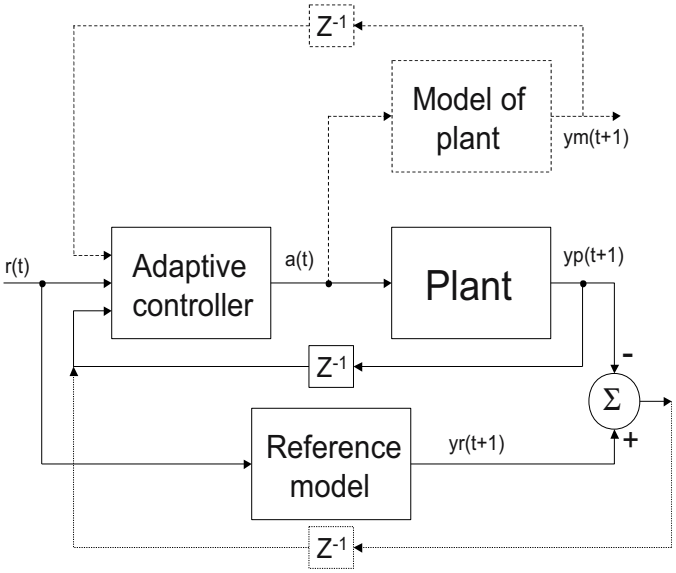

Fig. 6. The closed-loop system in model reference adaptive control. If the plant model is used (*dashed lines*), then the system is called indirect adaptive system; otherwise the system is called direct. The state vector $\mathbf{x}$ (not shown) includes state vectors of not only the plant but also the controller, the model of the plant and the reference model. The output vector $\mathbf{y}$ includes $\mathbf{a}$, $\mathbf{yp}$, $\mathbf{ym}$ and $\mathbf{yr}$. The error between $\mathbf{yp}$ and $\mathbf{yr}$ may be used as an input to the controller and for controller adaptation (*dotted lines*)

where $\mathbf{W}(i)$ is the controller weight vector at the ith training iteration (or epoch), $E_{\mathbf{X}}$ is a suitable expectation operator (e.g., average) in the domain of permissible initial state vectors $\mathbf{x}_0 \equiv \mathbf{x}(0)$, and $U(\cdot)$ is a non-negative definite function with second-order bounded derivatives often called the instantaneous utility (or cost) function. We assume that the goal is to increase $I(\mathbf{W}(i))$ with i. The state vector $\mathbf{x}$ evolves according to the closed-loop system

$$\mathbf{x}(t+1) = \mathbf{f}(\mathbf{x}(t), \mathbf{e}(t), \mathbf{W}(i)) + \boldsymbol{\varepsilon}(t) \tag{2}$$

$$\mathbf{y}(t) = \mathbf{h}(\mathbf{x}(t)) + \boldsymbol{\mu}(t), \tag{3}$$

where $\mathbf{e}(t)$ is a vector of external variables, e.g., reference signals $\mathbf{r}$, $\boldsymbol{\varepsilon}$ and $\boldsymbol{\mu}$ are noise vectors adding stochasticity to otherwise deterministic dynamic system, and $\mathbf{y}(t)$ is a vector of relevant outputs. Our closed-loop system includes not just the plant and its controller, which are usual components of the closed-loop system, but also the plant model and the reference model.

In reality, both E and ∞ in (1) must be approximated. Assuming that all initial states $\mathbf{x}_0(k)$ are equiprobable, the average operator can be approximated as

$$R(\mathbf{W}(i)) = \frac{1}{N} \sum_{\mathbf{x}_0(k) \in \mathbf{X}, k=1,2,\ldots,N} \sum_{t=0}^{T} U(i,t), \tag{4}$$

where N is the total number of trajectories of length T along which the closed-loop system performance is evaluated at the iteration i. The first evaluation trajectory begins at time $t = 0$ in $\mathbf{x}_0(1)$, i.e., $\mathbf{x}(0) = \mathbf{x}_0(1)$,

the second trajectory starts at $t = 0$ in $\mathbf{x}(0) = \mathbf{x}_0(2)$, etc. The coverage of the domain $\mathbf{X}$ should be as broad as practically possible for a reasonably accurate approximation of I.

Training the NN controller may impose computational constraints on our ability to compute (4) many times during our iterative training process. It may be necessary to contend with this approximation of R

$$A(\mathbf{W}(i)) = \frac{1}{S} \sum_{\mathbf{x}_0(s) \in \mathbf{X}, s=1,2,\ldots,S} \sum_{t=0}^{H} U(i,t). \tag{5}$$

The advantage of A over R is in faster computations of derivatives of A with respect to $\mathbf{W}(i)$ because the number of training trajectories per iteration is $S \ll N$, and the trajectory length is $H \ll T$. However, A must still be an adequate replacement of R and, possibly, I in order to improve the NN controller performance during its weight training. And of course A must also remain bounded over the iterations, otherwise the training process is not going to proceed successfully.

We assume that the NN weights are updated as follows:

$$\mathbf{W}(i+1) = \mathbf{W}(i) + \mathbf{d}(i), \tag{6}$$

where $\mathbf{d}(i)$ is an update vector. Employing the Taylor expansion of I around $\mathbf{W}(i)$ and neglecting terms higher than the first order yields

$$I(\mathbf{W}(i+1)) = I(\mathbf{W}(i)) + \frac{\partial I(i)}{\partial \mathbf{W}(i)}^T (\mathbf{W}(i+1) - \mathbf{W}(i)). \tag{7}$$

Substituting for $(\mathbf{W}(i+1) - \mathbf{W}(i))$ from (6) yields

$$I(\mathbf{W}(i+1)) = I(\mathbf{W}(i)) + \frac{\partial I(i)}{\partial \mathbf{W}(i)}^T \mathbf{d}(i). \tag{8}$$

The growth of I with iterations i is guaranteed if

$$\frac{\partial I(i)}{\partial \mathbf{W}(i)}^T \mathbf{d}(i) > 0. \tag{9}$$

Alternatively, the decrease of I is assured if the inequality above is strictly negative; this is suitable for cost minimization problems, e.g., when $U(t) = (yr(t) - yp(t))^2$, which is popular in tracking problems.

It is popular to use gradients as the weight update

$$\mathbf{d}(i) = \eta(i) \frac{\partial A(i)}{\partial \mathbf{W}(i)}, \tag{10}$$

where $\eta(i) > 0$ is a learning rate. However, it is often much more effective to rely on updates computed with the help of second-order information; see Sect. 4 for details.

The condition (9) actually clarifies what it means for A to be an adequate substitute for R. The plant model is often required to train the NN controller. The model needs to provide accurate enough $\mathbf{d}$ such that (9) is satisfied. Interestingly, from the standpoint of NN controller training it is not critical to have a good match between plant outputs $\mathbf{yp}$ and their approximations by the model $\mathbf{ym}$. Coarse plant models which approximate well input-output sensitivities in the plant are sufficient. This has been noticed and successfully exploited by several researchers [58–61].

In practice, of course it is not possible to guarantee that (9) always holds. This is especially questionable when even simpler approximations of R are employed, as is sometimes the case in practice, e.g., $S = 1$ and/or $H = 1$ in (5). However, if the behavior of $R(i)$ over the iterations i evolves towards its improvement, i.e., the trend is that R grows with i but not necessarily $R(i) < R(i+1), \forall i$, this would suggest that (9) does hold.

Our analysis above explains how the NN controller performance can be improved through training with imperfect models. It is in contrast with other studies, e.g., [62, 63], where the key emphasis is on proving the

uniform ultimate boundedness (UUB) [64], which is not nearly as important in practice as the performance improvement because performance implies boundedness.

In terms of NN controller adaptation and in addition to the division of control to indirect and direct schemes, two adaptation extremes exist. The first is represented by the classic approach of fully adaptive NN controller which learns "on-the-fly," often without any prior knowledge; see, e.g., [65, 66]. This approach requires a detailed mathematical analysis of the plant and many assumptions, relegating NN to mere uncertainty compensators or look-up table replacement. Furthermore, the NN controller usually does not retain its long-term memory as reflected in the NN weights.

The second extreme is the approach employing NN controllers with weights fixed after training which relies on recurrent NN. It is known that RNN with fixed weights can imitate algorithms [67–72] or adaptive systems [73] after proper training. Such RNN controllers are not supposed to require adaptation after deployment/in operation, thereby substantially reducing implementation cost especially in on-board applications. Figure 7 illustrates how a fixed-weight RNN can replace a set of controllers, each of which is designed for a specific operation mode of the time-varying plant. In this scheme the fixed-weight, trained RNN demonstrates its ability to generalize in the space of tasks, rather than just in the space of input-output vector pairs as non-recurrent networks do (see, e.g., [74]). As in the case of a properly trained non-recurrent NN which is very good at dealing with data similar to its training data, it is reasonable to expect that RNN can be trained to be good interpolators only in the space of tasks it has seen during training, meaning that significant extrapolation beyond training data is to be neither expected nor justified.

The fixed-weight approach is very suitable to such practically useful direction as training RNN off-line, i.e., on high-fidelity simulators of real systems, and preparing RNN through training to various sources of uncertainties and disturbances that can be encountered during system operation. And the performance of the trained RNN can also be verified on simulators to increase confidence in successful deployment of the RNN.

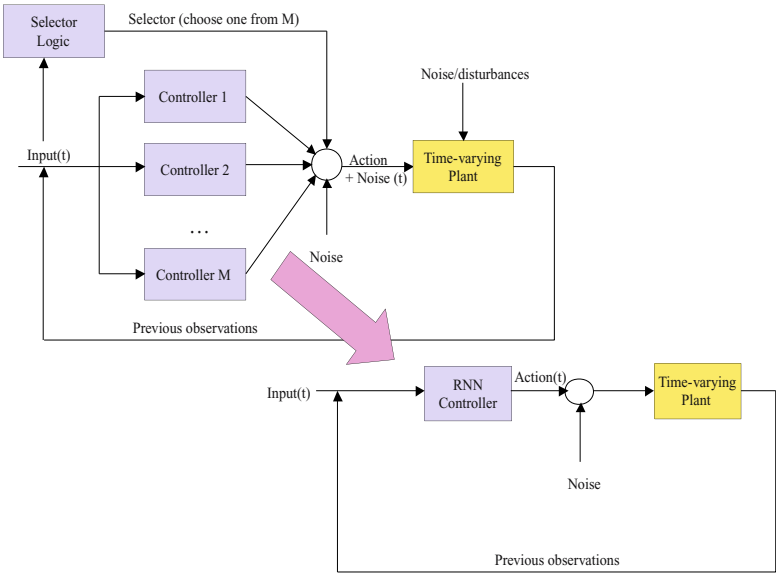

Fig. 7. A fixed-weight, trained RNN can replace a popular control scheme which includes a set of controllers specialized to handle different operating modes of the time-varying plant and a controller selector algorithm which chooses an appropriate controller based on the context of plant operation (input, feedback, etc.)

The fully adaptive approach is preferred if the plant may undergo very significant changes during its operation, e.g., when faults in the system force its performance to change permanently. Alternatively, the fixed-weight approach is more appropriate if the system may be repaired back to its normal state after the fault is corrected [32]. Various combinations of the two approaches above (hybrids of fully adaptive and fixed-weight approaches) are also possible [75].

Before concluding this section we would like to discuss on-line training implementation. On-line or continuous training occurs when the plant can not be returned to its initial state to begin another iteration of training, and it must be run continuously. This is in contrast with off-line training which assumes that the plant (its model in this case) can be reset to any specified state at any time.

On-line training can be done in a straightforward way by maintaining two distinct processes (see also [58]): foreground (network execution) and background (training). Figures 8 and 9 illustrate these processes.

The processes assume at least two groups of copies of the controller C labeled $C1$ and $C2$, respectively. The controller $C1$ is used in the foreground process which directly affects the plant P through the sequence of controller outputs **a1**.

The controller $C1$ weights are periodically replaced by those of the NN controller $C2$. The controller $C2$ is trained in the background process of Fig. 9. The main difference from the previous figure is the replacement of the plant P with its model M. The model serves as a sensitivity pathway between utility U and controller $C2$ (cf. Fig. 5), thereby enabling training $C2$ weights.

The model M could be trained as well, if necessary. For example, it can be done through adding another background process for training model of the plant. Of course, such process would have its own goal, e.g., minimization of the mean squared error between the model outputs $\mathbf{ym}(t+i)$ and the plant outputs $\mathbf{yp}(t+i)$. In general, simultaneous training of the model and the controller may result in training instability, and it is better to alternate cycles of model-controller training.

When referring to training NN in this and previous sections, we did not discuss possible training algorithms. This is done in the next section.

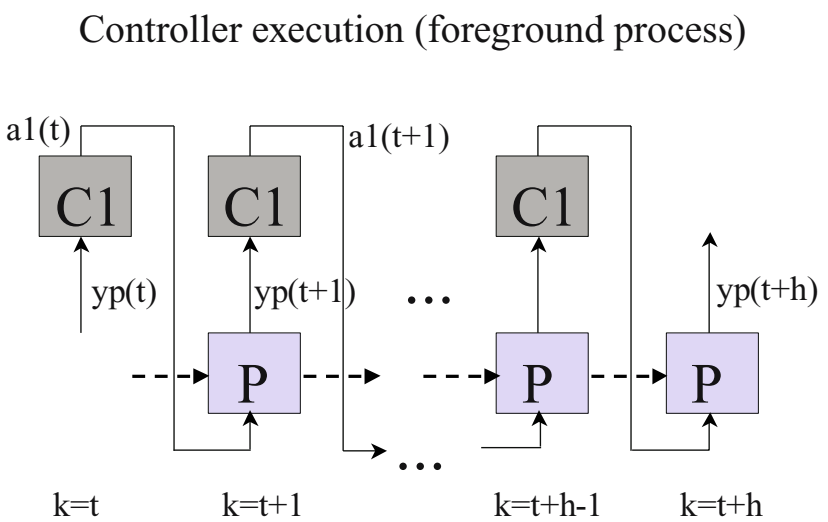

Fig. 8. The fixed-weight NN controller $C1$ influences the plant P through the controller outputs **a1** (actions) to optimize utility function U (not shown) in a temporal unfolding. The plant outputs **yp** are also shown. Note that this process in general continues for much longer than h time steps. The *dashed lines* symbolize temporal dependencies in the dynamic plant

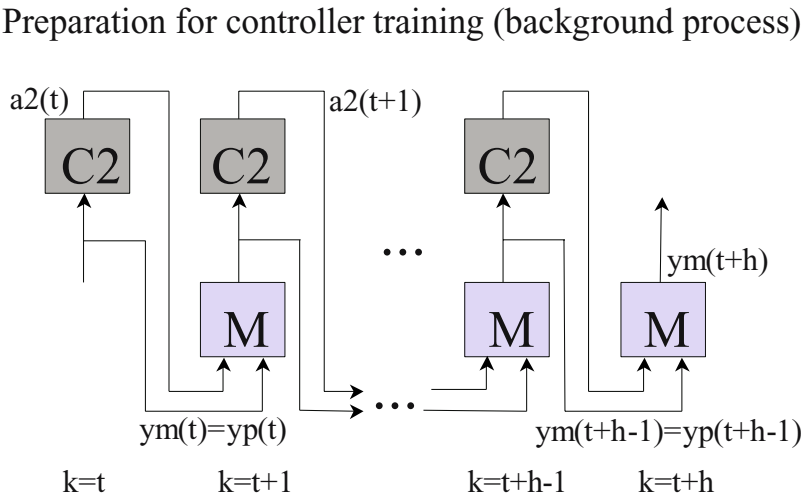

Fig. 9. Unlike the previous figure, another NN controller $C2$ and the plant model M are used here. It may be helpful to think of the current time step as step $t+h$, rather than step t. The controller $C2$ is a clone of $C1$ but their weights are different in general. The weights of $C2$ can be trained by an algorithm which requires that the temporal history of $h+1$ time steps be maintained. It is usually advantageous to align the model with the plant by forcing their outputs to match perfectly, especially if the model is sufficiently accurate for one-step-ahead predictions only. This is often called *teacher forcing* and shown here by setting $\mathbf{ym}(t+i) = \mathbf{yp}(t+i)$. Both $C2$ and M can be implemented as recurrent NN

4 Training NN

Quite a variety of NN training methods exist (see, e.g., [13]). Here we provide an overview of selected methods illustrating diversity of NN training approaches, while referring the reader to detailed descriptions in appropriate references.

First, we discuss approaches that utilize derivatives. The two main methods for obtaining dynamic derivatives are real-time recurrent learning (RTRL) and backpropagation through time (BPTT) [76] or its truncated version BPTT(h) [77]. Often these are interpreted loosely as NN training methods, whereas they are merely the methods of obtaining derivatives to be combined subsequently with the NN weight update methods. (BPTT reduces to just BP when no dynamics needs to be accounted for in training.)

The RTRL algorithm was proposed in [78] for a fully connected recurrent layer of nodes. The name RTRL is derived from the fact that the weight updates of a recurrent network are performed concurrently with network execution. The term "forward method" is more appropriate to describe RTRL, since it better reflects the mechanics of the algorithm. Indeed, in RTRL, calculations of the derivatives of node outputs with respect to weights of the network must be carried out during the forward propagation of signals in a network.

The computational complexity of the original RTRL scales as the fourth power of the number of nodes in a network (worst case of a fully connected RNN), with the space requirements (storage of all variables) scaling as the cube of the number of nodes [79]. Furthermore, RTRL for a RNN requires that the dynamic derivatives be computed at every time step for which that RNN is executed. Such coupling of forward propagation and derivative calculation is due to the fact that in RTRL both derivatives and RNN node outputs evolve recursively. This difficulty is independent of the weight update method employed, which

might hinder practical implementation on a serial processor with limited speed and resources. Recently an effective RTRL method with quadratic scaling has been proposed [80] which approximates the full RTRL by ignoring derivatives not belonging to the same node.

Truncated backpropagation through time (BPTT(h), where h stands for the truncation depth) offers potential advantages relative to forward methods for obtaining sensitivity signals in NN training problems. The computational complexity scales as the product of h with the square of the number of nodes (for a fully connected NN). BPTT(h) often leads to a more stable computation of dynamic derivatives than do forward methods because its history is strictly finite. The use of BPTT(h) also permits training to be carried out asynchronously with the RNN execution, as illustrated in Figs. 8 and 9. This feature enabled testing a BPTT based approach on a real automotive hardware as described in [58].

As has been observed some time ago [81], BPTT may suffer from the problem of vanishing gradients. This occurs because, in a typical RNN, the derivatives of sigmoidal nodes are less than the unity, while the RNN weights are often also less than the unity. Products of many of such quantities can become naturally very small, especially for large depths h. The RNN training would then become ineffective; the RNN would be "blind" and unable to associate target outputs with distant inputs.

Special RNN approaches such as those in [82] and [83] have been proposed to cope with the vanishing gradient problem. While we acknowledge that the problem may be indeed serious, it is not insurmountable. This is not just this author's opinion but also reflection on successful experience of Ford and Siemens NN Research (see, e.g., [84]).

In addition to calculation of derivatives of the performance measure with respect to the NN weights $\mathbf{W}$, we need to choose a weight update method. We can broadly classify weight update methods according to the amount of information used to perform an update. Still, the simple equation (6) holds, while the update $\mathbf{d}(i)$ may be determined in a much more complex process than the gradient method (10).

It is useful to summarize a typical BPTT(H) based training procedure for NN controllers because it highlights steps relevant to training NN with feedback in general:

1. Initiate states of each component of the system (e.g., RNN state): $\mathbf{x}(0) = \mathbf{x}_0(s)$, $s = 1, 2, \ldots, S$.
2. Run the system forward from time step $t = t_0$ to step $t = t_0 + H$, and compute U (see (5)) for all S trajectories.
3. For all S trajectories, compute dynamic derivatives of the relevant outputs with respect to NN controller weights, i.e., backpropagate to t_0. Usually backpropagating just $U(t_0 + H)$ is sufficient.
4. Adjust the NN controller weights according to the weight update $\mathbf{d}(i)$ using the derivatives obtained in step 3; increment i.
5. Move forward by one time step (run the closed-loop system forward from step $t = t_0 + H$ to step $t_0 + H + 1$ for all S trajectories), then increment t_0 and repeat the procedure beginning from step 3, etc., until the end of all trajectories ($t = T$) is reached.
6. Optionally, generate a new set of initial states and resume training from step 1.

The described procedure is similar to both model predictive control (MPC) with receding horizon (see, e.g., [85]) and optimal control based on the adjoint (Euler–Lagrange/Hamiltonian) formulation [86]. The most significant differences are that this scheme uses a parametric nonlinear representation for controller (NN) and that updates of NN weights are incremental, not "greedy" as in the receding-horizon MPC.

We henceforth assume that we deal with root-mean-squared (RMS) error minimization (corresponds to $-\frac{\partial A(i)}{\partial \mathbf{W}(i)}$ in (10)). Naturally, gradient descent is the simplest among all first-order methods of minimization for differentiable functions, and is the easiest to implement. However, it uses the smallest amount of information for performing weight updates. An imaginary plot of total error versus weight values, known as the error surface, is highly nonlinear in a typical neural network training problem, and the total error function may have many local minima. Relying only on the gradient in this case is clearly not the most effective way to update weights. Although various modifications and heuristics have been proposed to improve the effectiveness of the first-order methods, their convergence still remains quite slow due to the intrinsically ill-conditioned nature of training problems [13]. Thus, we need to utilize more information about the error surface to make the convergence of weights faster.

In differentiable minimization, the Hessian matrix, or the matrix of second-order partial derivatives of a function with respect to adjustable parameters, contains information that may be valuable for accelerated convergence. For instance, the minimum of a function quadratic in the parameters can be reached in one iteration, provided the inverse of the nonsingular positive definite Hessian matrix can be calculated. While such superfast convergence is only possible for quadratic functions, a great deal of experimental work has confirmed that much faster convergence is to be expected from weight update methods that use second-order information about error surfaces. Unfortunately, obtaining the inverse Hessian directly is practical only for small neural networks [15]. Furthermore, even if we can compute the inverse Hessian, it is frequently ill-conditioned and not positive definite, making it inappropriate for efficient minimization. For RNN, we have to rely on methods which build a positive definite estimate of the inverse Hessian without requiring its explicit knowledge. Such methods for weight updates belong to a family of second-order methods. For a detailed overview of the second-order methods, the reader is referred to [13]. If $\mathbf{d}(i)$ in (6) is a product of a specially created and maintained positive definite matrix, sometimes called the approximate inverse Hessian, and the vector $-\eta(i)\frac{\partial A(i)}{\partial \mathbf{W}(i)}$, we obtain the quasi-Newton method. Unlike first-order methods which can operate in either pattern-by-pattern or batch mode, most second-order methods employ batch mode updates (e.g., the popular Levenberg–Marquardt method [15]). In pattern-by-pattern mode, we update weights based on a gradient obtained for every instance in the training set, hence the term *instantaneous gradient*. In batch mode, the index i is no longer applicable to individual instances, and it becomes associated with a training iteration or epoch. Thus, the gradient is usually a sum of instantaneous gradients obtained for all training instances during the epoch i, hence the name *batch gradient*. The approximate inverse Hessian is recursively updated at the end of every epoch, and it is a function of the batch gradient and its history. Next, the best learning rate $\eta(i)$ is determined via a one-dimensional minimization procedure, called line search, which scales the vector $\mathbf{d}(i)$ depending on its influence on the total error. The overall scheme is then repeated until the convergence of weights is achieved.

Relative to first-order methods, effective second-order methods utilize more information about the error surface at the expense of many additional calculations for each training epoch. This often renders the overall training time to be comparable to that of a first-order method. Moreover, the batch mode of operation results in a strong tendency to move strictly downhill on the error surface. As a result, weight update methods that use batch mode have limited error surface exploration capabilities and frequently tend to become trapped in poor local minima. This problem may be particularly acute when training RNN on large and redundant training sets containing a variety of temporal patterns. In such a case, a weight update method that operates in pattern-by-pattern mode would be better, since it makes the search in the weight space *stochastic*. In other words, the training error can jump up and down, escaping from poor local minima. Of course, we are aware that no batch or sequential method, whether simple or sophisticated, provides a complete answer to the problem of multiple local minima. A reasonably small value of RMS error achieved on an independent testing set, not significantly larger than the RMS error obtained at the end of training, is a strong indication of success. Well known techniques, such as repeating a training exercise many times starting with different initial weights, are often useful to increase our confidence about solution quality and reproducibility.

Unlike weight update methods that originate from the field of differentiable function optimization, the extended Kalman filter (EKF) method treats supervised learning of a NN as a nonlinear sequential state estimation problem. The NN weights $\mathbf{W}$ are interpreted as states of the trivially evolving dynamic system, with the measurement equation described by the NN function $\mathbf{h}$

$$\mathbf{W}(t+1) = \mathbf{W}(t) + \boldsymbol{\nu}(t), \tag{11}$$

$$\mathbf{y}^d(t) = \mathbf{h}(\mathbf{W}(t), \mathbf{i}(t), \mathbf{v}(t-1)) + \boldsymbol{\omega}(t), \tag{12}$$

where $\mathbf{y}^d(t)$ is the desired output vector, $\mathbf{i}(t)$ is the external input vector, $\mathbf{v}$ is the RNN state vector (internal feedback), $\boldsymbol{\nu}(t)$ is the process noise vector, and $\boldsymbol{\omega}(t)$ is the measurement noise vector. The weights $\mathbf{W}$ may be organized into g mutually exclusive weight groups. This trades off performance of the training method with its efficiency; a sufficiently effective and computationally efficient choice, termed node decoupling, has been to group together those weights that feed each node. Whatever the chosen grouping, the weights of group j are denoted by $\mathbf{W}_j$. The corresponding derivatives of network outputs with respect to weights $\mathbf{W}_j$ are placed in N_{out} columns of $\mathbf{H}_j$.

To minimize at time step t a cost function $cost = \sum_t \frac{1}{2}\boldsymbol{\xi}(t)^T\mathbf{S}(t)\boldsymbol{\xi}(t)$, where $\mathbf{S}(t) > 0$ is a weighting matrix and $\boldsymbol{\xi}(t)$ is the vector of errors, $\boldsymbol{\xi}(t) = \mathbf{y}^d(t) - \mathbf{y}(t)$, where $\mathbf{y}(t) = \mathbf{h}(\cdot)$ from (12), the decoupled EKF equations are as follows [58]:

$$\mathbf{A}^*(t) = \left[\frac{1}{\eta(t)}\mathbf{I} + \sum_{j=1}^{g}\mathbf{H}_j^*(t)^T\mathbf{P}_j(t)\mathbf{H}_j^*(t)\right]^{-1}, \tag{13}$$

$$\mathbf{K}_j^*(t) = \mathbf{P}_j(t)\mathbf{H}_j^*(t)\mathbf{A}^*(t), \tag{14}$$

$$\mathbf{W}_j(t+1) = \mathbf{W}_j(t) + \mathbf{K}_j^*(t)\boldsymbol{\xi}^*(t), \tag{15}$$

$$\mathbf{P}_j(t+1) = \mathbf{P}_j(t) - \mathbf{K}_j^*(t)\mathbf{H}_j^*(t)^T\mathbf{P}_j(t) + \mathbf{Q}_j(t). \tag{16}$$

In these equations, the weighting matrix $\mathbf{S}(t)$ is distributed into both the derivative matrices and the error vector: $\mathbf{H}_j^*(t) = \mathbf{H}_j(t)\mathbf{S}(t)^{\frac{1}{2}}$ and $\boldsymbol{\xi}^*(t) = \mathbf{S}(t)^{\frac{1}{2}}\boldsymbol{\xi}(t)$. The matrices $\mathbf{H}_j^*(t)$ thus contain scaled derivatives of network (or the closed-loop system) outputs with respect to the jth group of weights; the concatenation of these matrices forms a global scaled derivative matrix $\mathbf{H}^*(t)$. A common global scaling matrix $\mathbf{A}^*(t)$ is computed with contributions from all g weight groups through the scaled derivative matrices $\mathbf{H}_j^*(t)$, and from all of the decoupled approximate error covariance matrices $\mathbf{P}_j(t)$. A user-specified learning rate $\eta(t)$ appears in this common matrix. (Components of the measurement noise matrix are inversely proportional to $\eta(t)$.) For each weight group j, a Kalman gain matrix $\mathbf{K}_j^*(t)$ is computed and used in updating the values of $\mathbf{W}_j(t)$ and in updating the group's approximate error covariance matrix $\mathbf{P}_j(t)$. Each approximate error covariance update is augmented by the addition of a scaled identity matrix $\mathbf{Q}_j(t)$ that represents additive data deweighting.

We often employ a multi-stream version of the algorithm above. A concept of multi-stream was proposed in [87] for improved training of RNN via EKF. It amounts to training N_s copies (N_s streams) of the same RNN with N_{out} outputs. Each copy has the same weights but different, separately maintained states. With each stream contributing its own set of outputs, every EKF weight update is based on information from all streams, with the total effective number of outputs increasing to $M = N_s N_{out}$. The multi-stream training may be especially effective for heterogeneous data sequences because it resists the tendency to improve local performance at the expense of performance in other regions.

The Stochastic Meta-Descent (SMD) is proposed in [88] for training nonlinear parameterizations including NN. The iterative SMD algorithm consists of two steps. First, we update the vector $\mathbf{p}$ of local learning rates

$$\begin{aligned}\mathbf{p}(t) &= \mathtt{diag}(\mathbf{p}(t-1)) \\ &\quad \times \mathtt{max}(0.5, 1 + \mu\mathtt{diag}(\mathbf{v}(t))\boldsymbol{\nabla}(t)),\end{aligned} \tag{17}$$

$$\mathbf{v}(t+1) = \gamma\mathbf{v}(t) + \mathtt{diag}(\mathbf{p}(t))(\boldsymbol{\nabla}(t) - \gamma\mathbf{C}\mathbf{v}(t)), \tag{18}$$

where γ is a forgetting factor, μ is a scalar meta-learning factor, $\mathbf{v}$ is an auxiliary vector, $\mathbf{C}\mathbf{v}(t)$ is the product of a curvature matrix $\mathbf{C}$ with $\mathbf{v}$, $\boldsymbol{\nabla}$ is a derivative of the instantaneous cost function with respect to $\mathbf{W}$ (e.g., the cost is $\frac{1}{2}\boldsymbol{\xi}(t)^T\mathbf{S}(t)\boldsymbol{\xi}(t)$; oftentimes $\boldsymbol{\nabla}$ is averaged over a short window of time steps).

The second step is the NN weight update

$$\mathbf{W}(t+1) = \mathbf{W}(t) - \mathtt{diag}(\mathbf{p}(t))\boldsymbol{\nabla}(t). \tag{19}$$

In contrast to EKF which uses explicit approximation of the inverse curvature $\mathbf{C}^{-1}$ as the $\mathbf{P}$ matrix (16), the SMD calculates and stores the matrix-vector product $\mathbf{C}\mathbf{v}$, thereby achieving dramatic computational savings. Several efficient ways to obtain $\mathbf{C}\mathbf{v}$ are discussed in [88]. We utilize the product $\mathbf{C}\mathbf{v} = \boldsymbol{\nabla}\boldsymbol{\nabla}^T\mathbf{v}$ where we first compute the scalar product $\boldsymbol{\nabla}^T\mathbf{v}$, then scale the gradient $\boldsymbol{\nabla}$ by the result. The well adapted $\mathbf{p}$ allows the algorithm to behave as if it were a second-order method, with the dominant scaling *linear* in $\mathbf{W}$. This is clearly advantageous for problems requiring large NN.

Now we briefly discuss training methods which do not use derivatives.

ALOPEX, or ALgorithm Of Pattern EXtraction, is a correlation based algorithm proposed in [89]

$$\Delta W_{ij}(n) = \eta\Delta W_{ij}(n-1)\Delta R(n) + r_i(n). \tag{20}$$

In terms of NN variables, $\Delta W_{ij}(n)$ is the difference between the current and previous value of weight W_{ij} at iteration n, $\Delta R(n)$ is the difference between the current and previous value of the NN performance function R (not necessarily in the form of (4)), η is the learning rate, and the stochastic term $r_i(n) \sim N(0, \sigma^2)$ (a non-Gaussian term is also possible) is added to help escaping poor local minima. Related correlation based algorithms are described in [90].

Another method of non-differential optimization is called particle swarm optimization (PSO) [91]. PSO is in principle a parallel search technique for finding solutions with the highest fitness. In terms of NN, it uses multiple weight vectors, or particles. Each particle has its own position $\mathbf{W}_i$ and velocity $\mathbf{V}_i$. The particle update equations are

$$V_{i,j}^{next} = \omega V_{i,j} + c_1 \phi_{i,j}^1 (W_{ibest,j} - W_{i,j}) + c_2 \phi_{i,j}^2 (W_{gbest,j} - W_{i,j}), \tag{21}$$

$$W_{i,j}^{next} = W_{i,j} + V_{i,j}^{next}, \tag{22}$$

where the index i is the ith particle, j is its jth dimension (i.e., jth component of the weight vector), $\phi_{i,j}^1, \phi_{i,j}^2$ are uniform random numbers from zero to one, $\mathbf{W}_{ibest}$ is the best ith weight vector so far (in terms of evolution of the ith vector fitness), $\mathbf{W}_{gbest}$ is the overall best weight vector (in terms of fitness values of all weight vectors). The control parameters are termed the accelerations c_1, c_2 and the inertia ω. It is noteworthy that the first equation is to be done first for all pairs (i, j), followed by the second equation execution for all the pairs. It is also important to generate separate random numbers $\phi_{i,j}^1, \phi_{i,j}^2$ for each pair (i, j) (more common notation elsewhere omits the (i, j)-indexing, which may result in less effective PSO implementations if done literally).

The PSO algorithm is inherently a batch method. The fitness is to be evaluated over many data vectors to provide reliable estimates of NN performance.

Performance of the PSO algorithm above may be improved by combining it with particle ranking and selection according to their fitness [92–94], resulting in hybrids between PSO and evolutionary methods. In each generation, the PSO-EA hybrid ranks particles according to their fitness values and chooses the half of the particle population with the highest fitness for the PSO update, while discarding the second half of the population. The discarded half is replenished from the first half which is PSO-updated and then randomly mutated.

Simultaneous Perturbation Stochastic Approximation (SPSA) is also appealing due to its extreme simplicity and model-free nature. The SPSA algorithm has been tested on a variety of nonlinearly parameterized adaptive systems including neural networks [95].

A popular form of the gradient descent-like SPSA uses two cost evaluations *independent* of parameter vector dimensionality to carry out one update of each adaptive parameter. Each SPSA update can be described by two equations

$$W_i^{next} = W_i - aG_i(\mathbf{W}), \tag{23}$$

$$G_i(\mathbf{W}) = \frac{cost(\mathbf{W} + c\boldsymbol{\Delta}) - cost(\mathbf{W} - c\boldsymbol{\Delta})}{2c\Delta_i}, \tag{24}$$

where $\mathbf{W}^{next}$ is the updated value of the NN weight vector, $\boldsymbol{\Delta}$ is a vector of symmetrically distributed Bernoulli random variables generated anew for every update step (e.g., the ith component of $\boldsymbol{\Delta}$ denoted as Δ_i is either $+1$ or -1), c is size of a small perturbation step, and a is a learning rate.

Each SPSA update requires that two consecutive values of the cost function *cost* be computed, i.e., one value for the "positive" perturbation of weights $cost(\mathbf{W} + c\boldsymbol{\Delta})$ and another value for the "negative" perturbation $cost(\mathbf{W} - c\boldsymbol{\Delta})$ (in general, the cost function depends not only on $\mathbf{W}$ but also on other variables which are omitted for simplicity). This means that one SPSA update occurs no more than once every other time step. As in the case of the SMD algorithm (17)–(19), it may also be helpful to let the cost function represent changes of the cost over a short window of time steps, in which case each SPSA update would be even less frequent. Variations of the base SPSA algorithm are described in detail in [95].

Non-differential forms of KF have also been developed [96–98]. These replace backpropagation with many forward propagations of specially created test or *sigma* vectors. Such vectors are still only a small fraction of probing points required for high-accuracy approximations because it is easier to approximate a nonlinear

transformation of a Gaussian density than an arbitrary nonlinearity itself. These truly nonlinear KF methods have been shown to result in more effective NN training than the EKF method [99–101], but at the price of significantly increased computational complexity.

Tremendous reductions in cost of the general-purpose computer memory and relentless increase in speed of processors have greatly relaxed implementation constraints for NN models. In addition, NN architectural innovations called liquid state machines (LSM) and echo state networks (ESN) have appeared recently (see, e.g., [102]), which reduce the recurrent NN training problem to that of training just the weights of the output nodes because other weights in the RNN are fixed. Recent advances in LSM/ESN are reported in [103].

5 RNN: A Motivating Example

Recurrent neural networks are capable to solve more complex problems than networks without feedback connections. We consider a simple example illustrating the need for RNN and propose an experimentally verifiable explanation for RNN behavior, referring the reader to other sources for additional examples and useful discussions [71, 104–110].

Figure 10 illustrates two different signals, all continued after 100 time steps at the *same* level of zero. An RNN is tasked with identifying two different signals by ascribing labels to them, e.g., $+1$ to one and -1 to another. It should be clear that only a recurrent NN is capable of solving this task. Only an RNN can retain potentially arbitrarily long memory of each input signal in the region where the two inputs are no longer distinguishable (the region beyond the first 100 time steps in Fig. 10).

We chose an RNN with one input, one fully connected hidden layer of 10 recurrent nodes, and one bipolar sigmoid node as output. We employed the training based on BPTT(10) and EKF (see Sect. 4) with 150 time steps as the length of training trajectory, which turned out to be very quick due to simplicity of the task. Figure 11 illustrates results after training. The zero-signal segment is extended for additional 200 steps for testing, and the RNN still distinguishes the two signals clearly.

We examine the internal state (hidden layer) of the RNN. We can see clearly that all time series are different, depending on the RNN input; some node signals are very different, resembling the decision (output) node signal. For example, Fig. 12 shows the output of the hidden node 4 for both input signals. This hidden node could itself be used as the output node if the decision threshold is set at zero.

Our output node is non-recurrent. It is only capable of creating a separating hyperplane based on its inputs, or outputs of recurrent hidden nodes, and the bias node. The hidden layer behavior after training suggests that the RNN spreads the input signal into several dimensions such that in those dimensions the signal classification becomes easy.

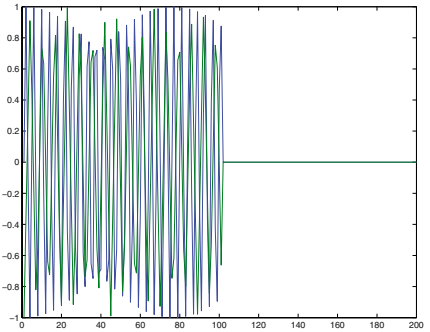

Fig. 10. Two inputs for the RNN motivating example. The *blue curve* is $\sin(5t/\pi)$, where $t = [0 : 1 : 100]$, and the *green curve* is $sawtooth(t, 0.5)$ (Matlab notation)

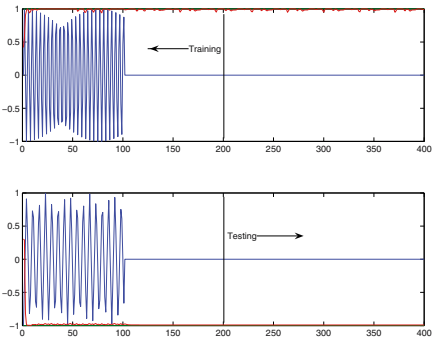

Fig. 11. The RNN results after training. The segment from 0 to 200 is for training, the rest is for testing

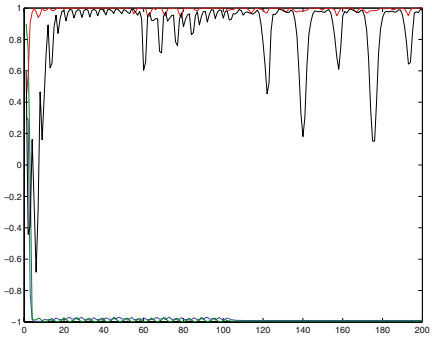

Fig. 12. The output of the hidden node 4 of the RNN responding to the first (*black*) and the second (*green*) input signals. The response of the output node is also shown in *red* and *blue* for the first and the second signal, respectively

The hidden node signals in the region where the input signal is zero do not have to converge to a fixed point. This is illustrated in Fig. 13 for the segment where the input is zero (the top panel). It is sufficient that the hidden node behavior for each signal of a particular class belong to a distinct region of the hidden node state space, non-overlapping with regions for other classes. Thus, oscillatory or even chaotic behavior for hidden nodes is possible (and sometimes advantageous – see [110] and [109] for useful discussions), as long as a separating hyperplane exists for the output to make the classification decision. We illustrate in Fig. 11 the long retention by testing the RNN on added 200-point segments of zero inputs to each of the training signals.

Though our example is for two classes of signals, it is straightforward to generalize it to multi-class problems. Clearly, not just classification problems but also regression problems can be solved, as demonstrated previously in [73], often with the addition of hidden (not necessarily recurrent) layers.

Though we employed the EKF algorithm for training of all RNN weights, other training methods can certainly be utilized. Furthermore, other researchers, e.g., [102], recently demonstrated that one might replace training RNN weights in the hidden layer with their random initializations, provided that the hidden layer nodes exhibit sufficiently diverse behavior. Only weights between the hidden nodes and the outputs would have to be trained, thereby greatly simplifying the training process. Indeed, it is plausible that even random weights in the RNN could sometimes result in sufficiently well separated responses to input signals of different

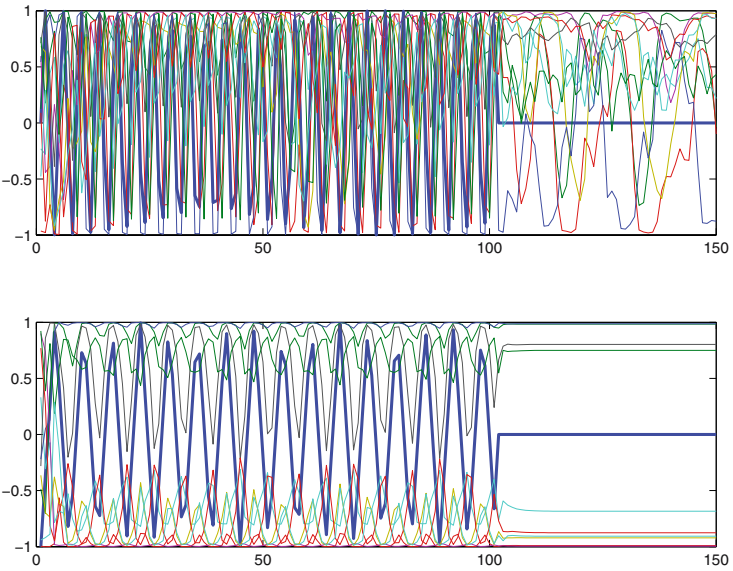

Fig. 13. The hidden node outputs of the RNN and the input signal (*thick blue line*) of the first (*top*) and the second (*bottom*) classes

classes, and this would also be consistent with our explanation for the trained RNN behavior observed in the example of this section.

6 Verification and Validation (V & V)

Verification and validation of performance of systems containing NN is a critical challenge of today and tomorrow [111, 112]. Proving mathematically that a NN will have the desired performance is possible, but such proofs are only as good as their assumptions. Sometimes too restrictive, hard to verify or not very useful assumptions are put forward just to create an appearance of mathematical rigor. For example, in many control papers a lot of efforts is spent on proving the uniform ultimate boundedness (UUB) property without due diligence demanded in practice by the need to control the value of that ultimate bound. Thus, stability becomes a proxy for performance, which is not often the case. In fact, physical systems in the automotive world (and in many other worlds too) are always bounded because of both physical limits and various safeguards.

As mentioned in Sect. 3, it is reasonable to expect that a trained NN can do an adequate job *interpolating* to other sets of data it has not seen in training. *Extrapolation* significantly beyond the training set is not reasonable to expect. However, some automotive engineers and managers who are perhaps under pressure to deploy a NN system as quickly as possible may forget this and insist that the NN be tested on data which differs as much as possible from the training data, which clearly runs counter to the main principle of designing experiments with NN.

The inability to prove rigorously superior performance of systems with NN should not discourage automotive engineers from deploying such systems. Various high-fidelity simulators, HILS, etc., are simplifying the work of performance verifiers. As such, these systems are already contributing to growing popularity of statistical methods for performance verification because other alternatives are simply not feasible [113–116].

To illustrate statistical approach of performance verification of NN, we consider the following performance verification experiment. Assume that a NN is tested on N independent data sets. If the NN performance in terms of a performance measure m is better than m^d, then the experiment is considered successful, otherwise failed. The probability that a set of N experiments is successful is given by the classic formula of Bernoulli trials (see also [117])

$$Prob = (1-p)^N, \tag{25}$$

where p is unknown true probability of failure. To keep the probability of observing no failures in N trials below κ even if $p \geq \epsilon$ requires

$$(1-\epsilon)^N \leq \kappa, \tag{26}$$

which means

$$N \geq \frac{\ln\kappa}{\ln(1-\epsilon)} = \frac{\ln\frac{1}{\kappa}}{\ln\frac{1}{1-\epsilon}} \approx \frac{1}{\epsilon}\ln\frac{1}{\kappa}. \tag{27}$$

If $\epsilon = \kappa = 10^{-6}$, then $N \geq 1.38 \times 10^7$. It would take less than 4 h of testing (3.84 h), assuming that a single verification experiment takes 1 ms.

Statistical performance verification illustrated above is applicable to other "black-box" approaches. It should be kept in mind that a NN is seldom the only component in the entire system. It may be useful and safer in practice to implement a hybrid system, i.e., a combination of a NN module ("black box") and a module whose functioning is more transparent than that of NN. The two modules together (and possibly the plant) form a system with desired properties. This approach is discussed in [8], which is the next chapter of the book.

References

1. Ronald K. Jurgen (ed). *Electronic Engine Control Technologies*, 2nd edition. Society of Automotive Engineers, Warrendale, PA, 2004.
2. Bruce D. Bryant and Kenneth A. Marko, "Case example 2: data analysis for diagnostics and process monitoring of automotive engines", in Ben Wang and Jay Lee (eds), *Computer-Aided Maintenance: Methodologies and Practices*. Berlin Heidelberg New York: Springer, 1999, pp. 281–301.
3. A. Tascillo and R. Miller, "An in-vehicle virtual driving assistant using neural networks," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, vol. 3, July 2003, pp. 2418–2423.
4. Dragan Djurdjanovic, Jianbo Liu, Kenneth A. Marko, and Jun Ni, "Immune systems inspired approach to anomaly detection and fault diagnosis for engines," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN) 2007*, Orlando, FL, 12–17 August 2007, pp. 1375–1382.
5. S. Chiu, "Developing commercial applications of intelligent control," *IEEE Control Systems Magazine*, vol. 17, no. 2, pp. 94–100, 1997.
6. A.K. Kordon, "Application issues of industrial soft computing systems," in Fuzzy Information Processing Society, 2005. NAFIPS 2005. Annual Meeting of the North American Fuzzy Information Processing Society, 26–28 June 2005, pp. 110–115.
7. A.K. Kordon, *Applied Soft Computing*, Berlin Heidelberg New York: Springer, 2008.
8. G. Bloch, F. Lauer, and G. Colin, "On learning machines for engine control." Chapter 8 in this volume.
9. K.A. Marko, J. James, J. Dosdall, and J. Murphy, "Automotive control system diagnostics using neural nets for rapid pattern classification of large data sets," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN) 1989*, vol. 2, Washington, DC, July 1989, pp. 13–16.
10. G.V. Puskorius and L.A. Feldkamp, "Neurocontrol of nonlinear dynamical systems with Kalman filter trained recurrent networks," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 279–297, 1994.
11. Naozumi Okuda, Naoki Ishikawa, Zibo Kang, Tomohiko Katayama, and Toshio Nakai, "HILS application for hybrid system development," *SAE Technical Paper No. 2007-01-3469*, Warrendale, PA, 2007.
12. Oleg Yu. Gusikhin, Nestor Rychtyckyj, and Dimitar Filev, "Intelligent systems in the automotive industry: applications and trends," *Knowledge and Information Systems*, vol. 12, no. 2, pp. 147–168, 2007.

13. S. Haykin. *Neural Networks: A Comprehensive Foundation*, 2nd edition. Upper Saddle River, NJ: Prentice Hall, 1999.
14. Genevieve B. Orr and Klaus-Robert Müller (eds). *Neural Networks: Tricks of the Trade*, Springer Lecture Notes in Computer Science, vol. 1524. Berlin Heidelberg New York: Springer, 1998.
15. C.M. Bishop. *Neural Networks for Pattern Recognition*. Oxford: Oxford University Press, 1995.
16. Normand L. Frigon and David Matthews. *Practical Guide to Experimental Design*. New York: Wiley, 1997.
17. Sven Meyer and Andreas Greff, "New calibration methods and control systems with artificial neural networks," *SAE Technical Paper no. 2002-01-1147*, Warrendale, PA, 2002.
18. P. Schoggl, H.M. Koegeler, K. Gschweitl, H. Kokal, P. Williams, and K. Hulak, "Automated EMS calibration using objective driveability assessment and computer aided optimization methods," *SAE Technical Paper no. 2002-01-0849*, Warrendale, PA, 2002.
19. Bin Wu, Zoran Filipi, Dennis Assanis, Denise M. Kramer, Gregory L. Ohl, Michael J. Prucka, and Eugene DiValentin, "Using artificial neural networks for representing the air flow rate through a 2.4-Liter VVT Engine," *SAE Technical Paper no. 2004-01-3054*, Warrendale, PA, 2004.
20. U. Schoop, J. Reeves, S. Watanabe, and K. Butts, "Steady-state engine modeling for calibration: a productivity and quality study," in *Proc. MathWorks Automotive Conference '07*, Dearborn, MI, 19–20 June 2007.
21. B. Wu, Z.S. Filipi, R.G. Prucka, D.M. Kramer, and G.L. Ohl, "Cam-phasing optimization using artificial neural networks as surrogate models – fuel consumption and NO_x emissions," *SAE Technical Paper no. 2006-01-1512*, Warrendale, PA, 2006.
22. Paul B. Deignan Jr., Peter H. Meckl, and Matthew A. Franchek, "The MI – RBFN: mapping for generalization," in *Proceedings of the American Control Conference*, Anchorage, AK, 8–10 May 2002, pp. 3840–3845.
23. Iakovos Papadimitriou, Matthew D. Warner, John J. Silvestri, Johan Lennblad, and Said Tabar, "Neural network based fast-running engine models for control-oriented applications," *SAE Technical Paper no. 2005-01-0072*, Warrendale, PA, 2005.
24. D. Specht, "Probabilistic neural networks," *Neural Networks*, vol. 3, pp. 109–118, 1990.
25. B. Wu, Z.S. Filipi, R.G. Prucka, D.M. Kramer, and G.L. Ohl, "Cam-phasing optimization using artificial neural networks as surrogate models – maximizing torque output," *SAE Technical Paper no. 2005-01-3757*, Warrendale, PA, 2005.
26. Silvia Ferrari and Robert F. Stengel, "Smooth function approximation using neural networks," *IEEE Transactions on Neural Networks*, vol. 16, no. 1, pp. 24–38, 2005.
27. N.A. Gershenfeld. *Nature of Mathematical Modeling*. Cambridge, MA: MIT, 1998.
28. N. Gershenfeld, B. Schoner, and E. Metois, "Cluster-weighted modelling for time-series analysis," *Nature*, vol. 397, pp. 329–332, 1999.
29. D. Prokhorov, L. Feldkamp, and T. Feldkamp, "A new approach to cluster weighted modeling," in *Proc. of International Joint Conference on Neural Networks (IJCNN)*, Washington DC, July 2001.
30. M. Hafner, M. Weber, and R. Isermann, "Model-based control design for IC-engines on dynamometers: the toolbox 'Optimot'," in *Proc. 15th IFAC World Congress*, Barcelona, Spain, 21–26 July 2002.
31. Dara Torkzadeh, Julian Baumann, and Uwe Kiencke, "A Neuro Fuzzy Approach for Anti-Jerk Control," *SAE Technical Paper 2003-01-0361*, Warrendale, PA, 2003.
32. Danil Prokhorov, "Toyota Prius HEV neurocontrol and diagnostics," *Neural Networks*, vol. 21, pp. 458–465, 2008.
33. K.A. Marko, J.V. James, T.M. Feldkamp, G.V. Puskorius, L.A. Feldkamp, and D. Prokhorov, "Training recurrent networks for classification," in *Proceedings of the World Congress on Neural Networks*, San Diego, 1996, pp. 845–850.
34. L.A. Feldkamp, D.V. Prokhorov, C.F. Eagen, and F. Yuan, "Enhanced multi-stream Kalman filter training for recurrent networks," in J. Suykens and J. Vandewalle (eds), *Nonlinear Modeling: Advanced Black-Box Techniques*. Boston: Kluwer, 1998, pp. 29–53.
35. L.A. Feldkamp and G.V. Puskorius, "A signal processing framework based on dynamic neural networks with application to problems in adaptation, filtering and classification," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2259–2277, 1998.
36. Neural Network Competition at IJCNN 2001, Washington DC, http://www.geocities.com/ijcnn/challenge.html (GAC).
37. G. Jesion, C.A. Gierczak, G.V. Puskorius, L.A. Feldkamp, and J.W. Butler, "The application of dynamic neural networks to the estimation of feedgas vehicle emissions," in *Proc. World Congress on Computational Intelligence. International Joint Conference on Neural Networks*, vol. 1, 1998, pp. 69–73.

38. R. Jarrett and N.N. Clark, "Weighting of parameters in artificial neural network prediction of heavy-duty diesel engine emissions," *SAE Technical Paper no. 2002-01-2878*, Warrendale, PA, 2002.
39. I. Brahma and J.C. Rutland, "Optimization of diesel engine operating parameters using neural networks," *SAE Technical Paper no. 2003-01-3228*, Warrendale, PA, 2003.
40. M.L. Traver, R.J. Atkinson, and C.M. Atkinson, "Neural network-based diesel engine emissions prediction using in-cylinder combustion pressure," *SAE Technical Paper no. 1999-01-1532*, Warrendale, PA, 1999.
41. L. del Re, P. Langthaler, C. Furtmüller, S. Winkler, and M. Affenzeller, "NO_x virtual sensor based on structure identification and global optimization," *SAE Technical Paper no. 2005-01-0050*, Warrendale, PA, 2005.
42. I. Arsie, C. Pianese, and M. Sorrentino,"Recurrent neural networks for AFR estimation and control in spark ignition automotive engines." Chapter 9 in this volume.
43. Nicholas Wickström, Magnus Larsson, Mikael Taveniku, Arne Linde, and Bertil Svensson, "Neural virtual sensors – estimation of combustion quality in SI engines using the spark plug," in *Proc. ICANN 1998*.
44. R.J. Howlett, S.D. Walters, P.A. Howson, and I. Park, "Air–fuel ratio measurement in an internal combustion engine using a neural network," Advances in Vehicle Control and Safety (International Conference), AVCS'98, Amiens, France, 1998.
45. H. Nareid, M.R. Grimes, and J.R. Verdejo, "A neural network based methodology for virtual sensor development," *SAE Technical Paper no. 2005-01-0045*, Warrendale, PA, 2005.
46. M.R. Grimes, J.R. Verdejo, and D.M. Bogden, "Development and usage of a virtual mass air flow sensor," *SAE Technical Paper no. 2005-01-0074*, Warrendale, PA, 2005.
47. W. Thomas Miller III, Richard S. Sutton, and Paul. J. Werbos (eds). *Neural Networks for Control*. Cambridge, MA: MIT, 1990.
48. K.S. Narendra, "Neural networks for control: theory and practice," *Proceedings of the IEEE*, vol. 84, no. 10, pp. 1385–1406, 1996.
49. J. Suykens, J. Vandewalle, and B. De Moor. *Artificial Neural Networks for Modeling and Control of Non-Linear Systems*. Boston: Kluwer, 1996.
50. T. Hrycej. *Neurocontrol: Towards an Industrial Control Methodology*. New York: Wiley, 1997.
51. M. Nørgaard, O. Ravn, N.L. Poulsen, and L.K. Hansen. *Neural Networks for Modelling and Control of Dynamic Systems*. London: Springer, 2000.
52. M. Agarwal, "A systematic classification of neural-network-based control," *IEEE Control Systems Magazine*, vol. 17, no. 2, pp. 75–93, 1997.
53. R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT, 1998.
54. P.J. Werbos, "Approximate dynamic programming for real-time control and neural modeling," in D.A. White and D.A. Sofge (eds), *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*. New York: Van Nostrand, 1992.
55. R.S. Sutton, A.G. Barto, and R. Williams, "Reinforcement learning is direct adaptive optimal control," *IEEE Control Systems Magazine*, vol. 12, no. 2, pp. 19–22, 1991.
56. Danil Prokhorov, "Training recurrent neurocontrollers for real-time applications," *IEEE Transactions on Neural Networks*, vol. 18, no. 4, pp. 1003–1015, 2007.
57. D. Liu, H. Javaherian, O. Kovalenko, and T. Huang, "Adaptive critic learning techniques for engine torque and air–fuel ratio control," *IEEE Transactions on Systems, Man and Cybernetics. Part B, Cybernetics*, accepted for publication.
58. G.V. Puskorius, L.A. Feldkamp, and L.I. Davis Jr., "Dynamic neural network methods applied to on-vehicle idle speed control," *Proceedings of the IEEE*, vol. 84, no. 10, pp. 1407–1420, 1996.
59. D. Prokhorov, R.A. Santiago, and D.Wunsch, "Adaptive critic designs: a case study for neurocontrol," *Neural Networks*, vol. 8, no. 9, pp. 1367–1372, 1995.
60. Thaddeus T. Shannon, "Partial, noisy and qualitative models for adaptive critic based neuro-control," in *Proc. of International Joint Conference on Neural Networks (IJCNN) 1999*, Washington, DC, 1999.
61. Pieter Abbeel, Morgan Quigley, and Andrew Y. Ng, "Using inaccurate models in reinforcement learning," in *Proceedings of the Twenty-third International Conference on Machine Learning (ICML)*, 2006.
62. P. He and S. Jagannathan, "Reinforcement learning-based output feedback control of nonlinear systems with input constraints," *IEEE Transactions on Systems, Man and Cybernetics. Part B, Cybernetics*, vol. 35, no. 1, pp. 150–154, 2005.
63. Jagannathan Sarangapani. *Neural Network Control of Nonlinear Discrete-Time Systems*. Boca Raton, FL: CRC, 2006.
64. Jay A. Farrell and Marios M. Polycarpou. *Adaptive Approximation Based Control*. New York: Wiley, 2006.
65. A.J. Calise and R.T. Rysdyk, "Nonlinear adaptive flight control using neural networks," *IEEE Control Systems Magazine*, vol. 18, no. 6, pp. 14–25, 1998.

66. J.B. Vance, A. Singh, B.C. Kaul, S. Jagannathan, and J.A. Drallmeier, "Neural network controller development and implementation for spark ignition engines with high EGR levels," *IEEE Transactions on Neural Networks*, vol. 18, No. 4, pp. 1083–1100, 2007.
67. J. Schmidhuber, "A neural network that embeds its own meta-levels," in *Proc. of the IEEE International Conference on Neural Networks*, San Francisco, 1993.
68. H.T. Siegelmann, B.G. Horne, and C.L. Giles, "Computational capabilities of recurrent NARX neural networks," *IEEE Transactions on Systems, Man and Cybernetics. Part B, Cybernetics*, vol. 27, no. 2, p. 208, 1997.
69. S. Younger, P. Conwell, and N. Cotter, "Fixed-weight on-line learning," *IEEE Transaction on Neural Networks*, vol. 10, pp. 272–283, 1999.
70. Sepp Hochreiter, A. Steven Younger, and Peter R. Conwell, "Learning to learn using gradient descent," in *Proceedings of the International Conference on Artificial Neural Networks (ICANN)*, 21–25 August 2001, pp. 87–94.
71. Lee A. Feldkamp, Danil V. Prokhorov, and Timothy M. Feldkamp, "Simple and conditioned adaptive behavior from Kalman filter trained recurrent networks," *Neural Networks*, vol. 16, No. 5–6, pp. 683–689, 2003.
72. Ryu Nishimoto and Jun Tani, "Learning to generate combinatorial action sequences utilizing the initial sensitivity of deterministic dynamical systems," *Neural Networks*, vol. 17, no. 7, pp. 925–933, 2004.
73. L.A. Feldkamp, G.V. Puskorius, and P.C. Moore, "Adaptation from fixed weight dynamic networks," in *Proceedings of IEEE International Conference on Neural Networks*, 1996, pp. 155–160.
74. L.A. Feldkamp, and G.V. Puskorius, "Fixed weight controller for multiple systems," in *Proceedings of the International Joint Conference on Neural Networks*, vol. 2, 1997, pp. 773–778.
75. D. Prokhorov, "Toward effective combination of off-line and on-line training in ADP framework," in *Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL), Symposium Series on Computational Intelligence (SSCI)*, Honolulu, HI, 1–5 April 2007, pp. 268–271.
76. P.J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.
77. R.J. Williams and J. Peng, "An efficient gradient-based algorithm for on-line training of recurrent network trajectories," *Neural Computation*, vol. 2, pp. 490–501, 1990.
78. R.J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Computation*, vol. 1, pp. 270–280, 1989.
79. R.J. Williams and D. Zipser, "Gradient-based learning algorithms for recurrent networks and their computational complexity," in Chauvin and Rumelhart (eds), *Backpropagation: Theory, Architectures and Applications*. New York: L. Erlbaum, 1995, pp. 433–486.
80. I. Elhanany and Z. Liu, "A fast and scalable recurrent neural network based on stochastic meta-descent," *IEEE Transactions on Neural Networks*, to appear in 2008.
81. Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994.
82. T. Lin, B.G. Horne, P. Tino, and C.L. Giles, "Learning long-term dependencies in NARX recurrent neural networks," *IEEE Transactions on Neural Networks*, vol. 7, no. 6, p. 1329, 1996.
83. S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
84. H.G. Zimmermann, R. Grothmann, A.M. Schfer, and Tietz, "Identification and forecasting of large dynamical systems by dynamical consistent neural networks," in S. Haykin, J. Principe, T. Sejnowski, and J. Mc Whirter (eds), *New Directions in Statistical Signal Processing: From Systems to Brain*. Cambridge, MA: MIT, 2006.
85. F. Allgöwer and A. Zheng (eds). *Nonlinear Model Predictive Control*, Progress in systems and Control Theory Series, vol. 26. Basel: Birkhauser, 2000.
86. R. Stengel. *Optimal Control and Estimation*. New York: Dover, 1994.
87. L.A. Feldkamp and G.V. Puskorius, "Training controllers for robustness: Multi-stream DEKF," in *Proceedings of the IEEE International Conference on Neural Networks,* Orlando, 1994, pp. 2377–2382.
88. N.N. Schraudolph, "Fast curvature matrix-vector products for second-order gradient descent," *Neural Computation*, vol. 14, pp. 1723–1738, 2002.
89. E. Harth, and E. Tzanakou, "Alopex: a stochastic method for determining visual receptive fields," *Vision Research*, vol. 14, pp. 1475–1482, 1974.
90. S. Haykin, Zhe Chen, and S. Becker, "Stochastic correlative learning algorithms," *IEEE Transactions on Signal Processing*, vol. 52, no. 8, pp. 2200–2209, 2004.
91. James Kennedy and Yuhui Shi. *Swarm Intelligence*. San Francisco: Morgan Kaufmann, 2001.
92. Chia-Feng Juang, "A hybrid of genetic algorithm and particle swarm optimization for recurrent network design," *IEEE Transactions on Systems, Man, and Cybernetics. Part B, Cybernetics*, vol. 34, no. 2, pp. 997–1006, 2004.

93. Swagatam Das and Ajith Abraham, "Synergy of particle swarm optimization with differential evolution algorithms for intelligent search and optimization," in Javier Bajo et al. (eds), *Proceedings of the Hybrid Artificial Intelligence Systems Workshop (HAIS06)*, Salamanca, Spain, 2006, pp. 89–99.
94. Xindi Cai, Nian Zhang, Ganesh K. Venayagamoorthy, and Donald C. Wunsch, "Time series prediction with recurrent neural networks trained by a hybrid PSO-EA algorithm," *Neurocomputing*, vol. 70, no. 13–15, pp. 2342–2353, 2007.
95. J.C. Spall and J.A. Cristion, "A neural network controller for systems with unmodeled dynamics with applications to wastewater treatment," *IEEE Transactions on Systems, Man and Cybernetics. Part B, Cybernetics*, vol. 27, no. 3, pp. 369–375, 1997.
96. S.J. Julier, J.K. Uhlmann, and H.F. Durrant-Whyte, "A new approach for filtering nonlinear systems," in *Proceedings of the American Control Conference*, Seattle WA, USA, 1995, pp. 1628–1632.
97. M. Norgaard, N.K. Poulsen, and O. Ravn, "New developments in state estimation for nonlinear systems," *Automatica*, vol. 36, pp. 1627–1638, 2000.
98. I. Arasaratnam, S. Haykin, and R.J. Elliott, "Discrete-time nonlinear filtering algorithms using Gauss–Hermite quadrature," *Proceedings of the IEEE*, vol. 95, pp. 953–977, 2007.
99. Eric A. Wan and Rudolph van der Merwe, "The unscented Kalman filter for nonlinear estimation," in *Proceedings of the IEEE Symposium 2000 on Adaptive Systems for Signal Processing, Communication and Control (AS-SPCC)*, Lake Louise, Alberta, Canada, 2000.
100. L.A. Feldkamp, T.M. Feldkamp, and D.V. Prokhorov, "Neural network training with the nprKF," in *Proceedings of International Joint Conference on Neural Networks '01*, Washington, DC, 2001, pp. 109–114.
101. D. Prokhorov, "Training recurrent neurocontrollers for robustness with derivative-free Kalman filter," *IEEE Transactions on Neural Networks*, vol. 17, no. 6, pp. 1606–1616, 2006.
102. H. Jaeger and H. Haas, "Harnessing nonlinearity: predicting chaotic systems and saving energy in wireless telecommunications," *Science*, vol. 308, no. 5667, pp. 78–80, 2004.
103. Herbert Jaeger, Wolfgang Maass, and Jose Principe (eds), "Special issue on echo state networks and liquid state machines," *Neural Networks*, vol. 20, no. 3, 2007.
104. D. Mandic and J. Chambers. *Recurrent Neural Networks for Prediction*. New York: Wiley, 2001.
105. J. Kolen and S. Kremer (eds). *A Field Guide to Dynamical Recurrent Networks*. New York: IEEE, 2001.
106. J. Schmidhuber, D. Wierstra, M. Gagliolo, and F. Gomez, "Training recurrent networks by Evolino," *Neural Computation*, vol. 19, no. 3, pp. 757–779, 2007.
107. Andrew D. Back and Tianping Chen, "Universal approximation of multiple nonlinear operators by neural networks," *Neural Computation*, vol. 14, no. 11, pp. 2561–2566, 2002.
108. R.A. Santiago and G.G. Lendaris, "Context discerning multifunction networks: reformulating fixed weight neural networks," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, Budapest, Hungary, 2004.
109. Colin Molter, Utku Salihoglu, and Hugues Bersini, "The road to chaos by hebbian learning in recurrent neural networks," *Neural Computation*, vol. 19, no. 1, 2007.
110. Ivan Tyukin, Danil Prokhorov, and Cees van Leeuwen, "Adaptive classification of temporal signals in fixed-weights recurrent neural networks: an existence proof," *Neural Computation*, to appear in 2008.
111. Brian J. Taylor (ed). *Methods and Procedures for the Verification and Validation of Artificial Neural Networks*. Berlin Heidelberg New York: Springer, 2005.
112. Laura L. Pullum, Brian J. Taylor, and Marjorie A. Darrah. *Guidance for the Verification and Validation of Neural Networks*. New York: Wiley-IEEE Computer Society, 2007.
113. M. Vidyasagar, "Statistical learning theory and randomized algorithms for control," *IEEE Control Systems Magazine*, vol. 18, no. 6, pp. 69–85, 1998.
114. R.R. Zakrzewski, "Verification of a trained neural network accuracy," in *Proceedings of International Joint Conference on Neural Networks (IJCNN)*, vol. 3, 2001, pp. 1657–1662.
115. Tariq Samad, Darren D. Cofer, Vu Ha, and Pam Binns, "High-confidence control: ensuring reliability in high-performance real-time systems," *International Journal of Intelligent Systems*, vol. 19, no. 4, pp. 315–326, 2004.
116. J. Schumann and P. Gupta, "Monitoring the performance of a neuro-adaptive controller," in *Proc. MAXENT*, American Institute of Physics Conference Proceedings 735, 2004, pp. 289–296.
117. R.R. Zakrzewski, "Randomized approach to verification of neural networks," in *Proceedings of International Joint Conference on Neural Networks (IJCNN)*, vol. 4, 2004, pp. 2819–2824.