Training Dynamic Neural Networks Using the Extended Kalman Filter for Multi-Step-Ahead Predictions

Artem Chernodub

Abstract. This paper is dedicated to single-step-ahead and multi-step-ahead time series prediction problems. We consider feedforward and recurrent neural network architectures, different derivatives calculation and optimization methods and analyze their advantages and disadvantages. We propose a novel method for training feedforward neural networks with tapped delay lines for better multi-step-ahead predictions. Special mini-batch calculations of derivatives called Forecasted Propagation Through Time for the Extended Kalman Filter training method are introduced. Experiments on well-known benchmark time series are presented.

Keywords: multi-step-ahead prediction, mini-batch Extended Kalman Filter, Forecasted Propagation Through Time, Backpropagation Through Time.

1 Introduction

Dynamics are everywhere around us. Economic and financial processes, ecological systems, industrial systems, weather fluctuations, and any kind of control system has internal dynamics governing its behavior. Even life is a dynamic system that consists of processing the information input from our senses and producing reactions to the outer environment. Surely, dealing with dynamics is required to perform many real-life tasks. Dynamic and static systems have an important difference: whereas in static systems all events are independent, dynamical systems have a memory of past events. This frequently makes solving dynamic problems much more difficult because errors are independent in the static case. For example, a single misclassification in an automated product sorting facility likely will

Artem Chernodub
Institute of Mathematical Machines and Systems NASU, Neurotechnologies Dept.,
Glushkova 42 ave., 03187 Kyiv, Ukraine

P. Koprinkova-Hristova et al. (eds.), *Artificial Neural Networks*,
Springer Series in Bio-/Neuroinformatics 4, DOI: 10.1007/978-3-319-09903-3_11

not significantly change the overall outcome of the task. In the dynamic case if the plant is unstable, errors can snowball: a single incorrect control action could result in a catastrophic accident. Good prediction quality is essential for successful control in any dynamic plant.

Neural networks are an effective and friendly tool for black-box modeling of a plant's unknown dynamics [1-3]. The neural network may be trained on examples of the dynamic plant's recorded output and then be exploited for prediction of new values. Usually, neural networks are trained to perform single-step-ahead (SS) predictions, where the predictor uses some available input and output observations to estimate the variable of interest for the time step immediately following the latest observation [4-6]. Knowing only the next nearest value is enough for many problems. However, recently there has been growing interest in multi-step-ahead (MS) predictions, where the values of interest must be predicted for some horizon in the future. Knowing the sequence of future values allows for estimation of projected amplitudes, frequencies, and variability, which are important for Model Predictive Control [7], modeling flood forecasts [8] and fault diagnostics [9]. Generally speaking, the ability to perform MS predictions is frequently treated as the "true" test for the quality of a developed empirical dynamic model. In particular, well-known echo state machine neural networks (ESNs) became popular because of their ability to perform good long-horizon ($H = 84$) multistep predictions [10].

This chapter is dedicated to time series prediction methods using dynamic neural networks. In Section 1 we consider a system identification problem that uses statistical methods to build empirical models of dynamic plants from the measured data. We formulate single-step-ahead (SS) and multi-step-ahead (MS) prediction problems using the developed empirical models. In Section 2 we consider different perceptron-like models of dynamic neural networks including feedforward (FFN) and recurrent (RNN) neural networks and provide an analysis of their strengths and weaknesses. We discuss methods for calculating static and dynamic (BPTT) derivatives for these architectures. In Section 3 we briefly consider the vanishing gradient effect and its impact on neural networks' long-term learning capabilities. In Section 4 we consider optimization algorithms for training these neural networks. We describe online and mini-batch implementations of the second-order Extended Kalman Filter (EKF) algorithm for training neural networks. We compare the EKF algorithm with the standard first-order gradient descent-based backpropagation algorithm. In Section 5 we propose a new method for training feedforward neural models to perform MS prediction, called Forecasted Propagation Through Time (FPTT). FPTT allows calculation of batch-like dynamic derivatives while minimizing the negative effect of vanishing gradients. We use mini-batch modification of the EKF algorithm that naturally deals with these batch-like dynamic derivatives for training the neural network. We present experiments on SS and MS predictions using well-known time series benchmarks.

2 Modeling the Dynamic Systems

Consider a dynamic plant:

$$\mathbf{S}(k+1) = \Phi(\mathbf{S}(k), \mathbf{u}(k)), \tag{1}$$

$$\mathbf{y}(k+1) = \Psi(\mathbf{S}(k)), \tag{2}$$

where $\mathbf{S}(k)$ is a state vector, $\mathbf{u}(k)$ is a plant's input, $\mathbf{y}(k+1)$ is an observable plant's output, and $\mathbf{\Phi}(\cdot)$ i $\mathbf{\Psi}(\cdot)$ are some nonlinear functions. The identification problem in the general case may be stated as follows: to identify *a priori* unknown functions $\Phi(\cdot)$ and $\Psi(\cdot)$ using the known history of inputs $\{\mathbf{u}(k)\}_{k=1}^{T}$ and the measured outputs $\{\mathbf{y}(k)\}_{k=1}^{T}$.

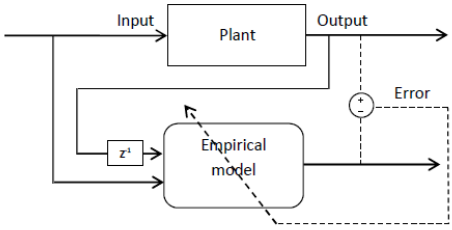

Fig. 1 General scheme of plant's identification

In this chapter we consider identification of the dynamic plants in the sense that we build empirical models of the dynamic plants, predicting time series using the concept of nonlinear autoregression (NAR). To use NAR we must make two additional important assumptions: 1) the plant's inputs $\{\mathbf{u}(k)\}_{k=1}^{T}$ are not observable and 2) the plant's states $\mathbf{S}(k)$ may be expressed as a function of the last N observable outputs of the plant:

$$\mathbf{S}(k) = \Omega(\mathbf{y}(k),...., \mathbf{y}(k-N+1)). \tag{3}$$

where k is the time step variable and $\Omega(\cdot)$ is an unknown function that defines the underlying dynamic process. Now it is possible to capture the plant's behavior using the following equation:

$$\mathbf{y}(k+1) = F(\mathbf{y}(k), \mathbf{y}(k-1),..., \mathbf{y}(k-N+1)). \tag{4}$$

Empirical models such as neural networks may be trained on known inputs and outputs of dynamic processes and then be used for estimation of new outputs using the unknown inputs (Fig. 1). By training the neural network we mean tuning

the neural network's free parameters (weights). The goal of training is to develop the empirical model $\tilde{F}(\cdot)$ of function $F(\cdot)$ as closely as possible.

If such an empirical model $\tilde{F}(\cdot)$ is available, one can perform single-step-ahead (SS) predictions $\tilde{\mathbf{y}}(\cdot)$ using the measured outputs $\mathbf{y}(\cdot)$:

$$\tilde{\mathbf{y}}(k+1) = \tilde{F}(\mathbf{y}(k), \mathbf{y}(k-1), ..., \mathbf{y}(k-N+1)). \tag{5}$$

Meanwhile, since we are working with autoregression models, we can perform iterated multi-step-ahead (MS) predictions $\hat{\mathbf{y}}(\cdot)$ using both the measured outputs $\mathbf{y}(\cdot)$ and the neural network's SS $\tilde{\mathbf{y}}(\cdot)$ and MS estimations $\hat{\mathbf{y}}(\cdot)$ if real outputs are not available:

$$\hat{\mathbf{y}}(k+2) = \tilde{F}(\hat{\mathbf{y}}(k+1), \mathbf{y}(k), ..., \mathbf{y}(k-N)), \tag{6}$$

…

$$\hat{\mathbf{y}}(k+H) = \tilde{F}(\hat{\mathbf{y}}(k+H-1), \hat{\mathbf{y}}(k+H-2), ..., \hat{\mathbf{y}}(k+H-N)), \tag{7}$$

$$\hat{\mathbf{y}}(k+H+1) = \tilde{F}(\hat{\mathbf{y}}(k+H), \hat{\mathbf{y}}(k+H-1), ..., \hat{\mathbf{y}}(k+H-N+1)), \tag{8}$$

where H is the horizon of prediction. Here and later we emphasize the differences between SS $\tilde{\mathbf{y}}(\cdot)$ and MS $\hat{\mathbf{y}}(\cdot)$ predictions, $\hat{\mathbf{y}}(k+H) \equiv \tilde{\mathbf{y}}(k+H)$ for $H=1$. We can also now define the Normalized Mean Squared Error, NMSE:

$$NMSE = \sum_k \frac{(\mathbf{t}(k) - \tilde{\mathbf{y}}(k))^2}{(\mathbf{t}(k) - \bar{\mathbf{t}}(k))^2}, \tag{9}$$

where $\mathbf{t}(\cdot)$ are target values, $\bar{\mathbf{t}}(k)$ is mean target value, $\tilde{\mathbf{y}}(\cdot)$ are predicted values.

3 Dynamic Neural Networks

Early neural network architectures (Adaline [11], Rosenblatt's Perceptron [1, p. 78], Multilayer Perceptron [1, p. 152]) were developed for solving static pattern recognition problems only. The dynamic neural networks for time series discussed in this chapter are adaptations of feedforward neural networks for pattern recognition. There are two main approaches for performing such "dynamization" of the static neural networks: 1) adding a tapped delay line to the network's inputs and 2) adding recurrent connections to the network's topology.

In the first case (Dynamic Linear Neural Network, Dynamic Multilayer Perceptron) the neural network receives the previous inputs, delayed in time, together with the current input. Training the neural network usually is performed using the well-known backpropagation method for calculating the derivatives and various gradient-based optimization methods. Advantages of this scheme are its simplicity

and convenience. Our estimation is that this scheme is used in more than 90% of cases. On the other hand, the number of delays in the tapped delay lines must be set *a priori*. If these properties do not correlate with the order of the underlying dynamic process, the neural network achieves poor results. Another disadvantage of this scheme is bad quality in the iterated multi-step-ahead predictions.

Another basic way to implement dynamics inside a feedforward neural network is adding internal recurrent connections to the input, hidden or output neurons (Recurrent Multilayer Perceptron, Elman Neural Network, etc.) A special methodology for calculating dynamic derivatives must be provided to take into account the influence of the previous time steps on the current state. Dynamic derivatives can be calculated using two techniques: Real-Time Recurrent Learning (RTRL) or Backpropagation Through Time (BPTT). Then, any gradient-based optimization algorithm for tuning weights may be used. Recurrent neural networks are more suitable for modeling the dynamic process's structure. However, training RNNs is a much more complex problem than training the FFNNs because of the additional degrees of freedom and more complicated error surface. Moreover, during calculation of the dynamic derivatives for neural networks with sigmoidal activation functions, the vanishing gradient effect occurs. This effect additionally complicates the detection of correlations between the neural network's previous inputs and current target outputs.

In this section we consider the most popular models of perceptron-like dynamic neural networks. Note that all neural network schemes described here were adopted for the autoregression case. Also, for simplicity we always assume that dimensionality of the modeled plant's output is 1, i.e. the neural network has only a single output neuron.

3.1 Dynamic Linear Neural Network

Dynamic Linear Neural Network, DLNN [2, p. 644] is the simplest model of a dynamic neural network. It consists of a single-layer perceptron and a time delay line (TDL) of order N (Fig. 2). Dynamic Linear Neural Network may be effectively used for the identification of linear dynamic plants only.

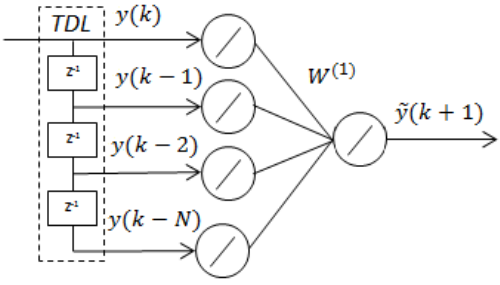

Fig. 2 Dynamic Linear Neural Network

The Neural Network receives the input

$$\mathbf{x}(k) = [\mathbf{y}(k) \quad \mathbf{y}(k-1) \quad \ldots \quad \mathbf{y}(k-N)]^T, \tag{10}$$

and calculates the output

$$\tilde{\mathbf{y}}(k+1) = \sum_{i=1}^{N_w} x_i(k) w_i^{(1)}(k), \tag{11}$$

where N_w is number of NN's weights. Error gradients $\frac{\partial E(k)}{\partial \mathbf{w}^{(1)}}$ for DLNN are calculated as:

$$\frac{\partial E(k)}{\partial w_i^{(1)}} = x_i(k) w_i^{(1)}. \tag{12}$$

3.2 Dynamic Multilayer Perceptron

The Dynamic Multilayer Perceptron [2, p. 665] (DMLP) is a much more powerful neural network architecture than the DLNN. It consists of a multilayer perceptron with non-linear units and a time delay line of order N (Fig. 3).

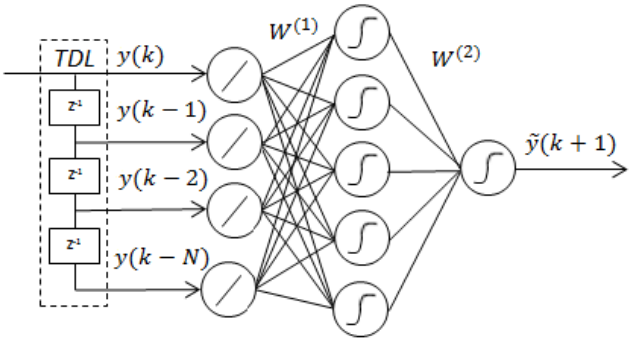

Fig. 3 Dynamic Multilayer Perceptron

DLNN receives the input vector $\mathbf{x}(k)$ (10) and calculates the output $\tilde{\mathbf{y}}(k+1)$ as:

$$z_j = f(\sum_i w_{ji}^{(1)} x_i), \tag{13}$$

$$\tilde{y}(k+1) = g(\sum_j w_j^{(2)} z_j), \tag{14}$$

where z_j is the postsynaptic value for the j-th hidden neuron, $\mathbf{w}^{(1)}$ are the hidden layer's weights, $f(\cdot)$ are the hidden layer's activation functions, $\mathbf{w}^{(2)}$ are the output layer's weights, and $g(\cdot)$ are the output layer's activation functions.

Error gradients $\frac{\partial E(k)}{\partial \mathbf{w}^{(1)}}$ and $\frac{\partial E(k)}{\partial \mathbf{w}^{(2)}}$ for the DMLP are calculated using the backpropagation technique. The procedure is the same as for the static case. Local gradients for the hidden layer $\boldsymbol{\delta}^{HID}$ are calculated and local gradients for input layer $\boldsymbol{\delta}^{IN}$ are calculated on demand as follows:

$$\delta^{OUT} = t(k+1) - \bar{y}(k+1), \tag{15}$$

$$\delta_j^{HID} = f'(z_j) w_j^{(2)} \delta^{OUT}, \tag{16}$$

$$\delta_i^{IN} = \sum_{n=1}^{K} w_{ni}^{(1)} \delta_n^{HID}, \tag{17}$$

where $t(k+1)$ is the target value, and K is number of neurons in the hidden layer. Then the error gradients are calculated:

$$\frac{\partial E(k)}{\partial w_j^{(2)}} = \delta^{OUT} z_j, \tag{18}$$

$$\frac{\partial E(k)}{\partial w_{ji}^{(1)}} = \delta_j^{IN} x_i. \tag{19}$$

3.3 Recurrent Multilayer Perceptron

The Recurrent Multilayer Perceptron (RMLP) is an adaptation of the multilayer perceptron. It is made dynamic by adding recurrent connections to the hidden layer, a so-called “context layer”.

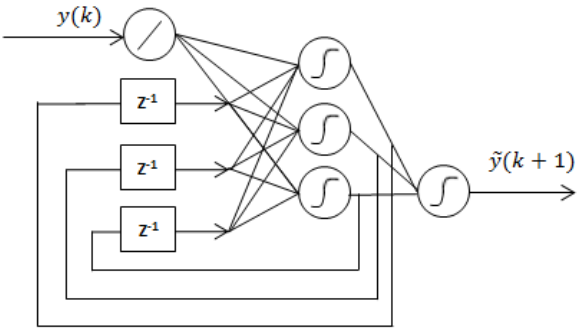

Fig. 4 Recurrent Multilayer Perceptron

The RMLP's input vector $\mathbf{x}(k)$ is:

$$\mathbf{x}(k) = [\, y(k) \quad z_1(k-1) \quad z_2(k-1) \quad \ldots \quad z_K(k-1)]^T, \tag{20}$$

where z_j is the postsynaptic value of the j-th neuron of the hidden layer, and K is the number of hidden neurons.

To consider the influence of the previous time steps to the current error residual, special dynamic Backpropagation Through Time derivatives [1, p. 836] are calculated, described in Fig. 4. An alternative for the BPTT method for calculating the dynamic derivatives is Real Time Recurrent Learning [1, p. 840]. RTRL is now rarely used because it requires more computational resources than BPTT, yet is less accurate.

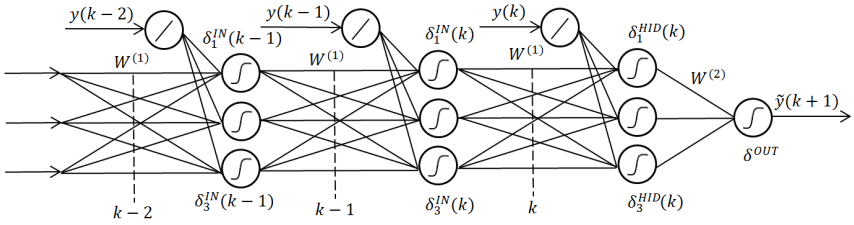

Fig. 5 Calculation of dynamic BPTT derivatives for RMLP, truncation depth h = 2

After calculating the output, the neural network is unfolded back through time (Fig. 5). The recurrent neural network is presented as a feedforward neural network with many layers, each corresponding to one of the retrospective time steps $k-1, k-2, \ldots, k-h$, where h is the BPTT truncation depth. The hyperparameter h corresponds to N, the order of the time delay line in the DMLP. Derivatives are calculated using the standard backpropagation method. Local gradients for backpropagation are calculated using the following equations:

$$\delta_j^{HID}(k) = f_j'(k) w_j^{(2)} \delta^{OUT}, \tag{21}$$

$$\delta_j^{IN}(k) = f_j'(k-1) \sum_{i=1}^{K} w_{ij}^{(1)} \delta_i^{HID}(k), \tag{22}$$

$$\delta_j^{IN}(k-n) = f_j'(k-n-1) \sum_{i=1}^{K} w_{ij}^{(1)} \delta_i^{IN}(k-n+1), \tag{23}$$

where $\mathbf{w}^{(1)}$ and $\mathbf{w}^{(2)}$ are weights of the hidden and output layer, $\boldsymbol{\delta}_j^{HID}$ is the local gradient for the j-th neuron of the hidden layer, $\boldsymbol{\delta}_j^{IN}(k-n)$ is the local gradient

for the hidden layer at retrospective time step $k-n$, $1 \le n \le h$, and h is the truncation depth.

Error gradients for the output layer $\frac{\partial E_{BPTT}(k)}{\partial \mathbf{w}^{(2)}} = \frac{\partial E(k)}{\partial \mathbf{w}^{(2)}}$ are calculated using (16). For the hidden layer $\frac{\partial E_{BPTT}(k)}{\partial \mathbf{w}^{(1)}}$, the procedure is more complex and has two stages: First, gradients for each unfolded layer are calculated for each retrospective time step $k-n$. Second, the BPTT derivatives are calculated as averaged static gradients:

$$\frac{\partial E_{BPTT}(k)}{\partial w_{ji}^{(1)}} = \frac{1}{h}\sum_{n=0}^{h}\frac{\partial E(k-n)}{\partial w_{ji}^{(1)}}. \tag{24}$$

3.4 NARX Neural Network

The NARX neural network is a dynamic neural network with two main approaches for implementing its dynamics. It is a multilayer perceptron that is equipped with both a tapped delay line at the input and global recurrent output feedback connections (Fig. 6).

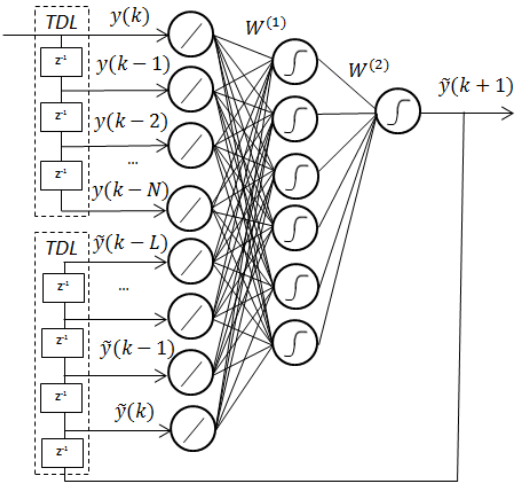

Fig. 6 NARX neural network

As it was shown in [12], NARX networks with sigmoidal activation functions are universal approximators of dynamic systems, i.e. in theory they can simulate any Turing machine. The input vector $\mathbf{x}(k)$ is:

$$\mathbf{x}(k) = [\, y(k) \quad \ldots \quad y(k-N) \quad \mathfrak{y}(k) \quad \ldots \quad \mathfrak{y}(k-L)]^T, \tag{25}$$

where N is an order of the time delay line for the input values, and L is the order of the time delay line for the recurrent feedback connections. The scheme for calculating the dynamic BPTT derivatives for NARX networks is shown in Fig. 7. Similarly to the RMLP, the NARX network must be unfolded back though time; it is presented as feedforward neural network with many layers, each layer corresponding to one of the previous time steps $k-1, k-2, \ldots, k-h$, and error is propagated back through this feedforward network.

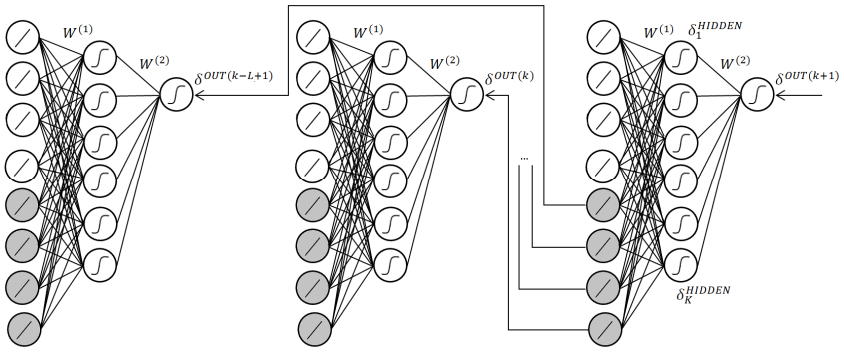

Fig. 7 Calculating the dynamic BPTT derivatives for the NARX neural network, truncation depth h = 1

During the backpropagation pass, local gradients are calculated as follows:

$$\delta_j^{HID} = f_j'(k) w_j^{(2)} \delta^{OUT(k+1)}, \tag{26}$$

$$\delta_j^{IN} = f_j'(k-1) \sum_{i=1}^{K} w_{ij}^{(1)} \delta_i^{HID}, \tag{27}$$

$$\delta^{OUT(k-l+1)} = \delta_{N+l}^{IN}, \tag{28}$$

where δ_j^{HID} is the local gradient for the j-th hidden neuron, δ_j^{IN} is the local gradient for the l-th input neuron, $1 \le l \le L$, L is the order of the time delay line for the recurrent connections, $\delta_j^{OUT(n)}$ is the output local gradient for the n-th step back through time. The final dynamic gradients for both layers $\frac{\partial E_{BPTT}(k)}{\partial \mathbf{w}^{(1)}}$ and $\frac{\partial E_{BPTT}(k)}{\partial \mathbf{w}^{(2)}}$ are averaged static gradients $\frac{\partial E(k)}{\partial \mathbf{w}^{(1)}}$ and $\frac{\partial E(k)}{\partial \mathbf{w}^{(2)}}$.

3.5 *Experiment on SS Predictions with Different Network Types*

In order to test the dynamic neural network architectures described above, we prepared 6 well-known time series: “MG17”, “ICE”, “SOLAR”, “CHICKENPOX”, “RIVER”,”LASER” (Fig. 8).

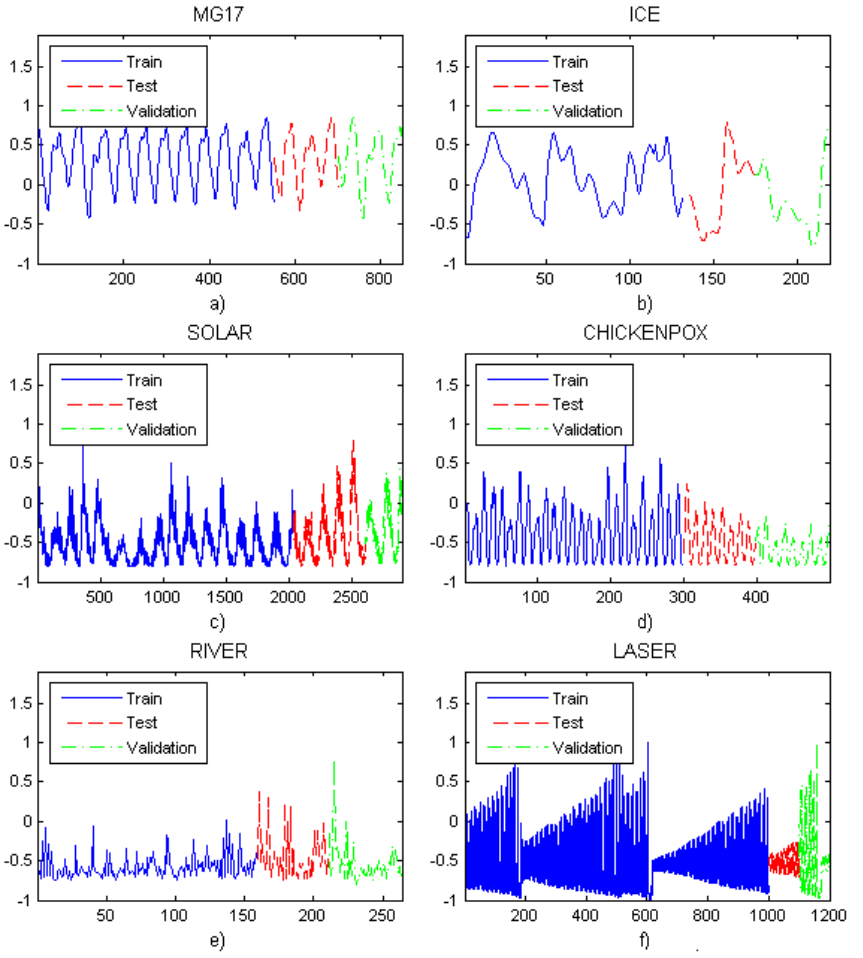

Fig. 8 Datasets for the SS-prediction experiment. a) Mackey-Glass chaotic series. b) Global ice volume dataset. c) River flow dataset. d) Monthly chickenpox instances dataset. e) River flow dataset. f) Santa-Fe Laser Dataset

The dataset “MG17” is the Mackey-Glass chaotic process. It is a famous benchmark for time series predictions. The discrete-time equation is given by the following difference equation (with delays):

$$x_{t+1} = (1-b)x_t + a\frac{x_{t-\tau}}{1+(x_{t-\tau})^{10}}, t = \tau, \tau+1, \ldots \ , \tag{29}$$

where $\tau \geq 1$ is an integer. We used the following parameters: $a = 0.1$, $b = 0.2$, $\tau = 17$ as in [6]. The dataset “ICE” represents 219 measurements of global ice volume over the last 440,000 years [13]. The dataset “SOLAR” consists of recording 2899 months of mean solar sunspots. The dataset “CHECKPOX” represents 498 months of chickenpox cases in New York City for 1931-1972 [14]. The dataset “RIVER” consists of the monthly river flows of the Sacramento River [14]. The dataset “LASER” consists of far-infrared laser intensity over a period of chaotic activity, taken from the Santa Fe competition.

For each dataset, we trained 100 DLNN, DMLP, RMLP and NARX networks using the Extended Kalman Filter (EKF) method (see Section 4). The training parameters for the EKF were set to $\eta = 10^{-3}$ and $\mu = 10^{-8}$. The number of neurons in the hidden layer for MLP-based networks was varied from 3 to 8, the order of input tapped delay line was set to $N = 5$ for the DLNN and DMLP and $N = 5$, $L = 5$ for NARX. The initial weights were set to small random values. Each network was trained for 50 - 500 epochs depending on the dataset. The best performing network on the ‘Test’ sequence was then tested on the ‘Validation’ subset of the data. The final results are presented in Table 1. The DLNN is only performing linear regression and so is not a serious competitor to the nonlinear MLP-based architectures; it is given for comparison.

Table 1 Mean NMSE Single-Step-Ahead predictions for different neural network architectures

	MG17	ICE	SOLAR	CHICKENPOX	RIVER	LASER
DLNN	0.0042	0.0244	0.1316	0.8670	4.3638	0.7711
DMLP	0.0006	0.0376	0.1313	0.4694	0.9979	0.0576
RMLP	0.0050	0.0273	0.1493	0.7608	6.4790	0.2415
NARX	0.0010	0.1122	0.1921	1.4736	2.0685	0.1332

We can see that in most cases, feedforward DMLP networks outperform recurrent networks for SS predictions. However, the DLNN shows approximately the same quality as the DMLP for the SOLAR dataset, indirectly confirming the hypothesis that the dynamic process generating solar spots is actually linear in nature. For recurrent networks it is not possible to say which architecture is significantly better: the highest and lowest performing networks vary between datasets and prediction quality may differ by 2-5 times.

4 The Vanishing Gradient Effect

The vanishing gradient effect [1, p. 846], [15], [16] significantly complicates training of ordinary perceptron-like recurrent neural networks. As described in Section 2, recurrent neural networks are first considered feedforward neural networks with many layers, each layer corresponding to one retrospective time step, in order to calculate the dynamic derivatives. When the error is propagated back through the layers the absolute values of the target derivatives exponentially decrease. This makes the detection of long-term dependencies between events difficult. To understand the internal mechanics of the vanishing gradient effect, consider the provisional feedforward multilayer perceptron that has N layers (Fig. 9):

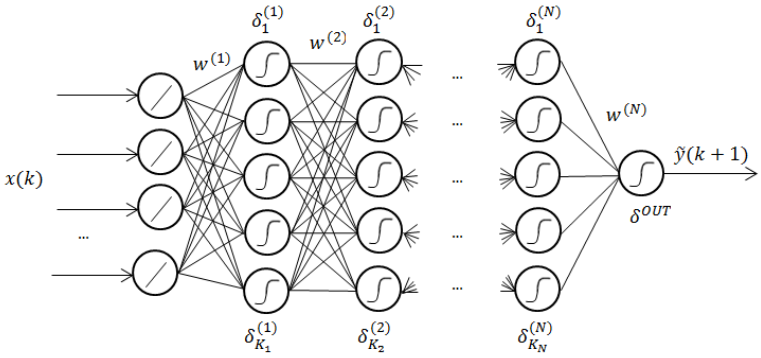

Fig. 9 Scheme of *N*-layer perceptron

Calculation of the error gradients $\frac{\partial E(k)}{\partial \mathbf{w}}$ or Jacobians $\frac{\partial \hat{\mathbf{y}}(k+1)}{\partial \mathbf{w}}$ must be performed during the backward pass. Since we use backpropagation for calculating the derivatives for training, all relevant local gradients $\boldsymbol{\delta}^{OUT} = \mathbf{e}(k)$ or $\boldsymbol{\delta}^{OUT} = \mathbf{1}$ must be propagated back through N layers respectively. Error gradients are the product of the local gradients $\delta_j^{(m)}$ and the corresponding values $z_i^{(m-1)}$:

$$\frac{\partial E(k)}{\partial w_{ji}^{(m)}} = \delta_j^{(m)} z_i^{(m-1)}. \tag{30}$$

The Jacobians $\frac{\partial \hat{\mathbf{y}}(k+1)}{\partial \mathbf{w}}$ for the neural network's training procedure are calculated using the standard backpropagation technique by propagating a constant value $\boldsymbol{\delta}^{OUT} = \mathbf{1}$ at each backward pass instead of propagating the residual error $\boldsymbol{\delta}^{OUT} = \mathbf{t}(k+1) - \hat{\mathbf{y}}(k+1)$, calculating Jacobians instead of error gradients since $\frac{\partial E(k)}{\partial \mathbf{w}} = \frac{\partial [\mathbf{e}(k)^2]}{\partial \mathbf{w}} = 2\mathbf{e}(k)\frac{\partial \hat{\mathbf{y}}(k+1)}{\partial \mathbf{w}}$.

Local gradients $\boldsymbol{\delta}^{(m)}$ are calculated for each neuron layer, starting from neuron layer $m = N$ and finishing with neuron layer $m = 1$:

$$\delta_j^{(m)} = f_j^{(m-1)\prime} \sum_i w_{ij}^{(m)} \delta_i^{(m+1)}. \tag{31}$$

In practice, it was found that for neural networks with $N > 2$ layers, the gradients (Jacobians) vanish, i.e. $\frac{\partial E(k)}{\partial w_{ji}^{(m)}} \to 0$ (or $\frac{\partial \hat{y}(k+1)}{\partial w_{ji}^{(m)}} \to 0$) for $m \to 1$. Therefore, updates to the weights become very small and the neural network cannot be trained properly.

The origins of the vanishing gradient effect lie in the calculation of the local gradients (30). During backpropagation, the absolute values of each local gradient are frequently smaller than those of the previous local gradient because they are the product of values which are all less than or equal to 1. We know that the values will always be less than or equal to 1 since the initial local gradients $\boldsymbol{\delta}^{OUT}$ never contain values greater than 1, the neural network's weights $w_{ij}^{(m)}$ cannot be large to avoid overfitting, and the derivatives of the activation functions $f_j^{(m-1)}$ are always less than 1. Moreover, in [16] it was proven that if overfitting occurs, the case where $\left| f_j^{(m-1)\prime} w_{ij}^{(m)} \text{weights } \delta_i^{(m+1)} \right| < 1.0$, the vanishing gradient effect still occurs. This happens because the relevant derivative goes to zero faster than the absolute weight can grow and local gradients $\delta^{(m)}$ exponentially decrease as a function of the current layer's number m.

For small, static pattern recognition problems a single hidden layer is often enough, so the vanishing gradient effect does not have a significant impact on the training quality. However, if one works with training datasets that contain hundreds of millions of high-dimensional training samples, the impact of the vanishing gradient effect is substantial. To use such data, one might need neural networks with many layers and millions of weights. The vanishing gradient effect plays a key role in these "Deep Neural Networks" which usually have 6-9 layers; a special "pre-training" procedure was invented [17] to avoid this effect in static networks. Unrolled dynamic recurrent neural networks may have 20-30 layers [9], so the importance of the vanishing gradient effect cannot be underestimated.

5 Optimization Methods for Dynamic Neural Network Training

There are many methods for training neural networks. Their key properties are calculating the weights' derivatives, which strongly depends on the neural network's topology and selected optimization algorithm. Here we consider gradient-based

optimization methods based on backpropagation. There are two understandings of term "backpropagation" – in a broad and in a narrow sense. In a broad sense, backpropagation is a method for calculating the derivatives of a neural network and a procedure for correcting the weights based on a gradient descent optimization technique. In a more narrow sense, which we will use here, backpropagation refers only to the technique for calculating derivatives.

5.1 Gradient Descent

Gradient descent is a first-order optimization algorithm. It is widely used for training neural networks due to its simplicity and low computational cost. Its disadvantages are low convergence speed and high risk of stopping in a local minimum. The idea of training is to move in the direction of the negative gradient in the weights space:

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \alpha\ \Delta\mathbf{w}(k), \tag{32}$$

$$\Delta\mathbf{w}(k) = -\frac{\partial E(k)}{\partial \mathbf{w}(k)}, \tag{33}$$

where α is training speed, $\mathbf{w}$ are weights coefficients, and $\frac{\partial E(k)}{\partial \mathbf{w}(k)}$ static or dynamic derivatives depending on the network's topology.

5.2 Extended Kalman Filter Training

The extended Kalman Filter method was proposed in the late 1980s [18] as an effective and efficient tool for supervised training of neural networks. Kalman Filter training shows much better fitting accuracy and faster convergence in comparison to the gradient descent-based methods. It is based on second order derivative information about the error surface, which is accumulated in the covariance matrix. This makes the Kalman Filter a good alternative to the popular second-order batch training methods such as conjugate gradient, BFGS, Levenburg-Marquardt, etc. Meanwhile, Kalman Filter methods are online (sample-by-sample), offering additional benefits. First, it makes it possible to operate in real-time. Second, KF training is less likely to converge to a local minimum due to the stochastic component in the training process [1, p. 24]. Moreover, it is not necessary to implement the regularization procedure here to decrease overfitting, because it is already implicitly implemented inside the Kalman recursion [19]. Mentioned above, a pioneering paper by S. Singhal and L. Wu produced a family of training methods called "Bayesian filtering for parameter estimation" [1], [19]. Several methods were developed: decoupled EKF training for reducing computational resources required [1, p. 855], [20, p. 33], [21], multi-stream [20, p. 35] and mini-batch [22], [23] training for escaping local minima. New designs of Kalman

Filters were used: The Ensemble Kalman Filter, EnKF [24], Unscented Kalman Filter, UKF [20, p. 221], [25], Cubature Kalman Filter, CKF) [1, p. 787], [26] and their more numerically stable square-root implementations [1, p. 773], [20, p. 273], etc. Although KF methods are usually used for training RNN networks, they can be applied to any differentiable model of a neural network including Multilayer Perceptrons, RBF networks, belief networks and others.

The Kalman Filter deals with the dynamics of the training process, so the existence of recurrent connections in the network's topology is not a necessary condition. Training the neural network is considered a state estimation problem of some unknown "ideal" neural network that provides zero residual. In this case, the states are the neural network's weights $\mathbf{w}(k)$, and the residual is the current training error $\mathbf{e}(k)$,

$$e(k) = t(k+1) - \hat{y}(k+1), \tag{34}$$

where $t(k+1)$ is a target and $\hat{y}(k+1)$ is NN's output.

The dynamic training process can be described by equations (33) and (34) using the state space model. The state transition equation (33) describes the evolution in time of the neural network's weights $\mathbf{w}(k)$ under the influence of the random Gaussian process $\xi(n)$ with zero mean and diagonal covariance matrix $\mathbf{Q}$:

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \xi(k). \tag{35}$$

Measurement equation (34) is considered a linearized NN model at the time step k. It is noised by random Gaussian noise $\zeta(k)$ with zero mean and known covariance matrix $\mathbf{R}$:

$$\mathbf{h}(k) = \frac{\partial \mathbf{y}(\mathbf{w}(k), \mathbf{z}(k), \mathbf{x}(k))}{\partial \mathbf{w}} + \zeta(k), \tag{36}$$

where $\mathbf{w}(k)$ are weights, $\mathbf{z}(k)$ are postsynaptic values , $\mathbf{x}(k)$ are input values of the neural network. The Jacobians $\frac{\partial \mathbf{y}}{\partial \mathbf{w}}$ for the neural network's training procedure are calculated using the standard backpropagation procedure by propagating the value $\delta^{OUT} = 1$ at each backward pass.

During the initialization step, the covariance matrices of measurement noise $\mathbf{R} = \eta \mathbf{I}$ and dynamic training noise $\mathbf{Q} = \mu \mathbf{I}$ are set. Matrix $\mathbf{R}$ has size $O \times O$ and matrix Q has size $N_w \times N_w$, where O is the number of output neurons (we assume $O = 1$) and N_w is the number of weight coefficients. Coefficient η is inverse to the training speed, usually $\eta \sim 10^{-2} ... 10^{-4}$, and coefficient μ defines the measurement noise, usually $\mu \sim 10^{-4} ... 10^{-8}$. Also, the identity covariance matrix $\mathbf{P}$ of size $N_w \times N_w$ and zero observation matrix $\mathbf{H}$ of size $O \times N_w$ are defined.

The following steps must be performed for all elements of the training dataset:
1) Forward pass: the neural network's output $\hat{\mathbf{y}}(k+1)$ is calculated.

2) Backward pass: Jacobians $\frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{w}}$ are calculated using backpropagation. Observation matrix $\mathbf{H}(k)$ is filled:

$$\mathbf{H}(k) = \left[\frac{\partial \hat{y}(k+1)}{\partial w_1} \quad \frac{\partial \hat{y}(k+1)}{\partial w_2} \quad \ldots \quad \frac{\partial \hat{y}(k+1)}{\partial w_{N_w}} \right]. \tag{37}$$

3) Residual matrix $\mathbf{E}(k)$ is filled:

$$\mathbf{E}(k) = \left[t(k+1) - \hat{y}(k+1) \right]. \tag{38}$$

4) New weights $\mathbf{w}(k)$ and correlation matrix $\mathbf{P}(k+1)$ are calculated:

$$\mathbf{K}(k) = \mathbf{P}(k)\mathbf{H}(k)^T [\mathbf{H}(k)\mathbf{P}(k)\mathbf{H}(k)^T + \mathbf{R}]^{-1}, \tag{39}$$

$$\mathbf{P}(k+1) = \mathbf{P}(k) - \mathbf{K}(k)\mathbf{H}(k)\mathbf{P}(k) + \mathbf{Q}, \tag{40}$$

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mathbf{K}(k)\mathbf{E}(k). \tag{41}$$

5.3 *Experiment: Gradient Descent vs Extended Kalman Filter*

In order to compare the sequential optimization algorithms we trained 100 Dynamic Multilayer Perceptrons on the Mackey-Glass chaotic process (please, see Section 2.5 for details). The training speed for the Gradient Descent $\alpha = 0.01$, the training parameters for the EKF were set to $\eta = 10^{-3}$ and $\mu = 10^{-8}$. Initial weights of all neural networks were exactly the same for the both methods.

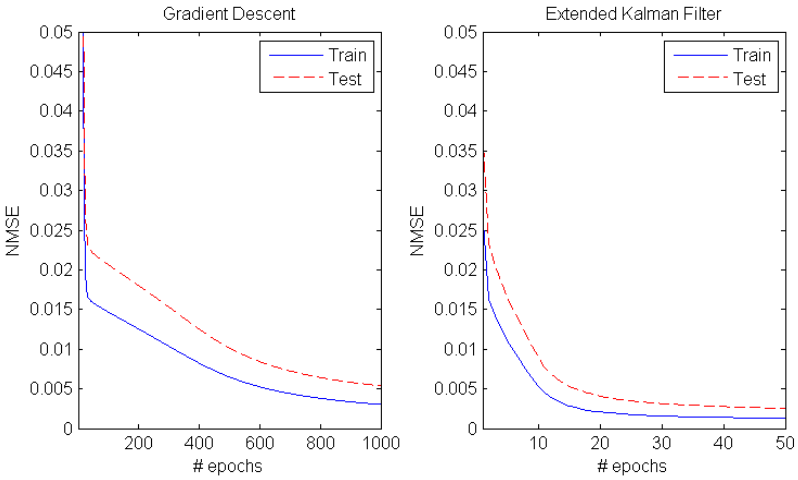

Fig. 10 Left: training 100 DMLPs with the Gradient Descent. Right: training 100 DMLPs with the Extended Kalman Filter.

The evolution of errors in time situation shown on Fig. 10 is typical for such comparison: we can see that EKF converges extremely faster than GD. Final test error was 0.0055 for GD vs 0.0026 for EKF (validation step was not performed in this experiment). At the same time, total training time of 100 DMLPs was 100.54 seconds for GD versus 7.18 seconds for EKF.

5.4 Mini-Batch Extended Kalman Filter Training

The EKF training algorithm also has a mini-batch form [23]. In this case, a batch size of B patterns and a neural network with O outputs is treated as training a single shared-weight network with $O \times B$ outputs, i.e. B data streams which feed B networks constrained to have identical weights are formed from the training set. A single weight update is calculated and applied equally to each stream's network. This weights update is sub-optimal for all samples in the mini-batch. If streams are taken from different places in the dataset, then this trick becomes equivalent to a Multistream EKF [21], a well-known technique for avoiding poor local minima. The mini-batch observation matrix $\mathbf{H}_{BATCH}(k)$ and residual matrix $\mathbf{E}_{BATCH}(k)$ now become:

$$\mathbf{H}_{BATCH}(k) = \begin{bmatrix} \frac{\partial \bar{y}(k+1)}{\partial w_1} & \cdots & \frac{\partial \bar{y}(k+1)}{\partial w_{N_w}} \\ \cdots & \cdots & \cdots \\ \frac{\partial \bar{y}(k+B)}{\partial w_1} & \cdots & \frac{\partial \bar{y}(k+B)}{\partial w_{N_w}} \end{bmatrix}, \tag{42}$$

$$\mathbf{E}_{BATCH}(k) = \begin{bmatrix} t(k+1) - \bar{y}(k+1) & \ldots & t(k+B) - \bar{y}(k+B) \end{bmatrix}. \tag{43}$$

Note that outputs $\bar{y}(\cdot)$ for calculating the training derivatives and residuals for mini-batch EKF are pure SS predictions. If derivatives $\frac{\partial \bar{y}(\cdot)}{\partial w}$ are calculated dynamically (e.g., BPTT) they provide better MS prediction quality.

The size of matrix $\mathbf{R}$ is $(O \times B) \times (O \times B)$, the size of matrix $\mathbf{H}_{BATCH}(k)$ is $(O \times B) \times N_w$, and the size of matrix $\mathbf{E}_{BATCH}(k)$ is $(O \times H) \times 1$. The remainder is identical to regular EKF. Again, here without loss of generality we assume $O = 1$. However, the mini-batch method requires at least B more calculations at each time step in comparison to the original EKF method.

6 Training FFNNs for MS Predictions Using FPTT and Mini-Batch EKF

Recurrent neural networks trained using Backpropagation Through Time show better multi-step-ahead prediction quality than feedforward neural networks trained using backpropagation. The underlying idea of BPTT is to calculate derivatives by propagating the errors back across the RNN, which is unfolded through time. This penalizes the network for accumulating errors in time and therefore provides better MS predictions. Nonetheless, RNNs have some disadvantages. First, the implementation of RNNs is harder than feedforward neural networks (FFNNs) in industrial settings. Second, training the RNNs is a difficult problem because of their more complicated error surfaces and vanishing gradient effects. Third, the internal dynamics of RNNs are less amenable to stability analysis. All of the above reasons prevent RNNs from becoming widely popular in industry. Meanwhile, RNNs have inspired a new family of methods for training FFNNs to perform MS predictions called direct methods [6]. Accumulated error is backpropagated through an unfolded through time FFNN in BPTT style that causes minimization of the MS prediction error. Nevertheless, the vanishing gradient effect still occurs in all multilayer perceptron-based networks with sigmoidal activation functions.

We propose a new, effective method for training the feedforward neural models to perform MS prediction, called Forecasted Propagation Through Time (FPTT), for calculating the batch-like dynamic derivatives that minimize the negative effect of vanishing gradients. We use the mini-batch modification of the EKF algorithm which naturally deals with these batch-like dynamic derivatives for training the neural network.

The scheme for calculating the dynamic derivatives for the FFNNs, called Forecasted Propagation Through Time, is depicted in Fig. 11.

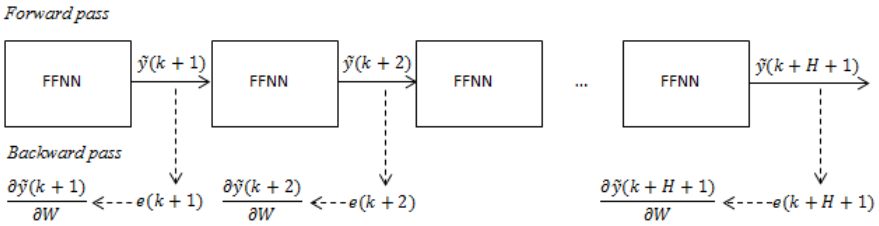

Fig. 11 Calculation of dynamic FPTT derivatives for feedforward neural networks

1. At each time step the neural network is unfolded forward through time H times using Eqs. (5)-(8) in the same way as it is performed for regular multi-step-ahead prediction, where H is a horizon of prediction. Outputs $\hat{y}(k+1),\ldots,\hat{y}(k+H+1)$ are calculated.

2. For each of the H forecasted time steps, prediction errors $e(k+h+1) = t(k+h+1) - \mathfrak{y}(k+h+1)$, $h = 1,...,H$ are calculated.
3. The set of independent derivatives $\left\{\frac{\partial \mathfrak{y}(k+h)}{\partial w}\right\}$, $h = 1,...,H+1$, are calculated for each copy of the unfolded neural network using the standard backpropagation of independent errors $\{e(k+h)\}$.

The mini-batch observation matrix $\mathbf{H}_{FPTT}(k)$ and residual matrix $\mathbf{E}_{FPTT}(k)$ now become:

$$\mathbf{H}_{FPTT}(k) = \begin{bmatrix} \frac{\partial \mathfrak{y}(k+1)}{\partial w_1} & \cdots & \frac{\partial \mathfrak{y}(k+1)}{\partial w_{N_w}} \\ \cdots & \cdots & \cdots \\ \frac{\partial \mathfrak{y}(k+H+1)}{\partial w_1} & \cdots & \frac{\partial \mathfrak{y}(k+H+1)}{\partial w_{N_w}} \end{bmatrix}, \tag{44}$$

$$\mathbf{E}_{FPTT}(k) = \begin{bmatrix} t(k+1) - \mathfrak{y}(k+1) & \ldots & t(k+B) - \mathfrak{y}(k+H+1) \end{bmatrix}. \tag{45}$$

Outputs $\mathfrak{y}(\cdot)$ for calculating the training derivatives and residuals here are the direct result of MS prediction at each time step. There are three main differences between the proposed FPTT and traditional BPTT. First, BPTT unfolds the neural network backward through time; FPTT unfolds the neural network recursively forward through time. This is useful from a technological point of view because this functionality must be implemented for MS predictions anyway. Second, FPTT does not backpropagate the accumulated error through the whole unfolded structure. Instead, it calculates BP for each copy of the neural network. Finally, FPTT does not average derivatives, it calculates a set of formally independent errors and a set of formally independent derivatives for future time steps. By doing this, we leave the question of contributions by each time step to the total MS error to the mini-batch EKF algorithm.

6.1 Multi-Step-Ahead on the Mackey-Glass Chaotic Process

In the first experiment on MS predictions we used the Mackey-Glass chaotic process (see details about data in Section 2.5). 500 values were used for training; the next 100 values were used for testing. First, we trained 100 DMLP networks with one hidden layer and hyperbolic tangent activation functions using traditional EKF and BP derivatives. The training parameters for EKF were set as $\eta = 10^{-3}$ and $\mu = 10^{-8}$. The number of neurons in the hidden layer was varied from 3 to 8, the order of input tapped delay line was set $N = 5$, and the initial weights were set to small random values. Each network was trained for 50 epochs. After each epoch,

MS prediction on horizon $H = 14$ on the training data was performed to select the best network. This network was then evaluated on the test sequence to achieve the final MS quality result. Second, we trained 100 DMLP networks with the same initial weights using the proposed mini-batch EKF technique together with FPTT derivatives and evaluated their MS prediction accuracy. Third, we trained 100 NARX networks (orders of tapped delay lines: $N = 5, L = 5$) using EKF and BPTT derivatives to make comparisons. The results of these experiments are presented in Table 1. Normalized Mean Square Error (NMSE) was used for the quality estimations.

Table 2 Mackey-Glass dataset: mean NMSE errors for different prediction horizon values

	H=1	H=2	H=6	H=8	H=10	H=12	H=14
DMLP EKF BP	0.0006	0.0014	0.013	0.022	0.033	0.044	0.052
DMLP BEKF FPTT	0.0017	0.0022	0.012	0.018	0.022	0.027	0.030
NARX EKF	0.0010	0.0014	0.012	0.018	0.023	0.028	0.032

6.2 *Multi-Step-Ahead on Santa-Fe Laser Dataset*

In order to explore the capability of the global behavior of DMLP using the proposed training method, we tested it on the laser data from the Santa Fe competition. The dataset consisted of laser intensities collected from the real experiment. Data was divided to training (1000 values) and testing (100 values) subsequences. This time the goal for training was to perform long-term (H=100) MS prediction. The order of the time delay line was set to $N = 25$ as in [4], the rest was the same as in the previous experiment. The obtained average NMSE for 100 DMLP networks was 0.175 for DMLP EKF BP (classic method) versus 0.082 for DMLP BEKF FPTT (proposed method). NARX networks shows NMSE 0.131 in average.

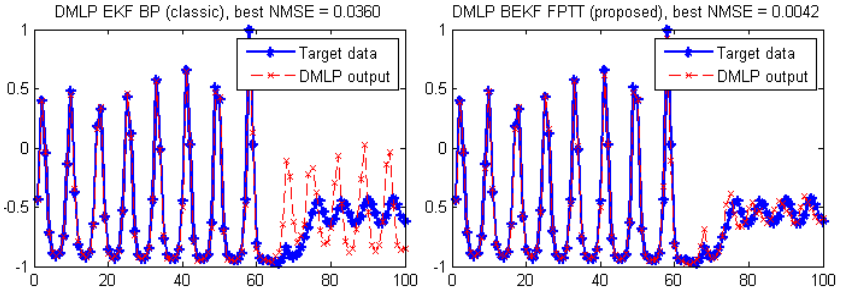

Fig. 12. The best results of the closed-loop long-term predictions (H=100) on testing data using DMLPs trained using different methods

Meanwhile, the best instance trained using mini-batch EKF+FPTT shows 10 times better accuracy than the best instance trained using the traditional approach.

7 Conclusions

We considered the multi-step-ahead prediction problem and discussed neural network based approaches as a tool for its solution. Feedforward and recurrent neural models were considered, and advantages and disadvantages of their usage were discussed. A novel direct method for training feedforward neural networks to perform multi-step-ahead predictions was proposed, based on the mini-batch Extended Kalman Filter. This method is considered to be useful from a technological point of view because it uses existing multi-step-ahead prediction functionality for calculating special FPTT dynamic derivatives that require a slight modification of the standard EKF algorithm. Our method demonstrates doubled long-term accuracy compared to standard training of the dynamic MLPs using the EKF due to direct minimization of the accumulated multi-step-ahead error.

References

1. Haykin, S.: Neural Networks and Learning Machines, 3rd edn., 936 p. Prentice Hall, New York (2009)
2. Haykin, S.: Neural Networks: A Comprehensive Foundation, 2nd edn., p. 842. Prentice Hall, Englewood Cliffs (1999)
3. Bishop, C.M.: Pattern Recognition and Machine Learning, 738 p. Springer (2006)
4. Giles, L.L., Horne, B.G., Sun-Yan, Y.: A Delay Damage Model Selection Algorithm for NARX Neural Networks. IEEE Transactions on Signal Processing 45(11), 2719–2730 (1997)
5. Parlos, A.G., Raisa, O.T., Atiya, A.F.: Multi-step-ahead prediction using dynamic recurrent neural networks. Neural Networks 13(7), 765–786 (2000)
6. Bone, R., Cardot, H.: Advanced Methods for Time Series Prediction Using Recurrent Neural Networks. In: Recurrent Neural Networks for Temporal Data Processing, ch. 2, pp. 15–36. Intech, Croatia (2011)
7. Qina, S.J., Badgwellb, T.A.: A survey of industrial model predictive control technology. Control Engineering Practice 11(7), 733–764 (2003)
8. Toth, E., Brath, A.: Multistep ahead streamflow forecasting: Role of calibration data in conceptual and neural network modeling. Water Resources Research 43(11) (2007), doi:10.1029/2006WR005383
9. Prokhorov, D.V.: Toyota Prius HEV Neurocontrol and Diagnostics. Neural Networks (21), 458–465 (2008)
10. Jaeger, H.: The “echo state” approach to analysing and training recurrent neural networks. Technical ReportGMDReport 148, German National Research Center for Information Technology (2001)
11. Anderson, J.A., Rosenfeld, E. (eds.): Talking nets: An oral history of neural networks, p. 54. MIT Press, Cambridge (1998)
12. Hava, T., Siegelmann, B.G., Horne, C.: Computational capabilities of recurrent NARX neural networks. IEEE Transactions on Systems, Man, and Cybernetics, Part B 27(2), 208–215 (1997)
13. Newton, H.J., North, G.R.: Forecasting global ice volume. J. Time Series Analysis 1991(12), 255–265 (1991)

14. Hyndman, R.J.: Time Series Data Library, http://data.is/TSDLdemo
15. Bengio, Y., Simard, P., Frasconi, P.: Learning long-term dependencies with gradient scent is difficult. IEEE Trans. Neural Networks 5(2), 157–166 (1994)
16. Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J.: Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In: A Field Guide to Dynamical Recurrent Neural Networks, 421 p. IEEE Press (2001)
17. Hinton, G., Deng, L., Yu, D., Dahl, G., Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T., Kingsbury, B.: Deep Neural Networks for Acoustic Modeling in Speech Recognition. IEEE Signal Processing Magazine 29(6), 82–97 (2012)
18. Singhal, S., Wu, L.: Training Multilayer Perceptrons with the Extended Kalman algorithm. In: Touretzky, D.S. (ed.) Advances in Neural Information Processing Systems 1, pp. 133–140. Morgan Kaufmann, San Mateo (1989)
19. Arasaratnam, I., Haykin, S.: Nonlinear Bayesian Filters for Training Recurrent Neural Networks. In: Gelbukh, A., Morales, E.F. (eds.) MICAI 2008. LNCS (LNAI), vol. 5317, pp. 12–33. Springer, Heidelberg (2008)
20. Haykin, S.: Kalman Filtering and Neural Networks, 304 p. John Wiley & Sons (2001)
21. Puskorius, G.V., Feldkamp, L.A.: Decoupled Extended Kalman Filter Training of Feedforward Layered Networks. In: International Joint Conference on Neural Networks, Seattle, July 8-14, vol. 1, pp. 771–777 (1991)
22. Li, S.: Comparative Analysis of Backpropagation and Extended Kalman Filter in Pattern and Batch Forms for Training Neural Networks. In: Proceedings on International Joint Conference on Neural Networks (IJCNN 2001), Washington, DC, July 15-19, vol. 1, pp. 144–149 (2001)
23. Chernodub, A.: Direct Method for Training Feed-Forward Neural Networks Using Batch Extended Kalman Filter for Multi-Step-Ahead Predictions. In: Mladenov, V., Koprinkova-Hristova, P., Palm, G., Villa, A.E.P., Appollini, B., Kasabov, N. (eds.) ICANN 2013. LNCS, vol. 8131, pp. 138–145. Springer, Heidelberg (2013)
24. Mirikitani, D.T., Nikolaev, N.: Dynamic Modeling with Ensemble Kalman Filter Trained Recurrent Neural Networks. In: Seventh International Conference on Machine Learning and Applications (ICMLA 2008), San Diego, USA, December 11-13 (2008)
25. Wan, E.A., van der Merwe, R.: The Unscented Kalman Filter for Nonlinear Estimation. In: Proceedings of IEEE Symposium (AS-SPCC), Lake Louise, Alberta, Canada, pp. 153–158 (October 2000)
26. Arasaratnam, I., Haykin, S.: Cubature Kalman Filters. IEEE Transactions on Automatic Control 56(6), 1254–1269