Improving Long-Term Online Prediction with Decoupled Extended Kalman Filters[⋆]

Juan Antonio Pérez-Ortiz[1], Jürgen Schmidhuber[2], Felix A. Gers[3], and Douglas Eck[2]

[1] DLSI, Universitat d'Alacant, E-03071 Alacant, Spain
[2] IDSIA, Galleria 2, 6928 Manno, Switzerland
[3] Mantik Bioinformatik GmbH, Neue Gruenstrasse 18, 10179 Berlin, Germany

Abstract. Long Short-Term Memory (LSTM) recurrent neural networks (RNNs) outperform traditional RNNs when dealing with sequences involving not only short-term but also long-term dependencies. The decoupled extended Kalman filter learning algorithm (DEKF) works well in online environments and reduces significantly the number of training steps when compared to the standard gradient-descent algorithms. Previous work on LSTM, however, has always used a form of gradient descent and has not focused on true online situations. Here we combine LSTM with DEKF and show that this new hybrid improves upon the original learning algorithm when applied to online processing.

1 Introduction

The decoupled extended Kalman filter (DEKF) [4,8] has been used successfully to optimize the training of recurrent neural networks (RNNs). In such a framework, DEKF considers learning as a filtering problem in which the optimum weights of the network are estimated efficiently in a recursive fashion. The algorithm is especially suitable for online learning situations, where weights are adjusted in a continuous fashion.

With DEKF it should be possible for a RNN to learn optimal weights for many difficult problems. However, RNNs in general [1,7,9] are hampered by *vanishing gradients* [5] that make networks unable to deal correctly with long-term dependencies. A recent novel RNN called *Long Short-Term Memory* (LSTM) [6] overcomes this problem and learns previously unlearnable solutions to numerous tasks [6,2,3], including tasks that require storing relevant events for more than 1000 subsequent discrete time steps without the help of any short training sequences.

In this study we use LSTM with forget gates [2] to predict subsequent symbols of a continual input stream (not segmented a priori into subsequences with

[⋆] Work supported by the Generalitat Valenciana through grant FPI-99-14-268, by the Spanish Comisión Interministerial de Ciencia y Tecnología through grant TIC2000-1599-C02-02, and by the Swiss National Foundation through grant 2100-49'144.96.

clearly defined ends) with long-term dependencies. Thus, unlike previous approaches with LSTM, the focus is on true *online* processing.

Gers et al. [2] studied a similar problem using a different setup. In their simulations, weight updates were performed after each new symbol presentation, a strategy from online learning. However, when network error became too high, a reset was done (clearing activations, states and partial derivatives) and the symbol stream was started over, a heuristic that presents information to the network in a way similar to batch learning. Thus, the previous attempt can be considered as half-way between online and offline learning. Here we apply the same LSTM architecture to the same kind of sequences, but with a *pure* online approach: there is only one single input stream; learning continues even when the network makes mistakes; and training and testing are not divided into separate phases.

All previous LSTM implementations have used a form [6] of gradient descent to adjust the weights of the network. In this paper we apply the DEKF training algorithm to the LSTM architecture for the first time; we compare experimental results obtained with the gradient descent algorithm to those of DEKF, and also comment on much worse results obtained with traditional RNNs.

2 LSTM Networks Trained by DEKF

Gradient descent algorithms, such as the original LSTM training algorithm, are usually slow when applied to time series because they depend on *instantaneous* estimations of the gradient: the derivatives of the error function with respect to the weights to be adjusted only take into account the distance between the current output and the corresponding target, using no history information for weight updating.

DEKF [8,4] overcomes this limitation. It considers training as an optimal filtering problem, recursively and efficiently computing a solution to the least-squares problem. At any given time step, all the information supplied to the network up until now is used, including all derivatives computed since the first iteration of the learning process. However, computation is done such that only the results from the previous step need to be stored.

Lack of space prohibits a complete description of DEKF; we refer the reader to previous citations for details. The extended Kalman filter is used for training neural networks (recurrent or not) by assuming that the optimum setting of the weights is stationary. However, when considering all the weights of the network together, the resulting matrices become so unmanageable (even for networks with moderate sizes) that a *node-decoupled* version of the algorithm is usually used instead to make the problem computationally tractable. The decoupled approach applies the extended Kalman filter independently to each neuron in order to estimate the optimum weights feeding it. By proceeding this way, only local interdependences are considered. The equations for iteration t of a DEKF minimizing the typical quadratic error measure can be formulated as follows:

$$\boldsymbol{G}_i(t) = \boldsymbol{K}_i(t-1)\boldsymbol{C}_i^T(t) \left[\sum_{i=1}^{n_g} \boldsymbol{C}_i(t)\boldsymbol{K}_i(t-1)\boldsymbol{C}_i^T(t) + \boldsymbol{R}(t) \right]^{-1} \tag{1}$$

$$\boldsymbol{w}_i(t) = \boldsymbol{w}_i(t-1) + \boldsymbol{G}_i(t)\left[\boldsymbol{d}(t) - \boldsymbol{y}(t)\right] \tag{2}$$

$$\boldsymbol{K}_i(t) = \boldsymbol{K}_i(t-1) - \boldsymbol{G}_i(t)\boldsymbol{C}_i(t)\boldsymbol{K}_i(t-1) + \boldsymbol{Q}_i(t) \tag{3}$$

where n_g is the number of neurons, i defines a particular neuron (with $1 \leq i \leq n_g$), $\boldsymbol{w}_i$ is a vector with all the weights leading to neuron i, $\boldsymbol{d}(t)$ is the desired response, and $\boldsymbol{y}(t)$ is the actual output of the network.

Let n_i denote the number of weights leading to neuron i, and n_Y the number of output neurons of the network. The Jacobian $\boldsymbol{C}_i(t)$ is an $n_Y \times n_i$ matrix containing the partial derivatives of the function defining the output $\boldsymbol{y}(t)$ of the network with respect to each weight leading to neuron i. Matrices $\boldsymbol{G}_i$, $\boldsymbol{K}_i$, $\boldsymbol{Q}_i$ and $\boldsymbol{R}$ are initialized in a problem-specific manner and denote, respectively, the Kalman *gain*, the *error* covariance matrix, the covariance matrix of *artificial process noise*, and the covariance matrix of the *measurement noise*.

Combining DEKF with the LSTM architecture is straightforward. We consider a group of weights for each neuron in LSTM, that is, a group for each different gate, cell and output neuron (see previous references on LSTM for a detailed description of the architecture and how error derivatives are computed). At time step t we calculate the derivatives required for matrix $\boldsymbol{C}_i(t)$ exactly the same way as the original LSTM gradient-descent training algorithm does, and then apply equations (1)–(3) in order to update weights $\boldsymbol{w}_i(t)$.

It should be noted that DEKF's time complexity [4, p. 771] is much larger than that of gradient descent because DEKF not only has to compute the same derivatives, but also many matrix operations at every time step.

3 Experiments

We use LSTM with forget gates to predict subsequent symbols in a sequence generated by the continual embedded Reber automaton (or grammar) [10] shown in Fig. 1. Due to existence of long-term dependencies, this task is suitably difficult to show the power of both LSTM and DEKF. The learning process is completely online. The network is trained to give in real-time an output as correct as possible for the input supplied at each time step; after normalization, $y_i(t)$ is interpreted as the probability of the next symbol being the i-th symbol of the alphabet; symbols, when considered as inputs or targets, are coded with unary vectors by means of *local* or *exclusive* coding.

Network Topology and Parameters. The LSTM network has 4 memory blocks with 2 cells each. The size of the alphabet of the automaton is 7, so we consider 7 neurons in the input and output layers. Bias weights to input and

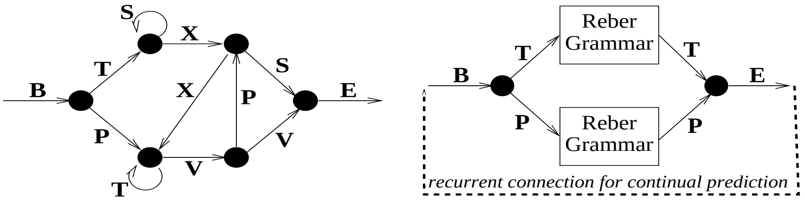

Fig. 1. Transition diagrams for standard (left) and embedded (right) Reber grammars

output gates are initialized block-wise: -0.5 for the first block, -1 for the second, -1.5 for the third, and so on. Forget gate biases are initialized with symmetric values: 0.5 for the first block, 1 for the second, and so on. The rest of the weights are randomly taken from a uniform distribution in $[-0.2, 0.2]$. The squashing function g is set to $\tanh(x)$, and h is set to the identity function — see [2,6] for details on the LSTM architecture. For the gradient-descent algorithm we set the learning rate to 0.5. In case of DEKF, the parameters of the algorithm suggested by Haykin [4, p. 771] turned out to be adequate for this task as well.

Training and Testing. We count the number of symbols needed by LSTM to attain error-free predictions for at least 1000 subsequent symbols (a large period of time); here "error-free" means that the symbol corresponding to the winner neuron in the network output is one of the possible transition symbols, given the current state of the Reber automaton.

Gers et al. [2] considered longer error-free sequences, but learning was not truly online, and the networks were tested with frozen weights. Therefore, although the criterion for sustainable prediction was stringent, the learning was easier in principle. On the other hand, when working online, the recurring presence of particular subsequences usually makes the network forget past history and trust more recent observations instead. This is what one would expect from an online model, which is supposed to deal with non-stationary environments.

After an initial training period, LSTM usually makes only few mistakes and tends to keep making correct predictions. To obtain a tolerant measure of prediction quality we measure the time at which the N-th error takes place after the first 1000 subsequent error-free predictions: here we consider two possible values for N, namely, 1 and 10.

4 Results and Analysis

Gradient-Descent LSTM Results. Table 1 shows the results for 9 different sequences with 9 independently initialized LSTM networks trained by the original LSTM training algorithm.[1] In one case (row 5) no correct prediction

[1] The average number of symbols required for learning to predict accurately in real-time (thousands of symbols) is much smaller than the number of symbols required in the offline set-up (millions). This deserves a more profound study.

Table 1. Time steps required by online LSTM (trained with gradient descent) to achieve 1000 subsequent correct predictions

Net	Sustainable prediction	Next 10 errors	Next error
1	39229	178229	143563
2	102812	144846	111442
3	53730	141801	104163
4	54565	75666	58936
5	1000000^+	–	–
6	111483	136038	113715
7	197748	235387	199445
8	54629	123595	123565
9	85707	92312	86742

Table 2. Time steps required by online LSTM (trained by DEKF) to achieve 1000 subsequent correct predictions

Net	Sustainable prediction	Next 10 errors	Next error
1	29304	30953	30347
2	19758	322980	25488
3	20487	24106	22235
4	26175	33253	27542
5	18015	22241	19365
6	16667	1000000^+	29826
7	23277	26664	24796
8	1000000^+	–	–
9	29742	594117	31535

sequences (for 1000 symbols in a row) are found before the 1000000-th sequence symbol; this is indicated in the table by 1000000^+.

LSTM with DEKF Results. With the DEKF training algorithm, the number of symbols needed for correct prediction is even lower — compare Table 2. Although the time required to achieve 1000 error-free predictions in a row is generally lower than with the original algorithm, the number of symbols before the 10-th error is also smaller. DEKF seems to reduce the long-term memory capabilities of LSTM while increasing its online learning speed. There are 3 remarkable cases (rows 2, 6 and 9 in Table 2), however, where a very long subsequence is necessary for the 10-th error to appear; row 6 shows a particularly good result: only 3 errors occur before the 1000000-th symbol.

LSTM Analysis. Study of the evolution of gate and state activations revealed that online LSTM learns a behavior similar to the one observed in previous non-online experiments [2], that is, one memory block specializes in bridging long-time information, while the others exhibit short-term behavior only.

RTRL-RNNs Results. Experiments with traditional RTRL-trained [11] RNNs (such as the simple recurrent net [1] or the recurrent error propagation network [9]) demonstrated that they are unable to obtain sustainable error-free predictions for 1000 subsequent symbols, even after extremely long training times. Even as few as 100 subsequent correct predictions were extremely rare. DEKF applied to these architectures, however, did allow for sustainable error-free predictions. But it always required many more than 100000 symbols.

5 Conclusion

LSTM variants are applicable to true online learning situations with never-ending continual input streams. On the difficult extended Reber automaton, online LSTM yields results comparable to that of previous LSTM applications involving offline learning, and clearly outperforms traditional RNNs.

For the first time the DEKF algorithm was applied to LSTM. This led to results even better than those obtained with the original training algorithm. The DEKF-based approach reduces significantly the number of training steps necessary for error-free prediction when compared to the standard algorithm. However, it forgets more easily than original LSTM, and requires more operations per time step and weight.

References

1. Elman, J. L.: Finding structure in time. Cognitive Science **14** (1990) 179–211.
2. Gers, F. A., Schmidhuber, J., Cummins, F.: Learning to forget: continual prediction with LSTM. Neural Computation **12**, 10 (2000) 2451–2471.
3. Gers, F. A., Schmidhuber, J.: LSTM recurrent networks learn simple context free and context sensitive languages. IEEE Transactions on Neural Networks **12**, 6 (2001) 1333–1340.
4. Haykin, S.: Neural networks: a comprehensive foundation. Prentice-Hall (1999).
5. Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J.: Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. Kremer, S. C., Kolen, J. F. (eds.): A field guide to dynamical recurrent neural networks (2001). IEEE Press.
6. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Computation **9**, 8 (1997) 1735–1780.
7. Pearlmutter, B. A.: Gradient calculations for dynamic recurrent neural networks: a survey. IEEE Transactions on Neural Networks **6**, 5 (1995) 1212–1228.
8. Puskorius, G. V., Feldkamp, L. A.: Neurocontrol of nonlinear dynamical systems with Kalman filter trained recurrent networks. IEEE Transactions on Neural Networks **5**, 2 (1994) 279–297.
9. Robinson, A. J., Fallside, F.: A recurrent error propagation speech recognition system. Computer Speech and Language **5** (1991) 259–274.
10. Smith, A. W., Zipser, D.: Learning sequential structures with the real-time recurrent learning algorithm. Intl. Journal of Neural Systems **1**, 2 (1989) 125–131.
11. Williams, R. J., Zipser, D.: A learning algorithm for continually training recurrent neural networks. Neural Computation **1** (1989) 270–280.