Incremental Construction of LSTM Recurrent Neural Network

Evandsa Sabrine Lopes-Lima Ribeiro
Universitat Politècnica de Catalunya
Departament de Llenguatges e Sistemes Informàtics
Programa d'Intelligència Artificial
Jordi Girona Salgado, 1-3, C6-201,
08034, Barcelona, España.
e-mail: eslopes@lsi.upc.es

December 2, 2002

Advisor:
Dr. René Alquézar Mancho.

Technical Report UPC-LSI Juny, 2002

Preface

In this work, the outcomes of my research carried out in the field of Recurrent Neural Networks (RNNs) are presented. In particular, it has been focused on the Long Short-Term Memory (LSTM) RNN architecture and its application to signal forecasting tasks.

The investigation accomplished is summarized in two different works. The former is an article that has been accepted in the Third International NAISO Symposium on Engineering of Intelligent Systems (EIS'2002) to be held in Malaga next September, whereas the latter is a Technical Report to be published shortly at the LSI Department of UPC.

In the first work, LSTM is applied to the prediction of maximum ozone concentrations from meteorological data (temperature, wind, cloud covering) and previous ozone concentrations. The obtained results are compared to those achieved by other techniques (including other types of neural networks) on the same data. This paper has been attached in Appendix **??**.

In the second work, the incremental construction of the LSTM network is investigated and some different options are identified for growing the network. These have been applied to another signal forecasting task concerning the identification of models for the Central Nervous System Control. The results obtained by the growing LSTM have been shown to be better than those provided by the original LSTM (without the incremental construction) on the same problem. This work is indeed a preliminary step towards the definition of my Ph.D.thesis project, that will deepen in the incremental construction of RNNs for forecasting tasks, maybe combining LSTM units (memory blocks) with other RNN architectures or even with Time-Delay Neural Networks.

Abstract

Long Short–Term Memory (LSTM) is a recurrent neural network that uses structures called memory blocks to allow the net remember significant events distant in the past input sequence in order to solve long time lag tasks, where other RNN approaches fail.

Throughout this work we have performed experiments using LSTM networks extended with growing abilities, which we call GLSTM. Four methods of training growing LSTM has been compared. These methods include cascade and fully connected hidden layers as well as two different levels of freezing previous weights in the cascade case. GLSTM has been applied to a forecasting problem in a biomedical domain, where the input/output behavior of five controllers of the Central Nervous System control has to be modelled. We have compared growing LSTM results against other neural networks approaches, and our work applying conventional LSTM to the task at hand.

Contents

List of Figures

List of Tables

1 Introduction

In this work, the application of four growing methods are studied to improve the learning of a recurrent neural network (RNN) called Long Short–Term Memory (LSTM) in signal prediction tasks.

Recurrent neural networks are an interesting class of models which preserves information through time. Their recurrent links bring past information back to the network itself so that the state at time t depends on both the current input and its previous state at time $t-1$. This kind of network is a better candidate for dynamical problems in comparison with the static feedforward networks (FNNs).

LSTM [Hoc95, GS00a] is a recurrent network that uses structures called memory blocks to allow the net remember significant events distant in the past input sequence in order to solve long time lag tasks, where other RNN approaches fail. LSTM have been quite successfully applied to standard benchmarks related to classification problems [GS00a, GS00b, HS97b, HS96, Hoc95], and more recently to signal forecasting problems [RA01, GS00a].

A key issue in neural network (NN) design is determining the number of hidden units required to perform input/output mapping with satisfactory performance. In recent years, attempts have been made to build neural networks incrementally in an automatic way. The techniques used to solve this problem are called constructive or growing methods.

Growing methods are designed to automate the process of determining the network topology, by modifying both the weights and connectivity of the network during learning. Thus, these methods eliminate the need to guess the network size what makes the use of NNs more user–friendly.

Throughout this work we have performed experiments using LSTM networks extended with growing abilities, which we call GLSTM. Four methods of training growing LSTM has been compared. GLSTM has been applied to a forecasting problem in a biomedical domain, where the input/output behavior of five controllers of the Central Nervous System (CNS) control has to be modelled. We have compared growing LSTM results against those reported in [JC97, BVA98] using other NN approaches and our previously work applying conventional LSTM [RA01] to the task at hand.

In the following section we discuss about the most important dynamic input/output approaches in the field of neural networks. Subsequently, in section 3 we give an introduction to growing methods for NNs. In section 4 we describe the LSTM architecture. In section 5 we introduce the growing LSTM approaches. In section 6 we show the case of study. In section 7 the experimental methodology used is described. In section 8 the obtained results are presented. Finally, some conclusions are given in section 9.

2 Dynamic Input/Output Neural Networks

Dynamic input/output NNs provides de possibility to process time-varying information, i.e., the network can learn dynamic input-output mapping instead of static mapping as in feedforward NNs.

The following subsections summarize the key ideas of the most popular NN architectures dedicated to process time-varying data.

2.1 Time-Delay Neural Networks

The non recurrent architectures commonly used in prediction and temporal association tasks are named Time-Delay Neural Networks (TDNNs) and were first applied by [WHH+89, KL90].

The memory in a TDNN is implemented as a so called tapped delay line. A tapped delay line is a sequence of elements that represent several consecutive values of the same unit, each with a different delay in time steps. They are also called delay elements and are often denoted by z^{-1}, which is the transfer function of such an element in the z-space.

Hence, an TDNN can be thought as a Multi-layer perceptron (MLP) in which each input unit is replaced by a tapped delay line that stores past values. In other words, the output of a layer is buffered several time steps and then forward propagated to the next layer (see Figure 1).

The main disadvantage of this architecture is the limited past history horizon thereby preventing modelling of arbitrary long time dependencies between inputs and desired outputs. It is also difficult to set the number of delay elements which must be chosen in advance and may not represent the existing time correlation in the sequence.

Therefore, TDNNs could result inappropriate if arbitrarily large sequences having important events distant in the past sequence are used; on the other hand, if the window length required is known but large, then the big number of inputs units (and their respective weights) can demand a great number of training examples to reach an optimal generalization performance.

TDNNs, as any feedforward net, can be trained by a variety of methods of which standard backpropagation is the most usually used. A more sophisticated and powerful training algorithm (that is referred to as TDNN-AC) is given by coupling a conjugate gradient optimization method with an annealing scheme that allows the exploration of several local minima during the training phase. Another, completely different TDNN model, is the Time-delay Heterogenous Neural Network (TD-HNN) [BVA98] which is trained by a genetic algorithm.

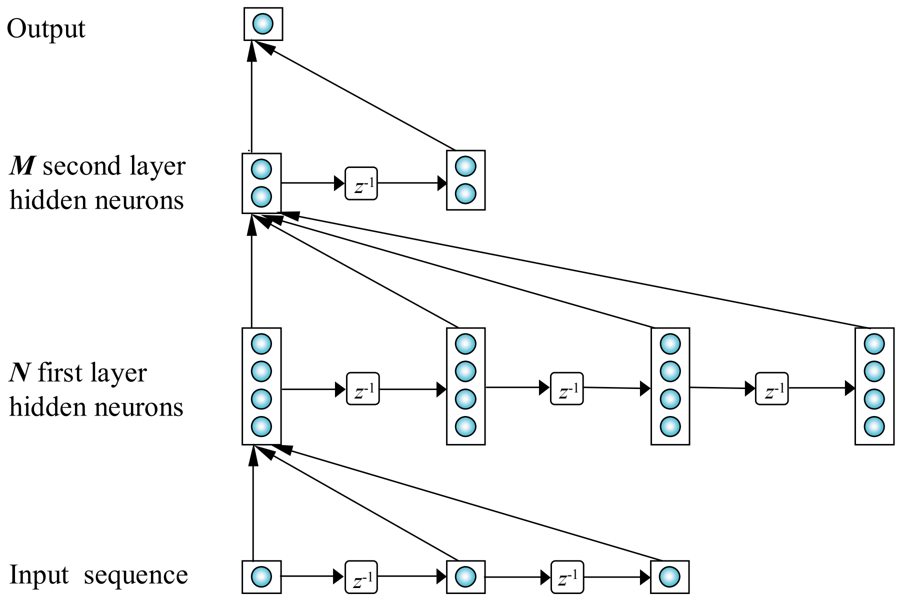

Output
M second layer hidden neurons
N first layer hidden neurons
Input sequence
z^{-1}

Figure 1: TDNN representation: All output activations in a given layer are buffered several time steps and then propagated to the next layer.

2.2 Recurrent Neural Networks

In the last fifteen years, Recurrent Neural Networks (RNNs) have been widely studied. The motivation to explore this kind of architecture is its capacity to learn from examples how to process spatio-temporal data (i.e., signals and sequences that has a spatial or temporal dimension) in three basic ways [HKP91]:

- *Sequence Recognition and Classification*: the net produces a particular output pattern once the whole input sequence is seen.
- *Sequence Reproduction and Prediction*: the net can generate the rest of a sequence when it sees part of it.
- *Temporal Association*: the net will produce an output sequence in response to a specific input sequence.

It is necessary to distinguish the recurrent neural nets that can solve these tasks, which do not have any weight restriction, from those such as Hopfield nets and other recurrent architectures with symmetric weights, whose dynamic converges to a fixed point and hardly deal with these three tasks mentioned [HKP91].

Recurrent Neural Networks can be classified into three main classes: partially, fully and locally RNNs, which are discussed in the following subsections.

2.2.1 Partially Recurrent Neural Networks

Partially recurrent neural networks [HKP91] are mainly feedforward nets that include feedback connections to a set of units called context units. A context unit is basically an input unit that receives information from the network units activations at the previous time step [Elm90] and them feed it in the next time step. The context units do not perform any signal computation, only act as memories.

In these models, the recurrent connections to context layer use fixed weights (set to 1.0), which are not adapted during learning. So, the recurrence lets the network remember signals from the past, but does not appreciably complicate the training [HKP91]. Hence, this kind of RNNs is still being trained by conventional back-propagation methods (as multi-layer perceptron) and, thus, does not include recurrence terms in the learning rule.

Examples of these models are Jordan [Jor86] and Elman [Elm90] recurrent neural networks (the latter also called Simple Recurrent Network or SRN). These models are able to learn simple tasks that only need very short–term memory, since their training algorithm does not accomplish a real gradient descent calculation with respect to the weights, but rather makes only an approximation, which is based on not considering the terms originated by the influence of the weights in previous time steps (*truncated gradient*).

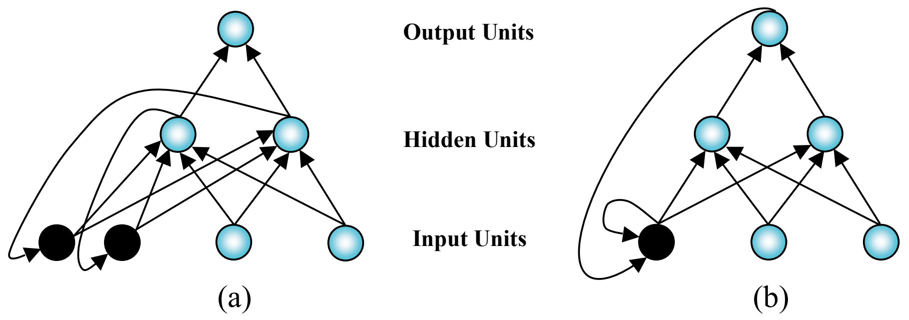


Figure 2: Elman (a) and Jordan (b) recurrent neural networks.

Figure 2 illustrates Elman and Jordan architectures. As it can be seen, in the former NN the recurrence is restricted from the hidden layer to context layer, whereas in the latter the context layer has a self–recurrent connection in addition to the feedback connection, in this case, from output layer.

Another partially RNNs that have shown great results are the NARX networks [LHG96, HG95]. These networks, as the Jordan model have limited feedback connections which come only from the output units, but they also include time delays for inputs and outputs.

2.2.2 Fully Recurrent Neural Networks

To improve the behavior of partially recurrent nets, fully recurrent neural networks, also called Single Layer Recurrent Neural Networks (SLRNNs), together with gradient-descent based algorithms that correctly calculate the real gradient of the error were proposed.

Fully RNNs make use of global recurrences and in that way do not impose any conditions on the way the units of a net are connected. Thus, more general networks are allowed, where every unit might be connected to all other units, including or not itself. The network has an input layer consisting of some external inputs (the network inputs) and feedback activations of all the units, together with a single processing-layer of computation units (some of them are output units with target values at some time steps). These networks offer the highest computation power. It can be shown, that such a network can model any non-linear dynamical system with arbitrary accuracy [FN93, LNG95].

The two main gradient-based learning approaches for SLRNNs are Real-Time Recurrent Learning (RTRL) [WZ89] and Back-Propagation Through Time (BPTT) [WP90]. BPTT algorithm is an extension of the standard back-propagation algorithm. It may be derived by unfolding the temporal operation of the network at the time steps $1, 2, \ldots, T$ into a layered feedforward network with T stages of computation, the topology grows by one layer at each time step. Figure 3 illustrates the diagram of the unfolded network.

While BPTT uses the backward propagation of error information to compute the error gradient, an alternative approach is to propagate gradient information forward. This leads to the RTRL learning algorithm which derives its name from the fact that adjustments are made to the synaptic weights of a fully connected recurrent network on-line, that is, while the networks continues to perform its signal processing function.

The above methods were the firstly studied for general fully RNNs and more recently other algorithms have been presented. We can emphasize among them the Time-Block RTRL algorithm [Sch92a], which calculates the gradient by updating the weight influence forward at each p time steps and back propagating the error at each block of p steps.

If n is the number of units in a SLRNN, the time complexity per time step of these methods (BPTT, RTRL and Time-Block RTRL) is $O(n^2)$, $O(n^4)$ and

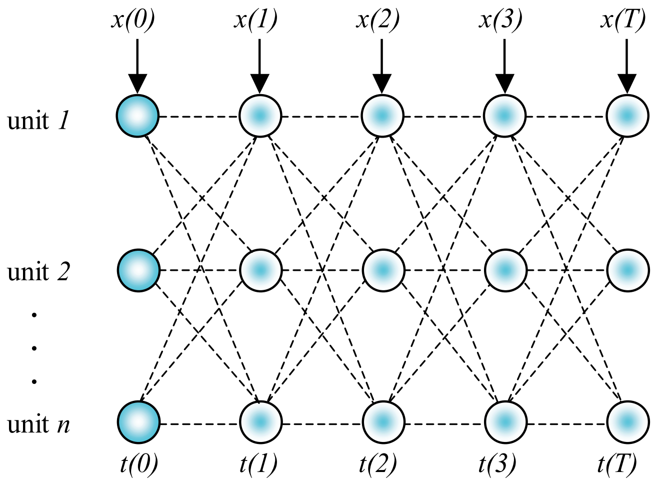


Figure 3: Back-Propagation through time. Each connection in the network is assumed to have a delay of one time step.

$O(n^3)$, respectively, where the first (BPTT) needs a space proportional to the length of the longest sequence in the training set.

The SLRNNs are classified according to their units' connectivity (i.e., according to how the weights are combined with external inputs and feedback activations), which can be of first– [WP90] or second–order [GMC+92]. The second order SLRNN has a greater power of representation due to the fact that they can represent any finite state machine whereas the first cannot [GGC94, AS95]. However, it is possible to increase a first order SLRNN by adding a feedforward output layer (or even several layers, where the last one is an output layer) so that the resulting architecture, denominated first-order Augmented Single Layer RNN (ASLRNN), has the same capacity of representation and learning as a second order SLRNN [AS95, Alq97]. The ASLRNNs can be trained by using back-propagation for the feedforward layer combined with any of the three algorithms mentioned previously for the recurrent layer [Alq97].

Other discrete-time RNNs have been proposed in the literature, such as the Recurrent Cascade Correlation network [Fah91], the DOLCE architecture [DM93] and the Manolios-Fanelli network [MF94]. An interesting review of different RNN architecture can be found in [Cn98].

2.2.3 Locally Recurrent Neural Networks

A network is called locally recurrent, when no feedback connections between different units exist. This condition causes the overall behavior of the network to be feed forward and makes the leaning procedure less complex.

A typical example of a locally recurrent neural network is the focused MLP [Moz89, FGS92] which is showed in Figure 4. It is called focused because there are only self feedback loops. This avoids the backpropagated error to disperse over several units, but keeps it ”focused” on a single unit.

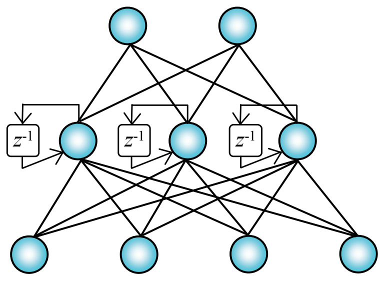


Figure 4: Focused MLP with self feed back at all hidden units.

Feedbacks are necessary when a long and complex temporal dynamics is required. Fully recurrent networks are general but difficult to train [BT94]. An alternative approach able to process temporal signals is to use units with local temporal memory and processing capabilities. This kind of units is used mainly in feedforward NN architectures, since all time processing is concentrated in the neurons rather than in the recurrent feedback links.

Lapedes and Farber [LF87] introduced a new neuron model, where the synapses with constant weights were substituted by linear transference function. Their synapses functions are basic FIR (Finite Impulse Response) like filters, where all poles are located in the origin. In fact, this model has no feedback links, so it is still a pure feedforward NN, however it was the basis for future structures.

Back and Tsoi [BT91] generalized the Lapedes and Faber structure and introduced a new architecture called Infinite Impulse Response MLP (IIR-MLP). IIR-MLP can be considered as a nonlinear extension of the linear adaptive IIR filter. The FIR and IIR-MLP networks are similar to the standard MLP except each synapse is replaced by FIR and IIR2 filters respectively. IIR-MLP can exhibit better capabilities, due to the prewired forgetting behaviour (typical of locally recurrent networks) [CUP97] especially for digital signal processing problems, for which, in the case of stability, a forgetting behaviour [FGS92] is usually required. Thus, Tsoi and Back [BT94] called this architectures Locally Recurrent Globally Feedforward (LRGF) neural networks. The LRGF net is a generalization and unification of many kinds of NN models that were developed independently.

Vries and Principe [VP92] proposed an architecture named Gamma Model which is obtained by replacing the weights of a standard MLP with Gamma filters. This is accomplished in a way similar to FIR and IIR-MLPs. The gamma memory contains as special cases the context unit [Jor86] and the tap delay line as used in TDNN [WHH$^+$89]. However, the gamma memory is also a special case of the generalized feedforward filters where which leads to the gamma functions as the tap signals.

The local output feedback was also studied by many authors such as Poddar and Unnikrishnan [PU91], Gori et al. [GBM89], Tsoi and Back [BT94]. For information about locally recurrent NNs taxonomy see [BT94, NRRP$^+$93, Moz93].

2.3 Summary

Even though RNNs are fascinating from a theoretical point of view and can be applied to several interesting problems (speech recognition, language translation, time series and dynamic systems prediction, musical composition, non linear control, signal processing and compression), in practice they have some drawbacks, where the most important one is the vanishing gradient problem.

In principle, and differently from time-delay nets, the RNNs should be able to capture long-term dependencies (i.e., decisions that depends on events that occurred in the distant past). However, it has been shown that this does not happen as it was expected. The reasons were theoretically analyzed concluding that the error signal *flowing backwards in time* in gradient-based algorithms (e.g., BPTT and RTRL [Pea95]) tend to either, blow up or vanish [HS97a, HS97b]. In the first case, that can happen, for example, if a linear activation function is used [AJ94], the weights oscillate and its magnitude grows exponentially leading to the net instability and error overflowing. In the second case, that is more usual, if a sigmoid activation function is used, for example, the gradient magnitude decreases exponentially in time, preventing the net from learning long-term dependencies and from reaching the optimal task performance.

To resolve the vanishing gradient problem different alternatives have been proposed. Mozer introduced time constants that influence the changes in the units' activation; however these constants should be adjusted carefully to capture the long-term dependencies [Moz92]. Schmidhuber has proposed hierarchical recurrent nets that slice and compress an input sequence adaptively, but they can only connect big jumps of time if the intermediate subsequence is predictable in a local way (in time), and in addition, they are also too sensitive to noise [Sch92b]. Lin et al. [LHG96] defined the NARX nets, that is an extension of time-delay nets where some output delays are fed back; this net, if trained with BPTT, allows to soothe the vanishing gradient problem by providing short cuts for propagating the error backwards [LHG96, SHG97].

More recently, Hochereiter and Schmidhuber presented a recurrent architecture, denominated Long Short–Term Memory (LSTM), designed to overcome error back-flow problems, as vanishing gradient [HS97a, HS97b]. This architecture, within its gradient descent training algorithm, facilitates a constant error flowing in time by using special activation units. According to [Hoc95] LSTM has the same update complexity per time step as BPTT, i.e., $O(n^2)$.

We are interested in the LSTM recurrent net for several reasons; first of all because of its recurrent nature and its particular features, that, in principle, adapts better to a signal forecasting problem, second because LSTM has been shown to outperform some RNNs architecture when applied to long-term tasks [HS97b] and finally, due to the fact that this architecture is rather recent and still not totally explored. In §4 we describe the LSTM architecture in more detail.

3 Neural Networks Growing Methods

The need to fix the size and topology of a network before its training phase is one the most important practical problems when using common NN approaches. Therefore, the use of incremental growing methods that adapt the net parameters automatically are seen as a practical way of making the use and configuration of neural networks less complex.

To that end, growing methods attempt to reach a suitable network topology by starting the training phase with a very small network which will be adapted, according to the task at hand, by adding units during the training process aiming to obtain an optimal performance.

In the latest years a great deal of effort has been directed towards finding efficient growing algorithms for determining the weights and topology of a neural network [RM99, KY97, Smi93]. In the following subsections we review the main growing methods used in feedforward and recurrent nets.

3.1 Feedforward Neural Networks

3.1.1 Dynamic Node Creation

The Dynamic Node Creation (DNC) method [Ash89] was developed to add nodes to the hidden layer of the network during training. After a new node is added with this procedure, the whole network is trained with standard backpropagation algorithm until the desired mapping is learned or another unit needs to be added. A new hidden unit is added when the average error curve begins to flatten out too quickly. The new unit receives complete connections from the inputs, and is connected to all outputs, and its weights are initialized with small random values. A more sophisticated method in order to train the network that uses sophisticated nonlinear least squares and quasi-Newton optimization techniques can be found in [Bel94].

3.1.2 Projection Pursuit

Projection Pursuit (PP) is a family of optimization methods that appeared in the statistics literature. Its name is derived from the fact that the data are projected onto several interesting directions, which are selected to maximize a certain objective function. This notion of interesting projections is motivated by an observation that for most high-dimensional data clouds, most low-dimensional projections are approximately normal [DF84]. Hence, a projection is less interesting the more nearly normal it is. In general, given a random variable X, the methods based in PP search for a linear projection A optimizing an objective function $Q(F_A)$, where F_A is the distribution of the random variable $A \cdot X$. By changing the objective function $Q(F_A)$, the particular PP methods are obtained.

PP generalizes classical methods such as principal components and discriminant analysis. As a drawback, they use to be high-demanding on computation time. Due to this computational cost, and to the interest in getting an ordered set of projections, the stepwise methods are very attractive. As particular case of function approximation, Projection Pursuit Regression (PPR) [FS81] estimates the conditional expectation of a random variable $Y \in \mathbb{R}$ given $X \in \mathbb{R}^I$ by means of a sum of ridge functions

$$E[Y|X=\vec{x}] = f(\vec{x}) \cong \sum_{j=1}^{N} g_j(\vec{a_j}^{\,t} \cdot \vec{x})$$

as follows (the $\vec{a}_j$'s act as the frequencies). Suppose that the first $n-1$ terms of the approximation have been determined. That is, the vectors $\vec{a}_j$ and the functions g_j, $1 \leqslant j \leqslant n-1$ have been calculated. Let

$$r_{n-1}(\vec{x}) = f(\vec{x}) - f_{n-1}(\vec{x}) = f(\vec{x}) - \sum_{j=1}^{n-1} g_j(\vec{a_j}^{\,t} \cdot \vec{x})$$

be the residue at step $n-1$. Find $\vec{a}_n$ and g_n such that $\|r_{n-1}(\vec{x}) - g_n(\vec{a_n}^{\,t} \cdot \vec{x})\|$ is the minimum. This process is repeated until the residue is smaller than a user-defined threshold.

An NN that implements PPR is the Projection Pursuit Learning Network (PPLN). An PPLN is modelled as a one hidden layer MLP that learns neuron by neuron, and layer by layer cyclically after all the training patters are presented [HRM$^+$94]. An incremental neural network algorithm that is very similar to PPR is the Incremental linear quasiparallel algorithm, presented in [KB95].

Several methods with the same underlying ideas than PP have appeared in the area of Signal Processing. Among then we can highlight the Matching Pursuit algorithm, which is described in [MZ93] as an algorithm that decomposes any signal into a linear expansion of waveforms that are selected from a redundant dictionary of functions.

3.1.3 Cascade Correlation

In the field of Neural Networks the most used growing method is the Cascade Correlation (CC) [Fah90]. CC combines two key ideas. The former is the cascade architecture, in which hidden units are added only one at a time. The newly added hidden units receives inputs from the input layer as well as from the previously added hidden units. The latter is the learning algorithm, which create and installs the new hidden units.

Initially, the network contains only inputs, output units, and the connections between them. The single layer of connections is trained using the Quickprop algorithm [Fah91] (a version of back-propagation) to minimize the error of the training set. When the level of the error stops decreasing, the performance of the network is evaluated. If the performance is good enough, the learning ends. Otherwise, a new hidden unit is added to the network in an attempt to reduce the residual error.

Before adding a new hidden unit, a pool of candidate units is tested, such that each of these units receives weighted connections form the network's inputs and from any hidden units already present in the net, but their output activations are not yet connected to the output units. Then, weights of each candidate unit are adjusted to maximize the correlation between the unit's output and the residual error signal of the network. When the correlation scores stop improving, the candidate unit with the best correlation is selected as the new hidden unit, the weights associated with its incoming connections are frozen, and only the weights of the output units are re-trained, including those from the new hidden unit. The process of adding a new hidden unit and re-training the output layer is repeated until the error is small enough. Different versions of CC can be found in [Pre97] and [LR96].

3.1.4 Orthogonal Methods

In [GAP98] a sequential orthogonal approach to the building and training of single hidden layer neural networks is described. When adding a unit, the new information introduced by this unit is caused by that part of its output vector which is orthogonal to space spanned by the output vectors of previously added hidden units. In this context, a vector is an element of $\mathbb{R}^T$, where T is the number of patterns. The Gram-Schmidt method is used to form a set of orthogonal bases for the space spanned by the output vectors of the hidden units. Hidden units weights ω_n are found through optimization (gradient descent) of $\|E_{n-1} - \lambda_n R_n(\omega_n)\|$, where E_{n-1} is the network error with the previously added hidden units. Output layer weights λ_n are obtained from the Least Square (LS) regression. When the training procedure is finished, output layer weights need to be recalculate. Output layer weights are determined through orthogonal LS using the Gram-Schmidt orthogonalization results obtained at each step.

With the same underlying ideas, [CCG91] proposed a learning procedure for RBF Networks based on the Orthogonal Least Squares method in which the use of gradient descent is not necessary, because the frequencies are selected from the dataset. Although the coefficients are recalculated when the training finishes, the frequencies are obtained again approximating the residue at the previous step with only one term (one frequency), exactly the same as in PPR. Other versions of this algorithm can be found in [GY00] and [SK00].

3.1.5 SAOCIF

Sequential Approximating with Optimal Coefficients and Interacting Frequencies (SAOCIF) [RA02], is a sequential method that combines two key ideas. The first one is the optimization of the coefficients (or output layer weights), which provide the linear part of the approximation. The second one is the flexibility to choose the frequencies (or hidden layer weights), which provide the non-linear part.

Concerning the architecture needed to implement SAOCIF, it must present the following characteristics.

- It must be feedforward architecture with a hidden layer of units (including both MLPs with one hidden layer and RBFNs).
- There are no restrictions about the dimension of the input and the output. There will be so many as the target function have. If there

are several outputs, the total inner products must be calculated as the summation of the individual inner products of every output.

- There is no restriction about the biases in the hidden units. The biases can be treated as part of the frequencies.
- The output units cannot have biases.
- There is no restriction about the activation functions in the hidden units. In particular, they can be sines, cosines, sigmoidal functions, gaussian functions, wavelets, etc. Obviously, different units have different activation functions.
- The output units must have a linear activation function.

As can be seen, the restrictions only refer to the output units. The biases are not a real problem, since they can be considered as frequencies with a simple transformation. Hence, the only real restriction in the output units is the linear activation function. An algorithm to construct an approximation based on SAOCIF using FNNs is proposed in [RA02]. Since the frequency goodness does not depend on the norm of its associated vector, the range of weights to look for candidate frequencies may be as large as desired. The strategy to select the candidate frequency is probably the most important part of the algorithm. SAOCIF has been trained using three different strategies. In the first one, the frequencies are selected at random. In the second one (called Input strategy), the frequencies are selected from the points in the dataset (as often in RBFs, but not exclusively) in a deterministic manner: for every hidden unit to be added, every point in the training set is tested as a candidate frequency. The third one is a more sophisticated strategy from the field of Evolutionary Algorithms, where a population of frequencies evolves driven by a Breeder Genetic Algorithm (BGA) with the squared error as the fitness function.

3.2 Recurrent Neural Networks

3.2.1 Recurrent Cascade Correlation

Recurrent Cascade-Correlation (RCC) [Fah91] is an architecture that adds recurrent operation to the Cascade-Correlation architecture. Therefore, some changes were needed in order to make the two models fit together.

As commented before, in the CC architecture new hidden units are added one by one, and are frozen once they are added to the network. Therefore, the insertion of the outputs from new hidden units back into existing hidden units as new inputs would not be appropriate since this concept would certainly be violated. On the other hand, the network must be able to form recurrent loops if it is to retain state for an indefinite time. So, to solve that, in RCC each of the hidden and candidate units is provided with a single weighted self–recurrent link that feeds back its own activation value on the previous time step, as can be seen in Figure 5.

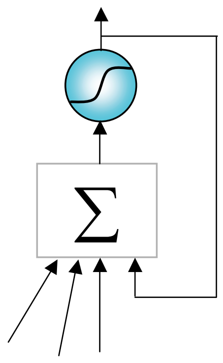


Figure 5: Candidate or hidden unit with a self–recurrent link.

That self-recurrent link is trained along with the unit's other input weights to maximize the correlation of the candidate with the residual error. If the recurrent link adopts a strongly positive value, the unit will function as a flipflop, retaining its previous state unless the other inputs force it to change; if the recurrent link adopts a negative value, the unit will tend to oscillate between positive and negative outputs on each timestep unless the other inputs hold it in place; if the recurrent weight is near zero, then the unit will act as a gate of some kind.

When a candidate unit is added to the active network as a new hidden unit, the self-recurrent weight is frozen, along with all the other weights. Each new hidden unit is in effect a single state variable in a finite-state machine that is built specifically for the task at hand.

3.2.2 Parallel-modular RCC

As an alternative to the original RCC architecture, [KTA95] introduced the Parallel-modular RCC that is trained with natural connectionist glue, which is a concept for modularity and scaling in large phonemic neural networks [AW90, KL90]. This approach aim to provide an improvement to the recognition rates for tasks involving large numbers of features to be learned.

Parallel-modular RCC differs from RCC in the way of structuring the net. That is, to train the cascade–correlation network with large training sets, Fahlman [Fah91] proposed to divide the training set into a series of short "lessons", and train them one after the other, going from the simplest to the most complicated one. Then retrain the network with all the samples in a single training set. The retraining is done in the same way each lesson was trained, that is, keeping the incoming connections of the previously installed hidden units frozen, as well as their respective selfrecurrent links. The groups of hidden units generated during the training of each lesson grow in a cascaded fashion one on top of the other. When the network is finally retrained with the complete training set, one last group of hidden units is created. These groups of hidden units are referred in Parallel-modular RCC as units modules.

By altering the connections of the original RCC, interrupting the cascade and locating every new module parallel to the previous ones, with no connections between the modules, the cascaded RCC is transformed into the Parallel RCC. In that way, each module becomes totally independent from the activation of the others.

This is done in order to concentrate the "knowledge" about a group of patterns in a module, instead of distributing it across the whole network. The modules are connected in parallel, in contrast to the completely cascaded structure of the original RCC.

3.2.3 GNARL

Another more recent effort attempting to learn weights and topology of neural nets is the work of Angeline [ASP94]. The GNARL Algorithm, which stands for GeNeralized Acquisition of Recurrent Links, is an evolutionary algorithm that non-monotonically constructs recurrent networks to solve a given task. The name GNARL reflects the types of networks that arise from a generalized network induction algorithm performing both structural and parametric learning. In that way, GNARL evolves neural networks using structural levels of mutations for topology selection as well as simultaneously evolving the connection weights through mutation.

Thus, instead of having uniform or symmetric topologies, the resulting networks have “gnarled” interconnections of hidden units which reflect constraints inherent in the task. The input and output nodes, as in other neural nets, are considered to be provided by the task and are immutable by the algorithm. The number of hidden nodes varies from 0 to a user-supplied maximum.

4 LSTM Recurrent Neural Network

LSTM [Hoc95, HS97a] belongs to a class of recurrent networks that has time-varying inputs and targets. That is, points in the time series or input sequence are presented to the network one at a time. The network can be asked to predict the next point in the future or classify the sequence or to perform some dynamic input/output association. Error signals will either be generated at each point of the sequence or at the end of the sequence.

4.1 LSTM Structure

A fully connected LSTM architecture is a three-layer neural network composed of an input layer, a hidden layer and an output layer. The hidden layer has a feedback loop to itself, i.e., at time step t of a sequence with n time steps, presented to the network, the hidden layer receives as input the activation values of the input layer and the activation values of the hidden layer at time step $t-1$. Figure 6 illustrates a LSTM with a fully connected hidden layer consisting of two memory blocks, each one consisting of two cells. The LSTM showed has an input dimension of two and an output dimension of one. Only a limited subset of connections are shown.

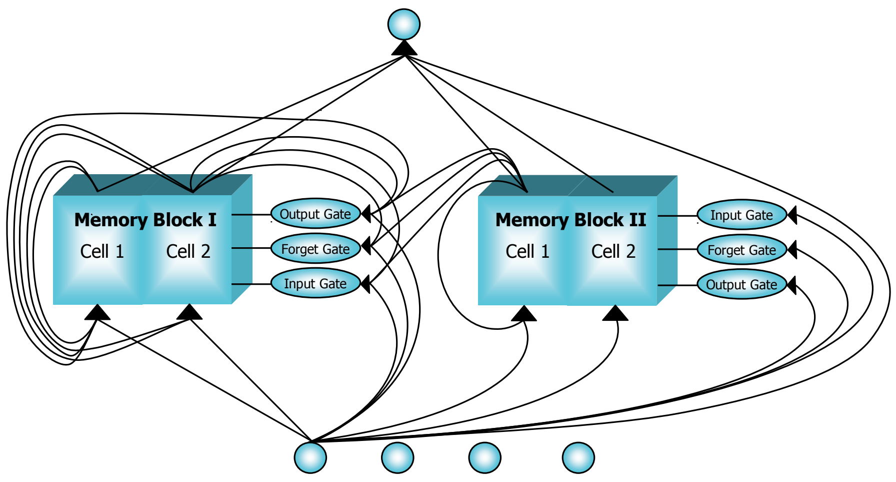


Figure 6: Example of LSTM net consisting of 4 inputs units, 1 output unit and 2 memory blocks of size 2. Only a limited subset of connections are shown.

The basic unit in the hidden layer is known as a memory cell block. A memory cell block (Figure 7) consists of S memory cells and three multiplicative gates, called the input gate, output gate and forget gate. Each memory cell has at its core a recurrently self-connected linear unit called *Constant Error Carousel* (CEC), whose activation is called the cell state. The CECs solve the vanishing error problem: in the absence of a new input or error signals to the cell, the CEC's local error back flow to remains constant, neither growing nor decaying. Input and output gates regulate write and read access to a cell whose state is denoted S_c. The CEC is protected from both flowing activation and backward flowing error by the input and output gates respectively. When gates are closed (activation around zero), irrelevant inputs and noise do not enter the cell, and the cell state does not perturb the remainder of the network. The forget gate feed the self–recurrent connection with its output activation and is responsible for do not allow the internal state values of the cells grow without bound by resetting the internal states S_c as long as it needs. In addition to the self–recurrent connection, the memory cells receive input from input units, other cells and gates.

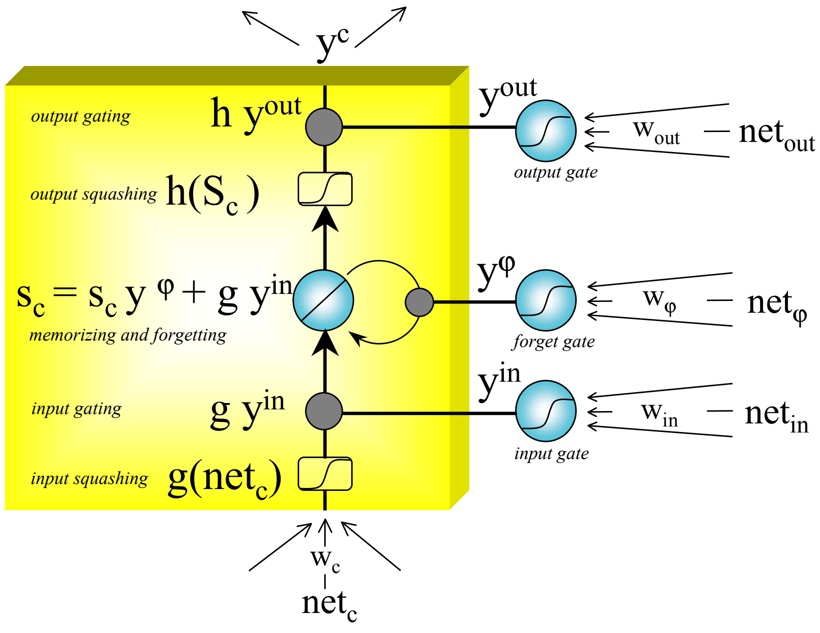


Figure 7: The standard LSTM cell with a recurrent self-connected connection (CEC) and its respective gates.

While the cells are responsible for maintaining information over long periods of time, the responsibility for deciding what information to store, and when to apply that information lies with the input and output gate units, respectively.

A single step involves the update of all units (forward pass) and the computation of error signals for all weights (backward pass). Input gate activation y^{in} and output gate y^{out} are computed as follows:

$$net_{in_j}(t) = \sum_m w_{in_j m} y^m(t-1), \quad y^{in_j}(t) = f_{in_j}(net_{in_j}(t)). \tag{1}$$

$$net_{out_j}(t) = \sum_m w_{out_j m} y^m(t-1), \quad y^{out_j}(t) = f_{out_j}(net_{out_j}(t)). \tag{2}$$

where $y^{in_j}(t)$ denotes the activation of the input gate at time t and $y^{out_j}(t)$ denotes the activation of the output gate at time t.

The w_{jm} is the weight on the connection from unit m to unit j. The summation indices m may stand for input units, memory cells, or even conventional hidden units if there are any. All these different types of units may convey useful information about the current state of the net.

The forget gate activation $y^{\varphi_j}(t)$ is calculated like the other gates above:

$$net_{\varphi_j}(t) = \sum_m w_{\varphi_j m} y^m(t-1), \quad y^{\varphi_j}(t) = f_{\varphi_j}(net_{\varphi_j}(t)). \tag{3}$$

Here, net_{φ_j} is the input from the network to the forget gate. For all gates, the squashed function f is the logistic sigmoid with range [0,1].

The internal state of the memory cell $S_c(t)$ is calculated by adding the squashed, gated to the input to the state at the previous time step $S_{c_j^v}(t-1)$, $(t > 0)$:

$$net_{c_j^v}(t) = \sum_m w_{c_j^v m} y^m(t-1),$$

$$S_{c_j^v}(t) = y^{\varphi j}(t) S_{c_j^v}(t-1) + y^{in_j}(t)\ g(net_{c_j^v}(t)), \tag{4}$$

where j indexes memory blocks, v indexes memory cells in block j, such that c_j^v denotes the v-th cell of the j-th memory block. The cell initial state is given by $S_{c_j^v}(0) = 0$.

The cell's input squashing function g used is a sigmoid function with range $[-1, 1]$. The cell output y^c is calculated by squashing the internal state S_c via the output squashing function h and then multiplying (gating) it by the output gate activation y^{out}.

$$y_{c_j^u}(t) = y^{out_j}(t)\ h(s_{c_j^u}(t)). \tag{5}$$

Here we used the identity function as output squashing function h.

Lastly, assuming a layered network topology with a standard input layer, a hidden layer consisting of memory blocks, and a standard output layer, the equations for the output units k are:

$$net_k(t) = \sum_m w_{km} y^m(t-1), \quad y^k(t) = f_k(net_k(t)), \tag{6}$$

where m ranges over all units feeding the outputs units. As squashing functions f_k we again use the logistic sigmoid, range [0,1].

4.2 Learning

During the learning phase the input gate scales the activation value flowing into the cell before it has the opportunity to change the internal state S_c of the CEC. The activation value of the CEC is worked through a sigmoid step function before it is scaled by the output multiplication gate and is produced as the output signal of the cell. The output gate and the output non-linearity h, first scale an error signal arriving into a cell unit, before it is allowed to flow into the CEC. Error signals trapped within a cell's CEC cannot change - but different error signals flowing into the cell via its output gate may get superimposed. Error signals will be scaled again before they are allowed to run out of the CEC to make an update to the weights coming into the cell. Essentially, the multiplicative gate units open and close access to constant error flow through the CEC.

LSTM's backward pass [HS97b] is basically a fusion of slightly modified truncated back-propagation through time (BPTT) [WP90], which is obtained by truncating the backward propagation of error information, and a customized version of RTRL [RF87] which properly takes into account the altered (sigma-pi-like) dynamics caused by input and output gates (see details in [HS97b]).

Output units use BPTT; output gates use a truncated version of BPTT. However, weights to the cells and forget gates use a truncated version of RTRL. Truncation means that all errors are cut off once they leak out of a memory cell or gate, although they do serve to change the incoming weights. The effect is that the CECs are the only part of the system through which errors can flow forever. So, the error signals flowing out of the CEC and the multiplicative gates are truncated after they are used to update the incoming, weighted connections.

LSTM's learning algorithm is local in space and time; its computational complexity per time step and weight is $O(1)$, that means $O(n^2)$ where n is the number of hidden units if we measure the complexity per time step. This is very efficient in comparison to the RTRL algorithm. The time step complexity is essentially that of BPTT, but unlike BPTT, LSTM only needs to store the derivatives of the CEC's, this is a fixed-size storage requirement independent of the sequence length.

5 Growing LSTM

Aiming to improve the learning abilities of LSTM neural net, in this work deals with a version of LSTM where the network topology is incrementally adapted by the addition of new memory blocks. We call this version Growing LSTM (GLSTM).

In growing algorithms, many heuristics can be used to guide the search in the possible solution space. An important problem is how to set the weight connections of a newly added block. LSTM starts training with just one memory block and grows by inserting blocks, one at time, on the hidden layer in two basic ways: *cascade* and *fully connected*, that we discuss in the subsequent subsections.

In both architectures, each new memory block receives a connection from each of the network's original inputs. However the connections that comes from other blocks changes according to the architecture used. Every new block has the same number of cells than the first initial block and does not change after it has been added.

5.1 Cascaded Growing LSTM

As commented in §5.1, in cascade architectures, the new units are added one at a time and each new unit receives inputs from every preexisting units and also from itself. So, carrying out this concept on LSTM, each new memory block in addition to have a self-connect connection will also receives a link from each of the network's original inputs and also from every memory cell on preexisting blocks, as can be seen in Figure 8.

In this work, the cascade weights arriving and leaving from the memory blocks were frozen in two different ways. In the first one, all the preexisting weights in the whole net were frozen, leaving only the new block weights to be trained. In the second way, previous weights were frozen except for those of the output units, that remained trainable.

Figure 8 illustrates these two configurations. In both, it can be supposed that there exist just two memory blocks (MB1 and MB2) and a new memory block (MB3) will be added.

For the first configuration (Figure 8(a)), the weights arriving at already existing blocks (MB1 and MB2) are kept frozen (solid lines) and those arri-

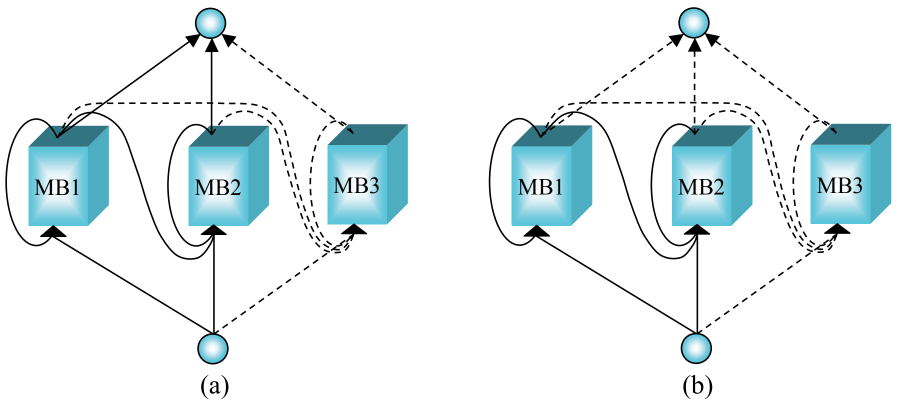


Figure 8: Cascade growing. Whereas in (a) all preexisting weights are frozen, in (b) only weights arriving at already existing blocks are frozen, letting the hidden-output weights free.

ving at new memory block (MB3) are trained repeatedly (dashed lines). The output unit weights are configured in a similar way. In the second configuration (Figure 8(b)) the same procedure is carried out, but now all hidden-output weights are modified during subsequent training.

A third configuration , not shown in the figure, is not to freeze any weight in the net, so the preexisting weight connections still can be trained further after the addition of a new memory block.

5.2 Fully Connected Growing LSTM

In addition to the fact that the new memory block receives connections from every preexisting blocks, as occurs in cascade architecture, in fully connected architectures the preexisting blocks also receives weight connections from each new added block.

As can be seen in Figure 9 no weight in the net is frozen during training, since every cell of each memory block receives new connections after adding a new block.

6 Case of Study

The human cardiovascular system is composed of the hemodynamical system and the Central Nervous System (CNS) control. In this work we try to model the latter by capturing its input/output dynamic behavior.

The CNS generates the regulating signals for the blood vessels and the

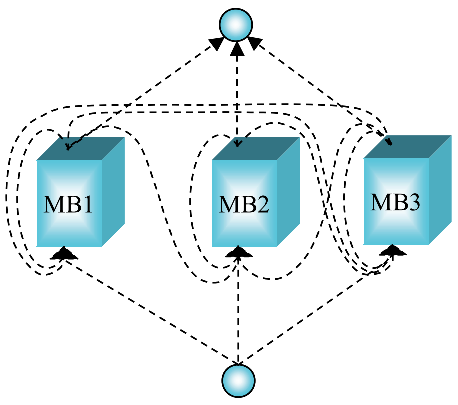


Figure 9: Fully GLSTM.

heart, and it is composed of five controllers: the heart rate controller (HRC), the peripheric resistance controller (PRC), the myocardiac contractility controller (MCC), the venous tone controller (VTC), and the coronary resistance controller (CRC). A simplified diagram of the cardiovascular system is shown in Figure 10. All of these controllers are single–input/single-output (SISO) systems driven by the same input variable, namely the carotid sinus pressure. Although the Carotid Sinus Pressure is not easily measurable, it can be extracted from the differential equation model describing the hemodynamics of the cardiovascular system [Val93]. The five output variables of the controller models are not even amenable to a physiological interpretation, except for the heart rate controller variable, which is the inverse heart rate, measured in seconds between beats.

Whereas the structure and functioning of the hemodynamical system are well known and a number of quantitative models, mostly based on differential equations, have been developed, the functioning of the central nervous system control is of high complexity and still not completely understood. Although some differential equation models for the central nervous system have been postulated [SR69, SS74, Hyn70], these models are not accurate enough, and therefore, the use of other modelling approaches like neural networks may offer an interesting alternative for capturing the behavior of the CNS control [LK90].

7 Experimental Methodology

Temporal pattern recognition involves processing of patterns that evolve over time. The appropriate response at a particular point in time depends

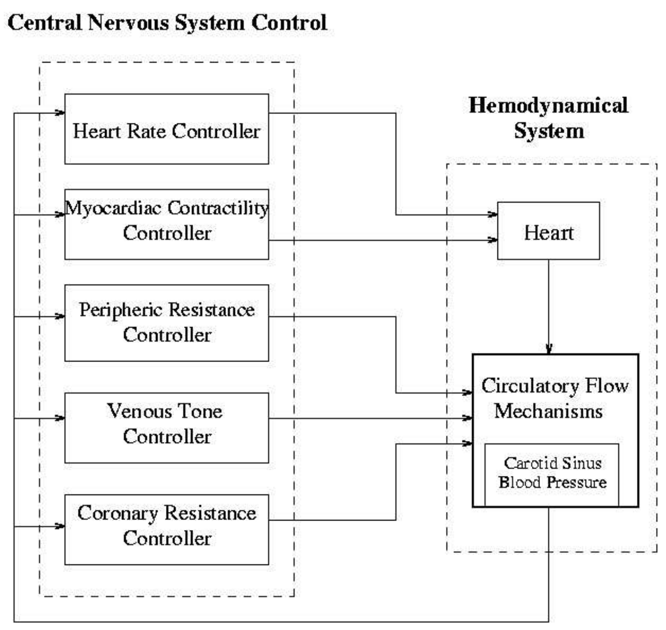

Central Nervous System Control
Heart Rate Controller
Myocardiac Contractility Controller
Peripheric Resistance Controller
Venous Tone Controller
Coronary Resistance Controller
Hemodynamical System
Heart
Circulatory Flow Mechanisms
Carotid Sinus Blood Pressure

Figure 10: Simplified diagram of the cardiovascular system model, composed of the hemodynamical system and the CNS control.

not only on the current input, but potentially on all the previous inputs. Following the strategy used to carry out the prediction of output signals of CNS are described.

7.1 Prediction Strategy

Prediction tasks involve the use currently available input and output points are used to predict a future output point,i.e., given two finite sequences $x(1), x(2), x(3), \ldots, x(t)$ of input signal values and $y(1), y(2), y(3), \ldots, y(t-1)$ of output signal values, predict the value $y(t-1+T)$ of the output signal.

To prepare the data conveniently, we have replaced the original target output $y(t)$ by the difference between the $y(t)$ output value and the previous value $y(t-1)$ multiplied by a scaling factor fs, so that the target is calculated as $t_g(t) = fs * (y(t) - y(t-1)) = \triangle y(t) * fs$. fs scales $\triangle y(t)$ between -1 and 1.

Stepwise and iterated predictions are made. In single-step prediction the network predicts the next output point, $y(t)$, after being fed with the current input $x(t)$ and the last known value of the output, $y(t-1)$. In this case, $T = 1$. It should be noted that, both the inputs and the desired response are provided from the known training points.

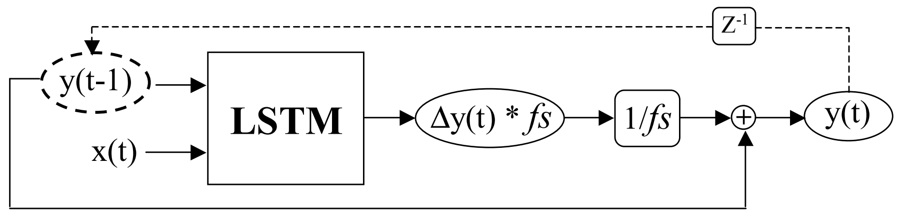


Figure 11: Setup for the output signals.

During iterated prediction with $T = n$ the output is clamped to the $y-$input and the predicted values are fed back n times, i.e. the $y-$input samples are progressively substituted by the output of the network. This closed loop system is illustrated in Figure 11.

7.2 The Data

The data used in the training and test phases are composed of the five controllers mentioned in §6: HRC, PRC, MCC, VTC and CRC, that were recorded with a sampling rate of 0.12 seconds from simulations of a purely differential equation model. Figure 12 illustrates the input and output signals used in the training phase for the HRC controller.

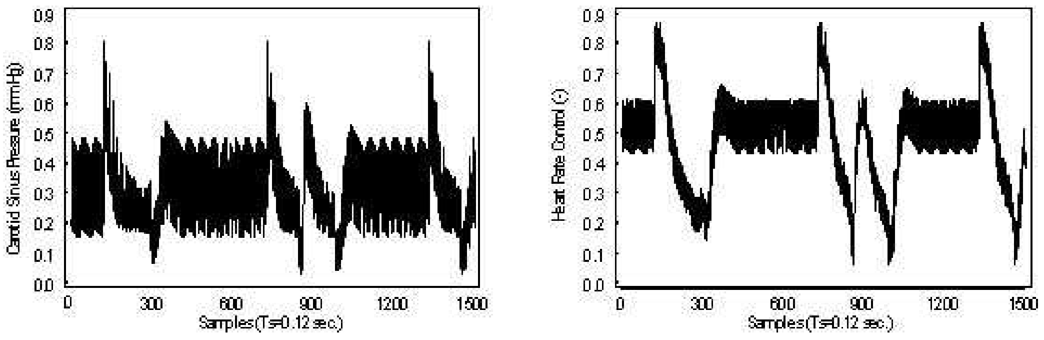


Figure 12: Carotid sinus pressure and Heart rate control signal.

This model had been tuned to represent a specific patient suffering from coronary arterial obstruction, by making the four different physiological variables (right auricular pressure, carotid pressure, coronary blood flow, and heart rate) of the simulation model agree with the measurement data taken from the patient.

The training set consists of 1,500 data points for each controller. Each trained network was validated by using it to forecast six data sets that had not been employed in the learning process. Each one of these six test sets (for each controller), with a size of 300 points each, contains signals representing specific morphologies, allowing the validation of the model for different system behaviors.

7.3 Experimental Setup

The LSTM network architecture used is made up of an input layer with 2 inputs units, an output layer with 1 output unit and a hidden layer consisting of memory cell blocks of size 1.

A new memory block is added to the hidden layer when the previous configuration has been trained at least 1000 epochs and the mean of the error in the last 10 epochs has not improved the previous error mean.

After preliminary experiments, the bias weights for input and output gates in successive blocks were fixed as: -0.5, -1.0, -1.5, and so forth. The initialization of output gates pushes initial memory cells activations towards zero, whereas that of the input gates prevents memory cells from being modified by incoming inputs. As training progresses, the biases become progressively less negative, allowing the serial activation of cells as active participants in the network computation.

The forget gates were initialized with symmetric positive values: $+0.5$ for the first block, $+1.0$ for the second, $+1.5$ for the third, and so forth. The bias initialization must be positive in this case, since it prevents the cells from forgetting everything [GS00a], i.e., when positive signal is used the gates are open what means that no gates are used.

For the cell's input squashing function g different configurations were used, depending on the controller. More specifically, for the HRC and CRC training sets the antisymmetric logarithm function [Alq97] was used, whereas for VTC, MCC and PRC controls g was the logistic sigmoid in $[-1, 1]$. As regard to the output squashing function h and the activation function of the output unit were, they fixed as the linear identity function.

The error measure is given by the normalized mean square error (NMSE), in percent, between the predicted output value and the target value, t_g:

$$NMSE = \frac{E[(t_g(t) - \hat{y}(t))^2]}{t_{g_{\mathrm{var}}}} \cdot 100\% \qquad (7)$$

where $t_{g_{var}}$ is the variance of t_g defined as:

$$t_{g_{\mathrm{var}}} = E[t_g^2(t)] - \{E[t_g(t)]\}^2 \qquad (8)$$

	Fully		Cascade Freezing		Cascade Not-Freezing		Cascade Trainable-Out	
	Train	Test	Train	Test	Train	Test	Train	Test
HRC	4.85	3.99	3.09	4.11	3.26	4.15	2.11	3.27
PRC	0.12	0.69	0.13	0.66	0.17	1.33	0.09	0.61
MCC	0.39	1.11	0.14	0.99	0.11	1.02	0.09	0.98
VTC	0.08	0.96	0.10	0.96	0.18	1.05	0.08	0.96
CRC	0.22	0.37	0.11	0.35	0.18	0.38	0.09	0.26
Av.	1.35	1.42	0.71	1.41	0.78	1.58	0.48	1.21

Table 1: Average of three trials NMSE errors (in percent) for training and test sets of each controller using different growing architectures.

During training, the above NMSE error is used to determine when to finish the learning process, as explained earlier.

8 Results

In this section, two kinds of analysis are carried out. In the former, comparisons among different growing architectures are made and in the latter, LSTM's results obtained throughout the present study are showed and compared with those reported in previous studies on the same task using neural net approaches.

After some preliminary experiments, the number of epochs chosen to stop training was 5000 for HRC, MCC and VTC controllers, 2500 for PRC and 2000 for CRC control. The learning rate has been changed according to the dataset to be learned, but, in essence, it took on values from 0.01 to 0.025. The momentum parameter for each controller was 0.0 for HCR, PRC and CRC training sets and 0.5 for VTC and MCC training sets.

Table 1 displays the outcomes obtained by each controller on the training and test sets using the different growing architectures and configurations (see §5). *Fully* stands for fully connected growing without freezing weights. In order to investigate the effects of freezing weights, we tested three different choices for cascade growing LSTM. *Cascade Freezing* refers to the configuration illustrated by Figure 8(a), *Cascade Not-Freezing* refers to growing cascaded without freezing any weights, and *Cascade Trainable-Out* refers to Figure 8(b) scheme. The results showed are the average of three different training trials using different initial random weights.

As it can be seen, the results achieved by *Cascade Trainable-Out* is far better than those obtained by the other configurations. From now on, GLSTM outcomes achieved uses this configuration.

In the second part of the experiments, the outcomes accomplished in [RA01], which applies LSTM to the task at hand, are used as a baseline to illustrate GLSTM improvements on LSTM. In [BVA98] four different approaches were performed over the task at hand, where three of them are TDNNs [KL90, HKP91] that differ in the training method used: a standard backpropagation algorithm (TDNN-BP), a hybrid procedure composed by repeated cycles of simulated annealing coupled with conjugate gradient algorithm (TDNN-AC), and a genetic algorithm (TD-HNN). This last network uses indeed a different neuron model based on a similarity computation. The other one is a RNN approach, an ASLRNN net, similar to Elman's SRN net, except that is trained by a true gradient descent algorithm that does not truncate error propagation backwards in time.

Concerning to stepwise prediction results, Table 2 shows the average NMSE errors (in percent) of each of the above mentioned architectures against those yielded by GLSTM. For each controller, three different training trials using different random weight initialization in the range $[-0.1, 0.1]$ were carried out. Each trial was applied to the six test sets associated with the controller.

	TD-HNN		TDNN-BP		TDNN-AC		ASLRNN		LSTM		GLSTM	
	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test
HRC	0.11	0.18	1.15	1.52	0.15	0.13	1.63	1.91	2.16	3.41	2.11	3.27
PRC	0.09	0.12	0.94	1.27	0.26	0.14	0.84	1.10	0.25	0.65	0.09	0.61
MCC	0.03	0.06	0.81	1.33	0.09	0.08	0.71	1.18	0.19	1.04	0.09	0.98
VTC	0.03	0.06	0.81	1.33	0.09	0.08	0.71	1.18	0.19	1.01	0.08	0.96
CRC	0.10	0.11	0.47	0.66	0.03	0.04	0.41	0.53	0.18	0.31	0.09	0.26
Av.	0.07	0.11	0.84	1.22	0.12	0.09	0.86	1.18	0.59	1.28	0.48	1.21

Table 2: Average NMSE errors (in percent) for step-wise prediction on training and test sets.

In order to compare the long-term prediction results, where the whole test set is attempted to be predicted ($T = 300$), Table 3 shows the average NMSE errors for different dynamic input/output architectures of three independent training trials for the six test sets of each controller. As can be seen, GLSTM clearly outperforms those results achieved by ASLRNN, TDNN-BP and LSTM.

Next, the prediction results achieved by LSTM and GLSTM are displayed. Figure 13 shows the error curve of LSTM trained with 1, 2 and 3 memory blocks and LSTM trained with growing methods (GLSTM). The black points showed in the graph indicate the epoch in which a new memory block was added to the net. As it can be observed, GLSTM finds a certain stability

		TDNN-BP	ASLRNN	LSTM	GLSTM
HRC	Data Set 1	24.31	28.25	32.81	25.52
	Data Set 2	7.47	8.62	28.66	21.36
	Data Set 3	13.48	16.77	28.40	19.52
	Data Set 4	6.87	8.16	23.25	19.23
	Data Set 5	32.12	38.24	31.70	24.73
	Data Set 6	7.86	9.80	27.86	33.96
Average Error		15.35	18.31	28.78	24.05
PRC	Data Set 1	58.15	50.07	12.30	6.68
	Data Set 2	17.80	16.11	17.29	6.55
	Data Set 3	41.56	36.89	12.15	6.53
	Data Set 4	29.09	26.97	7.15	35.00
	Data Set 5	34.73	38.54	21.15	71.47
	Data Set 6	21.22	18.40	14.22	5.48
Average Error		33.76	31.16	14.04	21.95
MCC	Data Set 1	41.72	55.83	27.48	17.16
	Data Set 2	20.92	17.18	42.88	26.26
	Data Set 3	40.22	35.60	26.26	17.60
	Data Set 4	39.80	42.08	14.44	33.88
	Data Set 5	34.32	36.87	16.15	19.27
	Data Set 6	27.20	23.38	30.91	20.01
Average Error		34.04	35.16	26.35	22.36
VTC	Data Set 1	41.68	54.25	27.01	11.36
	Data Set 2	20.90	16.93	35.43	9.99
	Data Set 3	40.22	35.68	10.16	10.02
	Data Set 4	39.80	41.86	14.58	32.37
	Data Set 5	34.41	36.77	15.50	40.19
	Data Set 6	27.22	23.12	29.90	9.41
Average Error		34.04	34.77	22.02	18.89
CRC	Data Set 1	147.73	148.65	3.70	6.67
	Data Set 2	28.35	36.17	4.63	15.75
	Data Set 3	84.35	83.75	3.00	4.29
	Data Set 4	4.69	4.49	5.48	6.59
	Data Set 5	56.20	58.50	72.29	44.01
	Data Set 6	12.32	11.16	2.99	4.64
Average Error		55.69	57.12	14.73	13.65

Table 3: Average NMSE errors of the CNS controller models inferred by TDNN-BP, ASLRNN, LSTM and GLSTM.

by adding memory blocks incrementally and keeping previous hidden-layer weights frozen.

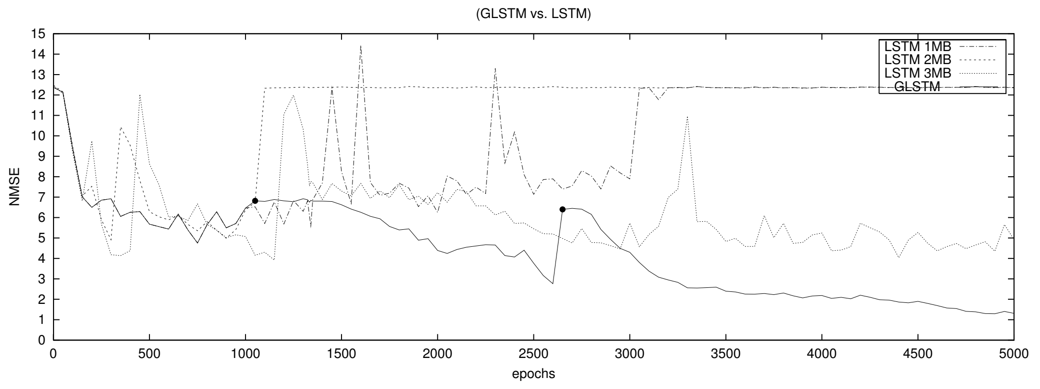
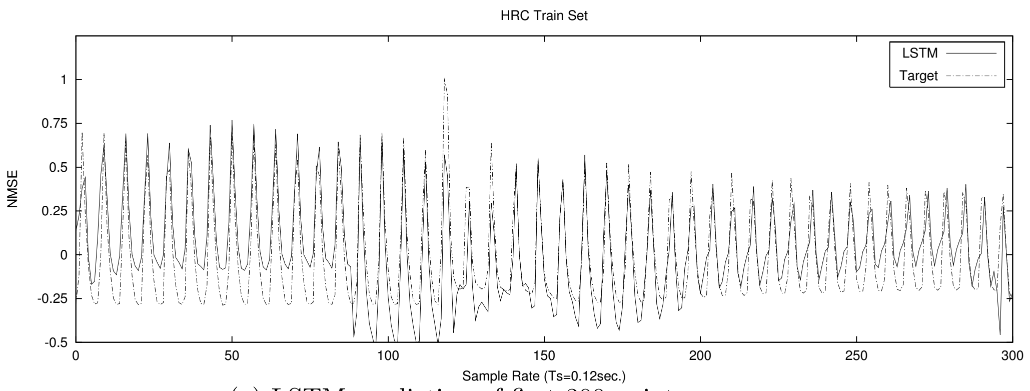


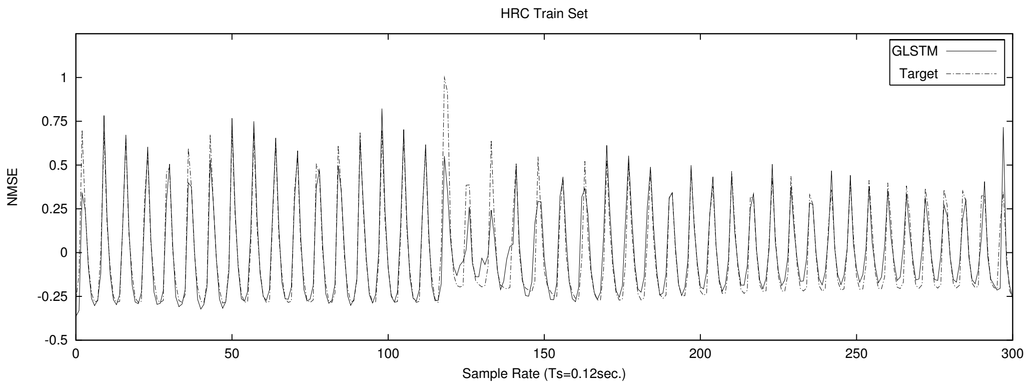
Figure 13: GLSTM vs. LSTM prediction.

The stepwise prediction of the HRC output signal on two part of the training set are illustrated in Figure 14. Where, figures (a) and (c) regard to LSTM prediction, whereas figures (b) and (d) represent GLSTM prediction. The target output signal is shown as dashed line and the predicted signal as solid line.

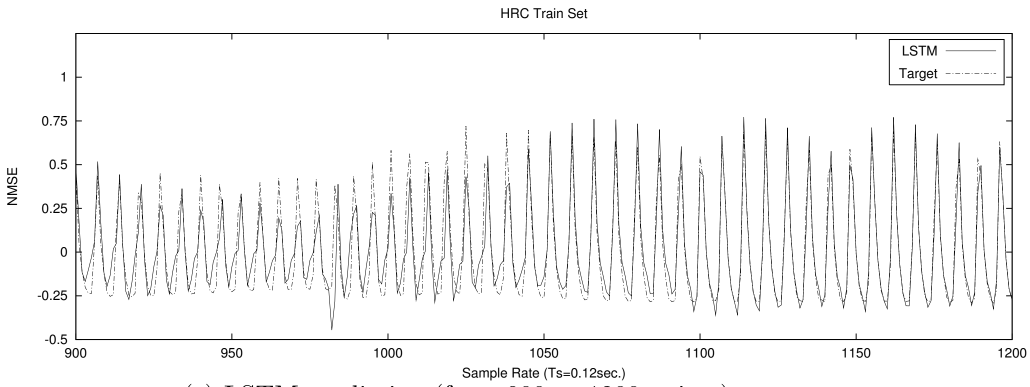
Stepwise and long-term iterated prediction versus the true output signal from test set 1 of the same controller are showed in Figure 15. As can be seen, GLSTM could capture the signal oscillation better than LSTM.



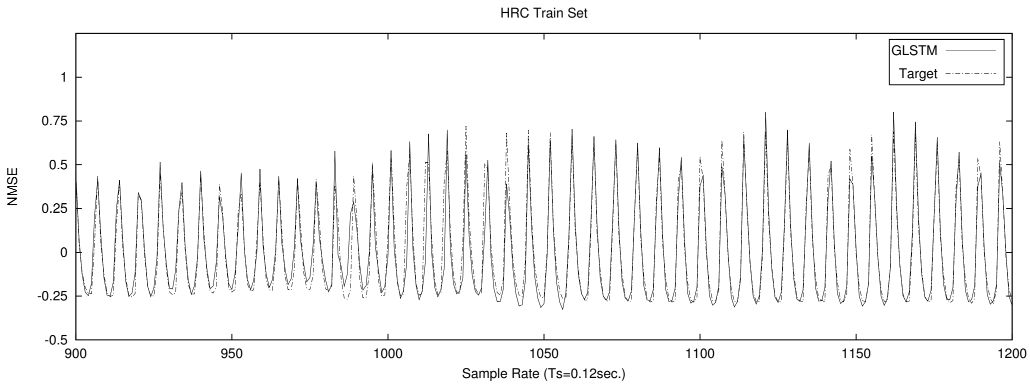
(a) LSTM prediction of first 300 points.



(b) GLSTM prediction of first 300 points.

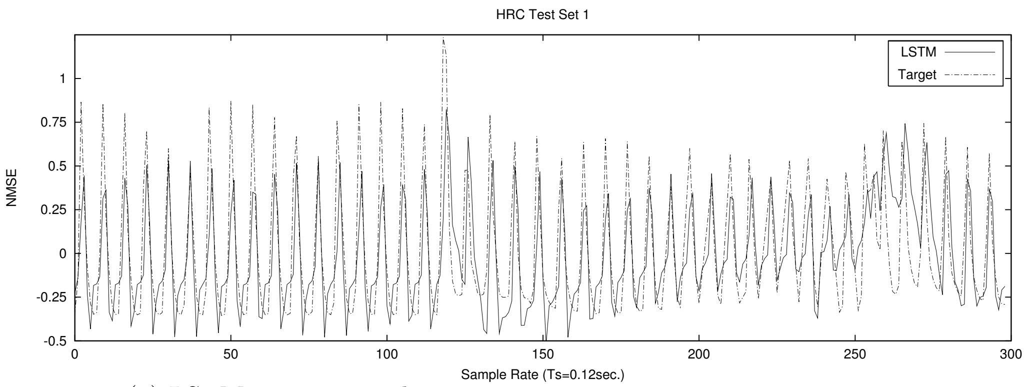


(c) LSTM prediction (from 900 to 1200 points).

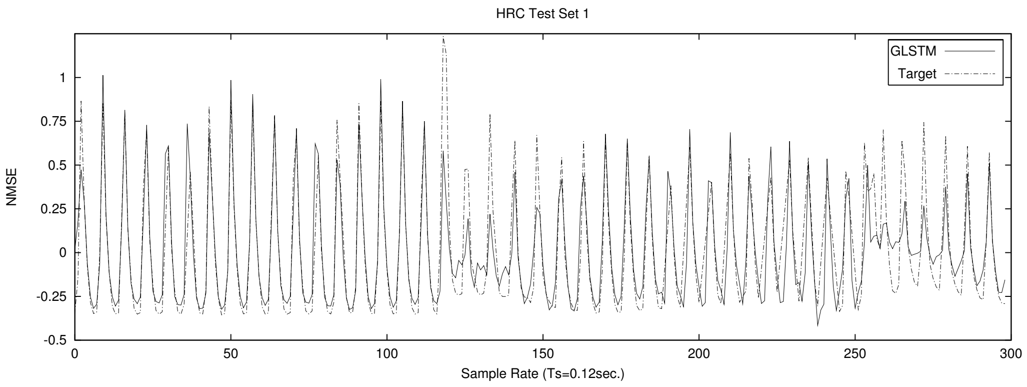


(d) GLSTM prediction (from 900 to 1200 points).

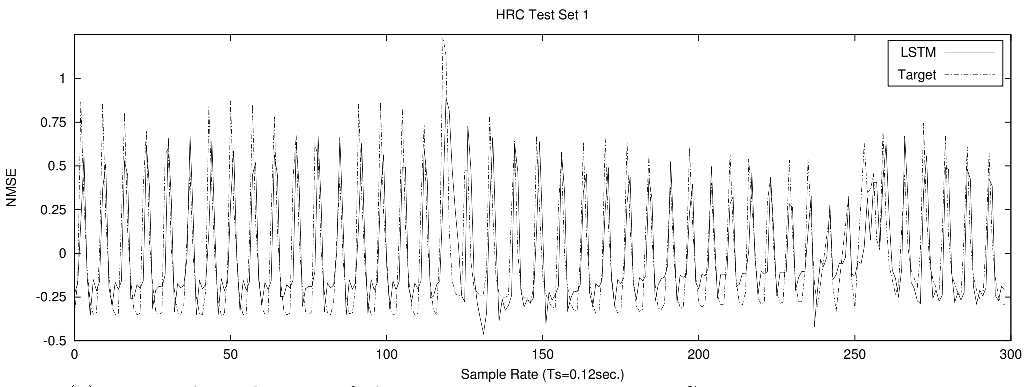
Figure 14: Prediction of the HRC training signals using LSTM and GLSTM.



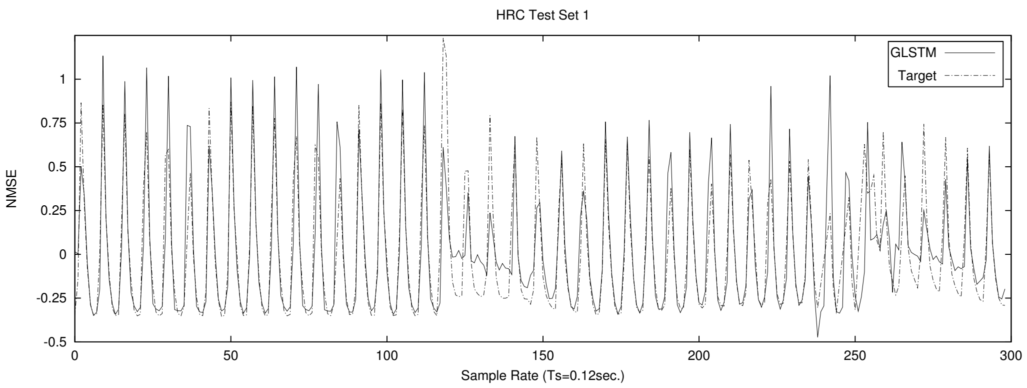
(a) LSTM stepwise prediction.



(b) GLSTM stepwise prediction.



(c) Iterated prediction of the next 300 points using LSTM.



(d) Iterated prediction of the next 300 points using GLSTM.

Figure 15: Prediction of the HRC test set 1 for stepwise prediction and iterated prediction ($T = 300$) using LSTM and GLSTM.

9 Conclusions

The need to fix the size and topology of a network before its training phase is one of the most important practical problems when using common neural network approaches. Therefore, the use of incremental growing methods that build the network automatically are seen as a practical way of making the use of neural networks less complex. This is specially desirable in the case of recurrent neural networks, where the topology can be extremely complex.

Most of the previous work on NN growing methods has dealt with feed-forward NNs and only a few techniques have been proposed for some particular types of recurrent NNs. In this work, we have studied the incremental construction of the LSTM net, which is maybe the more powerful RNN architecture proposed so far. Some different alternatives have been identified and tested on a signal forecasting task concerning the learning of models for the Central Nervous System Control. These include cascade and fully connected hidden layers as well as different levels of freezing previous weights in the cascade case.

It has been shown that, in addition to remove the need to fix the number of hidden units in advance, the growing LSTM can yield a better performance both in the training and test phases. Moreover, the behavior of the error minimization during the training phase appears to be more stable when using Growing LSTM with frozen weights.

Nevertheless, the experiments carried out here need to be complemented with new studies on many other different problems. In addition, the final objective of our work is not only to build LSTM nets incrementally, but rather to develop a methodology for the incremental construction of recurrent NNs for prediction tasks that can combine LSTM units (memory blocks) with other RNN architectures or even with Time-Delay Neural Networks. The underlying idea is to build a RNN as simplest as possible for a given problem and only adding more sophisticated elements (such as LSTM memory blocks) in a parsimonious way when strictly required to improve the approximation and/or generalization performance.

References

[AJ94] R. Alquézar and Sopena J.M. Effect of Unbounded Activation Functions on Learning Performance of Recurrent Networks. Technical Report IC-DT-9402, Institut de Cibernètica, UPC-CSIC, Barcelona, 1994.

[Alq97] R. Alquézar. *Symbolic and Connectionist Learning Techniques for Grammatical Inference*. PhD thesis, Technical University of Catalonia, 1997.

[AS95] R. Alquézar and A. Sanfeliu. An Algebraic Framework to Represent Finite-state Machines in Single Layer Recurrent Neural Networks. *Neural Networks*, 7(5):931–949, 1995.

[Ash89] T. Ash. Dynamic Node Creation in Backpropagation Networks. *Connection Science*, 1(4):365–375, 1989.

[ASP94] P.J. Angeline, G.M. Saunders, and J.B. Pollack. An Evolutionary Algorithm that Constructs Recurrent Neural Networks. *IEEE Transactions on Neural Networks*, 5(1):54–64, 1994.

[AW90] K. Shikano A. Waibel, H. Sawai. Consonant Recognition by Modular Construction of Large Phonemic Time-Delay Neural Networks. In *Proc of the Int. Conf. on Acoust, Speech, and Signal Processing*, volume S3.9, pages 112–115, 1990.

[Bel94] M.G. Bello. Enhanced Training Algorithms, and Integrated Training/Architecture Selection for Multilayer Perceptron Networks. *IEEE Transactions on Neural Networks*, 3(6):864–875, 1994.

[BT91] A.D. Back and A.C. Tsoi. FIR and IIR Synapses, a Neural Net Architecture for Time Series Modelling. *Neural Networks*, 3:375–385, 1991.

[BT94] A.D. Back and A.C. Tsoi. Locally Recurrent Globally Feedforward Networks, a Critical Review of Architectures. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(2):229–239, 1994.

[BVA98] L. Belanche, J.J. Valdés, and R. Alquézar. Fuzzy Heterogeneous Neural Networks for Signal Processing. In *Proc. of Int. Conf. of Artificial Neural Network*, 1998.

[CCG91] S. Chen, C.F.N. Cowan, and P.M. Grant. Orthogonal Least Squares Learning Algorithm for Radial Basis Function Networks. *IEEE Transactions on Neural Networks*, 2(2):302–309, 1991.

[Cn98] M.A. Castaño. *Redes Neuronales Recurrentes para Inferencia Gramatical y Traducción Automática.* PhD thesis, Technical University of Valencia, 1998.

[CUP97] P. Campolucci, A. Uncini, and F. Piazza. A new IIR-MLP Learning Algorithm for on-line Signal Processing. In *Proc. of the Int. Conf. of Acoustic Speech and Signal Processing*, Munich, 1997.

[DF84] P. Diaconis and D. Freedman. Ansymptotics of Graphical Projection Pursuit. *The Annals of Statistics*, 12:793–815, 1984.

[DM93] S. Das and L. Mozer. A Connectionist Symbol Manipulator that Induces Rewrite Rules in Context-free Grammars. In S. Lucas, editor, *Proc. of the 1st Int. Colloquium on Grammatical Inference: Theory Applications and Alternatives*, Essex, UK, 1993. IEEE Press.

[Elm90] J.L. Elman. Finding Structure in Time. *Connection Science*, 14(1):179–211, 1990.

[Fah90] Fahlman, S.E. and Lebiere, C. The Cascade-correlation Learning Architecture. *Advances in Neural Information Processing Systems*, 2:524–532, 1990.

[Fah91] Fahlman, S.E. The Recurrent Cascade-correlation Architecture. *Advances in Neural Information Processing Systems*, 3:190–196, 1991.

[FGS92] P. Frasconi, M. Gori, and G. Soda. Local Feedback Multilayered Networks. *Neural Networks*, 4:120–130, 1992.

[FN93] K. I. Funahashi and Y. Nakamura. Approximation of Dynamical Systems by Continuous Time Recurrent Neural Networks. *IEEE Transactions on Neural Networks*, 6:801–806, 1993.

[FS81] J.H. Friedman and W. Stuetzle. Projection Pursuit Regression. *Journal of American Statistical Association*, 3:817–823, 1981.

[GAP98] A.H. Gee, S.V.B. Aiyer, and R.W. Prager. A Sequential Learning Approach for Single Hidden Layer Neural Networks. *Neural Computation*, 11:65–80, 1998.

[GBM89] M. Gori, Y. Bengio, and R.D. Mori. A Learning Algorithm for Capturing the Dynamic Nature of Speech. In *Proc. of the Int. Joint Conf. on Neural Networks*, volume 2, pages 417–423, Como, Italy, 1989.

[GGC94] M.W. Goudreau, C.L. Giles, and S.T. Chakradhar. First-order vs. Second-order Single Layer Recurrent Neural Networks. *IEEE Transactions on Neural Networks*, 5(3):511–513, 1994.

[GMC$^+$92] C.L. Giles, C.B. Miller, D. Chen, H.H. Chen, G.Z. Sun, and Y.C. Lee. Learning and Extracting Finite State Automata with Second-order Recurrent Neural Networks. *Neural Networks*, 4:393–405, 1992.

[GS00a] F.A. Gers and J. Schmidhuber. Applying LSTM to Time Series Predictable Through Time-Window Approaches. Technical Report 22–00, IDSIA, 2000.

[GS00b] F.A. Gers and J. Schmidhuber. Recurrent Nets that Time and Count. In *Proc. of the Int. Joint Conf. on Neural Networks*, page 273, Como, Italy, 2000.

[GY00] J.B. Gomm and D.L. Yu. Selecting Radial Basis Function Network Centers with Recursive Orthogonal Least Squares Training. In *IEEE Transactions on Neural Networks*, volume 11, pages 306–314, 2000.

[HG95] B.G. Horne and C.L. Giles. An Experimental Comparison of Recurrent Neural Networks. In *Advances in Neural Information Processing Systems*, volume 7, pages 697–704, Cambridge, MA, 1995. MIT Press.

[HKP91] J. Hertz, A. Krogh, and R.G. Palmer. *Introduction to the Theory of Neural Computation.* Addison Wesley, Redwood City, CA, 1991.

[Hoc95] Hochereiter, S. and Schmidhuber, J. Long Short–Term Memory. Technical Report 207–95, FKI, 1995.

[HRM^{+}94] J.N. Hwang, S.R. Ray, M. Maechler, D. Martin, and J. Schimert. Regression Modelling in Backpropagation and Projection Pursuit Learning. *IEEE Transactions on Neural Networks*, 5(3):54–64, 1994.

[HS96] S. Hochereiter and J. Schmidhuber. Bridging Long Time Lags by Weight Guessing and Long Short-Term Memory. In F. L. Silva, J. C. Principe, and L. B. Almeida, editors, *Frontiers in Artificial Intelligence and Applications*, volume 37, pages 65–72, Amsterdam, Netherlands, 1996. IOS Press.

[HS97a] S. Hochereiter and J. Schmidhuber. Long Short-Term Memory. *Neural Networks*, 9:1681–1726, 1997.

[HS97b] S. Hochereiter and J. Schmidhuber. LSTM can Solve Hard Long Time Lag Problems. In M. C. Mozer, M. I. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems*, volume 9, pages 473–479, Cambridge, MA, 1997. MIT Press.

[Hyn70] P.W. Hyndman. *A Digital Simulation of the Human Cardiovascular System and its use in the Study of Sinus Arrhythmia.* PhD thesis, Imperial College, University of London, 1970.

[JC97] A. Nebot J. Cueva, R. Alquézar. Experimental Comparison of Fuzzy and Neural Network Techniques in Learning Models of the Central Nervous System Control. In *Proc. of IEUFIT97*, 1997.

[Jor86] M.I. Jordan. Serial Order: A Parallel Distributed Processing Approach. Technical Report 86–04, Institute of Cognitive Science, University of California, 1986.

[KB95] V. Kurková and B. Beliczynski. Incremental Approximation by one-Hidden-Layer Neural Networks. In *Proc. of the Int. Conf. on Neural Networks*, volume 2, pages 505–510, Perth, Australia, 1995.

[KL90] G. Hinton K. Lang, A. Waibel. A TimeDelay Neural Network Architecture for Isolated Word Recognition. *Neural Computation*, 3:23–34, 1990.

[KTA95] I. Kirschning, H. Tomabechi, and J.I. Aoe. A Parallel Recurrent CascadeCorrelation Neural Network with Natural Connectionist

Glue. In *Proc. of the Int. Conf. on Neural Networks*, volume 2, pages 953–956, Perth, Australia, 1995.

[KY97] T.Y. Kwok and D.Y. Yeung. Constructive Algorithms for Structure Learning in Feedforward Neural Networks for Regression Problems. In *IEEE Transactions on Neural Networks*, volume 8, pages 630–645, 1997.

[LF87] A. Lapedes and R. Faber. Nonlinear signal processing using neural networks and system modelling. Technical Report LAUR-262, Los Alamos National Laboratory, Los Alamos, 1987.

[LHG96] T. Lin, B.G. Horne, and C.L. Giles. Learning Long-term Dependencies is not as Difficult with NARX Recurrent Neural Networks. In *IEEE Transactions on Neural Networks*, volume 6, pages 1329–1338, 1996.

[LK90] A. Law and D. Kelton. *Simulation Modelling and Analysis.* McGraw Hill, New York, 1990.

[LNG95] J. Liang, P.N. Nikiforuk, and M.M. Gupta. Approximation of Discrete-time State-space Trajectories using Dynamic Recurrent Neural Networks. In *IEEE Transactions on Automatic Control*, volume 40, pages 266–1270, 1995.

[LR96] E. Littman and H. Ritter. Learning and Generalisation in Cascade Network Architectures. *Neural Networks*, 8:1521–1539, 1996.

[MF94] P. Manolios and R. Fanelli. First-order Recurrent Neural Networks and Deterministic Finite State Automata. *Neural Networks*, 6:1155–1173, 1994.

[Moz89] M.C. Mozer. A Focused Backpropagation Algorithm for Temporal Pattern Recognition. *Complex Systems*, 3:349–381, 1989.

[Moz92] M.C. Mozer. Induction of Multi-scale Temporal Structure. *Advances in Neural Information Processing Systems*, 5:275–282, 1992.

[Moz93] M.C. Mozer. Neural Net Architectures for Temporal Sequence Processing. *Predicting the Future and Understanding the Past*, 15:243–264, 1993.

[MZ93] S.G. Mallat and Z. Zhang. Matching Pursuits with Time-Frequency Dictionaries. In *IEEE:TSP*, volume 41, pages 3397–3415, 1993.

[NRRP$^+$93] O. Nerrand, P. Roussel-Ragot, L. Personnaz, G. Dreyfus, and S. Marcos. Neural Networks and Nonlinear Adaptive Filtering: Unifying Concepts and New Algorithms. *Neural Networks*, 5:165–199, 1993.

[Pea95] B.A. Pearlmutter. Gradient Calculations for Dynamic Recurrent Neural Networks: A Survey. *IEEE Transactions on Neural Networks*, 6(5):1212–1228, 1995.

[Pre97] L. Prechelt. Investigation of the CasCor Family of Learning Algorithms. *Neural Computation*, 10:885–896, 1997.

[PU91] P. Podar and K. P. Unnikrishnan. Nonlinear Prediction of Speech Signals using Memory Neuron Networks. *Neural Networks for Signal Processing I*, pages 885–896, 1991.

[RA01] S. Ribeiro and R. Alquézar. A Comparative Study on a Signal Forecasting Task applying Long Short-Term Memory (LSTM) Recurrent Neural Networks. In *VI Simpósio Ibero-Americano de Reconhecimento de Padrões*, pages 487–495, Florianopolis, Brazil, 2001.

[RA02] E. Romero and R. Alquézar. A New Incremental Method for Function Appoximation using Feed-forward Neural Networks. In *Proc. of the Int. Joint Conf. on Neural Networks*, Honolulu, Hawaii, 2002.

[RF87] A. J. Robinson and F. Fallside. The utility driven dynamic error propagation network. Technical Report CUED/F-INFENG/TR.1, Engineering Department, Cambridge University, 1987.

[RM99] R.D. Reed and R.J Marks. *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks.* MIT Press, Cambridge, MA, 1999.

[Sch92a] J. Schmidhuber. A Fixed Size Storage O(n^3) Time Complexity Learning Algorithm for Fully Recurrent Continually running Networks. *Neural Networks*, 4(2):243–248, 1992.

[Sch92b] J. Schmidhuber. Learning Complex, Extended Sequences using the Principle of History Compression. *Neural Networks*, 4(2):208–214, 1992.

[SHG97] H.T. Siegelmann, B.G. Horne, and C.L. Giles. Computational Capabilities of Recurrent NARX Neural Networks. In *IEEE Transactions on Systems, Man and Cybernetics*, volume 26, pages 208–216, 1997.

[SK00] O. Stan and E. Kamen. A Local Linearized Least Squares Algorithm for Training Feedforward Neural Networks. In *IEEE Transactions on Neural Networks*, volume 11, pages 487–495, 2000.

[Smi93] F.J. Smieja. Neural Network Constructive Algorithms: Trading Generalisation for Learning Efficiency Circuits. *Systems and Signal Processing*, 12:331–374, 1993.

[SR69] M.F. Snyder and V.C. Rideout. Computer Simulation Studies of the Venous Circulation. In *IEEE Transaction on Biomedical Engineering*, volume 16, pages 325–334, 1969.

[SS74] H. Suga and K. Sagawa. Instantaneous Pressure-volume Relationships and their Ratio in the Excised, Supported Canine Left Ventricle. *Circulation Research*, 53:117–126, 1974.

[Val93] M. Vallverdú. *Modelado y Simulación del Sistema de Control Cardiovascular en Pacientes con Lesiones Coronarias*. PhD thesis, Technical University of Catalonia, 1993.

[VP92] B.C. de Vries and J.C. Principe. The Gamma Model - A New Neural Network Model for Temporal Processing. *Neural Computation*, 5:565–576v, 1992.

[WHH$^+$89] A. Waibel, T. Hanazawa, G. Hinton, K. Shiano, and K. Lang. Phoneme Recognition using Time-Delay Neural Networks. In *IEEE Transactions on Acoustics, Speech and Signal Processing*, volume 37, pages 328–339, 1989.

[WP90] R.J. Williams and J. Peng. An Efficient Gradient-based Algorithm for on-line Training of Recurrent Network Trajectories. *Neural Networks*, 2:491–501, 1990.

[WZ89] R.J. Williams and D. Zipser. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Networks*, 2(1):270–280, 1989.