Training Recurrent Networks by Evolino

Jürgen Schmidhuber*† Daan Wierstra* Matteo Gagliolo*
Faustino Gomez*
*IDSIA, Galleria 2, 6928 Manno (Lugano), Switzerland
†TU Munich, Boltzmannstr. 3, 85748 Garching, München, Germany
{juergen,daan,matteo,tino}@idsia.ch

Abstract

In recent years, gradient-based LSTM recurrent neural networks (RNNs) solved many previously RNN-unlearnable tasks. Sometimes, however, gradient information is of little use for training RNNs, due to numerous local minima. For such cases we present a novel method, namely, EVOlution of systems with LINear Outputs (Evolino). Evolino evolves weights to the nonlinear, hidden nodes of RNNs while computing optimal linear mappings from hidden state to output, using methods such as pseudo-inverse-based linear regression. If we instead use quadratic programming to maximize the margin, we obtain the first evolutionary recurrent Support Vector Machines. We show that Evolino-based LSTM can solve tasks that Echo State nets [15] cannot, and achieves higher accuracy in certain continuous function generation tasks than conventional gradient descent RNNs, including gradient-based LSTM.

1 Introduction

Recurrent Neural Networks (RNNS; [27, 32, 33, 49, 52]) are mathematical abstractions of biological nervous systems that can perform complex mappings from input sequences to output sequences. In principle one can wire them up just like microprocessors, hence RNNs can compute anything a traditional computer can compute [35]. In particular, they can approximate any dynamical system with arbitrary precision [44]. However, unlike traditional, programmed computers, RNNs *learn* their behavior from a training set of correct example sequences. As

training sequences are fed to the network, the error between the actual and desired network output is minimized using gradient descent, whereby the connection weights are gradually adjusted in the direction that reduces this error most rapidly. Potential applications include adaptive robotics, speech recognition, attentive vision, music composition, and innumerably many others where retaining information from arbitrarily far in the past can be critical to making optimal decisions.

Recently, *Echo State Networks* (ESNs; [15]) and a very similar approach, *Liquid State Machines* [17], have attracted significant attention. Composed primarily of a large pool of hidden neurons (typically hundreds or thousands) with fixed random weights, ESNs are trained by computing a set of weights from the pool to the output units using fast, linear regression. The idea is that with so many random hidden units, the pool is capable of very rich dynamics that just need to be correctly "tapped" by setting the output weights appropriately. ESNs have the best known error rates on the Mackey-Glass time series prediction task [15].

The drawback of ESNs is that the only truly computationally powerful, nonlinear part of the net does not learn, whereas previous supervised, gradient-based learning algorithms for sequence-processing RNNs [27, 32, 36, 50, 52] adjust *all* weights of the net, not just the output weights. Unfortunately, early RNN architectures could not learn to look far back into the past because they made gradients either vanish or blow up exponentially with the size of the time lag [9, 10].

A recent RNN called Long Short-Term Memory (LSTM; [11]), however, overcomes this fundamental problem through a specialized architecture that does not impose any unrealistic bias towards recent events by maintaining constant error flow back through time. Using gradient-based learning for both linear and nonlinear nodes, LSTM networks can efficiently solve many tasks that were previously unlearnable using RNNs, e.g. [1–3, 8, 11, 29, 38].

However, even when using LSTM, gradient-based learning algorithms can sometimes yield suboptimal results because rough error surfaces can often lead to inescapable local minima. As we showed [12, 39], many RNN problems involving long-term dependencies that were considered challenging benchmarks in the 1990s, turned out to be trivial in that they could be solved by random weight guessing. That is, these problems were difficult only because learning relied solely on gradient information—there was actually a high density of solutions in the weight space, but the error surface was too rough to be exploited using the local gradient. By repeatedly selecting weights at random, the network does not get stuck in a local minimum, and eventually happens upon one of the plentiful solutions.

One popular method that uses the advantage of random weight guessing in a

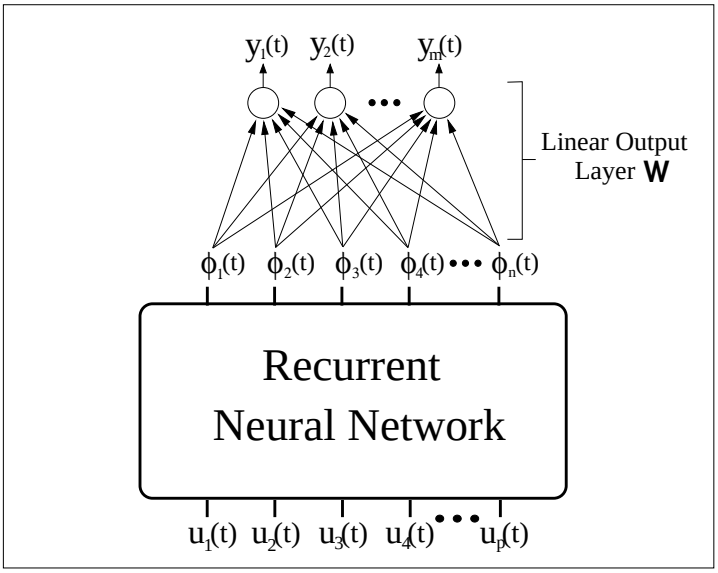

Figure 1: **Evolino network** A recurrent neural network receives sequential inputs $u(t)$ and produce the vector $(\phi_1, \phi_2, \ldots, \phi_n)$ at every time step t. These values are linearly combined with the weight matrix W to yield the network's output vector $y(t)$. While the RNN is evolved, the output layer weights are computed using a fast, optimal method such as linear regression or quadratic programming.

more efficient and principled way is to search the space of RNN weight matrices [20, 21, 26, 45, 53, 54] using evolutionary algorithms [13, 31, 41]. The applicability of such methods is actually broader than that of gradient-based algorithms, since no teacher is required to specify target trajectories for the RNN output nodes. In particular, recent progress has been made with cooperatively coevolving recurrent neurons, each with its own rather small, local search space of possible weight vectors [6, 23, 30]. This approach can quickly learn to solve difficult reinforcement learning control tasks [5, 6, 22], including ones that require use of deep memory [7].

Successfully evolved networks of this type are currently still rather small, with not more than several hundred weights or so. At least for supervised applications, however, such methods may be unnecessarily slow, since they do not exploit gradient information about good directions in the search space.

To overcome such drawbacks, in what follows we limit the domain of evolutionary methods to weight vectors of hidden units, while using fast traditional methods for finding optimal linear maps from hidden to output units. We present

a general framework for training RNNs called EVOlution of recurrent systems with LINear Outputs (Evolino) [40, 51]. Evolino evolves weights to the nonlinear, hidden nodes while computing optimal linear mappings from hidden state to output, using methods such as pseudo-inverse-based linear regression [28] or Support Vector Machines [48], depending on the notion of optimality employed. This generalizes methods such as those of Maillard [19] and Ishii et al. [14, 47] that evolve radial basis functions and ESNs, respectively. Applied to the LSTM architecture, Evolino can solve tasks that ESNs [15] cannot, and achieves higher accuracy in certain continuous function generation tasks than conventional gradient descent RNNs, including gradient-based LSTM (henceforth called G-LSTM).

The next section describes the Evolino framework as well as two specific instances, PI-Evolino (section 2.3) and Evoke (section 2.4) that both combine a cooperative coevolution algorithm called Enforced SubPopulations (section 2.1) with LSTM (section 2.2). In section 3 we apply Evolino to four different time series prediction problems, and in section 4 we provide some concluding remarks.

2 Evolino

Evolino is a general framework for supervised sequence learning that combines neuroevolution (i.e. the evolution of neural networks) and analytical linear methods that are optimal in some sense, such as linear regression or quadratic programming (see section 2.4). The underlying principle of Evolino is that often a linear model can account for a large number of properties of a problem. Properties that require non-linearity and recurrence are then dealt with by evolution.

Figure 1 illustrates the basic operation of an Evolino network. The output of the network at time t, $y(t) \in \mathbb{R}^m$, is computed by the following formulas:

$$y(t) = W\phi(t), \tag{1}$$

$$\phi(t) = f(u(t), u(t-1), \ldots, u(0)), \tag{2}$$

where $\phi(t) \in \mathbb{R}^n$ is the output of a recurrent neural network $f(\cdot)$, and W is a weight matrix. Note that because the networks are recurrent, $f(\cdot)$ is indeed a function of the entire input history, $u(t), u(t-1), \ldots, u(0)$. In the case of maximum margin classification problems [48] we may compute W by quadratic programming. In what follows, however, we focus on mean squared error minimization problems and compute W by linear regression.

In order to evolve an $f(\cdot)$ that minimizes the error between y and the correct output, d, of the system being modeled, Evolino does not specify a particular

evolutionary algorithm, but rather only stipulates that networks be evaluated using the following two-phase procedure.

In the first phase, a training set of sequences obtained from the system, $\{u^i, d^i\}$, $i = 1..k$, each of length l^i, is presented to the network. For each sequence u^i, starting at time $t = 0$, each input pattern $u^i(t)$ is successively propagated through the recurrent network to produce a vector of activations $\phi^i(t)$ that is stored as a row in a $\sum_i^k l^i \times n$ matrix Φ. Associated with each $\phi^i(t)$, is a *target* vector $d^i(t)$ in matrix D containing the correct output values for each time step. Once all k sequences have been seen, the output weights W (the output layer in figure 1) are computed using linear regression from Φ to D. The row vectors in Φ (i.e. the values of each of the n outputs over the entire training set) form a non-orthogonal basis that is combined linearly by W to approximate D.

In the second phase, the training set is presented to the network again, but now the inputs are propagated through the recurrent network $f(\cdot)$ *and* the newly computed output connections to produce predictions $y(t)$. The error in the prediction or the *residual error* is then used as the fitness measure to be minimized by evolution. Alternatively, the error on a previously unseen validation set, or the sum of training and validation error, can be minimized.

Neuroevolution is normally applied to reinforcement learning tasks where correct network outputs (i.e. targets) are not known *a priori*. Evolino uses neuroevolution for supervised learning to circumvent the problems of gradient-based approaches. In order to obtain the precision required for time-series prediction, we do not try to evolve a network that makes predictions directly. Instead, the network outputs a set of vectors that form a basis for linear regression. The intuition is that finding a sufficiently good basis is easier than trying to find a network that models the system accurately on its own.

One possible instantiation of Evolino that we have explored thus far with promising results coevolves the recurrent nodes of LSTM networks using a variant of the Enforced SubPopulations (ESP) neuroevolution algorithm. The next sections describe ESP, LSTM, and the details of how they are combined in the Evolino framework to form two algorithms: PI-Evolino which uses the mean squared error optimality criterion, and Evoke which uses the maximum margin.

2.1 Enforced SubPopulations (ESP)

Enforced SubPopulations differs from standard neuroevolution methods in that instead of evolving complete networks, it *coevolves* separate subpopulations of network components or *neurons* (figure 2). ESP searches the space of networks

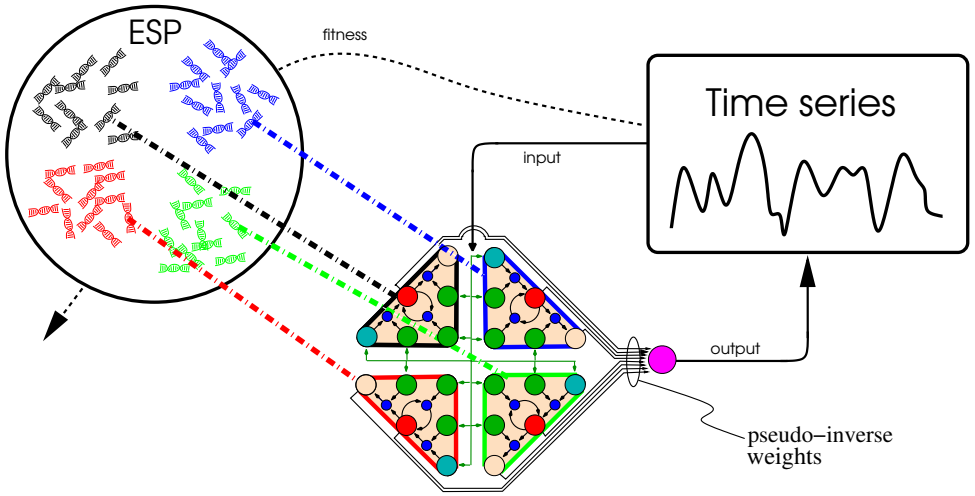

Figure 2: **Enforced SubPopulations (ESP).** The population of neurons is segregated into subpopulations. Networks are formed by randomly selecting one neuron from each subpopulation. A neuron accumulates a fitness score by adding the fitness of each network in which it participated. The best neurons within each subpopulation are mated to form new neurons. The network shown here is an LSTM network with four memory cells (the triangular shapes).

indirectly by sampling the possible networks that can be constructed from the subpopulations of neurons. Network evaluations serve to provide a fitness statistic that is used to produce better neurons that can eventually be combined to form a successful network. This cooperative coevolutionary approach is an extension to Symbiotic, Adaptive Neuroevolution (SANE; [23]) which also evolves neurons, but in a single population. By using separate subpopulations, ESP accelerates the specialization of neurons into different sub-functions needed to form good networks because members of different evolving sub-function types are prevented from mating. Subpopulations also reduce noise in the neuron fitness measure because each evolving neuron type is guaranteed to be represented in every network that is formed. Both of these features allow ESP to evolve networks more efficiently than SANE [4].

ESP normally uses crossover to recombine neurons. However, for the present Evolino variant, where fine local search is desirable, ESP uses Cauchy-distributed mutation to produce all new individuals, making the approach in effect an Evolution Strategy [42]. More concretely, evolution proceeds as follows:

1. Initialization: The number of hidden units H in the networks that will be evolved is specified and a subpopulation of n neuron chromosomes is created for each hidden unit. Each chromosome encodes a neuron's input and recurrent connection weights with a string of random real numbers.

2. Evaluation: A neuron is selected at random from each of the H subpopulations, and combined to form a recurrent network. The network is evaluated on the task and awarded a fitness score. The score is added to the *cumulative fitness* of each neuron that participated in the network. This procedure is repeated until each neuron participated in m evaluations.

3. Reproduction: For each subpopulation the neurons are ranked by fitness, and the top quarter of the chromosomes or *parents* in each subpopulation are duplicated and the copies or *children* are mutated by adding noise to all of their weight values from the Cauchy distribution $f(x) = \frac{\alpha}{\pi(\alpha^2+x^2)}$, where the parameter α determines the width of the distribution. The children then replace the lowest-ranking half of their corresponding subpopulation.

4. Repeat the Evaluation–Reproduction cycle until a sufficiently fit network is found.

If during evolution the fitness of the best network evaluated so far does not improve for a predetermined number of generations, a technique called *burst mu-*

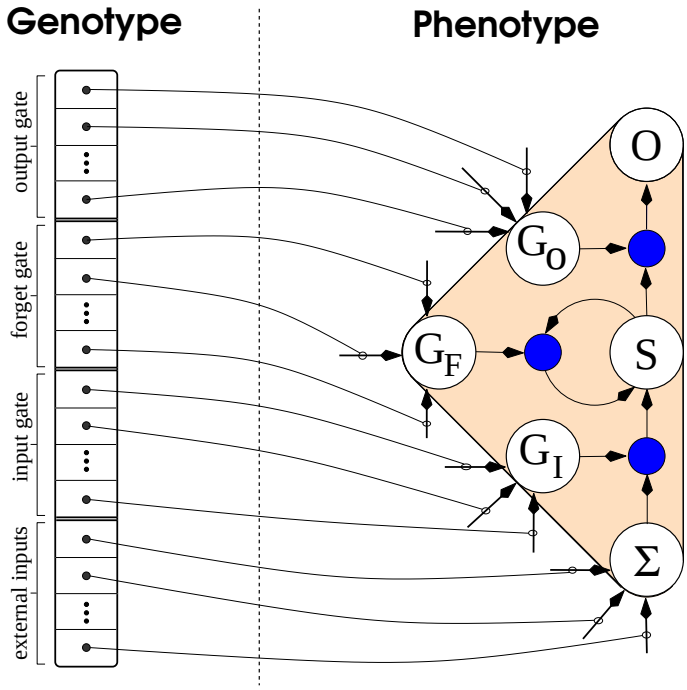

Figure 3: **Genotype-Phenotype mapping.** Each chromosome (genotype, left) in a subpopulation encodes the external input, and input, output, and forget gate weights of an LSTM memory cell (right). The weights leading out of the state (S) and output (O) units are not encoded in the genotype, but are instead computed at evaluation time by linear regression.

tation is used. The idea of burst mutation is to search the space of modifications to the best solution found so far. When burst mutation is activated, the best neuron in each subpopulation is saved, the other neurons are deleted, and new neurons are created for each subpopulation by adding Cauchy distributed noise to its saved neuron. Evolution then resumes, but now searching in a neighborhood around the previous best solution. Burst mutation injects new diversity into the subpopulations and allows ESP to continue evolving after the initial subpopulations have converged.

2.2 Long Short-Term Memory

LSTM is a recurrent neural network purposely designed to learn long-term dependencies via gradient descent. The unique feature of the LSTM architecture is

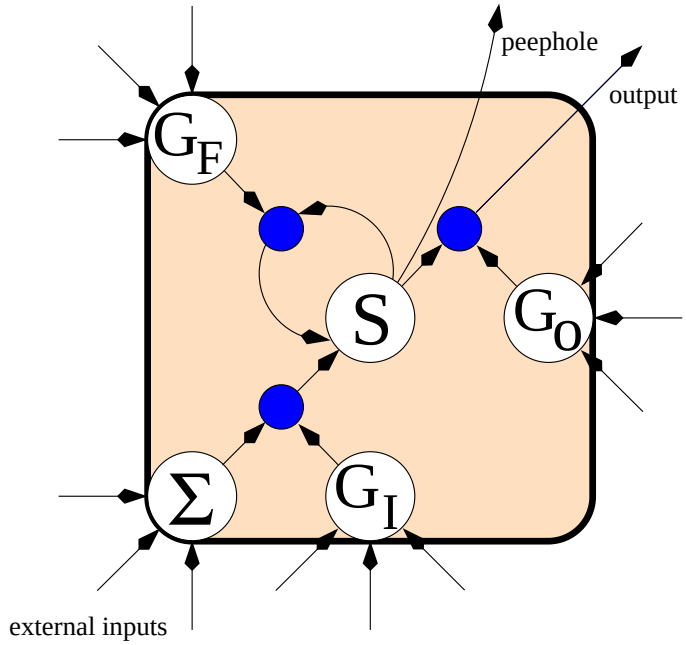

Figure 4: **Long Short-Term Memory** The figure shows an LSTM *memory cell*. The cell has an internal state S together with a forget gate (G_F) that determines how much the state is attenuated at each time step. The input gate (G_I) controls access to the cell by the external inputs that are summed into the Σ unit, and the output gate (G_O) controls when and how much the cell fires. Small dark nodes represent the multiplication function.

the *memory cell* that is capable of maintaining its activation indefinitely (figure 4). Memory cells consist of a linear unit which holds the *state* of the cell, and three gates that can open or close over time. The input gate "protects" a neuron from its input: only when the gate is open, can inputs affect the internal state of the neuron. The output gate lets the state out to other parts of the network, and the forget gate enables the state to "leak" activity when it is no longer useful.

The state of cell i is computed by:

$$s_i(t) = net_i(t)g_i^{in}(t) + g_i^{forget}(t)s_i(t-1), \tag{3}$$

where g^{in} and g^{forget} are the activation of the input and forget gates, respectively, and net is the weighted sum of the external inputs (indicated by the Σs in figure 4):

$$net_i(t) = h(\sum_j w_{ij}^{cell} c_j(t-1) + \sum_k w_{ik}^{cell} u_k(t)), \tag{4}$$

where h is usually the identity function, and c_j is the output of cell j:

$$c_j(t) = tanh(g_j^{out}(t)s_j(t)). \tag{5}$$

where g^{out} is the output gate of cell j. The amount each gate g_i of memory cell i is open or closed at time t is calculated by:

$$g_i^{type}(t) = \sigma(\sum_j w_{ij}^{type} c_j(t-1) + \sum_k w_{ik}^{type} u_k(t)), \tag{6}$$

where $type$ can be $input$, $output$, or $forget$, and σ is the standard sigmoid function. The gates receive input from the output of other cells c_j, and from the external inputs to the network.

2.3 Combining LSTM, ESP, and Pseudoinverse in Evolino

We apply our general Evolino framework to the LSTM architecture, using ESP for evolution and regression for computing linear mappings from hidden state to outputs. ESP coevolves subpopulations of LSTM memory cells instead of standard recurrent neurons (figure 2). Each chromosome is a string containing the external input weights and the input, output, and forget gate weights, for a total of $4 * (I + H)$ weights in each memory cell chromosome, where I is the number of external inputs and H is the number of memory cells in the network. There are four sets of $I + H$ weights because the three gates and the cell itself receive input from outside the cell and the other cells. Figure 3 shows how the memory

cells are encoded in an ESP chromosome. Each chromosome in a subpopulation encodes the connection weights for a cell's input, output, and forget gates, and external inputs.

The linear regression method used to compute the output weights (W in equation 2) is the Moore-Penrose pseudo-inverse method, which is both fast and optimal in the sense that it minimizes the summed squared error [28]—compare [19] for an application to feedforward RBF nets and [14] for an application to Echo State Networks. The vector $\phi(t)$ consists of both the cell outputs c_i, and their internal states, s_i, so that the pseudo-inverse computes two connection weights for each memory cell. We refer to the connections from internal states to the output units as "output peephole" connections, since they peer into the interior of the cells.

For continuous function generation, *backprojection* (or *teacher forcing* in standard RNN terminology) is used where the predicted outputs are fed back as inputs in the next time step: $\phi(t) = f(u(t), y(t-1), u(t-1), \ldots, y(0), u(0))$.

During training, the correct target values are backprojected, in effect "clamping" the network's outputs to the right values. During testing, the network backprojects its own predictions. This technique is also used by ESNs, but whereas ESNs do not change the backprojection connection weights, Evolino evolves them, treating them like any other input to the network. In the experiments described below, backprojection was found useful for continuous function generation tasks, but interferes to some extent with performance in the discrete context-sensitive language task.

2.4 Evoke: Evolino for Recurrent Support Vector Machines

As outlined in section 2, the Evolino framework does not prescribe a particular optimality criterion for computing the output weights. If we replace mean squared error with the maximum margin criterion of Support Vector Machines (SVMs) [48], the optimal linear output weights can be evaluated using e.g. quadratic programming, as in traditional SVMs. We call this Evolino variant *EVOlution of systems with KErnel-based outputs* (Evoke; [37]). The Evoke variant of equation 1 becomes:

$$y(t) = w_0 + \sum_{i=1}^{k} \sum_{j=0}^{l_i} w_{ij} K(\phi(t), \phi^i(j)) \tag{7}$$

where $\phi(t) \in \mathbb{R}^n$ is, again, the output of the recurrent neural network $f(\cdot)$ at

time t (equation 2), $K(\cdot, \cdot)$ is a predefined kernel function, and the weights w_{ij} correspond to k training sequences ϕ^i, each of length l_i, and are computed with the support vector algorithm.

Support Vector Machines are powerful regressors and classifiers that make predictions based on a linear combination of kernel basis functions. The kernel maps the input feature space to a higher dimensional space where the data is linearly separable (in classification), or can be approximated well with a hyperplane (in regression). A limited way of applying existing SVMs to time series prediction [24, 25] or classification [34] is to build a training set either by transforming the sequential input into some static domain (e.g., a frequency and phase representation), or by considering restricted, fixed time windows of m sequential input values. One alternative presented in [43] is to average kernel distance between elements of input sequences aligned to m points. Of course such approaches are bound to fail if there are temporal dependencies exceeding m steps. In a more sophisticated approach by Suykens and Vandewalle [46], a window of m previous output values is fed back as input to a recurrent model with a fixed kernel. So far, however, there has not been any recurrent SVM that *learns* to create internal state representations for sequence learning tasks involving time lags of arbitrary length between important input events. For example, consider the task of correctly classifying arbitrary instances of the context-free language $a^n b^n$ (n a's followed by n b's, for arbitrary integers $n > 0$).

For Evoke, the evolved recurrent neural network (RNN) is a preprocessor for a standard SVM kernel. The combination of both can be viewed as an adaptive kernel learning a task-specific distance measure between pairs of input sequences. Although Evoke uses SVM methods, it can solve several tasks that traditional SVMs cannot even solve in principle. We will see that it also outperforms recent state-of-the-art RNNs on certain tasks, including Echo State Networks (ESNs) [15] and previous gradient descent RNNs [11, 27, 32, 33, 49, 52].

3 Experiments

Experiments with PI-Evolino were carried out on four test problems: context-sensitive languages, multiple superimposed sine waves, parity problem with display, and the Mackey-Glass time series. The first two were chosen to highlight Evolino's ability to perform well in both discrete and continuous domains, and to solve tasks that neither ESNs [15] nor traditional gradient descent RNNs [27, 32, 33, 49, 52] can solve well. We also report successful experiments with

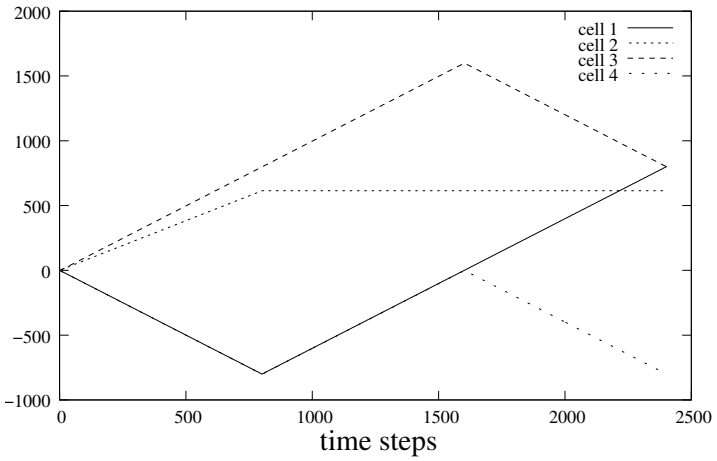

Figure 5: **Internal state activations.** The state activations for the 4 memory cells of an Evolino network being presented the string $a^{800}b^{800}c^{800}$. The plot clearly shows how some units function as "counters," recording how many as and bs have been seen. More complex, non-linear behavior by the gates is not shown.

Evoke-trained RSVMs on these tasks. The parity problem with display demonstrates PI-Evolino's ability to cope with rough error surfaces that confound gradient based approaches including G-LSTM. Although the Mackey-Glass system can be modeled very accurately by non-recurrent systems [15], it was selected to compare PI-Evolino with ESNs, the reference method on this widely used time series benchmark.

3.1 Context-Sensitive Grammars

Context-sensitive languages are languages that cannot be recognized by deterministic finite-state automata, and are therefore more complex in some respects than regular languages. In general, determining whether a string of symbols belongs to a context-sensitive language requires remembering all the symbols in the string seen so far, ruling out the use of non-recurrent architectures.

To compare Evolino-based LSTM with published results for G-LSTM [1], we chose the language $a^n b^n c^n$. The task was implemented using networks with four input units, one for each symbol (a, b, c) plus the start symbol S, and four output units, one for each symbol plus the termination symbol T. Symbol strings were presented sequentially to the network, with each symbol's corresponding input unit set to 1, and the other three set to -1. At each time step, the network must

predict the possible symbols that could come next in a legal string. Legal strings in $a^n b^n c^n$ are those in which the number of as, bs, and cs is equal, e.g. ST, $SabcT$, $SaabbccT$, $SaaabbbcccT$, and so forth. So, for $n = 3$, the set of input and target values would be:

Input:	S	a	a	a	b	b	b	c	c	c
Target:	a/T	a/b	a/b	a/b	b	b	c	c	c	T

Evolino-based LSTM networks were evolved using 8 different training sets, each containing legal strings with values for n as shown in the first column of Table 1. In the first four sets, n ranges from 1 to k, where $k = 10, 20, 30, 40$. The second four sets consist of just two training samples, and were intended to test how well the methods could induce the language from a nearly minimal number of examples.

LSTM networks with memory cells were evolved (4 for PI-Evolino, 5 for Evoke), with random initial values for the weights between -0.1 and 0.1 for Evolino and between -5.0 and 5.0 for Evoke. The Cauchy noise parameter α for both mutation and burst mutation was set to 0.00001 for Evolino and to 0.1 for Evoke, i.e. 50% of the mutations is kept within these bounds. In keeping with the setup in [1], we added a bias unit to the Forget gates and Output gates with values of $+1.5$ and -1.5, respectively. For Evoke, the parameters of the SVM module were chosen heuristically: a Gaussian kernel with standard deviation 2.0 and capacity 100.0. Evolino evaluates fitness on the entire training set, but Evoke uses a slightly different way of evaluating fitness: while the training set consists of the first half of the strings, fitness was defined as performance on the second half of the data, the validation set. Evolution was terminated after 50 generations, after which the best network in each simulation was tested.

Table 1 compares the results of Evolino-based LSTM, using Pseudoinverse as supervised learning module (PI-Evolino), with those of G-LSTM from [1]; "Standard PI-Evolino" uses parameter settings that are a compromise between discrete and continuous domains. If we set h to the $tanh$ function, we obtain "Tuned PI-Evolino." We never managed to train ESNs to solve this task, presumably because the random pre-wiring of ESNs rarely represents an algorithm for solving such context sensitive language problems.

The Standard PI-Evolino networks had generalization very similar to that of G-LSTM on the $1..k$ training sets, but slightly better on the two-example training sets. Tuned PI-Evolino showed a dramatic improvement over G-LSTM on all of

Training data	Standard PI-Evolino	Tuned PI-Evolino	Gradient-LSTM
1..10	1..29	1..53	1..28
1..20	1..67	1..95	1..66
1..30	1..93	1..355	1..91
1..40	1..101	1..804	1..120
10,11	4..14	3..35	10..11
20,21	13..36	5..39	17..23
30,31	26..39	3..305	29..32
40,41	32..53	1..726	35..45

Table 1: **Results for the $a^n b^n c^n$ language.** The table compares Pseudoinverse based Evolino (PI-Evolino) with Gradient-based LSTM (G-LSTM) on the $a^n b^n c^n$ language task. "Standard" refers to Evolino with the parameter settings used for both discrete and continuous domains ($a^n b^n c^n$ and superimposed sine waves). The "Tuned" version is biased to the language task: we additionally squash the cell input with the $tanh$ function. The leftmost column shows the set of strings used for training in each of the experiments. The other three columns show the set of legal strings to which each method could generalize after 50 generations (3000 evaluations), averaged over 20 runs. The upper training sets contain all strings up to the indicated length. The lower training sets only contain a single pair. PI-Evolino generalizes better than G-LSTM, most notably when trained on only two examples of correct behavior. The G-LSTM results are taken from [1].

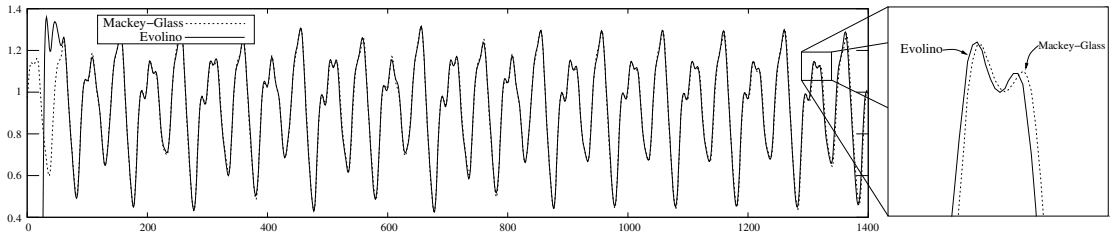

Figure 6: **Performance of PI-Evolino on the Mackey-Glass time-series.** The plot shows both the Mackey-Glass system and the prediction made by a typical Evolino-based LSTM network evolved for 50 generations. The obvious difference between the system and the prediction during the first 100 steps is due to the washout time. The inset shows a magnification that illustrates more clearly the deviation between the two curves.

the training sets, but, most remarkably on the two-example sets where it was able to generalize on average to all strings up to $n = 726$ after being trained on only $n = \{40, 41\}$. Evoke's performance was superior for $N = 10$ and $N = 20$, generalizing up to $n = 257$ and $n = 374$ respectively, but degraded for larger values of N, for which both PI-Evolino and G-LSTM achieved better results. Figure 3.1, shows the internal states of each of the 4 memory cells of one of the networks evolved by PI-Evolino while processing $a^{800}b^{800}c^{800}$.

3.2 Parity Problem with Display

The Parity Problem with Display involves classifying sequences consisting of 1's and -1's according to whether the number of 1's is even or odd. The target, which depends on the entire sequence, is a display of 10×10 output neurons depicting "O" for odd and "E" for even. The display prevents the task from being solved by guessing the network weights [12], and makes the error gradient very rough.

We trained PI-Evolino with 2 memory cells on 50 random sequences of length between 100 and 110. Unlike G-LSTM, which typically cannot solve this task due to a lack of global gradient information, PI-Evolino learned a perfect display classification on a test set within 30 generations, in all 20 experiments.

3.3 Mackey-Glass Time-Series Prediction

The Mackey-Glass system (MGS; [18]) is a standard benchmark for chaotic time series prediction. The system produces an irregular time series that is produced

by the following differential equation: $\dot{y}(t) = \alpha y(t-\tau)/(1+y(t-\tau)^{\beta}) - \gamma y(t)$, where the parameters are usually set to $\alpha = 0.2, \beta = 10, \gamma = 0.1$. The system is chaotic whenever the delay $\tau > 16.8$. We use the most common value for the delay $\tau = 17$.

Although the MGS can be modeled very accurately using feedforward networks with a time-window on the input, we compare PI-Evolino to ESNs (currently the best method for MGS) in this domain to show its capacity for making precise predictions. We used the same setup in our experiments as in [15].

Networks were evolved in the following way. During the first phase of an evaluation, the network predicts the next function value for 3000 time steps with the benefit of the backprojected target from the previous time step. For the first 100 "washout" time steps, the vectors $\phi(t)$ are not collected, i.e only the $\phi(t), t = 101..3000$, are used to calculate the output weights using the pseudo-inverse. During the second phase, the previous target is backprojected only during the washout time, after which the network runs freely by backprojecting its own predictions. The fitness score assigned to the network is the MSE on time steps $101..3000$.

Networks with 30 memory cells were evolved for 200 generations, and a Cauchy noise α of 10^{-7}. A bias input of 1.0 was added to the network, the backprojection values were scaled by a factor of 0.1, and the cell input was squashed with the $tanh$ function.

At the end of an evolutionary run, the best network found was tested by having it predict using the backprojected previous target for the first 3000 steps, and then run freely from time step 3001 to 3084[1]. The average NRMSE_{84} for PI-Evolino with 30 cells over the 15 runs was 1.9×10^{-3} compared to $10^{-4.2}$ for ESNs with 1000 neurons [15]. The PI-Evolino results are currently the second-best reported so far.

Figure 6 shows the performance of an Evolino network on the MG time-series with even fewer memory cells, after 50 generations. Because this network has fewer parameters, it is unable to achieve the same precision as with 30 neurons, but it demonstrates how Evolino can learn such functions very quickly; in this case within approximately 3 minutes of CPU time.

[1]The normalized root mean square error (NRMSE_{84}) 84 steps after the end of the training sequence is the standard comparison measure used for this problem.

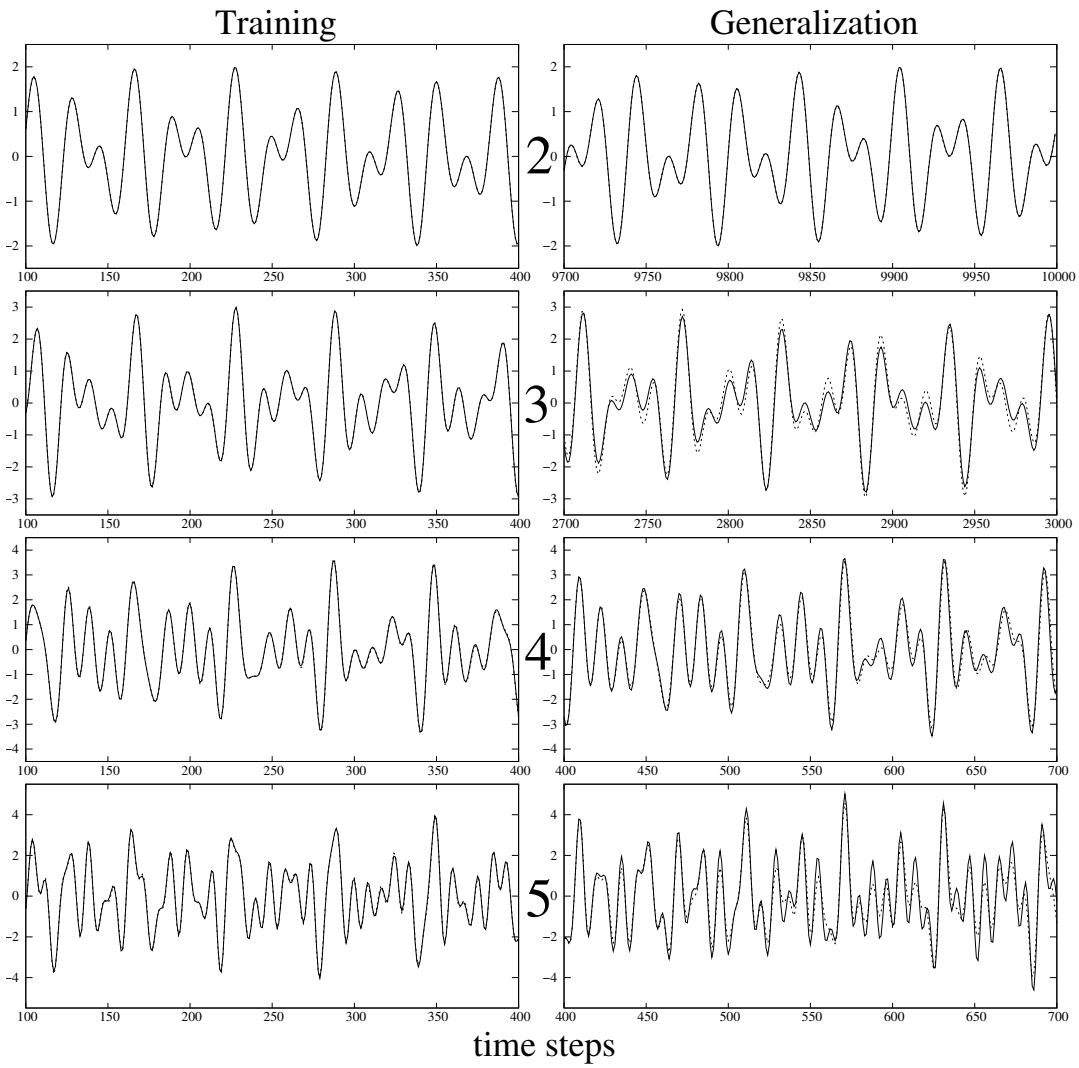

Figure 7: **Performance of PI-Evolino on the superimposed sine wave tasks.** The plots show the behavior of a typical network produced after a specified number of generations: 50 for the two-, three-, and four-sine functions, and 150 for the five-sine function. The first 300 steps of each function, in the left column, were used for training. The curves in the right column show values predicted by the networks (dashed curves) further into the future vs. the corresponding reference signal (solid curves). While the onset of noticeable prediction error occurs earlier as more sines are added, the networks still track the correct behavior for hundreds of time steps, even for the five-sine case.

No. sines	No. cells	Training NRMSE	Gen. NRMSE
2	10	2.01×10^{-3}	4.15×10^{-3}
3	15	2.44×10^{-3}	8.04×10^{-3}
4	20	1.51×10^{-2}	1.10×10^{-1}
5	20	1.60×10^{-2}	1.66×10^{-1}

Table 2: **PI-Evolino results for multiple superimposed sine waves.** The table shows the number of memory cells, training error, and generalization error for each of the superimposed sine wave functions. The training NRMSE is calculated on time steps 100 to 400 (i.e. the washout time is not included in the measure); the generalization NRMSE is calculated for time steps 400 to 700 (averaged over 20 experiments).

3.4 Multiple Superimposed Sine Waves

Learning to generate a sinusoidal signal is a relatively simple task that requires only one bit of memory to indicate whether the current network output is greater or less than the previous output. When sine waves with frequencies that are not integer multiples of each other are superimposed, the resulting signal becomes much harder to predict because its wavelength can be extremely long, i.e. there are large number of time steps before the periodic signal repeats. Generating such a signal accurately without recurrency would require a prohibitively large time-delay window using a feedforward architecture.

Jaeger reports [16] that Echo State Networks are unable to learn functions composed of even two superimposed oscillators, in particular $sin(0.2x)+sin(0.311x)$. The reason for this is that the dynamics of all the neurons in the ESN "pool" are coupled, while this task requires that the two underlying both oscillators be represented by the network's internal state.

Here we show how Evolino-based LSTM not only can solve the two-sine function mentioned above, but also more complex functions formed by superimposing up to three more sine waves. Each of the functions was constructed by $\sum_{i=1}^{n} sin(\lambda_i x)$, where n is the number of sine waves and $\lambda_1 = 0.2, \lambda_2 = 0.311, \lambda_3 = 0.42, \lambda_4 = 0.51$, and $\lambda_5 = 0.74$.

For this task, PI-Evolino networks were evolved using the same setup and procedure as for the Mackey-Glass system except that steps $101..400$ were used the calculate the output weights in the first evaluation phase, and fitness in the second phase. Again, during the first 100 washout time steps the vectors $\phi(t)$

were not collected for computing the pseudo-inverse.

The first three tasks, $n = 2, 3, 4$, used subpopulations of size 40 and simulations were run for 50 generations. The five-sine wave task, $n = 5$, proved much more difficult to learn requiring a larger subpopulation size of 100, and simulations were allowed to run for 150 generations. At the end of each run, the best network was tested for generalization on data points from time-steps $401..700$, making predictions using backprojected previous predictions.

For Evoke, a slightly different setting was used, in which networks were evolved to minimize the sum of training and validation error, on points $100..400$ and $400..700$ respectively, and tested on points $700..1000$. The weight range was set to $[-1.0, 1.0]$, and a Gaussian kernel with standard deviation 2.0 and capacity 10.0 was used for the SVM.

Table 2 shows the number of memory cells used for each task, and the average summed squared error on both the training set and the testing set for the best network found during each evolutionary run of PI-Evolino. Evoke achieved a relatively low generalization NRMSE of 1.03×10^{-2} on the double sines problem, but gave unsatisfactory results for three or more sines.

Figure 7 shows the behavior of one the successful networks from each of the tasks. The column on the left shows the target signal from Table 2, and the output generated by the network on the training set. The column on the right shows the same curves forward in time to show the generalization capability of the networks. For the two-sine function, even after 9000 time-steps, the network continues to generate the signal accurately. As more sines are added, the prediction error grows more quickly, but the overall behavior of the signal is still retained.

Figure 8 reveals how the two-sine wave is represented internally by a typical PI-Evolino network. For the purpose of illustration, a less accurate network containing only three cells instead of ten, is shown. The upper graph shows the overall output of the network, while the other graphs show the output peephole and output activity of each cell multiplied by the corresponding pseudoinverse-generated output weight.

Although the network can generate the function very accurately for thousands of time steps, it does not do so by implementing sinusoidal oscillators. Instead, each cell by itself behaves in a manner that is qualitatively similar to the two-sine but scaled, translated, and phase-shifted. These six separate signals are added together to produce the network output.

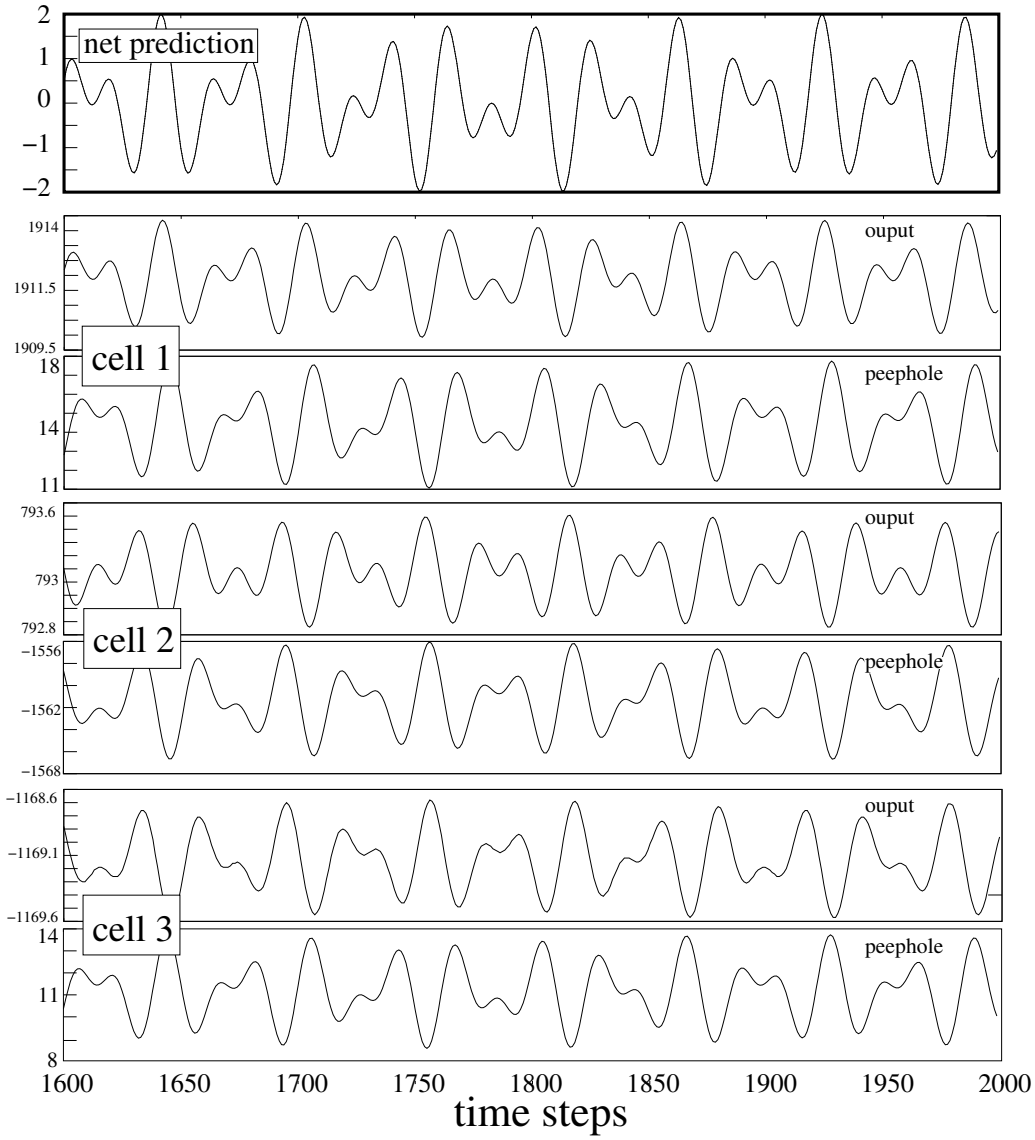

Figure 8: **Internal representation of two-sine function.** The upper graph shows the output of a PI-Evolino LSTM network with three cells predicting the two-sine function. The three pairs of graphs below show the output (upper) and output peephole (lower) values of each cell in the network multiplied by their respective (pseudoinverse-generated) output weight. These six signals are added to generate the signal in the upper graph.

4 Concluding Remarks

The human brain is a biological, learning RNN. Previous successes with artificial RNNs have been limited by problems overcome by the LSTM architecture. Its algorithms for shaping not only the linear but also the nonlinear parts allow LSTM to learn to solve tasks unlearnable by standard feed-forward nets, Support Vector Machines, Hidden Markov Models, and previous RNNs. Previous work on LSTM has focused on gradient-based G-LSTM [1–3, 8, 11, 29, 38]. Here we introduced the novel Evolino class of supervised learning algorithms for such nets that overcomes certain problems of gradient-based RNNs with local minima. Successfully tested instances with hidden coevolving recurrent neurons use either the pseudoinverse to minimize the MSE of the linear mapping from hidden units to outputs (PI-Evolino), or quadratic programming to maximize the margin. The latter yields the first evolutionary recurrent SVMs or RSVMs, trained by an Evolino variant called Evoke.

In the experiments of our pilot study, RSVMs generally performed better than G-LSTM and previous gradient-based RNNs, but typically worse than PI-Evolino. One possible reason for this could be that the kernel mapping of the SVM component induces a more rugged fitness landscape that makes evolutionary search harder.

All of the evolved networks were comparatively small, usually featuring less than 3,000 weights. On the other hand, for large data sets such as those used in speech recognition we typically need much larger LSTM networks with on the order of 100,000 weights [8]. On such problems, we have so far generally obtained better results with G-LSTM than with Evolino. This seems to reaffirm the heuristic that evolution of large parameter sets is often harder than gradient search in such sets. Currently it is unclear when exactly to favor one over the other. Future work will explore hybrids combining G-LSTM and Evolino in an attempt to leverage the best of both worlds. We will also explore ways of improving the performance of Evoke, including the coevolution of SVM kernel parameters.

We have barely tapped the set of possible applications of our new approaches: in principle, any learning task that requires some sort of adaptive short-term memory may benefit.

References

[1] F. A. Gers and J. Schmidhuber. LSTM recurrent networks learn simple context free and context sensitive languages. *IEEE Transactions on Neural Networks*, 12(6):1333–1340, 2001.

[2] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with LSTM. *Neural Computation*, 12(10):2451–2471, 2000.

[3] F. A. Gers, N. Schraudolph, and J. Schmidhuber. Learning precise timing with LSTM recurrent networks. *Journal of Machine Learning Research*, 3:115–143, 2002.

[4] F. Gomez and R. Miikkulainen. Solving non-Markovian control tasks with neuroevolution. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, Denver, CO, 1999. Morgan Kaufmann.

[5] F. Gomez and J. Schmidhuber. Evolving modular fast-weight networks for control. In W. Duch, J. Kacprzyk, E. Oja, and S. Zadrozny, editors, *Proceedings of the Fifteenth International Conference on Artificial Neural Networks: ICANN-05*, pages 383–389, 2005.

[6] F. J. Gomez. *Robust Nonlinear Control through Neuroevolution*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, August 2003. Technical Report AI-TR-03-303.

[7] F. J. Gomez and J. Schmidhuber. Co-evolving recurrent neurons learn deep memory pomdps. In *Proceedings of the Genetic Evolutionary Computation Conference (GECCO-05)*, Berlin; New York, 2005. Springer-Verlag.

[8] A. Graves and J. Schmidhuber. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, 18:602–610, 2005.

[9] S. Hochreiter. Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München, 1991.

[10] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In S. C.

Kremer and J. F. Kolen, editors, *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press, 2001.

[11] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[12] S. Hochreiter 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 9 (NIPS 9)*, pages 473–479. MIT Press, 1997.

[13] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[14] K. Ishii, T. van der Zant, V. Bečanović, and P. G. Plöger. Identification of motion with echo state network. In *Proc. IEEE Oceans04*, pages 1205–1230, Kobe, Japan, 2004. IEEE.

[15] H. Jaeger. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, 304:78–80, 2004.

[16] H. Jaeger. http://www.faculty.iu-bremen.de/hjaeger/courses/seminarspring04/esnstandardslides.pdf, 2004.

[17] W. Maass, T. Natschläger, and H. Markram. A fresh look at real-time computation in generic recurrent neural circuits. Technical report, Institute for Theoretical Computer Science, TU Graz, 2002.

[18] M. C. Mackey and L. Glass. Oscillation and chaos in physiological control systems. *Science*, 197:287–289, 1977.

[19] E. P. Maillard and D. Gueriot. RBF neural network, basis functions and genetic algorithms. In *IEEE International Conference on Neural Networks*, pages 2187–2190, Piscataway, NJ, 1997. IEEE.

[20] O. Miglino, H. Lund, and S. Nolfi. Evolving mobile robots in simulated and real environments. *Artificial Life*, 2(4):417–434, 1995.

[21] G. Miller, P. Todd, and S. Hedge. Designing neural networks using genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 379–384. Morgan Kauffman, 1989.

[22] D. E. Moriarty. *Symbiotic Evolution of Neural Networks in Sequential Decision Tasks*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 1997. Technical Report UT-AI97-257.

[23] D. E. Moriarty and R. Miikkulainen. Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22:11–32, 1996.

[24] S. Mukherjee, E. Osuna, and F. Girosi. Nonlinear prediction of chaotic time series using support vector machines. In J. Principe, L. Giles, N. Morgan, and E. Wilson, editors, *IEEE Workshop on Neural Networks for Signal Processing VII*, page 511. IEEE Press, 1997.

[25] K. Müller, A. Smola, G.Rätsch, B. Schölkopf, J. Kohlmorgen, and V. Vapnik. Predicting time series with support vector machines. In W. G. et al., editor, *Proceedings of the Seventh International Conference on Artificial Neural Networks: ICANN-97*, pages 999–1004. Springer-Verlag, 1997.

[26] S. Nolfi, D. Floreano, O. Miglino, and F. Mondada. How to evolve autonomous robots: Different approaches in evolutionary robotics. In R. A. Brooks and P. Maes, editors, *Fourth International Workshop on the Synthesis and Simulation of Living Systems (Artificial Life IV)*, pages 190–197. MIT, 1994.

[27] B. A. Pearlmutter. Gradient calculations for dynamic recurrent neural networks: A survey. *IEEE Transactions on Neural Networks*, 6(5):1212–1228, 1995.

[28] R. Penrose. A generalized inverse for matrices. In *Proceedings of the Cambridge Philosophy Society*, volume 51, pages 406–413, 1955.

[29] J. A. Pérez-Ortiz, F. A. Gers, D. Eck, and J. Schmidhuber. Kalman filters improve LSTM network performance in problems unsolvable by traditional recurrent nets. *Neural Networks*, 16(2):241–250, 2003.

[30] M. A. Potter and K. A. De Jong. Evolving neural networks with collaborative species. In *Proceedings of the 1995 Summer Computer Simulation Conference*, 1995.

[31] I. Rechenberg. Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Dissertation, 1971. Published 1973 by Fromman-Holzboog.

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

[33] D. E. Rumelhart and J. L. McClelland, editors. *Parallel Distributed Processing*, volume 1. MIT Press, 1986.

[34] J. Salomon, S. King, and M. Osborne. Framewise phone classification using support vector machines. In *Proceedings International Conference on Spoken Language Processing*, Denver, 2002.

[35] J. Schmidhuber. Dynamische neuronale Netze und das fundamentale raumzeitliche Lernproblem. Dissertation, Institut für Informatik, Technische Universität München, 1990.

[36] J. Schmidhuber. A fixed size storage $O(n^3)$ time complexity learning algorithm for fully recurrent continually running networks. *Neural Computation*, 4(2):243–248, 1992.

[37] J. Schmidhuber, M. Gagliolo, D. Wierstra, and F. Gomez. Evolino for recurrent support vector machines. In *Proc. ESANN'06, in press*, 2006.

[38] J. Schmidhuber, F. Gers, and D. Eck. Learning nonregular languages: A comparison of simple recurrent networks and LSTM. *Neural Computation*, 14(9):2039–2041, 2002.

[39] J. Schmidhuber, S. Hochreiter, and Y. Bengio. Evaluating benchmark problems by random guessing. In S. C. Kremer and J. F. Kolen, editors, *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press, 2001.

[40] J. Schmidhuber, D. Wierstra, and F. J. Gomez. Evolino: Hybrid neuroevolution / optimal linear search for sequence prediction. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*. Morgan Kaufman, 2005, in press.

[41] H. P. Schwefel. Numerische Optimierung von Computer-Modellen. Dissertation, 1974. Published 1977 by Birkhäuser, Basel.

[42] H. P. Schwefel. *Evolution and Optimum Seeking*. Wiley Interscience, 1995.

[43] H. Shimodaira, K.-I. Noma, M. Nakai, and S. Sagayama. Dynamic time-alignment kernel in support vector machine. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems 14*, Cambridge, MA, 2002. MIT Press.

[44] H. T. Siegelmann and E. D. Sontag. Turing computability with neural nets. *Applied Mathematics Letters*, 4(6):77–80, 1991.

[45] K. Sims. Evolving virtual creatures. In A. Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 1994)*, Computer Graphics Proceedings, Annual Conference, pages 15–22. ACM SIGGRAPH, ACM Press, jul 1994. ISBN 0-89791-667-0.

[46] J. Suykens and J. Vandewalle. Recurrent least squares support vector machines. *IEEE Transactions on Circuits and Systems-I*, 47(7):1109–1114, July 2000.

[47] T. van der Zant, V. Becanovic, K. Ishii, H.-U. Kobialka, and P. G. Plöger. Finding good echo state networks to control an underwater robot using evolutionary computations. In *Proceedings of the 5th IFAC symposium on Intelligent Autonomous Vehicles (IAV04)*, Lisboa, Portugal, 2004.

[48] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer, New York, 1995.

[49] P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

[50] P. J. Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1, 1988.

[51] D. Wierstra, F. J. Gomez, and J. Schmidhuber. Modeling non-linear dynamical systems with Evolino. In *Proc. GECCO 2005, Washington, D. C.*, 2005. GECCO best paper award in Learning Classifier Systems and Other Genetics-Based Machine Learning.

[52] R. J. Williams. Complexity of exact gradient computation algorithms for recurrent neural networks. Technical Report Technical Report NU-CCS-89-27, Boston: Northeastern University, College of Computer Science, 1989.

[53] B. M. Yamauchi and R. D. Beer. Sequential behavior and learning in evolved dynamical neural networks. *Adaptive Behavior*, 2(3):219–246, 1994.

[54] X. Yao. A review of evolutionary artificial neural networks. *International Journal of Intelligent Systems*, 4:203–222, 1993.