Evolino: Hybrid Neuroevolution / Optimal Linear Search for Sequence Learning

Jürgen Schmidhuber[1,2], **Daan Wierstra**[2], and **Faustino Gomez**[2]
{juergen, daan, tino}@idsia.ch

[1] TU Munich, Boltzmannstr. 3, 85748 Garching, München, Germany

[2] IDSIA, Galleria 2, 6928 Lugano, Switzerland

Abstract

Current Neural Network learning algorithms are limited in their ability to model non-linear dynamical systems. Most supervised gradient-based recurrent neural networks (RNNs) suffer from a vanishing error signal that prevents learning from inputs far in the past. Those that do not, still have problems when there are numerous local minima. We introduce a general framework for sequence learning, EVOlution of recurrent systems with LINear outputs (Evolino). Evolino uses evolution to discover good RNN hidden node weights, while using methods such as linear regression or quadratic programming to compute optimal linear mappings from hidden state to output. Using the Long Short-Term Memory RNN Architecture, the method is tested in three very different problem domains: 1) context-sensitive languages, 2) multiple superimposed sine waves, and 3) the Mackey-Glass system. Evolino performs exceptionally well across all tasks, where other methods show notable deficiencies in some.

1 Introduction

Real world non-linear dynamical systems are black-box in nature: it is possible to observe their input/output behavior, but the internal mechanism that generates this behavior is often unknown. Modeling such systems to accurately predict their behavior is a huge challenge with potentially far-reaching impact on areas as broad as speech processing/recognition, financial forecasting, and engineering.

Artificial Neural Networks with feedback connections or Recurrent Neural Networks (RNNs; [Werbos, 1990; Robinson and Fallside, 1987; Williams and Zipser, 1989]) are an attractive formalism for non-linear modeling because of their ability, in principle, to approximate any dynamical system with arbitrary precision [Siegelmann and Sontag, 1991]. However, training RNNs with standard gradient descent algorithms is only practical when a short time window (less than 10 time-steps) is sufficient to predict the correct system output. For longer temporal dependencies, the gradient vanishes as the error signal is propagated back through time so that network weights are never adjusted correctly to account for events far in the past [Hochreiter *et al.*, 2001].

Echo State Networks (ESNs; [Jaeger, 2004a]) deal with temporal dependencies by simply ignoring the gradients associated with hidden neurons. Composed primarily of a large pool of neurons (typically hundreds or thousands) with fixed random weights, ESNs are trained by computing a set of weights analytically 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 adjusting the output weights. This simple approach is currently the title holder in the Mackey-Glass time-series benchmark, improving on the accuracy of all other methods by as much as three orders of magnitude [Jaeger, 2004a].

The drawback of ESNs, of course, is that the only truly computationally powerful, nonlinear part of the net does not learn at all. This means that on some seemingly simple tasks, such as generating multiple superimposed sine waves, the method fails. According to our experience, it is also not able to solve a simple context-sensitive grammar task [Gers and Schmidhuber, 2001]. Moreover, because ESNs use such a large number of processing units, they are prone to overfitting, i.e. poor generalization.

One method that adapts *all* weights and succeeds in using gradient information to learn long-term dependencies is Long Short-Term Memory (LSTM; [Hochreiter and Schmidhuber, 1997; Gers and Schmidhuber, 2001]). LSTM uses a specialized network architecture that includes linear *memory cells* that can sustain their activation indefinitely. The cells have input and output gates that learn to open and close at appropriate times either to let in new information from outside and change the state of the cell, or to let activation out to potentially affect other cells or the network’s output. The cell structure enables LSTM to use gradient descent to learn dependencies across almost arbitrarily long time spans. However, in cases where gradient information is of little use due to numerous local minima, LSTM becomes less competitive.

An alternative approach to training RNNs is neuroevolution [Yao, 1999]. Instead of using a single neural network, the space of network parameters is searched in parallel using the principle of natural selection. A population of *chromosomes* or strings encoding, for instance, network weight values and connectivity is evaluated on the problem, and each chromosome is awarded a *fitness* value that quantifies its relative performance. The more highly fit chromosomes are combined by exchanging substrings (*crossover*) and by randomly changing some values (*mutation*), producing new so-

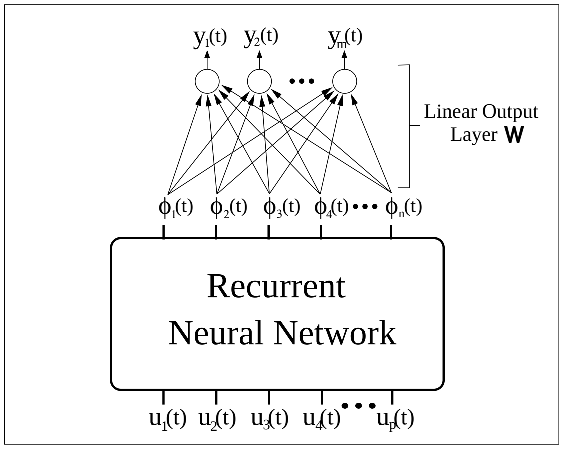

Figure 1: **Evolino network.** A recurrent neural network receives sequential inputs $u(t)$ and produce the vector $(\phi_1, \phi_2, \ldots, \phi_n)$ at every time step t. These values are linearly combined with the weight matrix W to yield the network's output vector $y(t)$. While the RNN is evolved,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

lutions that hopefully improve upon the existing population. This approach has been very effective in solving continuous, partially observable reinforcement learning tasks where the gradient is not directly available, outperforming conventional methods (e.g. Q-learning, SARSA) on several difficult learning benchmarks [Moriarty and Miikkulainen, 1996; Gomez and Miikkulainen, 1999]. However, neuroevolution is rarely used for supervised learning tasks such as time series prediction because it has difficulty fine-tuning solution parameters (e.g. network weights), and because of the prevailing maxim that gradient information should be used when it is available.

In this paper, we present a novel framework called EVOlution of recurrent systems with LINear outputs (Evolino) that combines elements of the three aforementioned methods, to address the disadvantages of each, extending ideas proposed for feedforward networks of radial basis functions (RBFs) [Maillard and Gueriot, 1997]. Applied to the LSTM architecture, Evolino can solve tasks that ESNs cannot, and achieves higher accuracy in certain continuous function generation tasks than conventional gradient descent RNNs, including gradient-based LSTM.

Section 2 explains the basic concept of Evolino and describes in detail the specific implementation used in this paper. Section 3 presents our experiments using Evolino in three different domains: context-sensitive grammars, continuous function generation, and the Mackey-Glass time-series. Section 4 and 5 discuss the algorithm and the experimental results, and summarize our conclusions.

2 The Evolino Framework

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. 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 [Vapnik, 1995] 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 $n \times \sum_i^k l^i$ matrix Φ. Associated with each $\phi^i(t)$, is a *target* row vector d^i in 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 column 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.

Neuroevolution is normally applied to reinforcement learning tasks where correct network outputs (i.e. targets) are not known *a priori*. Here we use 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.

In this study, Evolino is instantiated using Enforced Sub-Populations to evolve LSTM networks. The next sections describe ESP and LSTM, and the details of how they are combined within the Evolino framework.

2.1 Enforced Subpopulations

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). Evolution in ESP proceeds as follows:

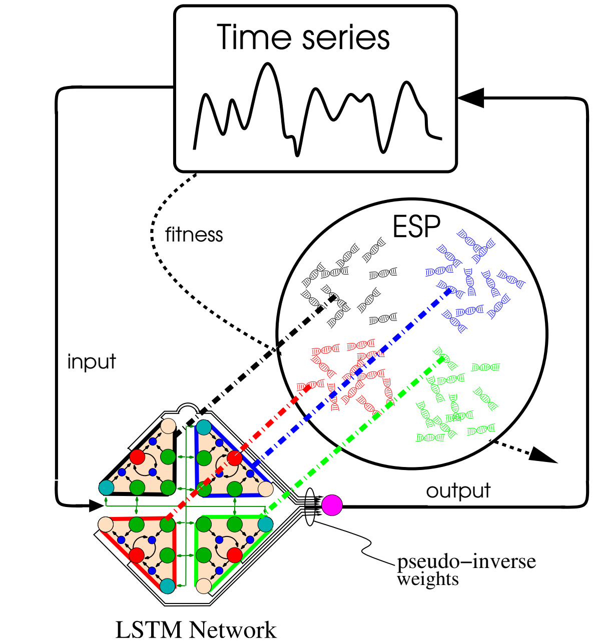

Figure 2: **Enforced SubPopulations (ESP).** The population of neurons is segregated into subpopulations. Networks are formed by randomly selecting one neuron from each subpopulation. A neuron accumulates a fitness score by adding the fitness of each network in which it participated. The best neurons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

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, output, 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.
3. Recombination: For each subpopulation the neurons are ranked by fitness, and the top quartile is recombined using 1-point crossover and mutated using Cauchy distributed noise to create new neurons that replace the lowest-ranking half of the subpopulation.
4. Repeat the Evaluation–Recombination cycle until a sufficiently fit network is found.

ESP searches the space of networks 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; [Moriarty and Miikkulainen, 1996]) 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. This allows ESP to evolve recurrent networks, where SANE could not.

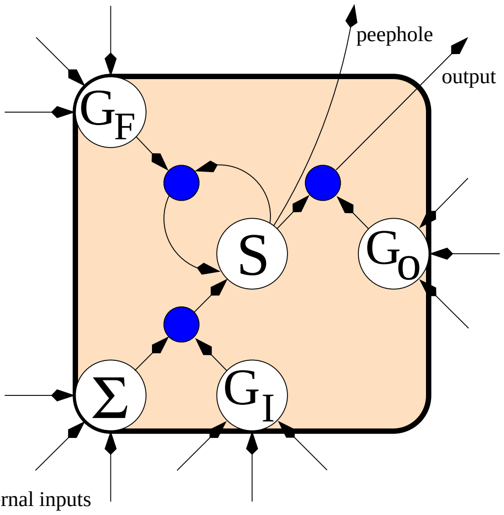

Figure 3: **Long Short-Term Memory.** The figure shows an LSTM *memory cell*. The cell has an internal state S together with a forget gate (G_F) that determines how much the state is attenuated at each time step. The input gate (G_I) controls access to the cell by the external inputs that are summed 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.

If the performance of ESP does not improve for a predetermined number of generations, a technique called *burst mutation* 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 the *memory cell* that is capable of maintaining its activation indefinitely (figure 3). 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:

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

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 3):

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

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

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

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:

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

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 ESP with LSTM 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 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 (equation 6) and the cell itself (equation 4) receive input from outside the cell and the other cells. ESP, as described in section 2.1, normally uses crossover to recombine neurons. However, for the present Evolino variant, where fine local search is desirable, ESP uses only mutation. The top quarter of the chromosomes in each subpopulation are duplicated and the copies are mutated by adding Cauchy noise to all of their weight values.

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 [Penrose, 1955]—compare [Maillard and Gueriot, 1997] for an application to feedforward RBF nets. The vector $\phi(t)$ consists of both the cell outputs, c_i (equation 5), and their internal states, s_i (equation 3), 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.

Training data	Gradient LSTM	Evolino LSTM
1..10	1..28	1..53
1..20	1..66	1..95
1..30	1..91	1..355
1..40	1..120	1..804

Table 1: **Generalization results for the $a^n b^n c^n$ language.** The table compares Evolino-based LSTM to Gradient-based LSTM. The left column shows the set of legal strings used to train each method. The other columns show the set of strings that each method was able to accept after training. The result for LSTM with gradient descent are from [Gers and Schmidhuber, 2001]. Averages of 20 runs.

3 Experimental Results

Experiments were carried out on three test problems: context-sensitive languages, multiple superimposed sine waves, 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. For a more detailed description of setups used in these two problems, and further experiments, we direct the reader to [Wierstra *et al.*, 2005]. The Mackey-Glass system was selected to compare Evolino with ESNs, the reference method on this widely used time series benchmark.

3.1 Context-Sensitive Grammars

Learning to recognize context-sensitive languages is a difficult and often intractable problem for standard RNNs because it can require unlimited memory. For instance, recognizing the language $a^n b^n c^n$ (i.e. strings where the number of as, bs, and cs is equal) entails counting the number of consecutive as, bs, and cs, and potentially having to remember these quantities until the whole string has been read. Gradient-based LSTM has previously been used to learn $a^n b^n c^n$, so here we compare the results in [Gers and Schmidhuber, 2001] to those of Evolino-based LSTM.

Four sets of 20 simulations were run each using a different training set of legal strings, $\{a^n b^n c^n\}, n = 1..N$, where N was 10, 20, 30, and 40. Symbol strings were presented to the networks, one symbol at a time. The networks had 4 input units, one for each possible symbol: S for start, a, b, and c. An input is set to 1.0 when the corresponding symbol is observed, and -1.0 when it is not present. At every time step, the network predicts what symbols could come next, a, b, c, and the termination symbol T, by activating its 4 output units. An output unit is considered to be "on" if its activation is greater than 0.0.

ESP evolved LSTM networks with 4 memory cells, weights randomly initialized to values between -0.1 and 0.1. The Cauchy noise parameter α for both mutation and burst mutation was set to 0.00001, i.e. 50% of the mutations is kept within this bound. Evolution was terminated after 50 generations, after which the best network in each simulation was tested.

The results are summarized in Table 1. Evolino-based LSTM learns in approximately 3 minutes on average, but, more importantly, it is able to generalize substantially better than gradient-based LSTM.

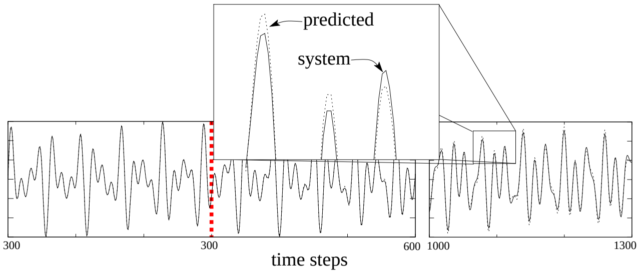

Figure 4: **Performance of Evolino on the triple superimposed sine wave task.** The plot show the behavior of a typical network produced after 50 generations (3000 evaluations). The first 300 steps (the data-points left of the vertical dashed line) were used as training data, the rest must be predicted by the network during testing. Time-steps above 300 show the network predictions (dashed curve) during testing plotted against the correct system output (solid curve). The inset is a magnified detail that more clearly shows the two curves.

3.2 Multiple Superimposed Sine Waves

Jaeger [Jaeger, 2004b] reports that Echo State Networks are unable to learn functions composed of multiple superimposed oscillators. Specifically, functions like $sin(0.2x) + sin(0.311x)$, in which the individual sines have the same amplitude but their frequencies are not multiples of each other. ESNs have difficulty solving this problem because the dynamics of all the neurons in the ESN "pool" are coupled, whereas truly solving the task requires an internal representation of multiple attractors due to the non-periodic behavior of the function.

We evolved networks with 10 memory cells to predict the aforementioned double sine, $sin(0.2x) + sin(0.311x)$, and network with 15 cells for a more complex triple sine, $sin(0.2x) + sin(0.311x) + sin(0.42x)$. Evolino used the same parameter settings as in the previous section, except that backprojection was used (see section 2.3). Networks for both tasks were evolved for 50 generations to predict the first 300 time steps of each function, and then tested on data points from time-steps $300..600$.

The average summed squared error over the training set was 0.011 for the double sine and 0.2 for the triple sine. The average error over the test set was 0.044 and 1.58, respectively. These error levels are barely visible out to time-step 600. Figure 4 shows the behavior of one of the triple sine wave Evolino networks out to time-step 1300. The magnified inset illustrates how even beyond 3 times the length of the training set, the network still makes very accurate predictions.

3.3 Mackey-Glass Time-Series Prediction

The Mackey-Glass system (MGS; [Mackey and Glass, 1977]) 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 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 [Jaeger, 2004a]. Networks were trained on the first 3000 time steps of the series using a "washout time" of 100 steps. During the washout time the vectors $\phi(t)$ are not collected for calculating the pseudo-inverse.

We evolved networks with 30 memory cells for 200 generations, and a Cauchy noise α of 10^{-7}. A bias input of 1.0 was added to the network, and the backprojection values were scaled by a factor of 0.1. For testing, the outputs were clamped to the correct targets for the first 3000 steps, after which the network backprojected its own prediction for the next 84 steps[1]. The cell input (equation 4) was squashed with the $tanh$ function. The average NRMSE$_{84}$ for Evolino with 30 cells over the 15 runs was 1.9×10^{-3} compared to $10^{-4.2}$ for ESNs with 1000 neurons [Jaeger, 2004a]. The Evolino results are currently the second-best reported so far.

Figure 5 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 complex functions very quickly; in this case within approximately 3 minutes of CPU time.

4 Discussion

The real strength of the Evolino framework is its generality. Across different classes of sequence prediction problems, it was able to compete with the best known methods and convincingly outperform them in several cases. In particular, it generalized much better than gradient-based LSTM in the context-sensitive grammar task, and it solved the superimposed sine wave task, which ESNs cannot. These results suggest that Evolino could be widely applicable to modeling complex processes that have both discrete and continuous properties, such as speech.

Evolino avoids the problem of vanishing gradient and local minima normally associated with RNN training by searching the space of networks in parallel through evolution. Furthermore, by using LSTM memory cells, Evolino searches in a weight space that is already biased toward extracting, retaining, and relating discrete events that may be very far apart in time. And, by borrowing the idea of linear regression from ESNs, Evolino is capable of making very precise predictions in tasks like the Mackey-Glass benchmark.

Apart from its versatility, another advantage of Evolino over ESNs is that it produces more parsimonious solutions. ESNs have large pools of neurons that are more likely to overfit the data. Evolino networks can be made much smaller and, therefore, potentially more general, less susceptible to noise, and more easily comprehensible by, for instance, RNN rule extraction techniques.

Evolino is a template that can be instantiated by plugging in (1) alternative analytical methods for computing optimal linear mappings to the outputs, given the hidden state, (2) different neuroevolution algorithms, and (3) various recurrent network architectures. In particular, our implementation used

[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.

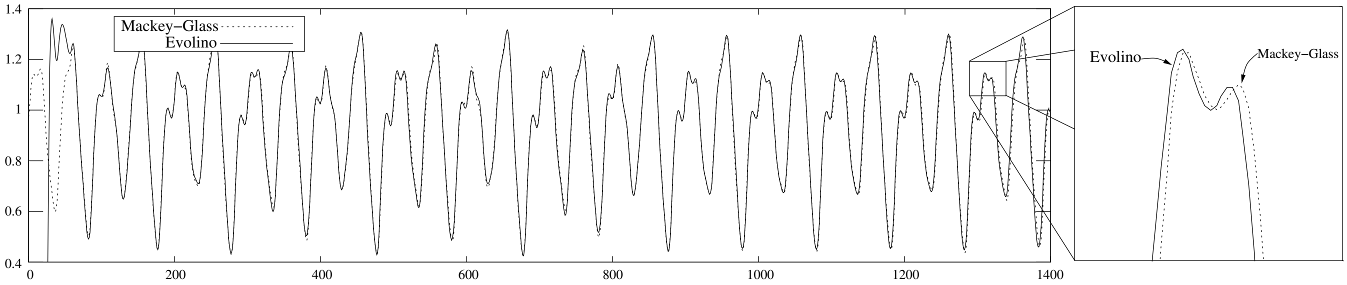

Figure 5: **Performance of Evolino on the Mackey-Glass time-series.** The plot shows both the Mackey-Glass system and the prediction made by a typical Evolino-based LSTM network evolved for 50 generations. The obvious difference between the system and the prediction during the first 100 steps is due to the washout time. The inset shows a magnification more clearly showing the deviation between the two curves.

mean squared error and linear regression, but we could as well use the maximum margin optimality criterion [Vapnik, 1995] and use quadratic programming to find optimal linear mappings from hidden state to sequence classifications, obtaining a hitherto unknown species of *sequential* support vector machines.

We could also use neuroevolution methods that evolve network topology as well, so that network complexity is also determined through genetic search. Other RNNs, such as higher-order networks could be used instead of LSTM. Generalizations to nonlinear readout mechanisms (e.g., nonlinear neural networks) with gradient-based search are obvious. We may also start training LSTM by Evolino, then fine-tune by traditional pure gradient search.

Future work will further explore this space of possible implementations to provide potentially even more powerful predictors, classifiers, and sequence generators.

5 Conclusion

We introduced EVOlution of recurrent systems with LINear outputs (Evolino), a general framework that combines evolution of recurrent neural networks and analytical linear methods to solve sequence learning tasks. The implementation of Evolino in this paper combined the pseudo-inverse and Enforced Subpopulations algorithms to search a space of Long-Short Term Memory networks. This yielded a versatile method that can solve both tasks that require long-term memory of discrete events such as context-sensitive languages, and continuous time-series such as the Mackey-Glass benchmark and multiple superimposed sine waves.

Acknowledgments

This research was partially funded by CSEM Alpnach and the EU MindRaces project, FP6 511931.

References

[Gers and Schmidhuber, 2001] 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.

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

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

[Hochreiter *et al.*, 2001] 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.

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

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

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

[Maillard and Gueriot, 1997] 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.

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

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

[Robinson and Fallside, 1987] 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.

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

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

[Werbos, 1990] P. Werbos. Backpropagation through time: what does it do and how to do it. In *Proceedings of IEEE*, volume 78, pages 1550–1560, 1990.

[Wierstra *et al.*, 2005] Daan Wierstra, Juergen Schmidhuber, and Faustino Gomez. Modeling non-linear dynamical systems with evolino. To appear in *Proceedings of the Genetic Evolutionary Computation Conference (GECCO-05)*. Springer-Verlag, Berlin; New York, 2005.

[Williams and Zipser, 1989] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent networks. *Neural Computation*, 1(2):270–280, 1989.

[Yao, 1999] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.