Evolving memory cell structures for sequence learning

Justin Bayer, Daan Wierstra, Julian Togelius and Jürgen Schmidhuber

IDSIA, Galleria 2, 6928 Manno-Lugano, Switzerland
{justin, daan, julian, juergen}@idsia.ch

Abstract. The best recent supervised sequence learning methods use gradient descent to train networks of miniature nets called memory cells. The most popular cell structure seems somewhat arbitrary though. Here we optimize its topology with a multi-objective evolutionary algorithm. The fitness function reflects the structure's usefulness for learning various formal languages. The evolved cells help to understand crucial structural features that aid sequence learning.

1 Introduction

The problem of sequence learning is to learn the underlying function of a dynamic system, so as to be able either to produce the next step in a sequence produced by the system (sequence prediction), or correctly classify a sequence produced by the system (sequence classification). Sequence learning is tremendously important in various applications, e.g. stock market prediction and speech and handwriting recognition.

Neural networks are among the best tools available for general sequence learning. Most often, the *sliding time window* approach is used, where a finite subsequence is presented to a feedforward neural network. This approach is ultimately limited by the size of the time window. In the last decade, sequence prediction using *recurrent* neural networks has become has attracted some attention because of the simplicity and potential power of RNNs. Here, the whole sequence is presented to the network, which is then trained through backpropagation through time (BPTT) [15]. However, there are some serious practical limitations of most types of RNNs due to their inability to capture long-term time dependencies. They suffer from the problem of *vanishing gradient* [7], the fact that the gradient signal vanishes as the error signal is propagated back through time. Because of this, events more than 10 time steps apart can typically not be related.

1.1 Dealing with vanishing gradient: LSTM

One method purposely designed to avoid this problem is Long Short-Term Memory (LSTM [8]), which is a special RNN architecture capable of capturing long term time dependencies. The defining feature of this architecture is that it consists of a number of *memory cells*, which can be used to store activations arbitrarily long. Access to the memory cell is *gated* by units that learn to open or

close depending on the context. The memory cell's internal structure consists of a number of connected computational units, including the sigmoid function, the tanh function, and the gating function, which are connected in a graph structure. The fact that these units are differentiable ensures the memory cell as a whole can be used in conjunction with BPTT using the chain rule as a connecting principle. LSTM networks have been shown to outperform other RNNs on numerous time series requiring the use of deep memory [13].

LSTM, unlike conventional RNNs, has been shown to be able to capture long-term time dependencies, learn precise timing, and generalize well on examples of both context-free and context-sensitive languages such as $a^n b^n$ and $a^n b^n c^n$, respectively, whereas normal RNNs completely failed to capture the underlying structure of the problem [12].

Interestingly, the development of LSTM was incremental (see figure 1). First, the concept of an internal *state* was introduced, guarded by input and output gates [8]. A time delay connection from the state to itself with weight one ensured that the state retained its value, unless the input gate was opened. Then, the concept of a *forget gate* was introduced, which modulates the state's self-connection and enables precise timing abilities [3]. Finally, *peepholes* were devised, which are direct connections from the state to all gates [4]. This final step enabled LSTM to learn the underlying structure of the context-sensitive language $a^n b^n c^n$ up to hundreds of time steps using just 10 sample sequences for training [2]. LSTM has recently been shown to perform excellently on many tasks, including speech processing and handwriting recognition (e.g. see [10]).

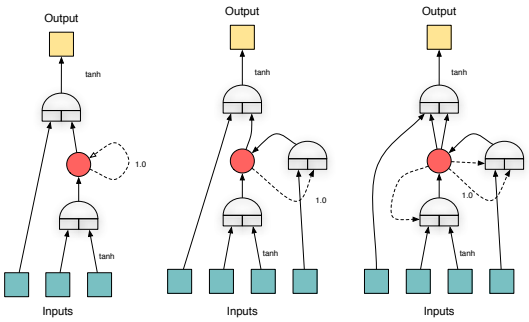

Fig. 1. Three versions of the LSTM cell. Input units are in teal, output units in yellow. The gate units are shown as a half circle with as the output part and two different squares as inputs. The time delayed connection is dashed, and the red circle is the state unit. The second version of the LSTM cell adds a forget gate, and the third version adds peepholes.

The incremental design evolution of the LSTM cell outlined above, taken together with its seemingly somewhat arbitrary structure, suggests that the de-

velopment of the LSTM could be retraced with artificial evolution, and that the LSTM design could even be bettered using the same means. In particular, we propose to use techniques introduced to evolve neural network topologies to evolve the internal structure of LSTM-like memory cells, using the sequence learning capability of networks of such cells as fitness functions.

1.2 Evolving neural topologies

A large body of work exists where evolutionary algorithms are used to create and optimize topologies of neural networks. Topologies have been evolved for a number of different purposes, including direct function approximation (without subsequent learning), reinforcement learning, and capacity to be trained through gradient descent methods.

A core distinction can be made between *indirect* or *generative* approaches to topology evolution, and *direct* approaches. The former try to replicate nature's ability to encode complex phenotypes (e.g. human brains) with vastly simpler genotypes (e.g. human DNA), using graph rewriting systems or models of biological processes [9, 6]. Apparently, the promise of scalability motivating these approaches have so far not been realized. The latter category, which includes the empirically successful NEAT algorithm [14], instead encode the structure directly in the genome. A central concept of NEAT is complexification; a network starts out small, but the mutation operators can add new connections as well as split existing connections to insert new neurons. The algorithm used in this paper has similarities to NEAT, but is simplified in order to fit our needs.

Usually, the weights of the neural connections are evolved at the same time as the topology. However, Whiteson [16] evolved network topologies without weights, with a fitness function based on their ability to be used as function approximators for TD-learning. Similarly, in this paper we do not evolve connection weights, but use fitness functions based on capacity for sequence learning.

1.3 This paper: Evolving cell structures

The purpose of this paper is to investigate the space of architectural alternatives to LSTM and understand the structural features promoting successful sequence learning through evolving structures of memory cells so as to optimize their sequence learning capability. We view each memory cell as a miniature neural network. The structure of the cell equates the topology of the network, and a neuron in the neural network a unit in the cell. We then use a NEAT-inspired direct topology evolution algorithm to evolve this structure.

The fitness functions for structures are based on how well *networks of cells with the tested structure* can learn different sequences using gradient descent. (Note that connection weights are reset between fitness evaluations; evolution is thus *not* "Lamarckian".) As it is crucial that all cell structures can be trained by gradient descent, we constrain the structures to be direct acyclic graphs (*DAGs*) of differentiable units, plus recurrent connections: time-delay connections which break the DAG property but only propagate activations between time steps.

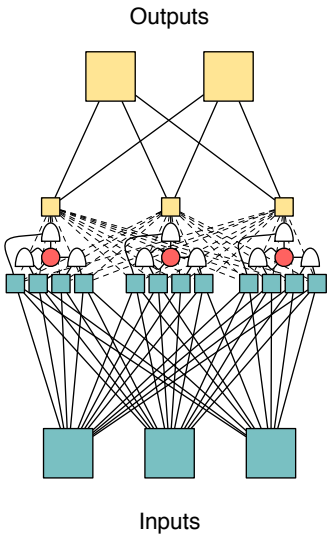

Fig. 2. A network constructed with hidden layer of three LSTM cells. The recurrent connections from the hidden layer to itself, necessary for the cells to communicate with each other, are shown as dashed.

We start with evolving cells capable of learning simple versions of the problems; once these problems can be learnt satisfactorily, we increase the complexity of the problem, a practice known as incremental evolution [5]. So as not to overspecialize and develop cell structures only capable of learning solutions to one type of problem, we test each cell on two problems. Using the learning capability on each problem as a separate fitness measure means that we pose cell structure evolution as a *multiobjective* optimization problem, requiring the use of a multiobjective evolutionary algorithm (*MOEA*) in our case the *NSGA-II* [1].

2 Methods

2.1 Memory cell representation

A memory cell structure is a set of computational units and a graph connecting them to each other. Connections between units possess a flag indicating whether the connection is time delayed and another flag indicating whether the connection is parameterized (i.e. has a trainable *weight*) or has a fixed parameter of 1.0. The former case is called a *linear connection* while the latter is called an *identity connection*. There are several types of computational nodes: linear, sigmoid, the hyperbolic tangent and the 'gate' unit, each having its own transfer function.

– The linear node takes input x and produces output $id(x) = x$.

- The sigmoid node takes input x, and produces output $\sigma(x) = 1/(1 + e^{-x})$.
- The tanh node is the familiar hyperbolic tangent.
- The most interesting type of node used in this paper, however, is the *gating* unit that was first introduced in the LSTM cell. Its structure could be thought of as a continuous version of the `if ... then ...` statement, and has two inputs: one condition and one signal. Its transfer function is $g(x_1, x_2) = \sigma(x_1)x_2$. It is this unit type that enables LSTM's internal state to open and close to incoming signals, depending on the context.

All units have two additional flags: one indicating whether a unit is an input unit to the cell, i.e. receives input from outside the cell, and one indicating whether the unit is an output unit, connecting to other cells and network outputs.

2.2 Evolutionary algorithm

We used the *NSGA-II* multiobjective evolutionary algorithm (MOEA), as it is one of the most widely used MOEAs and known for robust performance under diverse conditions [1]. A population size of 100 was used, and initial populations consisted either of random cell structures of varying size or of LSTM cells. For simplicity, no recombination was used; mutation was the only variation operator.

A cell structure is mutated through applying mutations from the list below a geometrically distributed number of times. The expected amount of mutations is given by $E_M = \sum_{m \in M} \frac{1}{1-\pi[m]}$, where $\pi[m]$ is the probability of each mutation type. The probabilities used in our experiments are given in parentheses in the following list of available mutations; these probabilities were chosen carefully in order to prevent bloating of the structure. If any mutation breaks the DAG property by making the structure cyclic, that mutation is simply rolled back.

- *Add unit* A random connection is split into two parts with a new linear unit in between. ($\pi[\cdot] = 0.1$)
- *Add gate unit* A unit is added as in *Add unit* but also assigned the gate transfer function. Its second input is connected to a random unit. ($\pi[\cdot] = 0.2$)
- *Add connection* Two units are randomly chosen and connected by an identity connection which is not time delayed. ($\pi[\cdot] = 0.15$)
- *Add recurrent connection* Two units are chosen in a way that connecting them would break the DAG property. The units are connected by an identity connection which is time delayed. ($\pi[\cdot] = 0.15$)
- *Change transfer function* The transfer function of a randomly chosen unit is set to another transfer function. In the case of the gate transfer function, a new connection to the second input of the unit is made. ($\pi[\cdot] = 0.3$)
- *Change connection* The type of a randomly chosen connection is switched from identity to linear or vice versa. ($\pi[\cdot] = 0.25$)
- *Flip time delay* The time delay flag of a connection is flipped. ($\pi[\cdot] = 0.25$)
- *Flip input* The input flag of a random unit is flipped. (($\pi[\cdot] = 0.15$)
- *Flip output* The output flag of a random unit is flipped. ($\pi[\cdot] = 0.15$)
- *Tidy up* If a random unit is not reachable from the input, or the output is not reachable from that unit, it is removed. ($\pi[\cdot] = 0.5$)

2.3 Fitness function

At every fitness evaluation, a cell structure was used to create a recurrent network with 5 hidden memory cells connected to all inputs and all outputs. (Similar to the LSTM network in figure 2, except for the nature and number of the cells.) To calculate the fitness of the structure, three separate BPTT training runs were performed using different weight initializations. (Since each unit is differentiable, we can apply standard BPTT to learn the parameters of the network.) The negative of the highest error was taken to be the actual fitness value. Weights were initialized between -0.1 and 0.1, and learning rate 0.001 with momentum 0.99 was used. Training went on for 500 epochs.

The training was repeated and a new fitness was assigned for each objective. The objectives were to recognize context-free or context-sensitive languages. In each case, the fitness calculation proceeded *incrementally*: only when structures had been evolved that could learn to recognize short strings of a language, longer strings were tried.

Context-free and Context-sensitive Languages Context-free languages are languages that can be recognized by a non-deterministic push-down automaton. In general, determining whether a string of symbols belongs to a context-free language requires remembering some symbols in the string seen so far, which rules out the use of non-recurrent architectures. In order to evolve memory cells, we chose the context-free language $a^n b^n$, which requires memory of up to n time steps. The task was implemented using networks with 3 input units, one for each symbol (a, b) plus the start symbol S, and three 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 $a^n b^n$ are those in which the number of as and bs is equal, e.g. ST, $SabT$, $SaabbT$, $SaaabbbT$, and so forth. So, for $n = 3$, the set of input and target values would be:

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

This language is very hard for conventional RNNs to learn, and the best reported results generalize only up to $n = 18$ given training sequences ranging from $n = 1 \ldots 10$ [17], while LSTM have been shown to generalize up to over $n = 100$ [2].

Context-*sensitive* languages constitute a more complex class of languages. These are languages that cannot be recognized by deterministic finite-state automata, and are therefore more complex in some respects than regular languages. We chose the language $a^n b^n c^n$ as a benchmark, and in this case 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

The $a^n b^n c^n$ is too hard for regular RNNs but LSTM achieves decent to superb performance on this task. To ensure that the evolved cells were not limited to being able to learn a single language, we used the related but significantly different language $a^n b^{\mathbf{m}} a^n$ as a second benchmark.

3 Results

	Benchmark	
Cell	$a^n b^n, n_t = 1..5$	$a^n b^n, n_t = 1..10$
594	23.2	29.0
6357	36.75	36.2
6379	55.3	40.0
350	30.4	19.0
LSTM	9.6	27.5
	$a^n b^n c^n, n_t = 1..5$	$a^n b^n c^n, n_t = 1..10$
594	7.4	2.0
6357	11.8	16.1
6379	18.1	13.7
350	9.8	0.2
LSTM	8.9	20.0
	$a^n b^m c^n n, n_t = m_t = 1..4$	
LSTM	7.7, 7.7	
594	8.0 $\leq$, 8.0 $\leq$	
6357	8.0 $\leq$, 8.0 $\leq$	
6379	8.0 $\leq$, 8.0 $\leq$	
350	8.0 $\leq$, 8.0 $\leq$	

Fig. 3. Results of four evolved cells on grammar benchmarks compared to LSTM. The table reports the longest strings to which a network constructed out of the indicated cells could generalize after training, averaged over ten runs. n_t and m_t give the ranges of the training sets. Remarkably, some cells learn better with less training data.

A typical evolutionary run required roughly one hour per objective per generation on a 3 Ghz processor. Cell structures capable of learning the desired languages were typically found within 10 generations. An overview of their performance on the selected languages is given in figure 3.

In one configuration, the context-free language $a^n b^n c^n$ was used as one objective and the context-free language $a^n b^n$ as the other. n was increased incrementally as learning capacity increased; when structures had evolved that could learn to recognize string of lengths 1-5, maximum length was increased to 10. During runs with this configuration, the cells shown in figure 4 were evolved.

In a second configuration, evolution started out with a context-free language ($a^n b^n, n \in [1, 5]$) and moved on to a multiobjective setting with one context-free

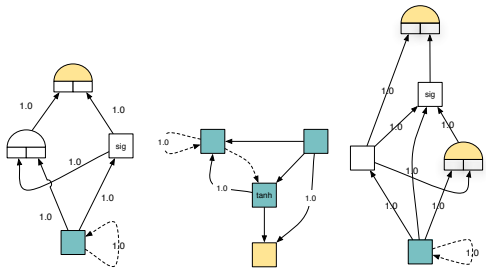

Fig. 4. An evolved cell that can reliably learn the $a^n b^n c^n$ grammar (left), and two others that can learn the $a^n b^n$ grammar (right and middle). Standard RNNs cannot learn these languages. Note the absence of any substantial similarity in their structure.

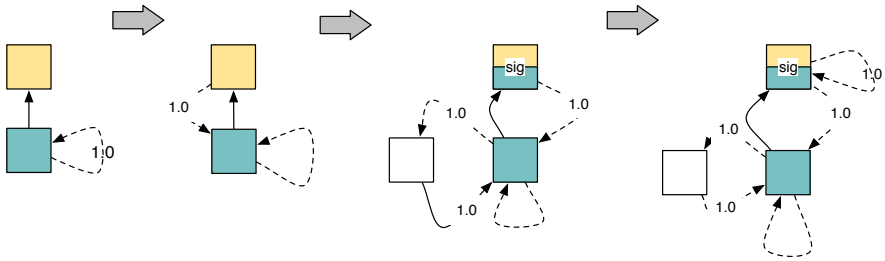

Fig. 5. The evolution of cell 350. Although happening over the course of nine generations, only four mutations were needed in order to evolve a cell capable of the context sensitive language $a^n b^n c^n$ and the context free language $a^n b^m a^n$.

and one context-sensitive language ($a^n b^m c^n$ and $a^t b^t c^t, (m, n) \in [1, 4] \times [1, 4], t \in [1, 5]$). In most runs with this configuration, a cell capable of learning both languages was found.

3.1 Genealogical analysis

Figure 5 depicts the evolution of a cell capable of learning the $a^n b^n c^n$ language. It is interesting to note that the very first step is just a simple recurrent network, which cannot even learn the $a^n b^n$ language to more than a rudimentary level. The second step increased its performance through a recurrent connection from the output back to the input, which lowered the learning error on $a^n b^n$ somewhat. The third stage added a new node, with a time-delay connection in and a linear connection back to the input, essentially creating three types of recurrence to the input node. At this stage, the node was able to learn $a^n b^n$ satisfactorily, but nothing more complex. The final mutation turned the linear connection back from the new unit into a time-delay connection, and added a new recurrent connection on the output. This suddenly enabled several steps of recurrence, which seems to be necessary to handle more complex languages. The final cell can successfully learn both $a^n b^n c^n$ and $a^n b^m a^n$. Note that it differs significantly from the LSTM in that it contains neither gates nor peepholes.

4 Conclusion and discussion

Using an algorithm similar to neural network topology evolution algorithms, we evolved structures for memory cells capable of learning context-sensitive formal languages through gradient descent. The fitness functions were based on the learning capacity of networks of the cells. The end products of evolution were cells that in many ways were comparable to LSTM, the current state-of-the-art in gradient-based sequence learning.

Analysis of the (very diverse) evolved cell structures and their genealogies provided interesting insights into what features contribute to the power of LSTM. The essential ingredients of LSTM's success seem to be (1) linear units with fixed self-connections and (2) nonlinear units (both of which are biologically plausible [11]) while the precise connection structure seems less important. Some of our evolved structures even lack gates (the main explanation for LSTM's advantage over traditional RNNs) yet retain similar learning ability; however, other sequence learning problems might require such gates.

An open question is how big the tradeoff between performance and generality of a specific cell is. Since LSTM is used in a wide range of applications, we believe that evolving general cells is actually quite possible. In order to evaluate the generality of our approach, it is crucial to try our methods on more benchmark problems from other domains, combining unrelated objectives in the same run. These could include learning to predict continuous functions (e.g. superimposed sines), real-world sequence learning problems (e.g. speech processing), and even reinforcement learning problems. It could also mean using non-gradient-based training algorithms, such as evolutionary algorithms, for some objectives.

References

1. K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6:182–197, 2002.
2. F. A. Gers and J. Schmidhuber. LSTM recurrent networks learn simple context free and context sensitive languages. *IEEE Transactions on Neural Networks*, 12:1333–1340, 2001.
3. F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with LSTM. *Neural Computation*, 12:2451–2471, 2000.
4. F. A. Gers and N. Schraudolph. Learning precise timing with LSTM recurrent networks. *Journal of Machine Learning Research*, 3:2002, 2002.
5. F. Gomez and R. Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5:317–342, 1997.
6. F. Gruau. Genetic synthesis of modular neural networks. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 318–325. Morgan Kaufmann, 1993.
7. 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.
8. S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.
9. H. Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4:461–476, 1990.
10. M. Liwicki, A. Graves, H. Bunke, and J. Schmidhuber. A novel approach to on-line handwriting recognition based on bidirectional long short-term memory networks. In *Proc. 9th Int. Conf. on Document Analysis and Recognition*, volume 1, pages 367–371, 2007.
11. R. C. O'Reilly, T. S. Braver, and J. D. Cohen. A biologically based computational model of working memory. In A. Miyake and P. Shah, editors, *Models of Working Memory: Mechanisms of Active Maintenance and Executive Control*. Cambridge University Press, New York, 1998.
12. P. Rodriguez and J. Wiles. Recurrent neural networks can learn to implement symbol-sensitive counting. In *NIPS '97: Proceedings of the 1997 conference on Advances in neural information processing systems 10*, pages 87–93, Cambridge, MA, USA, 1998. MIT Press.
13. J. Schmidhuber. RNN overview, 2004. http://www.idsia.ch/~juergen/rnn.html.
14. K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.
15. P. Werbos. Backpropagation through time: What it does and how to do it. In *Proceedings of the IEEE*, volume 78, pages 1550–1560, 1990.
16. S. Whiteson, M. E. Taylor, and P. Stone. Empirical studies in action selection with reinforcement learning. *Adaptive Behavior*, 15:33–50, 2007.
17. J. Wiles and J. Elman. Learning to count without a counter: A case study of dynamics and activation landscapes in recurrent networks. In *Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society*, pages 482–487, 1995.