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. Long Short-Term Memory (LSTM) is one of the best recent supervised sequence learning methods. Using gradient descent, it trains *memory cells* represented as differentiable computational graph structures. Interestingly, LSTM's cell structure seems somewhat arbitrary. In this paper we optimize its computational structure using 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 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 to either produce the next step in a sequence produced by the system (sequence prediction), or to correctly classify a sequence (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, a *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 attracted some attention because of their simplicity and potential power. Here, the whole sequence is presented to the network, which is then trained by backpropagation through time (BPTT) [16]. However, there are some serious practical limitations to most types of RNNs due to their inability to capture long-term time dependencies. They suffer from the problem of *vanishing gradient* [8], 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 [9]), 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 during

C. Alippi et al. (Eds.): ICANN 2009, Part II, LNCS 5769, pp. 755–764, 2009.

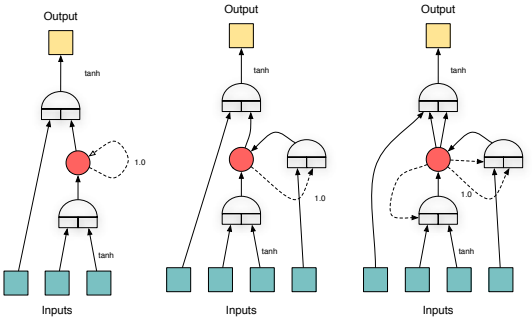

Fig. 1. The incremental development of the LSTM cell. Input units are in teal, output units in yellow. The gate units are shown as a half circle with as the output part and two different squares as inputs. The time delayed connection is dashed, and the red circle is the state unit. The second version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

arbitrarily long time spans. 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 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, 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]. LSTM networks have been shown to outperform other RNNs on numerous time series requiring the use of deep memory [13].

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 [9]. 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 [4]. Finally, *peepholes* were devised, which are direct connections from the state to all gates [5]. 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 [3]. LSTM has recently been shown to perform excellently on many tasks, including speech processing and handwriting recognition (e.g. see [11]).

The incremental design evolution of the LSTM cell outlined above, taken together with its somewhat arbitrary structure, suggests that the development of LSTM could be retraced with artificial evolution, and that LSTM's 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 the capacity to be trained by 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 [10,7]. Apparently, the promise of scalability motivating these approaches has so far not been realized. The latter category, which includes the empirically successful NEAT algorithm [15], instead encodes the structure directly into 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 lacks the recombination operator for simplicity.

Usually, the weights of the neural connections are evolved at the same time as the topology. However, Whiteson [17] 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 our work is to investigate the space of architectural alternatives to LSTM and to 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, consisting of a graph of connected computational units such as the sigmoid, the tanh and the gating unit. For every run, the structure of the cell is replicated a number of times to form a complete recurrent neural network. 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 memory cells 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 directed acyclic graphs ($DAGs$) of differentiable

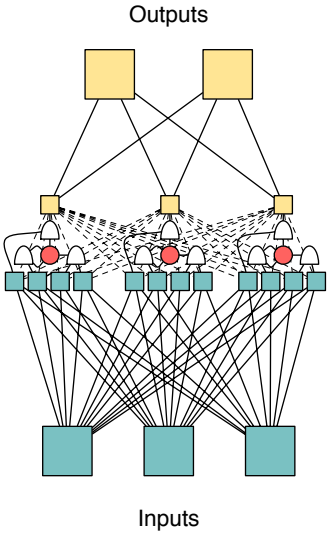

Fig. 2. A network constructed with a hidden layer of three LSTM cells. The recurrent connections from the hidden layer to itself, necessary for the cells to communicate with each other, are shown as dashed.

units, plus time delay connections: time delay connections which may break the DAG property but only propagate activations between time steps.

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 [6]. 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* [2].

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 weight 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 hyperbolic tangent $\tau(x) = tanh(x)$.
- The gating transfer has two inputs x_1 and x_2 and produces $g(x_1, x_2) = \sigma(x_1)x_2$.

The most interesting type of node used in this paper is the *gating* unit that was first introduced in the LSTM cell. Its structure can be thought of as a continuous version of the `if ... then ...` statement, and has two inputs: one condition and one signal. 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 [2]. A population size of 100 was used. For simplicity, no recombination was used; mutation was the only variation operator.

A cell structure is mutated by 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 time delay connection*. Two units are connected by an identity connection which is time delayed. This connection is allowed to break the DAG property. ($\pi[\cdot] = 0.15$)
- *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 mean squared 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 time was set to 2000 epochs.

Formal languages. Determining whether a string of symbols belongs to a particular formal language often 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 $a^n b^n$ [19] (yielding strings ST, $SabT$, $SaabbT$, $SaaabbbT$, etc.) and the context-sensitive language $a^n b^n c^n$ (which yields ST, $SabcT$, $SaabbccT$, $SaaabbbcccT$, etc.), which require memory of up to n and $2n$ time steps, respectively. Symbol strings were presented sequentially to the network, with each symbol's corresponding input unit set to `1`, and the other set to `-1`. At each time step, the network must predict the possible symbols that could come next in a legal string. The $a^n b^n c^n$ is too hard for regular RNNs but LSTM achieves decent to superb performance on this task [3]. 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 an additional objective. See [3] for a more complete explanation.

3 Results

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 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.

	Benchmark	
Cell	$a^n b^n, n_t = 0..5$	$a^n b^n, n_t = 0..10$
`Ana`	8.6	19.5
`Cathy`	0	8.5
`Charlotte`	8.2	19.5
`Mary`	1.675	3.325
LSTM	9.6	27.5
	$a^n b^n c^n, n_t = 0..5$	$a^n b^n c^n, n_t = 0..10$
`Ana`	18.55	47.0
`Cathy`	6.05	7.7
`Charlotte`	18.95	44.9
`Mary`	5.3	1.1
LSTM	15.05	44.85
	$a^n b^m c^n, n_t = m_t = 0..4$	
`Ana`	8.0	
`Cathy`	4.37	
`Charlotte`	8.0	
`Mary`	2.72	
LSTM	8.0	

Fig. 3. Results of four evolved cells, named `Ana`, `Cathy`, `Charlotte` and `Mary`, on grammar benchmarks compared to LSTM. The table reports the biggest parameter to which a network constructed out of the indicated cells could generalize after training, averaged over twenty runs. n_t and m_t give the ranges of the training sets.

3.1 Genealogical Analysis

Figure 5 depicts the evolution of a cell capable of learning the $a^n b^n c^n$ language in about 20% of the training runs. 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 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. 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. On the other hand, the cells `Ana` and `Charlotte`, which outperform `Mary` significantly, feature only a single recurrent internal connection themselves and are mostly constructed out of identity connections and gate units – this makes them similar to LSTMs.

3.2 Validation: Long-Term Dependency T-Maze

In order to validate the cells found, we performed validation tests on the deep memory T-Maze task as described in Bakker's work [1]. The T-Maze task was

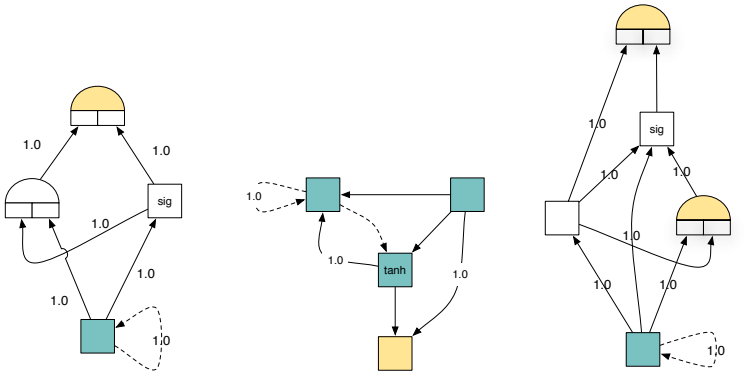

Fig. 4. An evolved cell, named `Charlotte` that can reliably learn the $a^n b^n c^n$ grammar (left), and two others (`Cathy` and `Ana`) that can learn the $a^n b^n$ grammar (right and middle). Standard RNNs cannot learn these languages. Note the absence of any substantial similarity in their structure.

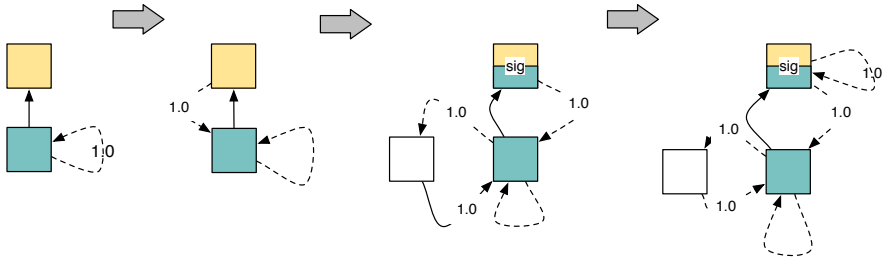

Fig. 5. The evolution of cell `Mary`. Although happening over the course of nine generations, only four mutations were needed in order to evolve a cell which is casually able to learn the underlying structures of the context-sensitive language $a^n b^n c^n$ and the context-free language $a^n b^m a^n$.

Cell	Success ratio	Average reward
`Ana`	0.45	-3.895
LSTM	0.35	-7.545
`Charlotte`	0.25	-8.915
`Cathy`	0.0	-54.485
`Mary`	0.0	-57.74

Fig. 6. The cells tested on the T-Maze task. Each cell was evaluated 20 times.

specifically designed to test a reinforcement learning algorithm's capability to relate events far apart in history. It involves having to remember a single observation at the beginning of the task until the very last time step. Applying the

recurrent policy gradient algorithm [18], a learning rate of 0.01 and a momentum of 0.99 was used in conjunction with a batch size of 100 and a discount factor of 0.99. The corridor length was set to 15.

We found that the cell structure `Ana` outperforms LSTM (see figure 6). Note that this is a reinforcement learning tasks instead of a supervised training task. This is significant, since although we evolved the cell structure to perform well on sequence prediction, it actually peforms well on an unrelated reinforcement learning task. This suggests the evolved cell structures might be quite general and capable of performing substantially different tasks.

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 such cells. The evolved memory cells were in many ways comparable in performance 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) gate units while the precise connection structure seems less important. It is important to note that the cells with gates significantly outperform those without. 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 one single 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. Cells developed using this method could also be incorporated into hybrid algorithms such as *Evolino* [14].

References

1. Bakker, B., Linker, F., Schmidhuber, J.: Reinforcement learning in partially observable mobile robot domains using unsupervised event extraction. In: Proc. IROS 2002, pp. 938–943 (2002)
2. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: Nsga-ii. IEEE Transactions on Evolutionary Computation 6, 182–197 (2002)
3. Gers, F.A., Schmidhuber, J.: LSTM recurrent networks learn simple context free and context sensitive languages. IEEE Transactions on Neural Networks 12, 1333–1340 (2001)

4. Gers, F.A., Schmidhuber, J., Cummins, F.: Learning to forget: Continual prediction with LSTM. Neural Computation 12, 2451–2471 (2000)
5. Gers, F.A., Schraudolph, N.: Learning precise timing with LSTM recurrent networks. Journal of Machine Learning Research 3, 2002 (2002)
6. Gomez, F., Miikkulainen, R.: Incremental evolution of complex general behavior. Adaptive Behavior 5, 317–342 (1997)
7. Gruau, F.: Genetic synthesis of modular neural networks. In: Proceedings of the Fifth International Conference on Genetic Algorithms, pp. 318–325. Morgan Kaufmann, San Francisco (1993)
8. Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J.: Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In: Kremer, S.C., Kolen, J.F. (eds.) A Field Guide to Dynamical Recurrent Neural Networks. IEEE Press, Los Alamitos (2001)
9. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Computation 9(8), 1735–1780 (1997)
10. Kitano, H.: Designing neural networks using genetic algorithms with graph generation system. Complex Systems 4, 461–476 (1990)
11. Liwicki, M., Graves, A., Bunke, H., Schmidhuber, J.: 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, vol. 1, pp. 367–371 (2007)
12. Rodriguez, P., Wiles, J.: Recurrent neural networks can learn to implement symbol-sensitive counting. In: NIPS 1997: Proceedings of the 1997 conference on Advances in neural information processing systems, vol. 10, pp. 87–93. MIT Press, Cambridge (1998)
13. Schmidhuber, J.: RNN overview (2004), `http://www.idsia.ch/~juergen/rnn.html`
14. Schmidhuber, J., Wierstra, D., Gagliolo, M., Gomez, F.: Training recurrent networks by evolino. Neural Computation 19(3), 757–779 (2007)
15. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evolutionary Computation 10(2), 99–127 (2002)
16. Werbos, P.: Backpropagation through time: What it does and how to do it. Proceedings of the IEEE 78, 1550–1560 (1990)
17. Whiteson, S., Taylor, M.E., Stone, P.: Empirical studies in action selection with reinforcement learning. Adaptive Behavior 15, 33–50 (2007)
18. Wierstra, D., Foerster, A., Peters, J., Schmidhuber, J.: Solving deep memory pOMDPs with recurrent policy gradients. In: de Sá, J.M., Alexandre, L.A., Duch, W., Mandic, D.P. (eds.) ICANN 2007. LNCS, vol. 4668, pp. 697–706. Springer, Heidelberg (2007)
19. Wiles, J., Elman, J.: 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, pp. 482–487 (1995)