Sequence Learning Through PIPE and Automatic Task Decomposition

Rafał P. Sałustowicz and Jürgen Schmidhuber
IDSIA
Corso Elvezia 36, 6900 Lugano, Switzerland
e-mail: {rafal, juergen}@idsia.ch, phone: +41-91-9919838

Abstract

Analog gradient-based recurrent neural nets can learn complex prediction tasks. Most, however, tend to fail in case of long minimal time lags between relevant training events. On the other hand, discrete methods such as search in a space of event-memorizing programs are not necessarily affected at all by long time lags: we show that discrete "Probabilistic Incremental Program Evolution" (PIPE) can solve several long time lag tasks that have been successfully solved by only one analog method ("Long Short-Term Memory" — LSTM). In fact, sometimes PIPE even outperforms LSTM. Existing discrete methods, however, cannot easily deal with problems whose solutions exhibit comparatively high algorithmic complexity. We overcome this drawback by introducing *filtering*, a novel, general, data-driven divide-and-conquer technique for automatic task decomposition that is not limited to a particular learning method. We compare PIPE plus filtering to various analog recurrent net methods.

1 INTRODUCTION

How can sequential behavior be learned from training examples? If there are long time lags between relevant events and later error signals, most analog gradient-based recurrent net learning algorithms, such as BPTT or RTRL (see overview by Pearlmutter, 1995) will not work. Their main problem is that error signals "flowing backwards in time" tend to decay exponentially. A recent gradient-based method called *Long Short-Term Memory* (LSTM – Hochreiter and Schmidhuber, 1997) eliminates this problem and can solve complex long time lag tasks involving distributed, high-precision, continuous-valued representations. Even LSTM, however, does not fully eliminate dependence on the time lag size. This motivates our interest in *discrete* search methods that do not care about time lag size at all. Discrete methods are of particular interest where the algorithmic complexity (AC) of a solution is low (i.e., the solution can be implemented by a short program in a given programming language representing initial bias). Discrete methods searching incrementally for better sequence-processing algorithms include Adaptive Levin Search (Schmidhuber et al., 1997), Genetic Programming (Cramer, 1985) with memory cells (e.g., Teller, 1994), and Probabilistic Incremental Program Evolution (PIPE) with memory cells (Sałustowicz and Schmidhuber, 1997).

We benchmark LSTM against PIPE and find that PIPE performs better where there are *"very long"* (as opposed to merely *"long"*) time lags in all training exemplars and there exist solutions with low AC. In case of high AC and not too long time lags, however, LSTM tends to be superior.

Filtering. To overcome AC-related drawbacks of discrete methods we will introduce a technique called *filtering*, a novel, general divide-and-conquer method for automatic task decomposition. Unlike with certain previous approaches, e.g., (Angeline and Pollack, 1992; Koza, 1992; Spector, 1996), the decomposition is not "ad-hoc" but data-dependent. Algorithmically complex tasks are separated into less complex subtasks as follows. The first "expert" is taught to fit as many training data points as possible. After a while the training set is split into learned and yet unlearned data. The next expert then tries to fit just the unlearned data, etc. Once all data points have been fit by various experts, each expert also needs to learn which incoming (test) data to process and which to pass on to the next expert. This possibly complex decision task also is adaptively decomposed into subtasks learned by sequences of "filters", each passing the current data to either its local expert or the next filter of the local expert or the first filter of the next expert. Filtering enables PIPE to achieve excellent generalization performance on complex tasks unsolvable by PIPE itself.

Overview. Section 2 summarizes LSTM. Section 3 summarizes PIPE and mentions different kinds of memory and implementations of multiple outputs for PIPE. Section 4 compares PIPE and LSTM on two long time lag tasks with low AC. Section 5 introduces filtering. Section 6 compares PIPE, LSTM and various other recurrent neural net approaches on a task with comparatively high AC. Section 7 concludes.

2 LSTM

LSTM (Hochreiter and Schmidhuber, 1997) is a recent, analog, gradient-based recurrent neural net approach for supervised learning of sequential processes. Unlike most alternative approaches it can learn from training sequences that do not exhibit any short time lags between relevant events. It does so by enforcing constant error flow through "constant error carrousels"(CEC) within special units, and applying multiplicative gate units that learn to open and close access to the constant error flow. LSTM combines CEC and multiplicative input and output gates to form memory cells that can store information over arbitrary periods of time. See (Hochreiter and Schmidhuber, 1997) for a detailed description of net structure and learning algorithm.

3 PIPE

PIPE (Sałustowicz and Schmidhuber, 1997) is a recent discrete method for automatic program synthesis. PIPE's functional programs consist of instructions from a function set F and a terminal set T. Programs are encoded in n-ary trees that are parsed depth first from left to right, with n being the maximal number of function arguments. PIPE generates programs according to a probability distribution over all possible programs composable from the instruction set ($F \cup T$). The probability distribution is stored in an underlying *probabilistic prototype tree* (PPT). The PPT contains at each node a probability for each instruction from $F \cup T$. Programs are generated by traversing the PPT depth first from left to right starting at the root node. At each node an instruction is picked according to the node's probability distribution. To adapt PPT's probabilities PIPE generates successive populations of programs. It evaluates each program of a population and adapts PPT's probabilities so that the probability of creating the best program of the current population increases (see Sałustowicz and Schmidhuber, 1997).

3.1 MEMORY

Overview. We use two types of memory: recurrent output links, and memorizing cells. We use two kinds of memory access strategies: direct and indexed.

Recurrent Output Links (ROLs). We simply add an instruction "o" to the terminal set. At each time step t, o contains the output of the program at time step $t-1$. For $t=0$, o is set to 0.

Memorizing Cells. There are two kinds of memorizing cells: output cells (OCs) and memory cells (MCs). Their data structures are identical, their applications different. OCs store the output of a program. The return value of the program is then ignored. MCs are only used as internal memory. OCs and MCs can be used simultaneously. There are n_{OC} OCs and n_{MC} MCs, where n_{OC} and n_{MC} are positive integer constants. OCs and MCs can be accessed and modified by programs during runtime. At any given time, OC_j and MC_i denote the current real-valued contents of the j-th and i-th output and memory cells, respectively ($j \in \{0..n_{OC}-1\}$, $i \in \{0..n_{MC}-1\}$). All OC_j and MC_i are initialized with 0. OCs and MCs are accessed by write and read functions that are added to the instruction set (see below).

Direct Memory Access (DMA). With DMA each OC_j and MC_i is associated with a distinct function for setting and reading it. Functions $set\_O_j(arg_1)$ ($set\_M_i(arg_1)$) set the j-th (i-th) output (memory) cell to arg_1 and return arg_1. Terminal instructions get_O_j (get_M_i) return the contents of OC_j (MC_i). The disadvantage of DMA is that the number of instructions in S (the instruction set) grows linearly with the number of output/memory cells. It is applicable only when few memory cells are used.

Indexed Memory Access (IMA). IMA overcomes DMA's problem. With IMA only two functions need to be added to set and read arbitrary many output and memory cells. Function $set\_O(arg_1,\ arg_2)$ ($set\_M(arg_1,\ arg_2)$) sets $OC_{(|round(arg_1)|\ mod\ n_{OC})}$ ($MC_{(|round(arg_1)|\ mod\ n_{MC})}$) $:=$ arg_2 and returns arg_2. Function $get\_O(arg_1)$ ($get\_M(arg_1)$) returns $OC_{(|round(\ arg_1)|\ mod\ n_{OC})}$ ($MC_{(|round(arg_1)|\ mod\ n_{MC})}$) (see, e.g., Teller, 1994).

3.2 MULTIPLE OUTPUTS

Overview. Two methods are used to allow for vector–valued outputs: multiple output cells and multiple programs. Both can be used in combination with each other, memory cells, and/or recurrent output links.

Multiple Output Cells. When output cells (OCs) are used, their contents are treated as the output of a program, while the program's return value is ignored. A program can have multiple output cells ($n_{OC} > 1$) and in this way accommodate for multiple outputs.

Multiple Programs (MPs). Alternatively, if n_O is the number of outputs then a "full" program will consist of n_O independent programs generated according to distinct probabilistic prototype trees. One program is generated for each output and the return value of each program is taken as an output value. In case MPs are used in combination with multiple output cells, MPs merely impose *a priori* structure, while *OC*s contain the output.

4 LOW-COMPLEXITY TASKS

Overview. In this section we compare PIPE and LSTM on two problems involving both long minimal time lags and low algorithmic complexity (AC). So far both the "adding problem" and the "temporal order problem" (Hochreiter and Schmidhuber, 1997) have been solved by only one single analog method (LSTM). The adding experiments show that LSTM's convergence speed depends on time lag size, while PIPE's does not. We will see that sometimes simple ROLs suffice. The temporal order experiments, however, will require memorizing (output) cells.

4.1 ADDING PROBLEM

Task. The task is to identify two relevant, real-valued input components occurring in a long sequence and to output their sum at the end. The task's AC is low because a single combination of only two (although widely separated) past events is necessary for correct prediction (of the sum).

Training and test sequences have random lengths varying from minimal sequence length T to $T + \frac{T}{10}$. Each element of each input sequence is a pair of components. The first component is a real value randomly chosen from the interval $[-1, 1]$; the second is either 1.0, 0.0, or -1.0 for LSTM and 1.0, or 0.0 for PIPE. It is used as a marker: at the end of each sequence, the task is to output the sum of the first components of those pairs that are *marked* by second components equal to 1.0. In a given sequence exactly two pairs are marked as follows: first randomly select and mark one of the first ten pairs (whose first component is then called X_1). Then randomly select and mark one of the first $\frac{T}{2} - 1$ still unmarked pairs (whose first component is then called X_2). The second components of all remaining pairs are zero. Since LSTM needs "trigger inputs" to mark the beginning and end of a sequence, the second components of the first and final pair are set to -1. (In Hochreiter and Schmidhuber's LSTM experiments X_1 is set to zero in the rare case where the *first* pair of the sequence gets marked.) An error signal is generated only at sequence end: the target for LSTM is $0.5 + \frac{X_1+X_2}{4.0}$ (the sum $X_1 + X_2$ scaled to $[0, 1]$) and for PIPE $X_1 + X_2$. A sequence is processed correctly if the absolute error at sequence end is below 0.04.

Table 1: Results for the adding problem. The minimal time lag is $T/2$. "# wrong predictions" is the number of LSTM's incorrectly processed sequences from a test set containing 2560 sequences (error > 0.04). For LSTM the "success after" column gives the number of training sequences required to achieve LSTM's stopping criterion, and all values are means of 10 trials. The "perfect solutions" column reports on how likely PIPE is to find perfectly predicting, algorithmically correct solutions. PIPE's "success after" column gives the number of training sequences required on average (means of 35, 33, 37, and 39 independent runs for T = 50, 100, 500, and 1000 respectively) to achieve a perfect solution (within the PE = 20,000 time limit).

	LSTM		PIPE	
min. lag	# wrong predictions	success after	perfect solutions	success after
25	n.a.	n.a.	70%	786,000
50	1 out of 2560	74,000	66%	832,000
250	0 out of 2560	209,000	74%	689,000
500	1 out of 2560	853,000	78%	832,000

LSTM Setup. Architecture and parameter settings are reported in (Hochreiter and Schmidhuber, 1997).

PIPE Setup. We use PIPE with ROLs. We set $F = \{+, -, *, \%, nop, sin, cos, exp, rlog\}$, where nop is a single argument identity function (all other functions are defined in Sałustowicz and Schmidhuber, 1997), and $T = \{x_0, x_1, o\}$, where x_0, x_1 are input variables and o is the ROL ($o = 0$ at sequence start). Each generation a new training data set is generated. It contains 100 randomly generated sequences. The fitness of a program is the averaged sum of absolute differences between program outputs at sequence end and targets. We use the following parameters: PE = 20,000, P_T=0.9, ε = 0.000001, P_{el}=0.0, PS=10, lr=0.01, P_M=0.1, mr=0.1, T_R=0.3, T_P=0.999999, FIT_s = 0. See (Sałustowicz and Schmidhuber, 1997) for a detailed parameter description.

Results. The minimal time lag between the most recent occurrence of relevant information and the point of prediction varies from 25 to 500 time steps. Table 1 summarizes all results. LSTM results are taken from (Hochreiter and Schmidhuber, 1997). LSTM always finds perfect or almost perfect solutions. With a test set consisting of 2560 randomly chosen sequences, in all 10 independent trials LSTM's average test set error is below 0.01, and there are never more than 3 incorrectly processed sequences. PIPE is able to find perfectly generalizing solutions (0 incorrectly processed sequences) in 66%-78% of all independent runs (50 for each time lag size). With an increasing minimal time lag LSTM needs more and more sequence presentations to solve the task. PIPE, however, always needs roughly the same number of sequence presentations regardless of whether the minimal time lag is 25, 50, 250, or 500 time steps. Although LSTM learns significantly faster than PIPE in case of smaller minimal time lags (50 and 250), PIPE outperforms LSTM in case of very long ones (500).

Discussion. LSTM's time lag dependence stems from error signal interference that increases with sequence length, just like it is harder for a feedforward net to discover 1 relevant input unit among 100 irrelevant ones, than 1 among 10. Since the task's AC remained constant, however, the competing discrete method (PIPE) was not affected by the time lag increase.

4.2 TEMPORAL ORDER PROBLEM

The task has been solved by only one analog method (LSTM). The goal is to classify sequences into four classes depending on the temporal order of two symbols in the sequence. Since there are only four relevant symbol combinations the task has a relatively low AC. It is sufficient, however, to prevent PIPE with ROLs from working efficiently. Memorizing cells are needed.

Task Definition. Inputs and targets of a sequence are represented locally (input vectors with only one non-zero bit). The sequence starts with an E, ends with a B (the "trigger symbol") and otherwise consists of randomly chosen symbols from the set $\{a, b, c, d\}$ except for two elements at positions t_1 and t_2 that are either X or Y. The sequence length is randomly chosen between 100 and 110, t_1 is randomly chosen between 10 and 20, and t_2 is randomly chosen between 50 and 60. There are 4 sequence classes Q, R, S, U which depend on the temporal order of X and Y. The rules are: $X, X \rightarrow Q$; $X, Y \rightarrow R$; $Y, X \rightarrow S$; $Y, Y \rightarrow U$. There are as many outputs as there are classes. Each class is locally represented by a binary target vector with one non-zero component. Error signals occur only at sequence end. A sequence is classified correctly if the final absolute error of all outputs is below 0.3 (only relevant to LSTM as PIPE uses Boolean instructions).

LSTM Setup. Architecture and parameter settings are reported in (Hochreiter and Schmidhuber, 1997).

PIPE setup. We use 4 MPs and 4 Boolean *OC*s with direct memory access. Since *OC*s are used, the MPs merely impose an *a priori* structure on the full program. At the beginning of each sequence all *OC*s are set to false. Boolean values are represented by integers: 1 for true and 0 for false. We set $F = \{if\_set\_O_0\_else, if\_reset\_O_0\_else, if\_set\_O_1\_else, if\_reset\_O_1\_else, if\_set\_O_2\_else, if\_reset\_O_2\_else, if\_set\_O_3\_else, if\_reset\_O_3\_else\}$, where the two argument function $if\_set\_O_i\_else(arg_1, arg_2)$ ($if\_reset\_O_i\_else(arg_1, arg_2)$) $(0 \leq i \leq 3)$ sets the i-th output cell to true (false) and returns true if arg_1 evaluates to true. Otherwise arg_2 is returned. We set $T = \{E, B, a, b, c, d, X, Y\}$, where E, B, a, b, c, d, X, Y are input variables. Each generation a new training data set is generated. It contains 100 randomly generated sequences. The fitness of a program is the number of training sequences the program misclassifies. In case the best program of a generation classifies 100% of the training data correctly, we test its performance on 5000 randomly created test sequences. We stop when a program classifies all training and test sequences correctly or the time constraint of $PE = 500{,}000$ is exceeded. We used the following parameters: P_T=0.8, ε = 0.000001, P_{el}=0.0, PS=10, lr=0.1, P_M=0.2, mr=0.2, T_R=0.3, T_P=0.99, $FIT_s = 0$.

Results. Table 2 summarizes all results. LSTM finds almost perfect or perfect solutions after on average just 31,390 sequence presentations. PIPE is able to solve the problem in 92% of the time, but needs significantly more presentations.

Table 2: Results for the temporal order problem. "# wrong predictions" is the number of sequences incorrectly classified by LSTM (error > 0.3 for at least one output unit) from a test set containing 2560 sequences. For LSTM the "success after" column provides the number of training sequences required to achieve LSTM's stopping criterion. The results are means of 20 trials. PIPE's "solved" column reports how often PIPE was able to find solutions that correctly classify all sequences of the training data set (containing 100 sequences) and the test data set (containing 5000 sequences). PIPE's "success after" column displays how many sequence presentations were necessary on average (means of 46 runs).

LSTM		PIPE	
# wrong predictions	success after	solved	success after
1 out of 2560	31,390	92%	6,048,000

Conclusion. A discrete method (PIPE) can employ memorizing cells to successfully solve a task that so far has been solved only by LSTM. LSTM, however, is faster because the time lags are not extremely long.

5 FILTERING

We have seen how long time lag tasks with *low* AC can be solved by PIPE in conjunction with either ROLs or memorizing cells. Although memorizing cells offer advantages over ROLs (which only work for problems with extremely low AC), discrete methods such as PIPE fail to learn programs that memorize a vast number of independent relevant event combinations within acceptable time. For instance, when we tried PIPE on a high AC task (see Section 6) we obtained only partial solutions. Varying the number of memorizing cells did not help much: the problem is not the memory limitation but PIPE's limited ability to integrate complex information into a single program. To enable discrete methods such as PIPE to deal with high AC tasks we need to split them into subtasks that can be solved independently and then be assembled into an overall solution. This is what filtering does. Filtering learns "experts" and "gates" (filters) to decompose a task. Experts learn target values of data points, while filters learn which data points to assign to which experts.

Filtering facilitates the task of the learning algorithm. Still, the discovery of algorithmic regularities allowing for good *generalization* on test data remains the burden of the learning algorithm. If it does not discover any then filtering will essentially yield a lookup table.

Filtering's basic idea is independent of a particular approach such as PIPE. It can be used in combination with PIPE, genetic programming, neural networks and many other learning algorithms. An important aspect of filtering is that it does *not* merely shift the problem without reducing its complexity: no single component (filter or expert) needs to be particularly powerful or significantly more potent than others.

5.1 FILTERING NON-TEMPORAL DATA

For clarity we will first show how filtering can help in learning static input patterns. It comprises two phases: (1) task decomposition (TD), and (2) task assembly (TA). During TD a task is automatically decomposed into subtasks that are solved independently. During TA partial solutions are assembled into a final one. Either one learning algorithm, or several different ones (hybrid system) can be used to perform TD and TA. We use a single algorithm (PIPE).

5.1.1 Task Decomposition

Given a learning algorithm ALG and a training set SET with n_{SET} data points, we want ALG to output a desired target value for each data point in SET. We treat data points in a discrete way: ALG is said to have learned a data point if the absolute difference between its target value and ALG's output falls below ϵ_d. We train ALG on all data points in SET until either the task is solved (according to ALG's termination criterion), or until ALG has not been able to improve (learn more data points) for some prespecified interval Els_{max}. We insist, however, on ALG learning at least $Ed_{min} \geq 1$ data points. If ALG stops and the task is not finished yet, we (1) save the learned partial solution (the first expert – E_1), (2) split SET into SET_{E_1} containing the data correctly learned by expert E_1 and into SET_{rest}, a set containing remaining data. We then apply ALG to SET_{rest} and repeat the procedure of saving experts and splitting the data set until all data points have been learned. This decomposes the task into subtasks in a way depending only on learning algorithm and data set. Note that E_i learns from a smaller data set than E_j for $i > j$.

Task decomposition by itself, however, is insufficient. After a task has been decomposed, the partial expert solutions need to be assembled in a way that allows for sensibly classifying new, previously unseen test data. The next subsection will address the question: which data points should be assigned to which expert?

5.1.2 Task Assembly

Filters. The task of assigning data to experts may be almost as difficult as the data fitting process itself, and may require similar decomposition. For this purpose we use chains of "filters" (sequentially invoked gates) associated with each expert. Let $F^j_{E_i}$ denote the jth filter of the ith expert. See Figure 1 for an example architecture with experts E_1, E_2, E_3 and corresponding filter chains $F^1_{E_1}$, $F^2_{E_1}$, $F^3_{E_1}$, $F^4_{E_1}$, and $F^1_{E_2}$, $F^2_{E_2}$, $F^3_{E_2}$ respectively. Each incoming data point first moves to

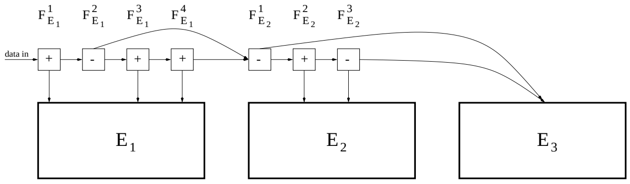

Figure 1: Three experts and their associated filters. Arrows indicate possible data flow.

the first expert's first filter. Filters are either positive or negative: positive filters take a data point and decide whether to pass it on to their expert or not. Negative filters decide whether the data should definitely *not* be passed to their expert. In this case it is passed to the first filter of the next expert or directly to the next expert if it is the final one. Data points that cannot be decided upon are simply passed on to the next filter in the chain.

Filter learning. Filters are learned sequentially in order of expert and filter numbers by dynamically relabelling the data in *SET*. To train $F^i_{E_j}$, all data points of *SET* that have not been learned by any previous filter $F^y_{E_x}$, for all $x < j$ and all $y < i$, if $x = j$, are labeled as belonging to class I if they are in SET_{E_j} and to class II otherwise. If there are more class I than class II data points then we will learn a positive filter, otherwise a negative one. A positive filter $F^i_{E_j}$ will learn to assign class I data points to E_j. A negative filter $F^i_{E_j}$ will learn to pass class II data points to the first filter of the next expert $F^1_{E_{j+1}}$, or to the next expert E_{j+1} in case E_{j+1} is the final one. No positive $F^i_{E_j}$ may pass any class II data points on to E_j and no negative $F^i_{E_j}$ may pass any class I data points on to $F^1_{E_{j+1}}$ or E_{j+1}. If a single filter has separated at least $Fd_{min} \geq 1$ data points, but not all of them, and was not able to improve its performance (by separating additional data points) for some prespecified interval Fls_{max}, we (1) preserve the filter ($F^i_{E_j}$), (2) eliminate the data learned by $F^i_{E_j}$ from class I or II, depending on filter type, and (3) train the next filter $F^{i+1}_{E_j}$ to separate the remaining data points. In this way we incrementally add filters until all class I and class II data points have been correctly classified. We then start learning filters $F^i_{E_{j+1}}$ for the next expert, and repeat the entire procedure until all filters for all experts (except for the final one) have been learned. Note that the number of data points to be separated decreases with each learned filter.

5.2 FILTERING TEMPORAL DATA

Because each training sequence may involve several intermediate target signals (e.g., each time step may require a new prediction), sets of training *sequences* are split and grouped into learned/unlearned and class I/II sets in a slightly different way. Since temporal dependencies can occur among unlearned and already learned points, we cannot simply exclude the learned points from the training data set of an expert/filter: all experts and filters need to see all inputs of the *entire* data set *SET*. Data set splits during task decomposition and assembly are achieved by measuring an experts'/filter's performance only on data points that have not been already learned by a previous expert/filter.

Also during later processing of (previously unseen) test data, each data point is given to each filter and expert. Filter outputs are then processed sequentially (starting with $F^1_{E_1}$) to determine which expert's output is valid.

Filtering facilitates the decomposition of temporal tasks with many independent relevant event combinations. Detecting the relevant dependencies within a single event combination, however, remains the duty of the learning algorithm.

We use PIPE with memory cells for training both experts and filters. In Section 6 we will see that PIPE plus filtering can extract the algorithmic regularities necessary for achieving perfect generalization.

6 A HIGH-COMPLEXITY TASK

The task is to learn the "embedded Reber grammar", e.g. Smith and Zipser (1989), Cleeremans et al. (1989), and Fahlman (1991). It allows for training sequences with very short time lags and can therefore be learned by many recurrent net algorithms. Its AC is rather high, though, since predictions are required at each time step, and numerous input combinations need to be learned. PIPE without filtering completely failed to solve this task. During its best runs PIPE was merely able to predict roughly 60% of all data points of a sequence correctly. Filtering, however, did enable PIPE to solve this popular recurrent net benchmark.

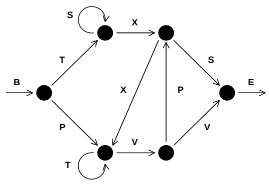

Figure 2: Transition diagram for the Reber grammar.

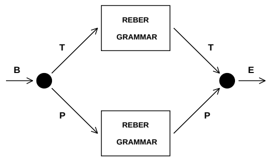

Figure 3: Transition diagram for the embedded Reber grammar. Each box represents a copy of the Reber grammar (see Figure 2).

Task Definition. Starting at the leftmost node of the directed graph in Figure 3, symbol strings are generated sequentially (beginning with the empty string) by following edges — and appending the associated symbols to the current string — until the rightmost node is reached. Edges are chosen randomly if there is a choice (probability: 0.5). The task is to read strings, one symbol at a time, and to permanently predict the next symbol (error signals occur at every time step). To correctly predict the symbol before last, the second symbol has to be remembered.

Comparison. We compare PIPE with *MC*s and filtering to LSTM (results taken from (Hochreiter and Schmidhuber, 1997)), "Elman nets trained by

Elman's training procedure" (ELM) (results taken from Cleeremans et al. 1989), Fahlman's "Recurrent Cascade-Correlation" (RCC) (results taken from Fahlman 1991), and RTRL (results taken from Smith and Zipser (1989), where only the few successful trials are listed). It should be mentioned that Smith and Zipser actually make the task easier by increasing the probability of short time lag exemplars.

Training/Testing. We use local input/output representation (7 inputs, 7 outputs). Following Fahlman, we use 256 training strings and 256 separate test strings. The training set is generated randomly. Test sequences are generated randomly, too, but sequences already used in the training set are not used for testing. For PIPE we use three pairs of training and test sets. The first two pairs (1,2) have training sets that contain on average shorter sequences than their corresponding test sets. For the third pair (3) the opposite is true. A trial is considered successful if all symbols of all sequences in both test set and training set are predicted correctly — that is, if the output value(s) corresponding to the possible next symbol(s) is(are) always the largest ones. We measure PIPE's test performance on all three test sets.

Neural Network Setups. Architectures and parameter settings for LSTM, RTRL, ELM, RCC are reported in the references listed above.

PIPE Setup. We use PIPE with *MC*s, MPs, and filtering. Each expert consists of 7 programs (one for each output) that share 10 *MC*s. Each filter consist of one program with 10 *MC*s. *MC*s are initialized to 0 before each sequence presentation. We set $F = \{+, -, *, \%, set\_M, get\_M, sin, cos, exp, rlog\}$ (see Sałustowicz and Schmidhuber, 1997 for function definitions), and $T = \{B, T, S, X, E, P, V, R\}$, where B, T, S, X, E, P, V are input variables and R is the "generic random constant" (see Sałustowicz and Schmidhuber, 1997). For filters, program outputs are mapped to class I if > 0 and to class II otherwise. Expert fitness is the number of wrong predictions (note: smaller fitness is better!). Filter fitness for a positive (negative) filter is the number of incorrectly classified class I (II) points, if all class II (I) points have been classified correctly, and infinite otherwise. We set $PE = 5,000,000$ (time constraint), $FIT_s = 0$, Els_{max}=1,000 program evaluations, Ed_{min}=1, Fls_{max}=10,000 program evaluations, Fd_{min}=1, P_T=0.9, $\varepsilon = 0.000001$, P_{el}=0.1, PS=10, lr=0.01, P_M=0.4, mr=0.4, T_R=0.3.

Results. Table 3 shows all results for the analog methods. LSTM is the only one that always learns to solve the task. RTRL and RCC perform better than ELM, but worse than LSTM. Results for PIPE with filtering are shown in Table 4. Filtering enabled PIPE to always *learn* the task. Due to short time lags, however, LSTM learned significantly faster. With the first two training sets (containing short sequences) PIPE was always able to find perfectly generalizing solutions. Training set 3 it *learned* in roughly 2/3 of the time needed to learn training sets 1 and 2. When trained on longer sequences (training set 3), however, it rarely achieved perfect generalization. The imperfect solutions performed close to optimal (1–8 wrong predictions out of 4630–4705) on longer test sequences, but worse (9–21 wrong predictions out of 3994) on shorter ones (from test set 3).

Table 3: Results of several analog approaches for the embedded Reber grammar: percentage of successful trials and number of sequence presentations until success for RTRL (results taken from Smith and Zipser 1989), "Elman net trained by Elman's procedure" (results taken from Cleeremans et al. 1989), "Recurrent Cascade-Correlation" (results taken from Fahlman 1991) and LSTM (results taken from Hochreiter and Schmidhuber 1997). Only LSTM always learned to solve the task. It also needed least sequence presentations on average (mean of 30 trials).

Analog Approaches			
method	hidden units	% of success	success after
RTRL	12	"a fraction"	25,000
ELM	15	0	>200,000
RCC	7-9	50	182,000
LSTM	3 blocks, size 2	100	8,440

Table 4: PIPE's results for the embedded Reber grammar: The "tr. set" column shows which training set is used. The "av. tr. err. / max. err." column reports PIPE's average training error (fitness) and the maximal error (worst possible fitness) on the training set. The "success after" column reports on how many sequence presentations (averaged over 20 runs) are necessary to achieve perfect performance on the training set. The rightmost two columns report PIPE's average (vs. worst possible) test set performance on all three test sets and on how often PIPE discovered perfectly generalizing solutions.

PIPE				
tr. set	av. tr. err. / max. err.	success after	av. test err. / max. err.	perf. sol.
1	0 / 4018	24,880,896	0 / 13329	100%
2	0 / 3909	28,062,976	0 / 13329	100%
3	0 / 4717	16,006,400	4.7 / 13329	10%

Discussion. Filtering enabled a discrete method (PIPE) to reliably (always) learn a task with comparatively high AC that PIPE by itself could not learn, and that has been reliably (always) solved by only one analog method (LSTM). PIPE's programs generalized extremely well, except for those learned from long sequences: one of the many non-minimal algorithmic representations of long sequences may be learned quickly but does not necessarily embody a small finite state automaton capable of generating both the long sequences and certain shorter ones outside the training set.

7 CONCLUSION

The traditional static approach to pattern processing typically involves statistic estimators and regressors of which feedforward neural nets are particular instances. Despite the sometimes obvious advantages of sequential pattern recognition, however, conventional statistics has not contributed too much insight into learning sequential analysis of temporally extended patterns. In fact, conceptionally novel approaches such as gradient-based learning algorithms for recurrent neural nets with time-varying inputs and methods for searching algorithm space are more closely related to each other than to more conventional static approaches. Each offers certain advantages and disadvantages. Among those identified in this paper are the following: discrete incremental program search (as embodied by PIPE) can outperform even the current state-of-the-art analog, gradient-based method (LSTM) in case of problems with solutions of low algorithmic complexity (relative to the algorithm-learning device) and *very long* (as opposed to merely *long*) time lags between relevant training events. LSTM, however, seems better suited to tasks that involve both long time lags and algorithmically complex solutions. On the other hand, discrete methods like PIPE can be augmented by a novel data and task decomposition technique called filtering, which not only splits complex tasks into several subtasks solvable by comparatively simple algorithms (experts) but also decomposes into managable subtasks the problem of finding an appropriate expert for given data. Filtering can enable discrete methods such as PIPE to reliably solve tasks with comparatively high algorithmic complexity.

Acknowledgments

Some of filtering's central ideas came up during discussions with Marco Wiering. This work was supported by SNF grant 2100-49'144.96 "Long Short-Term Memory".

References

Angeline, P. J. and Pollack, J. B. (1992). The evolutionary induction of subroutines. In *Proceedings of the 14th Annual Conference of the Cognitive Science Society*, pages 236–241, Hillsdale, NJ. Lawrence Erlbaum Associates.

Cleeremans, A., Servan-Schreiber, D., and McClelland, J. L. (1989). Finite-state automata and simple recurrent networks. *Neural Computation*, 1:372–381.

Cramer, N. L. (1985). A representation for the adaptive generation of simple sequential programs. In Grefenstette, J., editor, *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, pages 183–187, Hillsdale, NJ. Lawrence Erlbaum Associates.

Fahlman, S. E. (1991). The recurrent cascade-correlation learning algorithm. In Lippmann, R. P., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 3*, pages 190–196. San Mateo, CA: Morgan Kaufmann.

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

Koza, J. R. (1992). *Genetic Programming – On the Programming of Computers by Means of Natural Selection*. MIT Press.

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

Sałustowicz, R. P. and Schmidhuber, J. (1997). Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2):123–141.

Schmidhuber, J., Zhao, J., and Wiering, M. (1997). Shifting inductive bias with success-story algorithm, adaptive Levin search, and incremental self-improvement. *Machine Learning*, 28:105–130.

Smith, A. W. and Zipser, D. (1989). Learning sequential structures with the real-time recurrent learning algorithm. *International Journal of Neural Systems*, 1(2):125–131.

Spector, L. (1996). Simultaneous evolution of programs and their control structures. In Angeline, P. and K. E. Kinnear, J., editors, *Advances in Genetic Programming 2*, chapter 7. MIT Press, Cambridge, MA, USA.

Teller, A. (1994). The evolution of mental models. In Kinnear, Jr., K. E., editor, *Advances in Genetic Programming*, pages 199–219. MIT Press.