# ADAPTIVE HISTORY COMPRESSION FOR LEARNING TO DIVIDE AND CONQUER

In Proc. International Joint Conference on Neural Networks, Singapore, volume 2, pages 1130-1135. IEEE, 1991.

Jürgen Schmidhuber*
Department of Computer Science
University of Colorado
Campus Box 430
Boulder, CO 80309, USA

**Abstract**

Previous approaches to on-line supervised sequence learning (back-prop through time (BPTT), e.g. [7], the IID- or RTRL-algorithm [1][8], its more efficient relative [5], and others) do not try to selectively focus on *relevant* inputs, they waste efficiency and resources by focussing on *every* input. With many applications, a second drawback of these methods is the following: The longer the time lag between an event and the occurrence of a corresponding error the less information is carried by the corresponding back-propagated error signals. This paper asks: How can a system learn to reduce the descriptions of event sequences *without loosing information*? It is shown that the learning system ought to concentrate on *unexpected* inputs and ignore expected ones. This insight leads to the construction of neural systems which learn to 'divide and conquer' by recursively composing sequences. The first system (section 2) creates a self-organizing multi-level hierarchy of recurrent predictors. The second sytem (section 3) involves only two recurrent networks: It tries to collapse a multi-level predictor hierarchy into a single recurrent net. Experiments show that the system can require less computation per time step *and* much less training sequences than the conventional training algorithms for recurrent nets.

## 1 HISTORY COMPRESSION

Consider a discrete time predictor whose state at time $t$ is described by an environmental input vector $i(t)$, an internal state vector $h(t)$, and an output vector $o(t)$. At time 0, the predictor starts with $i(0)$ and an internal start state $h(0)$. At time $t \geq 0$, the predictor computes

$$o(t) = f(i(t), h(t)).$$

At time $t > 0$, the predictor furthermore computes

$$h(t) = g(i(t-1), h(t-1)).$$

If $o(t) = i(t+1)$ at a given time $t$, then the predictor was able to predict the input $i(t+1)$ from the previous inputs. The new input was *derivable* by means of $f$ and $g$. All information about the input at a given time $t_x$ can be reconstructed from the knowledge about

$$t_x, f, g, i(0), h(0), \textit{ and the pairs } (t_s, i(t_s)): \ 0 < t_s \leq t_x, \ o(t_s - 1) \neq i(t_s).$$

*This work has been done at Technische Universität München, Germany

The information about the observed input sequence can be even more compressed: There is no need to store all the $i(t_k), k = 1, \ldots, l$; it suffices to store only those components of the $i(t_k)$ that were not correctly predicted.

The observation above implies that we can differentiate various input sequences by knowing only the unpredicted inputs and the corresponding time steps: No discriminating information will be lost if we ignore the expected inputs. We call this the principle of history compression.

# 2 A MULTI-LEVEL PREDICTOR HIERARCHY

With the help of the principle of history compression we can build a hierarchical chunking system. Each $i$th-level predictor $P_i$ (a conventional dynamic recurrent neural network, e.g. [1][8][7][5]) is trained to predict its own next input (plus potentially available external target vectors) from the previous ones. At each time step the input of the lowest-level recurrent predictor $P_0$ is the current external input. We create a new higher-level adaptive predictor $P_{s+1}$ whenever a lower-level adaptive predictor $P_s$ stops to continue improving its predictions. Only if at a given time step $P_s$ ($s \geq 0$) fails to predict its next input (or target) then $P_{s+1}$ will receive as input the concatenation of this next input of $P_s$ *plus a unique representation of the corresponding time step*; the activations of $P_{s+1}$'s hidden and output units will be updated. Otherwise $P_{s+1}$ will not perform an activation update. This procedure ensures that $P_{s+1}$ is fed with a reduced description of the input sequence observed by $P_s$. In general, $P_{s+1}$ will receive fewer inputs over time than $P_s$. With the known learning algorithms, the higher-level predictor will have less difficulties in learning to predict the critical inputs than the lower-level predictor. This method [4] will lead to a hierarchy of predictors and is related to the recent chunking-method described in [2].

Here it should be mentioned that with many practical tasks there is no need for unique representations of time steps [4].

Often a multi-level predictor hierarchy will be the fastest and safest way of learning to deal with sequences with multi-level temporal structure (e.g speech). *Experiments have shown that multi-level systems as above can learn tasks which are practically not learnable by conventional recurrent networks.* One disadvantage of a predictor hierarchy, however, is that it is not known in advance how many levels there will be needed. Another disadvantage is that levels are explicitly separated from each other. It is possible, however, to collapse the hierarchy into a single network as described next.

# 3 COLLAPSING THE HIERARCHY INTO A SINGLE RECURRENT NET

## 3.1 OUTLINE

The 2-net system described below consists of two conventional recurrent networks: The 'automatizer' and the 'chunker'. At each time step the automatizer receives the current external input. Its error function is threefold: One term forces it to emit certain desired target outputs at certain times. The second term forces it at every time step to predict its own next input. The third (crucial) term will be explained below.

Only if the automatizer makes an error concerning the first and the second term of its error function, the unpredicted input (including a potentially available teaching vector) *plus a unique representation of the current time step* will become the new input of the chunker. Before this new input can be processed, the chunker (whose last input may have occurred way back) is trained to predict this higher-level input from its current internal state and its last input (a conventional recurrent net algorithm is employed). After this the chunker performs an activation update which contributes to a higher level internal representation of the input history. Note that according to the principle of history compression the chunker is fed with a *reduced description of the input history.* The information deducable by means of the predictions of the automatizer can be considered as *redundant.* (The beginning of an episode usually is not predictable, therefore it is fed to the chunking level, too.)

The chunker currently might be able to quickly learn to predict its next input although the automatizer currently is not. This is because the 'credit assignment paths' of the chunker often will be short compared to those of the automatizer (this will usually happen if the incoming inputs carry global temporal structure which has not yet been discovered by the automatizer). Then the chunker will develop useful internal representations of previous unexpected input events. Due to the final term of its error function, the automatizer will be forced to reproduce these internal representations, *by predicting the state of the chunker.* Therefore the automatizer will be able to create useful internal representations by itself in an *early* stage of input processing. These internal representations must carry the relevant information for enabling the automatizer to improve its predictions. Therefore the chunker will receive less and less inputs, since more and more inputs become predictable by the automatizer. This is the *collapsing operation.* Ideally, the chunker will become obsolete after some time.

## 3.2 DETAILS OF THE 2-NET CHUNKING ARCHITECTURE

The system described below is the *on-line* version of a representative of a number of variations of the basic principle described in 3.1. It must be mentioned that unlike with the *off-line* creation of a multi-level predictor hierarchy as in section 2 there is no formal proof that the *on-line* version described below will never be subject to instabilities. See [4] for a 2-net *off-line* version and various modifications.

The automatizer has $n_I + n_D$ input units, $n_{H_A}$ hidden units, and $n_{O_A}$ output units. The chunker has $n_{H_C}$ hidden units, and $n_{O_C}$ output units. All input units and all hidden units of the automatizer have directed forward connections to all non-input units of the automatizer. All input units of the automatizer have directed forward connections to all non-input units of the chunker. This is because the input units of the automatizer serve as input units for the chunker at certain time steps. There are additional $n_S$ input units for the chunker for providing unique representations of time steps. These additional input units also have directed forward connections to all non-input units of the chunker. All hidden units of the chunker have directed forward connections to all non-input units of the chunker.

At time $t$ the environment provides a $n_I + n_D$-dimensional real input vector $d(t) \circ x(t)$ to the system. (Here '$\circ$' is the concatenation operator). $d(t)$ is a $n_D$-dimensional teacher-defined target vector. With pure prediction tasks $n_D = 0$. For convenience we define $\delta_d(t) = 1$ if at time $t$ the teacher provides such a target $d(t)$ and $\delta_d(t) = 1$ otherwise. If $\delta_d(t) = 0$ then $d(t)$ takes on some default value, e.g. the zero vector. The $n_I + n_D$-dimensional real input vector of the automatizer at time $t$ is $i_A(t)$. The $n_{H_A}$-dimensional real activation vector of the hidden units of the automatizer at time $t$ is $h_A(t)$. The real $n_{O_A}$-dimensional output vector of the automatizer at time $t$ is $o_A(t)$. $h_A(t)$ and $o_A(t)$ are based on previous inputs and are computed without knowledge about $d(t)$ and $x(t)$. $o_A(t)$ is the concatenation $d_A(t) \circ p_A(t) \circ q_A(t)$ of the $n_D$-dimensional vector $d_A(t)$, the $n_I$-dimensional vector $p_A(t)$ and the $n_{H_C} + n_{O_C}$-dimensional vector $q_A(t)$. Therefore, $n_{O_A} = n_D + n_I + n_{H_C} + n_{O_C}$. The automatizer will try to make $d_A(t)$ equal to $d(t)$ if $\delta_d(t) = 1$, and it will try to make $p_A(t)$ equal to $x(t)$ (thus trying to predict $x(t)$). Here we define the target prediction problem as a special case of an input prediction problem. Finally, and most importantly, the automatizer will try to make $q_A(t)$ equal to $h_C(t) \circ o_C(t)$, *thus trying to predict the internal state of the chunker.*

The real $n_{H_C}$-dimensional activation vector of the hidden units of the chunker at time $t$ is $h_C(t)$. The real $n_{O_C} = n_D + n_I + n_S$-dimensional output vector of the chunker at time $t$ is $o_C(t)$. $o_C(t)$ is the concatenation $d_C(t) \circ p_C(t) \circ s_C(t)$ of the $n_D$-dimensional vector $d_C(t)$, the $n_I$-dimensional vector $p_C(t)$, and the $n_S$-dimensional vector $s_C(t)$. The chunker will try to make $d_C(t)$ equal to the externally provided teaching vector $d(t)$ if $\delta_d(t) = 1$ and if the automatizer failed to emit $d(t)$. Furthermore, it will always try to make $p_C(t) \circ s_C(t)$ equal to the next non-teaching input to be processed by the chunker. This input may be many time steps ahead.

Both chunker and automatizer simultaneously are trained by a conventional algorithm for recurrent networks. Both the IID-Algorithm and BPTT are appropriate. In particular, a computationally inexpensive variant of BPTT is interesting: There are tasks with hierarchical temporal structure where only a few iterations of 'back-propagation back into time' per time step are sufficient to bridge arbitrary time lags (see section 4).

The algorithm described below refers to the procedure of 'updating a network $N$'. Such an update is based on an activation spreading phase which can look as follows:

*Repeat for a constant number of iterations (typically one or two):*
*1. For each non-input unit $j$ of $N$ compute $\hat{a}_j = f_j(\sum_i a_i w_{ij})$, where $a_j$ is the current activation of unit $j$, $f_j$ is a semilinear differentiable function and $w_{ij}$ is the weight on the directed connection from unit $i$ to unit $j$.*
*2. For all non-input units $j$: Set $a_j$ equal to $\hat{a}_j$.*

Now it suffices to specify the input-output behavior of the chunker and the automatizer as well as the details of error injection:

*INITIALIZATION: All weights are initialized randomly. In the beginning, at time step 0, make $h_C(0)$ and $h_A(0)$ equal to zero, and make $i_A(0)$ equal $d(0) \circ x(0)$. Represent time step 0 in $s(0)$. Update the chunker to obtain $h_C(1)$ and $o_C(1)$.*

*FOR ALL TIMES $t > 0$ UNTIL INTERRUPTION DO:*
*1. Update the automatizer to obtain $h_A(t)$ and $o_A(t)$. The automatizer's error $e_A(t)$ is defined as*

$$2e_A(t) = (p_A(t) \circ q_A(t) - x(t) \circ h_C(t) \circ o_C(t))^T (p_A(t) \circ q_A(t) - x(t) \circ h_C(t) \circ o_C(t)) + \delta_d(t)(d_A(t) - d(t))^T (d_A(t) - d(t)).$$

*Use a gradient descent algorithm for dynamic recurrent nets to change each weight $w_{ij}$ of the automatizer in proportion to (the approximation of) $-\frac{\partial e_A(t)}{\partial w_{ij}}$. Make $i_A(t)$ equal to $d(t) \circ x(t)$. Uniquely represent $t$ in $s(t)$.*
*2. If the low-level error of the automatizer*

$$2e_P(t) = (p_A(t) - x(t))^T (p_A(t) - x(t)) + \delta_d(t)(d_A(t) - d(t))^T (d_A(t) - d(t))$$

*is less or equal to a small constant $\beta \geq 0$, then set $h_C(t+1) = h_C(t)$, $o_C(t+1) = o_C(t)$. Else define the chunker's prediction error $e_C(t)$ as*

$$2e_C(t) = (p_C(t) - x(t))^T (p_C(t) - x(t)) + \delta_d(t)(d_C(t) - d(t))^T (d_C(t) - d(t)) + (s_C(t) - s(t))^T (s_C(t) - s(t)),$$

*use a gradient descent algorithm for dynamic recurrent nets to change each weight $w_{ij}$ of the chunker in proportion to (the approximation of) $-\frac{\partial e_C(t)}{\partial w_{ij}}$, and update the chunker to obtain $h_C(t+1)$ and $o_C(t+1)$.*

# 4 EXPERIMENTS

Josef Hochreiter (a student at TUM) implemented variants of the chunking algorithm and tested them on a prediction task involving comparatively long time lags. He compared the results to the results obtained with the conventional learning algorithm for recurrent nets. It turned out that chunking systems can be superior to the conventional algorithm in two respects: They may require less computation per time step, and in addition they may require fewer training sequences.

A prediction task with a 20-step time lag was constructed. There were 22 possible input symbols $a, x, b_1, b_2, \ldots, b_{20}$. The learning systems observed one input symbol at a time. There were only two possible input sequences: $ab_1 \ldots b_{20}$ and $xb_1 \ldots b_{20}$. These were presented to the learning systems in random order. At a given time step, one goal was to predict the next input (note that in general it was not possible to predict the first symbol of each sequence due to the random occurrence of $x$ and $a$). The second (and more difficult) goal was to make the activation of a particular output unit (the 'target unit')

equal to 1 whenever the last 21 processed input symbols were $a, b_1, \ldots, b_{20}$ and to make this activation 0 whenever the last 21 processed input symbols were $x, b_1, \ldots, b_{20}$. No episode boundaries were used: Input sequences were fed to the learning systems without providing information about their beginnings and their ends. Therefore there was a continuous stream of input events: *On-line* versions of the methods had to be used. The task was considered to be solved if the local errors of all output units (including the target unit) were always below 0.3 (with the exception of the errors caused by the occurrences of $a$ and $x$ which were unpredictable)

With both the conventional and the novel approach, all non-input units employed the logistic activation function $f(x) = \frac{1}{1+e^{-x}}$. Weights were initialized between -0.2 and 0.2. Local input representations of 22 possible input symbols $a, x, b_1, \ldots, b_{20}$ were employed: Each symbol was represented by a bit-vector with only one non-zero component.

The conventional recurrent net had one hidden unit, one input unit for each of the input symbols $a, x, b_1, \ldots, b_{20}$, one input unit for the last target, and one input unit whose activation was always 1 for providing a modifiable bias for the non-input units. In addition, it had 23 output units for predicting the next input plus the target (if there was any) (the bias unit was not predicted by the system). 21 iterations of error propagation 'back into the past' were performed at each time step. This is the minimal number required for 20-step time lags (in [7] this method is referred to as 'truncated back-propagation through time'). Note that more iterations 'back into the past' would just cause additional confusion instead of being beneficial. Therefore one may say that external knowledge about the nature of the task was given to the system.

With various learning rates the result was: *Apparently it is not possible for the conventional algorithm to solve the task in reasonable time.* Of course, the network quickly learned to predict the occurrences of the symbols $b_1, \ldots, b_{20}$, but the 20-step time lags seemed to pose insurmountable problems (the test runs were interrupted after 1.000.000 training sequences). (It should be mentioned, however, that limited computer time did not allow a systematic test of all possible parameters.) Note that in the context of speech processing 20 time frames are not at all a long time.

To find out about the limits of the conventional algorithm (and to test whether something was wrong with the implementation of the conventional algorithm) the prediction problem was simplified such that an analoguous 5-step time lag problem was obtained. With this simplified task, in addition to $a$ and $x$ there were only 4 (instead of 20) more input symbols $b_1, \ldots, b_4$. With 4 test runs and a learning rate of 1.0, the following numbers of training sequences were necessary to obtain satisfactory solutions: 1.900.000, 900.000, 3.500.000, 250.000.

How did the chunking system perform on the 20-step task? Like the conventional network, the automatizer had one hidden unit, one input unit for each of the input symbols $a, x, b_1, \ldots, b_{20}$, one input unit for the last target, and one input unit which was always 1 for providing a modifiable bias for the non-input units. The error criterion was the following: The chunker was updated whenever the maximal error observed at one of the automatizer's low-level output units exceeded 0.2. The chunker had 1 hidden unit and 23 output units for predicting its next input plus the target (if there was any). With this experiment, the chunker did *not* need unique time step representations $s(t)$. The automatizer had 23 output units for predicting the next environmental input plus the target (if there was any), and 23 output units for predicting the non-input units of the chunker. One iteration per network update was performed. The 'unfolding in time' method (e.g. [7]) was applied to both the chunker and the automatizer. *Only 3 iterations of error propagation 'back into the past' were performed at each time step.* Both learning rates were equal to 1.0.

The chunking system was able to solve the task. 17 test runs were conducted. With 13 test runs the system needed less than 5000 training sequences to make the error of the automatizer's target unit always smaller than 0.12. With the remaining 4 test runs the following numbers represent upper bounds for the number of training sequences required to make the error of the automatizer's target unit smaller than 0.06: 30.000, 35.000, 25.000, 15.000.

The final weight matrix of the automatizer often looked like the one one would expect: Typically the hidden unit turned on whenever the terminal $a$ occurred. A strong recurrent connection from that hidden unit to itself kept it alive for the following 21 time steps, then it became inhibited if an $x$ occurred (symmetrical solutions were observed, too). A major result is that this structure evolved although only

3 iterations of error propagation 'back into the past' were performed at each time step! The particular chunking system needed *less computation per time step than the conventional method.* It was *local in both space and time. Still it required less training sequences* (due to limited computer time the experiments did not tell how many training sequences the conventional algorithm needs).

It is intended to apply both multi-level chunking systems and 2-net chunking systems to real world tasks. For instance, with speech processing tasks there seems to be an abundancy of multi-level temporal structure. Therefore chunking systems seem to be interesting candidates for learning to process and predict speech.

# 5 CONCLUDING REMARKS

It seems that humans tend to memorize and focus on non-typical and unexpected events and that they tend to try to explain new unexpected events by previous unexpected events. In the light of the principle of history compression this makes a lot of sense.

Once events become expected, they tend to become sub-conscious. There is an obvious analogy to the chunking algorithm: The chunker's attention is removed from events that become expected; they become 'sub-conscious' and give rise to even higher-level 'abstractions' of the chunker's 'consciousness'.

Aspects of hierarchical *reinforcement learning* can be found in [3] and [4] (the latter mentions an extension of the non-compositional approach described in [6]).

# 6 ACKNOWLEDGEMENTS

I wish to thank Josef Hochreiter for conducting the experiments.

# References

[1] 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.

[2] J. Schmidhuber. Adaptive decomposition of time. In T. Kohonen, K. Mäkisara, O. Simula, and J. Kangas, editors, *Artificial Neural Networks*, pages 909–914. Elsevier Science Publishers B.V., North-Holland, 1991.

[3] J. Schmidhuber. Learning to generate sub-goals for action sequences. In T. Kohonen, K. Mäkisara, O. Simula, and J. Kangas, editors, *Artificial Neural Networks*, pages 967–972. Elsevier Science Publishers B.V., North-Holland, 1991.

[4] J. Schmidhuber. Neural sequence chunkers. Technical Report FKI-148-91, Institut für Informatik, Technische Universität München, April 1991.

[5] J. Schmidhuber. An $O(n^3)$ learning algorithm for fully recurrent networks. Technical Report FKI-151-91, Institut für Informatik, Technische Universität München, May, 6 1991.

[6] J. Schmidhuber. Reinforcement learning in Markovian and non-Markovian environments. In D. S. Lippman, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 500–506. San Mateo, CA: Morgan Kaufmann, 1991.

[7] R. J. Williams and J. Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation*, 4:491–501, 1990.

[8] R. J. Williams and D. Zipser. Experimental analysis of the real-time recurrent learning algorithm. *Connection Science*, 1(1):87–111, 1989.