NEURAL SEQUENCE CHUNKERS

Jürgen Schmidhuber
Institut für Informatik
Technische Universität München
Arcistr. 21, 8000 München 2, Germany
schmidhu@informatik.tu-muenchen.de

Technical Report FKI-148-91, April 30, 1991

Abstract

This paper addresses the problem of learning to 'divide and conquer' by meaningful hierarchical adaptive decomposition of temporal sequences. This problem is relevant for time-series analysis as well as for goal-directed learning, particularily if event sequences tend to have hierarchical temporal structure. The first neural systems for recursively chunking sequences are described. These systems are based on a principle called the 'principle of history compression'. This principle essentially says: As long as a predictor is able to predict future environmental inputs from previous ones, no additional knowledge can be obtained by observing these inputs in reality. Only unexpected inputs deserve attention. A focus is on a class of 2-network systems which try to collapse a self-organizing (possibly multi-level) hierarchy of temporal predictors into a single recurrent network. Only those input events that were not expected by the first recurrent net are transferred to the second recurrent net. Therefore the second net receives a reduced discription of the input history. It tries to develop internal representations for 'higher-level' temporal structure. These internal representations in turn serve to create additional training signals for the first net, thus helping the first net to create longer and longer 'chunks' for the second net. Experiments show that chunking systems can be superior to the conventional training algorithms for recurrent nets.

1 OUTLINE

Section 2 motivates the search for sequence-composing systems by describing major drawbacks of 'conventional' learning algorithms for recurrent networks with time-varying inputs and outputs. Section 3 describes a simple observation which is essential for the rest of this paper: It describes the 'principle of history compression'. This principle essentially says: As long as a predictor is able to predict future environmental inputs from previous ones, no additional knowledge can be obtained by observing these inputs in reality. Only unexpected inputs deserve attention. This principle is of particular interest if typical event sequences have hierarchical temporal structure. Basic schemes for constructing sequence chunking systems based on the principle of history compression are described. Section 4 then describes on-line and off-line versions of a particular 2-network chunking system which tries to collapse a self-organizing (possibly multi-level) predictor hierarchy into a single recurrent network (the automatizer). The idea is to feed everything that is unexpected into a 'higher-level' recurrent net (the chunker). Since the expected things can be derived from the unexpected things by the automatizer, the chunker is fed with a reduced description of the input history. The chunker has a comparatively easy job in finding possibilities for additional reductions, since it typically works on a slower time scale and receives less inputs than the automatizer. *Useful internal representations of the chunker in turn are taught to the automatizer.* This leads to even more reduced input descriptions for the chunker, and so on. Section 5 describes modifications and extensions of the particular system described in section 4. Section 6 experimentally demonstrates that chunking systems can be superior to conventional recurrent net algorithms in two respects:

Chunking systems may require less computation per time step, and in addition they may require fewer training sequences. Section 7 briefly mentions a possible extension for reinforcement learning and adaptive control. Section 8 draws an analogy between the behavior of the chunking systems and the apparent behavior of humans.

2 DRAWBACKS OF PREVIOUS METHODS

In what follows the ith component of a vector $v(t)$ will be called $v_i(t)$. $dim(v)$ denotes the dimension of the vector v. A discrete time training sequence (called an episode) consists of n ordered pairs $((x(t), d(t)) \in R^l x R^m$, $0 < t \leq n$, each $x(t)$ being called an input event. At time t of an episode a learning system receives $x(t)$ as an input and produces output $y(t) \in R^m$. The goal of the learning system is to minimize

$$E = \frac{1}{2} \sum_t \sum_i (d_i(t) - y_i(t))^2.$$

In general the learner experiences many different episodes during training. Since the gradient of the error sum over all episodes is equal to the sum of the corresponding gradients, for convenience we renounce on indices for different episodes.

In general the task above requires to memorize past events. Previous approaches to solving this problem employed either (approximations of) gradient descent in recurrent nets [5] [17] [33] [14] [34] [12] [36], 'adaptive critic'-like methods [21] [23], or (more recently) adaptive 'fast weights' [28].

Definition [20]: A learning algorithm for dynamic neural networks is *local in time* if for given network sizes (measured in number of connections) during on-line learning the peak computation complexity at every time step is $O(1)$, for *arbitrary* durations of sequences to be learned. A learning algorithm for dynamic neural networks is *local in space* if during on-line learning for limited durations of sequences to be learned and for *arbitrary* network sizes (measured in number of connections) and for *arbitrary* network topologies the peak computation complexity per connection at every time step is $O(1)$.

In what follows we will focus on the gradient-based methods for supervised learning. These can be classified into two major categories: The 'forward-backward' methods [17] [33] [12] are based on the 'unfolding in time' principle which requires to store all time-varying activations of all units at all times of the activation spreading phase of each episode. In general, this requires an unknown amount of storage. At the end of each episode the desired error derivatives can be computed by 'back-propagation through time' (which is of the same computational complexity as the forward pass). The 'forward-backward' methods are not local in time.

The 'forward-only' methods [14] [36] [28] use a fixed number of special variables for computing derivatives of time-varying activations in an *on-line* fashion. No 'back-propagation through time' is necessary. But, the 'forward-only' methods are not local in space.

Neither the 'forward-backward' methods nor the 'forward-only' methods are local. Both are too expensive for large scale applications.

With many applications, a second drawback of both methods is the following: The longer the time lag between an event and the occurrence of a corresponding error signal the less the corresponding back-propagated error signals are significant: Long time lags tend to lead to uniform (and therefore uninformed) distributions of error signals[1].

[1] It should be mentioned, however, that Hochreiter recently found a method for carefully designing recurrent connections such that the error distribution problem becomes less severe [4]. Furthermore it should be mentioned that recently Rohwer's (non-compositional) moving target algorithm [15] has been successfully tested on a simple task with long time lags [16]. Mozer's recent approach for discovering global temporal structure [9] should be mentioned, too. Mozer uses a predefined *non-adaptive* distribution of decay rates for the hidden units of his recurrent net. With his approach, units with high decay rates tend to become sensitive for 'more local' structure while units with small decay rates tend to become sensitive for 'more global' structure. With the approaches described in sections 4 and 5 there is nothing like a pre-wired decay rate. The systems incrementally learn to define appropriate slower and slower time scales on their own.

In section 6 (experiments) a system will be tested which is local in both space and time while still being able to bridge arbitrary time lags and to generate weight matrices like the ones one would expect from the conventional methods. This system can work if there are 'local' temporal regularities in the input stream. It tries to detect these regularities and use them as building-blocks for 'higher-level' regularities. The next section describes the basic principles.

3 THE PRINCIPLE OF HISTORY COMPRESSION

Consider a discrete time predictor (not necessarily a neural network) 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 a particular n_t-step input sequence observed by the predictor is contained in the $l+5$-tuple

$$(n_t, f, g, i(0), h(0), (t_1, i(t_1)), (t_2, i(t_2)), \ldots, (t_l, i(t_l))),$$

where the $t_l > 0$ are those time steps where $o(t_l - 1) \neq i(t_l)$. The input at a given time t_x can be reconstructed from the knowledge about t_x, f, g, $i(0)$, $h(0)$, and the pairs $(t_s, i(t_s)), 0 < t_s \leq t_x, o(t_s - 1) \neq i(t_s)$.

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 above principle will be referred to as *'the principle of history compression'*. In an informal manner, it has been formulated in [25], [24], [19], and [27].

Now consider a second 'higher-level' discrete time predictor whose state at time $t_k, k \in \{l, \ldots, l\}$ is described by an environmental input vector $i_C(t_k) = t_k \circ i(t_k)$, an internal state vector $h_C(t_k)$, and an output vector $o_C(t_k)$. (Here '$\circ$' is the concatenation operator.) At time $t_0 = 0$, the higher-level predictor starts with $t_0 \circ i(t_0)$ and an internal start state $h_C(0)$. At time $t_k, k = 0, \ldots, l$, the higher-level predictor computes

$$o_C(t_k) = f_C(i_C(t_k), h_C(t_k)).$$

At time $t_k, k = 1, \ldots, l$, the higher-level predictor furthermore computes

$$h_C(t) = g_C(i_C(t_{k-1}), h_C(t_{k-1})).$$

If $o_C(t_k) = t_{k+1} \circ i(t_{k+1})$ at a given time t_k, then the higher-level predictor was able to predict $i_C(t_{k+1})$ from its previous inputs. The new higher-level input was *derivable* by means of f_C and g_C.

Now all information about a particular input sequence with n_t time steps can be derived from n_t, f, g, f_C, g_C, $i(0)$, $h(0)$, $h_C(0)$ and the unpredicted higher-level inputs together with their corresponding time steps. In an analoguous manner, we can recursively introduce new prediction levels.

3.1 History Compression and Temporal Invariances

Consider a trivial predictor which always emits $o(t) = i(t)$ at a given time t. With such a predictor, all information about a particular input sequence is contained in the length of the sequence, the

initial input, and those pairs $(t, i(t))$ of time steps and inputs where $i(t+1) \neq i(t)$. This may be one reason for the fact that many neurons in biological systems respond only to *changes* in input activity. The next subsection considers less trivial *adaptive* predictors and adopts a more general view on the advantages of history compression.

3.2 History Compression and Chunking

An implication of the principle of history compression is that in a certain sense only unexpected inputs deserve attention. The principle of history compression is central for the current paper, which demonstrates that it can be efficient to focus on unexpected inputs and ignore expected ones.

In the context of learning, $f, g, f_C, g_C, \ldots$ will be parameterized, and the parameters will be adaptive. Previous algorithms for sequence prediction show drawbacks if they receive too many inputs at too many different time steps. Therefore we are looking for a possibility to reduce the number of inputs and time steps. This can be possible if the temporal structure of the incoming input sequences tends to be organized hierarchically.

3.2.1 A Multi-Level Predictor Hierarchy

A straight-forward solution is to create a new higher-level adaptive predictor P_{s+1} as described above whenever a lower-level adaptive predictor P_s stops to continue improving. Each predictor is trained to predict its own next input. P_{s+1} receives as inputs concatenations of unexpected inputs of P_s *plus unique representations of the corresponding time steps.* Therefore 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 will lead to a hierarchy of predictors and is related to the method described in [27]. There are at least two important differences between the approach described in [27] and the approach described herein. One difference is the criterion for creating a new level in the hierarchy: With [27] this criterion is based on measuring the *reliability* of the predictor's predictions. The other difference is that the method described in [27] does not care for unique representations of 'critical' time steps.

With supervised learning tasks one wants the system to emit certain pre-defined target vectors at certain times. This can be achieved by simply treating each target vector as part of the input to be predicted. If one is not interested in having a 'single network representation' of past events then the 'multi-level method' can be the method of choice.

3.2.2 Collapsing a Predictor Hierarchy into a Single Net

One disadvantage of a predictor hierarchy as described in the last subsection 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. But, in theory it is possible to collapse the hierarchy into a single network. The basic idea behind the chunking systems described in the next sections can be expressed in the following loop:

1. Train a lower-level adaptive predictor to adjust the parameters of f and g to reduce the number of incorrect predictions. This can be possible if if the environmental inputs at least sometimes obey some *local* temporal laws. 2. Train a higher-level adaptive predictor to adjust the parameters of f_C and g_C such that the number of incorrect predictions of higher-level inputs is reduced. 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. 3. *Train the lower-level predictor to reproduce the meaningful internal states of the higher-level predictor.* The ability to reproduce these internal states will help the lower-level predictor to improve f and g in step 1. Go to 1.

Most of the remainder of this paper (sections 4 and 5) is dedicated to the description of various implementations of this basic idea.

4 A 2-NET CHUNKING SYSTEM

We use the principle of history compression to build a particular recursive neural sequence chunking system. A main advantage of this system is that in certain (quite realistic) environments it can incrementally learn to focus on (or 'shift attention to') those points in time which are relevant for solving certain tasks, even if these points are separated by long time lags. This property can make the system superior to previous learning algorithms for dynamic recurrent nets which treat every time step equally.

The particular system described below consists of two recurrent dynamic networks. It tries to collapse a self-organizing (possibly multi-level) predictor hierarchy into a single recurrent network. The second net, called the *chunker*, tries to develop internal representations for 'higher-level' temporal structure which currently is not predictable by the first network (the *automatizer*). The higher-level internal representations of the chunker in turn serve to generate additional training signals for the automatizer, thus teaching the automatizer to create internal representations that carry the same relevant information as the internal representations of the chunker. This, in turn, makes the higher-level temporal structure discovered by the chunker predictable by the automatizer. Therefore the automatizer recursively can learn to bridge longer and longer time spans, thus creating longer and longer 'chunks' for the chunker.

Subsection 4.1 describes the novel architecture and on-line and off-line variants of the algorithm. Subsection 4.2 gives comments on the behavior of the system. To get a feeling of what the algorithm does, the reader might wish to read subsection 4.2 before reading 4.1. Section 5 describes a number of useful modifications of the method.

4.1 The Chunking Architecture and the Algorithm

Again it must be mentioned that the algorithms described below are only representatives of a number of variations of the same basic principle. Section 5 lists a few of the possible modifications. The notation below is not intended to be consistent with section 3.

With the versions described below, 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. (With this version recurrence obviously is limited to the hidden units. This is not a necessary condition. In fact, the algorithm below is fairly independent of the network topology. We need some sort of internal feedback, however.)

With the variant described below, at time t the environment provides a $n_I + n_D$-dimensional real input vector $d(t) \circ x(t)$ to the system. (Again, '$\circ$' is the concatenation operator). $d(t)$ is a n_D-dimensional teacher-defined target vector. With pure prediction tasks there never is a target vector ($n_D = 0$). 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, $n_{O_A} = n_D + n_I + n_{H_C} + n_{O_C}$. As we will see, 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. (Since the desired output becomes part of the next input, this method provides an alternative to the teacher forcing method described in [36].) 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 $s_C(t)$. As we will see, 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 are trained by a conventional algorithm for recurrent networks. Both the IID-Algorithm [14][36] and back-propagation ([32][7][11][17]) in networks 'unfolded in time' [8] are appropriate. In particular, a computationally inexpensive variant of the 'unfolding in time' method can be very interesting: With many tasks, only a few iterations of 'back-propagation through time' per time step are sufficient to bridge arbitrary time lags (see the experiments in section 6).

We describe two versions of the algorithm: A safer off-line version and an esthetically more pleasing on-line version. With the off-line version, the chunker and the automatizer are trained in alternating phases. A phase ends if it seems that further training will not significantly improve performance in the near future. Weights are changed at the end of 'training episodes' (examples of sequences of inputs and desired outputs). With the on-line version, the chunker and the automatizer are trained simultaneously, and weights are changed immediately at each time step (the time steps may occur on two different time scales, however). Here the assumption is that the learning rates are small enough to avoid instabilities [36].

The algorithms described below refer to the procedure of 'updating a network N'. Such an update is based on an activation spreading phase which (in the simplest case) 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 $\hat{a}_j$.

By performing more than one iteration of activation spreading at each time tick, one can adjust the algorithm to environments that change in a manner which is not predictable by semilinear operations (theoretically three additional iterations are sufficient for any environment [6]). With different network architectures, the appropriate update procedures have to be employed (e.g. [5][3]).

For the specification of our method it suffices to specify the input-output behavior of the chunker and the automatizer as well as the details of error injection. First we will write down an off-line variant, then an on-line variant.

4.1.1 An Off-Line Version

After random weight initialization the off-line version described herein alternates between PASS 1 and PASS 2 (to be described below) until being externally interrupted.

PASS 1:

REPEAT UNTIL the automatizer has not improved significantly for some time:
1. Select a training episode $((x(0),d(0))((x(1),d(1)),\ldots,((x(k),d(k))$. *Make* $i_A(0)$ *equal to* $d(0) \circ x(0)$. *Represent time step 0 in* $s(0)$. *Initialize the activations of all non-input units with zero. Update the chunker to obtain* $h_C(1)$ *and* $o_C(1)$.

2. FOR ALL TIMES $0 < t < k$ *DO:*
2.1. 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 compute (or approximate) $-\frac{\partial e_A(t)}{\partial w_{ij}}$ *for each weight* w_{ij} *of the automatizer. Make* $i_A(t)$ *equal to* $d(t) \circ x(t)$. *Uniquely represent* t *in* $s(t)$.
2.2. 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 update the chunker to obtain $h_C(t+1)$ *and* $o_C(t+1)$.
3. Change each weight w_{ij} *of the automatizer in proportion to (the approximation of)* $\sum_{0<t\leq k} -\frac{\partial e_A(t)}{\partial w_{ij}}$.

PASS 2:

REPEAT UNTIL the chunker has not improved significantly for some time:
1. Select a training episode $((x(0),d(0))((x(1),d(1)),\ldots,((x(k),d(k))$. *Make* $i_A(0)$ *equal to* $d(0) \circ x(0)$. *Represent time step 0 in* $s(0)$. *Initialize the activations of all non-input units with zero. Update the chunker to obtain* $h_C(1)$ *and* $o_C(1)$.

2. FOR ALL TIMES $0 < t \leq k$ *DO:*
2.1. Update the automatizer to obtain $h_A(t)$ *and* $o_A(t)$. *Make* $i_A(t)$ *equal to* $d(t) \circ x(t)$. *Uniquely represent* t *in* $s(t)$.
2.2. If the low-level error of the automatizer (see PASS 1) $e_P(t) \leq \beta = const. \geq 0$, *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 compute (or approximate) $-\frac{\partial e_C(t)}{\partial w_{ij}}$ *for each weight* w_{ij} *of the chunker, and update the chunker to obtain* $h_C(t+1)$ *and* $o_C(t+1)$.
3. Change each weight w_{ij} *of the chunker in proportion to (the approximation of)* $\sum_{t,e_P(t)>\beta} -\frac{\partial e_C(t)}{\partial w_{ij}}$.

4.1.2 An On-Line Version

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* $o_C(t+1)$.

4.2 The Behavior of the 2-Net Chunking System

Let us have a look at how the chunking systems described in section 4.1 works. The gradient descent algorithm to be applied to the automatizer is a conventional one based on the assumption that the chunker's state does not depend on the automatizer's weights. One term of the automatizer's error function (the one defined by $d(t)$ and $d_A(t)$) forces it to behave like a conventional supervised learning dynamic recurrent network. (Since the desired output becomes part of the next input, this method provides an alternative to the teacher forcing method described in [36]. The last paragraph of section 5.3 describes a trivial modification where the desired output is not part of the next input.) The corresponding errors will be called errors of the first kind. Another term of the error function of the automatizer (the one defined by $x(t)$ and $p_A(t)$) forces it to predict its next non-teaching input. The corresponding errors will be called errors of the second kind. Both errors of the second kind and errors of the first kind are called low-level errors. The better the automatizer, the fewer low-level errors it will make. Only if it makes a low-level error, the unpredicted input (including the potentially available teaching vector) plus a unique representation of the current time step are transferred to the chunking level, where they contribute to a higher level internal representation of the input history. Therefore it is possible that the chunker is updated at a low rate compared to the automatizer: The chunker may work on a much slower time scale. The beginning of an episode usually is not predictable, therefore it is fed to the chunking level, too.

Whenever the automatizer makes a low-level error, the unpredicted input (including the potentially available teaching vector) is used to train the chunker (whose last input may have occurred way back in time). Again, the gradient descent algorithm to be applied to the chunker is a con-

ventional one which is based on the assumption that the chunker's inputs do not depend on the chunker's weights.

Let us now assume that the chunker in certain situations learns to predict its next environmental input. The chunker currently might be able to learn this rather quickly although the automatizer currently is not. This is because the 'credit assignment paths' of the chunker often will be short in comparison to those of the automatizer. Such situations may occur if the incoming inputs obey some global temporal structure which has not yet been discovered by the automatizer. Due to the principle of history compression the chunker receives a reduced description of the past input history (modulo loss of information caused by a non-zero error threshold β): The information deducable by means of the predictions of the automatizer can be considered as *redundant*.

Given the assumption above, the chunker will develop internal representations of previous unexpected input events that allow to make good predictions. Due to the final term of the automatizer's error function, the automatizer will be trained to reproduce these internal representations, *by predicting the internal state of the chunker*. These additional training signals can be available long before the automatizer receives error signals from its own low-level learning process. Therefore the automatizer might be able to create useful internal representations by itself in an early stage of input processing. After some time, the automatizer will be able to use these internal representations for improving on its own prediction tasks, simply because they must carry the relevant information. Therefore the chunker will receive less and less inputs, since more and more inputs become predictable by the automatizer. This is the collapsing operation. In a way, the chunker builds bridges through time for the automatizer. It tries to detect additional redundancy in those inputs that are currently not considered to be redundant by the automatizer. Ideally, in deterministic environments the chunker will become obsolete after some time.

Noise will be treated as an unexpected input. It does not pose a fundamental problem. Noise may reduce learning speed, however, since in a 'regular' environment with hierarchical temporal structure noise tends to shorten the time-bridges established by the chunker. See section 5.4 for an important comment on this issue.

Of course, there is no need for strict temporal hierarchies to make the system advantageous. For instance, heterarchies also can provide possibilities for greatly reducing the description of the input history. This is the point: *Every* temporal environmental regularity which allows reduced descriptions of the external dynamics might be helpful for shortening 'credit assignment paths' for improving predictions and for making goal-directed learning easier.

The chunking system aims at learning 'abstractions' of event sequences, without sacrificing the advantages of so-called 'sub-symbolic' neural information processing in favor of so-called 'symbolic' processing. The chunker does more than just putting beginnings and endings of sub-programs together (like the sub-goal generating systems described in [25], [19], and [29]). All levels of the abstraction hierarchy are represented in the same network. Potentially, each level is directly accessible from each other level. This makes this 'neural' solution attractive.

With a chunking system based on on-line weight modification we currently cannot prove that it will always work as desired. This is due to the potential for instabilities introduced by on-line weight changing. Theoretically, an input which has been predictable by the automatizer can become unpredictable again because of on-line interactions between chunker and automatizer. However, in the simulation described in section 6 some satisfactory results were obtained with an on-line version.

5 MODIFICATIONS OF THE CHUNKING SYSTEM

At the beginning of section 4.1 it was mentioned that the algorithms described in 4.1 are only representatives of a number of variations on the same basic principle. In what follows, a few of the possible modifications are listed.

5.1 Various Representations of Time Steps and Critical Moments

The consequent transformation of the principle of history compression requires to feed unique representations of 'critical' time steps to the chunker, and let it predict a unique representation of the next 'critical' time step. With training sequences of indefinite length, how can we represent time steps uniquely? One possibility for doing this is to make $s(t)$ one-dimensional and set $s(t) = \frac{1}{t}$. In the context of neural nets, a problem with this unique time representation is that it probably will not work very well if there are long time lags, because very different time steps will have very similar representations.

With many problems precise knowledge about critical time steps is not at all neccessary to profit from the properties of the chunking system (see the experiments described in section 6). Instead, it may be beneficial to have an explicit representation of the particular situation which occurred at the time of a particular prediction error. In many cases the situation in which the error occured will carry more easily accessible information than a simple-minded representation of the corresponding time step.

Thus, in addition to $s(t)$ (or even instead of $s(t)$) we can feed the whole current state of the automatizer to a chunker in case of low-level errors. Then we have to train the chunker not only to predict the next unpredicted input of the automatizer but also the next critical automatizer state (of course, the chunker then will need more input and output units). Here one assumption is that the state of the automatizer contains all relevant information. But, this assumption need not always be true: The current time step need not be uniquely defined by the state of the automatizer.

A similar alternative would be to use the *last* correctly predicted input as an additional input for the chunker, as proposed by Josef Hochreiter (personal communication).

5.2 Alternative Error Criteria

Instead of using the global low-level error $e_P(t)$ as a criterion for deciding about the chunker input at time t, chunker updates can be triggered by other criteria. For instance, the chunker can be updated whenever the *maximal* error observed at one of the automatizer's low-level output units exceeds a certain threshold. (This method has been applied in the experiments described below.) Section 5.3 describes a more selective strategy which is based on local errors instead of global ones.

In practical applications we often do not want to insist on *perfect* automatizer predictions. For instance, with the variant described in 4.1 we would like to set $\beta > 0$. If there are non-zero error thresholds then a simple practical modification concerning the error propagation strategy is the following: We can introduce a second error threshold π for defining acceptable errors. At a given time the automatizer receives error signals only from those units whose local error currently is $> \pi$ (π should be smaller than the error threshold which determines whether the chunker is updated or not). Why can this be useful? In typical applications the automatizer soon will be updated more often than the chunker. If we changed the automatizer's weights at each time step then low-level erros could start to dominate the whole error function of the automatizer: The simple modification described above can be useful for preventing the automatizer from always receiving error signals from all of its output units.

Of course, other error functions than mean-squared error (e.g. entropy measures etc.) are possible.

5.3 Strategies that are Selective in Space

This important modification is based on the comment in section 3: It suffices to memorize only those *components* of the input vectors that were not correctly predicted.

Define $dp_A(t) = d_A(t) \circ p_A(t)$, $dp_C(t) = d_C(t) \circ p_C(t)$. Instead of using the global low-level error $e_P(t)$ as a criterion for deciding about the chunker input at time t, we can employ a more selective strategy: If the automatizer's local prediction errors exceed a certain low threshold only for the k components $dp_{A_{i_1}}(t), \ldots, dp_{A_{i_k}}(t)$ of $dp_A(t)$ then the chunker's input consists of a representation of the current time step and a vector whose components are all set to zero except

for those corresponding to the local mismatches. The training signal for the chunker has to be modified correspondingly: The output activation $dp_{C_{l_j}}(t)$ is trained to be equal to $dp_{A_{l_j}}(t)$ for all $l_j \in \{l_1, \ldots, l_k\}$. The remaining $dim(dp_C(t)) - k$ output units need not be trained. This selective strategy still preserves the unpredicted (noteworthy) information. It is assumed that this method will speed up learning because it focusses not only on relevant points in time but also on relevant points in space.

If the chunker has many more output units than hidden units then it makes sense to let the automatizer predict only those output units of the chunker that recently have received an error signal. The weight changing algorithm has to be adapted correspondingly.

A related modification is the following: At time t we need not feed the target vector to be predicted back to the next input $i_A(t)$. Instead, we can set $i_A(t) = x(t)$. Suppose that the automatizer has made an error of the first kind but not of the second kind: When updating the chunker, we need not transfer the correctly predicted 'next input' of the automatizer, we just have to transfer the representation of the current time step.

5.4 Variants of Predicting Chunker States

1. Instead of letting the automatizer predict only the hidden and output units of the chunker, it sometimes can speed up learning to train it to emit the last environmental chunker input, too. In noisy environments, this might be a less clever idea: If the chunker learns to recognize noise as what it is, it will build internal representations that ignore noise. If the automatizer is not forced to emit the last input of the chunker, then it will not have to corrupt its weights in favour of unpredictable things.

2. Instead of letting the automatizer predict all the output units of the chunker, it sometimes can speed up learning to let it predict only the hidden units. This makes sense if the chunker has no chance to transport input information directly to its output units. This can be achieved by constructing a 'hidden units bottleneck' by removing all direct connections from input units to output units.

5.5 Using a Confidence Network

Instead of providing the chunker with a new input whenever the automatizer makes a prediction error, we can introduce a 'confidence network' as in [27] and [26] for modelling the reliability of the automatizer's predictions. A confidence network is a network whose input $i(t)$ is equal to the input of a predictor, and whose output $c(t)$ is interpreted as a measure of the system's *confidence in the predictor's predictions*. One implementation of this principle would be to let $dim(c(t)) = 1$, and to let the confidence network's error be

$$E_C = \frac{1}{2} \sum_t (m(t) - c(t))^2,$$

where $m(t)$ is 1 if the predictors output matches the desired output (within a certain tolerance), and 0 otherwise. C's one-dimensional output is trained to be high (interpreted as high confidence) for situations where the predictor usually works and to be low (interpreted as low confidence) for situations where the predictor usually fails. There are many variations of this principle. For instance, a confidence network can be trained to emit the probability that the predictor does not fail [26].

With this approach we want to provide the chunker with a new input whenever the output of the confidence net is clearly below 1 (this means low confidence in the predictions of the automatizer). An extension of this approach would be to use 'variation 2' described in section 3.1 of [27] and to employ the selective strategy of section 5.3.

5.6 Training the Chunker without the Automatizer

With the off-line version one can gain efficiency each time PASS 1 has been finished: For each training episode we need to store only those pairs $(x(t), d(t))$ that are not predictable by the automatizer. The ordered sequences of these pairs can be immediately used for training the chunker. This is more efficient than always determining critical points in time with the help of the automatizer.

5.7 Weight Elimination and Incremental Growing of the Automatizer

Experimentally it was found that chunking systems sometimes need comparatively few training sequences for learning to emit the correct output through the output units of the chunker (comparable results were obtained with the related multi-level chunking system described in [27]). Sometimes many more training sequences are needed for 'teaching the knowledge of the chunker to the automatizer'. In other words, sometimes most of the training time is needed for making the automatizer solve the same task as the two-level system. This indicates that if one is not interested in having a collapsed representation of different levels in the prediction hierarchy, then one might save time by working with a simple multi-network hierarchy as described in [27] and in section 3.2.1.

One reason for the comparatively slow collapsing operation is that in the beginning of the training phase the automatizer tends to activate its hidden units for sub-tasks that do not necessarily require hidden units. For instance, unnecessary hidden units sometimes are used much like the bias unit. In later stages of training, when the automatizer is forced to use its hidden units for reproducing chunker states, it can take a long time to retrain the weights to the hidden units. This is because the derivatives of the activation functions of the hidden units tend to be small.

There are some domain dependent ad-hoc methods for improving performance with respect to this problem (e.g. deviating from gradient descent by adding small numbers to derivatives of activation functions before multiplying with error signals).

An interesting alternative is to apply a 'weight elimination procedure' to the automatizer. This can be done by introducing an additional term for its error function which penalizes heavy weights (e.g. [1]). Then one would expect to obtain some hidden units with near-zero weights in the early stages of learning. These hidden units may become useful later when the goal is to reproduce meaningful chunker states.

An alternative is to start with an automatizer with only a few hidden units and to add more and more hidden units whenever the automatizer stops to continue improving. There are at least two variants of this procedure: 1. Never stop training all weights of the automatizer. 2. After having added a new hidden unit to the automatizer, train only the weights from and to the new unit and freeze the other weights.

5.8 Using Other Learning Algorithms for Automatizer and Chunker

Instead of implementing the chunker and the automatizer as supervised learning recurrent nets we can implement them as networks with adaptive delays or as fast-weight systems as described in [28]. But even reinforcement learning algorithms for recurrent nets [21][34] might be applied. In general the activation spreading phase described in 4.1 has to be replaced by the method designed for the particular architecture.

5.9 Other Modifications

1. Instead of applying PASS 1 for the first time, we can start the off-line version by applying gradient descent to the automatizer without ever providing teaching signals for changing $q_A(t)$ at any time t.

2. Other variants of the scheme would slightly affect the order of the computations performed by the algorithm.

3. Further variants can be generated by combining the methods described in the previous subsections. For instance, it is expected that a combination of the modifications described in 5.2, 5.6, and 5.7 will turn out to be the method of choice.

6 EXPERIMENTS

Note: This section describes preliminary results and is likely to grow in a future extended version of this paper.

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. (Here both the 'unfolding in time' method [17][33] and the 'Infinite Input Duration' method [14][36] will be referred to as 'the conventional algorithm'; their off-line versions compute the same gradient.) 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 [35] 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.

In the light of these experiments one might ask: If there are tasks with only 5-step time lags where the conventional method already seems to require hundreds of thousands of training sequences, then why do conventional recurrent networks have a good reputation for being a general tool for learning sequences, grammars etc.? I believe (but cannot prove) that the reason is: All the reported examples for grammar learning work only because there are at least *some* training sequences with very short time lags between relevant events. These 'easy' training sequences help the system to generalize to sequences with longer time lags (see also [27]).

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 one proposed in 5.2: 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 [17] 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).

Similar results were obtained with different numbers of hidden units. It must be mentioned, however, that limited computer time prevented a systematic examination of the effects of various learning rates and various numbers of hidden units: No systematic attempt has been undertaken to optimize performance.

Additional successful experiments with more complicated grammars, with variants of the architecture (sections 5.4 and 5.7) and with up to 100-step time lags were conducted. Much remains to be done, however, to obtain heuristics for determining useful learning rates, error thresholds, etc.

It is intended to apply both multi-level chunking systems (section 3.2.1) 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. In [2] it is shown that conventional recurrent nets can be successfully applied to predicting temporal structure below the phoneme level. But, the same results indicate that conventional recurrent nets seem to perform poorly if the task is to predict inter-phoneme structure.

7 CHUNKING FOR ADAPTIVE CONTROL

There are a few rather obvious extensions of the chunking architectures described above which allow the chunking of action sequences in the context of hierarchical reinforcement learning, hierarchical adaptive control, and look-ahead planning. These extensions are based on two separate chunking systems. One of them uses the chunking algorithm to construct an adaptive model of the environmental dynamics of an agent controlled by a recurrent control network. The second chunking system uses the first one to compute gradients for controller outputs. The whole architecture is an extension of the non-compositional architecture described in [30] and [22]. Together with a few less obvious ideas, the details of these architectures will be described in a separate paper [18].

It should be mentioned that Myers [10], Ring [13] and Watkins [31] also have described (quite different) ideas for hierarchical reinforcement learning.

8 AN ANALOGY TO THE BEHAVIOR OF HUMANS

Observing himself the author can hardly resist the impression that he tends to memorize and focus on non-typical and unexpected events. It seems that expected events often do not even call our attention. It seems that we tend to try to explain new unexpected events by previous unexpected events, hoping to be able to learn to expect similar events in the future. 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. The analogy to the chunking algorithm is rather obvious: Consider the chunker as the 'concious' part of an information processing system. Consider the automatizer as the 'sub-concious' part. The concious part attends to the unexpected events and tries to create high-level explanations for them. Higher-level 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'.

9 CONCLUDING REMARKS

The basic ideas of this paper allow a variety of implementations. Currently it is not clear which particular implementation will perform best within a given context. It has been shown, however, that 'temporal chunking' and 'learning to divide and conquer' with neural networks to a certain extent is possible. This may be considered as another step towards neural systems which learn higher-level abstractions without loosing typical advantages of neural information processing. It also may be viewed as an additional step towards bridging the gap between so-called sub-symbolic and symbolic computation.

One reason for the superiority of chunking systems over conventional learning algorithms for non-stationary environments is probably a very general one which might be expressed as the following recommendation for any kind of adaptive system: If your environment contains regularities of *any* kind, try to detect them and learn to use them for making your problems easier. A general criticism of more conventional goal-directed (supervised and reinforcement learning) algorithms can be formulated as follows: These algorithms do not try to selectively focus on *relevant* inputs, they waste efficiency and ressources by focussing on *every* input.

Future research will concentrate on applying both multi-network chunking systems and 2-network chunking systems to real world tasks (speech processing) and to the ambitious tasks of hierarchical reinforcement learning and adaptive neuro-control.

10 ACKNOWLEDGEMENTS

I would like to thank Josef Hochreiter for conducting the experiments and for providing useful comments on a draft of this paper.

References

[1] Y. Chauvin. Generalization performance of overtrained networks. In L. B. Almeida and C. J. Wellekens, editors, *Proc. of the EURASIP'90 Workshop, Portugal*, page 46. Springer, 1990.

[2] A. Doutriaux and D. Zipser. Unsupervised discovery of speech segments using recurrent networks. In D. S. Touretzky, J. L. Elman, T. J. Sejnowski, and G. E. Hinton, editors, *Proc. of the 1990 Connectionist Models Summer School*, pages 52–61. San Mateo, CA: Morgan Kaufmann, 1990.

[3] J. L. Elman. Finding structure in time. Technical Report CRL Technical Report 8801, Center for Research in Language, University of California, San Diego, 1988.

[4] Josef Hochreiter. Diploma thesis, 1991. Institut für Informatik, Technische Universität München.

[5] M. I. Jordan. Serial order: A parallel distributed processing approach. Technical Report ICS Report 8604, Institute for Cognitive Science, University of California, San Diego, 1986.

[6] A. Lapedes and R. Faber. How neural nets work. In D. Z. Anderson, editor, *'Neural Information Processing Systems: Natural and Synthetic' (NIPS)*. NY, American Institute of Physics, 1987.

[7] Y. LeCun. Une procédure d'apprentissage pour réseau à seuil asymétrique. *Proceedings of Cognitiva 85, Paris*, pages 599–604, 1985.

[8] M. Minsky and S. Papert. *Perceptrons*. Cambridge, MA: MIT Press, 1969.

[9] M. C. Mozer. Connectionist music composition based on melodic, stylistic, and psychophysical constraints. Technical Report CU-CS-495-90, University of Colorado at Boulder, 1990.

[10] C. Myers. Learning with delayed reinforcement through attention-driven buffering. Technical report, Imperial College of Science, Technology and Medicine, 1990.

[11] D. B. Parker. Learning-logic. Technical Report TR-47, Center for Comp. Research in Economics and Management Sci., MIT, 1985.

[12] B. A. Pearlmutter. Learning state space trajectories in recurrent neural networks. *Neural Computation*, 1:263–269, 1989.

[13] M. A. Ring. PhD Proposal. Technical report, University of Texas at Austin, 1991.

[14] 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.

[15] R. Rohwer. The 'moving targets' training method. In J. Kindermann and A. Linden, editors, *Proceedings of 'Distributed Adaptive Neural Information Processing', St.Augustin, 24.-25.5,*. Oldenbourg, 1989.

[16] R. Rohwer. Personal communication, 1990.

[17] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing*, volume 1, pages 318–362. MIT Press, 1986.

[18] J. H. Schmidhuber. Chunking for hierarchical reinforcement learning and adaptive control. Technical Report in preparation, Institut für Informatik, Technische Universität München.

[19] J. H. Schmidhuber. Dynamische neuronale Netze und das fundamentale raumzeitliche Lernproblem. Dissertation, Institut für Informatik, Technische Universität München, 1990.

[20] J. H. Schmidhuber. Learning algorithms for networks with internal and external feedback. In D. S. Touretzky, J. L. Elman, T. J. Sejnowski, and G. E. Hinton, editors, *Proc. of the 1990 Connectionist Models Summer School*, pages 52–61. San Mateo, CA: Morgan Kaufmann, 1990.

[21] J. H. Schmidhuber. A local learning algorithm for dynamic feedforward and recurrent networks. *Connection Science*, 1(4):403–412, 1990.

[22] J. H. Schmidhuber. Making the world differentiable: On using fully recurrent self-supervised neural networks for dynamic reinforcement learning and planning in non-stationary environments. Technical Report FKI-126-90 (revised), Institut für Informatik, Technische Universität München, November 1990. (Revised and extended version of an earlier report from February.).

[23] J. H. Schmidhuber. Recurrent networks adjusted by adaptive critics. In *Proc. IEEE/INNS International Joint Conference on Neural Networks, Washington, D. C.*, volume 1, pages 719–722, 1990.

[24] J. H. Schmidhuber. Talk at the NIPS'90 workshop on dynamic networks led by R. Rohwer, 1990.

[25] J. H. Schmidhuber. Towards compositional learning with dynamic neural networks. Technical Report FKI-129-90, Institut für Informatik, Technische Universität München, 1990.

[26] J. H. Schmidhuber. Adaptive curiosity and adaptive confidence. Technical Report FKI-149-91, Institut für Informatik, Technische Universität München, April 1991.

[27] J. H. Schmidhuber. Adaptive decomposition of time. In O. Simula, editor, *Proceedings of the International Conference on Artificial Neural Networks ICANN 91, to appear*. Elsevier Science Publishers B.V., 1991.

[28] J. H. Schmidhuber. Learning to control fast-weight memories: An alternative to recurrent nets. Technical Report FKI-147-91, Institut für Informatik, Technische Universität München, March 1991.

[29] J. H. Schmidhuber. Learning to generate sub-goals for action sequences. In O. Simula, editor, *Proceedings of the International Conference on Artificial Neural Networks ICANN 91, to appear*. Elsevier Science Publishers B.V., 1991.

[30] J. H. Schmidhuber. Reinforcement learning in markovian and non-markovian environments. In D. Touretzky and D. S. Lippman, editors, *Advances in Neural Information Processing Systems 3, in press*. San Mateo, CA: Morgan Kaufmann, 1991.

[31] C. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, 1989.

[32] P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

[33] P. J. Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1, 1988.

[34] R. J. Williams. Toward a theory of reinforcement-learning connectionist systems. Technical Report NU-CCS-88-3, College of Comp. Sci., Northeastern University, Boston, MA, 1988.

[35] 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.

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