Making the World Differentiable: On Using Self-Supervised Fully Recurrent Neural Networks for Dynamic Reinforcement Learning and Planning in Non-Stationary Environments

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

Abstract

First a brief introduction to reinforcement learning and to supervised learning with recurrent networks in non-stationary environments is given. The introduction also covers the basic principle of 'gradient descent through frozen model networks' *as employed by Werbos, Jordan, Munro, Robinson and Fallside, and Nguyen and Widrow. This principle allows supervised learning techniques to be employed for reinforcement learning.*

Then a general algorithm for a reinforcement learning neural network with internal and external feedback in a non-stationary reactive environment is described. Internal feedback is given by connections that allow cyclic activation flow through the network. External feedback is given by output actions that may change the state of the environment thus influencing subsequent input activations. The network's main goal is to receive as much reinforcement (or as little 'pain') as possible.

In theory, arbitrary time lags between actions and ulterior consequences are possible. The 'visible' environment may be 'non-Markovian'. Although the approach is based on 'supervised' learning algorithms for fully recurrent dynamic networks, no teacher is required. An adaptive model of the environmental dynamics is constructed which includes a model of future reinforcement to be received. This model is used for learning goal directed behavior. The algorithm may concurrently *learn the model and learn to pursue the main goal. To attack certain problems with the parallel version of the algorithm, 'adaptive randomness' is introduced. The algorithm is applied to a reinforcement learning task in a non-Markovian environment.*

A connection to 'meta-learning' (learning how to learn) is noted. An extension of the algorithm is described which includes a vector-valued *adaptive critic element (based on Sutton's TD-methods). The possibility of using the model for learning by planning (by 'mental simulation' of the environmental dynamics) is investigated.*

Finally it is described how the algorithm can be augmented by dynamic curiosity *and* boredom. *This can be done by introducing (delayed) reinforcement for controller actions that increase the model network's knowledge about the world. This in turn requires the model network to model its own ignorance, thus showing a rudimentary form of* introspective *behavior.*

*This work was supported by a scholarship from SIEMENS AG

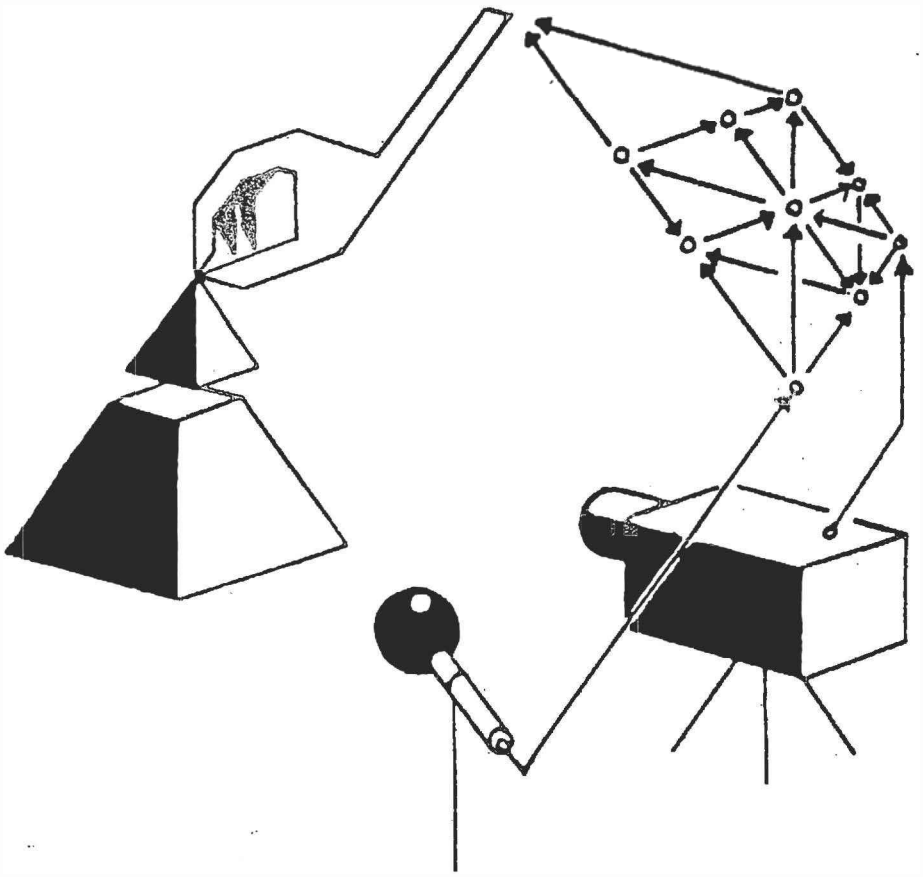

Figure 1: *Internal and external feedback.*

Introduction

***Note:** This is the revised and expanded version of an earlier report from February 1990, which was itself an expanded version of an earlier report from November 1989.*

Consider a dynamic neural network receiving inputs from a non-stationary environment and being able to produce actions that may have an influence on the environmental state. Since the new state may cause new inputs for the system we say that there is *external feedback*. In general, arbitrary time lags may exist between actions and their ultimate consequences in the environment.

If the network is cyclic, then input activations from a given time may alter the way in which inputs from later times are processed. In this case there is a potential for the 'representation of state', or 'short term memory', and we speak of *internal feedback*. In general, arbitrary time lags may occur between inputs to the network and later network consequences.

The principles of internal feedback are usually known (in most cases internal feedback is based on sigmoid transformations of weighted sums). In real world applications less is known *a priori* about the laws that govern the external feedback.

If the network is supposed to learn externally posed tasks then it faces Minsky's *basic credit assignment problem*[14]: If performance is not sufficient, then which component of the network at which time did in which way contribute to failure? How should critical components change behavior to increase future performance? In this report I describe how two 'self-supervised' recurrent networks can interact in

order to attack the fundamental credit assignment problem. The context will be given by reinforcement learning tasks (described below).

We will concentrate on discrete time versions of dynamic learning algorithms for neural networks. We assume that there are 'time steps', and that state changes only take place from one time step to the next one, not within a time step.

Locality in Space and Time. A learning algorithm for dynamic neural networks is said to be ***local in time*** if for given network size (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 said to be ***local in space*** if during on-line learning for limited durations of learned sequences 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 $O(1)$.

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

These definitions do not imply that a local algorithm is unable to consider actions that have taken place any time before the current time step.

Various kinds of tasks differ according to the complexity of the corresponding credit assignment problem. Often the distinction is made between supervised learning tasks and the more difficult reinforcement learning tasks.

Supervised Learning

A learning task is called a ***supervised*** learning task if there are externally defined desired outputs at certain time steps, but the network never needs to discover output actions on its own. This can be the case if there is an external teacher who gives the desired output activations at every time step. (Teaching information also may be given by another network.) Supervised learners need only consider the internal network dynamics for performing credit assignment. These dynamics are given by the ***known*** rules of internal information processing (e.g. sigmoid transformations of weighted activation signals). External feedback is not really important for supervised learners, since by definition they do not need to care for unknown environmental dynamics to achieve their goals. There is no such thing as an undesired input caused by the external feedback. A supervised learner therefore does not need to 'propagate errors through the environment'.

The goal of supervised learning is usually stated as the minimization of cumulative errors observed at the output units over time. This usually is achieved by gradient descent methods:

$$\Delta w = -\eta \frac{\partial \sum_t \|d_t - x_t\|^2}{\partial w},$$

where w is the networks weight vector, Δw is its increment caused by the learning procedure after a training episode (usually a time interval with fixed length during which the inputs and desired outputs are presented), t ranges over all time steps of the activation spreading phase of the episode, d_t is the desired output vector at time t, x_t is the actual output vector at time t, and η is a positive constant.

Straightforward but not general algorithms for dynamic supervised learning have been proposed by Jordan [7] and Elman [4]. General supervised learning algorithms for completely recurrent dynamic networks have been described in [25], [43], and [19]. These algorithms are based on conventional back-propagation (BP) [41] [12] [18] [25] and the 'unfolding in time' principle, which requires that each unit remembers its past activations on a stack. During the backward pass the stored activations are successively popped off their stacks and used to compute the error gradient in the conventional manner.

Robinson and Fallside [22] (see also [21]) were the first to note that a backward pass is not necessary. They described the 'Infinite Input Duration' (IID)-algorithm which minimizes the same cumulative error as the 'unfolding in time' methods, but uses only computations that go 'forward in time'. Their method

has the advantage that for arbitrary sequences of inputs and desired outputs, a fixed amount of storage is sufficient to compute the complete error gradient. A disadvantage of the method is that it is not local in space. Pearlmutter [19] and Gherrity [5] described continuous time versions of the method. Rohwer [24] also described a non-local algorithm for fully recurrent dynamic networks which needs only computations 'going forward in time'.

Williams and Zipser [47] implemented a variant of the IID-algorithm. In their implementation they assume the learning rate to be small enough to allow immediate weight changes during on-line learning. Although this means a deviation from true gradient descent, the important consequence is that there is no need for 'episode boundaries' any more. This property makes the algorithm very interesting for tasks where the external teacher does not structure environmental inputs into sequences that 'belong together'. In some experiments Williams and Zipser showed that such structuring information can be acquired by the learning system itself. Hence, this algorithm is of potential interest for autonomous agents in non-stationary environments.

Reinforcement Learning

A learning task is called a *reinforcement learning task* if the teacher only indicates once in a while whether the system is in a desirable state or not, without giving information about how to reach desirable states. In most cases the teacher occasionally provides a scalar signal, the reinforcement, whose value indicates success or failure. During training the network is supposed to discover on its own the outputs that eventually lead to desirable states. From the standpoint of the reinforcement learner, the nature of the external feedback in most cases is highly significant. In contrast to supervised learning, there can be undesired inputs caused by previous output actions. In general, the external *unknown* dynamics have to be taken into consideration to perform credit assignment. Since the external dynamics are not known *a priori* they have to be explored. An explorative capability can be achieved through units that obey probabilistic activation rules.

Williams [46] described a class of reinforcement learning algorithms for recurrent networks with probabilistic activation rules. These algorithms are suitable for the case where there are predefined episode boundaries. Though they are derived by using the 'unfolding in time' principle, the algorithms themselves do not require an unlimited amount of storage for past activations since they only use computations that go 'forward in time'. And although they are suited for the comparatively difficult reinforcement learning tasks, the algorithms require less peak computation than corresponding supervised methods. One reason for this is that the procedures are 'uninformed' in the sense that comparatively little information is considered for computing the gradient.

Williams showed that these methods, which he called extended REINFORCE algorithms, can be expected to perform hill climbing in a quantity that measures the expectation of the cumulative reinforcement to be received during one training episode. More precisely, for a given w, the inner product

$$E\{\Delta w \mid w\} \frac{\partial E\{\sum_t r(t) \mid w\}}{\partial w}$$

is positive, as long as the second factor is not zero. (Here $r(t)$ is the reinforcement at time t, w is the network's weight vector, Δw is its increment, t ranges over all time steps of a training episode, and E is the expectation operator.) However, it should be noted that Williams' result does not necessarily hold in the case of external feedback.

Another approach for reinforcement learning in recurrent networks, the 'neural bucket brigade algorithm', is described in [28]. This algorithm translates reinforcement given from the environment into 'weight-substance'. By performing only local computations the algorithm establishes recursive dependencies between strengths of connections that transport activation information during successive time steps. The result is a dissipative system which consumes weight substance given in the moments of reinforcement. Weight substance flows through the network in the opposite direction of the activation flow. A competition based on local computations aims at maximizing the amount of reinforcement (= weight substance) to be received.

Barto, Sutton, and Anderson's reinforcement learning algorithm (the AHC algorithm) [3] involves two 'networks' consisting of single units without recurrent connections, thus allowing only external feedback. One of the units is a 'self-supervised' learner that makes predictions about future reinforcement. Differences of successive predictions are used to compute internal reinforcement for the second unit, even in the absence of an external signal. The algorithm may be viewed as an extension of Samuel's early work [26], and bears interesting relationships to the bucket brigade algorithm.

Anderson [1] further extended the approach to systems consisting of feedforward networks. In [32] and [29] the approach was further extended to a combination of a recurrent reinforcement learner and a static supervised learner. Combinations of two interacting recurrent networks based on that principle were also considered.

Using only Supervised Techniques for Reinforcement Learning.

Although connectionist supervised learning techniques generally are not considered to be very fast, conventional reinforcement learning tends to be notoriously slow when compared to supervised learning (in tasks where both learning paradigms are applicable). There have been various approaches to employ the more informed supervised techniques for reinforcement learning tasks. In the next subsections we outline principles of previous approaches based on *system identification* and *gradient descent through frozen model networks.*

Munro

Munro [15] implemented a system identification approach to reinforcement learning that uses supervised learning techniques only. His system consists of two feedforward networks, which we call the control network and the model network, both being trained by conventional back-propagation. The system to be identified by the model network is the external process which maps pairs of situations and controller outputs to reinforcement. There may not be *delayed* reinforcement. The model network has a one-dimensional output and is trained in an exploratory phase to produce the reinforcement corresponding to the inputs and outputs of the control network. This is done by providing random examples of input/output pairs and the corresponding reinforcement. After the model network has learned a sufficient description of these relationships, its weight changing mechanism becomes disengaged, and the training phase of the control network takes place. During this phase differences between actual and desired reinforcement are propagated through the model network down into the control network, where they cause weight changes according to the rules of gradient descent.

Robinson and Fallside

Robinson and Fallside described an extension of Munro's approach to dynamic recurrent networks ([23] [21]). Both the model network and the control network may have internal and external feedback. A significant difference to Munro's approach (besides the recurrency of the networks involved) is that both the model network and the control network learn in parallel. Learning is based on the 'unfolding in time'-method which requires the existence of episode boundaries.

In order to make one network out of two, a single cost function is constructed by linearily combining the difference between the observed reinforcement signal and the predicted reinforcement signal, and the difference between the desired reinforcement signal and the predicted reinforcement signal. A consequence of this approach is that there are less-informed weight changes, since errors for the model network are mixed with errors for the control network.

As in Munro's approach, the only aspect of the external world which is explicitly described by Robinson and Fallside's recurrent model network is the reinforcement's dependence on past inputs and outputs. There is no model for the dependence of (non-reinforcement) inputs on past outputs (or on past inputs which again may have been caused by past outputs). This makes the model for the reinforcement itself incomplete: Paths for credit assignment leading 'through the environment' cannot be considered.

In [8] and [10] Jordan uses a model network for constructing a mapping from output actions of a control network to the environmental effects caused by these actions. His approach again involves two connected networks being trained in two separate phases. Jordan's work emphasizes the role of additional smoothness constraints on the output units of the controller. The output units of the control network, called 'articulatory units', are the input units for the model network. The output units of the model network, called 'target units', feed back to 'state units', which are among the input units for the control network. At every time step, a teacher provides desired states for the target units. In the meanwhile familiar way, errors can be propagated from the target units back through the model network into the control network.

A similar off-line approach (without internal feedback) also was described in Nguyen and Widrow's work (see [17] for an interesting application that does not require internal feedback). A related on-line approach has been applied to difficult problems of attentive vision [37]. There it has been shown that an *imperfect* model network (which does not see the full state of the environment) can nevertheless contribute to *perfect* start/goal trajectories.

Werbos sometimes also employs the system identification approach (e.g. [42]). An overview of adaptive system identification and Werbos' 'Heuristic Dynamic Programming' (related to Barto and Sutton's 'adaptive critic') as well as references to Werbos' earlier relevant work on reinforcement learning and control are given in [44](see also [13]). Barto also gives an overview of algorithms for neural controllers [2].

Viewing Reinforcement as 'Another Type of Input'

In this section we consider reinforcement as a special type of input to a fully recurrent dynamic control network. We dedicate one or more conventional input units for the purpose of reporting the actual reinforcement to the system, so there is a possibility for *multidimensional* reinforcement. We call these units *reinforcement units* or *pain units* and *pleasure units*. Since pain and pleasure units may have an influence on actions to be taken, they possess outgoing connections leading to other units of the control network. In contrast to pure supervised learning where the output units are provided with 'desirable activations', in our case only the reinforcement input units have desirable values at certain times. At each time step the desirable activation of a pain unit is zero. At each time step the desirable activation of a pleasure unit is equal to some predefined positive value. (The activation of a pain unit corresponds to negative reinforcement. The activation of a pleasure unit corresponds to positive reinforcement. The goal is to maximize the reinforcement received over time.)

In the case where there are episode boundaries (later we will try to avoid episode boundaries) the quantity to be minimized is

$$\sum_{t,i}(c_i - r_i(t))^2$$

where $r_i(t)$ is the activation of the ith pain or pleasure unit at time t, c_i is its constant desired value, and t ranges over all time steps of a training episode. (If, for instance, different types of pain have to be weighted in an asymmetric manner, then the quantity to be minimized may be a linear combination of reinforcement signals from different pain units. Such a linear combination can be implemented in a straight-forward manner by a linear unit whose input is the vector of all pain units, and whose fixed weights emphasize certain types of reinforcement. The quantity to be minimized then is the sum of all activations of this linear unit at different times.)

As a complicated example consider an autonomous agent whose movements are controlled by the output units of the control network, and which also receives sensory information through its input units. In the kitchen there is a socket. If the agent manages to put its plug into the socket then its battery will be recharged. In another room there is an oil can. The agent is able to experience different types of undesirable inputs by means of pain units that become activated whenever the agent bumps into

an obstacle with one of its extremities. Another pain unit gets activated whenever the agent's battery charge falls below a certain threshold. Yet another pain unit gets activated whenever the agent's joints begin to rust. The agent is autonomous in the sense that no intelligent external teacher is required to provide goals or subgoals for it.

A pure supervised learning algorithm will not help the autonomous agent to detect appropriate behavior for achieving its goal, namely to exist without ever getting undesirable inputs (corresponding to negative reinforcement). However, a supervised learning algorithm can be employed for training a fully recurrent model network to model the relationships between environmental inputs, output actions of the robot, and corresponding reinforcement. The model network in turn allows the system to compute gradients for 'minimizing pain' and 'maximizing pleasure'.

Since gradients for reinforcement and pain depend on 'credit assignment paths' leading 'backwards through the environment', the model should not only predict the reinforcement units but also the other input units. Here our approach extends the approach of Robinson and Fallside (which bears the most resemblance to ours). The purpose of the adaptive 'model network' is to *'make the whole visible dynamics of the external world differentiable'.*

In Jordan's terminology we may say that the purpose of the model network's 'target units' is to predict activations partly of the conventional input units and partly of the reinforcement units. Only a few of the target units, namely those corresponding to pain units, 'want' to predict zero values. But all target units contribute to credit assignment, as will be seen in what follows.

The adaptive model network bridges the whole 'credit assignment gap' between the output units and the input units of the controller. Since the model network is a fully recurrent one, the model represented by it may be as complete as can be. Unlike with Robinson and Fallside's approach credit assignment paths are provided that lead from pain units back to output units back to all input units and so on. Thus credit assignment is also possible for output units that had an influence on later inputs, which caused new outputs that later caused pain, etc....

Unlike with certain 'adaptive critic' approaches ([3] [1] [13]) we are not limited to 'game-like' *Markovian* environments. Specifically, the model network is potentially able to represent the environmental dynamics even if future inputs are not always fully determined by current activations of the control network's input and output units, providing they can be derived through consideration of past inputs and outputs. This feature allows credit assignment for the controller even in the general case. (Later we will introduce a *combination* of an adaptive critic component and a recurrent model/controller component. Such a combiation may be even more appropriate for certain spatio-temporal credit assignment problems.)

Note that the model network also bridges the credit assignment gap between time-varying activations of the input units. For instance, there are credit assignment paths leading from input units back to other input units, and from there to the output units. These paths are important in the common case when the environment can change even if there are no recent output actions.

Consider figure 2. It shows a control network with internal feedback, which is connected to a robot. It also shows a recurrent network which is supposed to model the dynamics of the environment, including the effects caused by the output units of the control network. The model network receives as input the current activation levels of the control network's input (including the current reinforcement) and output units. The output of the model network at a given time is trained to be equal to the *complete* input of the control network, including the state of the reinforcement units and pain units.

Preliminarily let us assume that the model network has already learned to predict future inputs exactly. The learning procedure can be any supervised learning algorithm for fully recurrent dynamic networks. The training sequences can be chosen from a random distribution of possible sequences of interactions between control network and environment.

Following the system identification approach of Werbos, Jordan, Munro, Robinson and Fallside, and Nguyen and Widrow, we can compute a gradient for the weights of the control network with respect to undesirable activations (errors) of the reinforcement unit(s). In case of existing externally defined 'episode boundaries', this can be done by using the 'unfolding in time' technique: After the activation spreading phase of an episode, in the conventional cyclic manner errors can be propagated from the

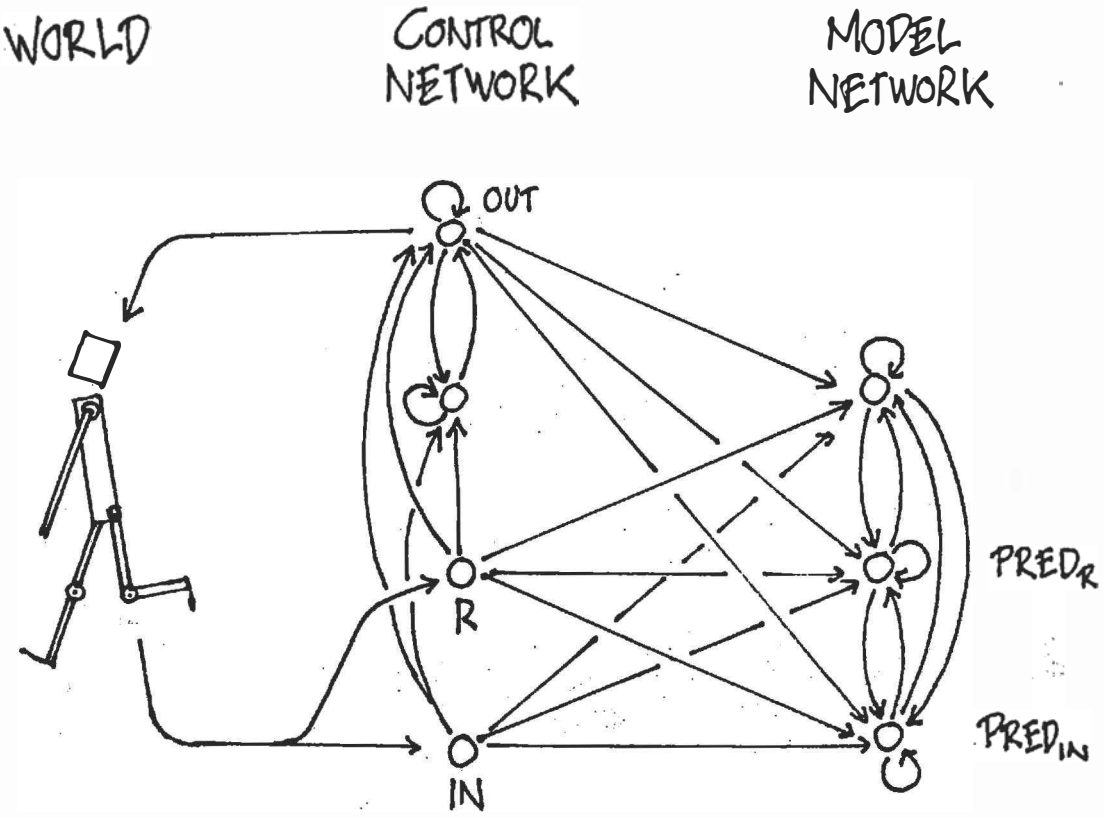

Figure 2: *A control network with internal and external feedback is shown. For simplicity, only one normal input unit (IN), only one reinforcement input unit (R), only one hidden unit and only one output unit (OUT) are depicted. A model network (only one hidden unit is shown) is trained to simulate the environmental dynamics by predicting the control network's input ($PRED_{IN}$ and $PRED_R$). The model network provides credit assignment paths for the control network.*

pain units back through the model network and partly through the control network back into the model network etc., until all activation stacks are empty. Alternatively we can also employ the IID-Algorithm [22] which only needs computations that go forward in time.

Of course, in both cases only weights of the control network may change according to the rules of gradient descent. The weights of the model network have to remain fixed.

Parallel On-Line Learning of Model Network and Controller

The scheme outlined in the last section assumes that the model network has already learned to be a perfect predictor of the environmental dynamics. This requires an exploratory training phase for the model network before training of the control network can start. This approach will be referred to as the *sequential version.*

With the sequential version, the search element that is usually incorporated within reinforcement learning systems through probabilistic activation rules [46] [3] is buried in the exhaustive search of the first phase. In theory all possible kinds of relations between inputs and actions of the control network and subsequent inputs have to be explored. However, this set of possible relations is infinite in general. An alternative approach would employ an external teacher who provides carefully chosen examples of typical event sequences in the environment. Then one might hope that the model network correctly generalizes from these examples to unseen event sequences. In that case the teacher must know more about the environment.

For realistic large scale applications it is highly desireable for the model network and the control network to learn in parallel. (This approach will be referred to as the *parallel version.*) In general, with the sequential version the model network will not be able to explore *all* possible combinations of inputs and actions and their consequences. The control network must therefore start learning with an incomplete representation of the external dynamics in the model network. The model network should concentrate just on those parts of the external dynamics that are necessary for achieving the goals of the control network. Just as Kohonen's self organizing feature maps [11] dedicate more storage capacity to fine grained representations of common similar inputs, the model network should dedicate more storage capacity and time for fine grained modeling of those aspects of the world that are likely to be relevant for the system's main goal.

Besides efficiency, there are other fundamental reasons for considering parallel on-line learning procedures. Consider the evolution of language in the case of two communicating agents, where each agent includes a model of the other's output and its meaning. If their communication ability actually improves then this implies that the outputs and their meanings change. This in turn also requires the respective models to change on-line. In general: Changing environments require changing models.

The parallel on-line version faces other problems as well, due to a trade-off between mathematical exactness and the degree of 'on-line-ness'. If we want an on-line procedure then we will have to deviate from pure gradient descent in several respects. A list of potential problems associated with such deviations from pure gradient descent will follow.

1. Immediate weight changes. One deviation which is common to both the parallel and the sequential version is the following: Instead of accumulating contributions to weight changes over time and actually changing the weights after spreading activation, the weights are changed immediately. Here the assumption is that the model network's learning rate is small enough to avoid instabilities [47]. Immediate weight changes relieve dependence on 'episode boundaries'.

2. Imperfect model networks. A problem which particularily affects the beginning of the learning phase of the parallel version is that the model which is used to compute gradient information for the controller may be wrong. What should we expect to happen if the weights of the control network start changing inappropriately because of an inaccurate model?

2.A.: A situation where the control network experiences pain and where its weights are based on an inaccurate model will not remain stable, as long as the model network and the control network are not both trapped in local minima. *If we assume that the model network always finds a zero-point of its error*

function (which means that it sooner or later will always correctly predict future inputs no matter how the controller behaves), then over time we can expect the control network to perform gradient descent according to a perfect model of the visible parts of the real world.

2.B: The assumption that the model network can always find a zero-point of its error function is not valid in the general case. One of the reasons is the old problem of local minima, for which this paper does not suggest any solutions.

2.C: It should be noted that even before the model becomes perfect performance can improve: Robinson and Fallside's approach [23] to parallel learning was mentioned above. In the context of environments where the environmental state is fully determined by the current input, Jordan [8] also notes that a model network does not need to be perfect to allow increasing performance of the control network.

If the error of the control network is not given by the difference of the desired input (zero pain) for the control network and the model output but by the difference of the desired input and the actual input of the control network, then the minima of this difference still are fixed points of the weight changing mechanism, as long as the model network has already reached a local minimum. The zero-points of the controller's pain are fixed points even if the model network has not yet found a local minimum. The minima of the error for the control network can be found if the inner products of the approximated gradients for the control network's weights and the exact gradients (according to a perfect model) tend to be positive. (See [37] for an application of imperfect models to attentive vision.)

3. Instabilities. One additional source of instability (apart from immediate weight changes *à la* [47]) could arise if the model network 'forgets' information about the environmental dynamics because the activities of the controller push it into a new sub-domain, such that the weights responsible for the old well-modeled sub-domain become over-written.

4. Deadlock. One remaining problem may turn out to be the most serious. Even if the model's predictions are perfect for all actions executed by the controller, this does not imply that the algorithm will always behave as desired. There is the possibility of a special kind of *deadlock*: Let us assume that the controller enters a local minimum relative to the current state of an imperfect model network. This 'relative' minimum *does not have to be a minimum relative to a hypothetical perfect model* of the world. However, it might cause the controller to execute the same action again and again (in a certain spatio-temporal context), such that the model does not get a chance to learn something about *the consequences of alternative actions*. This in turn may cause a state of the model/controller system from which it cannot escape any more (the *deadlock*).

The sequential version of the algorithm described below represents a rather safe way of eliminating these deadlock and instability problems. However, it lacks the flavor of on-line learning and is bound to fail as soon as the environment changes significantly.

To attack the *deadlock* problem of the parallel version we will introduce probablilistic output units for the controller. This requires a modification of the deterministic algorithm (given in the section 'Useful Extensions of the Algorithm').

Dynamic stability problems in general seem to be mathematically quite intractable, since they are domain-dependent. I assume that experiments are needed to find out whether they have to be taken seriously. In the experimental section the sequential version of the algorithm is contrasted with the parallel version.

The Algorithm

The reinforcement learning algorithm described in this section attempts to be a very general one [27]. The quantity to be minimized by the model is $\sum_{t,i}(y_i(t)-y_{ipred}(t))^2$, where $y_i(t)$ is the activation of the ith input unit at time t, and $y_{ipred}(t)$ is the model's prediction of the activation of the ith input unit at time t. The quantity to be minimized by the controller is $\sum_{t,i}(c_i - r_i(t))^2$, where $r_i(t)$ is the activation of the ith reinforcement input unit at time t and c_i is its desired activation for all times. t ranges over all (discrete) time steps. Here the assumption is that what can be learned from the past will be useful

in the future.

There are two different versions of the algorithm: There is the sequential version and the parallel version. With the sequential version, the model network is first trained by providing it with randomly chosen examples of sequences of interactions between controller and environment. Then the model's weights are fixed to their current values, and the controller begins to learn.

With the parallel version both the controller and the model learn concurrently. As mentioned above, the advantage of the parallel version is that the model network focusses only on those parts of the environmental dynamics with which the controller typically is confronted. The disadvantages are the various problems mentioned in the section on parallel learning, particularly the *deadlock* problem.

Below we describe the parallel version (an improved version of the systems described in [30], [33], and the February version of this report). The sequential version can be obtained in a straight-forward manner. An on-line version of the Infinite Input Duration (IID) learning algorithm for fully recurrent networks [22] is employed for training both the model network and the control network. (The IID algorithm was first experimentally tested in [47].)

At every time step, the parallel version of the algorithm is performing essentially the same operations:

In step 1 of the main loop of the algorithm, actions to be performed in the external world are computed. Due to the internal feedback, these actions are based on both current and previous inputs and outputs. For all new activations, the corresponding derivatives with respect to all controller weights are updated.

In step 2 actions are executed in the external world, and the effects of the current action and/or previous actions may become visible.

In step 3 the model network sees the last input and the current output of the controller at the same time. The model network tries to predict the new input without seeing it. Again the relevant gradient information is computed.

In step 4 the model network is updated in order to better predict the input (including reinforcement and pain) for the controller. The weights of the control network are updated in order to minimize the cumulative differences between desired and actual activations of the pain and pleasure units. Since the control network continues activation spreading based on the actual inputs instead of using the predictions of the model network, 'teacher forcing' [47] is used in the model network (although there is no teacher besides the environment). The partial derivatives of the controller's inputs with respect to the controller's weights are approximated by the partial derivatives of the corresponding predictions generated by the model network.

Notation (the reader may find it convenient to compare with [47]):

C is the set of all non-input units of the control network,
A is the set of its output units,
I is the set of its 'normal' input units,
P is the set of its pain and pleasure units,
M is the set of all units of the model network,
O is the set of its output units,
$O_P \subset O$ is the set of all units that predict pain or pleasure,
W_M is the set of variables for the weights of the model network,
W_C is the set of variables for the weights of the control network,
$y_{k_{new}}$ is the variable for the updated activation of the kth unit from $M \cup C \cup I \cup P$,
$y_{k_{old}}$ is the variable for the last value of $y_{k_{new}}$,
w_{ij} is the variable for the weight of the directed connection from unit j to unit i,
δ_{ik} is the Kronecker-delta, which is 1 for $i = k$ and 0 otherwise,
$p^k_{ij_{new}}$ is the variable which gives the current (approximated) value of $\frac{\partial y_{k_{new}}}{\partial w_{ij}}$,
$p^k_{ij_{old}}$ is the variable which gives the last value of $p^k_{ij_{new}}$,
if $k \in P$ then c_k is k's desired activation for all times,
if $k \in I \cup P$, then kpred is the unit from O which predicts k,
α_C is the learning rate for the control network,
α_M is the learning rate for the model network.

$| I \cup P | = | O |$,
$| O_P | = | P |$.

Each unit in $I \cup P \cup A$ has one forward connection to each unit in $M \cup C$,
each unit in M is connected to each other unit in M,
each unit in C is connected to each other unit in C,
each weight variable of a connection leading to a unit in M is said to belong to W_M,
each weight variable of a connection leading to a unit in C is said to belong to W_C,
for each weight $w_{ij} \in W_M$ there are p^k_{ij}-values for all $k \in M$,
For each weight $w_{ij} \in W_C$ there are p^k_{ij}-values for all $k \in M \cup C \cup I \cup P$.

The parallel version of the algorithm works as follows:

INITIALIZATION:
For all $w_{ij} \in W_M \cup W_C$:
begin $w_{ij} \leftarrow$ random,
for all possible k: $p^k_{ij_{old}} \leftarrow 0, p^k_{ij_{new}} \leftarrow 0$ end.
For all $k \in M \cup C$: $y_{k_{old}} \leftarrow 0, y_{k_{new}} \leftarrow 0$.
For all $k \in I \cup P$:
Set $y_{k_{old}}$ according to the current environment, $y_{k_{new}} \leftarrow 0$.

UNTIL TERMINATION CRITERION IS REACHED :

1. *For all* $i \in C : y_{i_{new}} \leftarrow \frac{1}{1+e^{-\sum_j w_{ij} y_{j_{old}}}}$.
   For all $w_{ij} \in W_C, k \in C$:
   $p^k_{ij_{new}} \leftarrow y_{k_{new}}(1 - y_{k_{new}})(\sum_l w_{kl} p^l_{ij_{old}} + \delta_{ik} y_{j_{old}})$.
   For all $k \in C$:
   $y_{k_{old}} \leftarrow y_{k_{new}}$,
   for all $w_{ij} \in W_C : p^k_{ij_{old}} \leftarrow p^k_{ij_{new}}$.
2. *Execute all actions based on activations of units in A. Update the environment.*
   For all $i \in I \cup P$:
   Set $y_{i_{new}}$ *according to environment.*
3. *For all* $i \in M : y_{i_{new}} \leftarrow \frac{1}{1+e^{-\sum_j w_{ij} y_{j_{old}}}}$.
   For all $w_{ij} \in W_M \cup W_C, k \in M$:
   $p^k_{ij_{new}} \leftarrow y_{k_{new}}(1 - y_{k_{new}})(\sum_l w_{kl} p^l_{ij_{old}} + \delta_{ik} y_{j_{old}})$.
   For all $k \in M$:
   $y_{k_{old}} \leftarrow y_{k_{new}}$,
   for all $w_{ij} \in W_C \cup W_M : p^k_{ij_{old}} \leftarrow p^k_{ij_{new}}$.
4. *For all* $w_{ij} \in W_M$:
   $w_{ij} \leftarrow w_{ij} + \alpha_M \sum_{k \in I \cup P}(y_{k_{new}} - y_{kpred_{old}}) p^{kpred}_{ij_{old}}$.
   For all $w_{ij} \in W_C$:
   $w_{ij} \leftarrow w_{ij} + \alpha_C \sum_{k \in P}(c_k - y_{k_{new}}) p^{kpred}_{ij_{old}}$.
   For all $k \in I \cup P$:
   $y_{k_{old}} \leftarrow y_{k_{new}}$, $y_{kpred_{old}} \leftarrow y_{k_{new}}$,
   for all $w_{ij} \in W_M : p^{kpred}_{ij_{old}} \leftarrow 0$,
   for all $w_{ij} \in W_C : p^k_{ij_{old}} \leftarrow p^{kpred}_{ij_{old}}$.

Comments on the algorithm.

1. The on-line algorithm described above is local in time, but not in space. The computation complexity per time step is given by

$$O(\mid W_M \cup W_C \mid\mid M \mid\mid M \cup I \cup P \cup A \mid + \mid W_C \mid\mid C \mid\mid I \cup P \cup C \mid).$$

2. As mentioned above, we use 'teacher forcing' in the model network (step 4). This means that the state of the output units of the model network is replaced by the new inputs to the control network, and all variables storing gradient information for these units are set equal to zero. This is a natural approach, since the control network continues activation spreading based on the actual inputs, instead of using the predictions of the model network. The dynamics of the model network therefore are altered according to the 'real world'. Williams and Zipser describe an experiment where teacher forcing was actually necessary to achieve satisfactory performance [47].

3. In the version above, no teacher forcing is used for the control network. (Teacher forcing would need to continue with zero pain inputs even if there were undesirable activations of the pain units. Here the idea is that a little pain may be informative for the agent, and may have an explicit influence on future actions.)

4. Note that the cumulative updates of the control network's variables for its $\frac{\partial y_{k_{new}}}{\partial w_{ij}}$ values do not require the knowledge of such variables from the model network, but they do require knowlege about unit activations of the model network.

Useful Extensions of the Algorithm

More Network Ticks than Environmental Ticks

The algorithm above assumes that from one time step to the next the environment changes in a fashion that is predictable by linearly separable mappings from past states. If there is a 'higher degree of environmental non-linearity', then the algorithm has to be modified in a trivial manner such that the involved networks perform more than one iteration of step 1 and step 3 at each time step. In any case it suffices if there are four network time steps for each environmental time step. This is due to the fact that 4-layer-operations in principle are enough for an arbitrary approximation of any desired mapping.

Explicit Random Actions versus 'Imported Randomness'

As long as the model is inaccurate, the controller partly functions as a random explorer which uninformedly causes situations that help the model network to collect new data about the environmental dynamics, in order to make the relevant dynamics of the world differentiable. Note that in the version above, the control network does not have any explorative capabilities that are independent from the environment. One might say that randomness is imported from the environment.

To attack the *deadlock* problem with the parallel version of the algorithm we can introduce a probabilistic element for the controller actions. By employing probabilistic output units for C and by using 'gradient descent through random number generators' [45] we can introduce explicit explorative random search capabilities into the otherwise deterministic algorithm. In the context of the IID algorithm, this works as follows: A probabilistic output unit k consists of a conventional unit $k\mu$ which acts as a mean generator and a conventional unit $k\sigma$ which acts as a variance generator. At a given time, the probabilistic output $y_{k_{new}}$ is computed by

$$y_{k_{new}} = y_{k\mu_{new}} + z y_{k\sigma_{new}},$$

where z is distributed e.g. according to the normal distribution. The corresponding $p^k_{ij_{new}}$ must then be updated according to the following rule:

$$p^k_{ij_{new}} \leftarrow p^{k\mu}_{ij_{new}} + \frac{y_{k_{new}} - y_{k\mu_{new}}}{y_{k\sigma_{new}}} p^{k\sigma}_{ij_{new}}.$$

Augmenting the Algorithm by Temporal Difference Methods

So far, unlike 'adaptive critics' the approach does not profit from approximations to *dynamic programming*. The model network's prediction of total future reinforcement is only implicit, the model does not try to make a one-shot prediction of the overall usefulness of the controllers current 'program' (weight matrix). And, unlike TD-based systems [26][39] M does not use *its own prediction* for learning to explicitly predict the *sum* of all future reinforcement: However, it is easy to augment the algorithm with such an 'adaptive critic' method.

To simplify the discussion, let us assume that the only reinforcement units are pain units. (This is just a matter of scaling). The algorithm's goal is to minimize cumulative pain. Now we introduce the TD-principle by changing the error function of the units in O_P:

At a given time t, the contribution of each unit $kpred \in O_P$ to the error of the model network is

$$y_{kpred}(t) - \gamma y_{kpred}(t+1) - y_k(t+1).$$

Here again $y_i(t)$ is the activation of unit i at time t. (One must therefore wait for the prediction of the next time step to compute this part of M's error function.) $0 < \gamma < 1$ is a discount factor which serves to avoid predictions of infinite sums. (This procedure affects the first *For-loop* in step 4 of the algorithm.) By learning to predict the sum of the next pain vector plus the *next prediction* of cumulative pain vectors, O_P is trained to predict the *sum of all* (discounted) future pain vectors.

The controller's goal is to minimize the absolute value of M's pain predictions. Thus, the contribution of time t to the error function of the controller now becomes

$$\sum_{kpred \in O_P} (y_{kpred}(t))^2.$$

This affects the second *For-loop* in step 4 of the algorithm.

It should be noted that unlike the approach described in [3], it is not a *state* which is evaluated by the adaptive critic component, but a *combination of a state and an action.* In this sense the approach resembles the approaches of Watkins [40] and Jordan and Jacobs [9].

Note that we obtain a *multi-dimensional* and *recurrent* adaptive critic in contrast to previous adaptive critics. Since the model network tries to build a model of the context sensitive dependencies of actions and their cumulative negative or positive consequences, *informed* weight changes for the controller may be generated. This again contrasts with previous adaptive critics as described in [3] and [1], which use 'statistical' update rules. Here the current report and [29] converge. (In [29] a pole balancing experiment with a non-recurrent, vector-valued adaptive critic is described.)

It is assumed that model-building adaptive critics have better scaling properties than adaptive critics with statistical update rules [34], particularily when it comes to controllers with multi-dimensional actions. The adaptive critic component is responsible for looking into the future, while the recurrent model/controller component is responsible for looking into the past and for producing 'individually tailored reinforcement signals' (an expression created by Williams [46]). I believe that combinations of adaptive critics and recurrent model/controller systems will often be more appropriate than simple model/controller systems. However, only experiments with the simple version have been conducted so far.

Experiments in Non-Markovian Environments

The algorithm possesses a theoretical potential for reinforcement learning in non-Markovian environments, where at a given time the currently visible part of the world is not enough to make an optimal decision for the next action. Can this potential be realized, particularily if the safer, sequential version is relaxed in favor of the parallel on-line method?

Evolution of a Flip-Flop by Reinforcement Learning

This section demonstrates experimentally that the answer to the above question is 'yes'. Programming and tests were conducted by Josef Hochreiter, a student at TUM.

A controller C had to learn to behave like a flip-flop as described in [47]. C saw a continuous stream of input events. The task was to switch on an output unit whenever an event 'B' occurred for the first time after the last event 'A' had happened. In all other cases the output unit had to be switched off.

One difficulty with the problem was that there could be arbitrary time lags between relevant events. An additional difficulty was that no information about 'episode boundaries' was given. The activations of the networks were never reset. Thus, activations caused by events from past 'episodes' could have a disturbing influence on activation states appearing during later episodes.

The *main difficulty* (the one which makes this different from the supervised approach as described in [47]) was that there was no teacher for C's output units. Instead, the system had to generate alternative outputs in a variety of spatio-temporal contexts, and to build a model of the often 'painful' consequences. C's only goal information was the activation of a pain input unit in proportion to the difference between C's last real-valued action (from the interval $[0 \ldots 1]$) and the correct action (which would not have been punished by the environment). The proportionality factor was 0.5.

Figure 3 shows the topology of the controller/model combination. The controller had 5 input units, 3 of them being 'normal' input units for 3 possible events 'A', 'B', and 'X'. Events were represented in a local manner: At a given time, a randomly chosen normal input unit was activated with a value of 1.0,

the others were de-activated. An additional input unit which was always on provided a modifiable bias for each other unit. Finally there was one pain input unit. The controller's output was probabilistic and based on one unit for variance generation and one unit for mean generation. The contribution v of the variance generator to the probabilistic output was given by its current activation multiplied by

$$-ln(\frac{1}{rnd} - 1),$$

where rnd was a uniformly distributed random variable with values between 0.2 and 0.8. The activation of the ouput node was set equal to 0 for $m + v < 0$, it was set equal to 1 for $m + v > 1$, and it was set equal to $m + v$ in all other cases. (Here m is the current output of the mean generator). The model network M had three hidden units and one output unit for predicting the activations of the pain unit. To save computing time, M did not try to predict the random activations of the other input units.

Test runs for the parallel version and for the sequential version were conducted. In both cases, C performed *one* activation update per time step. M, whose task was more complex (since it had to model the consequences of all types of errors produced by C), was allowed to perform *two* activation updates per time step.

With the sequential version M first was trained for 150000 time steps (a uniform random number generator replaced the controller outputs). Then M's weights were frozen, and 50000 time steps were used for C's training. Both α_C and α_M were equal to 1.0. ***With 6 out of 10 test runs, the algorithm found a working weight matrix for the controller.*** (The weight matrices were inspected 'manually' to determine whether or not they worked correctly.)

Why does it take so much more time solving the reinforcement flip-flop problem than solving the corresponding supervised flip-flop problem [47])? One answer is: With supervised learning the controller gradient is *given* to the system, while with reinforcement learning the gradient has to be *discovered* by the system.

The parallel version also led to useful results (in [6] the behavior of the adaptive variance generator during the learning phase is described). The parallel version was even *more* time consuming than the sequential version. With $\alpha_C = 0.1$ and $\alpha_M = 1.0$ only 20 out of 30 test runs required less than 1000000 time steps to produce a solution.

This example indicates that with small problems the sequential version can be advantageous. It is assumed that a more complex environment would demonstrate the potential advantages of the parallel version. However, our current computer equipment does not allow the time consuming simulation of network combinations much larger than the one described above.

Modifying the algorithm such that C's error was *not* given by the difference between C's desired input (zero pain) and the actual input, but by the difference between C's desired input and M's corresponding prediction, did not lead to significant improvement. The reason was that M never became a *perfect* prophet. Often M tended to predict weak (but in the long term hazardous) pain activations, although C did not actually experience pain. It was important to use the *real* error instead of the predicted error.

Non-Markovian Balancing with a 'Perfect' Model

Instead of training a model network to simulate the environment one can sometimes gain a 'perfect' model by constructing an appropriate mathematical description of the environmental dynamics. This saves the time needed to train the model. However, additional external knowledge is required.

For instance, the description of the environment might be in form of differential or difference equations. In the context of the algorithm above, this means introducing new p^{η}_{ij} variables for each $w_{ij} \in W_C$ and each relevant state variable $\eta(t)$ of the dynamical environment. The new variables serve to accumulate the values of $\frac{\partial \eta(t)}{\partial w_{ij}}$. To update these variables at time t, one must simply differentiate the current environmental state variables with respect to the controller weights, instead of differentiating the activations of the model network (which does not exist any more). This can be done in exactly the same cumulative manner as with the algorithm described above.

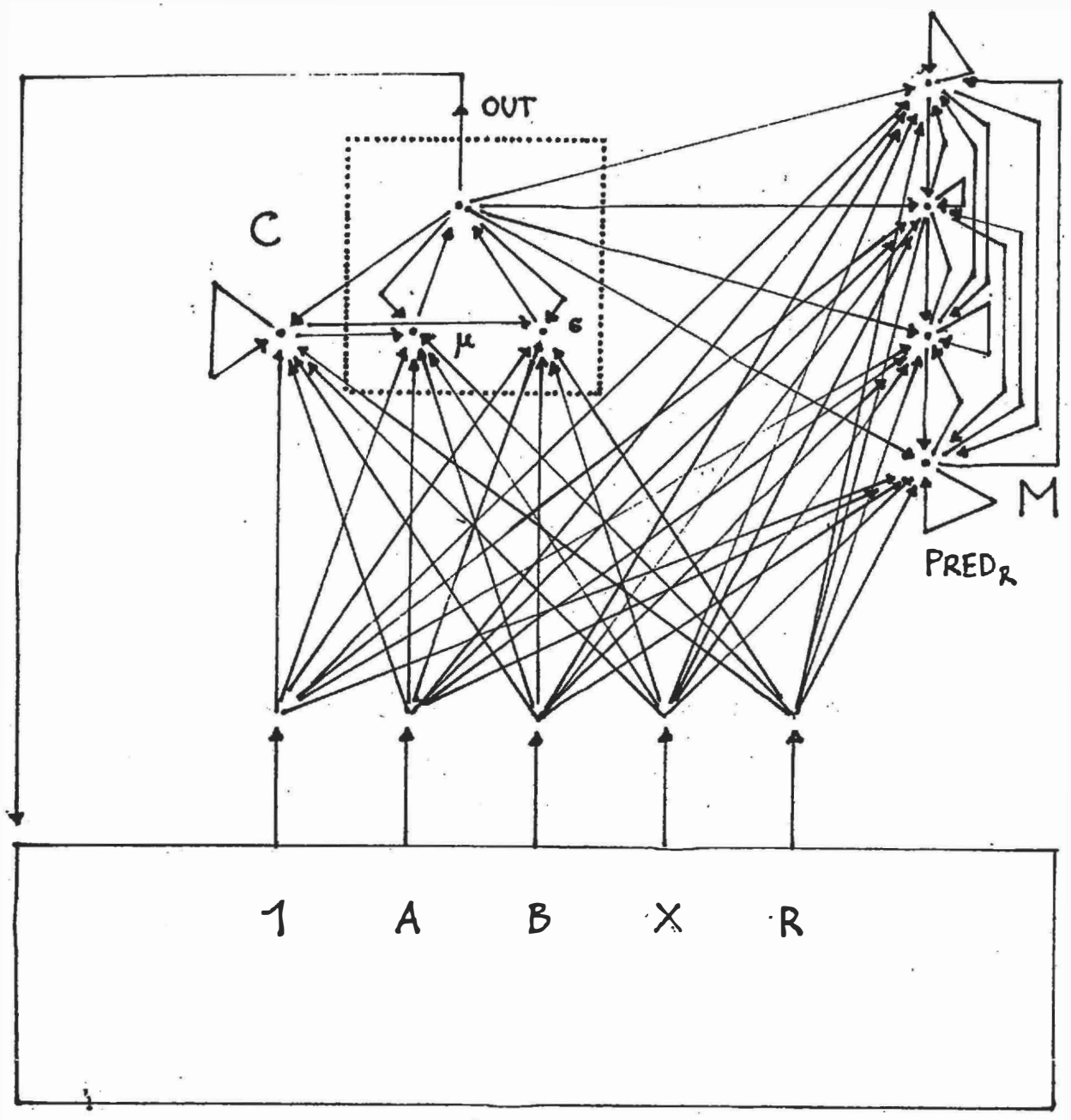

Figure 3: *Topology of the controller/model combination for the flip-flop experiment. The block below represents the environment. The dashed line includes the sub-nodes for mean and variance generation for the controller's output unit.*

Josef Hochreiter introduced these modifications for the special case of a non-Markovian balancing task. The outputs of the control network served to control forces applied to a cart to which a rigid pole was hinged. The cart was able to move along a one-dimensional track. The cart pole system was modeled by the same differential equations (given in the appendix) which were used for a related balancing task described in [3]. The task was to learn to balance the pole for as long as possible without hitting the edges of the track.

The task was difficult, because the input was real-valued (no decoder was used), *and no information about temporal derivatives of z and θ was provided.* (Steven Piché conducted experiments with a similar task [20].) Here we have again an example of an environment whose state is *not* directly visible. The environmental state must be derived by considering past inputs as well. This violates one of the preconditions of Markovian processes, which are essential for certain 'adaptive critic' approaches as described in [3], [1], and [13]. With Markovian processes one need not be concerned with the history that led to a given state in order to find an optimal strategy for future actions. (Most control problems that humans are faced with are *not* Markovian.) However, it should also be noted that in a certain sense the task was less difficult than the flip-flop task: To compute internal representations of pole and angle *velocity* it is not necessary to look back for an *arbitrary* number of time steps. The last few time steps are sufficient in principle.

The control network consisted of 8 non-input units and 3 input units. The inputs were the 2 'visible' scaled state variables $\bar{z}, \bar{\theta}$ (defined in the appendix), and a bias value which was always 1. One of the non-input units (called unit o) also served as the output unit. In step 1 of the algorithm a force of $(50y_{o_{new}} - 25)N$ was applied to the cart. There was no pain unit (although there could have been one, of course). Instead, since M was replaced by a perfect model as described above, the model contained a 'pain variable' which at a given time was activated by the value of

$$\frac{1}{2}((\frac{\theta}{0.21})^2 + (\frac{z}{2.4})^2).$$

Therefore the maximal value of the pain variable was 1 (see appendix). The goal was to minimize the activations of this variable. C performed one iteration per environmental time step. In contrast to the flip-flop experiment, at a given time step it was advantageous *not* to perform credit assignment for all past time steps: All p_{ij}^k-variables of the system were set to zero at every 8th time step. In the beginning of each test run, the weights were randomly initialized between -0.1 and 0.1. In step 2 of the algorithm the input of the continually running recurrent net changed according to a simulation of the cart-pole system (see the appendix) by Euler's method. The frequency of the simulation was $100Hz$, two 'visible' time steps were separated by $0.02s$. For the first time step, as well as for each time step following a failure state, a random state for the cart-pole system was generated according to the following rule: z was randomly chosen from a uniform distribution of all possible positions. θ was randomly chosen from a uniform distribution of all values from the interval $[-0.1, \ldots, 0.1]$. The time derivatives of both state variables were initialized with 0. (The randomness introduced in the beginning of each trial sometimes led to a near-failure state which made it impossible to obtain a long trial.)

In our experiments the cart-pole system would not stabilize indefinitely. However, significant performance improvement was obtained. To test learning performance, the following procedure was carried out: In the beginning of each test run (after weight initialization), 100 trials were conducted with the learning mechanism being disengaged. The average time until failure was about 25 time steps. Then 1000 training trials with $\alpha_C = 1.0$ were conducted. In 17 out of 20 test runs it was possible to obtain (within a few hundred trials) trials with more than 1000 time steps balancing time.

The frozen weight matrix with the longest average balancing time was again tested for 100 additional trials. The average balancing time often had increased significantly. When a trial was started with 'friendly' initial conditions, balancing times of more than 3000 time steps were often achieved, sometimes even many more than that.

Learning to Communicate

In future work I intend to investigate another interesting experiment: a study of the *evolution of language* in the case of two communicating agents, both using the algorithm above. Each agent will not only able to manipulate 'real world' objects but also to produce 'acoustic' outputs and to receive 'acoustic' inputs. Each agent will include a model of the other's output and its meaning. ('Meaning' is only implicitly given by an evaluative critic who judges the effects of what the agents do.) Tasks are given to the agents that cannot be solved by one agent alone such that there is a *need for cooperation*. This means that the agents will be forced to learn to communicate in a sequential manner.

Using the Model Network for Planning Action Sequences

Robinson and Fallside stated that their approach corresponds to Barto, Sutton, and Anderson's 'Adaptive Heuristic Critic' (AHC) algorithm [3], and said that the model network corresponds to the adaptive heuristic critic. A major difference, however, between the AHC and the model network is that the AHC has the potential to immediately look far into the future, while the model network usually looks forward just one time step. The AHC's evaluation of a system's state at time t can become overwritten by its evaluation at time $t+1$. Thus during successive training episodes expectations about future events can be transported 'back into time' for arbitrary numbers of time steps. This is a main motivation behind the adaptive critic/model/controller combination described in the section on 'Useful Extensions of the Algorithm'.

It is also possible to use the model network for predicting events that are 'hidden deep in the future' [30]. The model network, as long as it is perfect, contains all information about future reinforcement. By letting the combination of model network and control network 'run forward in time' for a predefined number of time steps, one can perform a simulation of future events. If such a run predicts pain, then the system can perform gradient descent in predicted pain without actually experiencing pain. This means that an immediate decision can be made about how to change future behavior.

The disadvantage of this is that a lot of computation is required to extract this information. With on-line learning, the consequences are high peak computation times. For instance, if the system at certain time steps plans future actions by looking 10 time steps into the future (without neglecting its usual credit assignment tasks), then it consumes about $10m$ times the amount of computation per step as without such simulation (using essentially the same algorithm for simulation-based weight changing as for normal weight changing), where m is the number of successive simulation repititions required for convergence of the gradient descent procedure.

The reason for this inefficiency is that *every* future event is predicted, not just the *relevant* events. The problem is, of course, to decide in the general case which future events will be relevant for planning, and which will not. (This leads to the old frame-problem of conventional AI.)

Implementing Dynamic Curiosity and Boredom

Only if the model network is a good predictor of the environmental dynamics can we expect the controller to converge. In the current section we motivate the introduction of the explicit desire to improve the world model and show a possibility for implementing it in on-line model-building systems such as the ones described above [31].

Many biological learning systems, particularily the more complex ones, show an interplay of goal-directed learning and explorative learning. In addition to certain permanent goals (like avoiding pain), goals are generated whose immediate purpose seems to be solely an increase of knowledge about the world. So far this interplay has not been addressed at all in the connectionist literature.

The explorative side of learning (related to something that usually is called *curiosity*) is not completely unsupervised, as is sometimes assumed. Curiosity helps to learn how the world works, which in turn

helps to satisfy certain goals. However, the goal-directedness of curiosity is less obvious than the goal-directedness of the algorithm described above (and of less general algorithms described in other papers on goal-directed learning).

Curiosity is related to what one already knows about the world. One gets curious as soon as one *believes that there is something that one does not know.* However, the goal of learning how the world works is dominated by other goals (like avoiding pain): One does not know *exactly* how it feels to put one's hand into a meat grinder; however, one does not *want* to know.

Since curiosity only makes sense for systems that can *dynamically influence what they learn*, and since curiosity aims at minimizing a dynamically changing value, namely, the degree of ignorance about something, it makes sense only in on-line learning situations where there is some sort of *dynamic attention. Thus the precondition of curiosity is something like the parallel version of our on-line learning algorithm described above.* This algorithm builds a world model for goal-directed learning of the controller. The controller's potential for *dynamic attention* is given by the external feedback. The world model adapts itself to whatever the controller focusses on (see [37] for an application of similar adaptive control techniques to the problem of learning selective attention). The direct goal of curiosity and boredom is to improve the world model. The indirect goal is to ease the learning of new goal-directed action sequences. The contribution of this section is to show one possibility for augmenting the algorithm by curiosity and its counterpart *boredom.*

The basic idea is simple: We introduce an additional reinforcement unit for the controller (see figure 4.). This unit, hereafter called the *curiosity unit*, gets activated by a process which at every time step measures the Euclidian distance between reality and prediction of the model network. The activation of the curiosity unit is a function of this distance. Its desired value is a positive number corresponding to the *ideal mismatch* between belief and reality. The effect of the algorithm described in the first section is that there is *positive reinforcement* whenever the model network *fails* to correctly predict the environment. Thus the usual credit assignment process for the controller encourages certain past actions in order to repeat situations similar to the mismatch situation.

As soon as the model network has learned to correctly predict the environment in former 'mismatch situations', actions leading to such situations are automatically weakened. This is because the activation of the curiosity unit goes back to zero. *Boredom* becomes associated with the corresponding situations.

The same complex mechanism which is used for 'normal' goal-directed learning is used for implementing curiosity and boredom. There is no need *for devising a* separate *system for improving the world model.*

The controller's credit assignment process is aimed at repeatedly entering situations where the model network's performance is not optimal. *It is important to observe that this process itself makes use of the model network!* The model network has to predict the activations of the curiosity unit. Thus the model network partly has to model its own ignorance, it has *to learn to know that it does not know* certain details.

What is the *ideal mismatch* mentioned above? In conventional AI the saying goes that a system can not learn something that it does not already *almost know.* A consequence of this is that the function which translates mismatches into reinforcement should not be a linear one. *Zero reinforcement should be given in case of perfect matches, high reinforcement should be given in case of 'near-misses', and low reinforcement again should be given in case of strong mismatches.* This corresponds to a notion from 'esthetic information theory' which tries to explain the feeling of 'beauty' by means of the quotient of 'subjective complexity' and 'subjective order' or the quotient of 'unfamiliarity' and 'familiarity' (measured in an information-theoretic manner). This quotient should achieve a certain ideal value. (See Nake [16] for an overview of approaches to formalizing 'esthetic information'. Interestingly, the number $\frac{1}{e}$ plays a significant role in at least some of these approaches.) However, at the moment, the precise nature of a good mapping between (mis)matches and reinforcement is unclear and subject of ongoing research.

Our currently experimental research is aimed at answering the following questions: What are useful learning rates? (The model network should clearly learn faster than the controller.) What are useful relative strengths of pure goal-directed reinforcement and 'curiosity reinforcement'? And what are the

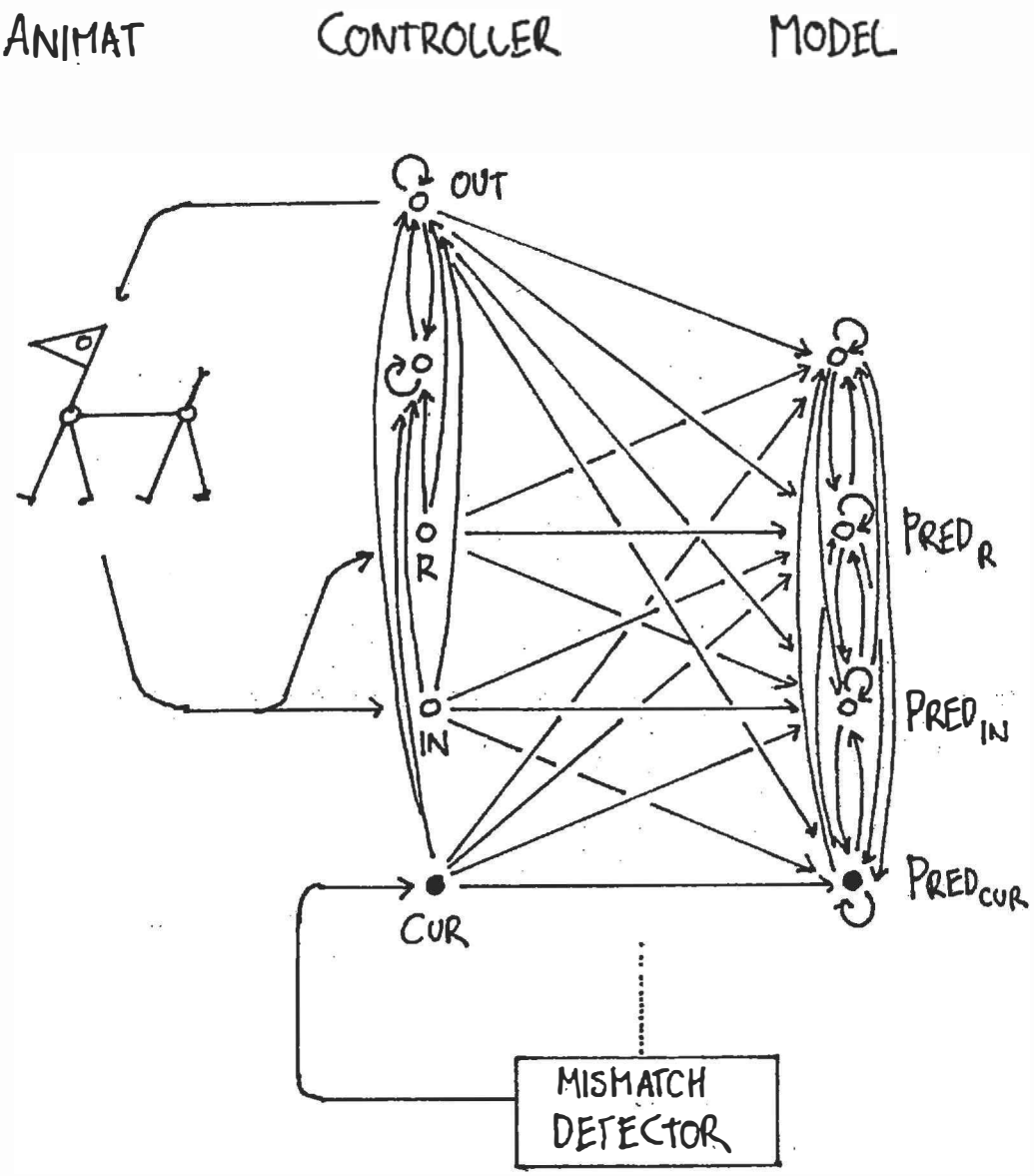

Figure 4: *This figure is derived from figure 2. An additional reinforcement unit CUR for the control network gets activated by 'ideal mismatches' between expectations of the model network and reality. The model network needs an additional output unit ($PRED_{CUR}$) for predicting CUR. It models its own ignorance, thus showing a rudimentary form of introspective behavior.*

properties of a good mapping from mismatches to reinforcement?

Although these questions are still open, in some preliminary experiments with a *linear* mapping from mismatches to reinforcement it has already been demonstrated that errors of the model network can be reduced by generating curiosity reinforcement in an on-line manner.

It should be mentioned that the basic idea of implementing curiosity and boredom is not limited to the particular algorithm described in the first section. *Every* model-building on-line algorithm for learning goal directed behavior might be augmented by a similar implementation of the desire to improve the world model. The basic motivation is: Instead of using some separate mechanism for improving the world model, we want to make use of the capabilities of the goal-directed learning algorithm itself.

In the context of learning algorithms as above, the interesting side effect is: Since the learning algorithm depends on the model network, the model network has to make a prediction about its own current prediction capabilities. The *activations* of the model network are (partly) interpreted as a statement about the quality of the current *weights* of the model network. Note that this is a rudimentary form of *introspective* behavior.

A Connection to Meta-Learning.

In the context of the previous section, a very interesting aspect of the notion of model networks should be mentioned. A model network can be used not only for predicting the controller's inputs but also for predicting its future outputs. A perfect model of this kind would model the internal changes of the control network. It would predict the evolution of the controller, and therby the effects of the gradient descent procedure itself. In this case, the flow of *activation* in the model network would model the *weight changes* of the control network. This in turn comes close to the notion of 'learning how to learn'. I believe that extensions of these rudimentary forms of introspective neural algorithms will be the key to learning systems which are much more sophisticated than the ones we know so far. However, although such concepts of 'meta-learning' are interesting by themselves and also potentially useful for systems with introspective capabilities, their consequences are beyond the scope of this paper.

Concluding Remarks

Program Inputs Differentiable with Respect to Programs

Let us view a network with a fixed topology as a computer. Its *program* is the weight matrix. One of the most interesting aspects of many connectionist algorithms is that program outputs are differentiable with respect to programs. A simple program generator (the gradient descent procedure) produces increasingly successful programs if the desired outputs are known.

In typical reinforcement learning situations, the environment is not *a priori* represented in a differentiable form. So the main reason for building connectionist world models in the style above is to 'make the world differentiable'. Thus even *program inputs* can become differentiable with respect to programs. World models thereby close the gap between outputs and inputs. A differentiable world model allows the program generator to perform an informed search for better goal directed programs.

The degree of informedness of this search for suitable programs is a principle difference between the approach presented in this paper and the reinforcement learning algorithms for recurrent nets in the style of Williams [46] or Schmidhuber [28] [35]. This degree of informedness also is not present in the system which bears the most relationships to our approach, namely, Robinson and Fallside's two-network-reinforcement-learner. As described above, they also model the reinforcement's dependence on past inputs and outputs, but their model is comparatively incomplete: Many credit assignment paths through the environment are lacking. The more general on-line approach described in this paper views reinforcement as another type of input, where all inputs have to be modelled, and where a few have desired (zero) activations at every step. This approach is based on the idea that understanding the world can greatly reduce the complexity of the search for adequate goal directed behavior.

Limitations

1. The algorithm is not local in space.

2. As with all gradient descent algorithms for complex non-linear error functions, there is the problem of local minima. This paper does not offer any solutions to this problem.

3. Even if the system described in this paper is augmented by probabilistic output units for the controller, deterministic reactions from the environment are assumed. In case of non-deterministic environments, the model network has a tendency to predict average values for non-deterministic inputs. This is not always what we want: There is no guarantee that gradient descent 'through the model network' will still make sense in this case.

A solution to this problem might be to employ differentiable random number generators for the output units of the model network itself, in order to approximate the 'true' context-sensitive probabilities of certain inputs. However, this possibility has not yet been investigated.

3. More severe limitations of the algorithm are the *inherent problems of the concepts of 'gradient descent through time' and adaptive critics themselves.* I have argued that neither gradient descent nor adaptive critics are practical for large scale dynamic problems where there are *long* time lags between actions and ultimate consequences [36]. Gradient descent procedures always consider *all* past events for credit assignment, which is too much. They do not selectively concentrate on certain *relevant* past events. They do not aim at incrementally composing long control sequences from short ones. They do not aim at 'dividing and conquering', and they do not have the concept of something like a *sub-program.* For this reason, first steps are made in [36] towards *adaptive sub-goal generators* and *adaptive 'causality detectors'.*

Acknowledgements

I wish to thank Josef Hochreiter who conducted the experiments.

I am grateful to the following persons (in alphabetical order) who gave useful comments on earlier versions of this report or on ideas contained in publications based on this report: Klaus Bergner, Josef Hochreiter, Mark Ring, Rich Sutton, Gerhard Weiss, and Ron Williams.

This work was supported by a scholarship from SIEMENS AG.

Appendix: Details of Cart-Pole Simulation

The cart-pole system, taken from [3], [38], and [1], was modeled by the equations

$$\ddot{\theta} = \frac{g sin\theta + cos\theta \frac{-F - ml\dot{\theta}^2 sin\theta + \mu_c sgn(\dot{z})}{m_c + m} - \frac{\mu_p \dot{\theta}}{ml}}{l(\frac{4}{3} - \frac{mcos^2\theta}{m_c + m})},$$

$$\ddot{z} = \frac{F + ml(\dot{\theta}^2 sin\theta - \ddot{\theta} cos\theta) - \mu_c sgn(\dot{z})}{m_c + m}$$

where $-0.21 < \theta < 0.21$ (angle of pole with the vertical), $-2.4m < z < 2.4m$ (position of cart on track), $g = 9.8\frac{m}{s^2}$ (gravitational acceleration), $m_c = 1kg$ (mass of cart), $m = 0.1kg$ (mass of pole), $l = 0.5m$ (half pole length), $\mu_c = 0.0005$ (coefficient of friction of cart on track), $\mu_p = 0.000002$ (coefficient of friction of pole on cart), $F \in [-25N, 25N]$ (force applied to cart's center of mass, parallel to track). (Note that there is a typing error in the equations given in [3], [38], and [1]: There the gravitational constant is given as $g = -9.8\frac{m}{s^2}$).

The two scaled input variables were $\bar{z} = \frac{1}{2} + \frac{1}{4.8}z$ and $\bar{\theta} = \frac{1}{2} + \frac{1}{0.42}\theta$.

References

[1] C. W. Anderson. *Learning and Problem Solving with Multilayer Connectionist Systems.* PhD thesis, University of Massachusetts, Dept. of Comp. and Inf. Sci., 1986.

[2] A. G. Barto. Connectionist approaches for control. Technical Report COINS Technical Report 89-89, University of Massachusetts, Amherst MA 01003, 1989.

[3] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13:834–846, 1983.

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

[5] M. Gherrity. A learning algorithm for analog fully recurrent neural networks. In *IEEE/INNS International Joint Conference on Neural Networks, San Diego*, volume 1, pages 643–644, 1989.

[6] Josef Hochreiter. Implementierung und Anwendung eines 'neuronalen' Echtzeit-Lernalgorithmus für reaktive Umgebungen (*in Vorbereitung*), 1990. Fortgeschrittenenpraktikum, Institut für Informatik, Technische Universität München.

[7] 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.

[8] M. I. Jordan. Supervised learning and systems with excess degrees of freedom. Technical Report COINS TR 88-27, Massachusetts Institute of Technology, 1988.

[9] M. I. Jordan and R. A. Jacobs. Learning to control an unstable system with forward modeling. In *Proc. of the 1990 Connectionist Models Summer School, in press.* San Mateo, CA: Morgan Kaufmann, 1990.

[10] M. I. Jordan and D. E. Rumelhart. Supervised learning with a distal teacher. Technical Report, Massachusetts Institute of Technology, 1990.

[11] T. Kohonen. *Self-Organization and Associative Memory.* Springer, second edition, 1988.

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

[13] G. Lukes, B. Thompson, and P. Werbos. Expectation driven learning with an associative memory. In *Proc. IEEE International Joint Conference on Neural Networks, Washington, D. C.*, volume 1, pages 521–524, 1990.

[14] M. Minsky. Steps toward artificial intelligence. In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 406–450. McGraw-Hill, New York, 1963.

[15] P. W. Munro. A dual back-propagation scheme for scalar reinforcement learning. *Proceedings of the Ninth Annual Conference of the Cognitive Science Society, Seattle, WA*, pages 165–176, 1987.

[16] F. Nake. *Ästhetik als Informationsverarbeitung.* Springer, 1974.

[17] Nguyen and B. Widrow. The truck backer-upper: An example of self learning in neural networks. In *IEEE/INNS International Joint Conference on Neural Networks, Washington, D.C.*, volume 1, pages 357–364, 1989.

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

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

[20] S. W. Piché. Draft: First order gradient descent training of adaptive discrete time dynamic networks. Technical report, Dept. of Electrical Engineering, Stanford University, 1990.

[21] A. J. Robinson. *Dynamic Error Propagation Networks.* PhD thesis, Trinity Hall and Cambridge University Engineering Department, 1989.

[22] 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.

[23] T. Robinson and F. Fallside. Dynamic reinforcement driven error propagation networks with application to game playing. In *Proceedings of the 11th Conference of the Cognitive Science Society, Ann Arbor*, pages 836–843, 1989.

[24] 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.

[25] 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.

[26] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, 3:210–229, 1959.

[27] J. H. Schmidhuber. Learning algorithms for networks with internal and external feedback. In *Proc. of the 1990 Connectionist Models Summer School, in press.* San Mateo, CA: Morgan Kaufmann, 1990.

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

[29] J. H. Schmidhuber. Networks adjusting networks. In J. Kindermann and A. Linden, editors, *Proceedings of 'Distributed Adaptive Neural Information Processing', St.Augustin, 24.-25.5. 1989*, pages 197–208. Oldenbourg, 1990. In November 1990 a revised and extended version appeared as FKI-Report FKI-125-90 (revised) at the Institut für Informatik, Technische Universität München.

[30] J. H. Schmidhuber. An on-line algorithm for dynamic reinforcement learning and planning in reactive environments. In *Proc. IEEE/INNS International Joint Conference on Neural Networks, San Diego*, volume 2, pages 253–258, 1990.

[31] J. H. Schmidhuber. A possibility for implementing curiosity and boredom in model-building neural controllers. In *Proc. of the International Conference on Simulation of Adaptive Behavior: From Animals to Animats, in press.* MIT Press/Bradford Books, 1990.

[32] 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.

[33] J. H. Schmidhuber. Reinforcement learning with interacting continually running fully recurrent networks. In *Proc. INNC International Neural Network Conference, Paris*, volume 2, pages 817–820, 1990.

[34] J. H. Schmidhuber. Response to G. Lukes' review of 'Recurrent networks adjusted by adaptive critics'. *Neural Network Review, in press*, 1990.

[35] J. H. Schmidhuber. Temporal-difference-driven learning in recurrent networks. In R. Eckmiller, G. Hartmann, and G. Hauske, editors, *Parallel Processing in Neural Systems and Computers*, pages 209–212. North-Holland, 1990.

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

[37] J. H. Schmidhuber and R. Huber. Learning to generate focus trajectories for attentive vision. Technical Report FKI-128-90, Institut für Informatik, Technische Universität München, 1990.

[38] R. S. Sutton. *Temporal Credit Assignment in Reinforcement Learning.* PhD thesis, University of Massachusetts, Dept. of Comp. and Inf. Sci., 1984.

[39] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

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

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

[42] P. J. Werbos. Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research. *IEEE Transactions on Systems, Man, and Cybernetics*, 17, 1987.

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

[44] P. J. Werbos. Backpropagation and neurocontrol: A review and prospectus. In *IEEE/INNS International Joint Conference on Neural Networks, Washington, D.C.*, volume 1, pages 209–216, 1989.

[45] R. J. Williams. On the use of backpropagation in associative reinforcement learning. In *IEEE International Conference on Neural Networks, San Diego*, volume 2, pages 263–270, 1988.

[46] 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.

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