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.

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

Abstract

This paper gives an overview of some novel algorithms for reinforcement learning in non-stationary possibly reactive environments. I have decided to describe many ideas briefly rather than going into great detail on any one idea. The paper is structured as follows: In the first section some terminology is introduced. Then there follow five sections, each headed by a short abstract. The second section describes the entirely local 'neural bucket brigade algorithm'. The third section applies Sutton's TD-methods to fully recurrent continually running probabilistic networks. The fourth section describes an algorithm based on system identification *and on two interacting fully recurrent 'self-supervised' learning networks. The fifth section describes an application of adaptive control techniques to adaptive attentive vision: It demonstrates how 'selective attention' can be learned. Finally, the sixth section critisizes methods based on system identification and adaptive critics, and describes an adaptive subgoal generator.*

1 Terminology

External feedback. Consider a 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 network we say that there is *external feedback*.

Internal feedback. If the network topology is cyclic, then input activations from a given time may alter the way that 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*.

Dynamic Learning Algorithms and Networks. A problem that requires credit assignment to past activation states is called a *dynamic problem*. Learning algorithms for handling dynamic problems are called *dynamic learning algorithms*. Learning algorithms that are not dynamic algorithms are called *static algorithms*. For instance, all algorithms that require settling into equilibria while the inputs have to remain stationary are considered to be static algorithms, although the settling process is a dynamic one based on internal feedback.

If a given network type can be employed for dynamic problems, and if there exists a corresponding learning algorithm, then we sometimes speak of a *dynamic network*.

The credit assignment problem. If a neural network is supposed to learn externally posed tasks then it faces Minsky's *fundamental credit assignment problem*: If performance is not sufficient, then which component of the network at which time did in which way contribute to the failure? How should critical components change behavior to increase future performance?

*This work was supported by a scholarship from SIEMENS AG

Supervised Learning. A learning task is a *supervised* learning task if there are externally defined desired outputs at certain times, but the network never needs to discover output actions on its own. Supervised learners have to consider only the internal feedback for performing credit assignment.

Reinforcement Learning. A learning task is 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. Usually an evaluative (non-instructive) teaching mechanism sometimes provides a scalar signal, the reinforcement, whose value indicates success or failure. During training the network is supposed to discover on its own outputs that eventually lead to desirable states. In contrast to supervised learning, there can be something like *undesired inputs* caused by former output actions. In general the external *unknown* dynamics have to be taken into consideration to perform credit assignment.

Reinforcement learning is strongly related to *control tasks.* With many control tasks more information is available about goal states than just a simple reinforcement signal. However, just as with reinforcement learning, the (sequential) outputs necessary to achieve the goal states in general are not known.

In the sequel 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.

A weak definition of 'locality in space and time' (there also is a stronger definition). 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 $O(1)$.

A learning algorithm for dynamic neural networks is *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.

In the sequel some novel learning algorithms designed for networks with internal and external feedback will be described. Due to limited space I will describe many ideas briefly rather than going into great detail on any one idea.

2 The Neural Bucket Brigade Algorithm

Abstract. Competitive Learning 'shifts weight substance' from certain incoming connections of a winner-take-all-unit to other incoming connections. A novel algorithm for goal directed learning *with hidden units shifts weight substance from* outgoing *connections to incoming connections. An evaluative critic sometimes provides weight-substance for connections leading to output units. The algorithm's most significant advantage over other goal directed learning algorithms like back-propagation (Werbos, 1974)(Parker, 1985)(LeCun, 1985)(Rumelhart et al., 1986) is: It is biologically more plausible, because* it solely depends on computations which are entirely local in space and time. *It has been successfully applied to some classical non-linear problems involving both feedforward and recurrent networks.*

Competitive Learning (heavily employed in work on unsupervised learning (Kohonen, 1988) (Grossberg, 1976)) may be interpreted as 'shifting weight substance' from certain incoming connections of a winner-take-all-unit to other incoming connections (Rumelhart and Zipser, 1986). A novel algorithm for goal directed learning with hidden units emerges if weight substance is shifted from *outgoing* connections to incoming connections in a certain fashion.

Consider the general reinforcement learning situation where an evaluative critic in the environment sometimes provides 'payoff' in response to successful behavior of a learning feedforward or recurrent network. We translate reinforcement or payoff into weight-substance for connections leading to output units that were active in the moment of payoff. All such connections are immediately strengthened proportional to their last contributions. (A contribution is the product of a weight and an activation.)

However, even in the absence of payoff there are weight changes for all weights, including the weights of connections leading to hidden units: Any connection transporting activation information from an active unit i to another active unit j has to give up a part of its weight substance, which is shifted to those weights that were setting the stage by contributing to the activation of unit i at the last time step. Thus recursive dependencies 'through time' are established between strengths of connections transporting contributions during successive time steps. The environmental critic terminates the recursion. The algorithm shares certain conceptual similarities with the 'bucket brigade learning algorithm' for rule-based systems (Holland, 1985) and is called the 'Neural Bucket Brigade Algorithm'. One of the many differences is that competition works locally instead of globally.

The algorithm's most significant advantage over other goal directed learning algorithms like back-propagation is: *It solely depends on computations which are entirely local in space and time. This means that during on-line learning the peak computation per connection is not affected by network size or by network topology or by the length of input sequences. It is always O(1). This makes it biologically more plausible than other algorithms* (Schmidhuber, 1989a) (Schmidhuber, 1989b).

The basic network structure is an arbitrary (possibly cyclic) graph which is partitioned into input units and small predefined winner-take-all-subsets, each having at least two members.

Notation: $x_j(t)$ is the activation of the jth unit at time t, $w_{ij}(t)$ is the weight on the directed connection from unit i to unit j at time t. $c_{ij}(t) = x_i(t-1)w_{ij}(t-1)$ denotes the 'contribution' of some connection between i and j at time t.

In the beginning weights are initialized with a positive value. The system is continuously receiving inputs, and continuously producing outputs, which again may have an influence on subsequent inputs (external feedback). Activations spread according to the following rules: At time t the input-units are clamped to values determined by the environment. Each non-input unit computes $net_j(t) = \sum_i c_{ij}(t)$. The winner-take-all-subsets ensure that only a fraction of the non-input units can be active simultaneously: $x_j(t)$ equals 1 if the non-input unit j is active, and 0 otherwise.

If unit j is active then its positive modifiable weights change according to

$$\Delta w_{ij}(t) = -\lambda c_{ij}(t) + \frac{c_{ij}(t-1)}{net_j(t-1)} \sum_{k\,active} \lambda c_{jk}(t) + \eta c_{ij}(t)$$

where $0 < \lambda < 1$ determines how much of its weight some particular connection has to pay to those connections that were responsible for *setting the stage* at the previous time step. η is a small constant if unit j is an output unit and if there is external payoff, and η is 0 otherwise. We get a dissipative system: 'Weight substance' enters the system in the case of payoff, flows through 'bucket brigade chains', and leaves the system through connections coming from input units.

Note again that the algorithm is entirely local. This makes a parallel implementation trivial. No teacher has to define something like beginnings and ends of back-propagation phases. No storage is required for past activations or contributions except for the most recent ones. The units do not care whether they are part of a feedforward or of a recurrent network. They do not care for concepts like 'layer structure' or network topology. *Each unit and each connection is performing the same simple operation at every time step.*

The algorithm has been successfully applied to some classical non-linear problems involving both feedforward and recurrent networks. Networks employing that algorithm learned to solve XOR-problems, encoding-problems, and sequence recognition (motion on a one-dimensional 'retina') as well as sequence generation (an oscillation task). To address the question of learning speed: The number of training cycles necessary to find some (not necessarily stable) solution for the XOR-problem is of the same order of magnitude as with conventional back-propagation. However, with the more complex encoding problems back-propagation seems to be faster by about an order of magnitude.

3 A Reinforcement Comparison Algorithm for Continually Running Fully Recurrent Probabilistic Networks

Abstract. The principle of reinforcement comparison (employed for learning to play checkers (Samuel, 1959) and learning to balance a pole (Barto et al., 1983)) says: Let the temporal derivative of the expectation of future reinforcement be the effective reinforcement. This principle is applied to fully recurrent continually running networks of probabilistic binary units. A main advantage of the resulting novel algorithm is its applicability to networks with internal (and possibly external) feedback and its locality in both space and time *(the absence of back-propagation-like operations makes it biologically more plausible than other algorithms).*

In addition to a fully recurrent continually running network with probabilistic binary output units the algorithm described in this section employs a second *linear* static network, called the critic, which learns to judge successive states of the recurrent network by learning to predict the final reinforcement to be received at the end of the current 'episode'. Differences of successive predictions serve to adjust both the critic and the recurrent network. Hereby the weights of the critic are updated according to the principles of Temporal Difference Methods (Sutton, 1988):

First all weights are randomly initialized with real values.

For all episodes:

In the beginning of each episode, at the first time step, the activations of input units of the recurrent network are initialized with values determined by sensory perceptions from the environment, and the activations of hidden and output units are initialized with 0. *For all following time steps, until there is external real-valued reinforcement R indicating failure or success:*

At any given time step t:

1. The critic's output $r = x^T(t-1)v(t)$ is interpreted as a prediction of the final reinforcement to be received in the future. ($v(t)$ is the the critics current weight vector, $x(t)$ is the activation vector of all units of the recurrent network).

2. Each probabilistic non-input unit i of the recurrent net sums its weighted inputs, this sum is passed to the logistic function $l(x) = \frac{1}{1+e^{-x}}$ which gives the probability that the activation $x_i(t)$ becomes 1, *or* 0, *respectively. Each unit i also stores its last activation $x_i(t-1)$. Output units may cause an action in the environment, and this may lead to new activations for the input units. So besides the internal feedback there may exist external feedback through the environment.*

3. If there is external reinforcement R (this means the end of the current episode) then the variable r' is defined to be equal to R.

Otherwise r' is defined to be a new estimation of final discounted reinforcement: $r' = \gamma x^T(t)v(t)$. ($0 < \gamma < 1$ is the discount rate).

The critic associates the last activation vector $x(t-1)$ of the recurrent network with r', thus 'transporting expectation back in time' for one time step. So the critic's error is given by $r' - r$. Its weight vector is immediately updated according to the Widrow-Hoff rule; the result is a new weight vector $v(t+1)$.

4. Each directed weight $w_{ij}(t)$ from unit i to unit j of the recurrent network is immediately altered according to $\Delta w_{ij}(t) = \lambda(r'-r)x_i(t-1)(x_j(t)-P(x_j = 1 \mid x(t-1), w(t-1))$, where $w(t-1)$ is the last weight vector, and λ is a positive constant. Thus the last transition gets encouraged (or discouraged, respectively).

The algorithm applies the principle of reinforcement comparison to dynamic recurrent neural networks (Schmidhuber, 1990c). Informally, this principle also can be formulated as follows:

If a system is in a state which it assumes to be a bad state, but there is a transition which leads to a state assumed to be a good state, then this transition should be encouraged. Furthermore, from now on the 'bad' state also can be considered to be a good state. Transitions from good states to bad states have to be treated in an analogue fashion.

Note again that *unlike with back-propagation-like algorithms for recurrent networks the algorithm above is local in both space and time.* This means that during on-line learning *the peak computation per connection is not affected by network size or input duration. It is always O(1).*

It is worth mentioning a counterintuitive fact: The critic may be linear, however, the task of the recurrent network may be of the non-linearily separable type. This has been shown by successfully applying the algorithm to a 'delayed XOR-problem': A reinforcement signal given in the end of each training episode (involving a small number of time steps) indicated whether the recurrent network correctly computed the *delayed* response to one of the four XOR patterns. The critic may be linear, because the final mapping to be implemented by the critic in general is simpler than the final mapping to be implemented by the main network.

The algorithm shares certain conceptual similarities with the 'neural bucket brigade algorithm' (Schmidhuber, 1990e). In (Schmidhuber, 1990c) it also has been described how a *recurrent* critic can interact with the recurrent primary network.

4 Two Interacting Fully Recurrent Self-Supervised Learning Networks for Reinforcement Learning

Abstract. An extension of system identification approaches for adaptive control by Werbos, Jordan, Munro, Widrow, and Robinson and Fallside is described. The algorithm is based on two interacting fully recurrent continually running networks which may learn in parallel. The algorithm has a potential for on-line learning and locality in time, it does not care for 'epoch-boundaries', it needs only reinforcement information for learning, it allows different kinds of reinforcement (or pain), it allows both internal and external feedback with theoretically arbitrary time lags, and it includes a full environmental model thus providing complete 'credit assignment paths' into the past.

An extension of system identification approaches for adaptive control ((Werbos, 1977), (Jordan, 1988), (Munro, 1987), (Nguyen and Widrow, 1989), (Robinson and Fallside, 1989)) is described.

The algorithm attempts to be a very general one. It attacks the fundamental spatio-temporal credit assignment problem as far as it is attackable at all by pure gradient descent methods (Schmidhuber, 1990b).

The output units of a dynamic recurrent *control network* may influence the state of a reactive non-stationary environment, thus influencing subsequent inputs of the control network. The input of a dynamic fully recurrent *model network* at every time is given by the input and the output of the control network. The model network is trained to predict future activations of the input units of the control network. *Among the control network's input units there are 'reinforcement units' whose desired activations are fixed for all times.* For instance, the desired activations of so-called *'pain-units'* are zero for all times. At a given time the quantity to be minimized by the controller is $\sum_{t,i}(c_i - y_i(t))^2$, where $y_i(t)$ 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 that are still to come.)

Following the approach of system identification, the model network helps to define desired output activations for the control network. Errors for the controller's weights are computed by measuring the partial derivatives of cumulative pain predictions of the model network with respect to controller weights. Hereby the *frozen* model network is taken to be an emulator of the environmental dynamics.

The algorithm can be run in two different modes: There is the sequential version and the parallel version. With the sequential version, first the model network is trained by providing it with randomly chosen examples of sequences of interactions between controller and environment. Then the model 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. The advantage of the parallel version is that the model network focusses only on those parts of the environmental dynamics which the controller typically is confronted with. Particularily with complex environments this represents an enormous potential for gaining efficiency. The disadvantage of the parallel version is that the controller sometimes receives wrong error gradients caused by an inperfect model. This should not be serious, as long as the model continues to improve. However, the controller might enter a local minimum relative to the current state of the model network's weights. This in turn may cause the controller to perform the same silly actions all the time, thus preventing the model network from improving (learning about the effects of alternative actions). Then the whole system might be caught in a state from which it cannot escape any more. The sequential version represents a safer way, but it lacks the flavor of real on-line learning and locality in time.

Below we describe the parallel version. The sequential version can be obtained in a straightforward manner. An on-line version of the Infinite Input Duration (IID) learning algorithm for fully recurrent networks (Robinson and Fallside, 1987) is employed for training both the model network and the control network. (The IID algorithm was first experimentally tested by (Williams and Zipser, 1989).)

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 in the external world are computed. Due to the internal feedback, these actions are based on 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 tries to predict these effects without seeing the new input. 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. Finally, the weights of the control network are updated in order to minimize the cumulative differences between desired and actual activations of the pain and reinforcement units. Since the control network continues activation spreading based on the actual inputs instead of using the predictions of the model network, 'teacher forcing' (Williams and Zipser, 1989) is used in the model network (although there is no teacher besides the environment).

One can find various improvements of the systems described in (Schmidhuber, 1990b) and (Schmidhuber, 1990d). For instance, 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. Furthermore, the model sees the last input and current output of the controller at the same time.

Notation (the reader may find it convenient to compare with (Williams and Zipser, 1989)):

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 reinforcement 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 reinforcement, 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, $p^k_{ij_{new}}$ 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, α_C is the learning rate for the control network, α_M is the learning rate for the model network.

$\mid I \cup P \mid = \mid O \mid$, $\mid O_P \mid = \mid P \mid$. If $k \in I \cup P$, then kpred is the unit from O which predicts k. Each unit from $I \cup P \cup A$ has one forward connection to each unit from $M \cup C$. Each unit from

M is connected to each other unit from M. Each unit from C is connected to each other unit from C. Each weight of a connection leading to a unit in M is said to belong to W_M. Each weight of a connection leading to a unit in C is said to belong to W_C. Each weight $w_{ij} \in W_M$ needs p_{ij}^k-values for all $k \in M$. Each weight $w_{ij} \in W_C$ needs p_{ij}^k-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 $y_{k_{old}}$ *by environmental perception,* $y_{k_{new}} \leftarrow 0.$

FOREVER REPEAT:
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 $k \in C$*:*
begin $y_{k_{old}} \leftarrow y_{k_{new}}$*,*
for all $w_{ij} \in W_C$ *:* $p^k_{ij_{old}} \leftarrow p^k_{ij_{new}}$ *end .*
2. Execute all motoric actions based on activations of units in A*. Update the environment.*
For all $i \in I \cup P$*:*
Set $y_{i_{new}}$ *by environmental perception.*
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 $k \in M$*:*
begin $y_{k_{old}} \leftarrow y_{k_{new}}$*,*
for all $w_{ij} \in W_C \cup W_M$ *:* $p^k_{ij_{old}} \leftarrow p^k_{ij_{new}}$ *end.*
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 $k \in I \cup P$*:*
begin $y_{k_{old}} \leftarrow y_{k_{new}}, y_{kpred_{old}} \leftarrow y_{k_{new}}$*,*
for all $w_{ij} \in W_M : p^{kpred}_{ij_{old}} \leftarrow 0,$
for all $w_{ij} \in W_C : p^k_{ij_{old}} \leftarrow p^{kpred}_{ij_{old}}$ *end.*

To attack the above-mentioned 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' (Williams, 1988) 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 $p^k_{ij_{new}}$ have to be updated according to the following rule:

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

By performing more than one iteration of step 1 and step 3 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).

The parallel version of the algorithm is local in time, but not in space. See (Schmidhuber, 1990b) for a justification of certain deviations from 'pure gradient descent through time', and for a description of how the algorithm can be used for planning action sequences.

Variants of the algorithm are currently tested on certain non-Markovian reinforcement learning tasks. For instance, a controller was able to learn to be a flip-flop similar to the one described in (Williams and Zipser, 1989). Of course, the important difference was that no teacher provided the desired outputs!

Other experiments are currently conducted with a non-Markovian pole balancing task. Unlike with tasks described in (Barto et al., 1983) and (Anderson, 1986), no information about temporal derivatives of the system's state variables (cart position, pole angle with the vertical) is provided. The recurrency of the model network provides a potential for extracting this kind of information, and to represent the state of the environment in a form that allows credit assignment for the controller.

In (Schmidhuber, 1990b) it is described how the algorithm can be employed for *planning* action sequences. It should be noted that the algorithm also could be used as a submodule in an *adaptive critic system* consisting of *three* networks (Schmidhuber, 1990a), where the adaptive critic computes *vector-valued* predictions of future events. This contrasts previous adaptive critics, whose output is just a *scalar* evaluation of the current state.

The parallel version of the algorithm described above has properties which allow to implement something like *the desire to improve the model network's knowledge about the world.* This is related to *curiosity.* In (Schmidhuber, 1991) it is described how the algorithm can be augmented by dynamic *curiosity* and *boredom* in a natural manner. This can be done by introducing (delayed) reinforcement for 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 *self-introspective* behavior.

5 An Example for Learning Dynamic Selective Attention: Adaptive Focus Trajectories for Attentive Vision

Abstract. It is shown how certain cases of selective attention can be learned*: 'Static' neural approaches to certain pattern recognition tasks can be replaced by a more efficient sequential approach. A system is described which* learns *to generate focus trajectories such that the final position of a moving focus corresponds to a target in a visual scene.* No teacher *provides the desired activations of 'eye-muscles' at various times. The only goal information is the* desired final input *corresponding to the target. The task involves a complex temporal credit assignment problem and an attention shifting problem. The system also learns to track moving targets.*

There is little doubt that selective attention is essential for large scale dynamic control systems. In this section we study the problem of *learning* selective attention in the context of attentive vision with dynamic neural networks. The problem, which in its general form has not been explored before, is the control of sequential physical focus-movements. Hereby we concentrate on the question: How can an attentive vision system *learn without a teacher* to generate focus trajectories such that the final visual input always looks like a desireable input corresponding to a target?

A visual scene is given by an object (with internal details) placed on a 512 x 512 pixel field. The object covers only a small part of the scene and may be rotated or translated in an arbitrary manner. Instead of using tenthousands of input units (as in a straight-forward static approach) only about 40 input units are employed. However, these units are sitting on a focus (a two-dimensional artificial retina) which can be moved across the pixel plane. The focus has high resolution in its center and low resolution in its periphery.

In our approach there is a neural *control network* C that controls sequential focus movements. Motoric actions like 'move focus left', 'rotate focus' are based on the activations of the C's output units at a given time. Thus output actions may cause new activations for the input units, and we say that there is *external feedback* (through the environment). The final desired input is an activation pattern corresponding to the target in a static visual scene. The task is to sequentially generate a focus trajectory such that the final input matches the target input. C's error at the end of a sequential recognition process is given by the *difference between the desired final input and the actual final input.* (Control theory calls this a 'terminal control problem'.)

Pure supervised learning techniques for neural networks work only if there is a teacher who provides target *outputs* at every time step of a trajectory (which in our case usually involves about 30 time steps). In our case, however, there never are externally given desired outputs. There only is one final desired *input*.

In order to allow credit assignment to past output actions of C, we employ a supervised learning *model network* M which separately learns to represent a model of the visible environmental dynamics. This is done by training M at a given time to predict C's next input. This prediction is based on previous inputs and outputs of the controller. M serves to 'make the world differentiable'. It serves to bridge the gap between output units and input units of the controller.

A learning algorithm for dynamic recurrent networks is employed to propagate gradient information for C's weights back through M down into C and back through M etc... M's weights remain fixed during this procedure. In different contexts and with different degrees of generality, this basic principle for credit assignment based on *system identification* has been previously described in (Werbos, 1977), (Jordan, 1988), (Munro, 1987), (Robinson and Fallside, 1989), (Nguyen and Widrow, 1989), and (Schmidhuber, 1990b).

Note that in most cases the model network will not be perfect. For instance, if objects in a visual scene may occupy random positions then it will be impossible for M to exactly predict future focus inputs from previous ones. However, it is not intended to make the model a perfect predictor whose output could replace the input from the environment (in that case not much would be gained compared to the static approach: There would be no need for dynamic attention). It suffices if the inner products of the approximated gradients (based on an inaccurate model) for the control network and the true gradients (according to a perfect model) tend to be positive.

M's main task is to help the controller to move the focus into regions of the plane which *allow to continue with more informed moves.* (Although one can not exactly predict what one will see after moving one's eyes to the door, one is setting the stage for additional eye-movements that help to recognize an entering person.)

One goal of this work is to demonstrate that imperfect models can contribute to perfect solutions. Our experiments show *that the system described above is able to learn (without a teacher) correct sequences of focus movements involving translations and rotations*, although M often makes erroneous predictions. At the end of a trajectory, the focus has moved towards a certain target part of the object and is rotated such that the final input corresponds to the desired input (Schmidhuber and Huber, 1990) (Huber, 1990).

Further experiments showed that the system is well-suited for *target tracking*. The desired detail of the moving object soon is focussed and tracked, as long as the objects velocity does not excess the maximal focus velocity.

Further experiments were conducted where C and M learned concurrently. It was found that two interacting conventional *deterministic* networks were *not* appropriate. So each of C's output units was replaced by a little network consisting of two units, one giving the mean and the other one giving the variance for a random number generator which produced random numbers according to a continuous distribution. (We approximated a Gauss distribution by a Bernoulli distribution.) Weight gradients were computed by applying William's concept of 'back-propagation through random number generators' (Williams, 1988).

It was found that such an on-line learning system can be able to learn appropriate focus trajectories. As it was expected, after training M was a good predictor *only* for those situations which the controller typically was confronted with.

6 An Adaptive Subgoal Generator for Planning Action Sequences

Abstract. None of the existing learning algorithms for sequentially working neural networks with internal and/or external feedback addresses the problem of learning 'to divide and conquer'. It is argued that algorithms based on pure gradient descent or on adaptive critic methods are not suitable for large scale dynamic control problems, and that there is a need for algorithms that perform 'compositional learning'. A system is described which solves at least one problem associated with compositional learning. The system learns *to generate sub-goals. This is done with the help of 'time-bridging' adaptive models that predict the effects of the system's sub-programs.*

The algorithms for attacking the fundamental credit assignment problem with dynamic learning algorithms in non-stationary environments can be classified into two major categories.

First, there is the approach of 'back-propagation through time'. This approach has been pursued by (Robinson and Fallside, 1987), (Werbos, 1988), (Pearlmutter, 1989), (Rumelhart et al., 1986), (Williams and Zipser, 1989), (Gherrity, 1989) and others in the case where there is only internal feedback. It has been pursued by (Nguyen and Widrow, 1989), (Robinson and Fallside, 1989), (Werbos, 1977), (Jordan, 1988), and (Schmidhuber, 1990b) in the case where there also is external feedback through a reactive environment.

Second, there is the 'Adaptive Critic' approach, which is of primary interest in the case of external feedback. This approach has been pursued by (Samuel, 1959), (Barto et al., 1983), (Werbos, 1990), and (Schmidhuber, 1990c).

Both the algorithms based on pure gradient descent as well as the 'Adaptive Critic' algorithms have at least one thing in common: They show significant drawbacks when the credit assignment process has to bridge long time gaps between past actions and later consequences.

Both approaches show awkward performance in the case where the learning system already has learned a lot of action sequences in the past. Both approaches tend to modify 'sub-programs', instead of modifying the trigger conditions for sub-programs. They do not have an explicit concept of something like a sub-program. Pure gradient descent methods *always* consider *all* past states for credit assignment. Adaptive critics based on Sutton's 'Temporal Differences' (reinforcement comparison methods) or on Werbos' 'Heuristic Dynamic Programming' consider only the most recent states for 'handing expectations back into time'. Both methods in general tend to consider the wrong states. This is a major reason for their slow performance.

In the next section we will isolate one problem associated with *'compositional learning'*, namely, the problem of learning to generate sub-goals when there already exist a number of working sub-programs (Schmidhuber, 1990f).

6.1 Learning to Generate Sub-Goals

The sub-goal generating system to be described in this section consists of three modules. The heart of the system is a neural network with internal and external feedback, called the control network C. C serves as a program executer. It receives as input a start state, a desired goal state, and time-varying inputs from the environment. The start and goal states serve as 'program names'. We assume that C already has learned to solve a number of tasks. This means that there already are various working programs that actually lead from the start states to the goal states by which the programs are indexed. These programs may have been learned by an algorithm for dynamic networks (as described by the authors mentioned above), *or by a recursive application of the principle outlined below.*

A second important module is a static evaluator network E which receives as input a start state and a goal state, and produces an output that indicates whether there is a program that leads from the start state to or 'close' to the goal state. An output of 1 means that there *is* an appropriate sub-program, an output of 0 means that there is no appropriate sub-program. An output between 0 and 1 means that there is a sub-program that leads from the start state to a state that comes close to the goal, in a certain sense. This measure of closeness has to be given by some evaluative process that may be adaptive or not, and which will not be specified in detail in this paper. (It may be based on TD-methods, for instance.) E represents the system's current model of its own capabilities. We assume that E has learned to correctly predict that each of the already existing sub-programs works. We also assume that E is able to predict the closeness of an end state of a sub-program to a goal state, given a start state. E can be trained in an exploratory phase during which various combinations of start and goal states are given to the program executer.

Finally, the system contains a static network which serves as a sub-goal generator. The sub-goal generator receives as input the external start-input to C, and the desired input (the goal) for C at the end of the task.

The output of the sub-goal generator is a sub-goal, of course. Like the goal, the sub-goal is an activation pattern describing the desired external input at the end of some sub-program, which also is the start input for another sub-program. We concentrate on the most simple case, namely, the case where solutions for given tasks can be found by generating only one sub-goal. The sub-goal generator should output a sub-goal for which there exists a sub-program leading from the start state to the sub-goal, and furthermore a sub-program leading from the sub-goal to the goal state.

How does the sub-goal generator, which initially is a *tabula rasa*, learn to generate appropriate sub-goals? We take two copies of E. The first copy sees the description of a start state and the description of the sub-goal generated by the sub-goal generator. The second copy sees the description of the same sub-goal and the description of the goal. The desired output of each of the copies is 1. Whenever one of the outputs of the copies is below 1, an error gradient is propagated through E's copies *down into the sub-goal generator.* E (as well as its copies, of course) remain unchanged during this procedure. Only the weights of the sub-goal generator change. For a given problem the procedure is iterated until the complete error is zero (corresponding to a solution obtained by combining the two sub-programs), or until a local minimum is reached (no solution found). The gradient descent procedure is used for a search in sub-goal space.

In some experiments with a simple environment a robot was taught to solve certain sequential tasks, like moving from one point to another one. Then more complicated tasks were posed that did not have an associated sub-program.

The sub-goal generator soon learned to generate appropriate sub-goals for the robot.

It should be noted that there also is a different slightly more complex architecture which allows *vector-valued* evaluations of the expected effects of sub-programs.

References

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

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

Gherrity, M. (1989). 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.

Grossberg, S. (1976). Adaptive pattern classification and universal recoding, 1: Parallel development and coding of neural feature detectors. *Biological Cybernetics*, 23:187–202.

Holland, J. H. (1985). Properties of the bucket brigade. In *Proceedings of an International Conference on Genetic Algorithms.* Hillsdale, NJ.

Huber, R. (1990). Selektive visuelle Aufmerksamkeit: Untersuchungen zum Erlernen von Fokustrajektorien durch neuronale Netze. Diploma thesis, Institut für Informatik, Technische Universität München.

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

Kohonen, T. (1988). *Self-Organization and Associative Memory.* Springer, second edition.

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

Munro, P. W. (1987). 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.

Nguyen and Widrow, B. (1989). The truck backer-upper: An example of self learning in neural networks. In *Proceedings of the International Joint Conference on Neural Networks*, pages 357–363. IEEE Press.

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

Pearlmutter, B. A. (1989). Learning state space trajectories in recurrent neural networks. *Neural Computation*, 1(2):263–269.

Robinson, A. J. and Fallside, F. (1987). The utility driven dynamic error propagation network. Technical Report CUED/F-INFENG/TR.1, Cambridge University Engineering Department.

Robinson, T. and Fallside, F. (1989). 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.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation. In *Parallel Distributed Processing*, volume 1, pages 318–362. MIT Press.

Rumelhart, D. E. and Zipser, D. (1986). Feature discovery by competitive learning. In *Parallel Distributed Processing*, pages 151–193. MIT Press.

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

Schmidhuber, J. (1989a). The neural bucket brigade. In Pfeifer, R., Schreter, Z., Fogelman, Z., and Steels, L., editors, *Connectionism in Perspective*, pages 439–446. Amsterdam: Elsevier, North-Holland.

Schmidhuber, J. (1989b). The Neural Bucket Brigade: A local learning algorithm for dynamic feedforward and recurrent networks. *Connection Science*, 1(4):403–412.

Schmidhuber, J. (1990a). Additional remarks on G. Lukes' review of Schmidhuber's paper 'Recurrent networks adjusted by adaptive critics'. *Neural Network Reviews*, 4(1):43.

Schmidhuber, J. (1990b). 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.

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

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

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

Schmidhuber, J. (1990f). Towards compositional learning with dynamic neural networks. Technical Report FKI-129-90, Institut für Informatik, Technische Universität München.

Schmidhuber, J. (1991). A possibility for implementing curiosity and boredom in model-building neural controllers. In Meyer, J. A. and Wilson, S. W., editors, *Proc. of the International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pages 222–227. MIT Press/Bradford Books.

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

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

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

Werbos, P. J. (1977). Advanced forecasting methods for global crisis warning and models of intelligence. In *General Systems*, volume XXII, pages 25–38.

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

Werbos, P. J. (1990). Consistency of HDP applied to a simple reinforcement learning problem. *Neural Networks*, 2:179–189.

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

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