ON PLANNING AND EXPLORATION IN NON-DISCRETE ENVIRONMENTS

Sebastian B. Thrun
German National Research Center for Computer Science (GMD)
D-5205 St. Augustin, FRG

Knut Möller
University of Bonn
Department of Computer Science
D-5300 Bonn, FRG

Abstract

The application of reinforcement learning to control problems has received considerable attention in the last few years [And86, Bar89, Sut84]. In general there are two principles to solve reinforcement learning problems: direct and indirect techniques, both having their advantages and disadvantages.

We present a system that combines both methods [TML91, TML90]. By interaction with an unknown environment a world model is progressively constructed using the backpropagation algorithm. For optimizing actions with respect to future reinforcement planning is applied in two steps: An experience network proposes a plan which is subsequently optimized by gradient descent with a chain of model networks. While operating in a goal-oriented manner due to the planning process the experience network is trained. Its accumulating experience is fed back into the planning process in form of initial plans, such that planning can be gradually reduced. In order to ensure complete system identification, a competence network is trained to predict the accuracy of the model. This network enables purposeful exploration of the world.

The appropriateness of this approach to reinforcement learning is demonstrated by three different control experiments, namely a target tracking, a robotics and a pole balancing task.

Keywords: backpropagation, connectionist networks, control, exploration, planning, pole balancing, reinforcement learning, robotics

Contents

1 Introduction

Reinforcement learning methods have recently found a growing application in intelligent control problems [And86, Bar89, SW90, Sut84]. The basic idea of reinforcement learning originates from animal learning theory and psychology [Ski38]. It is based on rather general evaluative feedback (called reinforcement or critic) which is provided by a teacher or an environment. A reinforcement signal can criticize the action a learning system has taken, providing "reward" for good actions and "penalty" for bad ones.

The application of reinforcement learning problems to control theory is straightforward and has found widespread application. Whenever the objective is to find actions which transfer an unknown system (world) into a desired goal state, and the effect of actions are only evaluated by a single scalar, reinforcement learning is appropriate. The task of finding actions becomes more complicated if the reinforcement does not evaluate each action separately. Often the reinforcement provides an evaluation of all actions in the past: the better those actions, the larger the reinforcement. In order to judge each particular action of a sequence, the problem arises how to assign particular reinforcements to particular actions in the past.

In the context of game playing Samuel [Sam59] formulated the so-called "temporal credit assignment problem" [Sut84], i.e. how should a move in the middle of a game be judged, if the only evaluation of performance ("win" or "loss") comes out at the end? Or stated more abstract: how to assign credit to individual actions, when only a final outcome is evaluated?

The mathematical foundations of reinforcement learning have been investigated by Narandra/Lakshmivarahan [NT89, Lak81], in control theory by Fu/Waltz [FW63] and in the context of connectionist learning especially by Barto et al. [BAS82], Williams [Wil87] and various others. They demonstrate how and how effective learning can be performed using reinforcement information.

In general, there are two principles for solving adaptive control problems, namely *direct* and *indirect*. We will shortly discuss their properties, before proceeding to our planning procedure which combines both principles.

Direct reinforcement learning systems try to identify a controller which maps states and goal states directly to a corresponding action or a sequence of actions, respectively. In order to find such a controller the world has to be explored, sequences of actions have to be performed and its results be used for progressively improving this controller. There is an important benefit with this approach, namely its speed in computing actions. Once a controller is found the recall of actions is easy and constant in time. On the other hand there are three important drawbacks:

- The set of optimal actions in a given situation is usually not unique. E.g., if a controller has to do two things A and B in turn, it can do A first and then B or vice versa. This many-to-many character is often reduced to a many-to-one function unless controllers behave probabilisticly. Most direct reinforcement approaches deal with deterministic functions.

  This restriction gains in importance if flexibility at runtime is required. Incorporation of additional optimization constraints without expensive retraining is not possible, since those controllers generate only one action regardless of all other possibilities. E.g., changing the objective function implies the need of retraining.

- Controllers which optimize actions with respect to the whole future must know about future effects. Since future is not considered explicitly, future effects must be directly encoded in the controller. This complicates training significantly.
- Most connectionist approaches to the temporal credit assignment problem rely on the interaction of a model and a controller. The model is trained to predict the world and is independently used for training the controller, e.g. by gradient descent [And86, Bar89, Jor89, JR90, Mun87, NW89, Rob89, RF89, Wer87]. Although there is now agreement about this [JR90, Chr90], we believe that the accuracy of the controller lags behind that of the model network, since the former is trained by gradients propagated through the latter. If the model is inaccurate the gradients cannot be expected to bring out more accurate information anyway. We will demonstrate this later.

In contrast, **indirect** methods are based on learning explicit models of the world to be controlled. Since no future dependencies have to be resolved this is a considerably simple and thus fast task[1]. It is not possible to generate actions in one shot using a pure model only – indeed this is the main advantage of direct learning methods. Instead, actions have to be searched in action space using the model for evaluation.

On the other hand certain properties make them look attractive:

- Model learning is simpler and thus fast, as mentioned before. This causes an improved performance of a system at early stages.
- The many-to-many character of the control task is usually not affected, even if many-to-one functions are used for models. This is because several actions might be mapped to the same optimal reinforcement. Depending on the search heuristic more than one might be found.
- The last point can be extended to a second argument. In general changing of the objective function at runtime requires no retraining. This is because the optimization is done at each step, contrary to direct methods at which optimization is part of the training. Thus, control is more flexible.

If the temporal credit assignment has to be solved planning becomes necessary for solving reinforcement learning problems indirectly. Planning has a long tradition in AI – the appropriateness of planning has been demonstrated whenever future effects must be taken into account, c.f. [Nil82], and in the context of connectionism and robotics [KMUS90, Mel89]. Planning has also been used for reinforcement learning. E.g., Sutton [Sut90] uses off-line planning for improving the controller without interacting with the world.

In this article we will present a planning technique which relys on a combination of direct and indirect learning control [TML91, TML90]. A model network which approximates the behavior of the world is used for looking ahead into future and optimizing actions by gradient descent with respect to future reinforcement. In addition, an experience network is trained like a controller but used for accelerating

[1] Even this is a science itself [Lju87].

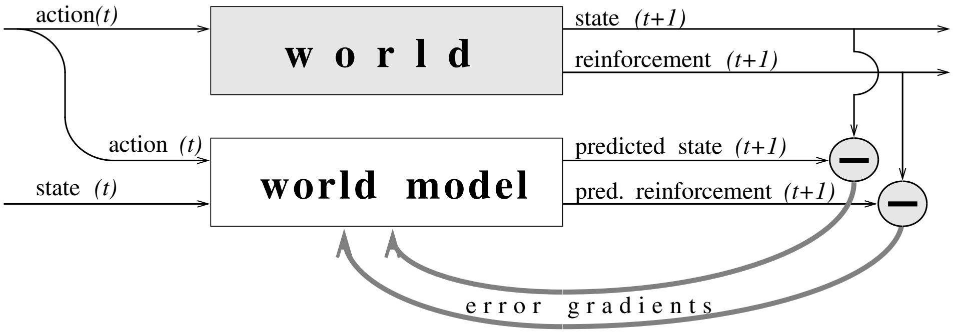


Figure 1: The training of the model network is a system identification task. Internal parameters are estimated by gradient descent, e.g. by backpropagation. The accuracy of the model is progressively improved.

planning. To ensure exploration the incorporation of a competence network is discussed which estimates the accuracy of the model. This information is used by a combined optimization: On the one hand reinforcement is optimized, but on the other hand poorer known regions in state space are explored purposefully.

2 System Identification

The world considered in this paper is defined as a mapping from states and actions to subsequent states and reinforcements. Each time an action is performed on the world, the world transforms its state due to this action to a new state and a reinforcement – the latter one is to be optimized.

Planning is a hypothetical process that optimizes actions with respect to future reinforcement. For considering future we need a world model. If we do not know the world in advance, we have to train a world model by examples to mimic the behavior of the world. Training of the world model is adopted from [And86, Bar89, Jor89, JR90, Mun87, Rob89, Wer87].

The world model used here is a standard backpropagation network [RHW86, DR89, Jor89] (Fig. 2 a), although each piecewise differentiable mapping would do also. At the beginning the world mapping is unknown. Thus, the internal parameters of the model– namely weights and biases[2] – are initialized with small random values. Then each observation of the world is used for adapting these parameters by means of gradient descent: each time an action is performed on the world we feed the actual state and that action into the world model in order to predict subsequent state and reinforcement. Now the real and the predicted state and reinforcement are compared. A gradient is computed from the difference and this is propagated back through the model network for adapting its internal parameters in order to improve its accuracy (Fig. 1). The resulting model approximates the behavior of the world.

[2]For an introduction to backpropagation and other basic connectionist learning rules see [RHW86].

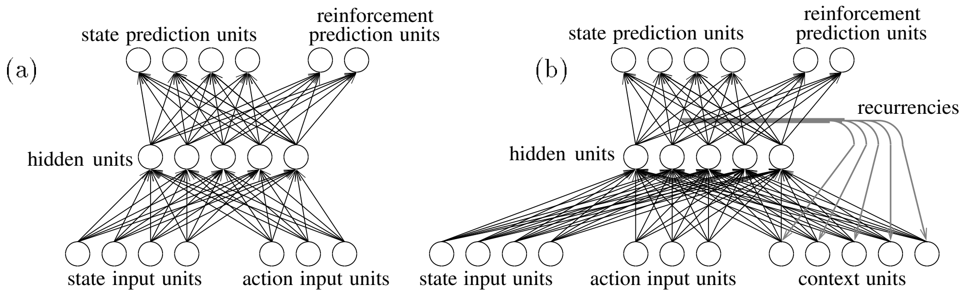


Figure 2: The model network is either a feed-forward network (a) or, more general, a recurrent one (b) (Elman-type).

Up to now we have assumed that the whole state of the world is observable. This is not the case in general. E.g. if some camera observes a three-dimensional scene, its output is just a two-dimensional projection of that scene. There is a principal problem by that. There might be some quantities of the internal state of the world currently not observable, although influencing the mapping of the world crucially. Then knowing just the current visible state and nothing about previous ones would make the world seem to behave randomly. In this case we will assume that these quantities have been visible some time before, otherwise the mapping of the world is not a function any more. Thus, the world model has to be able to "store" context information through time. By using a recurrent network [Jor86, Elm88, Ghe89, Moz88, Pin87, Pea88, Pea90, RF87, WZ88, TS90] instead of a feed-forward network as a model this problem can be overcome. We will focus on recurrent networks of the Elman-type [Elm88] (Fig. 2 b) which seem to be best suited for our purpose.

If we speak of states in turn, we mean externally observable states rather than internal ones. At a well-trained network both state and context activations together are sufficient to describe all crucial quantities of the internal state of the world. Note that this does not hold at the beginning of training, since recurrent networks need training for finding appropriate representations of hidden quantities.

It should be noted here that backpropagation might fail in system identification due to local minima of gradient descent parameter estimation [RHW86, KP90]. This is an inherent problem of gradient descent techniques. There is also no general theory concerning convergence time.

3 Look-Ahead Planning

For training the model network actions must be performed on the world, which must be generated somehow. Using knowledge in the world model to optimize those actions with respect to future reinforcement is the aim of the planning procedure described in turn. Exploration is discussed in more detail in section 8.

Planning by gradient descent is an optimization procedure which consists mainly of two parts.

- Firstly, we have the problem of finding an initial plan which must be generated

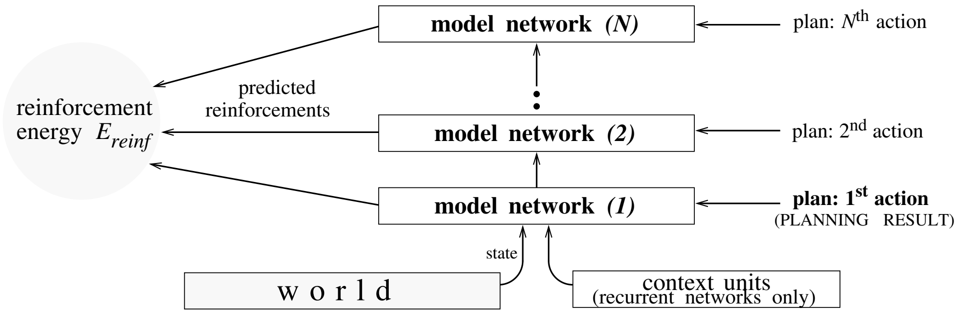


Figure 3: Looking ahead by the chain of model networks. The arrows indicate both activation and gradient propagation (see text).

in a reasonable amount of time and must be close enough to a good one.

- Secondly, the plan is optimized progressively in small steps by gradient descent due to future reinforcement. Therefore the plan is evaluated, i.e. future reinforcement is predicted by a chain of model networks. Then we define a differentiable reinforcement energy function which measures the deviation of some predefined optimal reinforcement and the predicted one. The plan is now updated with gradients of this energy with respect to the plan [KL90, LK89, Sch90, TL90]. The procedure is repeated several times which leads to a progressively optimization of reinforcement.

First we consider the second step.

3.1 Plan Optimization by Gradient Descent

Let us assume we have already an initial plan, i.e. a sequence of N actions. If, as it is discussed later, we use some heuristic for deriving initial plans, the quality of the plans in terms of expected reinforcement is usually poor. The optimization of this plan with respect to future reinforcement is to be investigated.

Step 1: Evaluation. Before improving the plan it has to be evaluated by using the world model. Let us consider the first action of the plan. The world model is trained to estimate a subsequent state and reinforcement from this first action, using the world's current state and the context information of the world model. By feeding these three quantities into the model (Fig. 3), the corresponding next state and reinforcement are predicted as well as a new subsequent context vector. If we assume that these predictions are good estimations of the real world's behavior, we can proceed by using the second action of the plan and predicting its subsequent quantities by a second copy of the world model. The procedure is repeated for each of the N stages of the plan, using a chain of N world model copies. Finally, we have a prediction of the corresponding N reinforcement vectors.

Step 2: Improvement. As mentioned before, the deviation of this reinforcement from the predefined optimal reinforcement is measured by some differentiable reinforcement energy function E_{reinf}. The value of this function is the evaluation of the plan (with respect to the knowledge in the model network): The smaller E_{reinf}, the better the plan. Thus, the problem of optimizing plans is reduced to a minimization problem. Since both E_{reinf} and the model network (and thus the chain) are differentiable, the gradients of this energy with respect to the plan can be computed. These gradients indicate how to change the plan in order to minimize E_{reinf} and are used for changing the plan in small steps.

To sum up, this is one step of the gradient descent approximation which is repeated several times. Although there might be a lot of different termination conditions, we will mention here only two: The whole update procedure is repeated either until convergence is observed or, which makes it more convenient for real-time applications, a predefined number of iterations – note that in the latter case the computational effort is linear in N. From the planning procedure we obtain the optimized plan, the first action of which is then performed on the world. After planning and performing the resulting action on the world the whole procedure is repeated.

The computation of these gradients is the contents of the next section. They can either be computed simply by backpropagation through time, as described in [RHW86] or computed at the same time as the model chain is constructed. The latter way leads to an algorithm which is related to that in [WZ88] which enables us to keep the plan lengths variable.

3.2 Mathematical Derivation of the Gradient Descent Algorithm

Next we derive the mathematics for feed-forward gradient descent in plan space. If the reader is just interested in the underlying ideas rather than the mathematical details, we recommend to skip this section and to believe that gradients can be computed in a feed-forward manner.

We will start with a couple of definitions.

- We distinguish between the following sets of units in the model network: action input units, state input units, context input units, state prediction units, reinforcement prediction units, context output units[3] and hidden units (c.f. Fig 2). In order to avoid index trouble we use a prime to mark the corresponding output unit of an input. E.g., in the chain of model networks output units of model τ–1 are connected one-to-one to the input units of model τ, thus input unit i at τ receives the output from unit i' at τ–1. We also use the prime to associate the corresponding vector component to an input unit.
- The plan is a sequence of N actions: $\vec{a}(1), \vec{a}(2), ..., \vec{a}(N)$. Each of this vectors has as many real-valued components as the model network action input units.
- By $\vec{s}$ we denote the current state vector of the world and by $\vec{c}$ the current context units' activation vector, which are to represent the current non-observable internal state of the world.
- Let $x_i(\tau)$ be the activation of node i in the τth copy of the model chain ($1 \leq \tau \leq N$). The weight from unit j to unit i is denoted by w_{ij} and the bias of unit i by θ_i; net_i denotes the so-called netinput of unit i.

[3] By context output units we mean those units whose activations are shifted into the context units (with the temporal delay 1)

- The activation function for propagating activations forward through a network is defined as

$$x_i(\tau) \;=\; \begin{cases} \text{input}_i(\tau) & \text{if } i \text{ input unit (and input}_i \text{ its} \\ & \text{corresponding net-external input)} \\ \sigma(\text{net}_i(\tau)) & \text{if } i \text{ hidden or output unit (otherwise)} \end{cases} \tag{1}$$

$$\text{with } \text{net}_i(\tau) \;=\; \sum_{l \in P(i)} w_{il} x_l(\tau) + \theta_i \tag{2}$$

Here $P(i)$ labels the set of all predecessors of unit i, i.e. all units which are connected with unit i by a link, and $\sigma(\cdot)$ is the logistic function, usually $\sigma(z) = \frac{1}{1+e^{-z}}$. See [RHW86] for further details.

Unlike in the first step (τ=1) at which the state $\vec{s}$ of the world and the context activations $\vec{c}$ are fed into the model, in all other steps the state and context predictions are used as the corresponding input. This leads to the following activation function (c.f. (1), (2) and Fig. 3):

$$x_i(\tau) \;=\; \begin{cases} a_{i'}(\tau) & \text{if } i \text{ action input unit} & \\ s_{i'} & \text{if } i \text{ state input unit} & \wedge\ \tau{=}1 \\ c_{i'} & \text{if } i \text{ context unit} & \wedge\ \tau{=}1 \\ x_{i'}(\tau{-}1) & \text{if } i \text{ state input or context unit} & \wedge\ \tau{>}1 \\ \sigma(\text{net}_i(\tau)) & \text{otherwise} & \end{cases} \tag{3}$$

For deriving the desired gradients we define $\forall 1 \le s \le \tau \le N$, $\forall i, j$ (i numbers the components of the action vector $\vec{a}$ and j all units) the auxiliary gradients

$$\xi^j_{is}(\tau) \;\equiv\; \frac{\partial x_j}{\partial a_i(s)}(\tau) \tag{4}$$

which measure how an infinitesimal small change of the ith component of the sth action of the plan influences the activation of node j at time τ. It will be shown next that the $\xi^j_{is}(\tau)$ are easily propagated forward through time together with the activations $x_j(\tau)$.

It is easy to see that $\xi^j_{is}(\tau)$=0 holds $\forall s{>}\tau$, since no action influences activations in the past. The same holds $\forall s{=}\tau$ if j is a state input or a context unit, or if j is an action input unit with $j{\neq}i'$ (i.e. the i-th component of $\vec{a}(s)$ is not fed into action input unit j). Conversely, if $s{=}\tau$ and $j{=}i'$ then $\xi^j_{is}(\tau)$=1. The propagation rule for all remaining cases (j is hidden or output unit) reduces to:

$$\begin{aligned} \xi^j_{is}(\tau) \;&\overset{(3),(2)}{=}\; \frac{\partial\ \sigma\left(\displaystyle\sum_{l \in P(i)} w_{jl} x_l(\tau) + \theta_j\right)}{\partial\ a_i(s)} \\ &= \sigma'(net_j(\tau)) \cdot \sum_{l \in P(i)} \frac{\partial\ w_{jl} x_l(\tau) + \theta_j}{\partial\ a_i(s)} \\ &= \sigma'(net_j(\tau)) \cdot \sum_{l \in P(i)} w_{jl} \xi^l_{is}(\tau) \end{aligned} \tag{5}$$

This leads to the propagation rule for the auxiliary gradients:

$$\xi_{is}^{j}(\tau) = \begin{cases} \delta_{i'j}\delta_{s\tau} & \text{if } j \text{ action input unit} \\ 0 & \text{if } j \text{ state input or context unit} \quad \wedge \ \tau{=}1 \\ \xi_{is}^{j'}(\tau{-}1) & \text{if } j \text{ state input or context unit} \quad \wedge \ \tau{>}1 \\ \sigma'(\mathrm{net}_j(\tau)) \cdot \sum\limits_{l \in P(j)} w_{jl}\, \xi_{is}^{l}(\tau) & \text{otherwise} \end{cases} \tag{6}$$

(Here δ denotes the Kronecker symbol: $\delta_{i'j}{=}1$ if $i'{=}j$ and 0 otherwise.)

Having now computed these gradients we have to show what they are useful for. In order to optimize reinforcement we define a reinforcement energy E_{reinf}. This function is to be minimized, since it measures the deviation of the current reinforcement predictions and its optimum.

$$E_{\mathrm{reinf}} \equiv \frac{1}{2} \sum_{\tau=1}^{N} \sum_{k} g_k(\tau) \cdot (r_k^*(\tau) - x_k(\tau))^2 . \tag{7}$$

Here k numbers all reinforcement output units, $r_k^*(\tau)$ is the corresponding predefined optimal reinforcement, i.e. the desired reinforcement. The function $g_k(\tau)$ weights the reinforcement, in the simplest case $g_k(\tau) \equiv 1$. Note that this weighting function allows to emphasize particular reinforcements and particular time steps – see the examples below.

The derivation of this energy function with respect to the predicted reinforcement output k at time τ is

$$\frac{\partial E_{\mathrm{reinf}}}{\partial x_k(\tau)} = -g_k(\tau) \cdot (r_k^*(\tau) - x_k(\tau)) \tag{8}$$

With this gradient we can compute ($\forall 1{\le}s{\le}N$, $\forall i$ action vector component) the gradients

$$\begin{aligned} \zeta_{is} &\equiv \frac{\partial\ E_{\mathrm{reinf}}}{\partial\ a_i(s)} \\ &\stackrel{(8)}{=} -\sum_{\tau=s}^{N} \sum_{k} g_k(\tau) \cdot (r_k^*(\tau) - x_k(\tau)) \cdot \frac{\partial x_k}{\partial a_i(s)}(\tau) \\ &= -\sum_{\tau=s}^{N} \sum_{k} g_k(\tau) \cdot (r_k^*(\tau) - x_k(\tau)) \cdot \xi_{is}^{k}(\tau) \end{aligned} \tag{9}$$

These gradients ζ_{is} indicate how to change the plan in order to optimize E_{reinf}. Hence the plan update rule is:

$$\begin{aligned} a_i(s) &\longleftarrow a_i(s) - \eta \cdot \zeta_{is} \\ &= \eta \cdot \sum_{\tau=s}^{N} \sum_{k} g_k(\tau) \cdot (r_k^*(\tau) - x_k(\tau)) \cdot \xi_{is}^{k}(\tau) \end{aligned} \tag{10}$$

for a sufficiently small stepsize $\eta > 0$.

By using this update rule (10) it cannot be ensured that actions are bounded. Therefore actions are subsequently cut, if they leave their domain:

$$a_i(s) \quad \longleftarrow \quad \begin{cases} 0 & \text{if } a_i(s){<}0 \\ a_i(s) & \text{if } 0{\leq}a_i(s){\leq}1 \\ 1 & \text{if } a_i(s){>}1 \end{cases} \tag{11}$$

This completes the derivation of the gradient descent update rule.

3.3 Variable Plan Lengths

An important issue arises from the feed-forward manner gradient computation. Both activations x_i and gradients ξ_{is}^j are propagated together forward in time (c.f. (3),(6)) which allows to keep plan lengths variable. The underlying idea is that planning makes little sense if the model is too inaccurate for producing reasonable predictions. This is the case if the current linear error of the model, measured by the training patterns, is larger than the effect of the first action of the plan – the forthcoming planning result – to the current state and reinforcement predictions. The latter effect is estimated by the gradients ξ_{is}^j. The heuristic to cut plans due to the current accuracy of the model might avoid senseless time-consuming look-ahead. The more accurate the model networks the larger the plan lengths N. Although in general it cannot be assured that the gradients falls beyond that error bound, it happened in all observed cases.

4 A Basic Example

It is time now to illustrate the planning procedure by an example. Consider the following situation: An autonomous agent has to track a target in a two-dimensional environment [TML90]. The position of the agent in x-y-space as well as that of the target represent the visible state of the world, and the actions of the agent are movements with a fixed, pre-given speed, i.e. the action vector $(\Delta x, \Delta y)$ is normalized before execution. Rather than fleeing in a fixed direction, the target has a certain strategy to escape which is not known in advance: Each time the agent moves, the target moves perpendicular to that movement. This is achieved by moving the target by $(\Delta y, -\Delta x)$ (a bit slower for avoiding frustrations of the agent). Note, that both movement equations are not known to the agent in advance.

The goal was to minimize the distance between agent and target. Thus, we used a continuous reinforcement (potential field) with two reinforcement units, one for the x-distance and one for the y-distance of agent and target. Minimizing this reinforcement is equivalent to tracking the target. The basic definition of the states and the reinforcement in particular made model learning very easy. All the model had to learn was adding vectors. Therefore we used a one-layered network with six input units (four for state, two for action) and six output units (again four for state, and two for reinforcement).

Before we illustrate the appropriateness of planning, we will face a basic solution without planning. Obviously, optimizing the immediate next reinforcement at each step would make the agent run directly into the target's direction. This would lead to trajectories as shown in Fig. 4.

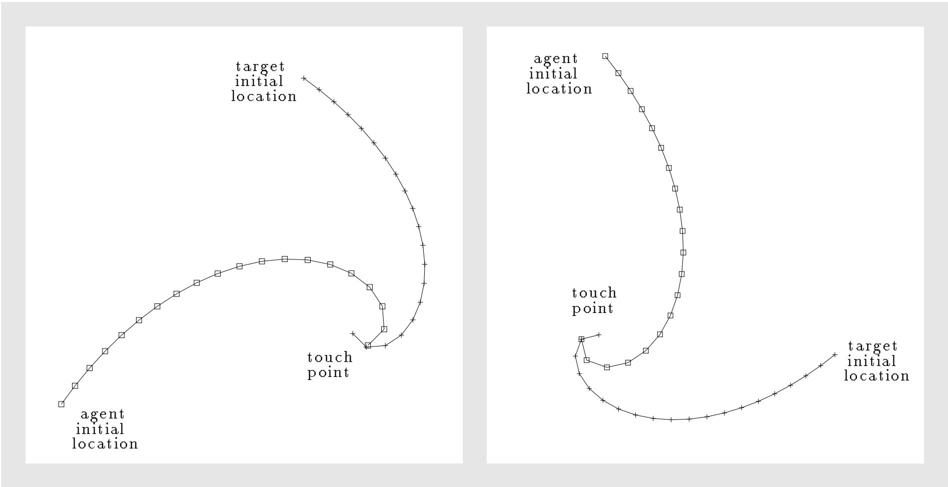


Figure 4: Target tracking – basic strategy: The agent (boxes) moves always into the target's (crosses) direction. This leads to non-optimal trajectories.

It is easy to see that in order to reach the target as quick as possible it is more clever to use short cuts instead. Then neither the agent nor the target (as a consequence) change it's direction. This analytical best solution looks like (Fig. 5):

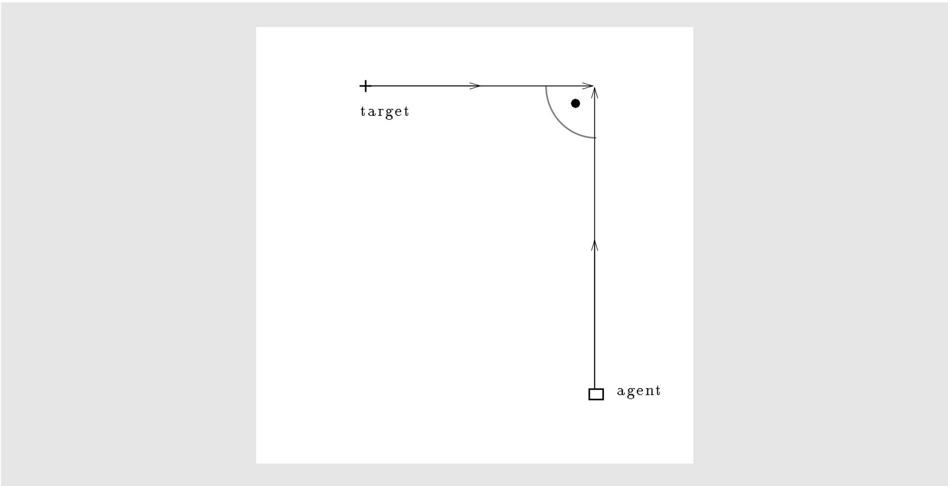


Figure 5: Target tracking – an analytical optimum

The aim of short cuts is to prefer optimization of later reinforcement to sooner. This

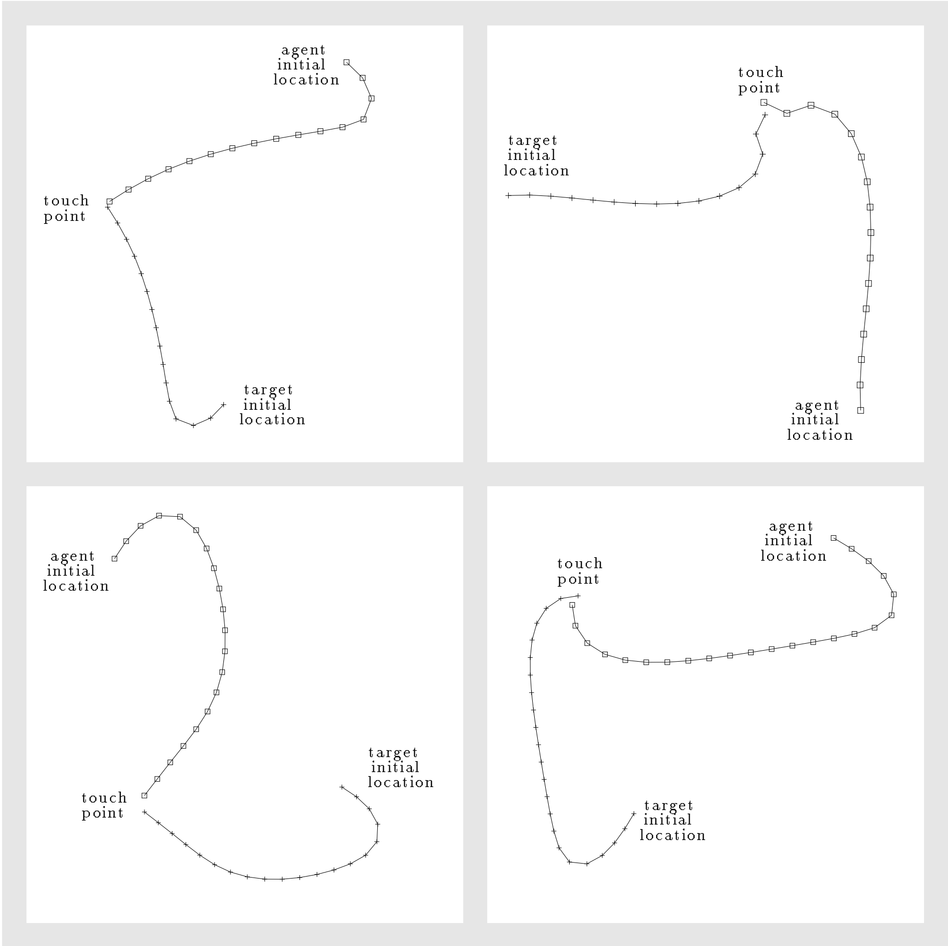


Figure 6: Target tracking – trajectories after 800 learning steps.

can be expressed by the reinforcement weighting function g (c.f. (7)) which was set to an exponentially increasing function $g_k(\tau)=2^{\tau}$. The further the look-ahead, the larger the weight of a reinforcement deviation. A second detail to mention is that the stepsize also depended on τ: $\eta=\eta(\tau)=2^{-\tau-1}$. This was because the τth action of the plan is optimized only due to the next $N-\tau+1$ reinforcements, i.e. the larger τ, the smaller the look-ahead from that point.

Note that the whole mapping of the world was not known in advance. The agent had to learn the model and hand in hand to find short cuts. We applied planning with a look-ahead $N=7$ and 10 iteration steps. Due to an inaccurate model at this early stage of training, observed trajectories are shown in Fig. 6. After about 3500 learning steps the model was sufficiently optimized (Fig. 7).

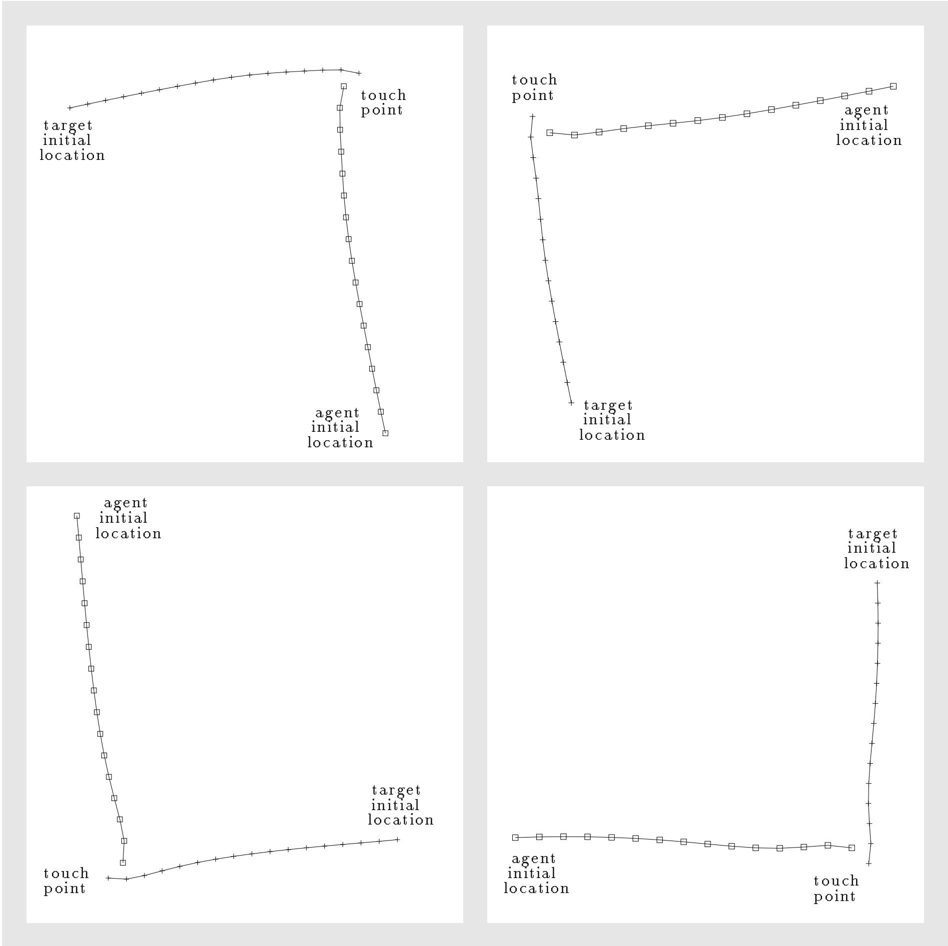


Figure 7: Target tracking – final trajectories (3500 learning steps).

5 Initial Plans – The Experience Network

Up to now we discussed how to optimize plans. This requires that some initial plan is already available. Now we will come back to the question how to obtain those initial plans.

After a planning phase, only the first action is executed by the environment. The rest of this plan is a good candidate for an initial plan at the next planning phase. This reduces the problem of finding initial plans to finding single new actions, namely the Nth action of an initial plan.

Basically, there are some simple heuristics for finding initial actions/plans:

- **Zero-actions**. Use that action as initial action which does not change the

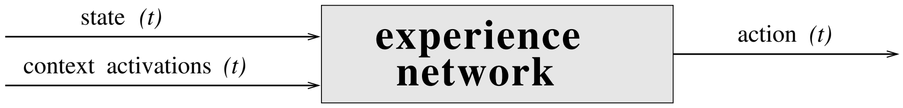


Figure 8: The *experience network* is trained to predict the result of the planning procedure from the current state and, in case of a recurrent model network, its context activations.

(internal) state of the world if it is executed. The problem is that zero actions must exist and be known a priori. If so, using zero-actions is an uninformed way of choosing initial actions. In the above example zero-actions are not defined since each action is a movement of fixed speed.

- **Random actions**. Use a random vector as initial action. This is what was taken in the example. Starting with random plans is obviously the second uninformed way of plan initialization which might cause trouble with local minima.
- **Average**. Use the average over previous planning results. If some planning took place before – otherwise the average is not defined – this might be an advantageous way depending on the problem. In the above example averaging over previous trajectories would not make sense.

All these strategies proposed above are more or less problem-dependent. Therefore the question arises whether there is a problem-independent way of generating initial actions. The answer is given by the experience network.

A good way of finding initial plans is to learn from previous planning results. This can be done with an *experience network* which is trained to predict the result of the planning procedure. Its input is the same as the input of the planner: the state of the world and, in case of recurrencies in the model network, its context units' activations. The target values of the experience network are the results of the planning procedure, such that this network is trained to mimic the whole planning procedure in one step – this is exactly what planning is used for in [Sut90]. Since the experience network is trained with the world model, the former lags behind the latter. Therefore we consider its outgoing actions for proposals rather than optimal actions, which are subsequently optimized.

In general planning may find different solutions in the same situations, depending on the initial values of the search. This is because the inverse model, i.e. the controller function which maps situations and goal descriptions to optimal actions, is not unique. It would be most general therefore to use non-deterministic associative networks for the experience task. Conversely, in the experiments described in the sequel there is only one global optimal action to each particular situation. Thus, we used again backpropagation for experience networks. Note that this restricts the planner to find only one action which could be avoided by adding small random variables during plan optimization.

There are three nice consequences by using experience networks.

- The more accurate the experience network, the more planning can be shortened.

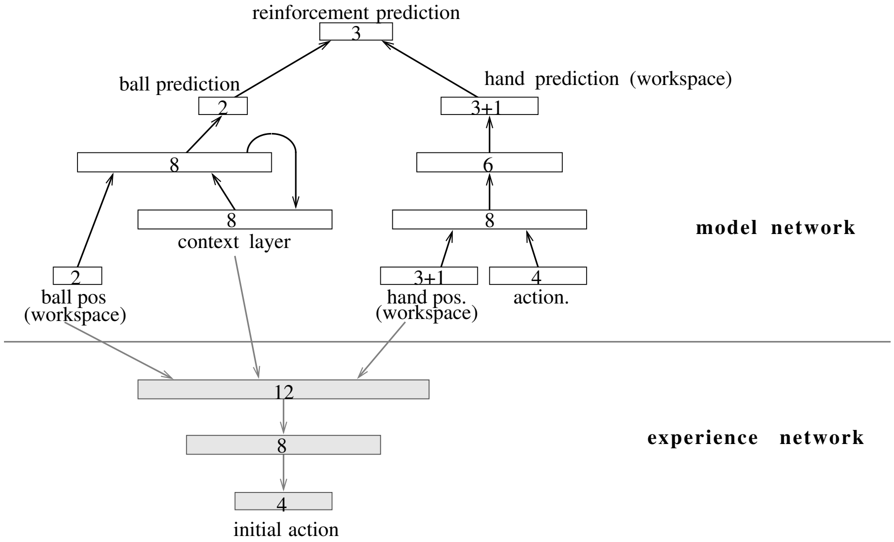


Figure 9: The recurrent model network (white) and the experience network (grey) at the robotics task.

If the proposed actions get closer to the corresponding planning result, gradient descent can be iterated less. Hence planning can be reduced progressively due to the "experience" in the experience network.

- The experience network is trained with optimized training patterns [Sut90]. No temporal dependencies have to be resolved as it is often done by direct controllers [And89] which would complicate training considerably.

- Optimization always happens with respect to the knowledge in the model network which performs better than the experience network.

These points make clear how advantages of both indirect and direct methods are combined.

6 Investigation of Planning Abilities

Now we will demonstrate properties of planning on a more difficult and realistic example. The experiment is a robot arm trying to approach a rolling ball. Both model and experience network are more complicated, especially since recurrencies are required.

With this example a) the knowledge in the experience and the model are compared in order to justify planning and b) the incorporation of additional constraints at run-time is demonstrated.

6.1 Approaching a Rolling Ball with a Robot Arm

The task described in this section is somehow similar to the experiment described previously: A 5-DOF (degrees of freedom) robot arm in three-dimensional space was to approach a rolling ball in real-time. A camera system observed the position in workspace of both robot arm manipulator (x_{robot}, y_{robot}, z_{robot} and hand angle h_{robot}) and ball (x_{ball} and y_{ball} on a plain surface). As mentioned before, these coordinates were observed in workspace. Conversely, actions were angular changes of the robot arm joints, i.e. they were defined in configuration space.[4]

The model and experience network are shown in Fig. 9. There was a recurrent part in the ball prediction, since it is impossible to predict the ball's next position without a temporal context. Reinforcement was separately measured for each one of the three coordinates. Once more the distance was defined as the (continuous) reinforcement to be minimized.

As described above planning makes sense because we wanted the robot arm to find short cuts rather than just "following" the ball. By using the basic strategy – namely to optimize reinforcement stepwise – trajectories as shown in Fig. 10 could be expected:

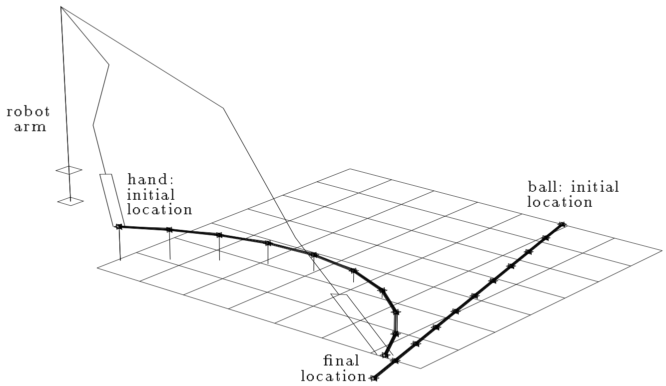
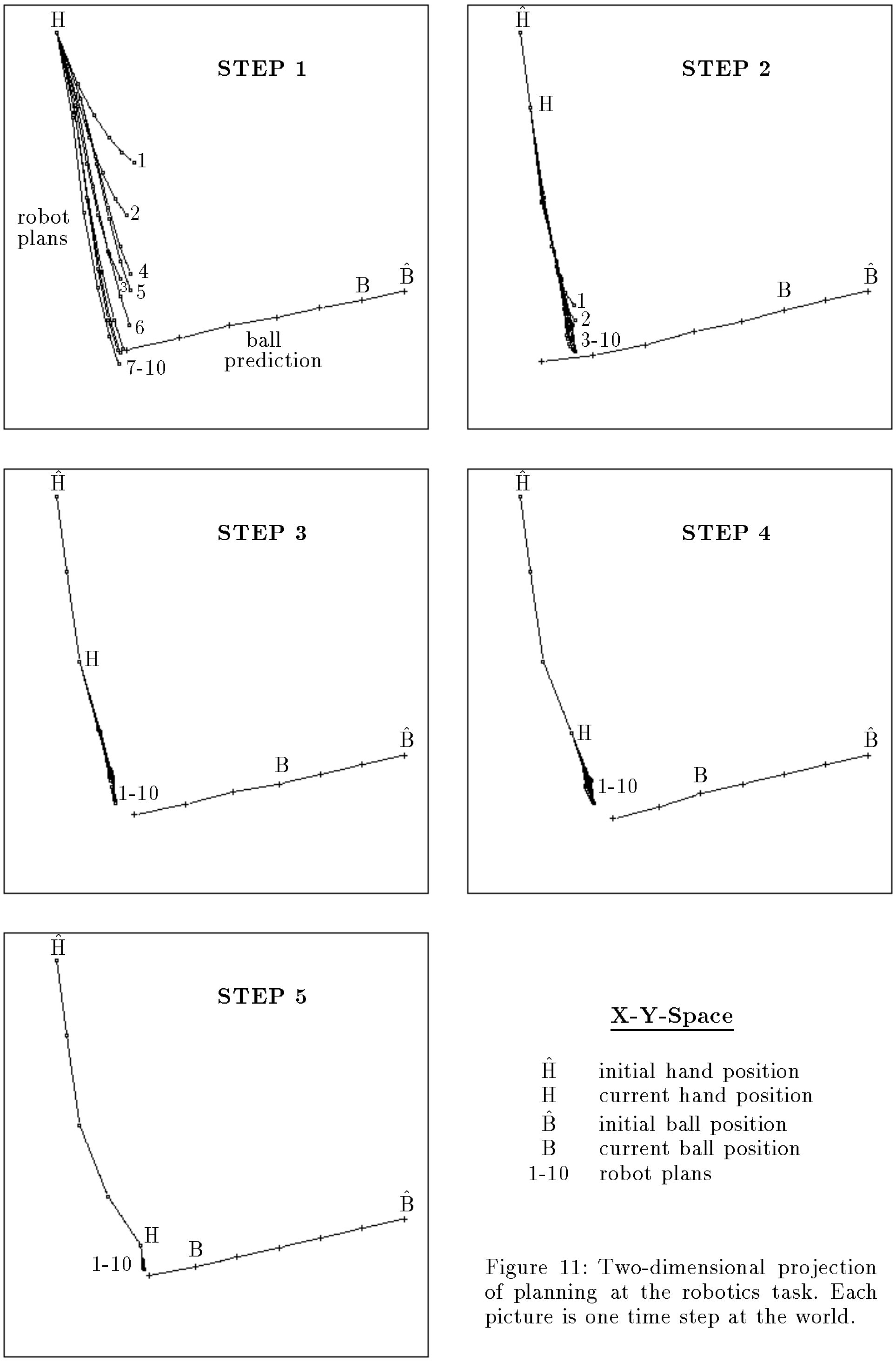


Figure 10: Basic strategy – the arm "follows" the ball.

Instead, we wanted the arm to find short cuts by making predictions about the ball's future movements. We will now demonstrate how planning works in that example. With a look-ahead $N \leq 5$, 9 planning (gradient descent) iterations, an exponentially reinforcement weighting function $g_k(\tau)=1.3^\tau$, a weighted stepsize $\eta=0.05 \cdot 0.9^\tau$, and a sufficiently trained model and experience network (20 000 training steps) the planning process is illustrated in Fig. 11. Note, that these are only projections in x-y-space. In the left upper picture the ball was already observed the second time (after the first observation planning is useless since the ball movement cannot be predicted). The experience network proposes the plan marked by 1 which is then optimized in the next 9 planning iterations. It is clear to see that the resulting plan

[4]To point out the difficulties with workspace and configuration space, it should be noted that for an analytical model of the arm's state transition function first inverse kinematics and then, after changing the angles, kinematics has to be applied [AAH88, Vuk89, Bri88]. This becomes even more complicated if a nonlinear camera system is involved.



X-Y-Space

$\hat{H}$	initial hand position
H	current hand position
$\hat{B}$	initial ball position
B	current ball position
1-10	robot plans

Figure 11: Two-dimensional projection of planning at the robotics task. Each picture is one time step at the world.

surpasses the initial one which demonstrates the appropriateness of the planning procedure[5]. The first action of this plan is then executed by the environment. In the subsequent plots the next steps can be seen, until finally the ball is touched. Note that plans were cut if the reinforcement was closer than an ϵ to its optimum, for the final goal was assumed to be achieved then.

The final trajectory of this trial is plotted in Fig. 12:

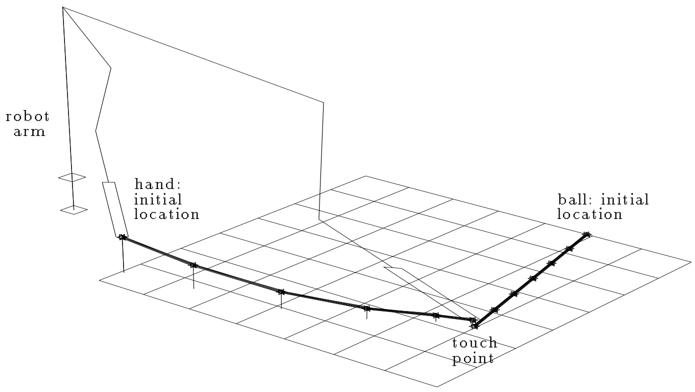


Figure 12: The final trajectory using look-ahead planning.

6.2 Comparison of Direct and Indirect Optimization

Next we will investigate the performance of the experience network. For that reason, we dropped planning and used only the experience network in a controller manner. The corresponding trajectory was clearly not optimal which justifies the subsequent plan optimization (Fig. 13, c.f. Fig. 12):

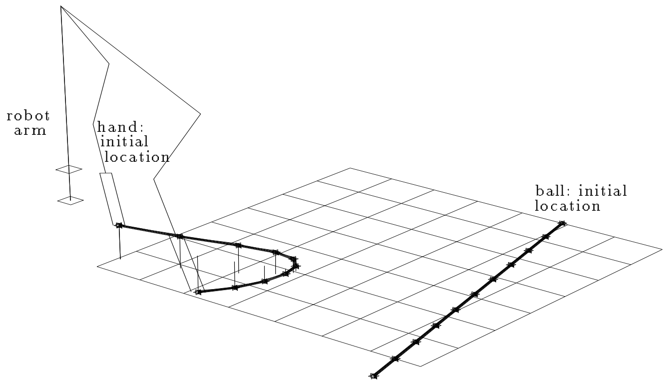


Figure 13: Using the experience network without planning yields poor solutions.

After further 20 000 training steps, during which the behavior of the arm did not change crucially, the experience network was trained better. Using now the expe-

[5]see the next subsection

rience network only yielded very good results with about the same quality of both experience network and planning combined. A resulting trajectory was for instance (Fig. 14):

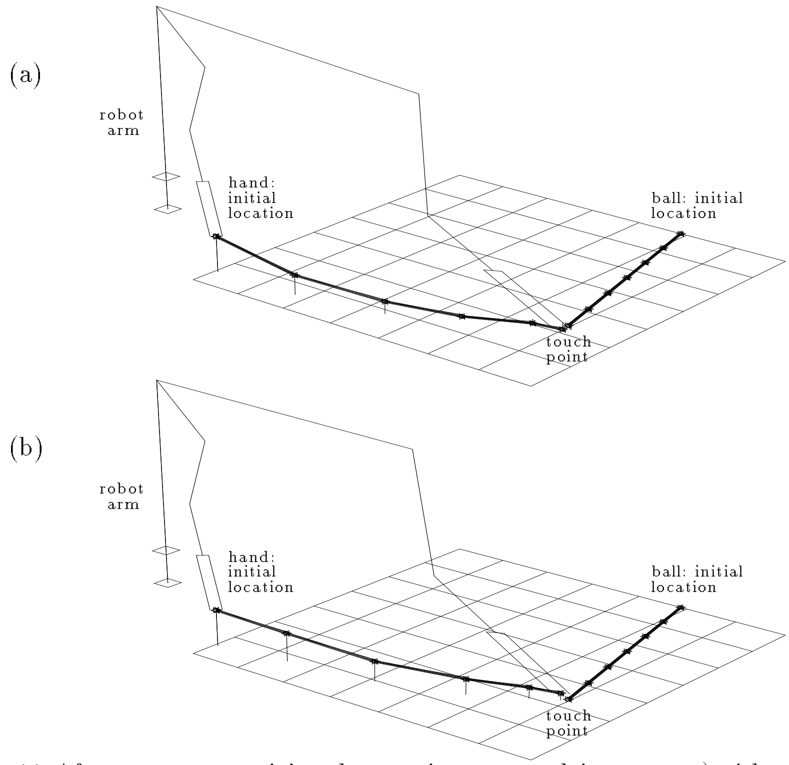


Figure 14: After some more training the experience network improves. a) without planning and b) with planning.

6.3 Changing of Optimization Constraints at Runtime

The experiment with the robot arm is well suited for demonstrating the ability of incorporating new optimization constraints at runtime without further retraining. What happens if the situation changes and the new goal is to approach the ball from above? Obviously many-to-one controllers (direct solution) would fail to adapt to this modification unless they are retrained, because they will generate only one action to each situation. Look-ahead planning, as it is described in this paper, enables such changes during online control. This is now demonstrated on the robotics task.

Changing the constraints is equivalent to changing the energy function (7). In this particular case of approaching the ball from above the energy was now set minimal if the altitude of the manipulator was a certain amount above the surface. The optimal z-reinforcement was switched to its old value in that moment a position above the ball was reached, i.e. the energy was lower than a certain bound. Since the experience network was not trained for this task, we doubled the number of iteration steps. This led without any relearning to (Fig. 15):

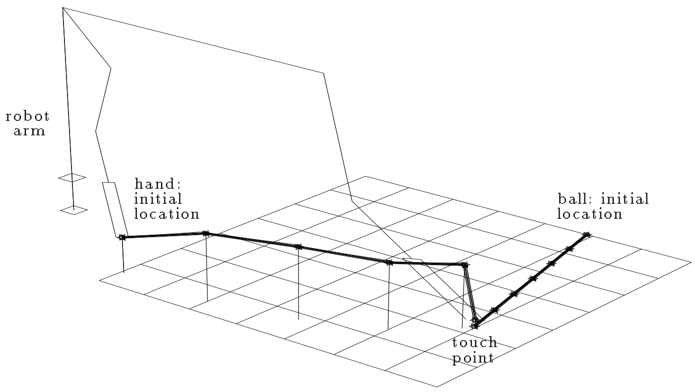


Figure 15: The arm approaches from above due to a modified energy function.

A first implementation on the corresponding real robot arm (Mitsubishi RM V1) showed similar results.

7 Planning in Discrete Spaces

7.1 From Continuous to Discrete Domains

All the results demonstrated above made use of a continuous world. States were encoded continuously as well as reinforcement. Conversely, many reinforcement learning tasks are defined in a discrete world. If states and reinforcements are discrete there is a problem with searching by gradient descent. The better the world model, the more the gradient vanishes (c.f. Fig. 17 described below). This happens although backpropagation nets are differentiable and therefore establish continuous functions. A well-trained model consists of flat regions with gradients closed to 0 and extremely steep regions with huge gradients, which is a consequence of huge weight values. A gradient descent search procedure would either get stuck on a flat plateau or make huge jumps.

The observation that huge weights are responsible for this effect is crucial, because it can be prevented by using a second model network which is trained with the same patterns like the model network but with weight decay. Weight decay prevents large weights and thus smoothes the binary mapping. By using the original model network for prediction only and the smoothed one for gradient propagation, discrete problems become tractable by gradient descent. This is demonstrated on

7.2 The Pole Balancing Task

This task is adopted from [And89, Wie90]. A pole hinged on the top of a wheeled cart travels along a limited track (Fig. 16 a). There were four real-valued state variables visible: the horizontal position of the cart x, its velocity $\dot{x}$, the angel of the cart θ, and its angular velocity $\dot{\theta}$. The reinforcement is binary: one reinforcement

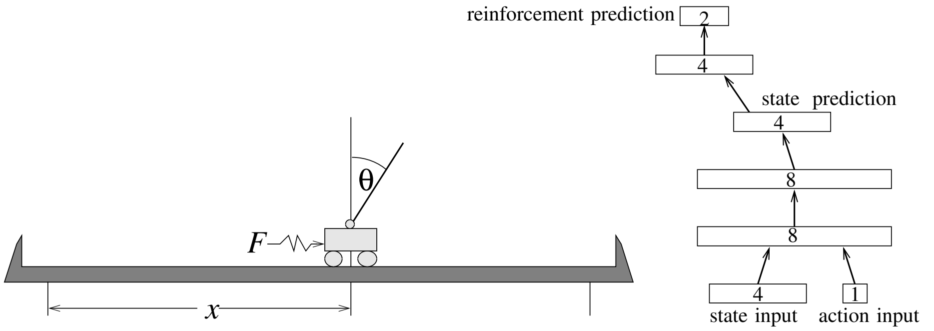


Figure 16: (a) The pole balancing task. The pole is to balanced by applying a force either from the left or from the right to the cart. (b) the model network.

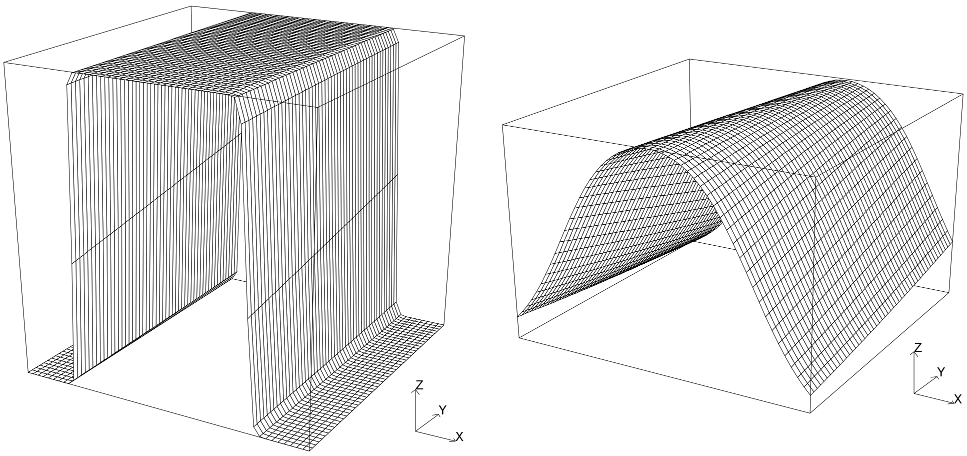


Figure 17: The reinforcement prediction at the pole balancing task. The x-axis corresponds to the angle of the pole, the y-axis to the position of the cart. Both velocities were set to 0. As an example the reinforcement prediction of the angle sensor is shown: the z-axis measures the reinforcement in the model network (a) and its smoothed counterpart (b).

component[6] was 1 as long as the cart did not touch the boundary, and 0 otherwise; the other was 1 as long as the angle of the pole did not cross a certain bound, 0 otherwise. (Its dynamics are listed in the appendix.)

The architecture of the model network is shown in Fig. 16 b (we dropped the experience network and initialized plans by zero-forces). For demonstration one of the reinforcement components of the model network after 1200 learning steps is plotted in Fig. 17 a. The corresponding predictions of the smoothed model (Fig. 17 b) were inaccurate, of course, but the gradient did not vanish indeed. Using this smoothing technique, planning was applicable to the pole balancing problem without any further modification.

[6] By using two reinforcement instead of one as it is described in [And89] model training becomes much faster.

We tested the algorithm with several different look-aheads and randomly selected start configurations. The results are summarized in Tab. 1.

initial configuration		look-ahead N						
x $\in [-2.4; 2.4]$ $\dot{x} = 0$	θ $\in [-0.21; 0.21]$ $\dot{\theta} = 0$	1	2	3	4	5	7	10
-2.076	0.022	1334	706	811	>20000	>10000	>10000	>10000
-0.993	0.011	276	301	1266	>10000	>10000	>10000	>10000
-0.652	-0.035	258	269	335	1547	582	>10000	>10000
0.000	0.000	313	330	367	367	662	>10000	>10000
-0.872	0.053	286	311	389	1579	>10000	952	884
-0.892	0.114	307	361	409	617	912	617	475
-0.149	0.055	313	342	391	499	670	499	710
0.413	-0.131	296	287	302	347	357	726	434
2.145	-0.112	100	157	207	256	336	256	1531
-0.676	-0.183	190	202	190	216	223	231	227
1.843	0.179	111	82	87	77	77	77	61
on average		344	304	432	–	–	–	–

Table 1: The results of pole balancing tasks in terms of number of balancing steps before the pole falls for different look-aheads N and several randomly chosen initial configurations. Steering the cart randomly would make the pole falling after on average 10 steps.

8 Exploration

Up to now we did not investigate exploration at learning time. The actions performed on the world were results of the planning procedure which has an important principal drawback, namely the problem of getting stuck in local optimal solutions. It might happen that a controller always generates the same trajectory, although there exists a better one which has never been tested before, because this trajectory is the optimal *known* trajectory.

The need of exploration has been recognized by many authors. We will summarize two ways of exploration.

- **Training models by randomly generated patterns**. In order to explore the whole world regularly, in a first phase, the phase of model training, actions are generated randomly [MB89, Mun87, Sut90]. Asymptotically, this ensures to explore the whole state space.

  This method has three important drawbacks. Firstly, during model training actions are not optimized. An autonomous agent would behave extremely poor until his model is sufficiently well trained because there is that artificial distinction of a random walk phase and a goal oriented phase [Mun87]. Secondly, training is a rather long task. We will discuss this second point by an example. In [Sut90] a standard reinforcement learning problem is described in which an autonomous agent has to find a target position in a discrete domain. At this

Figure 18: The competence network is trained to the squared error of the model. It is used for exploring states at which the accuracy of the model is poor.

target position the reinforcement is 1 and 0 elsewhere. By walking randomly it is obvious that a lot of locations will be visited several times before the target position is found by random. It can be shown that this expected first exploration time increases exponentially in the distance of start and target position. Lastly, even those random walks fail in assuring equal distributions. This is obvious for this example.

- Sato et al. [SAT90] overcome this problem for discrete domains by associating a counter to each position/action pair. The counter counts for how long each combination of state and action has not been visited. During the exploration walk positions are preferred which have been visited less. This method outperforms pure random walks, but has two drawbacks itself. Firstly, it works in discrete domains only and therefore cannot be used in those experiments described above. Secondly, the number of visits can usually not be assumed to be a measure for the accuracy of the model. E.g. there might be some regions in state space which are easy to learn, such that a few visits are sufficient, and other regions are difficult to learn and thus should be visited more often. In conclusion just counting the numbers of visits is not sufficient, too. Instead the agent should know about its model's accuracy at each position.

The idea is to solve the problem by a differentiable map which learns the accuracy of the model; similar ideas may be found in [Ś91, Sch90]. This network, the *competence network* receives similar to the experience network the state of the world and, in case of recurrent networks, the context information as input and has to learn the error of the model. Such a network can easily be used for exploration – the reinforcement energy would be extended by a competence term which is minimal if the plan leads to regions where the model is most inaccurate. Formally, let $\Gamma(\vec{s}, \vec{c}, \vec{a})$ be the function[7] realized by the competence network. It measures some normalized error of the model. Then the extended energy function can be defined as:

$$E \quad = \quad E_{\text{reinf}} + \gamma \cdot (\max(\Gamma) - \Gamma(\vec{s}, \vec{c}, \vec{a})) \tag{12}$$

The gain parameter γ weights the exploration and the goal optimization. At the beginning, a relatively large $\gamma > 0$ supports exploring the state space, until the output of the competence network becomes more and more flat. Later γ can be set to a negative value which makes the agent to avoid states with less accuracy.

Obviously, training the competence network to predict the error of the model network componentwise is of the same complexity as model training, since if the competence network once has learned to predict that error well enough, the sum of

[7]Reminder: $\vec{s}$ is the current state, $\vec{c}$ is the activation of the context units (in case of a recurrent model network) and $\vec{a}$ is the action to be performed.

the outputs of both networks predicts exactly the world's behavior (except a small remainder). Thus, we cannot expect that the task of learning the error is learnable with the same type of approximator at all. This changes if the experience network has to acquire the squared error. If the world's mapping is difficult to learn in a certain region, the experience network would approximately predict the variance of this region.

There is a second problem concerning training the competence network. Whenever some regions of the state space are more often visited (in terms of learning complexity) than others, the model slowly forgets about these others. This effect is known as *relearning* problem of backpropagation networks, i.e. for partially changing the function of a backpropagation network the whole training set or – more general – training patterns from the whole state space – must be represented again. For accomplishing a good exploration the competence network should take this into account.

- **Exploration by a single unit.** The simplest way of constructing a competence map is adding a single unit to the model network [Sch90]. This is expected to work fine if and only if the error of the model network is (close to) some semilinear combination of activations of the model network.
- **Exploration with feature maps.** Feature maps [Koh84, Koh88, RMS91] can be used for approximating the probability distribution over pattern spaces. By simply presenting each observed state-action-context combination to a feature map, its underlying distribution can be estimated. This can also be done in an error-driven manner by associating a simple value (or an adaptive function) to each feature unit, as described in [FHM+91]. Then feature units avoid regions with larger error[8]. With this technique it is possible to measure the accuracy of the model network. The larger the receptive field of a feature unit, the larger the corresponding error of the model network in this receptive field.

  A problem of using feature maps is that the number of feature units is finite, thus the corresponding competence network realizes a step function which has a discrete output. Thus, the appropriate number of feature units has to be determined.
- By associating an output value to each feature unit which averages the error value observed in its receptive field errors can be learned. A straightforward extension of this algorithm is the contents of [FHM+91].
- **Exploration with backpropagation networks.** Backpropagation for competence maps has the nice property that its learning abilities correspond to that of the model. Moreover, a backpropagation network has a non-discrete output and is differentiable in its whole domain. Thus, it seems to be an attractive candidate for learning the squared error function.

  There are problems in assuring that the error is learned in a reasonable amount of time. If a certain region has never been visited before, nothing prevents the network from predicting an arbitrarily small error in that region. This again is the relearning problem of continuous networks. There is a vague idea how to get rid of this and how to care about relearning problems of the model as

[8] That is somehow contrary to what has been described in [FHM+91].

well. Consider a region in state/action space which has not been explored for quite a long time. Then one expects the model to be inaccurate, because it was trained on other regions for a long time (relearning problem). Thus, only the regions which have recently been visited relatively often should have a low error prediction. This can be achieved by training the competence network with a large set of *shadow training patterns* which are equally distributed in input space and have all the same output value, namely the maximal squared error which should be normalized to 0.5 [9], and with a very small learning rate. Then the confidence network would be expected to predict approximately the largest possible error, if a region has not been explored for a long time. On the other hand, the network would associate approximately correct error values for regions explored recently. This is exactly what we want the competence network to do.

Note that we have not yet implemented the exploration strategies discussed above.

9 Discussion

We presented a planning procedure which optimizes actions iteratively. An experience network proposes an initial plan which is subsequently evaluated by a chain of model networks. In order to optimize reinforcement gradient descent is applied. Knowledge is gradually shifted into the experience network which reduces the planning duration progressively. In addition, a competence network ensures exploration in state/action space. This is important for constructing complete, accurate models.

By using gradient descent and a limited look-ahead we made two important assumptions about the world. These assumptions are crucial for applying this technique:

- By using finite plans, i.e. N is finite, effects beyond this time window are not taken into account. Conversely, in [And86] the probability that an action affects a reinforcement n time steps later is exponentially decreasing in n. Thus, in principle there is no temporal bound. On the other hand in our approach the weighting function g (c.f. (7)) allows to weight reinforcement arbitrarily within that time window.
- Gradient descent is an incomplete search heuristic which might get stuck in local minima. Drawbacks like this usually appear when incomplete search heuristics are used.

The technique presented in this paper combines direct and indirect reinforcement learning methods. The algorithm inherits benefits of both approaches while eliminating most of their deficiencies. a) The training is relatively fast, since the model network does not include temporal effects. b) Decisions are optimized due to the current "knowledge" in the system, and no controller lags behind the model network. c) The incorporation of additional constraints to the objective function at runtime is possible as demonstrated. d) By using a probabilistic experience network the planning algorithm is able to act as a non-deterministic many-to-many

[9] 0.5 is approximately the output of a backpropagation network initialized with very small random values, and hence serves well as the maximal error.

controller. e) The training of the experience network is more effective due to its (by planning) optimized training patterns.

10 Appendix

The dynamics of the pole to be balanced are given by the following equations [And89, Wie90]: Four state variables, namely horizontal position $x(t) \in [2.4\text{m}, 2.4\text{m}]$ of the cart, its velocity $\dot{x}(t) \in [1.5\frac{\text{m}}{\text{s}}, 1.5\frac{\text{m}}{\text{s}}]$, the pole angle $\theta(t) \in [0.21, 0.21]$ and its angular velocity $\dot{\theta}(t) \in [2, 2]$ are visible at each time t. The goal is to generate a sequence of forces $F \in \{10\text{nt}, 10\text{nt}\}$ such that the pole is balanced as long as possible. The dynamics of pole and cart are given by:

$$
\begin{aligned}
\ddot{\theta} &= \frac{-g\sin\theta + \cos\theta\left[\frac{-F - m_p l \dot{\theta}^2 \sin\theta + \mu_c \operatorname{sgn}(\dot{x})}{m_c + m_p}\right] - \frac{\mu_p \dot{\theta}}{m_p l}}{l\left[\frac{4}{3} - \frac{m_p \cos^2\theta}{m_c + m_p}\right]} \\
\ddot{x} &= \frac{F + m_p l\left[\dot{\theta}^2 \sin\theta - \ddot{\theta}\cos\theta\right] - \mu_c \operatorname{sgn}(\dot{x})}{m_c + m_p}
\end{aligned} \tag{13}
$$

with

$$
\begin{aligned}
m_c &= 1.0\text{ kg} &&= \text{mass of the cart,} \\
m_p &= 0.1\text{ kg} &&= \text{mass of the pole,} \\
l &= 0.5\text{ m} &&= \text{distance from center of mass of pole to the pivot,} \\
\mu_c &= 0.0005 &&= \text{coefficient of friction of cart on track,} \\
\mu_p &= 0.000002 &&= \text{coefficient of friction of pivot and} \\
g &= 9.81\ \tfrac{\text{m}}{\text{s}^2} &&= \text{acceleration due to gravity.}
\end{aligned}
$$

The two reinforcements measure whether the trial failed or not.

$$
\begin{aligned}
r_1 &= \begin{cases} 1 & \text{if } 0.21 < \theta < 0.21 \\ 0 & \text{otherwise} \end{cases} \\
r_2 &= \begin{cases} 1 & \text{if } 2.4\text{m} < x < 2.4\text{m} \\ 0 & \text{otherwise} \end{cases}
\end{aligned} \tag{14}
$$

These differential equations are discretized with the stepsize 0.02.

Acknowledgements

The authors with to thank Joachim Herzberg, Alexander Horz, Jörg Kindermann, Alexander Linden, Heinz Mühlenbein, Uwe Schnepf, Frank Śmieja, Donald Sofge, Christoph Tietz, Gerd Veenker and Christoph Windheuser and the groups at GMD and Bonn University for numerous fruitful discussions. Thanks also to Michael Contzen and Michael Faßbender for their help with the robot arm.

References

[AAH88] C.H. An, Ch.G. Atkeson, and J.M. Hollerbach. *Model-Based Control of a Robot Manipulator.* The MIT Press, Cambridge, MA, 1988.

[And86] C. W. Anderson. Learning and problem solving with multilayer connectionist systems. Technical Report COINS TR 86-50, Dept. of Computer and Information Science, University of Massachusetts, Amherst, MA, 1986.

[And89] C.W. Anderson. Learning to control an inverted pendulum using neural networks. *IEEE Control Systems Magazine*, 9(3):31–37, 1989.

[Bar89] A. G. Barto. Connectionist learning for control: An overview. Technical Report COINS TR 89-89, Dept. of Computer and Information Science, University of Massachusetts, Amherst, MA, September 1989.

[BAS82] A. G. Barto, C. W. Anderson, and R. S. Sutton. Synthesis of nonlinear control surfaces by a layered associative search network. *Biological Cybernetics*, 43:175–182, 1982.

[Bri88] C. Bridgewater. Tracking moving objects – a behaviour-based approach. Technical report, Dept. of AI, University of Edinburgh, Edinburgh, September 1988.

[BS90] A. Barto and S.P. Singh. On the computational economics of reinforcement learning. In D.S. et al. Touretzky, editor, *Connectionist Models, Proceedings of the 1990 Summer School*, pages 35–44, San Mateo, CA, 1990. Morgan Kaufmann.

[Chr90] R.L. Chrisley. Cognitive map construction and use: A parallel distributed approach. In D.S. et al. Touretzky, editor, *Connectionist Models, Proceedings of the 1990 Summer School*, pages 287–300, San Mateo, CA, 1990. Morgan Kaufmann.

[DR89] R. Durbin and D. E. Rumelhart. Product units: A computationally powerful and biologically plausible extension to backpropagation networks. *Neural Computation*, 1:133–142, 1989.

[Elm88] J. L. Elman. Finding structure in time. Technical Report CRL Technical Report 8801, Center for Research in Language, University of California, San Diego, 1988.

[FHM+91] D. Fox, V. Heinze, K. Möller, S.B. Thrun, and G. Veenker. Learning by error-driven decomposition. In Simula/Kohonen, editor, *Proceedings of International Conference on Artificial Neural Networks (to appear)*, Amsterdam, 1991. Elsevier Publisher.

[FW63] K.S. Fu and M.D. Waltz. A heuristic approach to reinforcement learning control systems. *IEEE Transactions on Information Theory*, 9:390–398, 1963.

[Ghe89] M. Gherrity. A learning algorithm for analog, fully recurrent neural networks. In *Proceedings of the First International Joint Conference on Neural Networks, Washington, DC*, San Diego, 1989. IEEE, IEEE TAB Neural Network Committee.

[Jor86] M. I. Jordan. Attractor dynamics and parallelism in a connectionist sequential machine. In *Proceedings of the Conference on Cognitive Science*, 1986.

[Jor89] M. I. Jordan. Generic constraints on unspecified target constraints. In *Proceedings of the First International Joint Conference on Neural Networks, Washington, DC*, San Diego, 1989. IEEE, IEEE TAB Neural Network Committee.

[JR90] M. Jordan and D. E. Rumelhart. Forward models: Supervised learning with a distal teacher. submitted to Cognitive Science, 1990.

[KL90] J. Kindermann and J. Linden. Inversion of neural networks by gradient descent. *Parallel Computing*, 14(3):277–286, 1990.

[KMUS90] M. Kawato, Y. Maeda, Y. Uno, and R. Suzuki. Trajectory formation of arm movement by cascade neural network model based on minimum torque-change criterion. *Biological Cybernetics*, (62):275 – 288, 1990.

[Koh84] T Kohonen. *Self-Organization and Associative Memory.* Springer, Berlin New York, 1984.

[Koh88] T Kohonen. *Self-Organization and Associative Memory, 2nd. edition.* Springer, Berlin NewYork, 1988.

[KP90] J. F. Kolen and J. B. Pollack. Bach propagation is sensitive to initial conditions. Technical Report 90-JK-BPSIC, Computer and Information Science Department, Ohio State University, Columbus, April 1990.

[Lak81] S. Lakshmivarahan. *Learning Algorithms Theory and Applications.* Springer-Verlag, New York, 1981.

[Lju87] L. Ljung. *System Identification – Theory for the user.* Prentice-Hall, New Jersey 07632, 1987.

[LK89] A. Linden and J. Kindermann. Inversion of multilayer nets. In *Proceedings of the First International Joint Conference on Neural Networks, Washington, DC*, San Diego, 1989. IEEE.

[MB89] M. C. Mozer and J. Bachrach. Discovering the structure of a reactive environment by exploration. Technical Report CU-CS-451-89, Dept. of Computer Science, University of Colorado, Boulder, November 1989.

[Mel89] B. W. Mel. Murphy: A neurally-inspired connectionist approach to learning and performance in vision-based robot motion planning. Technical Report CCSR-89-17A, Center for Complex Systems Research Beckman Institute, University of Illinois, 1989.

[Moz88] M. C. Mozer. A focused backpropagation algorithm for temporal pattern recognition. Technical Report CRG-TR-88-3, Depts. of Psychology and Computer Science, University of Toronto, Toronto, Jun 1988.

[Mun87] P. Munro. A dual backpropagation scheme for scalar-reward learning. In *Ninth Annual Conference of the Cognitive Science Society*, pages 165–176, Hillsdale, NJ, 1987. Cognitive Science Society, Lawrence Erlbaum.

[Nil82] N. J. Nilsson. *Principles of Artificial Intelligence.* Springer Verlag, Berlin, 1982.

[NT89] K. Narandra and M.A.L. Thathachar. *Learning automata: an introduction.* Prentice Hall, Englewood Cliffs, NJ, 1989, 1989.

[NW89] D. Nguyen and B. Widrow. The truck backer-upper: An example of self-learning in neural networks. In *Proceedings of the First International Joint Conference on Neural Networks, Washington, DC*, San Diego, 1989. IEEE, IEEE TAB Neural Network Committee.

[Pea88] B. Pearlmutter. Learning state space trajectories in recurrent neural networks. In D. Touretzky, G. Hinton, and T. Sejnowski, editors, *Proceedings of the 1988 Connectionist Models Summer School*, Palo Alto, 1988. Morgan Kaufmann.

[Pea90] B.A. Pearlmutter. Dynamic recurrent neural networks. Technical Report CMU-CS-88-191, School of Computer Science, Carnergie Mellon University, Pittsburgh, PA 15213, 1990.

[Pin87] F. J. Pineda. Generalization of backpropagation to recurrent neural networks. *Physical Review Letters*, 59(19):2229–2232, 1987.

[RF87] R. Rohwer and B. Forrest. Training time-dependence in neural networks. In M. Caudill and C. Butler, editors, *Proceedings of IEEE Conference on Neural Networks*, San Diego, 1987. IEEE.

[RF89] A. J. Robinson and F. Fallside. Dynamic reinforcement driven error propagation networks with application to game playing. to be presented at the Eleventh Annual Conference of the Cognitive Science Society, Ann Arbor, 1989.

[RHW86] 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. Vol. I + II.* MIT Press, 1986.

[RMS91] H. Ritter, T. Martinez, and K. Schulten. *Neuronale Netze (in German).* Addison-Wesley, Bonn, München, Reading MA, 1991.

[Rob89] A. J. Robinson. *Dynamic Error Propagation Networks.* PhD thesis, Cambridge University Engineering Dept., Cambridge, UK, February 1989.

[Ś91] F.J. Śmieja. Multiple network systems (MINOS) modules: Task division and module discrimination. In *Proceedings of the 8th AISB conference on Artificial Intelligence, Leeds, 16–19 April, 1991*, 1991.

[Sam59] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journnal on research and development*, 3:210–229, 1959.

[SAT90] M. Sato, K. Abe, and H. Takeda. Learning control of finite markov chains with explicit trade-off between estimation and control. In D.S. et al. Touretzky, editor, *Connectionist Models, Proceedings of the 1990 Summer School*, pages 287–300, San Mateo, CA, 1990. Morgan Kaufmann.

[Sch90] J.H. Schmidhuber. Making the world differentiable: On using supervised learning fully recurrent neural networks for dynamic reinforcement learning and planning in non-stationary environments. Technical report, Technische Universität München, München, West Germany, 1990.

[Ski38] B.F. Skinner. *The behavior of organisms: An experimantal analysis.* D. Appleton Century, New York, 1938.

[Sut84] R. S. Sutton. *Temporal Credit Assignment in Reinforcement Learning.* PhD thesis, University of Massachusetts, 1984.

[Sut90] R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. To appear in: Proceedings of the Seventh International Conference on Machine Learning, June 1990, 1990.

[SW90] A.D. Sofge and D.A. White. Neural network based process optimization and control. In *Proceedings of the 29th IEEE Conference on Decision and Control (to appear)*, Dezember 1990.

[TL90] S. Thrun and A. Linden. Inversion in time. In *Proceedings of the EURASIP Workshop on Neural Networks*, Sesimbra, Portugal, February 1990. EURASIP.

[TML90] S. Thrun, K. Möller, and A. Linden. Adaptive look-ahead planning. In *Proceedings OEGAI 90*, 1990.

[TML91] S. Thrun, K. Möller, and A. Linden. Planning with an adaptive world model. In D. S. Tourtezky and R. Lippmann, editors, *Advances in Neural Information Processing Systems 3*, San Mateo, 1991. Morgan Kaufmann.

[TS90] S. Thrun and F. Śmieja. A general feed-forward algorithm for gradient-descent in connectionist networks. Technical Report 483, GMD, Sankt Augustin, FRG, November 1990.

[Vuk89] M. Vukobratović. *Introduction to Robotics.* Springer Publisher, Berlin, New York, 1989.

[WB90] S. D. Whitehead and D. H. Ballard. Active perception and reinforcement learning. Technical report, Department of Computer Science, University of Rochester, New York, 1990.

[Wer87] 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*, SMC-17:7–19, 1987.

[Wie90] A.P. Wieland. Evolving control for unstable systems. In D.S. et al. Touretzky, editor, *Connectionist Models, Proceedings of the 1990 Summer School*, pages 91–102, San Mateo, CA, 1990. Morgan Kaufmann.

[Wil87] R. J. Williams. Reinforcement-learning connectionist systems. Technical Report NU-CCS-87-3, College of Computer Science, Northeastern University, Boston, 1987.

[WZ88] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. Technical Report ICS Report 8805, Institute for Cognitive Science, University of California, San Diego, CA, 1988.