# Concepts and facilities of a neural reinforcement learning control architecture for technical process control

Martin Riedmiller
Institut für Logik, Komplexität und Deduktionssysteme
Universität Karlsruhe, D-76128 Karlsruhe, Germany
e-mail: riedml@ira.uka.de

*Abstract*—
**The paper presents the concepts of a neural control architecture that is able to learn high quality control behaviour in technical process control from scratch. As the input to the learning system, only the control target must be specified. In the first part of the article, the underlying theoretical principles of dynamic programming methods are explained and their adaptation to the context of technical process control is described. The second part discusses the basic capabilities of the learning system on a typical benchmark problem, where a special focus lies on the quality of the acquired control law. The application to a highly nonlinear chemical reactor and to an instable multi output system shows the ability of the proposed neural control architecture to learn even difficult control strategies from scratch.**

**keywords:** intelligent control, reinforcement learning, neural control, technical process control

## I. Introduction

Today, there exits a large variety of controller design methods provided by classical analytical control theory. The methods differ in the type of systems they are tailored for - e.g. linear or nonlinear systems, systems with single input/ output or multiple inputs/ outputs - the type of input information they are using - e.g. state space approaches - and the expected properties of the closed-loop behaviour - e.g. robustness, optimality.

With the upcoming of new methods of controller design like fuzzy logic or neural networks, an additional aspect of controller design has to be taken into account: the type and quality of a priori knowledge that is assumed by the respective design method [10]. Basically three types of methods can be distinguished: a) methods, that assume a priori control knowledge (as *fuzzy control* does) , b) methods, that assume knowledge about system behaviour (as in conventional analytical controller design), and finally c) methods, that assume neither control knowledge nor system knowledge in advance.

Clearly, the last situation is the most challenging for a control system - the designer only specifies *what* goal has to be achieved and the system must learn by itself *how* to fulfill the requirements. Neural control systems that are able to learn control when only a minimum of information is given - namely the goal to be achieved - are the focus of this article. Note that in contrast to other neural controller methods no training examples (as assumed in the case of supervised learning) and no reference model (as assumed by the neural MRAC[1] approach [9]) is assumed here.

When controlling dynamical systems - where temporal dependencies between control input and system's output play a central role - one inevitably faces the so-called *temporal credit assignment problem*: which decision(s) in a control trial are responsible for the final outcome - or, formulated from the viewpoint of learning: which decision must be changed to what extent such that the overall control goal is achieved in a satisfactory way? As shown in [3], [1], problems of this kind of *sequential decision making* can be elegantly solved within the framework of *dynamic programming (DP)*. Using such techniques in learning systems to tackle hard-to-solve dynamic optimization tasks in various application areas has recently gained much interest [2], [7], [6].

Our work concentrates on developing concepts for a self-learning controller architecture based on neural dynamic programming methods, where only the control target must be specified by the designer [11], [14]. The underlying optimization approach guarantees a high quality of the final controller policy which is an important point for a self-learning controller.

The article is organized as follows: First, we will give an introduction to the ideas of dynamic programming and describe how it can be used in a learning system. We will then describe two frameworks, that allow to formulate the original control problem as an optimization problem where we especially consider the assumed learning situation. Further we explain the use of a neural network and derive a learning rule for the weights. Some discussion about model-free and model-based variants of the control architecture, about exploration issues and the control of stochastic systems is given at the end of section II. Section III discusses an extension of the basic architecture that allows to generate arbitrary control signals. Section IV is dedicated to the examination of the special capabilities of the proposed control architecture. In particular, to measure the quality of the learned control behaviour, it is compared to an analytically designed control law. Two further example tasks focus on the difficult control of a nonlinear chemical reactor and the control of a double inverted pendulum as an

[1] Model reference adaptive control

example for a multi output system.

## II. Learning control from scratch

### A. Formal problem description

In the following we consider the control of dynamical systems that can be described in a discrete-time state space notation

$$\begin{aligned} x_{t+1} &= f(x_t, u_t) \\ y_t &= g(x_t) \end{aligned} \tag{1}$$

Here, $x_t$ denotes the vector of state variables at time step $t$, $u_t$ denotes the vector of control signals and $y_t$ is the vector of observed output signals of the plant. The function $f$ describes the dynamical behaviour, and $g$ denotes the output function that computes $y$ out of current state information.

The control task is to find an appropriate control policy

$$\pi : \mathcal{X} \mapsto \mathcal{U}, u_t := \pi(x_t),$$

such that after a certain time $t_0 > 0$, the output of the plant $y_t$ equals a given target vector $y^{\text{target}}$, such that

$$||y_t - y^{\text{target}}|| < \delta, \forall t > t_0. \tag{2}$$

Up to now, this is a typical control problem formulation that is addressed by many design methods, of both classical control theory and alternative approaches, as for example fuzzy control. What makes a self-learning framework very special, is the assumed lack of both a priori knowledge about system behaviour (as required by analytical methods) and the lack of control knowledge (as required to establish a fuzzy controller). Thus a self-learning controller is characterized by the fact that

- system's equations $f$ and $g$ are unknown
- only the target value $y^{\text{target}}$ is given.

In the following we will show, how the control problem can be formulated as a dynamic optimization problem that can be effectively solved by dynamic programming methods.

### B. Dynamic Programming

Consider a dynamical system as defined in equation 1. The controller can make a control decision at discrete time steps $t_0, t_1, \ldots$, where the difference between two steps is called the control interval $\triangle_t$. In each time step, immediate costs $r_t$ arise, that may depend on both the current state $x_t$ and the applied action $u_t$, thus

$$r : (\mathcal{X}, \mathcal{U}) \mapsto \Re, r_t := r(x_t, u_t).$$

To evaluate the performance of a certain policy $\pi$, we now may consider the summed costs that occur during a control trial accumulated over a certain period in time, when the initial state of the system is $x_0$,

$$J^\pi(x_0) := \sum_{t=0}^{N} r_t = \sum_{t=0}^{N} r(x_t, \pi(x_t)). \tag{3}$$

The number of time steps $N$ is called the 'horizon' of the problem. For the control of technical processes we are interested in the costs that occur during a theoretically infinite period of time. In particular, we are interested in an optimal policy $\pi^*$ that results in the minimum accumulated costs

$$J^*(x_0) := \min_{\pi \in \Pi} \sum_{t=0}^{\infty} r(x_t, \pi(x_t)). \tag{4}$$

Note that by the summation of the immediate costs, the complete control trajectory that results out of the application of a certain policy $\pi$ has to be considered for optimization. This makes the optimization problem a really hard task: control decisions with low immediate costs $r(x, u)$ may look advantageous first, but due to the dynamical behaviour of the plant may have unfortunate consequences in the future - which results in high overall costs. On the other hand an 'expensive' decision may result in a fortunate situation, where the future costs to be expected are very low.

Problems of the above kind can be solved by dynamic programming methods. These methods have in common to be based on the 'principle of optimality' first formulated by Bellman [4]. It states that

$$J^*(x) = \min_{u \in \mathcal{U}(x)} \{r(x, u) + J^*(f(x, u)), \tag{5}$$

meaning, that the optimal solution for state $x$ is equal to the sum of immediate costs plus the optimal costs for the successor state, if the (locally) optimal action $u$ is applied.

The *value iteration* method which is based on the above dynamic programming principle is of special interest for the application within a self-learning controller. The underlying idea is to approximate the optimal accumulated cost function $J^*$ by repeatedly improving estimates for the optimal costs for each state $J_0, J_1, \ldots, J_k \ldots$. The iteration formula to compute $J_{k+1}$ out of $J_k$ is directly motivated from the principle of optimality (eq. 5):

$$\forall x \in \mathcal{X} : J_{k+1}(x) := \min_u \{r(x, u) + J_k(f(x, u))\}. \tag{6}$$

In the above equation the iteration is repeatedly carried out for each state in state space $\mathcal{X}$ until $J_k$ converges against the optimal cost function. This implicitly assumes a *finite* state space. To tackle problems with an infinite number of states - for example if the state variables can take continuous values - the original procedure must be modified. This is discussed in section II-C.

Under certain assumptions which are discussed in section II-D convergence of the value iteration method can be proved, i.e.

$$lim_{k \to \infty} J_k = J^*.$$

Typically, one is not only interested in determining the optimal costs for each state but moreover to find the according optimal policy. It can be shown [5] that once the

optimal cost function has been determined, an optimal policy $\pi^*$ can be easily derived:

$$\pi^*(x) := \arg\min_{u \in \mathcal{U}(x)} \{r(x,u) + J^*(f(x,u)).\} \qquad (7)$$

The optimal policy selects the action $u$ that minimizes the sum of *immediate* costs $r(x,u)$ and accumulated future costs $J^*$ that will occur, when this action is applied.

### C. Value Iteration and Learning

The implementation of the value iteration formula (eq. 6) assumes a finite number of states - which is typically not fulfilled for a real physical system. A solution of this problem is provided by the *Real Time Dynamic Programming (RTDP)* approach proposed in [1]. Roughly, the idea is to control the system with the current knowledge available - represented by the current estimation for the cost function $J_k$ - and to apply the value iteration formula only to those states, that actually occur during a control trial. To do so, the iteration rule 6 is split into three parts:

- *select an action* $u_t$

$$u_t = \arg\min_{u \in \mathcal{U}} \{r(x_t,u) + J_k(f(x_t,u))\}, \qquad (8)$$

- *apply* $u_t$ *to the plant*

$$x_{t+1} = f(x_t, u_t) \qquad (9)$$

- *update the cost function*

$$J_{k+1}(x_t) := r(x_t,u_t) + J_k(x_{t+1}) \qquad (10)$$

The above steps - action selection (eq. 8), application to the plant (eq. 9) and update of the value function (eq. 10) - then exactly implement the *value iteration* step in equation (6) for state $x_t$.

*Init* plant $x_0 \in \mathcal{X}^{init}$
*Repeat* {
  *select action*
  *apply action to plant*
  *update* $J$ }
*Until* stop = yes
*finish control trial*

Fig. 1. Basic algorithm for a self-learning controller

The basic learning algorithm as shown in figure 1 implements the repeated application of the three steps. At the beginning of a control trial, the system is assumed to be in an arbitrary initial state. Then, a control signal is selected according to the current knowledge as represented by the cost function $J_k$ (eq. 8). This action is applied to the plant (eq. 9) and the resulting state $x_{t+1}$ is observed. Now, the new costs $J_{k+1}$ for state $x_t$ can be computed as the sum of observed immediate costs $r(x_t,u_t)$ and the current costs $J_k$ of the successor state $x_{t+1}$. This procedure is repeated, until a certain stopping criterion is fulfilled. Then, the current control trial is finished and a new trial is started. This is done until the optimal or at least a sufficiently good policy is reached. By improving its estimation for the optimal costs, the policy of the controller also improves. The basic learning algorithm thus realizes a sort of 'learning from experience'.

What we have to assure, is that the assumptions for the convergence of the value iteration procedure are fulfilled. As we will see in the following section, there are two different kinds of frameworks which will lead to different variations of the basic learning algorithm.

### D. Conditions of convergence

Having introduced the basic principles of learning we now come to the question of how to formulate the original control problem within the framework of dynamic programming. More precisely, when only the control target $y^{target}$ is given, how to formulate the optimization problem - for example how to choose the immediate cost function $r(x,u)$ - such that both value iteration converges and the resulting optimal policy solves the control task? In the following, we will propose two possible frameworks and discuss both their advantages and problems.

The first approach is to formulate the control problem as the search for a path from an initial state to an absorbing terminal state. Problems of this kind are named 'shortest path' problems, where 'short' here means a path with low accumulated costs. Convergence of the value iteration algorithm against the optimal value function then can be shown under the following main assumptions[2]:

**Framework 1:** event based (shortest path) scenario

1. There exists an absorbing terminal state $x^+$, for which

$$\forall u : f(x^+,u) = x^+$$

holds.

2. There exists a 'proper' policy $\pi'$, i.e. there exists an integer $m \leq n$ after which a terminal state is reached, when policy $\pi'$ is applied.

Thus to formulate our control problem within the shortest path scenario, we have to choose a set of terminal target states $\mathcal{X}^+$, which fulfills

$$x \in \mathcal{X}^+ : ||g(x) - y^{\text{target}}|| < \delta. \qquad (11)$$

A control trial stops, whenever a state $x^+$ is reached for which 11 holds. Due to the stopping criterion the corresponding learning procedure is named *event based* algorithm [12]. Figure 2 shows the realization of the event based algorithm.

*Init* plant $x_0 \in \mathcal{X}^{init}$
*Repeat* {
  *select action*
  *apply action to plant*
  *update* $J$ }
*Until* $(\mathbf{x_t} \in \mathcal{X}^+)$
$J_{k+1}(x_t) := 0$

Fig. 2. The event based algorithm stops whenever a terminal state is reached.

[2]for a more precise discussion of the assumptions see [5]

A major problem with this approach is, that the assumption of $\mathcal{X}^+$ being a set of absorbing *terminal* states is typically not true when controlling a real physical system. Instead, control is an ongoing task and a certain target state indeed *can* be left in the future. The event based framework does not allow a proper treatment of this crucial point.

A feasible solution to this problem is to add an additional constraint to the terminal state: the output value has to reach the target with a certain slowness. Then, the controller may be able to keep the output at its target when control continuous. This obvious remedy has two major drawbacks: firstly, additional knowledge must be used to set up the exact specification of the target region. Secondly, a too cautious setting might lead to the loss of optimality of the solution.

Therefore, what we need is a framework that optimizes the control behaviour over an unlimited time period rather than until a certain event has occurred. This is the purpose of the fixed horizon framework [12] which makes the following assumptions:

**Framework 2:** fixed horizon scenario

1. there exists a set of states $\mathcal{X}^0$ with zero immediate costs: $\forall x \in \mathcal{X}^0 \, \forall u : r(x,u) = 0$
2. for all other states immediate costs are positive: $\forall x \notin \mathcal{X}^0 \, \forall u : r(x,u) > 0$
3. it is *possible* to control the system from an arbitrary start state to a cost-free state
4. it is *possible* to keep the system within the cost-free states

Then, the following properties can be shown [14]:

- value iteration converges against the optimal cost function:

$$\lim_{k \to \infty} J_k = J^*$$

- for the resulting optimal policy $\pi^*$ there exists a nonempty set $\mathcal{X}^* \subseteq \mathcal{X}^0$, for which

$$x \in \mathcal{X}^* \Rightarrow f(x, \pi^*(x)) \in X^*$$

holds, i.e. the *resulting* optimal control policy $\pi^*$ controls the system to a subset $\mathcal{X}^* \subseteq \mathcal{X}^0$ and *permanently keeps it within that region.*

To formulate our control problem within the fixed horizon framework we now have to choose

$$r(x,u) := 0 :\Leftrightarrow ||g(x) - y^{target}|| < \delta,$$

and $r(x,u) > 0$, else. Then, if it is *possible* for the controller to control the output close enough to its target value - note that such a policy is *not* assumed to be known - then the value iteration algorithm will converge. The resulting optimal policy will control the system such that after a certain time $t_0$ the output is kept permanently close to its target value. Moreover, the accumulated costs until $t_0$ are minimized. Thus the optimal policy exactly fulfills the requirement as expressed in equation 2.

The respective algorithm - called the fixed horizon algorithm - is shown in figure 3. During training, each control trial is stopped after a predefined fixed number $N$ of time steps rather than after a certain event has occurred. By considering the state at time step $N$ as a possible starting state for a future trial, control sequences of theoretically infinite length can be considered (for a further discussion of this point see [14]).

*Init* plant $x_0 \in \mathcal{X}^{init}$
*Repeat* {
  *select action*
  *apply action to plant*
  *update* $J$ }
*Until* $(\mathbf{t} = \mathbf{N})$
$\mathcal{X}^{init} := \mathcal{X}^{init} \cup \{x_N\}$

Fig. 3. The number of time-steps is the stopping criterion for the fixed horizon algorithm

### E. Neural value iteration

The assumed continuous nature of the state space not only has an influence on the algorithmic realization of the value iteration procedure, but also raises the question of the representation of the cost function. In case of a finite state space with a reasonably restricted number of states, a lookup table representation is an appropriate approach. In case of a continuous state space, a feasible solution is to use a function approximation approach to map the states to their corresponding estimated costs by adapting certain parameters of the function. In our case we use a multilayer perceptron neural network to approximate the cost function. The input layer receives the current state information $x_t$ and the output represents the estimated cost value $J(x_t)$, whereas the hidden nodes are used to deal with the expected nonlinearities of the mapping.

The learning rule for the parameters (=weights) of the net is governed by the value iteration formula in equation 10. However, when using a neural network this assignment cannot be implemented directly. Instead, the assignment has to be expressed in terms of a minimization of an error function that measures the difference between actual output and the new target value. For a complete control trial, the sum of errors for all the visited states must be considered, i.e. the minimization term is given by

$$E_{trial} := \sum_{x_t \in trial} (J(x_t) - (r(x_t,u_t) + J(x_{t+1})))^2.$$

To carry out this minimization, the weights in the network have to be updated accordingly. This can be done by using standard gradient descent techniques based on the application of the back-propagation algorithm to compute the derivatives with respect to the error. Thus the learning rule for a single weight $w$ is

$$w_{k+1} := w_k - \alpha \cdot \frac{\partial E(w_k)}{\partial w_k}$$

where

$$\begin{aligned} \frac{\partial E(w_k)}{\partial w_k} &= \frac{1}{2} \frac{\partial (J_k(x_t) - r(x_t,u_t) - J_{k-1}(x_{t+1}))^2}{\partial w_k} \\ &= \frac{\partial J_k(x_t)}{\partial w_k} (J_k(x_t) - r(x_t,u_t) - J_{k-1}(x_{t+1})). \end{aligned}$$

What crucially distinguishes this learning rule from the supervised learning case, is that here, the target value is not given externally, but a part of it is computed by the neural net itself. Due to this tricky relation between temporal related state evaluations, this special type of learning is called *Temporal Difference (TD)*-learning [15].

### *F. Model based and model free approach*

In the original value iteration formula (eq. 6) the system's transfer function $f$ is assumed to be known to update the cost value for a certain state $x$. In contrast to that, the update of the value function within the RTDP framework (eq. 10) is instead based on the observation of the successor state $x_{t+1}$ that results when the selected action has been applied to the system. Thus the system itself is used to 'compute' $f$ and therefore $f$ must not be known to determine the update. Unfortunately, the knowledge about $f$ is still used at another place: to determine the optimal action by evaluating potential successor states (eq. 8).

A tricky solution of this problem proposed by Watkins [16] is to choose another representation of the value function. Instead of evaluating states, the idea here is to evaluate state/action pairs, i.e. to use a mapping

$$Q : (\mathcal{X}, \mathcal{U}) \mapsto \Re.$$

Here, $Q$ denotes the function that estimates the expected accumulated costs, if an action $u$ was applied in state $x$. Hence the above framework is called *Q-learning*. The underlying idea of gradually approximating the optimal value function based on the value iteration procedure remains exactly the same as in the original framework where the cost function $J_k$ computes the estimated cumulated costs for each state. The relationship between the $Q$ function and the previous value function $J$ is given by

$$J(x) = \min_u \{Q(x, u)\}. \tag{12}$$

Using this relation, the action selection part and the adaptation part of the basic learning procedure (section II-C) can now be reformulated without the use of a model $f$:

- *select an action* $u_t$

$$u_t = \arg\min_{u \in \mathcal{U}} \{Q(x_t, u)\} \tag{13}$$

- *update the cost function*

$$Q_{k+1}(x_{t-1}, u_{t-1}) := r(x_{t-1}, u_{t-1}) + Q_k(x_t, u_t) \tag{14}$$

- *apply* $u_t$ *to the plant*

$$x_{t+1} = f(x_t, u_t) \tag{15}$$

Note that in the above setting, the order of the three steps has been changed and that the value of $x_{t-1}, u_{t-1}$ is now updated instead of the value for $x_t$ as in the case of pure state evaluation. This is due to the fact, that using the $Q$ representation the expected costs of the successor state can not be computed directly, but first the optimal action must be determined. However, this is just a matter of implementation and does not at all constitute a conceptual difference.

To summarize, the above framework allows a complete renunciation of a system model within the controller architecture. This is achieved by changing the representation of the estimated costs as being computed out of state/ action information rather than on state information alone. The underlying learning principle - realizing a value iteration during control to approximate the optimal value function - is exactly the same in both cases. The term $Q-$*representation* might stress this difference more precisely than the commonly used term $Q-$learning[3]

Clearly, the advantage of not assuming a system model is paid by a more complex learning problem. An obvious reason for this is that now, the $Q$ function must express both the direct consequences of an action and its future expected costs, whereas in a model based framework, the direct consequence - i.e. the resulting successor state - is computed by the model and thus the overall task is split into two separate jobs.

### *G. Additional issues*

The above sections explain the main underlying ideas and principles of the self-learning control architecture. However, in order to get it to work, some additional aspects have to be considered.

#### G.1 Exploration

In the above framework, the control signal $u_t$ is determined by always selecting the action that minimizes the expected costs. If we could guarantee, that the current estimation for the accumulated costs always underestimates the optimal costs, this would be a perfect strategy to eventually find the optimal path. But since we are using a neural network to represent the cost function the underestimation cannot be assured because due to interpolation and generalization effects the change of the cost value for one state has unpredictable influence on the evaluation of other states. Thus an extra mechanism must be provided that allows some exploration of the solution space in order to be sure to find an optimal solution.

In our framework, this is done by adding the value of an equally distributed random variable $\zeta$ to the neurally computed accumulated costs before the action is determined. Thus the action selection step becomes

$$u_t = \arg\min_{u \in \mathcal{U}} \{r(x_t, u) + J_k(f(x_t, u)) + \zeta\},$$

where $\zeta$ takes random values in the range of $[0 \ldots \eta]$ and $\eta$ is a parameter of the learning procedure. The heuristic idea behind this is that if the values of the cost function become more and more distinct for possible selections in course of learning, the influence of the random term gradually

[3]The original Q-learning formulation is more general by additionally realizing a stochastic approximation to deal with stochastic dynamic systems.

decreases and the controller follows more and more the current optimal policy represented by $J_k$.

### G.2 State representation

Up to now, we formulated the approach as a typical state space controller, that computes its control signal depending on the current state information $x_t$. This assumes, that the vector of state variables $x_t$ can be completely observed. For real physical systems, this is typically not true; instead only partial and/ or transformed information about the system's state is available in terms of the vector of observed sensor variables $y_t = g(x_t)$. In conventional control theory therefore the concept of an observer has been developed, that computes an estimation of the current state based on the sequence of observations of the output of the system. However, observer design requires the knowledge of the system's equations and thus cannot be applied in the framework of a self-learning controller.

Fortunately, the neural controller is not based on an analytical description of system behaviour, and thus it is consequently not dependent on a certain fixed representation of the system's state. The crucial point is that the state information is somehow reflected in the input to the controller.

One possible such representation of system's state is the vector of past measurements of sensor variables and control signals which is motivated by the *ARMA* or *NARMA* approach to model dynamical systems. Thus we might represent the state $\hat{x}$ in an alternative form,

$$\hat{x}_t := (y_t, y_{t-1}, \ldots, y_{t-n}, u_{t-1}, \ldots, u_{t-m}),$$

with $m, n$ being some constants depending on the order of the system. Accordingly, the cost function now maps the alternative state representation to its cost value

$$\hat{J}(\hat{x}) : \hat{\mathcal{X}} \mapsto \Re.$$

Clearly, $\hat{J}$ may considerably differ depending on different representations of the system's state $x_t$, but as long as the complete state information is captured, the basic principles remain unaffected.

### G.3 Stochastic systems

To give an overview of the basic principles, the above descriptions were given for the case of the optimal control of a *deterministic* system. It is worth noting however, that the approach can be straightforward extended to stochastic systems, where the system equation becomes

$$x_{t+1} = f(x_t, u_t, w_t).$$

Here, $w_t$ describes some random noise that obeys a certain probability distribution. In the stochastic case, the optimization goal is to minimize the *expected* optimal costs

$$J^*(x_0) := \min_{\pi \in \Pi} \mathcal{E}_w \sum_{t=0}^{\infty} r(x_t, \pi(x_t)).$$

The stochastic version of the value iteration is then given by

$$\forall x \in \mathcal{X} : J_{k+1}(x) := \min_u \mathcal{E}_w \{ r(x,u) + J_k(f(x,u,w)) \}. \tag{16}$$

### H. The basic structure

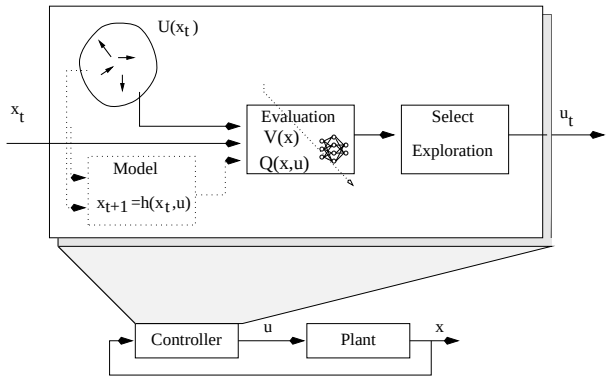


Fig. 4. Structure of the self-learning control architecture

The structure of the basic control architecture reflects the algorithmic considerations as presented in the previous sections (figure 4). The controller receives as an input the current state $x_t$ and computes the currently best control signal $u_t$. The available control decisions are determined by the action set $\mathcal{U}(x)$. To make a decision, the consequences of all actions are evaluated using the estimated cost function represented by a neural network. To compute the costs, two alternative approaches are possible resulting in a model-based or a model-free architecture as discussed in section II-F. In general, the action leading to the minimal expected costs is selected as a control signal, but during the training phase, random deviations from this policy are executed in order to allow a good exploration of interesting regions in state space.

## III. Extending the basic architecture

### A. Generating continuous control signals: the DOE approach

Within the basic framework the control signal $u$ is determined by selecting one alternative control action out of the finite set of admissible actions $\mathcal{U}(x)$ (equation 8). Although this procedure has the advantage of a simple and efficient determination of the control signal, it also poses some problems:

- if no a priori knowledge about the system to be controlled is known, it is difficult to specify a set of appropriate control signals in advance.
- additionally, the set of appropriate actions may depend on the state, which further aggravates the problem of choosing $\mathcal{U}(x)$.

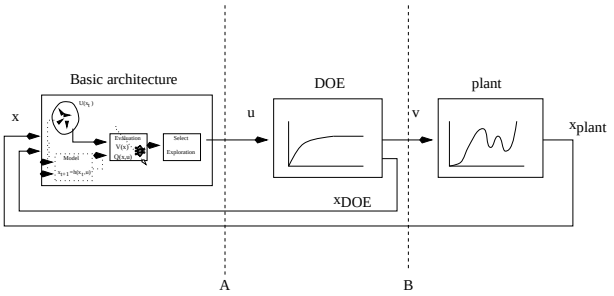


Fig. 5. *DOE* approach: The controller consists of the combination of the basic architecture and a *dynamic output element (DOE)* (line **B**). The control signal $v$ is determined by the output of the *DOE*. From the viewpoint of the basic neural controller, it has to control the combined system of *DOE* and plant (line **A**).

• each additional action in the action set drastically increases the number of possible policies and thus renders the optimization problem considerably more difficult.

The *DOE*-approach proposed in [13] - DOE stands for dynamic output element - aims at dealing with the above problems while preserving the favorable properties of the basic framework. The idea is to extend the basic control architecture by putting an extra dynamic element at the output of the original controller. Therewith, the output of the basic controller - which is the action selected out of $\mathcal{U}(x)$ - is not directly used as a control signal to the plant, but instead used as an input to the DOE.

The DOE itself has dynamic properties that can be arbitrarily specified by the designer. The control signal, that is applied to the plant, is now determined by the *output* value $v$ of the DOE. That means, the control signal is no longer determined by the selected action directly, but moreover by the *sequence* of the selected actions that produces a control signal depending on the dynamic behaviour of the DOE.

Figure 5 shows the schematic working principle of the DOE architecture. When looking at the entire system from the viewpoint of the basic architecture as defined in section II (line A), then the overall system that must be controlled now consists of a composition of DOE and plant. Thus the system state now is extended by the current state of the DOE. Note that besides the change of the dynamical system to be controlled nothing else has happened from the viewpoint of the basic architecture. Thus the basic principles of action selection and learning (as described in section II) remain unchanged.

When looking at the scene from the viewpoint of the controlled plant (line B), the control signal now is generated from the DOE, which thus became part of the controller.

Although DOEs with arbitrary dynamic behaviour may be realized, three basic types are of special interest:

• low pass-filter

$$v(t) + T\,\dot{v}(t) = u(t)$$

Changes in the incoming signal $u$ are transferred with some delay $T$. This DOE may be used to produce control signals with small amplitudes when the amplitude of the input signal oscillates with a high frequency.

• integrating behaviour

$$\dot{v}(t) = u(t)$$

Integrates the incoming signal over time. This DOE may be used to produce a wide range of amplitudes out of three (or more) basic actions (increase, decrease, no change).

• integrating with additional low pass-filter

$$\dot{v}(t) + T\,\ddot{v}(t) = u(t)$$

Integrates the incoming signal over time and smoothes the signal by an additional filter component. This DOE may be used to produce a smooth control signal with varying amplitudes out of three (or more) basic actions (increase, decrease, no change)

The special quality of the DOE approach lies in the fact, that the need for a fine and appropriate set of control signals is not realized by a finer resolution of the action set. Instead, by the use of a DOE the determination of appropriate control signals becomes itself a dynamic control problem - which can elegantly be solved within the basic framework. Thus the controller can *learn* to produce control signals of appropriate amplitude by itself.

## IV. Exploring the basic capabilities

The following experiments show how the self-learning controller behaves on a typical control task. The task is to control the output of a second-order dynamical system starting from arbitrary initial states to a predefined target value. The system behaviour is assumed to be unknown to the self-learning neural controller. We will particularly focus on three important issues: a) requirements of the learning system b) training time and c) quality of the acquired policy. Therefore, the final performance of the learning controller is compared to an analytically derived linear control law - which in contrast assumes the knowledge of the system's equations.

The input to both controllers consists of the current state which for the considered second-order system is

$$x(t) = (y(t), \frac{dy(t)}{dt}).$$

A control signal is applied to the plant every $\triangle_t = 0.05$ seconds. As a design constraint, the control signal should lie between $\pm 1.5$. The task for the controller is to quickly reach the target output value $y^{target} = 0$ and to keep this value as exactly as possible. To compare the performance of the respective controllers we thus measure the average time when the output deviates from $y^{target}$ by more than a tolerated value of 0.01

$$P := \int_0^T c(t)dt$$

where

$$c(t) := \begin{cases} 1 & , \quad if|y(t) - y^{target}| > 0.01 \\ 0 & , \quad else \end{cases}$$

averaged over 100 control trials with random initial conditions

$$\bar{P} = 1/100 \sum_{i=0}^{100} P_i.$$

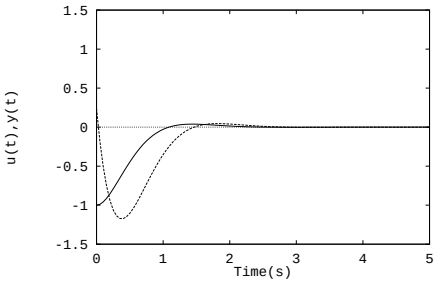


Fig. 6. Plant output (solid line) controlled by the linear controller. The dotted line shows the control signal.

### A. The analytical controller

The analytical controller is designed according to the LQR[4] principle, i.e. we are looking for a linear control law

$$u(t) = -R\,x(t)$$

that minimizes the quadratic temporal cost function

$$J_{LQR} = \int_0^\infty x^T Q x + u^T S u \, dt.$$

Q and S are a positive semidefinite and a positive definite matrix respectively that have to be chosen by the designer, such that the control objectives are fulfilled. Here they were chosen such that on one side the output target is quickly reached and on the other side the control signals stay within the allowed range. The control vector $R$ is derived by the Riccati approach, which assumes the knowledge of the linear system equations.

TABLE I
PERFORMANCE OF THE LINEAR ANALYTIC CONTROLLER

| linear LQR-Controller | | | |
|---|---|---|---|
| | $P$ | trials | realtime |
| linear LQR | 0.773 | - | - |

The performance of the controller with respect to the above defined performance measure is shown in table I. The value $\bar{P} = 0.773$ means that for 100 different control trials with random initial disturbances, in average the output equals the target value after about 0.77 seconds. Figure 6 shows the behaviour of the uncontrolled system and the behaviour of the system output (solid line) when controlled by the linear controller.

[4]linear quadratic regulator

### B. The self-learning neural controller

#### B.1 A model-free controller trained by the fixed horizon -algorithm

The fixed horizon learning framework as proposed in section II-D allows to specify the considered control task directly by the choice of the immediate cost function $r(x, u)$. For it is our goal to reach and permanently keep the target value $y^{\text{target}}$ as fast as possible, we set

$$r(x,u) := \begin{cases} 0 & , \quad if\,|y(t)| < 0.05 \\ 0.002 & , \quad else \end{cases}$$

Note that outside a certain target interval, the immediate cost function takes a constant value of 0.002 independently of the respective state or action. Optimizing the accumulated costs during a control trial then means that we try to reduce the number of time steps outside the target interval and thus will result in a time optimal control behaviour. Other choices of $r$ would for example prefer control signals with lower amplitudes instead of energy consuming high amplitude controls.

The design restriction of the control signals $u$ to lie in a certain range can be easily fulfilled within the proposed self-learning framework by allowing only admissible actions to be applied, i.e. to choose the action set $\mathcal{U}$ accordingly. In the following, we consider three different action sets $\mathcal{U} = \{\pm 1\}$, $\mathcal{U} = \{\pm 1, 0\}$, $\mathcal{U} = \{\pm 1, \pm 0.1, 0\}$.

We assume a total lack of a priori system knowledge meaning that no system model is available to the controller. This requires to represent the estimated cost function as a $Q$-function that considers state/ action - pairs as the input rather than a potential successor state (see section II-F). The neural network that approximates the estimated accumulated costs thus has 3 input neurons - two for the current state and one for the control action. We further use one hidden layer with 10 units and a single output neuron. At the beginning of training, the weights in the network are randomly initialized. When the initialized network is used to determine the control action according to equation 8, this results in an arbitrary initial policy.

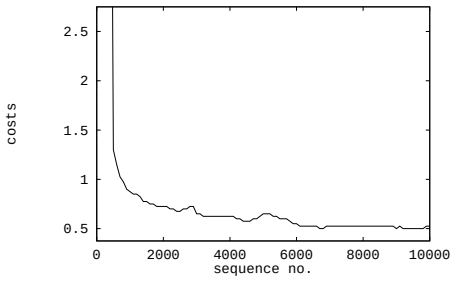


Fig. 7. Decrease of the average costs per control trial (= time to reach the target) of the neural controller with increasing trial number.

Training the controller means the repeated execution of control trials, where the control signal is determined by the use of the current neural cost function $J_k$. The observations are used to update the costs $J_{k+1}$ for the states

observed during control (section II-C). When training with the fixed horizon algorithm, each trial ends after a certain time period, which is set to 4 seconds here. Then a new trial with a new random initial disturbance is started.

In the first experiment, two actions are allowed for control, namely $u = \pm 1$. The learning progress is shown in figure 7. At the beginning, no knowledge is available to the controller, which accordingly fails to control the system resulting in a bad value of the performance measure. After about 600 trials (which corresponds to 2400 seconds in real-time and about 30 seconds, if a computer simulation is used) the control performance is already satisfactory ($\bar{P} = 1.025$) and after 1600 trials the neural controller has about the same performance as the analytical linear controller ($\bar{P} = 0.75$). However, the controller showed its best performance after 9000 trials (=36000 seconds in real time). The obtained performance of $\bar{P} = 0.500$ clearly beats the linear controller - the neural controller is in average 1.5 times as fast as the linear one.

TABLE II

Performance of the model-free neural controller

| fixed horizon neural control | | | |
|---|---|---|---|
| | $P$ | trials | realtime |
| $\mathcal{U} = \{-1, 1\}$ | 0.500 | 9 000 | 36 000 s |
| $\mathcal{U} = \{\pm 1, 0\}$ | 0.582 | 45 000 | 180 000 s |
| $\mathcal{U} = \{\pm 1, \pm 0.1, 0\}$ | 0.575 | 60 000 | 240 000 s |

Table II summarizes the training results when different sets of admissible actions are assumed. Training time considerably increases with the number of available actions. This may be explained by two reasons. First, the complexity of the search for an optimal solution path drastically increases with the number of available actions in each decision step. Secondly, in the model-free case, the action is a direct input to the network and the more values this input can take, the more complicated it becomes for the neural network to find weights to distinguish between them. The results of the model-based controller shown later stress this argumentation. A hint for the problem of the growing complexity of the search space is the fact, that the 3- and 5-action controller are doing very well but do not completely reach the performance of the 2-action controller - although they clearly would have the possibility to do so. Figure 8 shows the control policy (dotted line) and the resulting system output (solid line) of the 3-action neural controller. The controller has learned a sophisticated balance between the available actions to quickly reach the output target while carefully avoiding overshooting.

To summarize, the above experiment with the model-free control architecture trained by the fixed horizon algorithm shows the following

- the self-learning neural controller can learn without any a priori knowledge about the system's equations
- the trained controller achieves a very high control quality
- constraints on the control signals can be easily fulfilled by an appropriate selection of the action set $\mathcal{U}$
- the complete lack of a priori knowledge must be compensated by a considerable amount of training control trials
- a satisfactory policy can often be found relatively early

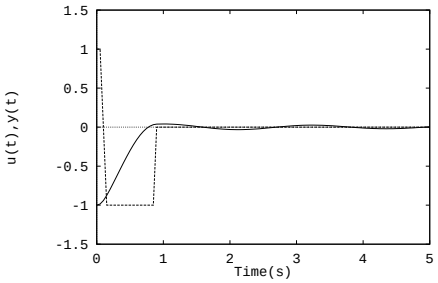


Fig. 8. 3-action neural controller. Output of the system (solid line) and control signal (dotted line).

### B.2 Using a model

As discussed in section II-F if a model of system behaviour is available, that can be used to determine the one-step consequences of an action, then it suffices to learn a mapping from states to their respective accumulated costs. The reason why we might apply learning if a model of the system is already available, is that the model description might be in a form, that is impossible to be tackled analytically. For example, we might use a neural network that was trained by examples to obtain a one-step model of system behaviour, or consider a physical description that contains some severe nonlinearities that make analytical controller design intractable or at least very costly.

In the following, we assume a model that computes the successor state, if a candidate action was applied. This successor state is the input for the neural cost function, which accordingly has only two input units. Beside a minor change in the order of the internal controller computation steps (section II-C), the remaining issues are the same as in the model-free case as discussed above.

TABLE III

Performance of the model-based neural controller

| fixed horizon model-based neural control | | | |
|---|---|---|---|
| | $P$ | trials | realtime |
| $\mathcal{U} = \{-1, 1\}$ | 0.450 | 24 000 | 96 000 s |
| $\mathcal{U} = \{\pm 1, 0\}$ | 0.450 | 27 000 | 108 000 s |
| $\mathcal{U} = \{\pm 1, \pm 0.1, 0\}$ | 0.475 | 25 000 | 100 000 s |

The results are shown in table III. Remarkably, the performance of the model-free controller (table II) could be slightly improved - the use of a model here obviously makes the search for an optimal solution easier. Secondly, the number of trials that are needed to find the final policy are quite independent of the number of admissible actions. For example in the case of five actions, the number of training trials is reduced by more than a half compared to the model-free approach.

To summarize the results of the model-based approach,
- if available, a system model can considerably help to simplify the learning problem
- the model need not be in an analytically tractable form

### B.3 Learning with the event based algorithm

When training the neural controller by the use of the event based algorithm, each control trial ends after a certain terminal state $x \in \mathcal{X}^+$ is reached. As already pointed out in section II-D, the terminal region $\mathcal{X}^+$ has to be carefully chosen, such that an overshooting of the output target in the application case can be avoided. For the following experiment, a control trial is successfully terminated, if both the output is close to its target $|y| < 0.05$ and its temporal change is sufficiently small $|\dot{y}| < 1.0$. The resulting control policy (table IV) shows a good performance. It was learned after 36 000 control trials, which seems rather large compared to the results of the fixed horizon algorithm. Note however, that the average time of a control trial is about 4 times shorter, because it is stopped, whenever $\mathcal{X}^+$ is reached.

TABLE IV

Performance of the event based neural controller

| event based neural control | | | |
|---|---|---|---|
| | $P$ | trials | realtime |
| $\mathcal{X}^+ = \{0.05, 1\}$ | 0.550 | 36 000 | 40 000 s |

To summarize,
- the event based algorithm allows to learn an appropriate control policy
- it needs additional constraints on the target region
- the average length of a control trial becomes shorter as training proceeds

### B.4 State Representation

As discussed in section II-G.2 the information about the current system's state $x_t$ is typically not available. What is available instead, is the information about the history of observed outputs $y(t), y(t-1), \ldots$ and control signals $u(t), u(t-1), \ldots$. In the following experiment we show, that information computed out of past observations is sufficient to learn control of the considered second order system. Again, we assume a model-free architecture trained by the fixed horizon approach as in section IV-B.1 that now receives as an input the vector $\hat{x} := \{y, \triangle y\}$ where

$$\triangle y(t) := \frac{y(t) - y(t-1)}{\triangle_t}.$$

As shown in table V, the controller achieves about the same performance as with the original representation - but now the state information is only computed with the knowledge of past output values.

For higher order dynamical systems one would consider output values that reach longer in the past or provide extra information about previous control signals. To determine the appropriate control input vector it is favorable to know something about the order of the dynamics of the system. If nothing is known, one should rather try to provide more information than actually needed, because in reasonable ranges, redundant information would not harm the learning process but positively influence it. To summarize,

TABLE V

Alternative state representation

| fixed horizon neural control using alternative input | | | |
|---|---|---|---|
| | $P$ | trials | realtime |
| $\hat{x} = (y(t), \triangle y(t))$ | 0.550 | 6 000 | 24 000 s |

- in contrast to analytical state space controllers, no observer is needed to compute an estimation for the current state variables
- the controller can learn with state information computed out of past observed output values and past control signals

### B.5 Nonlinearities and Robustness

The considered second order dynamical system can be controlled with a good quality both by the neural and by the linear controller. In the following we want to show, what happens in the case of an occurrence of a nonlinearity. To demonstrate this, we assume a nonlinear transformation of the control signal (hammerstein-model), which typically appears in real plants due to physical effects and constraints of the plant.

TABLE VI

nonlinearities

| dealing with nonlinearities | | | |
|---|---|---|---|
| | $P$ | trials | realtime |
| neural/ nonlinear | 0.450 | 10 000 | 40 000 s |
| neural/ linear | 0.450 | 0 | 0 s |
| linear LQR | 1.723 | - | - |

The first line in table VI shows the performance of a model-free self-learning controller when trained to control the nonlinear system. Both training time and performance are comparable to the previous results on the linear system. Interestingly, the neural controller that was trained on the linear system, also perfectly manages to control the nonlinear system without any further adaptation (second line in table VI). This demonstrates the impressive robustness of the controller against changes in the system behaviour. However, the analytical LQR controller that was tailored to the linear system drastically loses performance when the unmodelled nonlinearity occurs (third line in table VI).

To summarize,
- the neural controller can deal with nonlinearities without conceptual changes
- it is robust to changes in system behaviour

### B.6 Changing the control interval $\triangle_t$

A common way in controller design is to develop a controller under the assumption that the control signal can continuously vary in time. In practice, this is only true, if analog components are used to implement the control law. If the control algorithm is implemented on a digital system, the control signal is only updated at discrete steps. The time between two succeeding steps, the control interval $\triangle_t$, is preferably chosen to be small enough, that a 'quasi-continuous' control behaviour is achieved. However, in some cases the control interval can not be chosen sufficiently small - for example if the processor is too slow to do the required computations in time. Then, a controller that stabilizes the system in the (quasi-) continuous case, can suddenly show instable behaviour.

For the considered second order system the above effect can be observed, when the control interval is set to $\triangle_t = 0.5s$ - ten times as long as the original control interval $\triangle_t = 0.05$. The first two lines in table VII document the bad performance of both the linear controller and the neural controller trained with a control interval of $\triangle_t = 0.05s$. Both controllers now show instable behaviour. The question now is, if the neural controller can learn an appropriate policy, when it only is allowed to change the control signal every $0.5s$. Table VII shows some interesting effects. Firstly, the neural controller learns to deal with the situation - from an operating point of view, it now observes a system that is different from the previous one, but since it learns to control the system it observes, this makes no conceptual difference. Secondly, its performance significantly improves when it is allowed to apply a larger variety of control signals. Thus the problems that arise with a coarse resolution in time can be compensated to some extent by a finer resolution of the provided control signals.

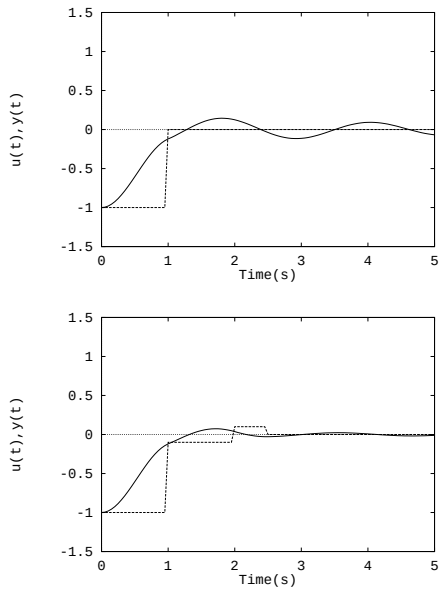


Fig. 9. Learned neural control behaviour when the control interval is set to $\triangle_t = 0.5s$ for the 3-action and the 7-action controller

TABLE VII
ENLARGING THE TIME STEP $\triangle_t$

| $\triangle_t = 0.5s$ | | | |
|---|---|---|---|
| | $P$ | trials | realtime |
| linear LQR | 8.45 | - | - |
| neural/ $\triangle_t = 0.05$ | 7.8 | - | - |
| $\mathcal{U} = \{\pm 1, 0\}$ | 4.370 | 21 000 | 84 000 s |
| $\mathcal{U} = \{\pm 1, \pm 0.1, 0\}$ | 2.070 | 48 000 | 192 000 s |
| $\mathcal{U} = \{\pm 1, \pm 0.5, \pm 0.1, 0\}$ | 1.06 | 58 000 | 232 000 s |

This effect is shown in figure 9. The above diagram shows the behaviour of a 3-action controller. It applies action -1 for two control steps ($1s$) and then selects $u = 0$. This results in a oscillation of the output. The second figure shows the same situation for the 7-action controller. It also selects $u = -1$ for 1 second, but then it additionally applies $u = -0.1$ for 1 second and $u = +0.1$ for another 0.5 seconds before it selects $u = 0$. This strategy suppresses the oscillations far better.

To summarize,

- the neural controller can learn to deal with different control intervals
- a reduced resolution of the control interval can be compensated to some extent by a finer resolution of the action set

## V. Example applications

### A. Control of a chemical reactor

The control of a chemical plant represents a challenging benchmark for nonlinear controller design. The task can be shortly described as follows (for a detailed description see [8]): In a reactor there is a chemical substance with concentration $x_1$. The substance chemically reacts with the fluid in the reactor in an energy-emitting decay process. This leads to a raise of the temperature $x_2$ in the reactor. On the other hand, the temperature $x_2$ influences the rate of decay of the substance. Thus we observe two highly interacting processes of concentration and temperature coupled via a nonlinear function $\gamma(x_1, x_2)$:

$$\begin{aligned} \dot{x_1} &= -a_1\, x_1 + \gamma(x_1, x_2) \\ \dot{x_2} &= -a_{21}\, x_2 + a_{22}\, \gamma(x_1, x_2) + b\, u \\ \gamma(x_1, x_2) &= (1 - x_1)\, k_0\, e^{-\frac{\epsilon}{1+x_2}} \end{aligned} \tag{17}$$

The task is to control the reactor from an initial working point $(x_1, x_2)^{init} = (0.42, 0.01)$ to a new working point $(x_1, x_2)^{target} = (0.75, 0.05)$. The behavior of the process can be influenced by applying external heating or cooling. Only the temperature inside the reactor, the state variable $x_2$, can be measured. Thus the input to the controller consists of the difference between the current temperature $x_2(t)$ and the target temperature $x_2^{target}$:

$$dx_2(t) := x_2(t) - x_2^{target}.$$

### B. Self-learning neural control

Training the neural controller is based on the fixed horizon algorithm (section II-D) which assumes, that the control target is specified in terms of the immediate cost function $r(x,u)$. For the control task is to reach a new target temperature of $x_2^{target} = 0.05$. we set

$$r(x,u) := \begin{cases} 0 & , \text{ if } |x_2(t) - x_2^{target}| < \delta \\ 0.002 & , \text{ else} \end{cases} ,$$

where $\delta = 0.01$ defines the range of the maximum allowed tolerance between output and target value. The minimization of constant immediate costs in case of a larger deviation from the target value leads to the learning of a time optimal control law. We further assume, that there is no detailed a priori knowledge available about the appropriate size of the control signals. Thus we use the DOE extension of the basic neural controller as proposed in section III-A. In particular, a DOE with integrating low-pass filtering behaviour is used. The basic control architecture can choose between three control decisions $\mathcal{U} = \{+0.01, -0.01, 0\}$ which by putting it through the DOE, leads accordingly to an increase or decrease of the resulting control signal. Thus the controller additionally has to learn to produce appropriate amplitudes of the control signals.

Figure 10 shows the temporal behaviour of the temperature of the controlled plant when controlled with three different controllers. Clearly, the linear controller behaves worst, by drastically overshooting the target temperature. The nonlinear controller, that was carefully designed by making use of a detailed knowledge of the system equations [8] shows a drastically improved behaviour compared to the linear controller, justifying the considerable design effort. Interestingly, the neural controller, that has learned the control law by itself, still performs better than the nonlinear controller in terms of a reduced overshooting and by faster reaching the final target value.

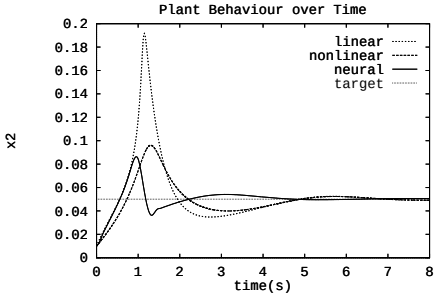


Fig. 10. Control of a nonlinear chemical reactor - comparison of analytical linear and nonlinear controller design and the self-learning neural controller.

As shown in figure 11, the controller has learned to use the facilities provided by the DOE to produce a smooth control signal with a varying amplitude that is tailored to the requirements of the nonlinear process.

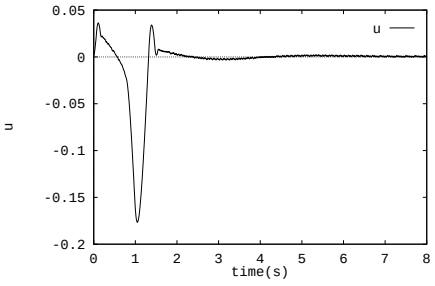


Fig. 11. Control signal produced by the DOE controller out of three basic actions

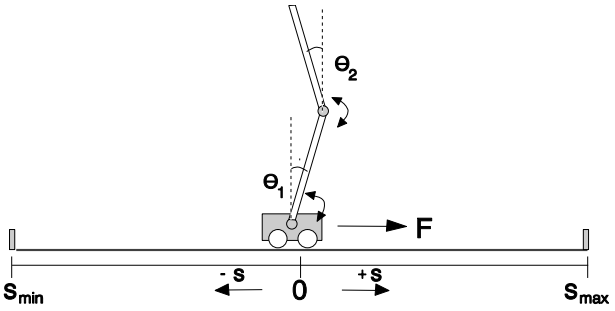


Fig. 12. Double inverted pendulum system

### C. Double inverted pendulum

While for the single inverted pendulum many controllers - analytical and 'intelligent' ones - have been proposed in the past, far less approaches are published for the problem of balancing *two* poles mounted on a cart (figure 12). This may be explained by the fact, that even the qualitative description of the dynamic relationship between the movement of each pole and the movement of the cart is not quite obvious for a human observer. Thus an approach that is based on a deeper human insight into the system behaviour - as required for example by a fuzzy control approach - will run into severe problems.

From a technical point of view, the double inverted pendulum is an instable single input, multiple output system, with the acceleration of the cart being the control input and the position of the cart and the angles of the two poles being the three outputs to be controlled. The control task considered here is to control the system from an initial disturbance to a stable state, where the two poles stand upright ($\theta_1 = \theta_2 = 0$) and additionally, the cart is placed in the middle of the track ($s = 0$). We allow the controller to use three control signals $\mathcal{U} = \{-15N, 15N, 0\}$ corresponding to an acceleration to the left, to the right and no acceleration at all.

#### C.1 Self-learning neural control

For training of the neural controller the fixed horizon algorithm is used. This means, that the given control target is expressed in terms of the immediate cost function $r(x,u)$. Comparable to previous experiments, we set the immediate costs to be zero, if each of the three output values is temporarily close to its target value. In all other

cases, constant immediate costs occur. As discussed earlier, this leads to a time optimal policy. In this case, this means, that the time until *all three* outputs are zero has to be minimized - thus an appropriate control policy has to consider its impact on all three output variables *simultaneously*. For example a modular control strategy that firstly will control the upper angle, then the lower angle and finally will move the cart towards the middle will be very likely to be *not* the optimal policy in this case. As we will see, the neural controller indeed learns a policy where all three outputs reach their target value nearly simultaneously - which obviously seems to be a very well but also a quite sophisticated policy.

As a novelty, here we additionally have to deal with the possibility of a 'crash' of the instable system - namely if the poles have fallen down ($|\theta_{1,2}| > 90^o$) or the cart hits the boundary of the track ($|s| > 1.0m$). In this case, the trial is immediately stopped, and the target value of the neural cost function is set to a maximum value.

The neural network that was used to learn the cost function consists of 6 input, 30 hidden units and a single output unit. Training was started with randomly initialized weights - a priori no knowledge about an appropriate strategy was given. After 49 000 trials the best policy was found.

Figures 13 and 14 give some feeling for the difficulty of the control task by showing the sophisticated policy that the neural controller was able to learn from scratch: First, the cart is shortly accelerated towards the middle, resulting in an increase of the lower angle $\theta_1$. Then after about 0.2 seconds it is immediately accelerated in the opposite direction, until both angles have approximately the same declination towards the middle of the track. Finally, the cart is again accelerated towards the middle position, and therewith eventually reaches its goal of two upright poles and the cart being placed in the middle of the track.

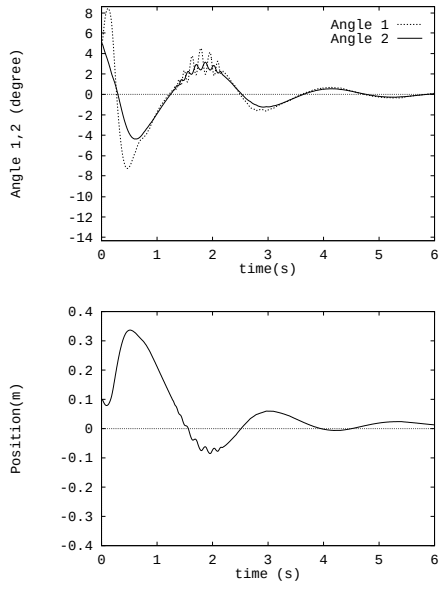


Fig. 13. Neural control of a double inverted pendulum: temporal behaviour of the three system variables $\theta_1, \theta_2$ and position $s$ of the cart.

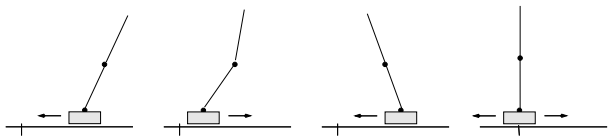

Fig. 14. Illustration of the sophisticated policy that the neural controller learned from scratch

## VI. Conclusions

The article presents concepts for a self-learning neural control architecture that is able to learn high quality control behaviour when only the control target is specified.

In crucial contrast to analytical controller design the control policy is not analytically derived from a system model, but instead is directly learned by interacting with the system itself. No knowledge about the system's equation is assumed a priori. This has three main advantages: Firstly, the expense of controller design is taken away from the human expert and is done automatically by a machine. Secondly, by not depending on a certain mathematical description of the system behaviour, the self-learning approach is applicable to a broad range of systems - as shown in the examples of a nonlinear chemical reactor or the simultaneous control of multi outputs in the case of the double inverted pendulum. Finally, the control policy is directly tailored to the system that is controlled, rather than to an approximate model of it. Therefore the self-learning controller may cope with effects, that are too difficult to model or too difficult to handle analytically.

The assumption of complete lack of a priori knowledge leads to a rather large amount of training experience needed until the controller learned an optimal control policy. Although there are situations, where we might afford long training times, this restricts the general applicability of the approach. Further work in this area is needed to accelerate the underlying learning mechanisms. Also, a combination of conventional techniques and the proposed neural approach might lead to a better treatment of stability issues at least to a certain degree.

Another very promising way is to incorporate a priori knowledge as far as available. This may be both system knowledge, that can be used in a model-based controller, and control knowledge. The integration of a priori control knowledge is the subject of a current research project, called the Fynesse approach - where we combine fuzzy and neural methods in a hybrid architecture.

## References

[1] A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, (72):81–138, 1995.

[2] A. G. Barto and R. H. Crites. Improving elevator performance using reinforcement learning. In M. E. Hasselmo D. S. Touretzky, M. C. Mozer, editor, *Advances in Neural Information Processing Systems 8*. MIT Press, 1996.

[3] A. G. Barto, R. S. Sutton, and C. J. C. H. Watkins. Learning and sequential decision making. Technical Report COINS TR 89-95,

Department of Computer and Information Science, University of Massachusetts, Amherst, September 1989.

[4] R. E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

[5] D. P. Bertsekas. *Dynamic Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[6] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro Dynamic Programming.* Athena Scientific, Belmont, Massachusetts, 1996.

[7] T. Dietterich and W. Zhang. A reinforcement-learning approach to job-shop scheduling. In *Proceedings of the 14.th International Joint Coference on Artificial Intelligence*, 1995.

[8] O. Föllinger. *Nichtlineare Regelungen.* Oldenbourg, 7. edition, 1993.

[9] K. S. Narendra and S. Mukhopadhyay. Intelligent control using neural networks. *IEEE Control Systems Magazine*, 12(5):11–18, April 1992.

[10] M. Riedmiller. Aspects of learning neural control. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, San Antonio, Texas, October 1994.

[11] M. Riedmiller. Application of sequential reinforcement learning to control dynamic systems. In *IEEE Intenational Conference on Neural Networks (ICNN '96)*, Washington, 1996.

[12] M. Riedmiller. Learning to control dynamic systems. In Robert Trappl, editor, *Proceedings of the 13th. European Meeting on Cybernetics and Systems Research - 1996 (EMCSR '96)*, Vienna, 1996.

[13] M. Riedmiller. Generating continuous control signals for reinforcement controllers using dynamic output elements. In *European Symposium on Artificial Neural Networks, ESANN'97*, Bruges, 1997.

[14] M. Riedmiller. Reinforcement learning without an explicit terminal state. In *Proc. of the International Joint Conference on Neural Networks, IJCNN '98*, Anchorage, Alaska, 1998.

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

[16] C. J. Watkins. *Learning from Delayed Rewards.* Phd thesis, Cambridge University, 1989.