published in: Proceeding of the AI 95, 13-18th November 1995, Canberra, Australia

Using Neural Reinforcement Controllers in Robotics

Martin Riedmiller and Barbara Janusz
Institut für Logik,
Komplexität und Deduktionssyteme
University of Karlsruhe, FRG
riedml@ira.uka.de

Abstract

Reinforcement Learning is a promising paradigm for the training of intelligent controllers when only a minimum of a priori knowledge is available. This is especially true for situations, where uncertainties and nonlinearities in sensors and actuators make the application of conventional control design techniques impossible or at least a very hard task. The article describes a neural control architecture based on the asynchronous dynamic programming approach (Barto *et al.*, 1993). Its capabilities are shown on two applications from the domain of robotics.

1 Introduction

Many applications in the robotics domain require a rather expanding and complex controller design, due to nonlinearities and uncertainties in both sensors and actuators. This is one of the main reasons for the development and application of so-called 'intelligent' or self-learning controller architectures in robotics (Riedmiller, 1994).

The following article focuses on a special class of *reinforcement learning*, the so-called *sequential decision making* problem. Based on minimal training information of final success or failure, the controller has to learn an appropriate policy, that transfers an initially unknown system from a current state to a target state. The learning controller is faced with the so-called 'temporal credit assignment problem': Which action in the sequence is responsible to what amount for the final outcome - and in what way has it to be changed in order to get a better performance in the future?

The neural self-learning controller architecture is based on the method of asynchronous dynamic programming (Barto *et al.*, 1993). The capabilities of the controller are shown on two typical sequential decision problems: The control of a robot arm to grasp a rolling ball and the steering of a mobile robot in an arbitrary environment using local sensor information.

2 Self learning control

2.1 Notation

In the following, we refer to the state space notation, where the vector x_t denotes the state of the system, and u_t denotes system input at time t. The system's transition function $f : (x, u) \mapsto x, f(x_t, u_t) = x_{t+1}$ describes the behavior of the system. The task of a controller is to transfer a system from a current state to an externally specified target state within a finite amount of time. This is done in a sequence of control decisions u_t. The mapping $\pi : x_t \mapsto u_t$ that relates the current state to an appropriate action is called the policy of the controller.

2.2 The dynamic programming approach

The search for a controller policy π that transfers a system from a start state x_0 to a specified goal state x_N can be mathematically formulated as the minimization of a temporal cost function $V^\pi(x_0)$ over a set of admissible policies π, where

$$V^\pi(x_0) = \sum_{t=0}^{t=N-1} r(x_t, \pi(x_t)) + r(x_N) \tag{1}$$

Here $r(x_t, \pi(x_t))$ denotes the scalar cost function or the *immediate payoff* for applying a certain control decision in state x_t, and $r(x_N)$ gives the cost for the final state x_N. N is called the horizon of the problem.

The key idea of asynchronous dynamic programming is to decompose the n step optimization problem (1) into an iterative solution of a one-step optimization problem and the solution of the reduced $n-1$-step optimization task. This is done by an iterative improvement of a so-called cost-to-go estimation function $V^\pi(x)$. Under the assumption, that the optimal cost-to-go evaluation $V^{\pi^*}(x_{t+1})$ for all possible successor state x_{t+1} was known, the decision making problem when being in state x_t reduces to a local minimization problem:

$$\pi^*(x_t) = \min_{u_i}(r(x_t, u_i) + V^{\pi^*}(x_{t+1})) \tag{2}$$

Learning to control the temporal behavior of a system in this framework basically means to incrementally improve the current cost-to-go estimation. This is done by updating the current cost-to-go estimation $V^\pi(x_t)$ by

$$V^\pi(x_t) = \min_{u_i}(r(x_t, u_i) + V^\pi(x_{t+1})) \tag{3}$$

Learning the correct cost-to-go estimation is the central task of a learning controller that allows a continuous improvement of the policy in an iterative process (for a detailed discussion see (Barto *et al.*, 1993), (Riedmiller, 1995))

2.3 The controller architecture

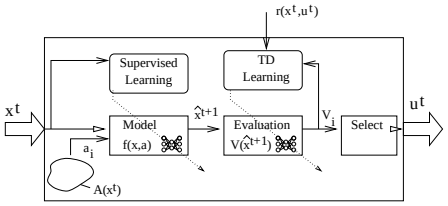


Figure 1: Principle structure of a self learning controller architecture

The structure of the neural network based control architecture is shown in figure 1. The controller receives current state information x_t [1]. Action selection is then done in three steps: First, a candidate action a_i is taken out of a set of available actions $\mathcal{A}(x_t)$. Then the successor state $\hat{x}_{i,t+1}$ of the plant is computed using a model of the plant. Finally, the estimated costs $V^\pi(\hat{x}_{i,t+1})$ of the successor state are computed by the so-called neural evaluation net. After this has been repeated for all possible candidate actions, the action resulting in the state with the minimum costs is chosen by the selection procedure (eq. (2)). This action is applied to the plant. The resulting state x_{t+1} is the input to the controller for the next time step.

Being now in the successor state, its estimated costs $V^\pi(x_{t+1})$ can be computed and the estimation value of the *previous* state, $V^\pi(x_t)$ can be updated according to eq. (3). This is done by updating the weights in the evaluation network using backpropagation combined with Sutton's TD-learning rule (Sutton, 1988).

In the proposed architecture a second neural network is used as a model of the plant to compute the successor state. The network can be trained to imitate input-/output behaviour of the plant using supervised learning techniques (Riedmiller, 1994). An alternative approach is to renounce the model and to learn the consequences of applying action a_t in state x_t directly. This second approach is called Q-Learning (Watkins, 1989) and is applied in the collisison avoiding task of the mobile robot.

3 Learning to grasp a rolling ball

3.1 Task description

The first task for the controller is to learn to move the tool center point (TCP) of a robot arm, such that a ball rolling on a table can be grasped (Thrun *et al.*, 1991). In each time step, the controller can select one out of 16 possible actions, that causes the

[1] state information may be computed out of temporal sensory information

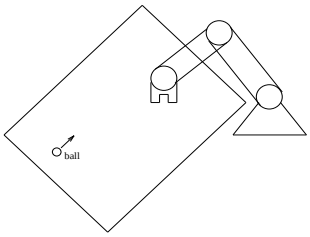
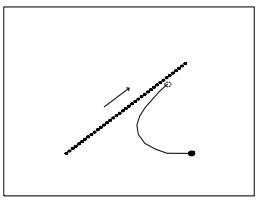


Figure 2: Robot arm grasping a rolling ball



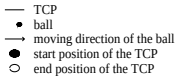


Figure 3: A simple tracking policy will fail, because the ball is faster than the robot arm

TCP to move in the x-y plane. The ball is rolling faster than the TCP can be moved. For this, a policy that simply reduces the current distance between TCP and ball will fail (figure 3). Instead, the controller has to consider the future consequences of the currently selected action. Only an appropriate sequence of actions will finally solve the problem.

In a first attempt, a constant immediate cost function was selected, i.e. $r(x, a) = c, c > 0$. This means, that the controller has to minimize the number of actions (or time steps respectively), until the ball is caught. In case of failure, i.e. when the controller fails to catch the ball at all, the final costs lie above the maximum costs of any positive trial (Riedmiller, 1995). The robot arm is controlled for a maximum number of $N = 40$ time steps. Using 16 different actions, this makes an overall number of 16^{40} possible policies.

3.2 Results

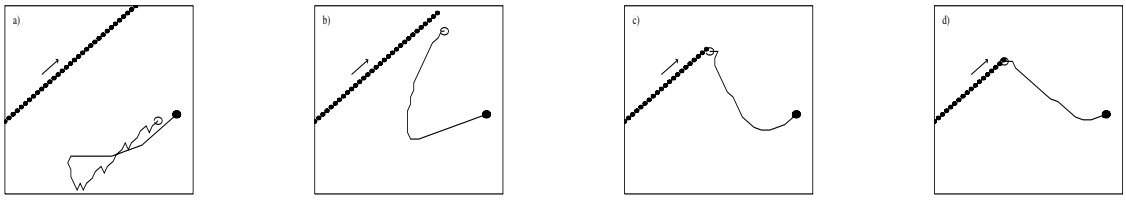


Figure 4: From left to right: Incremental improvement of TCP trajectories while learning proceeds (legend: see figure 3).

Figure 4 shows the development of controller's capabilities. Using an initialized controller, the TCP (white circle) is moved randomly (a). Then, the controller gradually learns to move the TCP in the direction of the ball applying a simple tracking policy, but still fails (b). After 35,000 trials, the controller finally has learned to account for future consequences of its policy. After first moving towards the current ball position, it then moves roughly vertical to the moving direction of the ball, and finally hits the ball (c). In the following 40,000 trials, the controller further optimizes its policy with respect to the number of steps needed to catch the ball (d).

3.3 Influence of the cost function

In the above experiment, a constant immediate cost function $r(x,a) \equiv c$ makes the controller to learn to reduce the number of actions until its task is fulfilled. No care is taken on the *type* of action itself. By an appropriate selection of the immediate cost function $r(x,a)$ a more exacting specification of the control objective is feasible. To demonstrate this important feature, the learning task is extended. Not only the *number* of actions is considered, but also the *system effort* that is needed to perform the action has to be reduced. The choice of $r(x,a)$ is made such that a small change in the moving direction of the TCP causes lower costs than a strong change (this roughly corresponds to the idea of minimizing the energy of a trajectory that reaches the goal).

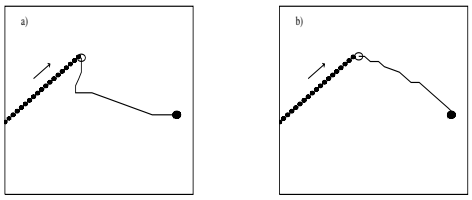


Figure 5: Influence of immediate costs. Both policies use the same number of actions. In (b), additionally the use of high-effort actions is punished.

Figure 5 shows the influence of the choice of the immediate cost function. The controller in (a) has learned to minimize the number of actions. The solution found is to move the TCP on a trajectory with a strong direction change at the end of the run. Although this may not *look* optimal, it definitely is with respect to the available action set and the specified cost function. Optimality in the sense of a smooth trajectory is achieved by the application of an appropriate cost function

$$r(x,a) = \begin{cases} c, & a \text{ does not change the moving direction} \\ 1.5c, & a \text{ causes a slight change in moving direction} \\ 2c, & a \text{ causes a strong change in moving direction.} \end{cases}$$

As shown in figure 5(b), the controller now learns to move the TCP avoiding big changes in the moving direction. The resulting trajectory is much smoother.

4 Controlling a mobile robot

4.1 The learning task

The miniature robot Khepera is a mobile robot moving on two wheels (figure 6). The speed of each wheel can be individually selected within a given range. Eight sensors

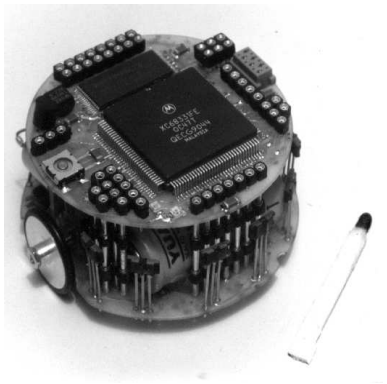


Figure 6: Miniature robot Khepera

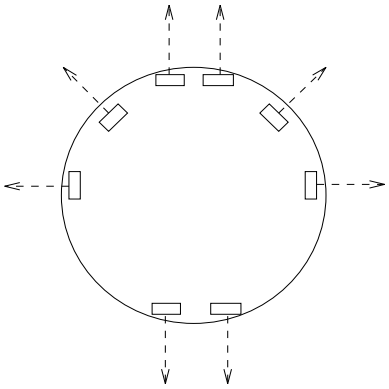


Figure 7: Kheperas sensors and their view direction.

are placed around the robot (figure 7). The sensors can measure the light reflected by obstacles, i.e., the sensor values roughly describe the distance from an obstacle.

The aim of the learning process is to control the robot in an unknown environment with arbitrary obstacles. Using its infrared sensors, the robot has only a local view of its environment. The task of the controller is to select the velocity of the wheels in order to avoid immediate or future collision, based on the *current* sensor information. Thus it is faced with a typical sequential decision making problem: A current situation detected by the sensors may not be dangerous at the moment, but may require some reasonable control of the wheels in order to avoid future trouble.

Training is done in simulation with both an idealized and a realistic model of the robot. The trained controller is then tested in both simulation and by controlling the real robot in an unknown environment.

4.2 Training the controller

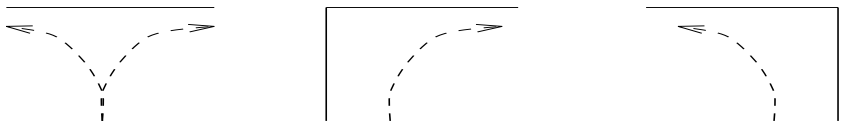


Figure 8: Three characteristic types of obstacles

In the training phase three characteristic types of obstacles are presented in random alteration - curve to the right, curve to the left and obstacle in front (figure 8). A run is terminated if the robot either had a collision with the obstacle (negative termination) or succeeded to avoid the obstacle (positive termination) or the maximum number of time steps $N = 48$ is exceeded. At every time step, the controller can choose between five different actions - fast/slow move to the left/right and forward move. This gives an overall number of 5^{48} possible policies.

Results Figure 9 shows the evolution of the controller's capabilities after 500, 15,000 and 50,000 training runs respectively for five different initial positions. After 500 runs, the controller failed to succeed at any of the three obstacles. After 15,000 runs, the curve-to-the-left obstacle is managed for all initial positions, whereas the controller still has problems to manage the other two types of obstacles. Finally, after 50,000 runs, all three types of obstacles are perfectly managed.

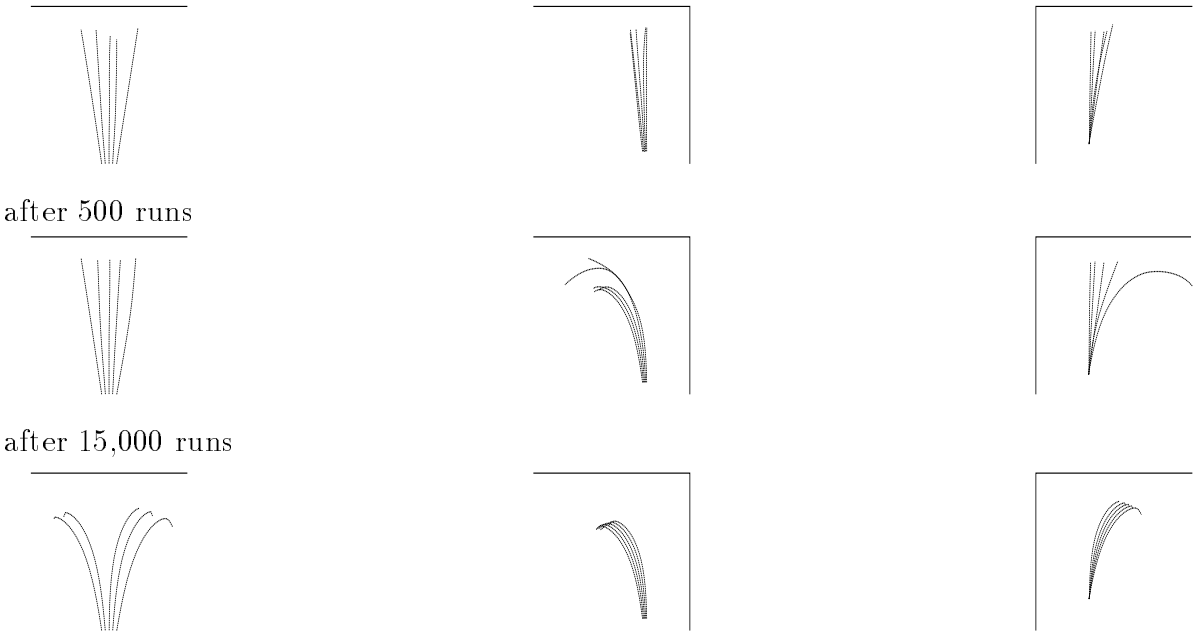


Figure 9: Development of a policy. For different start positions, the trajectories found after 500, 15000 and 50000 training runs are shown.

4.3 Evaluating the performance

To test the acquired capabilities, the controller has to control the robot in an unknown environment with several arbitrarily shaped obstacles. The overall policy consists of two main parts: If none of the sensors recognizes an obstacle, the robot moves forward. If at least one of the sensors recognizes an obstacle, the strategy is switched to the previously trained obstacle avoidance policy.

In a random environment (figure 10), the controller reached a final performance of 95%. That means, that in 95% of all situations, where a collision may occur, the controller achieved to avoid the collision. The remaining situations where collision still occurs are due to the insufficient resolution of the sensors.

When applying the trained controller to the real robot, it still has a performance of 90% in an arbitrary environment. This performance may be further improved by learning on-line while the real robot moves around.

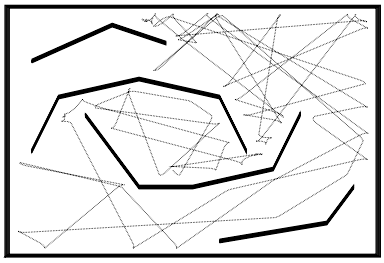


Figure 10: Testing the controller in an unknown environment. The figure shows the trajectory of the robot (thin line) moving around the obstacles (thick lines).

5 Conclusions

The theory of asynchronous dynamic programming provides a powerful framework for a wide variety of control tasks, where temporal aspects play a central role. A neural self-learning controller, based on the dynamic programming approach, was shown to be able to solve typical robotics tasks. It was demonstrated, how the choice of the immediate cost-function influences the quality of the learned trajectories. As shown in the mobile robot example, the controller is able to learn to deal with non-linear multi-sensor input. To overcome the problem with long training times, that becomes crucial when learning control of real plants, two strategies seem promising. One is to pre-train the controller on an accurate simulation model as demonstrated on the mobile robot task, the other is to investigate methods for the integration of available a priori knowledge.

References

Barto, A. G., S. J. Bradtke, and S. P. Singh (1993) Learning to act using real-time dynamic programming. *submitted to: AI Journal special issue on Computational Theories' of Interaction and Agency.*

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

Riedmiller, Martin (1995) Application of sequential reinforcement learning to control dynamic systems. Paris. submitted to ICANN'95.

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

Thrun, S., K. Möller, and A. Linden (1991) Planning with an adaptive world model. In Touretzky, D. S. and R. Lippmann, editors, *Advances in Neural Information Processing Systems 3.* Morgan Kaufmann.

Watkins, C. J. (1989) *Learning from Delayed Rewards.* Phd thesis, Cambridge University.