Machine Learning, 23, 221–249 (1996)
© 1996 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Learning Controllers for Industrial Robots

C. BAROGLIO baroglio@di.unito.it
Dipartimento di Informatica, Università di Torino, C.so Svizzera 185, 10149 Torino, Italy

A. GIORDANA attilio@di.unito.it
Dipartimento di Informatica, Università di Torino, C.so Svizzera 185, 10149 Torino, Italy

M. KAISER kaiser@ira.uka.de
University of Karlsruhe, Institute for Real-Time Computer Systems & Robotics

M. NUTTIN nuttin@mech.kuleuven.ac.be
Katholieke Universiteit Leuven, Department of Mechanical Engineering, Division PMA

R. PIOLA piola@di.unito.it
Dipartimento di Informatica, Università di Torino, C.so Svizzera 185, 10149 Torino, Italy

Editors: Judy A. Franklin, Tom M. Mitchell, and Sebastian Thrun

Abstract. One of the most significant cost factors in robotics applications is the design and development of real-time robot control software. Control theory helps when linear controllers have to be developed, but it doesn't sufficiently support the generation of non-linear controllers, although in many cases (such as in compliance control), nonlinear control is essential for achieving high performance. This paper discusses how Machine Learning has been applied to the design of (non-)linear controllers. Several alternative function approximators, including Multilayer Perceptrons (MLP), Radial Basis Function Networks (RBFNs), and Fuzzy Controllers are analyzed and compared, leading to the definition of two major families: Open Field Function Approximators and Locally Receptive Field Function Approximators. It is shown that RBFNs and Fuzzy Controllers bear strong similarities, and that both have a symbolic interpretation. This characteristic allows for applying both symbolic and statistic learning algorithms to synthesize the network layout from a set of examples and, possibly, some background knowledge. Three integrated learning algorithms, two of which are original, are described and evaluated on experimental test cases. The first test case is provided by a robot KUKA IR-361 engaged into the "peg-into-hole" task, whereas the second is represented by a classical prediction task on the Mackey-Glass time series. From the experimental comparison, it appears that both Fuzzy Controllers and RBFNs synthesised from examples are excellent approximators, and that, in practice, they can be even more accurate than MLPs.

Keywords: Robotics, Neural Networks, Fuzzy Controllers, Multistrategy Learning

1. Introduction

Traditionally, robotics has been a great challenge for artificial intelligence and machine learning, since it offers the opportunity to design truly intelligent machines that interact with their environment and exhibit human-like capabilities. Nowadays, the increasing availability of sophisticated sensor systems for both manipulation and mobile robots makes the development of smarter and more adaptive robots also technically possible. Such high-performance systems are opening new markets to the robot technology, e. g. the recently emerging market for intelligent service systems. However, the cost of programming such robots is high, and learning capabilities, i.e. the ability of the robot to gain profit from its experiences, become indispensable for fully exploiting the robot's potential autonomy (Kaiser et al., 1995b).

In this paper, we describe the results of an investigation done in the framework of the Esprit project No. 7274, B-LEARN II. The fundamental goal of the project is the enhancement of current industrial robots by incorporating learning capabilities on all levels of robot control. The expected benefits are a decrease in the cost for developing the control software itself and an increase in robots' reliability and flexibility. B-LEARN II is a wide-spectrum project, structured into several subprojects related to different applicative fields (Kaiser et al., 1994). In the research presented here, we focus on controllers for robots employed in assembly lines, performing tasks (such as cutting and deburring), in which the same operation is repeated thousands of times, and information coming from rough sensors only, such as force and torque sensors, is exploited. In this kind of application, it is not interesting to optimize any problem solving capability (as one might think reading about the application of "AI techniques"), but to achieve high accuracy as well as high speed in accomplishing simple operations such as following a contour.

In the traditional approach, such robots are usually controlled by means of linear controllers. However, the simplifications involved in linear controller design are not always justified. Especially the physical characteristics of the robot and the existing hardware and software limitations that must be taken into account in the real world introduce nonlinearities that are difficult to model.

The aim of this work is to assess the usability of ML techniques to increase the robot's performance by means of nonlinear control, to ease the development of these controllers, and to decrease the cost for realizing a new application. Learning to control robots on a low level means learning continuous, real-valued functions: it is a regression problem. In the literature, regression has been tackled by means of different techniques such as neural networks (Rumelhart & McClelland, 1986; Rumelhart et al., 1985), prediction trees, statistics (Cramer, 1974; Quinlan, 1993), regression trees (Breiman et al., 1984), and fuzzy logics (Berenji, 1992); moreover, it has been seen either as a supervised or as a reinforcement learning task (Barto et al., 1983; Gullapalli, 1990).

The aim of this paper is to perform a comparative analysis of the different methods in order to find out both the function approximators and the learning algorithms that are suitable for the class of robotic applications described above. The major result of this research is the characterization of a class of approximators (including Radial Basis Function Networks (Poggio & Girosi, 1990) and Fuzzy Controllers (Berenji, 1992)), that we will call Locally Receptive Field Networks (LRFN), which exhibits properties that are more interesting than those of the well-known multilayer perceptron. More specifically, LRFNs, beyond being excellent approximators, can be interpreted in terms of symbolic knowledge and can be synthesised by means of standard symbolic and/or statistic learning algorithms. Furthermore, LRFNs allow for incremental learning, which is an important issue especially when realizing adaptive controllers (Miller et al., 1990).

An important novelty is the multistrategy method we propose for learning LRFNs from examples of correct behavior and background knowledge. The emphasis will be given to supervised learning, which can also be a basis for reinforcement learning architectures, as shown by experiments based on Williams REINFORCE algorithms (Williams, 1992; Nuttin et al., 1995).

The paper is organized as follows. The next section investigates the requirements posed by the robotic applications we are interested in. Section 3 performs a comparative analysis of the main methods available for regression, and tries to set a unifying view. Section 4 describes the supervised learning methods, and Section 5 presents an experimental evaluation of both the approximators and the corresponding learning techniques. Finally, the results are discussed in Section 6 and the conclusions are taken in Section 7.

2. Learning Compliance: a Key Issue in Assembly Robots

In this section we will investigate the nature of the control activity in a robot oriented to industrial production in an assembly line. As it has been introduced above, we assume them to be devoted to repetitive tasks, guided by rough sensors[1].

Typically, the program controlling a robot employed in an industrial assembly line can be considered as organized in two layers. The highest one establishes the basic cycle executed by the robot. This layer can be realized by a traditional robot program, but it can as well be given as a *dispatcher* acting as a plan execution unit. The lowest level closes the loop between sensing and action and takes care of the compliant behavior of the robot. To this aim, data originating from force and torque sensors are employed.

In general, *compliant motion* refers to tasks that require the robot to establish or maintain contact with its environment such that the capability of continuously adapting the robot's action to its perception is needed (Mason, 1981). As an example, we can think of executing the insertion of a peg into a hole. The quality of this action can be evaluated as a trade-off between the global time required for achieving the goal state *peg-inserted*, and the strength of the forces which should remain as low as possible during the whole insertion. The solutions for dealing with this problem in the presence of uncertainty are the use of intelligently designed passive compliance by means of elastic joints as well the realization of an active compliance mechanism, for instance by means of conventional PID-controllers[2].

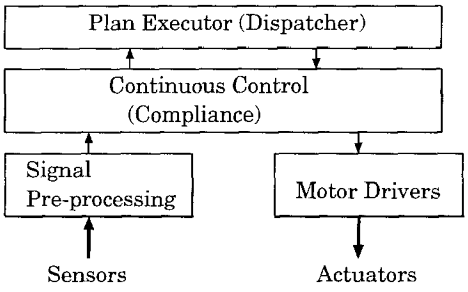

Figure 1. Two layer control architecture for an assembly robot.

Such a controller tries to keep the system in a nominal state; if the state changes, the controller reacts with a signal proportional to the difference between the nominal state and the actual one. In our case, for instance, a linear controller could be used to keep the contact force between the surfaces constant. Unfortunately, a reaction proportional to the error is not always the best solution to restore the nominal state as quickly as possible. For the peg-into-hole task, it has even been proven that the optimal controller must be non linear (Asada, 1990).

However, while control theory offers a good basis for developing linear controllers, it does not support the design of non-linear controllers sufficiently. Therefore, empirical tools such as Fuzzy Controllers (Berenji, 1992), that allow non-linear control functions to be realized while avoiding the burden of developing a mathematical model, are attracting a lot of attention. In this context, we claim that ML can be of substantial help for both the reduction of the development costs of an empirical solution as well as for the improvement of the implementation quality.

When and how can ML techniques be applied to this aim? A possible answer can be obtained by examining the knowledge sources available for an empirical approach to control synthesis. A first knowledge source can be a qualitative theory about the control problem. In fact, although it is difficult, or even impossible to derive an analytical control function from a quantitative theory, it is easy to find some expert having a qualitative idea of its general shape. In general, this knowledge is the starting point for manual fuzzy controller synthesis (Bonissone & Chiang, 1993; Nuttin et al., 1994).

A second source is represented by the ability of a human operator (or of some other machine) of executing the control task him/herself. It is extremely hard to explicitly formalize this knowledge as a theory; nevertheless, it is easy to apply it to generate examples of control, and even from bad examples, control knowledge can be obtained (Kaiser et al., 1995a).

Considering this second knowledge source, we immediately recognize the typical supervised learning framework, that can be applied for regression as well as for classification problems. We would like, however, to stress the importance of exploiting qualitative theories too. As a matter of fact, in many cases, it is difficult to have a set of examples of control actions covering all possible cases (a necessary condition for learning a complete control function). In this case, using a domain theory can help to reduce the amount of training examples significantly. The success of the manual synthesis (Bonissone & Chiang, 1993; Yih & Shieh, 1992) is an implicit proof of this claim.

However, learning only from examples provided by a teacher has an intrinsic limitation: it allows the synthesis of controllers performing *at most* as well as the teacher him/herself. Actually, taking an optimistical view (filtering the noise from the examples and using a good domain theory), we might even achieve a performance better than that of the teacher, but this does not mean to overcome the inherent limitation. The only way to get rid of this problem is to widen the learning framework by means of an integrated approach (Kaiser & Kreuziger, 1994), using also other learning paradigms (such as reinforcement learning) that are capable of continuously improving the controller's performance. In this respect, an important feature of the learning algorithms and the representations that are to be used is their support of *incremental* learning, to allow an expert to supply his/her knowledge progressively as well as to allow for on-line refinement of the control function.

In the following sections, we will analyse and compare several tools for approximating continuous functions. We will investigate which of them meet the requirements mentioned above and are therefore most suitable. We'll also identify those that allow for incremental learning and can use domain knowledge and examples. For two major reasons, the emphasis will be on supervised learning algorithms. Firstly, many reinforcement learning techniques do actually rely on supervised learning algorithms, such as backpropagation. Secondly, supervised learning is a good means for initializing a robot's knowledge off-line. The robots we are dealing with are delicate and potentially harmful. Therefore, we would like a robot controller to be at least partially operational from the very beginning. A reinforcement learning approach based on random exploration without any knowledge about the environment is simply not feasible: the robot will damage itself (or it will cause too many troubles) before learning anything. Therefore, the only viable solution is to off-line generate a controller that already fulfills minimal operationality requirements before beginning to learn by reinforcement.

3. Approximating Control Functions

The task of learning to approximate continuous functions has been investigated in many fields, using alternative approaches such as statistics (Breiman et al., 1984; Specht, 1988; Specht, 1990), connectionism (Rumelhart & McClelland, 1986; Barto et al., 1990), fuzzy logics (Zadeh, 1992; Berenji, 1992), and symbolic machine learning (Sammut et al., 1992).

In applicative domains such as robotics and automated controls, two approximators attracted a great attention: Multi-Layer Perceptrons (MLPs) and Fuzzy Controllers (FCs). The reason for their popularity is mostly due to their robustness and inherent simplicity, which allows applications to be quickly developed without requiring specific skills from the user. Furthermore, low cost hardware supporting real time computation is available for both architecture types.

Even if MLPs and/or FCs seem to be an obvious choice in a project with an industrial perspective, we are interested in examining the problem of control function approximation from a more general viewpoint. Therefore, many different architectures have been analysed and compared. Apart from regression and prediction trees, all existing approximators (independent on the approach they originate from) can be represented by some multi-layered network, in which nodes (or neurons) computing a simple function (activation function) are connected by means of weighted links. The differences among the various kinds of approximators reside in the activation functions and in the methods used to build the network and to update the weights (the learning algorithms).

The activation functions can be grouped into two major families: *Open Field Functions* (OFFs) and *Locally Receptive Field Functions* (LRFFs). The OFFs exhibit an anti-symmetry property with respect to a hyperplane splitting the input domain into two semi-spaces, whereas LRFFs have radial symmetry. A good example of the first family is the well-known sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$, widely used in MLPs and in all their descendants. Examples of the second family are the multidimensional Gaussian functions used in Radial Basis Function Networks (RBFNs) (Poggio & Girosi, 1990) or in Statistical Neural Networks (SNN) (Specht, 1988; Specht, 1990). The choice of an activation function belonging to one or the other family entails properties fundamentally different.

As it has been proven in (Hornik et al., 1989), a universal function approximator can be constructed using only three layers of nodes (input, hidden and output), if the activation function in the hidden layer is non-linear. In Figure 2 we compare the structure of two networks of this kind: one based on sigmoidal and the other on bell-shaped functions. From a topological viewpoint, they look identical. Nevertheless, the way they encode the target function is very different.

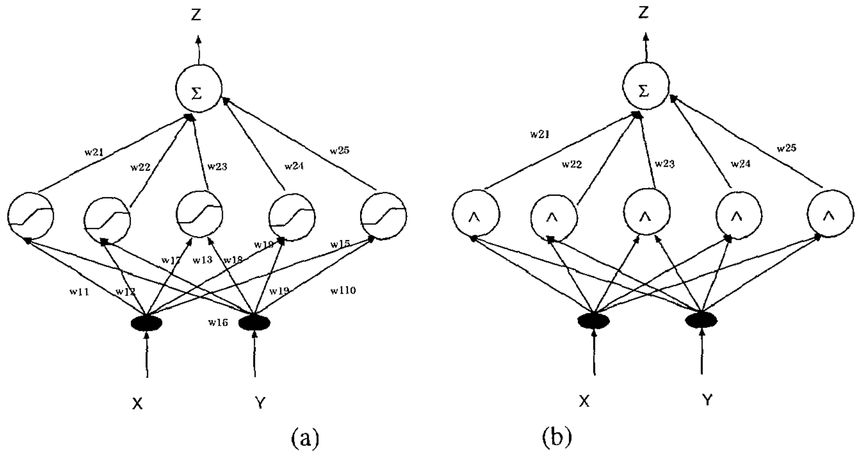

Figure 2. Multilayer perceptron (a) and Radial Basis Function Network (b) architecture. The fundamental difference is in the activation function in the hidden layer neurons. In both cases the output neuron performs the sum $\sum_{i=1}^{N} w_i o_i$.

Considering the activation function $O_i = \sigma(\sum_{j=1}^{N} w_{ij} I_j)$ of the MLP's hidden units, we see that each neuron (apart from a narrow region where the sigmoid transient occurs) splits the input space into two semi-spaces where its output is significantly close to 1 or to 0. Only the semispace where the output is close to 1 contributes to the value of the target function.

In case of the LRFN, each neuron is associated to a closed region in the input space (hyper-ellipse), where its response is significantly greater than zero, and dominates over that of every other node. The contribution to the approximation of the target function comes essentially from this region.

In order to understand the different behaviors of the two network types, suppose to modify a weight between two nodes in the MLP. The effect affects an *infinite* region of the input space and can also affect a large part of the co-domain of the target function. On the contrary, changing the amplitude or shifting the position of the activation function in a LRFN will have an effect *local* to the corresponding region. This *locality property* of the LRFN allows for the network layout to be incrementally constructed (e.g. (del R. Millán, 1994)) by adjusting existing neurons and/or adding new ones. As every change has a local effect, the knowledge encoded in the other parts of the network is not lost; so, it will not be necessary to go through a global revision process.

Another property of LRFNs that is related to locality is the tendency to *undergeneralize.* Depending on the way the hidden layer (the layout) is constructed, it is possible to have large regions of the input space that are not covered by any neuron. If the current input situation falls in such regions, the output of the network will constantly remain close to the null value (usually zero), because no neuron will fire. On the contrary, every neuron of a MLP covers an infinite region of the input space, so the network will usually be active in any situation. A trained MLP has, therefore, the tendency to overgeneralize. Depending on the application (e.g., if it is safety-critical or not), under(over)generalization can be either an advantage or a disadvantage.

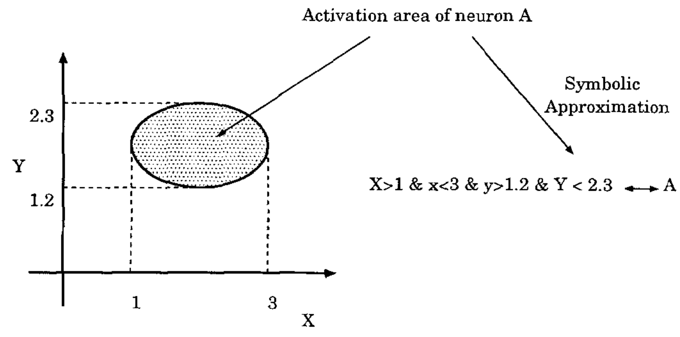

Figure 3. The closed region where the LRF-Function is dominant can be roughly approximated by a conjunctive logical assertion.

Finally, an important property of LRFNs (as opposed to OFFNs, which, with few exceptions (Towell et al., 1990), are black-boxes w.r.t. their interpretation) is the possibility to directly obtain a symbolic interpretation of the hidden neurons. In fact, a closed region in the input space can be approximated using a hyper-rectangle which, in turn, can be described by a propositional formula, as shown in Figure 3. This property has already been exploited for *manually* encoding fuzzy controller's layouts using qualitative domain knowledge (Bonissone & Chiang, 1993). We will show how symbolic learning algorithms can be used to *automatically* build excellent layouts for LRFNs.

In the following, four specific network architectures will be introduced: Time-Delay Neural Networks, Radial Basis Function Networks, Time-Delay Radial Basis Function Networks, and a Fuzzy Controller architecture belonging to the LRFN family. It is worth noticing that the choice of these four types originates from an earlier, much wider, explorative investigation.

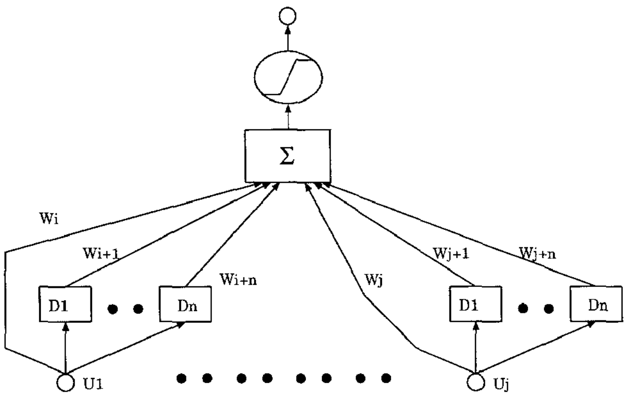

Figure 4. Basic structure of a TDNN elementary unit.

3.1. Time-Delay Neural Networks

The Time-Delay Neural Networks (TDNNs) have been introduced by A. Waibel (Waibel et al., 1989) to handle time variant signals, and are basically an extension of the multilayer perceptron. When coping with time variant signals, it is important to consider the short term history in order to predict the future values. The idea followed in TDNNs is based on the invention of the *time delays,* i.e., of a *First-In First-Out* memory, that enable a neuron to store the history of its input signals (Fig. 4).

Each delayed input is associated to a weight that can be trained using the generalized delta rule in combination with the backpropagation algorithm. Thus a neuron can detect relationships between the current and the former input values, i.e., the the network can adapt to *spatio-temporal patterns.* Delaying neurons' input in a hidden or output layer is similar to multiplying the layers in a multilayer perceptron. This may help in dealing with pattern scaling and translation. Of course, delaying also results in a higher network complexity.

3.2. Radial Basis Function Networks

RBFNs (Fig. 2b) are a family of LRF-Networks architecture that has already been investigated by several authors and recently received a strong mathematical foundation thanks to Poggio and Girosi (Poggio & Girosi, 1990).

Principally, a RBFN consists of an input layer holding the current input values, a hidden layer representing a number of closed regions (often called *kernels*) in the input space, and an output layer integrating the signals calculated by the hidden layer units. This calculation is performed on the basis of two parameters specific of each hidden unit: center and width.

The hidden neuron associated to a particular area returns the highest output value if the inputs it receives are close to its center. The actual output of a hidden neuron is determined by its activation function (mostly a multidimensional Gaussian function or a function representing a hypersphere or a hypercube), applied to a distance measure, usually computed by means of the Euclidean rule, i.e., for any input vector $\vec{x}$, a RBFN computes an output y, by means of equation

$$y = f(\vec{x}) = s((\sum_{i=1}^{n} w_i r_i(\vec{x})) + w_0)$$

where s denotes the activation function of the output neuron (which is often the identity), the w_i's are the weights on the connections between the hidden layer and the output neuron, w_0 is a bias and $r_i(\vec{x})$ is the activation function for hidden neuron i. Usually,

$$r_i(\vec{x}) = e^{-(\frac{||\vec{x}-\vec{\mu_i}||}{\sigma_i})^2}.$$

Here, $\vec{\mu_i}$ is the center and σ_i the width of neuron i, the transfer function r_i is a multidimensional Gaussian. Alternatively, a multidimensional Gaussian can be replaced by the product of N unidimensional Gaussians

$$r_i(\vec{x}) = e^{-(\sum_{j=1}^{dim(\vec{x})}(\frac{x_j-\mu_{ij}}{\sigma_{ij}})^2)},$$

where $dim(\vec{x})$ is the dimension of the input space. It has been proven that these networks are capable of universal approximation on $\Re$ or on compact subsets of $\Re$ (Park & Sandberg, 1993), given suitable centers μ_i.

3.3. *Time-Delays in Radial Basis Function Networks*

For handling time-variant inputs, *time delays* can also be introduced in RBFNs. Such an extension has been proposed by Berthold (Berthold, 1994).

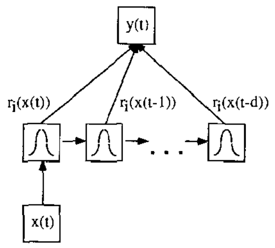

Figure 5. Structure of a Radial Basis Function Network with time delays in the hidden layer.

Figure 5 shows the structure of a RBFN network with d time-delays in the hidden (cluster) layer. The output neuron calculates its output y by adding current membership values and their *history* over the last d time steps, such that

$$y(t) = f(\vec{x}(t)) = s(\sum_{i=1}^{n}\sum_{j=0}^{d} w_{i,j} r_i(\vec{x}(t-j)))$$

In addition, it is also possible to delay the actual input vector $\vec{x}(t)$, i.e., to give each neuron access to $\vec{x}(t), \vec{x}(t-1), \ldots, \vec{x}(t-d)$ (Berthold, 1994). However, this increases the dimension of the input space from $dim(\vec{x})$ to $(d+1) \times dim(\vec{x})$ and, consequently, results in a much higher number of connections that are to be considered. Also, a higher number of examples is necessary to sufficiently cover the input space. Since the application investigated here is subject to real-time constraints, this option was not considered for the experiments described in Section 5.

3.4. A Fuzzy Controller Architecture

Fuzzy controllers are universal function approximators developed within the fuzzy logic community (Zadeh, 1992; Berenji, 1992). Even though they originate from a very different approach, they bear strong similarities to RBFNs and share the same learning algorithms. As a matter of fact, Fuzzy Controllers do not correspond to a single architecture but to a wide family that contains the topology of Figure 2b as a special case. Nevertheless, in our research, we focused on a specific topology, derived from the one proposed by Berenji (Berenji, 1992). Figure 6 shows the corresponding network representation.

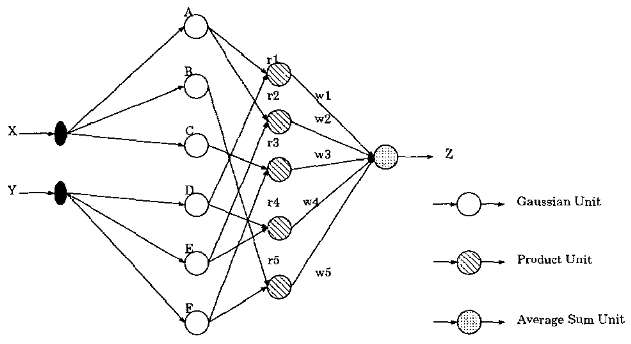

Figure 6. Network corresponding to the fuzzy controller used in current experiments. Gaussian units have a unidimensional Gaussian activation function. Product units compose the input values using arithmetic product or *softmin*. An average sum unit performs the weighted sum of the activation values received from the product units.

Differently than RBFNs, the fuzzy controller is organized as a three-layered network plus an input layer. The first hidden layer neurons are feature detectors with a unidimensional Gaussian activation field, which receives only one input signal. They compute the function

$$r_{ij} = e^{-\left(\frac{x_j - \mu_{ij}}{\sigma_{ij}}\right)^2} \tag{1}$$

where r_{ij} is the value of membership of the j-th component x_j of input $\vec{x}$ to a bell-shaped fuzzy set of center μ_{ij} and width σ_{ij}.

The neurons in the second hidden layer receive the membership values computed by two (or more) neurons of the previous layer, which are required to process different input signals. The output is obtained using the *softmin* composition rule

$$r_i = \frac{\sum_{j=1}^{dim(\vec{x})} r_{ij} e^{-kr_{ij}}}{\sum_{j=1}^{dim(\vec{x})} e^{-kr_{ij}}} \tag{2}$$

that tends to the regular min (the function usually employed for fuzzy AND) for $k \rightarrow \infty$, but is differentiable. Finally, the output neuron performs the linear combination

$$y = \frac{\sum_i w_i r_i}{\sum_i r_i} \tag{3}$$

of the weights w_i and the activation values r_i of the corresponding second layer neurons.

Comparing this structure to an RBFN, two differences can be noticed: first, the locally receptive field function is built as a combination of unidimensional functions using *softmin* (2) instead of product. This function is similar to a pyramid with smoothed corners and allows for a more precise symbolic interpretation. The second difference is the use of normalization (3) in the output unit. The consequence is an improved capability of generalization, so that good approximations can often be achieved using a smaller number of neurons than in a corresponding RBFN.

In the fuzzy set literature, the following symbolic interpretation is given to this architecture. The first hidden layer is interpreted as a mapping from continuous input features to linguistic variables (*Fuzzy Sets*) and is called *Fuzzifier*. The second hidden layer is seen as a set of implication rules in conjunctive form defining a logical mapping from the set of linguistic variables to a discretized set of output values. For instance, by referring to Figure 6, neuron r1 can be represented as a rule $A \cap D \rightarrow W_1$. Finally, the last layer is seen as a mapping from a discrete set of values to a continuous space and is called *Defuzzifier*.

4. Supervised Learning Algorithms for TDNN, RBFN and Fuzzy Controllers

Both in OFFNs and in LRFNs, layout construction and weight calculation are usually performed in separate steps. OFFNs' layout is usually defined in an empirical way by using rules of thumb to select both the number of neurons and the connnecting topology; algorithms such as KBANN (Towell et al., 1990) or Cascade-Correlation (Fahlmann & Lebiere, 1989) are designed for specific applications or architectures and do not help in the general case. On the contrary, several algorithms are available for automating LRFNs layout construction. The reason for this difference is to be sought in the *locality property* characterizing LRFNs; in fact, the hidden layer can be constructed by subdividing the input space into a mosaic of closed regions, allowing the approximation of the target function by means of a histogram. In RBFNs literature, this task is usually accomplished using clustering algorithms such as k-means or variants of it (Moody & Darken, 1989; Wilpon & Rabiner, 1985). Afterwards, the weights on the links to the output level can be learned either by performing a gradient descent of the quadratic error or using algorithms based on pseudo-inverse matrix transformation. Gradient descent can be quite slow but it can easily be adapted in order to be done incrementally on-line. On the contrary, the pseudo-inverse matrix transformation is fast but not incremental. Approaches to construct also the network layout in an incremental manner are, for instance, GAL (Alpaydin, 1991) and Fritzke's Growing Cell Structures (Fritzke, 1993).

The most common approach to design a Fuzzy Controller is to setup the network layout manually, relying on the domain knowledge of a human expert (Nuttin et al., 1994). Variants of the error gradient descent have been proposed in order to refine the fuzzy sets in a second step (Berenji, 1990).

In the framework of the present work, three alternative procedures have been defined in order to automate the layout construction of a LRFN. One of the procedures is based on a variant of the k-means according to the algorithm described in (Moody & Darken, 1989; Wilpon & Rabiner, 1985). A second procedure is based on CART (Breiman et al., 1984), an algorithm for generating Regression Trees. Finally, the third technique is based on symbolic learning methods in the line of (Sammut et al., 1992), and allows learning both from data and from background knowledge. In the present form, the procedure based on k-means is immediately applicable to the RBFN and TDRBFN architectures whereas the other two have been implemented in order to work on the architecture of the fuzzy controller. Nevertheless, they can be easily generalized to work indifferently on every kind of LRFN architecture. Afterwards, the networks can be trained on-line e.g. by error minimization using a gradient descent technique, which has the advantage that it is also applicable on-line.

4.1. Learning LRFNs' Layout by means of Statistical Clustering

The original proposal by Moody and Darken (Moody & Darken, 1988) for learning RBFNs' layouts was to use a modified k-means clustering algorithm to find a set of k clusters. However, k-means is an unsupervised algorithm, i.e., it does not take the output information into account. If the examples given for training the RBFN are in the form of $(\vec{x}, \vec{y})$ with $\vec{y}$ being the desired output vector for input $\vec{x}$, the network construction should use this information.

The procedure applied for the experiments described in Section 5 is based on work presented in (Musavi et al., 1992). Its purpose can be described as[3]:

Given: A set of examples $\{(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n)\}$
Determine: A set of clusters $\{(\vec{\mu}_1, \sigma_1), \ldots, (\vec{\mu}_C, \sigma_C)\}, C \ll n$ that covers the given $\vec{x}_i$ and classifies them correctly with respect to the desired output y_i.

The algorithm used for initializing RBFNs networks is the following:

1. Divide the output interval $[y_{min}, y_{max}]$ into K equidistant and distinct intervals $I_1, \ldots, I_K$
2. For all $i \in \{1, \ldots, n\}$: Define the index $k \in \{1, \ldots, K\}$ of the interval I_k covering y_i as the *class* of $\vec{x}_i$. This is denoted by $Class(\vec{x}_i) = k$.

3. Define a cluster $(\vec{\mu}, \sigma) := (\vec{x}_i, \beta\sigma_{min})$ for each $i \in \{1, \ldots, n\}$, with $\sigma_{min} := min^n_{i,j=1,i\neq j}||\vec{x}_i - \vec{x}_j||$. This step results in $C = n$ initial clusters.
4. For all $i, j \in \{1, \ldots, C\}, i \neq j$, if $Class(\vec{\mu}_i) = Class(\vec{\mu}_j) = k$ try to merge the clusters:
   (A) Compute $\vec{\mu}_{new} : \mu_{new_l} = \frac{\mu_{i_l}+\mu_{j_l}}{2}$ for $l \in \{1, \ldots, dim(\vec{\mu}_i)\}$.
   (B) Compute $\sigma_{new} = \frac{1}{2}||\vec{\mu}_i - \vec{\mu}_j|| + max(\sigma_i, \sigma_j)$.
   (C) Compute $r = max^n_{l=1}\{||\vec{\mu}_{new} - \vec{x}_l|| : Class(\vec{x}_l) = k\}$.
   (D) Compute $d = min^n_{l=1}\{||\vec{\mu}_{new} - \vec{x}_l|| : Class(\vec{x}_l) \neq k\}$.
   (E) If $d > \alpha r$ remove cluster j, set $(\vec{\mu}_i, \sigma_i) := (\vec{\mu}_{new}, \sigma_{new})$. This means $C = C - 1$.
5. If at least two clusters have been merged (i.e., the number C of clusters has been reduced) go back to 4.

The parameters α and β, used in this procedure, have the following meaning: α tells how *conservative* the algorithm is about generating clusters that cover some example belonging to classes different than the one at issue. If $\alpha > 1$, no cluster covering wrong examples will be generated. If $\alpha < 1$, the algorithm is allowed to generate clusters covering wrong examples. Therefore, changing α is a way to influence both the accuracy that can be expected from the RBFN generated on the found clusters (a greater α means higher accuracy) and the size of the network (the smaller α, the more clusters will be merged and the less will be kept). Differently than the observations documented in (Musavi et al., 1992), for function approximation problems as they are considered here, selecting $\alpha < 1$ has proven to provide very satisfactory results, both in terms of network size and final approximation accuracy (see Section 5).

The β parameter determines the initial size of the clusters. A conservative selection for β is $\beta < 0.5$ or even $\beta = 0$, which guarantees that the initial clusters do not interfere. A setting of $\beta > 0.5$ or even $\beta > 1$ results in an initial cluster set that already provides ambiguous classifications. This is usually not desired.

4.2. Learning LRFNs' Layouts with CART

Regression Trees were introduced by (Breiman et al., 1984) for function approximation. A regression tree partitions the domain of a function $f(\vec{x})$ into rectangular regions (Fig. 7) where the value of $f(\vec{x})$ is similar, such that it can be approximated by a constant. Therefore, $f(\vec{x})$ is approximated by a histogram as accurately as the subdivision of the input domain permits.

The algorithm for generating a regression tree (CART) is simple and a detailed description can be found in (Breiman et al., 1984).

Given a learning set $\{(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n)\}$, CART works as follows:

1. Start with a tree containing only one leaf, classifying all samples in the same category, and let the label of this leaf be the average value $f(\vec{x})$;

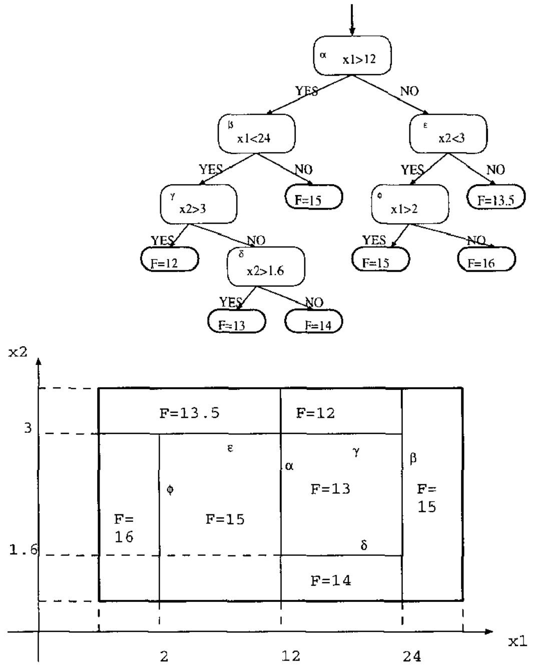

Figure 7. A Regression Tree

2. While the desired accuracy ϵ has not been achieved, repeat:

   (A) Choose a leaf F having an impurity $I(F)$ (the mean square error) greater than ϵ;

   (B) Find the feature x_j for which there exists a split value k_j so that the global impurity $I(F^{(1)}) + I(F^{(2)})$ for the two leaves $F^{(1)}$ and $F^{(2)}$, obtained by splitting F in k_j, is minimized;

   (C) Convert F into a node with the test $x_j \geq k_j$ and two leaves $F^{(1)}$ and $F^{(2)}$ as sons;

   (D) Label $F^{(1)}$ and $F^{(2)}$ with the average value of $f(\vec{x})$ in the corresponding regions of the domain.

To control overfitting, the regression tree can be pruned: the tree is explored bottom-up and every inner node (which corresponds to a test) is removed if this does not cause the error computed on a second learning set (the *pruning set*) to increase.

A regression tree can be employed directly to build a controller, but it has some major disadvantage: it approximates the function with a histogram (Fig. 10a), while smoothness of the output is desirable to avoid harmful mechanical solicitations to the robot; only its consequents (output values) can be fine-tuned, and not the input classification part, finally, the known learning algorithms for regression trees are not incremental, whereas we need an incremental tool for on-line refinement.

For all these reasons, we will convert a regression tree into the layout of a LRF network (both a Fuzzy Controller or a RBFN) that can the be further refined. This is done by the following algorithm:

1. For each leaf F_i in the tree, identify the rectangular region R_i it defines in the input space. This is done by collecting the tests on the path from the root to the leaf.

2. If the target network is a RBFN, every region R_i becomes a neuron r_i, centered on the center of R_i and spanning on it; in the case of a Fuzzy Controller, every edge σ_{ij} of R_i becomes a neuron in the first hidden layer, associated to a unidimensional Gaussian of amplitude σ_{ij}, connected to a neuron r_i in the second hidden layer and having the $softmin$ as activation function.

3. Every neuron r_i is connected to the output neuron by a link with weight w_i equal to the average value of $f(\vec{x})$ in the corresponding region R_i.

4.3. Constructing LRFNs' Layouts using a Concept Learning Algorithm

A regression problem can be mapped into a classification problem using a simple method: the real co-domain of the target function $f(\vec{x})$ is approximated by a finite set $\mathcal{O}$ of values. $f(\vec{x})$ is transformed into a histogram $f^*(\vec{x})$ by replacing each output value with the nearest value in $\mathcal{O}$. In this way, the regression problem becomes a classification problem, being $\mathcal{O}$ the set of classes and $f^*(\vec{x})$ the learning set, which can be solved using an algorithm such as C4.5. An example of this method is also described in (Sammut et al., 1992).

The classification knowledge in a decision tree can be processed similarly to the one encoded in regression trees, in order to obtain an LRFN network. The improvement, w.r.t. the work of Sammut and Michie, is that both fuzzy controllers and RBFNs can be further trained, obtaining a greater accuracy.

In the present case, we didn't use decision trees since they did not offer advantages over regression trees. Instead, we applied a more flexible learner called SMART+ (Bergadano et al., 1988; Botta & Giordana, 1993), that allows the induction process to be guided by the background knowledge of a domain expert. SMART+ is a complex system capable of learning in First Order Logics as well as in Propositional Calculus (as in the present case). A thorough description of the system is beyond the aim of this paper, thus we will only describe the subset of it that has been used here.

As well as FOIL (Quinlan, 1990) and FOCL (Pazzani & Kibler, 1992), SMART+ uses a general to specific learning strategy. Given a set of classes $\mathcal{O}$, it starts the search from a formula ϕ, which can be the predicate $True$ or a more complex formula (suggested by an expert or deduced from a domain theory) (Bergadano & Giordana, 1988).

One major feature, distinguishing SMART+ from other first order language learning systems, is its ability to deal with continuous attributes. This is done using predicate schemes such as

$$P(x_1, x_2, ..., x_n, K : [v_{min}, v_{max}, \delta]). \tag{4}$$

In (4) $x_1, x_2, ..., x_n$ are variables and K denotes a numerical threshold to be determined during the learning process. The interval $[v_{min}, v_{max}, \delta]$ defines the range for K and the required precision. Obviously, in a learning problem framed in propositional calculus, predicate schemes will not have any variable but only constants to tune. An example of a predicate used for learning a controller's layout can be $GreaterForce_x(K : [10, 25, 0.1])$ stating that the force along the X axis be greater than a constant K, ranging from 10 to 25 Newtons, with granularity 0.1.

In this case, the learning events are simply described as attribute vectors where $Force_x$ is one of them. The semantics of the predicate $GreaterForce_x$ is simply given by a function which compares the attribute $Force_x$ with the value assigned to K and returns true if the former is greater and false otherwise. SMART+ environment offers a function library designed in order to define the predicate semantics on the attributes of the learning events. In this way, background knowledge can be easily encapsulated in the predicate semantics.

A second method for supplying background knowledge is provided by necessity constraints which are statements of the type:

$$P(x_1, x_2, ..., x_n) \; needs \; \phi(y_1, y_2, ..., y_m) \tag{5}$$

where $\phi(y_1, y_2, ..., y_m)$ is an and/or formula belonging to the concept description language. The meaning of (5) is: $P(x_1, x_2, ..., x_n)$ can be true only if $\phi(y_1, y_2, ..., y_m)$ has been proven true. In general, SMART+ will use a predicate P for making a formula ψ more specific, only if its necessity constraint ϕ is a subformula of ψ. The induction space can be arbitrarily constrained by properly defining a set of necessity constraints.

The induction algorithm used by SMART+ for learning controller layouts learns one class o at a time:

Let $E(o_i)$ be the set of examples for the current target class o_i; let moreover, $\nu(\phi)$ be a measure of the quality of an inductive hypothesis ϕ. While $E(o_i)$ is not covered, or no new inductive hypotheses are found, do:

1. Start with an "or" tree T containing the formula "True", or a tree previously initialized with a set of formulas derived from a theory (or given by an expert).

2. Insert in the list $OPEN$ all the leaves currently existing in T.

3. Select from $OPEN$ the formula ϕ having the best ranking $\nu(\phi)$.

4. Determine the set P_A of predicates applicable to ϕ according to the predicate constraints.

5. For each predicate $p(K) \in P_A$, find the assignment k for the parameter K in p which maximizes $\nu(\phi \wedge p(k))$. If $\nu > 0$ put $\phi \wedge p(k)$ in the list $TODO$ sorted according to decreasing values of ν.

6. If $TODO$ is longer than a parameter $n \geq 1$, limiting the branching factor of T, truncate $TODO$ after the n-th item. Then, expand T with the formulas in $TODO$ and put them in $OPEN$.

7. If $OPEN$ contains some classification rule ψ(according to an assigned criterion), go to step 1; otherwise go to step 2.

The function ν used to rank an inductive hypothesis ϕ, has been chosen to be the function

$$if(v \geq v_0)\ then\ \nu = vw\ else\ \nu = 0 \tag{6}$$

with $0 \leq v \leq 1$ and $0 \leq w \leq 1$ being the proportion of positive examples covered by ϕ and the correctness of ϕ, respectively. The condition $v \leq v_0$ prevents the generation of too specific classification rules, in order to control the overfitting. Additionally, SMART+ offers a wide choice of evaluation criteria, including the well known $information\ gain$.

Inductive hypotheses are accepted as classification rules also if they are slightly inconsistent. In particular, a classification rule ϕ is allowed to cover some examples of the two classes corresponding to the two values in $\mathcal{O}$ adjacent to the target class o_i, provided that its correctness doesn't drop below an assigned threshold w_0. The tuning of w_0 and v_0 is left to the user and can require several trials to find a proper setting. A second constraint which can be posed on the acceptance of a classification rule is on the syntactic structure. In particular, it is possible to require such a rule to contain one or more predicates specified by the user. In this way, it is possible to achieve a complete characterization of the input region associated to the value represented by the target class.

The antecedents of the rules generated by SMART+ describe hyper-rectangles in the input space that can be processed as in the case of regression trees. Rule consequents are used for initializing the weights on the connections from the generated networks' hidden layer to the output neurons. Incompleteness and incorrectness of the rule set can easily be recovered by subsequently training the network's weights.

4.4. Refining LRFNs On-line

Let $E = \frac{1}{2}(Y - y)^2$ denote the square error exhibited by an approximation $y = f(\vec{x})$ of the function $Y = F(\vec{x})$ for a specific input $\vec{x}$. Let moreover $f(\vec{x})$ depend upon a set $\mathcal{P}$ of tunable parameters. The error gradient descent is performed by iteratively updating each parameter $P_i \in \mathcal{P}$ with the following rule (Rumelhart et al., 1985):

$$\Delta P_i = -\eta \frac{\partial E}{\partial P_i} = -\eta \frac{\partial E}{\partial y}\frac{\partial y}{\partial P_i} = \eta(Y - y)\frac{\partial y}{\partial P_i} \tag{7}$$

If the update of the parameter $\mathcal{P}$ takes place immediately, the learning will be characterized as *on-line*. If the effect is delayed until the whole learning set has been processed, the learning method is called *off-line* or *batch* learning. Batch learning guarantees more accurate results, and convergence proofs for backpropagation rely on batch learning as well. However, in robotics often on-line learning is required for obvious reasons.

In the following, we will derive the update rules for the different parameters of LRFNs, and we will show how they can be combined with other incremental learning rules.

4.4.1. *Refining RBFNs*

The typical RBFN training procedure is to consider the clusters $(\vec{\mu}, \sigma)$ as static. Only the weights between the cluster neurons and the output neuron are trained by means of the the rule (7) which for the case of $s = id$ simply reduces to: $\Delta w_i = \eta(Y - y)$. In the experimental section, networks trained using this simple method will be referred to as $RBFN_S$ or $TDRBFN_S$ (S stands for Simple).

Keeping the cluster centers constant during the whole life of the network is, however, not appropriate in all cases. Sometimes, not all data are available to perform the initial clustering. Also, the effort to analyze all given examples might be too high, thus only a subset considered representative is taken for setting up the network. A factor to be taken into special consideration w.r.t. real time applications is the size of the network, i.e. the time necessary for evaluation. The number of clusters should be small in such cases, making it much more difficult to yield the desired output accuracy by training the output weights only.

An obvious way to train the cluster centers during training is to apply a Kohonen-style nearest neighbour algorithm, also referred to as *adaptive incremental k-means* algorithm (Moody & Darken, 1988). Given an example $(\vec{x}, y)$, the center of the cluster closest to $\vec{x}$ is moved into the direction of $\vec{x}$ according to

$$\vec{x}_{closest} = \vec{x}_{closest} + \eta(\vec{x} - \vec{x}_{closest})$$

where η denotes a learning rate. During training, this adaptation can be interleaved with training the output weights, in order to avoid interference. Networks trained using this procedure will be referred to as $RBFN_{NN}$ and $TDRBFN_{NN}$.

Alternatively, if the cluster neurons' transfer function r is differentiable with respect to $\vec{\mu}$ and σ, it is possible to train both parameters by performing the error gradient descent, too (Weymaere & Martens, 1991; Wettschereck & Dietterich, 1991). Considering expression (7), for the case of a multidimensional Gaussian transfer function with individual σ_i for each dimension, i.e., for

$$r(\vec{x}) = e^{-(\sum_{i=1}^{dim(\vec{x})}(\frac{x_i - \mu_i}{\sigma_i})^2)},$$

the partial derivatives of y with respect to μ_i and σ_i are

$$\frac{\partial y}{\partial \mu_i} = 2\frac{\mu_i - x_i}{\sigma_i^2} r(\vec{x}) \quad \text{and} \quad \frac{\partial y}{\partial \sigma_i} = -2\frac{(\mu_i - x_i)^2}{\sigma_i^3} r(\vec{x})$$

In both cases, the error surface becomes extremely steep if the clusters are small, i.e., $\sigma_i \ll 1$. Since we are only interested in the sign of the gradient, we took care of this phenomenon by setting $\sigma_i = 1$ for the calculation of the partial derivatives. The networks trained using these methods are referred to as $RBFN_C$ or $TDRBFN_C$ and $RBFN_W$ or $TDRBFN_W$, respectively. If both methods are used in an interleaved way, this is referred to as $RBFN_{WC}$ or $TDRBFN_{WC}$.

4.4.2. Refining a Fuzzy Controller via Back-Propagation

The fuzzy controller of Figure 6 can be fine-tuned in a similar way as RBFNs; however, the *softmin* composition function must be taken into account. When updating the weights in the last layer, equation (7) becomes

$$\Delta w_i = \eta(Y - y)\frac{r_i}{\sum_i r_i} \tag{8}$$

where $\frac{\partial y}{\partial w_i} = \frac{r_i}{\sum_i r_i}$ is the derivative of equation (3). For updating first layer weights (i.e., to adjust antecedent fuzzy sets), equation (7) must be rewritten as:

$$\Delta P_i = \eta(Y - y)\frac{\partial y}{\partial P_i} = \eta(Y - y)\sum_c \frac{\partial y}{\partial r_c}\frac{\partial r_c}{\partial \mu_i}\frac{\partial \mu_i}{\partial P_i} \tag{9}$$

where $\{P_i\} = \{C_i\} \cup \{\sigma_i\}$,

$$\frac{\partial y}{\partial r_s} = \frac{w_i\left(\sum_i r_i\right) - 1 \cdot \left(\sum_i w_i r_i\right)}{\left(\sum_i r_i\right)^2} \tag{10}$$

is the derivative of (3),

$$\frac{\partial r_c}{\partial \mu_i} = \frac{\frac{\partial \mathcal{N}}{\partial \mu_i}\mathcal{D} - \frac{\partial \mathcal{D}}{\partial \mu_i}\mathcal{N}}{\mathcal{D}^2} \tag{11}$$

$$\left(\mathcal{D} = \sum_l e^{-k\mu_l} \quad ; \quad \mathcal{N} = \sum_j \mu_j e^{-k\mu_j} \quad ; \quad r_c = \mathcal{N}/\mathcal{D} \quad \text{for brevity}\right)$$

is the derivative of the *softmin* operator (2) and

$$\frac{\partial \mu_i}{\partial \sigma_i} = \frac{2\mu_i(x - C_i)^2}{\sigma_i^3} \quad ; \quad \frac{\partial \mu_i}{\partial C_i} = \frac{2\mu_i(x - C_i)}{\sigma_i^2} \tag{12}$$

are the derivatives of the Gaussian membership functions (1).

5. Experimental Evaluation

In the following we will compare the function approximators and the learning procedures proposed in Sections 2 and 4, on two test cases. The first is a robotic application, in which the task is to approximate the output of a controller that is already operational on a robot manipulator. The goal of the experiments was to check the accuracy of the methods in capturing control functions. The results proved that all the approximators are potentially able to capture the control function from examples of correct behavior only.

The second test case is a chaotic function, the Mackey-Glass temporal series, used by many authors to check the predictivity of a function approximator.

What emerges from the experiments is that LRFNs are in general more accurate, especially when they are synthesized using the symbolic approaches based on CART and on SMART+.

5.1. Experiments on the Robot Traces

In this case, the experimental test-bed was given by a six-degrees-of-freedom KUKA-IR 361 manipulator equipped with a force-torque sensor. As the robot is kinematically controlled, the controller generates a vector $\vec{V}$ of translational and rotational velocities of the wrist.

The selected task, *peg-into-hole*, consists in learning to insert a peg into a hole and to recover from error situations, in which, for instance, the peg is stuck midway because of a wrong inclination. This application is particularly interesting because the optimal control is known to be non-linear (Asada, 1990).

In our case, both the peg and the hole had a circular section. The diameter of the chamfered peg was $30[mm]$, the clearance between the peg and the hole was $0.15[mm]$. The hole was located on a plane surface. In the experiments, a PID-controller was already available for the robot (De Schutter & Van Brussel, 1988), and the goal was to learn to approximate the behavior of the PID-controller using a set of examples of control behavior.

These examples were obtained by recording the traces of the pairs of input sensors readings and corresponding speed values, during seven insertions with different initial conditions such as different misalignments between the peg and the hole. Three of these sequences were chosen as learning examples, the others were used as the test set. Figure 8 shows the input signals for the seven sequences. Figure 9 shows the three output signals for the linear velocity, generated by the PID-controller and used to train our function approximators.

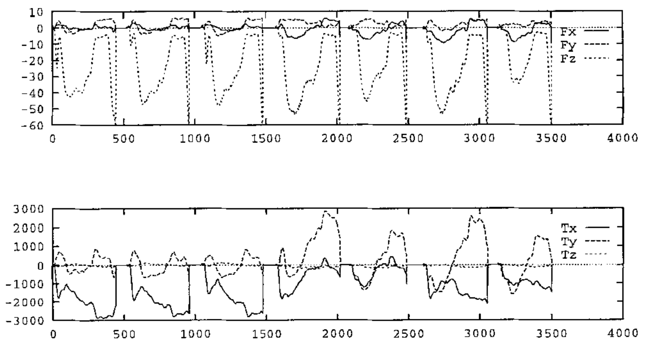

Figure 8. Input signals for the seven insertion sequences: F_x, F_y and F_z correspond to the force sensors along the three coordinates while T_x, T_y and T_z correspond to the torque sensors. The values of the functions have been sampled at 10 ms.

5.1.1. Comparative Results on Robot Traces

All learning methods have been applied to the case study, using the same learning and test sets described above. In order to make comparable the results, the refinement performing the error gradient descent has been done on-line and stopped after 1,500,000 learning steps.

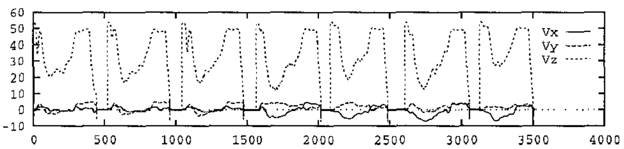

Figure 9. Velocity signals computed by the PID-Controller from the signals reported in the previous figure.

The method based on CART turned out very easy to apply because it didn't require any special setting on control parameters, while the one based on SMART+ required a little more work in order to decide the subdivision of the co-domain of the target functions into classes. After several trials it turned out that a subdivision into 12 equal intervals was a good choice for all three function V_x, V_y and V_z. Moreover, a set of predicate constraints has been defined in order to force SMART+ to generate rules necessarily accounting for the three features V_x, V_y and V_z. The complexity of the trees generated by CART and SMART+ was comparable and ranged from 20 to 80 leaves corresponding to a number of neurons in the hidden layer ranging from 60 to 160 (each neuron corresponds to an unidimensional Gaussian).

RBFNs and TDRBFNs were constructed using the algorithm described in section 4.1 and have been trained using the alternative algorithms we described in section 4. For the TDRBFN architecture only 1 delay has been used everywhere. The complexity of the networks ranged from 30 to 60 neurons in the hidden layer (each neuron corresponds to a 6 dimensional Gaussian).

For the TDNN experiments, the structure of the network and the number of delays at each layer was designed manually in a trial-and-error manner. Afterwards, the networks were trained by backpropagation (Rumelhart et al., 1985). The topologies used had a complexity ranging from 10 to 20 neurons. The best number of delay units found during the experimentation was 3 in the input layer and 0 in the other layers.

Finally, the experiment has been done using the basic multilayer perceptron for comparison purposes. Also in this case, the choice of the layout required several experiments.

The results are reported in Table 1. It is easy to see the methods based on CART and SMART+ obtained very good results (CART is the best in absolute) as well as the TDNN and the TDRBFN. On the contrary multilayer perceptron and normal RBFN were not so good even if in general they can achieve acceptable results.

Concerning the training algorithm for RBFN we notice the relevance of training the hidden layer neurons. In particular, the methods based on gradient descent performed better than the one based on incremental k-means.

In order to better understand the behavior of the two methods based on CART and SMART+, some details of the learning process are illustrated in Figure 10 and in Figure 11.

In particular, Figure 10a shows how CART alone approximates the target function with a histogram; Figure 10b shows the output for the same sequence generated by the fuzzy

Table 1. Statistical comparative results; all numbers are percentages (%)

System	V_x $Err_{avg} \pm Std.dev.$	V_y $Err_{avg} \pm Std.dev.$	V_z $Err_{avg} \pm Std.dev.$
CART (alone)	0.76 ± 1.05	1.02 ± 1.33	1.38 ± 5.01
FC_{CART}	1.16 ± 2.51	1.77 ± 3.75	11.84 ± 20.92
FC_{CART} + R	0.04 ± 0.17	0.05 ± 0.31	0.45 ± 2.75
FC_{SMART+}	1.49 ± 2.02	1.28 ± 1.93	2.12 ± 4.94
FC_{SMART+} + R	0.73 ± 0.42	0.56 ± 0.35	1.04 ± 4.18
$RBFN_S$	3.0 ± 0.61	2.4 ± 0.37	3.4 ± 3.81
$TDRBFN_S$ (1 TD)	6.2 ± 0.85	11.5 ± 1.11	2.6 ± 2.37
$RBFN_{NN}$	2.9 ± 0.67	7.6 ± 0.78	4.5 ± 4.62
$TDRBFN_{NN}$ (1 TD)	2.4 ± 0.45	1.8 ± 0.18	1.6 ± 3.15
$RBFN_C$	2.2 ± 0.45	3.6 ± 0.38	3.3 ± 3.80
$TDRBFN_C$ (1 TD)	1.7 ± 0.32	1.7 ± 0.18	1.0 ± 3.00
$RBFN_W$	2.1 ± 0.40	2.9 ± 0.29	3.3 ± 3.83
$TDRBFN_W$ (1 TD)	1.9 ± 0.29	0.8 ± 0.09	1.7 ± 3.10
$RBFN_{CW}$	2.4 ± 0.50	2.3 ± 0.33	2.5 ± 3.67
$TDRBFN_{CW}$ (1 TD)	1.5 ± 0.25	1.4 ± 0.15	1.9 ± 3.15
TDNN	3.9 ± 0.59	8.1 ± 0.90	2.7 ± 2.30
MLP	7.9 ± 1.13	10.9 ± 1.10	4.5 ± 3.84

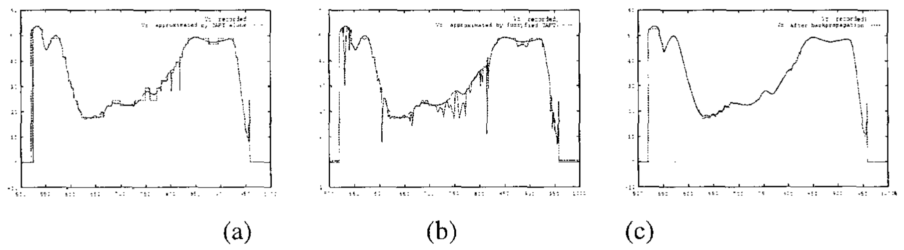

Figure 10. Approximation of the control function V_z obtained using: (a) CART only; (b) after transforming the tree of CART into a fuzzy controller; (c) after training the fuzzy controller performing the error gradient descent.

controller produced by converting the CART tree into a set of fuzzy rules. Figure 10c shows the signal after further training with backpropagation. The signal shown is obtained from the second training example and is the velocity V_z.

In a similar way, the procedure based on SMART+ is illustrated in Figure 11. In particular, 11a reports the histogram approximating the signal used as a learning set. Figure 11b shows the approximation obtained by the fuzzy controller before the training and 11c shows the result after the refinement step.

5.1.2. *Results on the Robot Simulator*

In order to test the learnt controllers "on the field", we used a professional robot simulation package (De Schutter et al., 1993).

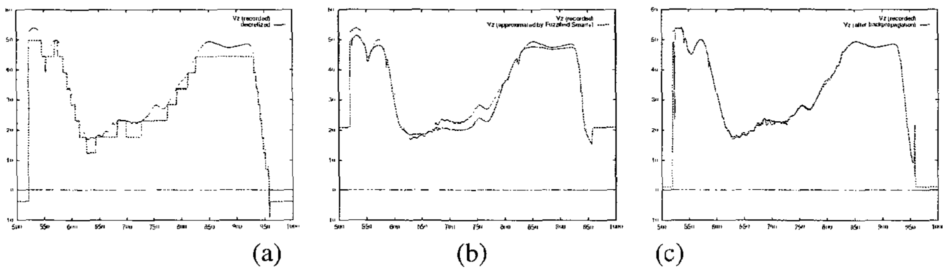

Figure 11. Approximation of the control function V_z obtained using SMART+: (a) histogram representing one of the signals in the learning set; (b) approximation generated by the fuzzy controller generated by SMART+; (c) after training by performing the error gradient descent.

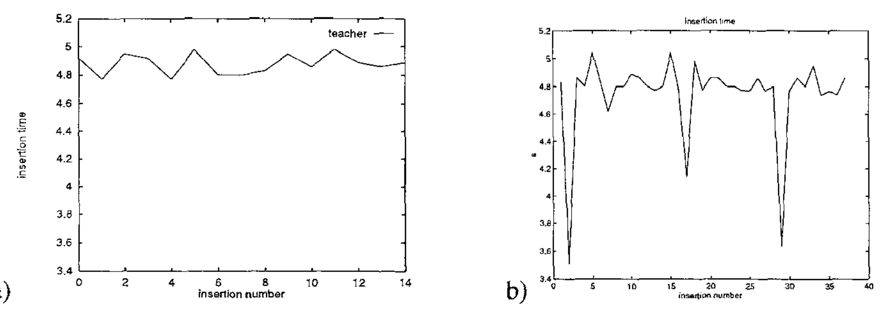

Figure 12. The behavior of the robot with the teacher (a) and with the learnt controller (b)

Figure 12 shows the behavior of the simulated robot arm: every trace depicts a series of insertions; each insertion is evaluated with the total insertion time (the lower, the better). As can be seen, the learnt controller exhibits a behavior comparable to the one of the teacher.

However, a test on the real robot should be more convincing. This has not been done for the simple reason that a controller generated using heuristic methods was not considered safe enough for a robot delicate and expensive as the KUKA IR-361 is. This fact points out another critical point which must be faced before using neural controllers in industrial applications, i.e., the development of validation techniques that are accepted by the mechanical engineers (ERA, 1995).

5.2. *Test on Mackey-Glass Chaotic Time Series*

As the learning procedures based on CART and SMART+ represent an important novelty which performed very well on robot traces, they have been tested also on a classical case study widely used in the literature in order to have a more extensive validation and a comparison with other methods. The task is to predict the value 84 time steps ahead in the

Mackey-Glass chaotic series, which is described by the following differential equation:

$$\dot{x} = \frac{0.2x(t-\tau)}{1+x^{10}(t-\tau)} - 0.1x(t)$$

with $x(t<0)=0$, $x(0)=1.2$ and $\tau=17$; the task was to predict $x(t+84)$ given $x(t-24), x(t-18), x(t-12)$ and $x(t-6)$.

The input features used by the approximators are the four values of the function at time $t, t-6, t-12$, and $t-18$. In order to correctly predict the function value, it is necessary to capture the generative model of the phenomenon. This task was also investigated by (Crowder, 1990; Jang, 1993; Jones et al., 1990; Lapedes & Farber, 1987; Moody, 1989; Moody & Darken, 1989; Sanger, 1991), so results can be compared directly.

The experiment has been organized as follows. First, a sequence of 1500 time steps has been generated. Then the learning set has been obtained by taking the first 1000 whereas the remaining 500 have been used as a test set. The results are reported in Table 2. The Non-Dimensional Error Index (NDEI) is defined as the Root Mean Square Error divided by the Standard Deviation of the target series.

Comparing the best results obtained in the literature in analogous experiments (Crowder, 1990; Jang, 1993), it appears that both the predictor generated by CART and the one generated by SMART+ show excellent performances after training. In fact, the NDEI value of 0.036 reported by ANFIS, followed by the 0.05 of the multilayer perceptron are the only performances really comparable to those presented here.

However, the big advantage of our new methods is that the networks have been synthesised automatically from examples, whereas for both ANFIS and the multilayer perceptron this has been done manually.

Regarding the learning procedure based on modified k-means, we refer to the work by (Moody & Darken, 1989). He reports a NDEI of 0.055 obtained with a training set of 10,000 examples and 1,000 hidden neurons. The method based on SMART+ obtained a better result using a fuzzy controller of 91 rules corresponding to a global number of 305 neurons in the first hidden layer. Moreover, the learning set was of only 1,000 examples.

The controller generated by CART was even smaller: 42 leaves in the tree corresponding to a global number of 161 neurons in the first hidden layer.

Table 2. Comparative results on the Mackey-Glass chaotic time series prediction

Method	Training cases	Non-Dimensional Error Index
CART (Alone)	800+200	0.36
FC_{CART} +R	1000	0.037
FC_{SMART+} +R	1000	0.032

6. Discussion

From the results presented in the previous section the excellent approximation capability of LRFNs clearly emerges. In fact, they obtained favourable results w.r.t. the multilayer perceptron and TDNNs, both on robot traces and on the Mackey-Glass series.

It is worth noticing that, considering the aim of robotic applications, the slight differences of performance between OFFNs and LRFNs (though certainly meaningful from a statistical viewpoint) cannot be decisive when looking for a technique to apply. Differences in the third decimal digit are not a sufficient reason for adopting a TDNN rather than an RBFN or a fuzzy controller; it is more fair to assume OFFNs and LRFNs as equivalent with respect to the accuracy.

Other considerations must, therefore, be taken into account in order to allow to select one architecture over another. LRFNs have specific properties that make them particularly suitable for robotic applications. The first is *support of incremental learning*, which is due to the locality property and has been discussed in Section 3. The possibility of incrementally building the mapping from the input space to the output space is equivalent to split the learning problem into subproblems; thus, different parts of the control function can be learnt during different learning sessions. For example the robot can first be trained to work slowly. Afterwards, specific experiments can be performed in order to increase its speed. The locality property also helps to avoid the *unlearning problem*, that arises in OFFNs when trained for a long time on a subset of the input space only: the network overspecializes and forgets knowledge about regions of the input space it doesn't see for a while. Although we didn't directly address this aspect of incrementality in our work, several examples can be found in the literature (see for instance (del R. Millán & Torras, 1992; del R. Millán, 1994)).

A second important feature is the possibility of giving a *symbolic interpretation* to LRFNs. We exploit this property when we apply symbolic and statistical learning algorithms for producing the network layout. In our opinion, this is a key point of the presented work, that establishes a *bridge* between approaches that (with a few exceptions) have traditionally been considered far apart, and thus scarcely interacted. Beyond the theoretical interest, we especially see practical benefits in the capability of merging the knowledge of an expert into a network.

In the literature, at least two other methods have already been proposed in order to integrate the symbolic and connectionist paradigms. The first is KBANN by Shavlik and Towell (Towell et al., 1990), who proposed to use a propositional theory in order to initialize a multilayer perceptron. The considered task was classification in a domain of Boolean features. The experiments showed that if a theory, which is a good approximation of the target Boolean function, is available, learning itself speeds up dramatically and the classification rate increases as well.

The main difference between the approach proposed in this paper and that of Shavlik's depends on the task they tackle. Learning a Boolean classifier is a simpler problem than learning a continuous function in a continuous domain. In this second case, the problem of learning the mapping from input signals to symbols, and then from symbols to an output signal, is also included in the process. Such mappings, which are naturally supported by the RBFNs are left out from the learning algorithm used by KBANN.

KBANN has been improved by adding algorithms for turning a multilayer perceptron back into a propositional theory (see (Towell & Shavlik, 1993) for a description of the method), a process we will call "inverse mapping of the network".

Inverse mappings can be found, in general, also in the case of LRFNs, for two reasons: (1) because of the locality property, that allows a discretization of inputs and outputs; (2) because they maintain their topological structure during learning. In particular, in the case of RBFNs and FCs, network inverse mappings can be obtained simply by reversing the process of encoding the theory into a neural net. Last, for the sake of completeness, it is necessary to pinpoint a difference between the kind of theories we can map onto the layout of an LRFN and those used by KBANN: the former are flat, in the sense that consequents of rules are not used in antecedents of other rules, whereas the latter can be structured into many layers.

A last point, needing a more detailed discussion, is the necessity of continuing the refinement of a controller beyond the teacher performances. Especially in the case of robot control, it is quite unlikely that a human operator will generate perfect examples of behavior (Kaiser et al., 1995c; Kaiser et al., 1995a). Therefore, some other solution has to be sought in order to optimize the induced controller, and Reinforcement Learning seems to be a promising approach. Actually, in RL literature we find many techniques for learning in continuous domains (Peng & Williams, 1992; Berenji & Khedkar, 1992; Williams, 1992; Gullapalli, 1990), in which agents implemented as neural networks are trained by means of variants of backpropagation. In all these cases, the ability to learn from a reinforcement signal is achieved by finding a quantity (depending on it) to backpropagate, i.e. playing the part played by the error in supervised learning rules. Most of the techniques proposed are, for instance, related to the TD(λ) method, in which differences of prediction in subsequent timesteps are exploited (see (Sutton & Barto, 1987)).

One system that integrates neural and reinforcement learning is Berenji's GARIC (Berenji & Khedkar, 1992). This system uses a fuzzy controller, implemented as a neural network, which is very similar to those described in Section 3. However, it is extremely easy to think to GARIC's variants in which the other kinds of LRFN approximators are used instead of fuzzy controllers. Therefore, the supervised learning methodologies presented here can be thought of as building blocks for more complex architectures.

7. Conclusions

Throughout this paper, we presented results achieved in the framework of B-LEARN II, a project that investigates the suitability of ML techniques for developing "advanced" industrial robots. Nonetheless, the fundamental claims that have been presented (supported by an extensive experimentation), are interesting also for ML researchers.

The first claim is that there is a class of problems related to non-linear control that can potentially be solved by employing existing learning algorithms. More specifically, the application of ML techniques to such problems can lead to real industrial applications, with an increase in robot performance and a decrease in costs of controller design and development. In particular, the experiments presented in this paper confirm that many techniques, originating from different areas, are available for facing the regression problem tied to non-linear control.

The second important claim concerns the integration of the symbolic and connectionist paradigms. We have shown that, on the way opened by Shavlik and Towell (Towell et al., 1990; Towell & Shavlik, 1993), it is possible to use symbolic knowledge in order to build a quite accurate approximator that can further be improved using, e.g., backpropagation. This result opens a wide perspective for the integration of symbolic and non-symbolic methods in regression tasks.

At the same time we also established a link between ML and a parallel field, Fuzzy Logic, in which function approximators have also been developed combining symbolic and non-symbolic paradigms.

We made the attempt of adopting a unifying view for comparing architectures developed by different communities with, sometimes, competing approaches. We believe we have shown that, beyond the cultural background, there is a lot in common among them, which *must* be taken into account.

Notes

1. This *is* to be considered an advanced technological solution, since today's industrial robots make a very little use of external sensors.
2. Proportional Integral Differential controller, i.e., a controller which produces an output that is proportional to its input, the input derivative, and the input accumulated over time. The input might be the error, e.g., the difference between the desired force and the actually measured force.
3. For clarity, only a one dimensional output vector is considered. However, the method works on examples featuring arbitrary dimensions. With $dim(\vec{y}) = n$, the intervals I_j are becoming hypercubes of dimension n, and the number of classes to be considered becomes $n \times K$.

References

Alpaydin, E. (1991). GAL: Networks that grow when they learn and shrink when they forget. Technical Report TR-91-032, International Computer Science Institute, Berkeley, USA.

Asada, H. (1990). Teaching and learning of compliance using neural nets: Representation and generation of nonlinear compliance. In *Proceedings of the 1990 IEEE International Conference on Robotics and Automation*, pages 1237 – 1244.

Barto, A., Sutton, R., & Watkins, C. (1990). Sequential decision problems and neural networks. In *Advances in neural information processing system*, volume 2. Morgan Kauffman, San mateo, Ca.

Barto, A. G., Sutton, R. S., & Anderson, C. W. (1983). Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics*, pages 835–846.

Berenji, H. (1990). Machine learning in fuzzy control. In *International Conference on Fuzzy Logic & Neural Networks*, pages 231–234, Iizuka, Fukuoka, Japan.

Berenji, H. (1992). Fuzzy logic controllers. In Yager, R. and Zadeh, L., editors, *An Introduction to Fuzzy Logic Applications in Intelligent Systems*. Kluwer Academic Publishers.

Berenji, H. & Khedkar, P. (1992). Learning and tuning fuzzy controllers through reinforcements. *IEEE Transactions on neural networks*, 3(5):724–740.

Bergadano, F. & Giordana, A. (1988). A knowledge intensive approach to concept induction. In *Proceedings of the 5th International Conference on Machine Learning*, pages 305–317, Ann Arbor, MI. Morgan Kauffman.

Bergadano, F., Giordana, A., & Saitta, L. (1988). Learning concepts in noisy environment. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, pages 555–578.

Berthold, M. (1994). A time delay radial basis function network for phoneme recognition. In *IEEE International Conference on Neural Networks*, Orlando, Florida.

Bonissone, P. & Chiang, K. (1993). Fuzzy logic controllers: from development to deployment. In *IEEE International Conference on Neural Networks*, volume 2, San Francisco, CA.

Botta, M. & Giordana, A. (1993). SMART+: A multi-strategy learning tool. In *IJCAI-93, Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, volume 2, Chambéry, France.

Breiman, L., Friedman, J., Ohlsen, R., & Stone, C. (1984). *Classification And Regression Trees*. Wadsworth & Brooks, Pacific Grove, CA.

Cramer, H. (1974). *Mathematical Methods of Statistics*. Princeton University Press.

Crowder, R. (1990). Predicting the mackey-glass time series with cascade-correlation learning. In D. Touretzky, G. H. and T.Sejnovsky, editors, *Proceedings of the 1990 Connectionist Models Summer School*, pages 117–123. Carnegie Mellon University.

De Schutter, J. & Van Brussel, H. (1988). Compliant robot motion II, a control approach based on external control loops. *The International Journal of Robotics Research*, 7(4).

De Schutter, J., Witvrouw, W., Van De Poel, P., & Bruyninckx, H. (1993). Rosi: a task specification and simulation tool for force sensor based robot control. In *24th International Symposium on Industrial Robots*.

del R. Millán, J. (1994). Learning efficient reactive behavioral sequences from basic reflexes in a goal-directed autonomous robot. In *Proceedings of the third International Conference on Simulation of Adaptive Behavior*.

del R. Millán, J. & Torras, C. (1992). A reinforcement connectionist approach to robot path finding in non-maze-like environments. *Machine Learning*, 8:363–395.

ERA (1995). *Neural Networks: Producing Dependable Systems*, Solihull, West Midlands, UK. ERA Technology.

Fahlmann, S. E. & Lebiere, C. (1989). The cascade-correlation learning architecture. In *Advances in Neural Information Processing Systems 2 (NIPS-2)*, Denver, Colorado.

Fritzke, B. (1993). Growing cell structure: A self-organizing network for unsupervised and supervised learning. Technical Report TR-93-026, International Computer Science Institute.

Gullapalli, V. (1990). A stochastic reinforcement learning algorithm for learning real valued functions. *Neural Networks*, 3:671–692.

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feed-forward networks are universal approximators. *Neural Networks*, 2:359–366.

Jang, J. (1993). ANFIS: Adaptive-Network-Based Fuzzy Inference System. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-23(3):665–687.

Jones, R., Lee, Y., Barnes, C., Flake, G., Lee, K., & Lewis, P. (1990). Function approximation and time series prediction with neural networks. In *Proceedings of IEEE International Joint Conference on Neural Networks*, pages I–649–665.

Kaiser, M., Camarinha-Matos, L., Giordana, A., Klingspor, V., del R. Millàn, J., Nuttin, M., & Suarez, R. (1994). Robot learning - three case studies in robotics and machine learning. In *Proceedings of the IVAR '94*, Leuven, Belgium.

Kaiser, M., Friedrich, H., & Dillmann, R. (1995a). Obtaining good performance from a bad teacher. In *International Conference on Machine Learning, Workshop on Programming by Demonstration*, Tahoe City, California.

Kaiser, M., Klingspor, V., del R. Millàn, J., Accame, M., Wallner, F., & Dillmann, R. (1995b). Using machine learning techniques in real-world mobile robots. *IEEE Expert*.

Kaiser, M. & Kreuziger, J. (1994). Integration of symbolic and connectionist processing to ease robot programming and control. In *ECAI'94 Workshop on Combining Symbolic and Connectionist Processing*, pages 20 – 29.

Kaiser, M., Retey, A., & Dillmann, R. (1995c). Robot skill acquisition via human demonstration. In *Proceedings of the International Conference on Advanced Robotics (ICAR '95)*.

Lapedes, A. & Farber, R. (1987). Nonlinear signal processing using neural networks: Prediction and system modeling. Technical Report LA-UR-87-2662, Los Alamos National Laboratory.

Mason, M. (1981). Compliance and force control for computer controlled manipulators. *IEEE Transactions on Systems, Man and Cybernetics*, 11.

Miller, W. T., Sutton, R. S., & Werbos, P. J. (1990). *Neural networks for control*. The MIT Press.

Moody, J. (1989). Fast learning in multi-resolution hierarchies. In Touretzky, D., editor, *Advances in Neural Information Processing*. Morgan Kaufmann.

Moody, J. & Darken, C. (1988). Learning with localized receptive fields. In Sejnowski, T., Touretzky, D., and Hinton, G., editors, *Connectionist Models Summer School*, Carnegie Mellon University.

Moody, J. & Darken, C. (1989). Fast learning in networks of locally tuned units. *Neural Computations*, 1(2):281–294.

Musavi, M., Ahmed, W., Chan, K., Faris, K., & Hummels, D. (1992). On the training of radial basis function classifiers. *Neural Networks*, 5:595–603.

Nuttin, M., Van Brussel, H., Baroglio, C., & Piola, R. (1994). Fuzzy controller synthesis in robotic assembly: Procedure and experiments. In *FUZZ-IEEE-94: Third IEEE International Conference on Fuzzy Systems, World Congress on Computational Intelligence*.

Nuttin, M., Van Brussel, H., Peirs, J., Soembagijo, A. S., & Sonck, S. (1995). Learning the peg-into-hole assembly operation with a connectionist reinforcement technique. In *Second International CIRP Workshop on Learning in Intelligent Manufacturing Systems*, pages 335–357, Budapest, Hungary.

Park, J. & Sandberg, W. (1993). Universal approximation using radial-basis functions. *Neural Computation*, 5.

Pazzani, M. & Kibler, D. (1992). The utility of knowledge in inductive learning. *Machine Learning*, 9:57–94.

Peng, J. & Williams, R. (1992). Efficient learning and planning within the Dyna framework. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, Honolulu, HI.

Poggio, T. & Girosi, F. (1990). Networks for approximation and learning. *Proceedings of the IEEE*, 78(9):1481–1497.

Quinlan, J. (1993). Combining instance-based and model-based learning. In *Proceedings of the* 10^{th} *machine learning conference*, pages 236–243, Amherst, MA.

Quinlan, R. (1990). Learning logical definitions from relations. *Machine Learning*, 5:239–266.

Rumelhart, D., Hinton, G., & Williams, R. (1985). Learning internal representations by error propagation. Technical Report 8506, Institute for Cognitive Science, La Jolla: University of California, San Diego.

Rumelhart, D. E. & McClelland, J. L. (1986). *Parallel Distributed Processing : Explorations in the Microstructure of Coginition, Parts I & II.* MIT Press, Cambridge, Massachusetts.

Sammut, C., Hurst, S., Kedzier, D., & Michie, D. (1992). Learning to fly. In Sleeman, D. and Edwards, P., editors, *Machine Learning - Proceedings of the Ninth International Workshop (ML92)*, pages 385–393. Morgan Kaufmann.

Sanger, T. (1991). A tree-structured adaptive network for function approximate in high-dimensional spaces. *IEEE Transactions on Neural Networks*, 2(2):285–293.

Specht, D. (1988). Probabilistic neural networks for classification mapping, or associative memory. In *IEEE International Conference on Neural Networks*, volume 1, pages 525–532.

Specht, D. (1990). Probabilistic neural networks. *Neural Networks*, 3:109–118.

Sutton, R. & Barto, A. (1987). A temporal-difference method of classical conditioning. In *proceedings of the Ninth Annual Conference of the Cognitive Science Society*, pages 355–378, Seattle, WA. Lawrence Erlbaum.

Towell, G. & Shavlik, J. (1993). Extracting refined rules from knowledge-based neural networks. *Machine Learning*, 13(1):71–101.

Towell, G., Shavlik, J., & Noordwier, M. (1990). Refinement of approximate domain theories by knowledge-based neural networks. In *Proceedings of the* 8^{th} *National Conference on Artificial Intelligence AAAI'90*, pages 861–866.

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., & Lang, K. (1989). Phoneme recognition using time-delay neural networks. *IEEE Transactions on acoustics, speech and signal processing*, pages 328–339.

Wettschereck, D. & Dietterich, T. (1991). Improving the performance of radial basis function networks by learning center locations. In *Advances in Neural Information Processing Systems 4 (NIPS-4).*

Weymaere, N. & Martens, J. (1991). A fast and robust learning algorithm for feedforward neural networks. *Neural Networks*, 4.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, pages 229–256.

Wilpon, J. & Rabiner, L. (1985). A modified k-means clustering algorithm for use in isolated work recognition. *IEEE transactions on acoustics, speech and signal processing*, ASSP-33:587–594.

Yih, J. & Shieh, J. (1992). On the development of a fuzzy model-based controller for robotic manipulators. In *Proceedings of the IEEE/RSJ Conference on Intelligent Robots and Systems*, Raleigh, NC.

Zadeh, L. (1992). Knowledge representation in fuzzy logic. In Yager, R. and Zadeh, L., editors, *An Introduction to Fuzzy Logic Applications in Intelligent Systems.* Kluwer Academic Publishers.