Part III

APPLICATIONS

8 Robot Control

An important area of application of neural networks is in the field of robotics. Usually, these networks are designed to direct a manipulator, which is the most important form of the industrial robot, to grasp objects, based on sensor data. Another applications include the steering and path-planning of autonomous robot vehicles.

In robotics, the major task involves making movements dependent on sensor data. There are four, related, problems to be distinguished (Craig, 1989):

Forward kinematics. Kinematics is the science of motion which treats motion without regard to the forces which cause it. Within this science one studies the position, velocity, acceleration, and all higher order derivatives of the position variables. A very basic problem in the study of mechanical manipulation is that of *forward kinematics*. This is the static geometrical problem of computing the position and orientation of the end-effector ('hand') of the manipulator. Specifically, given a set of joint angles, the forward kinematic problem is to compute the position and orientation of the tool frame relative to the base frame (see figure 8.1).

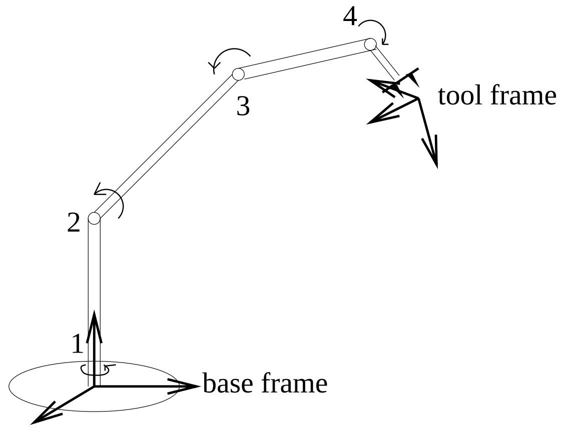

Figure 8.1: An exemplar robot manipulator.

Inverse kinematics. This problem is posed as follows: given the position and orientation of the end-effector of the manipulator, calculate all possible sets of joint angles which could be used to attain this given position and orientation. This is a fundamental problem in the practical use of manipulators.

The inverse kinematic problem is not as simple as the forward one. Because the kinematic equations are nonlinear, their solution is not always easy or even possible in a closed form. Also, the questions of existence of a solution, and of multiple solutions, arise.

Solving this problem is a least requirement for most robot control systems.

Dynamics. Dynamics is a field of study devoted to studying the *forces* required to cause motion. In order to accelerate a manipulator from rest, glide at a constant end-effector velocity, and finally decelerate to a stop, a complex set of torque functions must be applied by the joint actuators. In dynamics not only the geometrical properties (kinematics) are used, but also the physical properties of the robot are taken into account. Take for instance the weight (inertia) of the robotarm, which determines the force required to change the motion of the arm. The dynamics introduces two extra problems to the kinematic problems.

1. The robot arm has a 'memory'. Its responds to a control signal depends also on its history (e.g. previous positions, speed, acceleration).

2. If a robot grabs an object then the dynamics change but the kinematics don't. This is because the weight of the object has to be added to the weight of the arm (that's why robot arms are so heavy, making the relative weight change very small).

Trajectory generation. To move a manipulator from here to there in a smooth, controlled fashion each joint must be moved via a smooth function of time. Exactly how to compute these motion functions is the problem of *trajectory generation*.

In the first section of this chapter we will discuss the problems associated with the positioning of the end-effector (in effect, representing the inverse kinematics in combination with sensory transformation). Section 8.2 discusses a network for controlling the dynamics of a robot arm. Finally, section 8.3 describes neural networks for mobile robot control.

8.1 End-effector positioning

The final goal in robot manipulator control is often the positioning of the hand or end-effector in order to be able to, e.g., pick up an object. With the accurate robot arm that are manufactured, this task is often relatively simple, involving the following steps:

1. determine the target coordinates relative to the base of the robot. Typically, when this position is not always the same, this is done with a number of fixed cameras or other sensors which observe the work scene, from the image frame determine the position of the object in that frame, and perform a pre-determined coordinate transformation;

2. with a precise model of the robot (supplied by the manufacturer), calculate the joint angles to reach the target (i.e., the inverse kinematics). This is a relatively simple problem;

3. move the arm (dynamics control) and close the gripper.

The arm motion in point 3 is discussed in section 8.2. Gripper control is not a trivial matter at all, but we will not focus on that.

Involvement of neural networks. So if these parts are relatively simple to solve with a high accuracy, why involve neural networks? The reason is the applicability of robots. When 'traditional' methods are used to control a robot arm, accurate models of the sensors and manipulators (in some cases with unknown parameters which have to be estimated from the system's behaviour; yet still with accurate models as starting point) are required and the system must be calibrated. Also, systems which suffer from wear-and-tear (and which mechanical systems don't?) need frequent recalibration or parameter determination. Finally, the development of more complex (adaptive!) control methods allows the design and use of more flexible (i.e., less rigid) robot systems, both on the sensory and motory side.

8.1.1 Camera–robot coordination is function approximation

The system we focus on in this section is a work floor observed by a fixed cameras, and a robot arm. The visual system must identify the target as well as determine the visual position of the end-effector.

The target position $\mathbf{x}^{\text{target}}$ together with the visual position of the hand $\mathbf{x}^{\text{hand}}$ are input to the neural controller $\mathcal{N}(\cdot)$. This controller then generates a joint position $\boldsymbol{\theta}$ for the robot:

$$\boldsymbol{\theta} = \mathcal{N}(\mathbf{x}^{\text{target}}, \mathbf{x}^{\text{hand}}). \tag{8.1}$$

We can compare the neurally generated $\boldsymbol{\theta}$ with the optimal $\boldsymbol{\theta}_0$ generated by a fictitious perfect controller $\mathcal{R}(\cdot)$:

$$\boldsymbol{\theta}_0 = \mathcal{R}(\mathbf{x}^{\text{target}}, \mathbf{x}^{\text{hand}}). \tag{8.2}$$

The task of learning is to make the $\mathcal{N}$ generate an output 'close enough' to $\boldsymbol{\theta}_0$.

There are two problems associated with teaching $\mathcal{N}(\cdot)$:

1. generating learning samples which are in accordance with eq. (8.2). This is not trivial, since in useful applications $\mathcal{R}(\cdot)$ is an unknown function. Instead, a form of self-supervised or unsupervised learning is required. Some examples to solve this problem are given below;

2. constructing the mapping $\mathcal{N}(\cdot)$ from the available learning samples. When the (usually randomly drawn) learning samples are available, a neural network uses these samples to represent the whole input space over which the robot is active. This is evidently a form of interpolation, but has the problem that the input space is of a high dimensionality, and the samples are randomly distributed.

We will discuss three fundamentally different approaches to neural networks for robot end-effector positioning. In each of these approaches, a solution will be found for both the learning sample generation and the function representation.

Approach 1: Feed-forward networks

When using a feed-forward system for controlling the manipulator, a self-supervised learning system must be used.

One such a system has been reported by Psaltis, Sideris and Yamamura (Psaltis, Sideris, & Yamamura, 1988). Here, the network, which is constrained to two-dimensional positioning of the robot arm, learns by experimentation. Three methods are proposed:

1. *Indirect learning.*
   In indirect learning, a Cartesian target point $\mathbf{x}$ in world coordinates is generated, e.g., by a two cameras looking at an object. This target point is fed into the network, which generates an angle vector $\boldsymbol{\theta}$. The manipulator moves to position $\boldsymbol{\theta}$, and the cameras determine the new position $\mathbf{x}'$ of the end-effector in world coordinates. This $\mathbf{x}'$ again is input to the network, resulting in $\boldsymbol{\theta}'$. The network is then trained on the error $\epsilon_1 = \boldsymbol{\theta} - \boldsymbol{\theta}'$ (see figure 8.2).

   However, minimisation of ϵ_1 does not guarantee minimisation of the overall error $\epsilon = \mathbf{x} - \mathbf{x}'$. For example, the network often settles at a 'solution' that maps all $\mathbf{x}$'s to a single $\boldsymbol{\theta}$ (i.e., the mapping $\mathbb{I}$).

2. *General learning.*
   The method is basically very much like supervised learning, but here the plant input $\boldsymbol{\theta}$ must be provided by the user. Thus the network can directly minimise $|\boldsymbol{\theta} - \boldsymbol{\theta}'|$. The success of this method depends on the interpolation capabilities of the network. Correct choice of $\boldsymbol{\theta}$ may pose a problem.

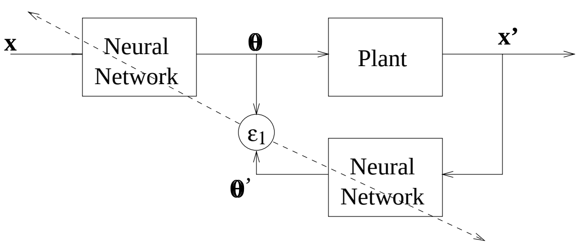

Figure 8.2: Indirect learning system for robotics. In each cycle, the network is used in two different places: first in the forward step, then for feeding back the error.

3. *Specialised learning.*

   Keep in mind that the goal of the training of the network is to minimise the error at the output of the plant: $\epsilon = \mathbf{x} - \mathbf{x}'$. We can also train the network by 'backpropagating' this error trough the plant (compare this with the backpropagation of the error in Chapter 4). This method requires knowledge of the Jacobian matrix of the plant. A Jacobian matrix of a multidimensional function F is a matrix of partial derivatives of F, i.e., the multidimensional form of the derivative. For example, if we have $Y = F(X)$, i.e.,

$$\begin{aligned} y_1 &= f_1(x_1, x_2, \ldots, x_n), \\ y_2 &= f_2(x_1, x_2, \ldots, x_n), \\ &\cdot \\ &\cdot \\ &\cdot \\ y_m &= f_m(x_1, x_2, \ldots, x_n) \end{aligned}$$

then

$$\begin{aligned} \delta y_1 &= \frac{\partial f_1}{\partial x_1}\delta x_1 + \frac{\partial f_1}{\partial x_2}\delta x_2 + \ldots + \frac{\partial f_1}{\partial x_n}\delta x_n, \\ \delta y_2 &= \frac{\partial f_2}{\partial x_1}\delta x_1 + \frac{\partial f_2}{\partial x_2}\delta x_2 + \ldots + \frac{\partial f_2}{\partial x_n}\delta x_n, \\ &\cdot \\ &\cdot \\ &\cdot \\ \delta y_m &= \frac{\partial f_m}{\partial x_1}\delta x_1 + \frac{\partial f_m}{\partial x_2}\delta x_2 + \ldots + \frac{\partial f_m}{\partial x_n}\delta x_n \end{aligned}$$

or

$$\delta Y = \frac{\partial F}{\partial X}\delta X. \tag{8.3}$$

Eq. (8.3) is also written as

$$\delta Y = J(X)\delta X \tag{8.4}$$

where J is the Jacobian matrix of F. So, the Jacobian matrix can be used to calculate the change in the function when its parameters change.

Now, in this case we have

$$J_{ij} = \left[\frac{\partial P_i}{\partial \theta_j}\right] \tag{8.5}$$

where $P_i(\boldsymbol{\theta})$ the ith element of the plant output for input $\boldsymbol{\theta}$. The learning rule applied here regards the plant as an additional and unmodifiable layer in the neural network. The

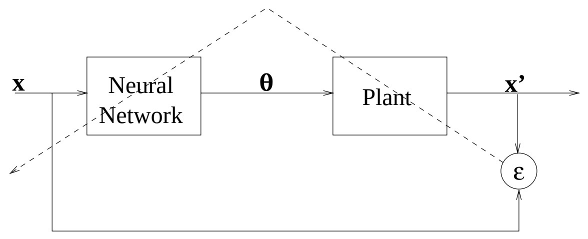

Figure 8.3: The system used for specialised learning.

total error $\epsilon = \mathbf{x} - \mathbf{x}'$ is propagated back through the plant by calculating the δ_j as in eq. (4.14):

$$\delta_j = \mathcal{F}'(s_j) \sum_i \delta_i \frac{\partial P_i(\boldsymbol{\theta})}{\partial \theta_j},$$
$$\delta_i = x_i - x_i',$$

where i iterates over the outputs of the plant. When the plant is an unknown function, $\frac{\partial P_i(\boldsymbol{\theta})}{\partial \theta_j}$ can be approximated by

$$\frac{\partial P_i(\boldsymbol{\theta})}{\partial \theta_j} \approx \frac{P_i(\boldsymbol{\theta} + h\theta_j \mathbf{e}_j) - P_i(\boldsymbol{\theta})}{h} \tag{8.6}$$

where $\mathbf{e}_j$ is used to change the scalar θ_j into a vector. This approximate derivative can be measured by slightly changing the input to the plant and measuring the changes in the output.

A somewhat similar approach is taken in (Kröse, Korst, & Groen, 1990) and (Smagt & Kröse, 1991). Again a two-layer feed-forward network is trained with back-propagation. However, instead of calculating a desired output vector the input vector which should have invoked the current output vector is reconstructed, and back-propagation is applied to this new input vector and the existing output vector.

The configuration used consists of a monocular manipulator which has to grasp objects. Due to the fact that the camera is situated in the hand of the robot, the task is to move the hand such that the object is in the centre of the image and has some predetermined size (in a later article, a biologically inspired system is proposed (Smagt, Kröse, & Groen, 1992) in which the visual flow-field is used to account for the monocularity of the system, such that the dimensions of the object need not to be known anymore to the system).

One step towards the target consists of the following operations:

1. measure the distance from the current position to the target position in camera domain, $\mathbf{x}$;
2. use this distance, together with the current state $\boldsymbol{\theta}$ of the robot, as input for the neural network. The network then generates a joint displacement vector $\Delta\boldsymbol{\theta}$;
3. send $\Delta\boldsymbol{\theta}$ to the manipulator;
4. again measure the distance from the current position to the target position in camera domain, $\mathbf{x}'$;
5. calculate the move made by the manipulator in visual domain, $\mathbf{x} - {}^{t+1}_{t}R\mathbf{x}'$, where ${}^{t+1}_{t}R$ is the rotation matrix of the second camera image with respect to the first camera image;

6. teach the learning pair $(\mathbf{x} - {}_t^{t+1}R\mathbf{x}', \theta; \Delta\theta)$ to the network.

This system has shown to learn correct behaviour in only tens of iterations, and to be very adaptive to changes in the sensor or manipulator (Smagt & Kröse, 1991; Smagt, Groen, & Kröse, 1993).

By using a feed-forward network, the available learning samples are approximated by a single, smooth function consisting of a summation of sigmoid functions. As mentioned in section 4, a feed-forward network with one layer of sigmoid units is capable of representing practically any function. But how are the optimal weights determined in finite time to obtain this optimal representation? Experiments have shown that, although a reasonable representation can be obtained in a short period of time, an accurate representation of the function that governs the learning samples is often not feasible or extremely difficult (Jansen et al., 1994). The reason for this is the global character of the approximation obtained with a feed-forward network with sigmoid units: every weight in the network has a *global* effect on the final approximation that is obtained.

Building local representations is the obvious way out: every part of the network is responsible for a small subspace of the total input space. Thus accuracy is obtained locally (Keep It Small & Simple). This is typically obtained with a Kohonen network.

Approach 2: Topology conserving maps

Ritter, Martinetz, and Schulten (Ritter, Martinetz, & Schulten, 1989) describe the use of a Kohonen-like network for robot control. We will only describe the kinematics part, since it is the most interesting and straightforward.

The system described by Ritter *et al.* consists of a robot manipulator with three degrees of freedom (orientation of the end-effector is not included) which has to grab objects in 3D-space. The system is observed by two fixed cameras which output their (x, y) coordinates of the object and the end effector (see figure 8.4).

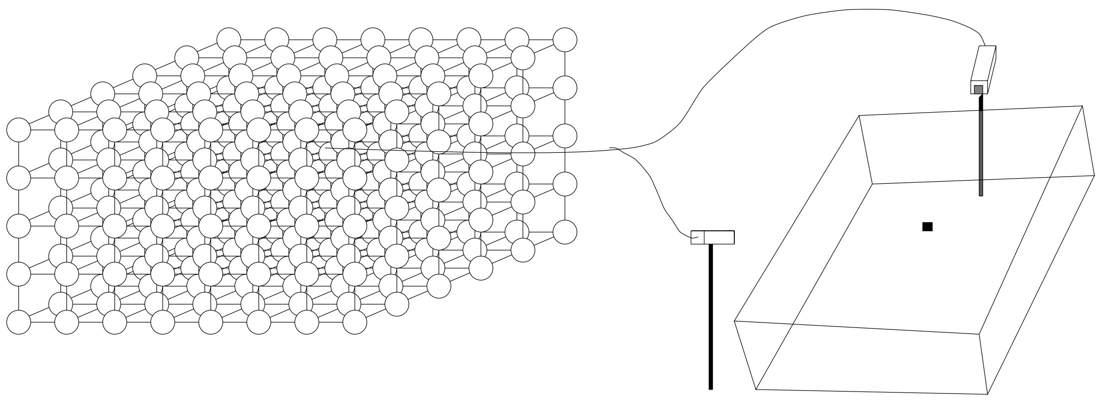

Figure 8.4: A Kohonen network merging the output of two cameras.

Each run consists of two movements. In the gross move, the observed location of the object $\mathbf{x}$ (a four-component vector) is input to the network. As with the Kohonen network, the neuron k with highest activation value is selected as winner, because its weight vector $\mathbf{w}_k$ is nearest to $\mathbf{x}$. The neurons, which are arranged in a 3-dimensional lattice, correspond in a $1-1$ fashion with subregions of the 3D workspace of the robot, i.e., the neuronal lattice is a *discrete representation* of the workspace. With each neuron a vector θ and Jacobian matrix A are associated. During gross move θ_k is fed to the robot which makes its move, resulting in retinal coordinates $\mathbf{x}_g$ of the end-effector. To correct for the discretisation of the working space, an additional move is made which is dependent of the distance between the neuron and the object in space $\mathbf{w}_k - \mathbf{x}$; this small displacement in Cartesian space is translated to an angle change using the Jacobian A_k:

$$\boldsymbol{\theta}^{\text{final}} = \boldsymbol{\theta}_k + A_k(\mathbf{x} - \mathbf{w}_k) \tag{8.7}$$

which is a first-order Taylor expansion of $\boldsymbol{\theta}^{\text{final}}$. The final retinal coordinates of the end-effector after this fine move are in $\mathbf{x}_f$.

Learning proceeds as follows: when an improved estimate $(\boldsymbol{\theta}, A)^*$ has been found, the following adaptations are made *for all neurons* j:

$$\begin{aligned} \mathbf{w}_j{}^{\text{new}} &= \mathbf{w}_j{}^{\text{old}} + \gamma(t)\, g_{jk}(t)\, \left(\mathbf{x} - \mathbf{w}_j{}^{\text{old}}\right), \\ (\boldsymbol{\theta}, A)_j^{\text{new}} &= (\boldsymbol{\theta}, A)_j^{\text{old}} + \gamma'(t)\, g'_{jk}(t)\, \left((\boldsymbol{\theta}, A)_j^* - (\boldsymbol{\theta}, A)_j^{\text{old}}\right). \end{aligned}$$

If $g_{jk}(t) = g'_{jk}(t) = \delta_{jk}$, this is similar to perceptron learning. Here, as with the Kohonen learning rule, a distance function is used such that $g_{jk}(t)$ and $g'_{jk}(t)$ are Gaussians depending on the distance between neurons j and k with a maximum at $j = k$ (cf. eq. (6.6)).

An improved estimate $(\boldsymbol{\theta}, A)^*$ is obtained as follows.

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_k + A_k(\mathbf{x} - \mathbf{x}_f), \tag{8.8}$$

$$\begin{aligned} A^* &= A_k + A_k(\mathbf{x} - \mathbf{w}_k - \mathbf{x}_f + \mathbf{x}_g) \times \frac{(\mathbf{x}_f - \mathbf{x}_g)^T}{\|\mathbf{x}_f - \mathbf{x}_g\|^2} \\ &= A_k + (\Delta\boldsymbol{\theta} - A_k \Delta\mathbf{x}) \frac{\Delta\mathbf{x}^T}{\|\Delta\mathbf{x}\|^2}. \end{aligned} \tag{8.9}$$

In eq. (8.8), the final error $\mathbf{x} - \mathbf{x}_f$ in Cartesian space is translated to an error in joint space via multiplication by A_k. This error is then added to $\boldsymbol{\theta}_k$ to constitute the improved estimate $\boldsymbol{\theta}^*$ (steepest descent minimisation of error).

In eq. (8.9), $\Delta\mathbf{x} = \mathbf{x}_f - \mathbf{x}_g$, i.e., the change in retinal coordinates of the end-effector due to the fine movement, and $\Delta\boldsymbol{\theta} = A_k(\mathbf{x} - \mathbf{w}_k)$, i.e., the related joint angles during fine movement. Thus eq. (8.9) can be recognised as an error-correction rule of the Widrow-Hoff type for Jacobians A.

It appears that after 6,000 iterations the system approaches correct behaviour, and that after 30,000 learning steps no noteworthy deviation is present.

8.2 Robot arm dynamics

While end-effector positioning via sensor–robot coordination is an important problem to solve, the robot itself will not move without dynamic control of its limbs.

Again, accurate control with non-adaptive controllers is possible only when accurate models of the robot are available, and the robot is not too susceptible to wear-and-tear. This requirement has led to the current-day robots that are used in many factories. But the application of neural networks in this field changes these requirements.

One of the first neural networks which succeeded in doing dynamic control of a robot arm was presented by Kawato, Furukawa, and Suzuki (Kawato, Furukawa, & Suzuki, 1987). They describe a neural network which generates motor commands from a desired trajectory in joint angles. Their system does not include the trajectory generation or the transformation of visual coordinates to body coordinates.

The network is extremely simple. In fact, the system is a feed-forward network, but by carefully choosing the basis functions, the network can be restricted to one learning layer such that finding the optimal is a trivial task. In this case, the basis functions are thus chosen that the function that is approximated is a linear combination of those basis functions. This approach is similar to that presented in section 4.5.

Dynamics model. The manipulator used consists of three joints as the manipulator in figure 8.1 without wrist joint. The desired trajectory $\boldsymbol{\theta}_d(t)$, which is generated by another subsystem, is fed into the inverse-dynamics model (figure 8.5). The error between $\boldsymbol{\theta}_d(t)$ and $\boldsymbol{\theta}(t)$ is fed into the neural model.

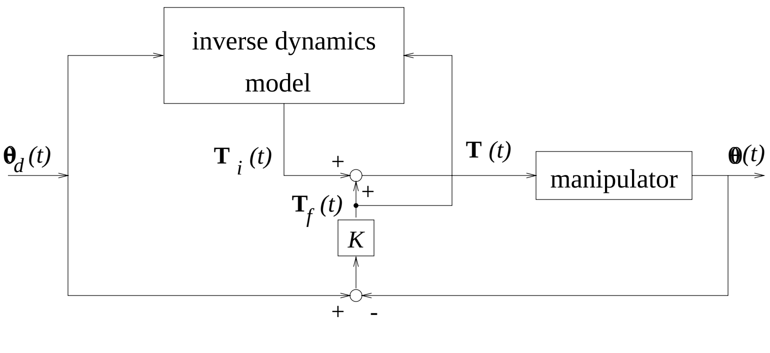

Figure 8.5: The neural model proposed by Kawato *et al.*

The neural model, which is shown in figure 8.6, consists of three perceptrons, each one feeding in one joint of the manipulator. The desired trajectory $\boldsymbol{\theta}_d = (\theta_{d1}, \theta_{d2}, \theta_{d3})$ is fed into 13 nonlinear subsystems. The resulting signals are weighted and summed, such that

$$T_{ik}(t) = \sum_{l=1}^{13} w_{lk} x_{lk}, \qquad (k = 1, 2, 3), \tag{8.10}$$

with

$$\begin{aligned} x_{l1} &= f_l(\theta_{d1}(t), \theta_{d2}(t), \theta_{d3}(t)), \\ x_{l2} = x_{l3} &= g_l(\theta_{d1}(t), \theta_{d2}(t), \theta_{d3}(t)), \end{aligned}$$

and f_l and g_l as in table 8.1.

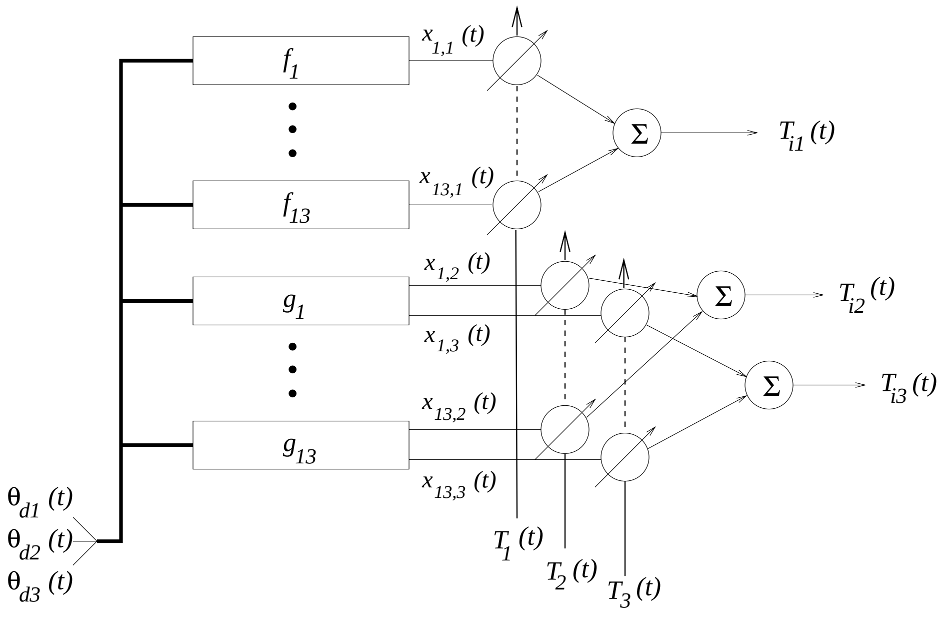

Figure 8.6: The neural network used by Kawato *et al.* There are three neurons, one per joint in the robot arm. Each neuron feeds from thirteen nonlinear subsystems. The upper neuron is connected to the rotary base joint (cf. joint 1 in figure 8.1), the other two neurons to joints 2 and 3.

l	$f_l(\theta_1, \theta_2, \theta_3)$	$g_l(\theta_1, \theta_2, \theta_3)$
1	$\ddot{\theta}_1$	$\ddot{\theta}_2$
2	$\ddot{\theta}_1 \sin^2 \theta_2$	$\ddot{\theta}_3$
3	$\ddot{\theta}_1 \cos^2 \theta_2$	$\ddot{\theta}_2 \cos \theta_3$
4	$\ddot{\theta}_1 \sin^2(\theta_2 + \theta_3)$	$\ddot{\theta}_3 \cos \theta_3$
5	$\ddot{\theta}_1 \cos^2(\theta_2 + \theta_3)$	$\dot{\theta}_1^2 \sin \theta_2 \cos \theta_2$
6	$\ddot{\theta}_1 \sin \theta_2 \sin(\theta_2 + \theta_3)$	$\dot{\theta}_1^2 \sin(\theta_2 + \theta_3) \cos(\theta_2 + \theta_3)$
7	$\dot{\theta}_1 \dot{\theta}_2 \sin \theta_2 \cos \theta_2$	$\dot{\theta}_1^2 \sin \theta_2 \cos(\theta_2 + \theta_3)$
8	$\dot{\theta}_1 \dot{\theta}_2 \sin(\theta_2 + \theta_3) \cos(\theta_2 + \theta_3)$	$\dot{\theta}_1^2 \cos \theta_2 \sin(\theta_2 + \theta_3)$
9	$\dot{\theta}_1 \dot{\theta}_2 \sin \theta_2 \cos(\theta_2 + \theta_3)$	$\dot{\theta}_2^2 \sin \theta_3$
10	$\dot{\theta}_1 \dot{\theta}_2 \cos \theta_2 \sin(\theta_2 + \theta_3)$	$\dot{\theta}_3^2 \sin \theta_3$
11	$\dot{\theta}_1 \dot{\theta}_3 \sin(\theta_2 + \theta_3) \cos(\theta_2 + \theta_3)$	$\dot{\theta}_2 \dot{\theta}_3 \sin \theta_3$
12	$\dot{\theta}_1 \dot{\theta}_3 \sin \theta_2 \cos(\theta_2 + \theta_3)$	$\dot{\theta}_2$
13	$\dot{\theta}_1$	$\dot{\theta}_3$

Table 8.1: Nonlinear transformations used in the Kawato model.

The feedback torque $\mathsf{T}_f(t)$ in figure 8.5 consists of

$$\begin{aligned} T_{fk}(t) &= K_{pk}(\theta_{dk}(t) - \theta_k(t)) + K_{vk}\frac{d\theta_k(t)}{dt}, \qquad (k = 1, 2, 3), \\ K_{vk} &= 0 \quad \text{unless } |\theta_k(t) - \theta_{dk}(\text{objective point})| < \varepsilon. \end{aligned}$$

The feedback gains K_p and K_v were computed as $(517.2, 746.0, 191.4)^T$ and $(16.2, 37.2, 8.4)^T$.

Next, the weights adapt using the delta rule

$$\gamma \frac{dw_{ik}}{dt} = x_{ik} T_1 = x_{ik}(T_{fk} - T_{ik}), \qquad (k = 1, 2, 3). \tag{8.11}$$

A desired move pattern is shown in figure 8.7. After 20 minutes of learning the feedback torques are nearly zero such that the system has successfully learned the transformation. Although the applied patterns are very dedicated, training with a repetitive pattern $\sin(\omega_k t)$, with $\omega_1 : \omega_2 : \omega_3 = 1 : \sqrt{2} : \sqrt{3}$ is also successful.

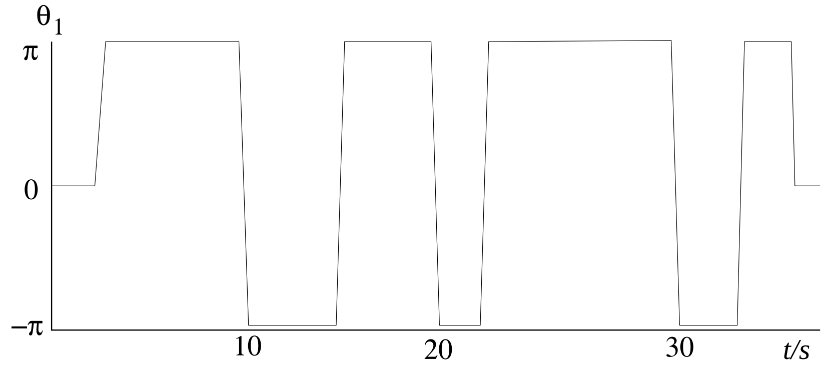

Figure 8.7: The desired joint pattern for joints 1. Joints 2 and 3 have similar time patterns.

The usefulness of neural algorithms is demonstrated by the fact that novel robot architectures, which no longer need a very rigid structure to simplify the controller, are now constructed. For example, several groups (Katayama & Kawato, 1992; Hesselroth, Sarkar, Smagt, & Schulten, 1994) report on work with a pneumatic musculo-skeletal robot arm, with rubber actuators replacing the DC motors. The very complex dynamics and environmental temperature dependency of this arm make the use of non-adaptive algorithms impossible, where neural networks succeed.

8.3 Mobile robots

In the previous sections some applications of neural networks on robot arms were discussed. In this section we focus on mobile robots. Basically, the control of a robot arm and the control of a mobile robot is very similar: the (hierarchical) controller first plans a path, the path is transformed from Cartesian (world) domain to the joint or wheel domain using the inverse kinematics of the system and finally a dynamic controller takes care of the mapping from set-points in this domain to actuator signals. However, in practice the problems with mobile robots occur more with path-planning and navigation than with the dynamics of the system. Two examples will be given.

8.3.1 Model based navigation

Jorgensen (Jorgensen, 1987) describes a neural approach for path-planning. Robot path-planning techniques can be divided into two categories. The first, called *local planning* relies on information available from the current 'viewpoint' of the robot. This planning is important, since it is able to deal with fast changes in the environment. Unfortunately, by itself local data is generally not adequate since occlusion in the line of sight can cause the robot to wander into dead end corridors or choose non-optimal routes of travel. The second situation is called *global path-planning*, in which case the system uses global knowledge from a topographic map previously stored into memory. Although global planning permits optimal paths to be generated, it has its weakness. Missing knowledge or incorrectly selected maps can invalidate a global path to an extent that it becomes useless. A possible third, 'anticipatory' planning combined both strategies: the local information is constantly used to give a best guess what the global environment may contain.

Jorgensen investigates two issues associated with neural network applications in unstructured or changing environments. First, can neural networks be used in conjunction with direct sensor readings to associatively approximate global terrain features not observable from a single robot perspective. Secondly, is a neural network fast enough to be useful in path relaxation planning, where the robot is required to optimise motion and situation sensitive constraints.

For the first problem, the system had to store a number of possible sensor maps of the environment. The robot was positioned in eight positions in each room and 180° sonar scans were made from each position. Based on these data, for each room a map was made. To be able to represent these maps in a neural network, the map was divided into 32×32 grid elements, which could be projected onto the 32×32 nodes neural network. The maps of the different rooms were 'stored' in a Hopfield type of network. In the operational phase, the robot wanders around, and enters an unknown room. It makes one scan with the sonar, which provides a partial representation of the room map (see figure 8.8). This pattern is clamped onto the network, which will regenerate the best fitting pattern. With this information a global path-planner can be used. The results which are presented in the paper are not very encouraging. With a network of 32×32 neurons, the total number of weights is 1024 squared, which costs more than 1 Mbyte of storage if only one byte per weight is used. Also the speed of the recall is low: Jorgensen mentions a recall time of more than two and a half hour on an IBM AT, which is used on board of the robot.

Also the use of a simulated annealing paradigm for path planning is not proving to be an effective approach. The large number of settling trials (> 1000) is far too slow for real time, when the same functions could be better served by the use of a potential field approach or distance transform.

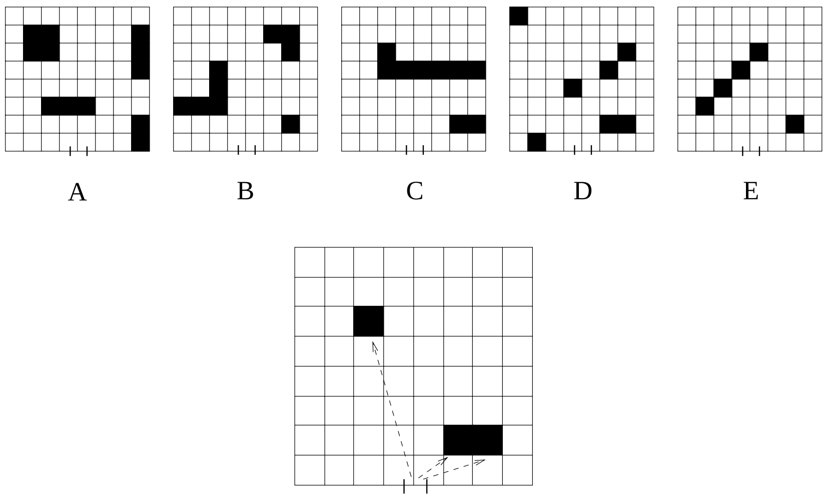

Figure 8.8: Schematic representation of the stored rooms, and the partial information which is available from a single sonar scan.

8.3.2 Sensor based control

Very similar to the sensor based control for the robot arm, as described in the previous sections, a mobile robot can be controlled directly using the sensor data. Such an application has been developed at Carnegy-Mellon by Touretzky and Pomerleau. The goal of their network is to drive a vehicle along a winding road. The network receives two type of sensor inputs from the sensory system. One is a 30×32 (see figure 8.9) pixel image from a camera mounted on the roof of the vehicle, where each pixel corresponds to an input unit of the network. The other input is an 8×32 pixel image from a laser range finder. The activation levels of units in the range finder's retina represent the distance to the corresponding objects.

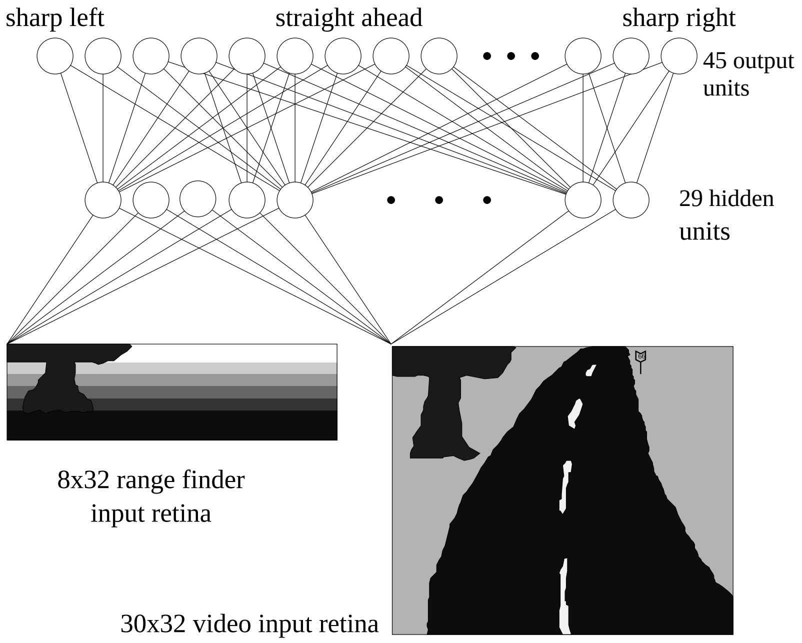

Figure 8.9: The structure of the network for the autonomous land vehicle.

The network was trained by presenting it samples with as inputs a wide variety of road images taken under different viewing angles and lighting conditions. 1,200 Images were presented, 40 times each while the weights were adjusted using the back-propagation principle. The authors claim that once the network is trained, the vehicle can accurately drive (at about 5 km/hour) along '... a path though a wooded area adjoining the Carnegie Mellon campus, under a variety of weather and lighting conditions.' The speed is nearly twice as high as a non-neural algorithm running on the same vehicle.

Although these results show that neural approaches can be possible solutions for the sensor based control problem, there still are serious shortcomings. In simulations in our own laboratory, we found that networks trained with examples which are provided by human operators are not always able to find a correct approximation of the human behaviour. This is the case if the human operator uses other information than the network's input to generate the steering signal. Also the learning of in particular back-propagation networks is dependent on the sequence of samples, and, for all supervised training methods, depends on the distribution of the training samples.