

# 29. NEURAL NETWORKS

Topics:

- Basic laws of motion
- Gravity, inertia, springs, dampers, cables and pulleys, drag, friction, FBDs
- System analysis techniques
- Design case

Objectives:

- To be able to develop differential equations that describe translating systems.

## I. Introduction

Over the past decade, the artificial intelligence community has undergone a resurgence of interest in the research and development of artificial neural networks. An artificial neural network is an attempt to simulate the manner in which the brain interprets information as determined by the current knowledge of biology, physiology, and psychology [14]. Artificial neural networks behave in much the same manner as biological neural networks, giving many of the same benefits. Artificial neural nets are fault tolerant, exhibit the ability to learn and adapt to new situations, and have the ability to generalize based on a limited set of data. This arises because of the structure which allows neural nets to process information simultaneously, as opposed to the serial nature of traditional digital computers. This parallel nature, inherent in neural networks, achieves the increase in speed by distributing the calculations among many neurons. The network structure provides a level of fault tolerance, which allows the network to withstand component failures without having the entire network fail.

This paper focuses on a popular feedforward model of neural networks. In this model a set of inputs are applied to the network, and multiplied by a set of connection weights. All of the weighted inputs to the neuron are then summed and an activation function is applied to the summed value. This activation level becomes the neuron's output and can be either an input for other neurons, or an output for the network. Learning in this network is done by adjusting the connection weights based upon training vectors (input and corresponding desired output). When a training vector is presented to a neural net, the connection weights are adjusted to minimize the difference between the desired and actual output. After a network is trained with a set of training vectors, the network should produce a good output match for the inputs.

Artificial neural networks are mainly used in two areas --- *pattern recognition*, and *pattern matching*. *Pattern recognition* is performed by classifying an unknown pattern through comparisons with previously learned pat-

terns. This ability is termed associative recall. An example of this is the use of neural networks to recognize hand-written digits [4]. In pattern recognition, when a particular pattern is noisy or distorted, the network can generalize and choose the closest match [3][4][10][13]. *Pattern matching* uses continuous input patterns to evoke continuous output patterns (i.e. one-to-one mappings). An example is the use of a neural network as a basic controller for a plant. The controller would accept the plant conditions as the inputs, and a set of control outputs would drive the current manufacturing process. This approach is discussed in more detail in [11]. This paper uses the pattern matching strategy for the inverse kinematics of a typical three link manipulator.

This paper focuses on the kinematic control of a three link manipulator working in three dimensional space, within a quarter of the robot workspace. When manipulators interact with the environment, the position of the end effector must be described. Human users are more suited to working in cartesian coordinates than the joint coordinates of robots; hence, there is a need for conversion between the two coordinate systems. The conversion from world coordinates to robot joint angles is called *inverse kinematics*. In the past, the problem of solving the inverse kinematics has been done through the application of various methods. These methods take one of two basic forms (i) *Closed form* equations or, (ii) *Iterative*. A *closed form* solution is found by decoupling the joints responsible for position and those responsible for orientation. These solution types are more desirable since explicit equations are available. *Iterative techniques*, such as, Newton-Raphson, Newton-Gauss and Linearization are available when no closed form solutions can be obtained. Many of these solutions require that the kinematics be performed off-line because of the computational demands of the algorithms. A brief survey of these various algorithms is given in [2] and [15]. Both of the forms are generally rigid and do not account for uncontrollable variables such as manufacture tolerances, calibration error, and wear. A neural network approach could adapt to changes in the robot due to wear from extended use over time. Such algebraic and algorithmic procedures cannot incorporate these situations.

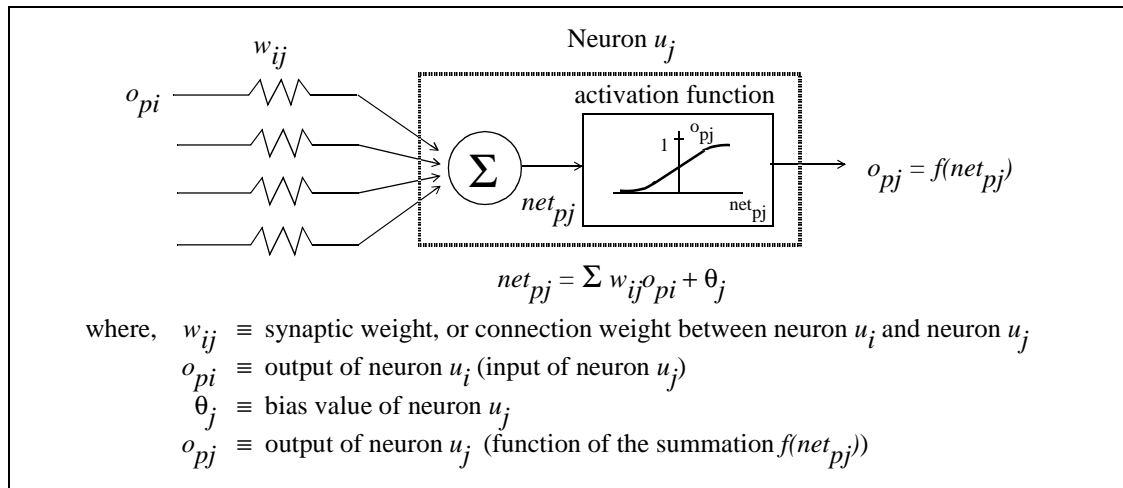
Previous research in the application of neural networks to kinematic control was performed for a to a two-link planar manipulator (working in a plane), and a five degree of freedom robot working in three dimensional space. The manipulators were controlled with a high accuracy; however, the training region for the two link robot was a small square in the centre of the workspace, and the five link robot was limited to a small wedge volume within the robot workspace [5][6][7]. This paper explains research extended to investigate the performance throughout the entire work space of the robot. Investigations of a more complete representation of inverse kinematics throughout the entire workspace is provided.

The paper begins with a general introduction to the artificial neuron, a brief description of feedforward neural networks, and a description of the backpropagation learning algorithm. The inverse kinematics problem is described, and the application of a two-layer neural network is discussed. The use of compensation networks is explored and the results of the experimental work are presented. The implications of the neural network approach to the kinematic control problem are presented at the conclusion of this paper.

## II. The Basic Artificial Neuron

The basic building block of an artificial neural network is the neuron. In a popular model which will be

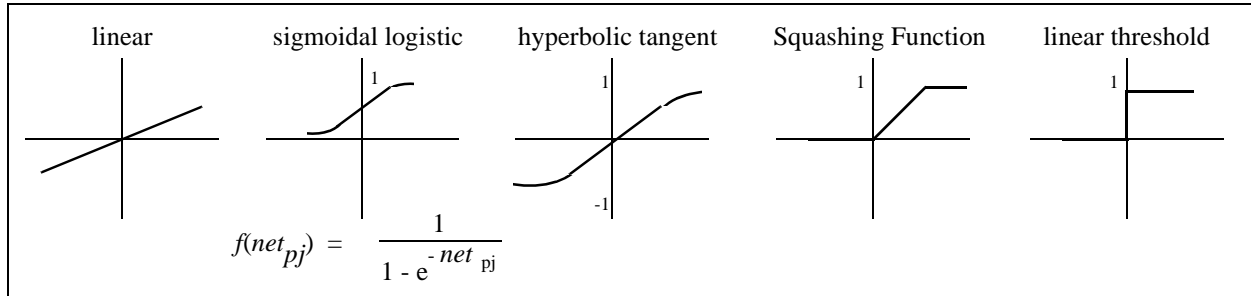
used in this paper, the connection weights between neurons are adjusted. The neuron receives inputs  $o_{pi}$  from neuron  $u_i$  while the network is exposed to input pattern  $p$ . Each input is multiplied by a connection weight  $w_{ij}$ , where  $w_{ij}$  is the connection between neurons  $u_i$  and  $u_j$ . The connection weights correspond to the strength of the influence of each of the preceding neurons. After the inputs have been multiplied by the connection weights for input pattern  $p$ , their values are summed,  $net_{pj}$ . Included in the summation is a bias value  $\theta_j$  to offset the basic level of the input to the activation function,  $f(net_{pj})$ , which gives the output  $o_{pj}$ . Figure 1 shows the structure of the basic neuron.



**Figure 1** Basic structure of an artificial neuron.

In order to establish a bias value  $\theta_j$ , the bias term can appear as an input from a separate neuron with a fixed value (a value of +1 is common). Each neuron requiring a bias value will be connected to the same bias neuron. The bias values are then self-adjusted as the other neurons learn, without the need for extra considerations.

In calculating the output of the neuron, the activation function may be in the form of a threshold function, in which the output of the neuron is +1 if a threshold level is reached and 0 otherwise. Squashing functions limit the linear output between a maximum and minimum value. These linear functions, however, do not take advantage of multi-layer networks [14]. Hyperbolic tangents and the sigmoidal functions are similar to real neural responses; however, the hyperbolic tangent is unbounded and hard to implement in hardware. In this paper, the Sigmoidal function is used because of its ability to produce continuous non-linear functions, which can be implemented in hardware in future research areas. Figure 2 shows some commonly used activation functions.



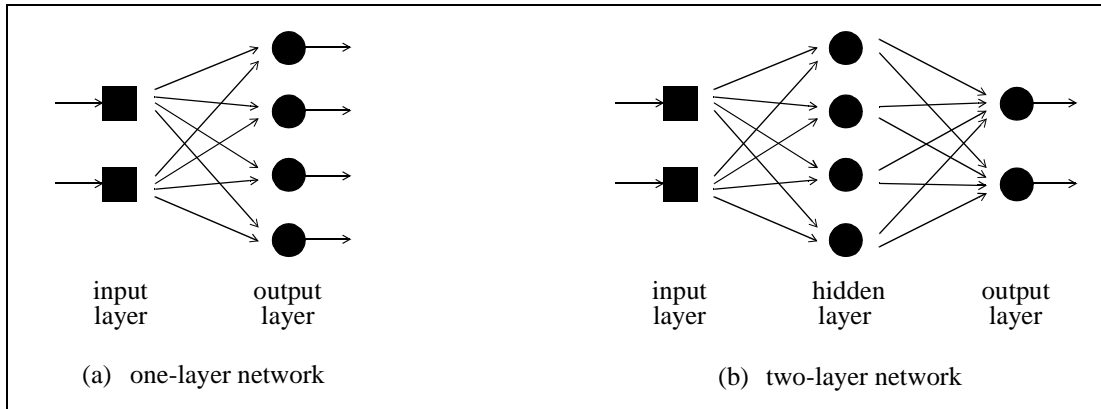
**Figure 2** Common activation functions.

For more in depth information on neural networks, please refer to introductions to the subject in [3], [8], [14] and [16].

### III. Neural Network Architectures in Feedforward Systems

A single neuron can only simulate 14 of the 16 basic boolean logic functions. It cannot emulate the X-OR, or the X-NOR gates [9]. These limitations require that more than a single neuron be used, and thus, the architecture becomes an important consideration. In Feedforward systems, networks propagate the inputs forward through the neural net. Each neuron has inputs that only come from neurons in preceding layers. A one-layer neural net consists of a layer of input neurons and a layer of output neurons. A multi-layer network consists of an input layer of neurons, one or more hidden layers, and an output layer.

The circular nodes in figure 3 represent basic neurons, as described in the previous section. The input neurons are shown as squares because they only act as terminal points (i.e.  $o_{pi}$  = input). The input layer does not process information; thus, it is not considered to be a part of the structure and is numbered layer 0. A simple one-layer network can effectively map many sets of inputs to outputs. In **pattern recognition** problems, this depends upon the linear separability of the problem domain. In **pattern matching** problems, this depends upon continuity, and topography of the function. If such a set of connection weights cannot be found using one-layered networks, then multi-layered networks must be considered.



**Figure 3** Simple feedforward artificial neural networks.

## IV. Backpropagation of Errors

Learning methods may be either supervised, or unsupervised. Supervised learning is required for pattern matching, and backpropagation is the most popular of the supervised learning techniques. In the backpropagation Training Paradigm, the connection weights are adjusted to reduce the output error. In the initial state, the network has a random set of connection weights. When a system starts with all connection weights equal, the network begins at a sort of local optimum, and will not converge to the global solution. In order for the network to learn, a set of inputs are presented to the system and a set of outputs are calculated. A difference between the actual outputs and desired outputs is calculated and the connection weights are modified to reduce this difference [14].

After inputs have been applied and the network output solution has been calculated, the estimated error contribution,  $\delta$ , by each neuron must be calculated. The calculations begin at the output layer of the network. The delta value for any output neuron is computed as,

$$\delta_{pj} = (t_{pj} - o_{pj})f'(net_{pj}) \quad (1)$$

where,  $t_{pj}$  is the desired output of output neuron  $u_j$ , and  $o_{pj}$  is the actual output, and  $f'(net_{pj})$  is the first derivative of the activation function with respect to the total input ( $net_{pj}$ ) for the given input pattern  $p$  evaluated at neuron  $u_j$ .

For the output layer, the change in connection weights can easily be calculated since the  $\delta$  of the output neurons is easily determined. With the introduction of hidden layers, the desired outputs of these hidden neurons becomes more difficult to estimate. In order to estimate the delta of a hidden neuron, the error signal from the output

layer must be propagated backwards to all preceding layers. The delta,  $\delta_{pj}$ , for the hidden neurons is calculated as,

$$\delta_{pj} = f'(net_{pj}) \sum_k \delta_{pk} w_{jk} \quad \text{-----} \quad (2)$$

where,  $f'(net_{pj})$  is the first derivative of the activation function with respect to  $net_{pj}$  at hidden neuron  $u_j$ ;  $\delta_{pk}$  is the delta value for the subsequent neuron  $u_k$ , and  $w_{jk}$  is the connection weight for the link between hidden neuron  $u_j$  and subsequent neuron  $u_k$ . This process of calculating the  $\delta$ s is performed for all layers until the input layer is reached

After all the error terms have been calculated for all the neurons, the weights may be adjusted. The estimated weight change  $\Delta_p w_{ij}(n+1)$  is calculated for each input connection to neuron  $u_j$  from neuron  $u_i$  by,

$$\Delta_p w_{ij}(n+1) = \underbrace{\eta}_{\text{new delta}} \delta_{pj} o_{pi} + \underbrace{\alpha}_{\text{momentum term}} \Delta w_{ij}(n) \quad \text{-----} \quad (3)$$

where  $\eta$  is the learning rate,  $\alpha$  is the smoothing term,  $\delta_{pj}$  is the delta of neuron  $u_j$ ,  $o_{pi}$  is the output of preceding neuron  $u_i$ , and  $\Delta w_{ij}(n)$  is the weight change in the previous training interval. The estimated errors are used to estimate the weight changes with the least mean squares method.

The backpropagation method is essentially a gradient descent approach which minimizes the errors. In order to adjust the connection weights, the gradient descent approach requires that steps be taken in weight space. If the steps are too large, then the weights will overshoot their optimum values. If the steps are too small, it may take longer to converge and may become caught in local minima. The step size is then a function of the learning rate. The learning rate, which varies between 0 and 1, should be large enough to come up with a solution in a reasonable amount of time without having the solution oscillate within weight space. By introducing the *momentum* term, which varies between 0 and 1, the learning rate can be increased while preventing oscillation. The momentum term is introduced by adding to the weight change some percentage of the last weight change. Common values for  $\eta$  range from 0.1 to 0.9, and common values of  $\alpha$  range from 0.2 to 0.8

The training set is comprised of points which have inputs and desired outputs, which may or may not be in a random order. In this paper, the training points were chosen at random from an ordered table. This allows a weight correction after each point is trained. This method helps avoid local optimum points in weight space, by creating a random walk type of weight optimization, and gives a much faster global convergence. This process is then repeatedly performed for all training examples until a satisfactory output error is achieved. For an in-depth derivation of the back propagation method, the reader is suggested to consult Rumelhart and McClelland [14].

## V. A Typical Three Link Manipulator

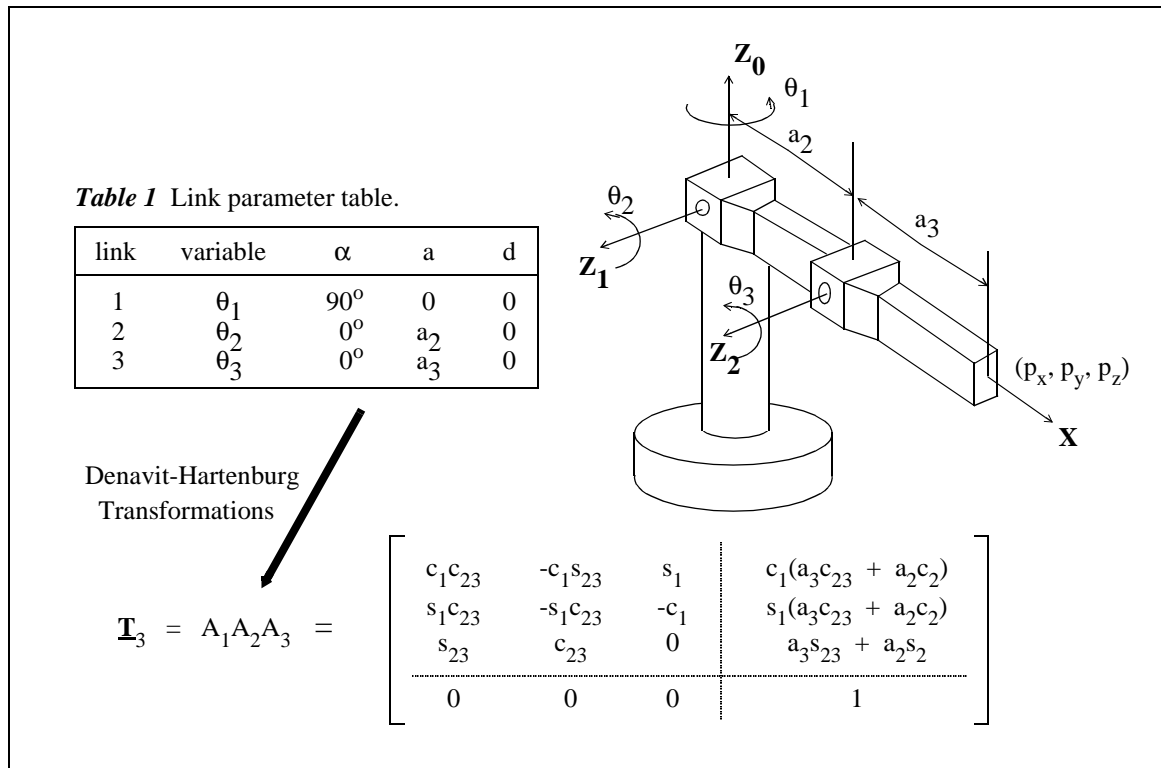
In order for a robotic manipulator to perform tasks within space, a user must specify a location in three dimensional space. The robotic controller must then determine the correct joint coordinates to locate the manipulator. This area of robotics, Inverse Kinematics, has been well researched and many good solutions exist. In all cases the

---

\* note: The subscript notation  $(i, j, k)$  means that neuron  $u_i$  precedes neuron  $u_j$ , which precedes  $u_k$ , etc.

resultant solutions are highly specific to a particular robot configuration, with exact dimensions. Such explicit solutions do not tolerate changes over time. These changes may be caused by poor tolerances in manufacture, wear over extended periods of operation, damage to the robot, and poor calibration. Computations are often complex, can be quite slow, and require a computer capable of performing complex mathematical functions.

An example of a robot which may be used with an artificial neural network control system is the three link manipulator as shown in figure 4. The forward and inverse kinematic equations, used by the authors in testing, are given in the equations (4) and (5).



**Figure 4** A typical three link manipulator.

The  $\pm$  sign in  $\theta_2$  indicates the general configuration of the robot arm. The positive sign (+ve) corresponds to an elbow down solution, and the negative sign (-ve) corresponds to an elbow up solution.

$$\left. \begin{aligned} p_x &= c_1(a_3c_{23} + a_2c_2) \\ p_y &= s_1(a_3c_{23} + a_2c_2) \\ p_z &= a_3s_{23} + a_2s_2 \end{aligned} \right\} \text{forward kinematic equations} \quad (4)$$

$$\left. \begin{aligned} \theta_1 &= \tan^{-1} \left[ \frac{p_y}{p_x} \right] \\ \theta_2 &= \tan^{-1} \left[ \frac{\beta}{\pm \sqrt{p_z^2 + \alpha^2 - \beta^2}} \right] - \tan^{-1} \left[ \frac{\alpha}{p_z} \right] \\ \theta_3 &= \tan^{-1} \left[ \frac{p_z - a_2s_2}{\alpha - a_2c_2} \right] - \theta_2 \end{aligned} \right\} \text{inverse kinematic equations} \quad (5)$$

$$\begin{aligned} \text{where} \quad & s_1 = \sin \theta_1 \\ & c_1 = \cos \theta_1 \\ & c_{12} = \cos (\theta_1 + \theta_2) \\ & s_{12} = \sin (\theta_1 + \theta_2) \end{aligned} \quad \text{and} \quad \begin{aligned} \alpha &= + \sqrt{p_x^2 + p_y^2} \\ \beta &= \frac{\alpha^2 + p_z^2 + a_2^2 - a_3^2}{2a_2} \end{aligned}$$

It should be noted that the inverse kinematics is not determinate. The sources of indeterminacy for this manipulator occur:

- for Elbow up and Elbow Down Configurations, and
- when the end of the arm is directly over the origin,  $\theta_1$  is undefined.

Other examples of indeterminacy occur in PUMA style robots where left and right arm configurations would give indeterminacy. These sources of indeterminacy may all be overcome by constraining the inverse kinematics to a particular case.

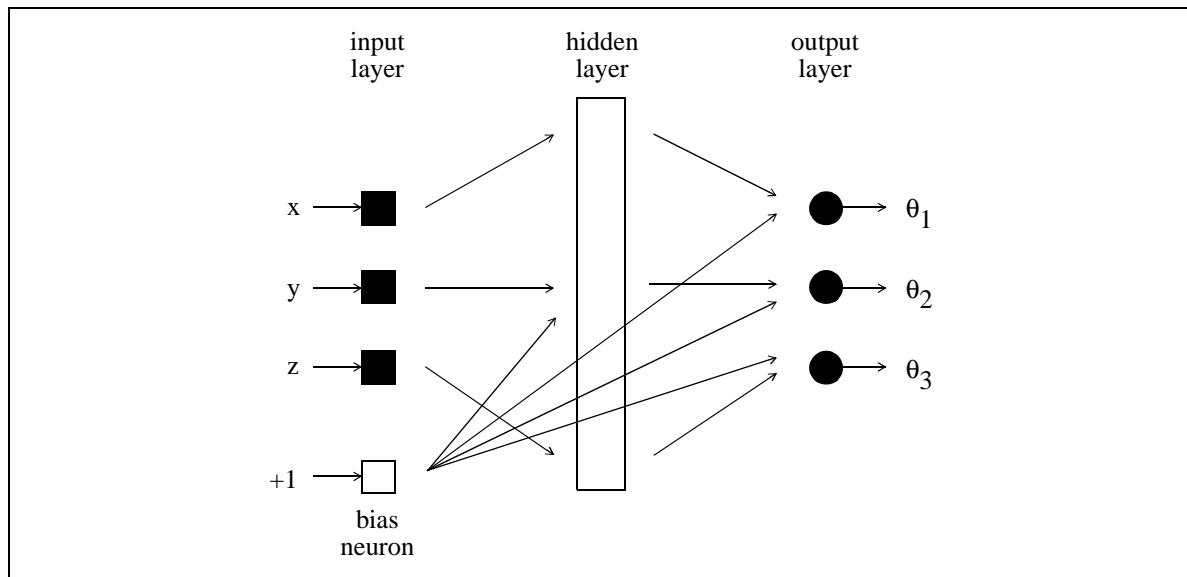
## VI. Training a Neural Network for Inverse Kinematics

Neural Network solutions have the benefit of having faster processing times since information is processed in parallel. The solutions may be adaptive and still be implemented in hardware by using specialized electronics. Neural systems can generalize to approximate solutions from small training sets. Neural systems are fault tolerant and robust. The network will not fail if a few neurons are damaged, and the solutions may still retain accuracy. When implementing a neural network approach, complex computers are not essential and robot controllers need not be spe-

cific to any one manipulator.

The nature of neural networks require that a set of training points be chosen which represent the nature of the inputs ( $x$ ,  $y$ ,  $z$ ) and the corresponding outputs ( $\theta_1$ ,  $\theta_2$  and  $\theta_3$ ). If the training points chosen tend to be clustered, then the network will be very accurate when dealing with points near the cluster. In this case, the robot should be familiar with points throughout the entire workspace; thus, points should be evenly distributed. The order in which points are presented to the network also affect the speed and quality of convergence. If the points are not presented in a random order, each training update will tend to train the network for the current section of space which the points are from.

The neural network architecture chosen for this problem is a simple network with one hidden layer. The network has three input neurons for the desired position, and three output neurons for the estimated joint angles. A bias neuron (with a value of +1) is attached to all of the neurons in the hidden layer, and to all of the output neurons. The number of neurons in the hidden layer varied from 10, 20, and 40. This network architecture is pictured in figure 5.



**Figure 5** Neural architecture for solving the inverse kinematic problem. For simplicity, the network is shown with a small amount of arrows; however, the network is actually fully connected.

## 29.1 SUMMARY

•

## **29.2 PRACTICE PROBLEMS**

## **29.3 PRACTICE PROBLEM SOLUTIONS**

## **29.4 ASSIGNMENT PROBLEMS**

## **29.5 REFERENCES**



