Study on the use of neural networks in control systems

F. Rinaldi
Politecnico di Torino & Italian Air Force, Italy

Keywords: neural networks, automatic control

Abstract

The purpose of this report is to provide a quick overview of neural networks and to explain how they can be used in control systems. We introduce the multilayer perceptron neural network and describe how it can be used for function approximation. The backpropagation algorithm (including its variations) is the principal procedure for training multilayer perceptrons; it is briefly described here. Care must be taken, when training perceptron networks, to ensure that they do not overfit the training data and then fail to generalize well in new situations. Several techniques for improving generalization are discussed. The report also presents the model reference adaptive control. We demonstrate the practical implementation of this controller on the automatic altitude control of a quad-rotor system. Finally various sensors setups are conceived and their implications on the control law are evaluated.

1 Introduction

In this report we want to give a brief introduction to neural networks and their application in control systems. The report is written for readers who are not familiar with neural networks but are curious about how they can be applied to practical control problems. The field of neural networks covers a very broad area. It is not possible in this report to discuss all types of neural networks. Instead, we will concentrate on the most common neural network architecture, the multilayer perceptron. We will describe the basics of this architecture, discuss its capabilities and show how it has been used in several different control system configurations.

For the purposes of this report we will look at neural networks as function approximators. As shown in Fig. 1 we have some unknown function that we wish to approximate. We want to adjust the parameters of the network so that it will produce the same response as the unknown function, if the same input is applied to both systems. For our applications, the unknown function may correspond to a system we are trying to control, in which case the neural network will be the identified plant model. The unknown function could also represent the inverse of a system we are trying to control, in which case the neural network can be used to implement the controller. In the next section we will present the multilayer perceptron neural network, and will demonstrate how it can be used as a function approximator.

Figure 1: Neural network as function approximator
2 Multilayer perceptron architecture

2.1 Neuron model

The multilayer perceptron neural network is built up of simple components. We will begin with a single-input neuron, which we will then extend to multiple inputs. We will next stack these neurons together to produce layers. Finally, we will cascade the layers together to form the network.

A single-input neuron is shown in Fig. 2. The scalar input $p$ is multiplied by the scalar weight $w$ to form $wp$, one of the terms that is sent to the summer. The other input, 1, is multiplied by a bias $b$ and then passed to the summer. The summer output $n$, often referred to as the net input, goes into a transfer function $f$, which produces the scalar neuron output $a$.

$$a = f(wp + b) \quad (1)$$

Note that $w$ and $b$ are both adjustable scalar parameters of the neuron. Typically the transfer function is chosen by the designer, and then the parameters $w$ and $b$ are adjusted by some learning rule so that the neuron input/output relationship meets some specific goal. The transfer function in Fig. 2 may be a linear or a nonlinear function of $n$. One of the most commonly used functions is the log-sigmoid transfer function, which is shown in Fig. 3.

![Figure 3: Log-sigmoid transfer function](image)

This transfer function takes the input (which may have any value between plus and minus infinity) and squashes the output into the range 0 to 1, according to the expression

$$a = \frac{1}{1 + e^{-n}} \quad (2)$$

The log-sigmoid transfer function is commonly used in multilayer networks that are trained using the backpropagation algorithm, in part because this function is differentiable. Typically, a neuron has more than one input. A neuron with $R$ inputs is shown in Fig. 4. The individual inputs $p_1, p_2, ..., p_R$ are each one weighted by corresponding elements $w_{1,1}, w_{1,2}, ..., w_{1,R}$ of the weight matrix $W$.

$$n = w_{1,1}p_1 + w_{1,2}p_2 + w_{1,R}p_R + b \quad (3)$$

This expression can be written in matrix form as

$$n = Wp + b \quad (4)$$
where the matrix $W$ for the single neuron case has only one row.

Now the neuron output can be written as

$$a = f(Wp + b)$$ (5)

Figure 5 represents the neuron in matrix form.

![Figure 5: Neuron with $R$ inputs, matrix notation](image)

**2.2 Network architecture**

Commonly one neuron, even with many inputs, is not sufficient. We might need five or ten, operating in parallel, in what is called a layer. A single-layer network of $S$ neurons is shown in Fig. 6. Note that each of the $R$ inputs is connected to each of the neurons and that the weight matrix now has $S$ rows. The layer includes the weight matrix $W$, the summers, the bias vector $b$, the transfer function boxes and the output vector $a$. Some authors refer to the inputs as another layer, but we will not do that here. It is common for the number of inputs to a layer to be different from the number of neurons (i.e., $R \neq S$).

The $S$-neuron, $R$-input, one-layer network also can be drawn in matrix notation, as shown in Fig. 7.

**2.2.1 Multiple layers of neurons**

Now consider a network with several layers. Each layer has its own weight matrix $W$, its own bias vector $b$, a net input vector $n$ and an output vector $a$. We need to introduce some additional notation to distinguish between these layers. We will use superscripts to identify the

![Figure 6: Layer of $S$ neurons](image)

![Figure 7: Layer of $S$ neurons, matrix notation](image)
layers. Thus, the weight matrix for the first layer is written as $W^1$, and the weight matrix for the second layer is written as $W^2$. This notation is used in the three-layer network shown in Fig. 8. As shown, there are $R$ inputs, $S^1$ neurons in the first layer, $S^2$ neurons in the second layer, etc. As noted, different layers can have different numbers of neurons.

The outputs of layers one and two are the inputs for layers two and three. Thus layer 2 can be viewed as a one-layer network with $R = S^1$ inputs, $S = S^2$ neurons, and an $S^1 \times S^2$ weight matrix $W^2$. The input to layer 2 is $a^1$, and the output is $a^2$. A layer whose output is the network output is called an output layer. The other layers are called hidden layers. The network shown in Fig. 8 has an output layer (layer 3) and two hidden layers (layers 1 and 2).

### 3 Approximation capabilities of multi-layer networks

Two-layer networks, with sigmoid transfer functions in the hidden layer and linear transfer functions in the output layer, are universal approximators [1]. A simple example can demonstrate the power of this network for approximation. Consider the two-layer, 1-2-1 network shown in Fig. 9. For this example the transfer function for the first layer is log-sigmoid and the transfer function for the second layer is linear. In other words,

$$f^1(n) = \frac{1}{1 + e^{-n}} \quad \text{and} \quad f^2(n) = n \quad (6)$$

Suppose that the nominal values of the weights and biases for this network are

$$w^1_{1,1} = 10$$
$$w^1_{2,1} = 10$$
$$b^1_1 = -10$$
$$b^1_2 = 10$$
$$w^2_{1,1} = 1$$
$$w^2_{1,2} = 1$$
$$b^2_2 = 0$$

The network response for these parameters is shown in Fig. 10, which plots the network output $a^2$ as the input $p$ is varied over the range $[-2, 2]$.

Notice that the response consists of two steps, one for each of the log-sigmoid neurons in the first layer. By adjusting the network parameters we can change the shape and location of each step, as we will see in the following discussion. The centers of the steps occur where the net input to a neuron in the first layer is zero:

$$n^1_1 = w^1_{1,1}p + b^1_1 = 0 \Rightarrow p = -\frac{b^1_1}{w^1_{1,1}} = -\frac{-10}{10} = 1 \quad (7)$$

$$n^1_2 = w^1_{2,1}p + b^1_2 = 0 \Rightarrow p = -\frac{b^1_2}{w^1_{2,1}} = -\frac{10}{10} = -1 \quad (8)$$

The steepness of each step can be adjusted by changing the network weights. Figure 11 illustrates the effects of parameter changes on the network response. The nominal response is repeated from Fig. 10. The other curves correspond to the network response when one parameter at a time is varied over the following ranges:

$$-1 \leq w^2_{1,1} \leq 1$$
$$-1 \leq w^2_{1,2} \leq 1$$
$$0 \leq b^2_2 \leq 20$$
$$-1 \leq b^2_1 \leq 1$$

Figure 11(a) shows how the network biases in the first (hidden) layer can be used to locate the position of the steps. Fig. 11(b) and Fig. 11(c) illustrate how the weights determine the slope of the steps. The bias in the second (output) layer

![Diagram of a network](image)
shifts the entire network response up or down, as can be seen in Fig. 11(d).

Figure 11: Effect of parameter changes on network response

From this example we can see how flexible the multilayer network is. It would appear that we could use such networks to approximate almost any function, if we had a sufficient number of neurons in the hidden layer. In fact, it has been shown that two-layer networks, with sigmoid transfer functions in the hidden layer and linear transfer functions in the output layer, can approximate virtually any function of interest to any degree of accuracy, provided sufficiently many hidden units are available. It is beyond the scope of this report to provide detailed discussions of approximation theory, but there are
many papers in the literature that can provide a deeper discussion of this field. In [1], Hornik, Stinchcombe and White present a proof that multilayer perceptron networks are universal approximators. Pinkus gives a more recent review of the approximation capabilities of neural networks in [2]. Niyogi and Girosi, in [3], develop bounds on function approximation error when the network is trained on noisy data.

4 Training multilayer networks

Now that we know multilayer networks are universal approximators, the next step is to determine a procedure for selecting the network parameters (weights and biases) that will best approximate a given function. The procedure for selecting the parameters for a given problem is called training the network [8]. In this section we will outline a training procedure called backpropagation, which is based on gradient descent (more efficient algorithms than gradient descent exist and are often used in neural network training). As we discussed earlier, for multilayer networks the output of one layer becomes the input to the following layer (see Fig. 8). The equations that describe this operation are

\[ a_{m+1} = f_{m+1}(W_{m+1}a + b_{m+1}) \]

for \( m = 0, 1, ..., M - 1 \) (9)

where \( M \) is the number of layers in the network. The neurons in the first layer receive external inputs:

\[ a^0 = p \] (10)

which provides the starting point for Eq. (9). The equations that describe this operation are

\[ a^0 = p \] (10)

The outputs of the neurons in the last layer are considered the network outputs:

\[ a = a^M \] (11)

4.1 Performance index

The backpropagation algorithm for multilayer networks is a gradient descent optimization procedure in which we minimize a mean square error performance index. The algorithm is provided with a set of examples of proper network behavior:

\( \{p_1, t_1\}, \{p_2, t_2\}, \ldots, \{p_Q, t_Q\} \) (12)

where \( p_Q \) is an input to the network, and \( t_Q \) is the corresponding target output. As each input is applied to the network, the network output is compared to the target. The algorithm should adjust the network parameters in order to minimize the sum squared error:

\[ F(x) = \sum_{q=1}^{Q} e_q^2 = \sum_{q=1}^{Q} (t_q - a_q)^2 \] (13)

where \( x \) is a vector containing all network weights and biases. If the network has multiple outputs this generalizes to

\[ F(x) = \sum_{q=1}^{Q} e_q^T e_q = \sum_{q=1}^{Q} (t_q - a_q)^T (t_q - a_q) \] (14)

Using a stochastic approximation, we will replace the sum squared error by the error on the latest target:

\[ \hat{F}(x) = (t(k) - a(k))^T (t(k) - a(k)) = e^T(k) e(k) \] (15)

where the expectation of the squared error has been replaced by the squared error at iteration \( k \).

The steepest descent algorithm for the approximate mean square error is

\[ w_{i,j}^m(k + 1) = w_{i,j}^m(k) - \alpha \frac{\partial \hat{F}}{\partial w_{i,j}^m} \] (16)

\[ b_i^m(k + 1) = b_i^m(k) - \alpha \frac{\partial \hat{F}}{\partial b_i^m} \] (17)

where \( \alpha \) is the learning rate.

4.2 Chain rule

For a single-layer linear network, these partial derivatives in Eq. (16) and Eq. (17) are conveniently computed, since the error can be written as an explicit linear function of the network weights. For the multilayer network, the error is not an explicit function of the weights in the
hidden layers, therefore these derivatives are not computed so easily. Because the error is an indirect function of the weights in the hidden layers, we will use the chain rule of calculus to calculate the derivatives in Eq. (16) and Eq. (17):

\[
\frac{\partial \hat{F}}{\partial w_{i,j}^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial w_{i,j}^m} \tag{18}
\]

\[
\frac{\partial \hat{F}}{\partial b_i^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial b_i^m} \tag{19}
\]

The second term in each of these equations can be easily computed, since the net input to layer \( m \) is an explicit function of the weights and bias in that layer:

\[
n_i^m = \sum_{j=1}^{n_i^{m-1}} w_{i,j}^m a_j^{m-1} + b_i^m \tag{20}
\]

Therefore

\[
\frac{\partial n_i^m}{\partial w_{i,j}^m} = a_j^{m-1} \tag{21}
\]

\[
\frac{\partial n_i^m}{\partial b_i^m} = 1
\]

If we now define

\[
s_i^m \equiv \frac{\partial \hat{F}}{\partial n_i^m} \tag{22}
\]

(the sensitivity of \( \hat{F} \) to changes in the \( i \)-th element of the net input at layer \( m \)), then Eq. (18) and Eq. (19) can be simplified to

\[
\frac{\partial \hat{F}}{\partial w_{i,j}^m} = s_i^m a_j^{m-1} \tag{23}
\]

\[
\frac{\partial \hat{F}}{\partial b_i^m} = s_i^m \tag{24}
\]

We can now express the approximate steepest descent algorithm as

\[
w_{i,j}^m(k + 1) = w_{i,j}^m(k) - \alpha s_i^m a_j^{m-1} \tag{25}
\]

\[
b_i^m(k + 1) = b_i^m(k) - \alpha s_i^m \tag{26}
\]

In matrix form this becomes:

\[
W^m(k + 1) = W^m(k) - \alpha s^m(a^{m-1})^T \tag{27}
\]

\[
b^m(k + 1) = b^m(k) - \alpha s^m \tag{28}
\]

where the individual elements of \( s^m \) are given by Eq. (22).

4.3 Backpropagating the sensitivities

It now remains for us to compute the sensitivities \( s^m \), which requires another application of the chain rule. It is this process that gives us the term backpropagation, because it describes a recurrence relationship in which the sensitivity at layer \( m \) is computed from the sensitivity at layer \( m + 1 \):

\[
s^m = -2 \hat{F}^M(n^M)(t - a) \tag{29}
\]

\[
s^m = \hat{F}^m(n^m)(W^{m+1})^T s^{m+1},
\]

\[
m = M - 1, ..., 2, 1
\]

where

\[
\hat{F}^m(n^m) = \begin{bmatrix} \hat{f}^m(n_1^m) & 0 & \ldots & 0 \\ 0 & \hat{f}^m(n_2^m) & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \hat{f}^m(n_s^m) \end{bmatrix}
\]

See reference [4], chapter 11, for a derivation of this result.

4.4 Variation of backpropagation

In some ways it is unfortunate that the algorithm we usually refer to as backpropagation, given by Eq. (27) and Eq. (28), is in fact simply a steepest descent algorithm. There are many other optimization algorithms that can use the backpropagation procedure, in which derivatives are processed from the last layer of the network to the first (as given in Eq. (30)). For example, conjugate gradient and quasi-Newton algorithms are generally more efficient than steepest descent algorithms, and yet they can use the same backpropagation procedure to compute the necessary derivatives. The Levenberg-Marquardt algorithm is very efficient for training small to medium-size networks, and it uses a backpropagation procedure that is very similar to the one given by Eq. (30).

4.5 Generalization (interpolation & extrapolation)

We now know that multilayer networks are universal approximators, but we have not discussed
how to select the number of neurons and the number of layers necessary to achieve an accurate approximation in a given problem. We have also not discussed how the training data set should be selected. The trick is to use enough neurons to capture the complexity of the underlying function without having the network overfit the training data, in which case it will not generalize to new situations. We also need to have sufficient training data to adequately represent the underlying function. To illustrate the problems we can have in network training, consider the following general example. Assume that the training data is generated by the following equation:

\[ t_q = g(p_q) + e_q \]  

where \( p_q \) is the system input, \( g(.) \) is the underlying function we wish to approximate, \( e_q \) is measurement noise and \( t_q \) is the system output (network target). Figure 12 shows an example of the underlying function \( g(.) \) (thick line), training data target values \( t_q \) (circles), and total trained network response (thin line). The two graphs of Fig. 12 and Fig. 13 represent different training strategies.

In the example shown in Fig. 12, a large network was trained to minimize squared error (Eq. 13) over the 15 points in the training set. We can see that the network response exactly matches the target values for each training point. However, the total network response has failed to capture the underlying function. There are two major problems. First, the network has overfit on the training data. The network response is too complex, because the network has more than enough independent parameters, and they have not been constrained in any way. The second problem is that there is no training data for values of \( p \) greater than 0. Neural networks (and other nonlinear black box techniques) cannot be expected to extrapolate accurately. If the network receives an input that is outside of the range covered in the training data, then the network response will always be suspect.

While there is little we can do to improve the network performance outside the range of the training data, we can improve its ability to interpolate between data points. Improved generalization can be obtained through a variety of techniques. In one method, called early stopping, we place a portion of the training data into a validation data set. The performance of the network on the validation set is monitored during training. When overfitting begins, the validation error will begin to increase, and at this point the training is stopped.

Another technique to improve network generalization is called regularization. With this method the performance index is modified to in-
clude a term which penalizes network complexity. The most common penalty term is the sum of squares of the network weights:

\[ F(x) = \sum_{q=1}^{Q} e_q^T e_q + \rho \sum (w_{ij}^k)^2 \]  \hspace{1cm} (33)

This performance index forces the weights to be small, which produces a smoother network response. The trick with this method is to choose the correct regularization parameter \( \rho \). If the value is too large, then the network response will be too smooth and will not accurately approximate the underlying function. If the value is too small, then the network will overfit. There are a number of methods for selecting the optimal \( \rho \). One of the most successful is Bayesian regularization.

Figure 13 shows the network response when the network is trained with Bayesian regularization. Notice that the network response no longer exactly matches the training data points, but the overall network response more closely matches the underlying function over the range of the training data. In the next section we will describe how multilayer networks can be used in neurocontrol applications.

5 Control systems applications

Multilayer neural networks have been applied successfully in the identification and control of dynamic systems. Rather than attempt to survey the many ways in which multilayer networks have been used in control systems, we will concentrate on one typical neural network controller, the model reference control. This controller is representative of the ways in which multilayer networks are used in control systems.

There are typically two steps involved when using neural networks for control:

1. system identification,
2. control design.

In the system identification stage, we develop a neural network model of the plant that we want to control.

In the control design stage, we use the neural network plant model to design (or train) the controller. In each of the three control architectures described in this report, the system identification stage is identical. The control design stage, however, is different for each architecture.

The next subsection of this report discuss the model reference control and will describe how it can be applied in practice.

5.1 Model reference control

5.1.1 System identification

The first stage of model reference control is to train a neural network to represent the forward dynamics of the plant. The prediction error between the plant output and the neural network output is used as the neural network training signal. The process is represented by Fig. 14.

![Figure 14: Plant identification](image)

One standard model that can be used for nonlinear identification is the Nonlinear Autoregressive-Moving Average (NARMA[5]) model:

\[ y(k + d) = h[y(k), y(k - 1), ..., y(k - n + 1), u(k), u(k - 1), ..., u(k - m + 1)] \]  \hspace{1cm} (34)

where \( u(k) \) is the system input, \( y(k) \) is the system output and \( d \) is the system delay. For the identification phase, we train a neural network to approximate the nonlinear function \( h \). The structure of the neural network plant model is given in Fig. 15, where the blocks labeled TDL are tapped delay lines that store previous values.
of the input signal. The equation for the plant model is given by

\[ y_m(k + 1) = \hat{h}[y_p(k), ..., y_p(k - n + 1), u(k), ..., u(k - m + 1); x] \]  

(35)

where \( \hat{h}[.; x] \) is the function implemented by the neural network and \( x \) it the vector containing all network weights and biases.

We have modified our previous notation here, to allow more than one input into the network. \( IW^{i,j} \) is a weight matrix from input number \( j \) to layer number \( i \). \( LW^{i,j} \) is a weight matrix from layer number \( j \) to layer number \( i \). Although there are delays in this network, they occur only at the network input, and the network contains no feedback loops. For these reasons, the neural network plant model can be trained using the backpropagation methods for feedforward networks described in the first part of this report. It is important that the training data cover the entire range of plant operation, because we know from previous discussions that nonlinear neural networks do not extrapolate accurately. The input to this network is an \( (n_y + n_u) \)-dimensional vector of previous plant outputs and inputs. It is this space that must be covered adequately by the training data.

### 5.1.2 Neural network controller

Model reference control architecture uses two neural networks: a controller network and a plant model network, as shown in Fig. 16. The plant model is identified first and then the controller is trained so that the plant output follows the reference model output.

The online computation of the model reference controller is minimal. However the model reference architecture requires that a separate neural network controller be trained, in addition to the neural network plant model. The controller training is computationally expensive, since it requires the use of dynamic backpropagation.

On the positive side, model reference control applies to a large class of plants which requires that the plant be approximated by a companion form model.

Figure 17 shows the details of the neural network plant model and the neural network controller. There are three sets of controller inputs:

1. delayed reference inputs,
2. delayed controller outputs (plant inputs),
3. and delayed plant outputs.

For each of these inputs, we select the number of delayed values to use. Typically, the number of delays increases with the order of the plant. There are two sets of inputs to the neural network plant model: delayed controller outputs and delayed plant outputs.

The plant identification process for model reference control uses the NARMA model given by Eq. (34).

It is clear from Fig. 17 that the model reference control structure is a recurrent (feedback) network. This type of network is more difficult to train than the feedforward networks that were discussed in the first half of this report and that are used for plant identification. Suppose that we use the same gradient descent algorithm, Eq. (16), that is used in the standard backpropagation algorithm. The problem with recurrent networks is that when we try to find the equivalent of Eq. (23) (gradient calculation) we note that the weights and biases have two different effects on the network output.

The first is the direct effect, which is accounted for by Eq. (23). The second is an indirect effect, since some of the inputs to the network, such as \( u(t - 1) \), are also functions of the weights and biases. To account for this indirect effect we must use dynamic backpropagation to compute the gradients for recurrent networks.

A detailed description of dynamic backpropagation is anyway beyond the scope of this report. In addition to the difficulty in computing the gradients for recurrent networks, the error surfaces for these networks pose special difficulties for gradient-based optimization algorithms. Gradient-based training procedures that are well suited to training recurrent networks are discussed in reference [6]. The data used to train the model reference controller is generated while applying a random reference signal which consists of a series of pulses of random amplitude
Figure 15: Neural network plant model

Figure 16: Model reference control architecture
5.2 Application: altitude control of a quad-rotor

The potentiality of the model reference control technique will be demonstrated controlling the vertical dynamics of a quad-rotor. Referring to Fig. 18, the equation of motion that describes the vertical dynamics of a quad-rotor, can be simplified as follows

\[ T = m \ddot{z}_{\text{body}} \quad (36) \]

where

- \( T \) is the complessive thrust of the rotors,
- \( m \) is the quad-rotor mass,
- \( z_{\text{body}} \) is the axis perpendicular to the quad-rotor plane, upwards directed. The quad-rotor plane is unequivocally defined as the plane containing both the axes \( x_{\text{body}} \) and \( y_{\text{body}} \). The \( x_{\text{body}} \) axis is aligned with the quad-rotor arm forward directed. The \( y_{\text{body}} \) axis is directed consequentially.

5.2.1 Plant identification

The system was sampled at intervals of 0.05 seconds. To identify the system, input pulses with intervals between 0.01 and 5 seconds and amplitude between -10.791 N and 20 N have been used. The neural network plant model used two delayed values of thrust (\( m = 2 \)), two delayed
values of quad-rotor position \((n = 2)\) as input to the network and 15 neurons were used in the hidden layer (a 5-15-1 network). Figure 19 shows the MatLab tool that, opportunistically tuned, allows defining the plant neural network.

Figure 19: Plant identification and training, MatLab graphical user interface

5.2.2 Neural network controller

The objective of the control system is to have the quad-rotor altitude to track the reference model

\[
\dot{y}_r = -6\dot{y}_r - 9y_r + 9r
\]

where \(y_r\) is the output of the reference model and \(r\) is the input reference signal. For the controller network, it has been used a 5-13-1 architecture. The inputs to the controller consisted of two delayed reference inputs, two delayed plant outputs, and one delayed controller output. The controller was trained using a BFGS (Broyden, Fletcher, Goldfarb, and Shanno) quasi-Newton algorithm, with dynamic backpropagation used to calculate the gradients. Figure 20 shows the MatLab tool that, opportunistically tuned, allows defining the neural network controller.

Figure 20: Neural network controller identification and training, MatLab graphical user interface

are smooth. At certain set points there is some steady state error. This error could be reduced by adding more training data at those steady state conditions where the error is largest. The problem can occur in the plant model or in the controller network.

6 Distance sensor embodiment and consequent control tuning

In order to allow the implementation of an automatic sense & avoid system a suitable sensor embodiment has been conceived. This concept is shown in Fig. 23. It is mainly based on the Cardan shaft (see Fig. 24) mechanical property. One arm of the shaft is connected to a stepper motor that is linked to the quad-rotor structure. This arm axis is named \(c_1\) in the Fig. 25. This axis is aligned with the \(z_{\text{body}}\) axis. The stepper motor is inserted in order to allow rotations of the laser distance sensor around the \(c_2\) axis. For the mechanical properties of the Cardan shaft, in fact, a certain angular velocity around the \(c_1\) axis will be equally transmitted to the second arm, i.e. a rotation around \(c_2\) will take place. The joint conjunction between the first and the second arm is intended to be left free, i.e. without an active motor control, since, due to the gravity force it
Neural network control of a quad-rotor vertical dynamics

Figure 21: Neural network controlled quad-rotor altitude behaviour

Control action and quad-rotor controlled vertical dynamics

Figure 22: Control action and quad-rotor controlled vertical dynamics
will constantly (with certain oscillations to be passively damped) maintain its vertical position.

Control requisite is then derived from the obstacles perception with this, above explained, sensor setup. Starting from a plate surface condition the laser distance measurer will start sensing obstacles when

\[
\Delta = \delta_{\text{max}} \sin(\beta)
\]

(38)

where \(\delta_{\text{max}}\) is defined as the maximum distance that can be measured by the sensor. With the aim to define the required control reaction rise time of our **sense & avoid** system a **time to collision** \(t_c\) value has to be defined. This time is approximately found as follow

\[
t_c \approx \frac{\Delta}{V \cos(\gamma)}.
\]

(39)

For example, if a quad-rotor is flying with

\[V \cos(\gamma) = 10 \text{m/sec}\]

and the maximum distance that can be measured by the sensor is

\[\delta_{\text{max}} = 30 \text{m},\]

if \(\beta\) is equal to \(80^\circ\) then

\[t_c \approx \frac{30 \cdot \sin(80^\circ)}{10} = 2.95 \text{sec}.
\]

This value sets an important constraint for our controller performances. Having a rise time minor than 2.95 sec, the neural network controller previously designed could sustain the above presented condition but for more restrictive cases, as for example if the speed increased above 10 m/sec or for lower quality sensors, the previous neural controller could not be satisfying anymore, if we maintained the same network design.

### 6.1 Using only one forward looking distance sensor: simulation

The case in which only one sensor is used for altitude tracking has been simulated using MatLab Simulink environment. The idea behind the simulation is that the unique sensor is mounted
in a way that it allows a perception of the forward surface variation, $t_c$ seconds before that the quad-rotor arrives on it. Simulation results are shown in Fig. 25. In this case $t_c$ is equal to 2 sec and the quad-rotor is required to fly 1 m above the surface. As it could be foreseeable, the quad-rotor has a good approach to the obstacle but while it flies beyond, it is no more able to see the obstacle under itself and it crashes against the surface.

Just a note, the reader should be aware of the fact that, for the sake of simplicity, lasers shadowing has been accounted for. This because the simulation was conceived with the aim to highlight the not sufficient capability of this sensor setup for obstacles avoidance. Simple geometrical constructions could in fact, highlight, that also modelling the latter phenomenon, this sensor setup is not suited to guarantee our requisite.

### 6.2 A three sensors setup

In order to solve the issue explained in the precedent paragraph a three sensors setup is proposed. This sensor embodiment is reported in Fig. 26. To be noted that the normal to the sensor “1”, namely $z_{sensor1}$, has to be kept constantly aligned with the gravity vector. This target could be pursued following a passively controlled cardan shaft setup or using an actively controlled platform able to move constrasting with the pith/roll quad-rotor motion. This requires the development of a specific control law.

With the aim to have a draft sketch of what could be the quad-rotor vertical dynamics the simulation shown in Fig. 27 has been realized. This simulation shows a comforting behaviour of the quad-rotor altitude, giving the idea that this is the right way to follow. The reader has to be aware of the fact that the simulation presented in Fig. 27 does not account for the sen-
6.2.1 Limits of a three sensors setup

A three sensors setup like the one previously shown could be suited when the quad-rotor dynamics is bounded in the sensors plane. For this reason it is necessary to build a proper sensors platform able to rotate around the $z_{sensor_1}$ axis. This rotation angle could be derived from the pitch and roll angle, following the equations of the dynamics of the quad-rotor system. This clarifies the necessity for another specific control law with the aim to keep, automatically, the three sensors in the motion plane.

7 Summary and conclusion

A neural network plant model has been first developed. The plant model has been then used to train a neural network controller to force the plant output to follow the output of a reference model. This control architecture requires the use of dynamic backpropagation for training the controller. This generally takes more time than training static networks with the standard backpropagation algorithm. However, this approach applies to a very general class of plant. The controller requires minimal online computation.

This report has given a brief introduction to the use of neural networks in control systems. In the limited space it was not possible to discuss all possible ways in which neural networks have been applied to control system problems. One type of network has been selected, the multilayer perceptron. The capabilities of this network for function approximation has been shown and it has been described how it can be trained to approximate specific functions. Then a control architecture based on the model reference control approach has been presented using the neural network function approximators as basic building blocks. The control architecture has been demonstrated on a physical example with acceptable results. Finally various sensors embodiment setups have been conceived and their key parameters have been evaluated with respect to the consequent impact on the controller requisites.

References


