Politecnico di Milano

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE
Corso di Laurea Magistrale in Automation and Control Engineering

Tesi di Laurea Magistrale

Design of Internal Model Control Systems with Recurrent Neural Networks

Relatore
Prof. Riccardo Scattolini

Correlatore
Dott. Fabio Bonassi

Candidato
Luca Depari
Matricola 914792

Anno Accademico 2019–2020

Sommario

Negli ultimi anni l'utilizzo delle reti neurali (NN) ha avuto un forte incremento, fino a diventare parte integrante delle nostre vite (si pensi per esempio al sistema di sblocco di uno smartphone tramite riconoscimento facciale, piuttosto che l'elaborazione di immagini o il riconoscimento vocale). Di fatto, le reti neurali sono in grado di risolvere (con una precisione più o meno elevata) molti compiti che da un punto di vista computazionale richiederebbero altrimenti un costo estremamente elevato. Tuttavia, questo tipo di reti risulta trovare poca applicazione pratica in quello che è il mondo del controllo di sistemi reali, soprattutto per l'elevata quantità di dati necessari al loro corretto addestramento, i quali non sempre sono disponibili. Ciò nonostante, le grandi potenzialità delle reti neurali, quali la capacità di imitare il comportamento di un impianto (piuttosto che quello di un controllore) e la loro elevata versatilità, ci suggeriscono che il loro impiego all'interno di schemi di controllo potrebbe portare a risultati molto positivi.

É bene precisare che al giorno d'oggi esistono diversi tipi di reti neurali, ognuna caratterizzata da pro e contro. Le due principali categorie sono le reti neurali feedforward (FFNN) e le reti neurali ricorrenti (RNN): le prime hanno applicazioni in diversi settori, uno dei quali è quello della classificazione (per esempio image classification), mentre le reti neurali ricorrenti sono maggiormente adatte per effettuare operazioni di analisi predittiva tramite l'analisi di dati, ad esempio operazioni come il riconoscimento di lettere scritte a mano o il riconoscimento vocale.

Lo scopo di questa Tesi è quindi quello di fornire prima di tutto un'introduzione al mondo delle reti neurali (sia FFNN che RNN), discutendone la struttura e il funzionamento, sottolineando soprattutto come viene effettuato l'addestramento (algoritmi utilizzati e preparazione dei dati necessari). Dopodiché, ci si concentrerà sulle prestazioni che le reti neurali ricorrenti sono in grado di fornire nella modellizzazione di un sistema fisico reale, mettendo in evidenza le differenze legate all'utilizzo di diverse strutture. Come precedentemente anticipato, le reti neurali possono essere utilizzate all'interno di schemi di controllo per garantire il tracking di segnali di riferimento imposti ad un dato sistema (nel nostro caso un sistema MIMO non lineare): per questo motivo, la loro applicazione in uno schema di controllo di tipo Internal Model Control (IMC) verrà presa in esame, considerando due reti neurali ricorrenti, una per il modello e una per il controllore (per cui sarà proposta una linea guida inerente al suo training).

Le reti neurali perciò possono avere un ruolo sia come modello del sistema con cui ci si interfaccia, sia come controllore di quest'ultimo, andando a garantire buone (ma non ideali, se non considerando ipotesi stringenti) prestazioni in termini di accuratezza di tracking dei segnali di riferimento.

Abstract

In the last years the usage of Neural Networks (NNs) had a strong increase, assuming a central role in our nowadays life (just think about the unlock system of a smartphone through face recognition, or image processing and voice recognition). Indeed, NNs are able to solve many different tasks with a certain accuracy that, from a computational point of view, require a high cost if performed in other ways without neural networks. However, NNs find a limited practical usage in the control system world, especially for the high number of data required for the network training, that is not always available (the accuracy of a NN is strongly related to its training, i.e. to the used dataset). Nevertheless the high potentialities of neural networks, such as their capability to mimic a plant behaviour (or acting as a controller) and their versatility, suggest us that their usage for the design of control schemes may lead to interesting and positive results.

Let us notice that nowadays there exist different kinds of neural networks, each one characterised by pros and cons. The two main classes of NNs are the FeedForward NNs (FFNNs) and the Recurrent NNs (RNNs): the former is used for different operations, such as classification tasks (e.g. image classification), while the latter is adopted for predictive analysis through the examination of (time-series) data (e.g. for hand-written digits recognition and voice recognition).

The aim of this Thesis is to provide first of all a general overview on the neural networks world (focusing in particular on RNNs and FFNNs), discussing their structure and their working behaviour, highlighting more specifically how the training operation is performed (adopted algorithms and data collection). Then, we will deal with the performances that RNNs are able to provide in terms of real system modelling, pointing out the differences due to NN structure changes (as the type of cell, the number of units and layers and so on). As previously mentioned, neural networks can be used inside control schemes (for example as model or controller) in order to ensure the tracking of reference signals imposed to a given plant (in our case, a nonlinear MIMO system): this is why an Internal Model Control (IMC) scheme will be considered, characterised by two RNNs, one for the model and one for the controller (for the latter, a guideline for its training will be proposed).

Hence, neural networks can have a role as model of the system we deal with, and as controller of this one, ensuring good (but not ideal, if not under specific assumptions) performances in terms of tracking task accuracy.

Contents

List of Figures

List of Tables

Chapter 1

Neural networks basics

In this chapter, the basics of the neural network theory will be analysed, starting from the definition of a neuron and how a neural network works, considering first the standard *Feed Forward Neural Networks* (FFNNs) [2]. Then, the training of such networks will be discussed, focusing on the methods adopted (gradient descent and its variations for parameters update, together with the backpropagation method) and the main problems that may arise, e.g. vanishing and exploding gradient.

Afterwards, *Recursive Neural Networks* (RNNs) will be discussed, highlighting the main differences among these structures and the common FFNNs. The dissimilarities of the training algorithms will be pointed out too, introducing the so-called *Backpropagation Through Time* (BPTT) and its truncated variation. Hence, the two main types of gated cells, called LSTM and GRU cells respectively, used to build a RNN will be presented and analysed, focusing on their features and equations.

1.1 What is a neural network?

First of all, we shall explain what it is meant by the words *Neural Network* (NN). Every time we talk about a neural network, we are talking about a particular kind of systems that are, in some sense, capable of "learning" to perform tasks (like the recognition of specific items, persons, or the classification of data) starting from a set of examples. The idea, in other words, is that providing a sufficiently high number of examples to the network, it will be able to learn the patterns beyond data in order to correctly execute a specific task (e.g. image classification) even with previously unseen data.

A good example is provided by Michael A. Nielsen in [3], where the problem of hand-written digits recognition is discussed. Let us consider for example a string of five hand-written numbers: for a human being, the recognition of the letters is quite easy, but translating this concept into a computer program is not easy at all. However, if we train a NN with a dataset consisting of hand-written digit images (associating to each image the corresponding represented number), it will be able to recognize hand-written digits never seen before with a certain accuracy, performing a classification operation.

Nowadays neural networks are used for multiple operations: face recognition (e.g. to unlock our smartphones), predictions of vehicle trajectories, to filter e-mail spam.

1.1.1 Structure of a neural network

In order to correctly understand how a neural network works, let us focus on its structure: the basic unit of a NN is the so-called *artificial neuron* (also named *neuron* for simplicity, Figure 1.1), which takes in input one or more quantities u_i, does some operations with them and provides a single output y.

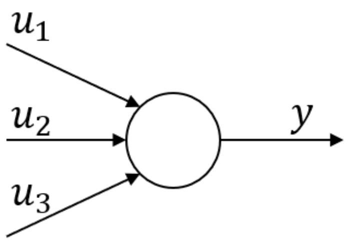


Figure 1.1. Scheme of an artificial neuron with 3 inputs.

In particular, to each input u_i is assigned a specific weight ω_i such that u_i becomes $u_i \leftarrow u_i \cdot \omega_i$. At this point, all the weighted inputs are summed, and a bias b is added, i.e.:

$$\Big(\sum_{i=1}^{n} u_i \cdot \omega_i\Big) + b$$

where n is the number of inputs of the neuron. Then, the result is passed through a specific activation function f, getting the final output y of the artificial neuron:

$$y = f\Big(\Big(\sum_{i=1}^{n} u_i \cdot \omega_i\Big) + b\Big) \tag{1.1}$$

1.1.2 Activation functions

Focusing on the activation function f of Eq. (1.1), there are different possibilities that can be considered:

- *binary step function*, which consists in a simple step where the output can be either 0 or 1, on the basis of the value the quantity $(\sum_{i=1}^{n} x_i\, \omega_i + b)$ assumes. In particular, it is considered a threshold for which the following relationship holds:

  $$y = \begin{cases} 0, & if \ (\sum_{i=1}^{n} u_i \cdot \omega_i) + b \leq threshold \\ 1, & if \ (\sum_{i=1}^{n} u_i \cdot \omega_i) + b > threshold \end{cases}$$

  When a neuron is characterised by a binary step function, it is called *perceptron* (discussed by Frank Rosenblatt in [4]): tuning the threshold value, the weights ω_i and the bias b, different decisions taken by the model could be obtained. It means that, by a good tuning, the network could take the decisions we desire. Taking as reference Figure 1.2, the binary step is a non linear and non continuous function. The main drawback of a perceptron neuron is that its output can be just 0 or 1: it implies that, if a small change of the weights and/or of the threshold occurs (maybe in order to try to impose the desired behaviour to the network), the final output of the NN could be completely different.

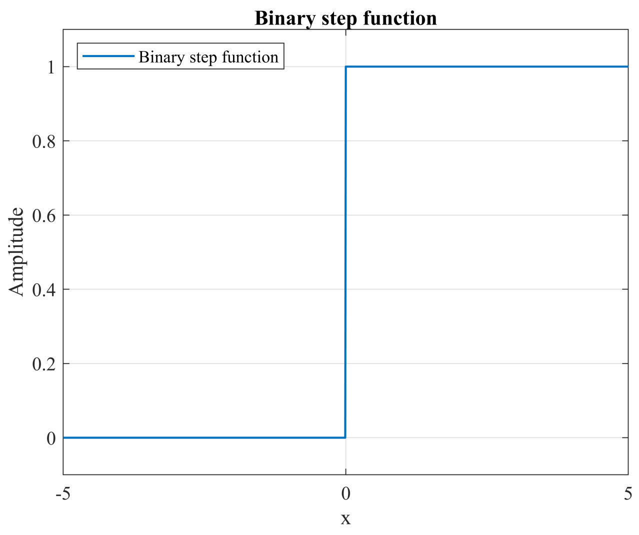


Figure 1.2. Plot of the binary step function from -5 to +5.

- *Sigmoid function*, σ_g, which allows any value in the range [0, 1] (take as reference Fig. 1.3). The mathematical relationship of such function is given by:

$$\sigma_g(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x} \tag{1.2}$$

  It can be noticed that, by replacing inside the sigmoid function the terms related to the neural network, Eq. (1.2) becomes:

$$\frac{1}{1+e^{(-\sum_i u_i \cdot \omega_i - b)}}$$

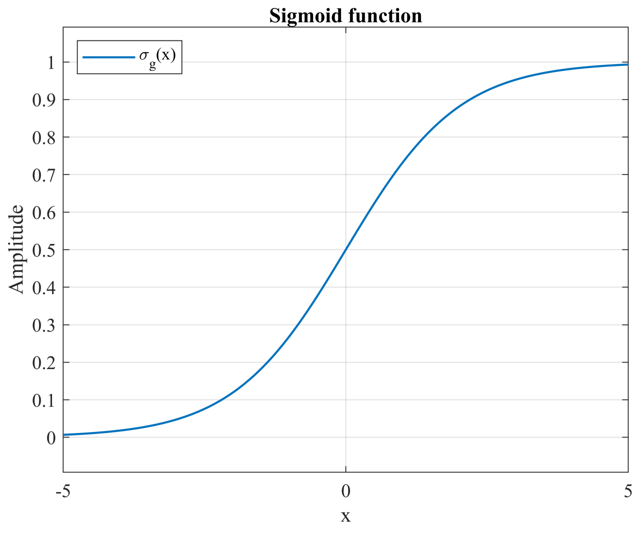


Figure 1.3. Plot of the sigmoid function $\sigma_g(x)$ between -5 and +5.

As noticeable from Fig. 1.3 the function is non linear, continuous and derivable. Let us notice that different sigmoidal functions can be used, with different expressions with respect to Equation (1.2), which is just a possibility.

- *Hyperbolic tangent function* ($tanh(x)$, also indicated as $\sigma_c(x)$). As noticeable from Figure 1.4 the activation function value in this case can change between -1 and 1, instead of 0 and 1 as seen for $\sigma_g(x)$ in Figure 1.3. Also, it is clearly a continuous, derivable function. Considering Equation (1.2), it can be proved that $tanh(x) = \sigma_c(x) = 2\sigma_g(2x) - 1$.

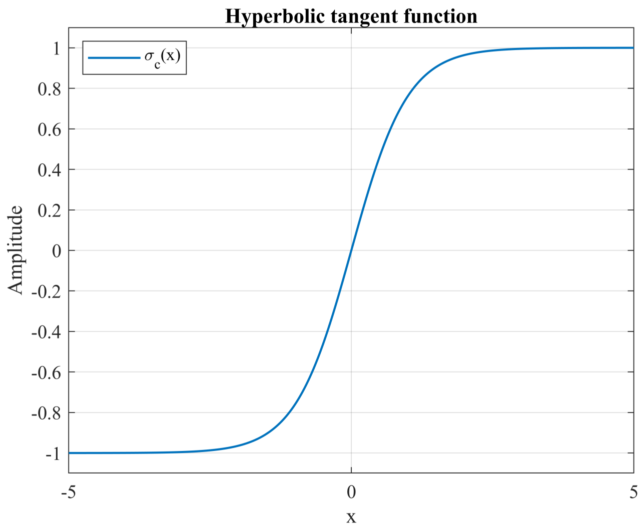


Figure 1.4. Plot of the hyperbolic tangent function $tanh(x)$ ($\sigma_c(x)$) between -5 and +5.

- *ReLU function* (Rectified Linear Unit, Fig. 1.5) [5], that can be described by:

$$f(x) = \begin{cases} 0, & if\ x < 0 \\ x, & otherwise \end{cases} = \max\,(0,\ x)$$

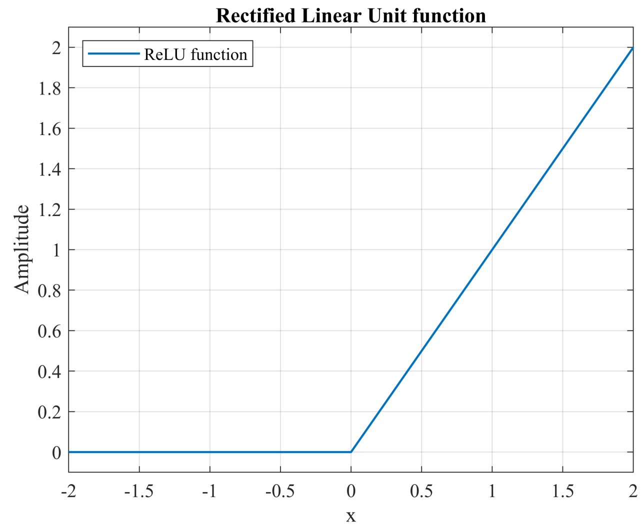


Figure 1.5. Plot of the ReLU activation function between -2 and +2.

Notice that it provides 0 as output for all $x \leq 0$, but for positive values it never saturates. From a mathematical point of view, it is a piecewise linear function and its derivative is the slope (for negative values it is 0 while for positive ones it is 1). However, let us notice that, since it is a piecewise linear function, the derivative cannot be computed in zero (non-differentiable at $x = 0$), so for simplicity it is assumed that the derivative in that point is equal to 0. In [6] it has been shown how ReLU, even though it is usually avoided in RNNs, can be used as the output activation in LSTMs (a particular type of RNNs discussed in Section 1.3.6), under the assumption that a careful initialization of the network weights is provided. An improved version of the ReLU function is the *leaky ReLU*, defined as:

$$f(x) = \max(\alpha x, x)$$

where the hyperparameter α defines the slope of the function for negative values of x (usually $\alpha = 0.01$).

- *Softmax function*, mainly used for classification problems. Its mathematical formulation is the following one:

  $$f(\mathbf{x}) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}, \quad for\ i = 1, .., K$$

  What we do is to apply the exponential function to each element x_i of the input vector $\mathbf{x}$ and normalize these values by dividing by the sum of all these exponentials.

1.1.3 Neural Networks

By connecting one with each other multiple neurons, what we get is a neural network, where different layers can be included. For example, the NN of Figure 1.6 has an input layer (the first one), two hidden layers (by hidden layer we mean each layer enclosed between the first and the last ones) and an output layer (i.e. the last one). Clearly, multiple layers can be introduced, each one with 1 or more neurons. This kind of structure is often called also *Artificial Neural Network* (ANN). Let us point out that the number of hidden layers is not fixed, we may have one or hundreds of hidden layers; clearly, the overall number of layers of the network is strongly related to what we want to obtain and to the difficulty of the problem, and there is not a rule to establish it. However, some methods have been proposed in the years, as in [7].

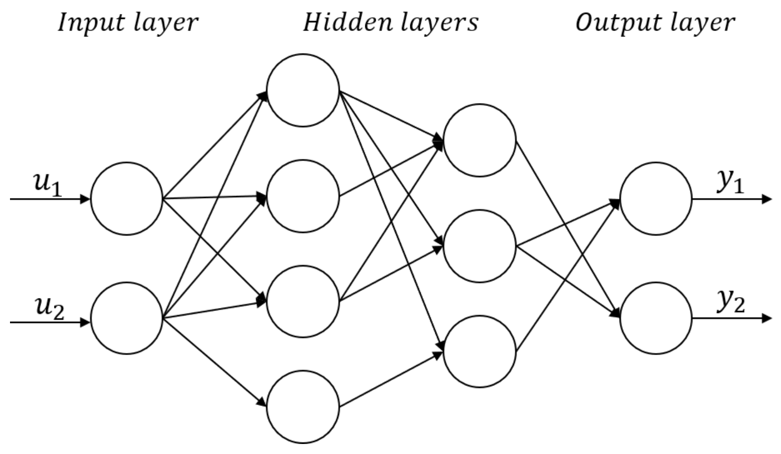


Figure 1.6. Scheme of an artificial neural network with four layers.

Let us point out that the input layer does not do any operation with the inputs u_i, but it just provides them to the next layer's neurons (in this case to the first hidden layer ones), which will follow Eq. (1.1) for each received u_i. For example, the first neuron of the first hidden layer receives both u_1 and u_2, multiplies them by the corresponding weights $w_{1i}^{(1)}$ summing the corresponding bias, and in the end it passes the result through the activation function in order to compute the output.

In a single NN, the neurons may have different activation functions f one from each other (as it happens with GRU and LSTM, explained in Sections 1.3.7 and 1.3.6 respectively).

If an ANN has more than one hidden layer, it is called *Deep Neural Network* (DNN), as the example shown in Figure 1.6.

1.1.4 Feed-forward passage

In the networks that have been considered up to now, the outputs provided by a layer are used as the inputs of the next layer: it means that the network is not characterized by loops in its structure. Hence, it is possible to generalize the computation of the outputs of a NN of this type starting from its inputs in an easy way.

Let us consider the example treated in [8] where a network characterized by three inputs u_i, an hidden layer with three neurons and an output layer with a single neuron is considered (notice the input layer is not considered since it consists only of the inputs u_i as already explained). For simplicity, we assume that each layer is fully-connected, i.e. each neuron of each layer receives as inputs all the outputs of the previous layer; for the first hidden layer, the inputs are all the u_i of the NN. We denote with $y_j^{(t)}$ the output of neuron j of layer t, with $\omega_{ji}^{(t)}$ the weight associated to the input i of the neuron j of layer t and with $b_j^{(t)}$ the bias associated to the neuron j of layer t. Hence, recalling the Eq. (1.1) and the notation previously adopted, we get:

$$\begin{aligned}
y_1^{(2)} &= f\Big(\sum_{i=1}^{3} u_i \cdot \omega_{1i}^{(1)} + b_1^{(1)}\Big) \\
y_2^{(2)} &= f\Big(\sum_{i=1}^{3} u_i \cdot \omega_{2i}^{(1)} + b_2^{(1)}\Big) \\
y_3^{(2)} &= f\Big(\sum_{i=1}^{3} u_i \cdot \omega_{3i}^{(1)} + b_3^{(1)}\Big)
\end{aligned}$$

The final output of the NN, indicated as y_{out} (or $y_1^{(3)}$), will be:

$$y_{out} = y_1^{(3)} = f\Big(\sum_{i=1}^{3} y_i^{(2)} \cdot \omega_{1i}^{(2)} + b_1^{(2)}\Big) \tag{1.3}$$

It can be noticed that, as expected, the input of each neuron is multiplied by the corresponding weight ω_{ji}, then all the weighted inputs are summed together with the bias b_j. This sum is (eventually) passed through the activation function f. The procedure is performed for all the layers of the network, until the last layer is reached. Clearly, the overall computation could be extended to a generic number of layers and of neurons; considering n layers, with m neurons on the last one, the result is given by:

$$
\begin{aligned}
y_1^{(n)} &= f\Big(\sum_{i=1} y_i^{(n-1)} \cdot \omega_{1i}^{(n-1)} + b_1^{(n-1)}\Big) \\
&\vdots \\
y_m^{(n)} &= f\Big(\sum_{i=1} y_i^{(n-1)} \cdot \omega_{mi}^{(n-1)} + b_m^{(n-1)}\Big)
\end{aligned}
$$

Notice the sums goes from $i = 1$ up to the number of the inputs the corresponding neuron j of layer n receives.

Summarizing, we can conclude that the feed-forward passage is used to compute the final output of a (feed-forward) NN, starting from its inputs u_i and passing through all the intermediate outputs of each layer (and correspondingly, of each neuron).

1.2 Training of a neural network

At this point, the main structure of a neural network is known, hence we are ready to discuss its training. First of all, in this section the gradient descent method will be treated, together with its main improved version (the stochastic gradient descent). Then the backpropagation method will be explained. Notice that in the following sections only the main ideas of the methodologies are reported. For more details, the interested reader is referred to [9].

1.2.1 Gradient descent

Since the outputs of a NN are strongly dependent from the weights and biases of each neuron (together with the input of the network itself), a good way to proceed is to implement an algorithm that allows us to find the optimal values of ω_i and b_i. The basic idea is to find the ideal set of parameters that minimises a given cost function (like for example the *Mean Square Error*, MSE). This is a non linear, non convex problem (with high computational complexity) that can be solved with the *gradient descent method* [10]: it is an iterative method in which, at every iteration, the direction along which the gradient of the cost function to be minimised is identified, and the algorithm takes a "step" along such direction.

Indicating with θ the parameter vector, it is usually initialized with random values, and, at every step, we try to improve it proceeding as previously said, looking for the optimal value of θ, i.e. $\hat{\theta}$. Since at every iteration we make a (usually small) step in order to improve θ, the size of such step (indicated with η, and called *learning rate*) represents a key aspect for the gradient descent method: taking a too small value for η leads to a very slow learning phase (i.e. in order to find the minimum of the cost function a lot of iterations are needed), while taking a too high value for the learning rate may prevent from reaching the minimum.

According to the discussion in [11], in order to properly implement the gradient descent method, it is needed to compute the gradient of the cost function (indicated from now on with $J(\theta)$) with respect to each parameter θ_j, which means we want to observe how much the variation of θ_j affects the value of J. Let us consider an easier case assuming we are dealing with a linear regression model such that:

$$\hat{y} = \sum_{i=1}^{m} (\theta_i \cdot v_i) + b = \theta^T \cdot \mathbf{v}$$

where v_i is the i-th feature value and b the bias term (included into θ vector). Hence, applying the gradient descent approach we have:

$$\begin{aligned} J(\theta) &= \frac{1}{m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 \\ \frac{\partial J(\theta)}{\partial \theta_j} &= \frac{2}{m} \sum_{i=1}^{m} (\hat{y}_i - y_i)\, v_j \end{aligned}$$

where y_i is the i-th desired output. By simply collecting all the partial derivatives of J into a vector, the gradient vector of the cost function $\nabla_\theta J(\theta)$ is obtained. Once the gradient is known, for a parameter variation $\Delta\theta$ the resulting variation of the cost function will be such that:

$$\Delta J(\theta) \simeq \nabla_\theta J(\theta) \cdot \Delta\theta$$

At this point, the learning rate η mentioned before comes into account, indeed by taking:

$$\Delta\theta = -\eta \cdot \nabla_\theta J(\theta)$$

we get the iterative rule:

$$\theta \leftarrow \theta - \eta \cdot \nabla_\theta J(\theta) \tag{1.4}$$

As previously said, the size of η is a key point for the algorithm: taking a too high value for η may lead to a huge change of parameter values, so the minimum may not be reached or, in the worst case, the algorithm may diverge. On the other hand, a too low value of the learning rate leads to a very small change $\Delta\theta$, i.e. the algorithm works very slowly since the parameters go through a small update procedure.

Since it is an iterative method, the general idea is to stop the iterations when the norm of the gradient vector becomes lower than a fixed tolerance. Supposing we are dealing with a generic cost function $J(\theta)$, it may happen that it has multiple minima: in order to guarantee the best result possible, we would like the algorithm to stuck on the global minimum of J, and not on a local minimum. This is a common problem in the gradient descent methods, related also to the random initialization of θ vector.

1.2.2 Stochastic gradient descent

An improved version of the gradient descent is the so-called *Stochastic Gradient Descent* (SGD) [12]: it can be observed that the main drawback of the previously described method is that it takes usage of the whole training set (i.e. the set of data used to train the NN) to compute the gradient at every step. This means that, if we are dealing with a huge training set, the computational effort may be unacceptably high. The stochastic gradient descent addresses this issue, selecting a random subset from the training set and computing the gradient based just on those extrapolated elements, and then using this gradient to update the weights and biases.

The procedure is repeated for all the remaining elements contained in the training set. Let us notice that it is not mandatory to extrapolate a single instance (where by instance we mean single observation or record of data) from the training set, we can simply take n random elements from it.

The steps previously described refers to just one epoch, where one epoch ends when all the instances in the training set are used to train the NN parameters. In order to properly train the network, it should be taken into account more than one epoch (there is not a fixed number, it strongly depends on the benchmark and on the network structure considered).

Clearly, the lower the size of the set on which the gradient is computed, the better the overall performance of the method in terms of convergence time. On the other hand, this kind of method will not be as "smooth" as the plain gradient descent, since the cost function will not likely be constantly decreasing, but it may grow and then decrease at every step. However, by taking its average over the steps, the cost function should show a decreasing trend, until it is close to the minimum. An important drawback is that, once the method is close to the minimum, it is not able to reach it but it starts to bounce around that value. A possibility to partially solve this problem is to adapt the learning rate η at every step, starting from a high value and progressively reducing it.

Due to the random picking of the sample set for the computation of the gradient, in the same epoch it may happen that some instances are used more than once for the training, and others very few times. That is why an alternative proposed is to shuffle the training set, use every instance for the training in an epoch, and then re-shuffle it again and repeat the procedure.

Recalling the problem related to the presence of possible local minima in the cost function discussed for gradient descent, since with SGD the cost function will bounce around the (local or global) minimum, it may help to escape from a local minimum we are stuck in.

1.2.3 Backpropagation

Up to now, we discussed the structure of a NN and the algorithm typically used to learn its parameters (i.e. weights ω_i and biases b_i), called gradient descent. Taking into account a generic FFNN (which is characterised by the absence of loops in its structures, i.e. the flow of signals is unidirectional from input layer to output layer, see Fig. 1.6), in order to train this kind of networks a new method (that, as we will see, takes usage of a forward phase and a backward phase, plus for example the gradient method for the parameters optimization), called *backpropagation method* [9], is presented. Briefly, the idea is to compute the loss function contribution of each connection of the NN, aiming to reduce such value applying the gradient descent (or SGD) for parameters update.

First of all, the backpropagation algorithm computes a prediction using the already seen forward phase (Section 1.1.4), taking as input a training instance: hence, the output of every neuron is computed. At this point, the value of the loss function (that could be considered for simplicity as the difference in absolute value between the current output and the desired one, however any kind of cost function can be used without problems, like for example the MSE) is computed, and then the backward phase starts: the algorithm aims to compute the partial derivatives of the cost function with respect to each weight (and bias) of the NN, starting from the connections of the last hidden layer with the output one, checking how much these connections contributed to the overall error (cost function value), extending then this computation for all the previous hidden layers until the input one is reached. Thus, it allows us to properly compute the loss function contribution (i.e. the loss function gradient) of each connection. Once the gradient is known, it is easy to use it for the gradient descent step on all the connection weights of the NN in order to minimize the loss function. In other words, the algorithm computes the loss function contribution of each connection of the network, and then it changes the values of the connection weights in order to reduce the achieved value for the loss.

It is important to highlight that this algorithm does not work with all the activation functions presented in Section 1.1.2: indeed, if for example the binary step function is considered (Fig. 1.2), the backpropagation algorithm does not work since there is not a gradient to work with. Therefore, the other activation functions need to be considered, e.g. the sigmoid or the hyperbolic tangent function, since they are continuous, derivable functions.

In conclusion, the backward-pass aims to compute the loss function gradients used for the parameters optimization (performed through for example gradient descent algorithm) in order to reduce an error or to improve a performance index, obtaining a trained NN.

1.3 Recurrent Neural Networks

In the previous sections only FFNNs have been described. However, for the purposes of this Thesis, the so-called *Recurrent Neural Networks* (RNNs) need to be introduced: as discussed by Goodfellow et al. in [9], RNNs are mainly used to process sequences of data (even with variable length), including time series data. An interesting property of RNNs is that once they are correctly trained (which requires enough data to be collected), they are able to forecast future outputs with certain accuracy using current (and previous) analysed data, since RNNs are characterised by a memory that allows them to memorise useful patterns observed in the already examined datasets.

Indeed, the main reason to consider RNNs instead of FFNNs is that the latter outputs are computed on the basis only of the current inputs, since they are not characterised by any dynamics. On the other hand, the RNNs are able to store informations from all the inputs received from the initial time instant, considering also them for the computation of the outputs.

Let us keep in mind that the training of a RNN is done in a very similar way to what has been said up to now, that is by the minimization of a given cost function that evaluates the difference between the computed output $\hat{y}(t)$ and the real one $y(t)$, however introducing some differences due to the change of the network structure.

1.3.1 Structure of a recurrent neural network

Basically, the difference between a RNN and a FFNN is how the neurons are connected one to each other: in RNN the flow of the input is not along a unique direction due to the presence of connections that link the outputs of the neurons with their inputs.

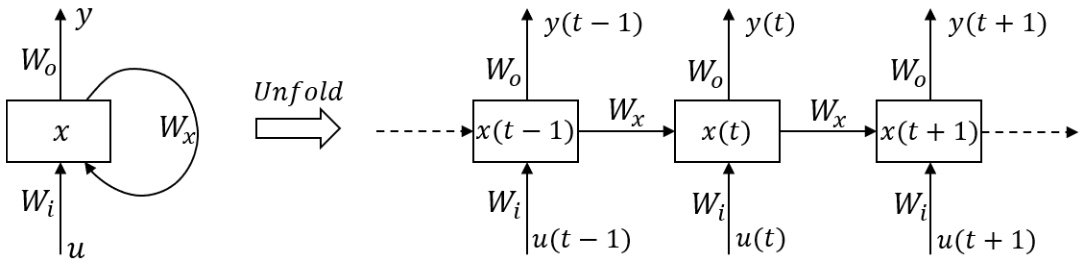


Figure 1.7. Schematic representation of the structure of a recurrent NN (left) and of an unrolled RNN (right).

As shown in Fig. 1.7, it is possible to observe the self-loop that characterises RNNs; also, through the so-called unfolding operation, we can transform the network into a common FFNN since all the loops are removed from its structure. Let us notice that the output y is obtained multiplying the state x by the weights W_y. Also, notice that the input of the generic neuron i is given by the weighted sum of both $u(t)$ and $x(t-1)$: indeed, the last state $x(t-1)$ is used as input for the neuron at the next time instant t, multiplied by W_x. Let us point out that the set of weights of neuron i associated to these two quantities (respectively called W_i and W_x) are generally different one from the other, i.e. $W_i \neq W_x$.

Indicating with f the generic activation function and with u its input, the hidden state x and the output y at time instant t (neglecting the bias terms) are equal to:

$$\begin{aligned} x(t) &= f\Big(W_i^T \cdot u(t) + W_x^T \cdot x(t-1)\Big) \\ y(t) &= W_o^T \cdot x(t) \end{aligned}$$

It is straightforward that, since the output of a recursive neuron $y(t)$ is dependent from $x(t)$ (which is dependent from $u(t)$ and $x(t-1)$), and since $x(t-1)$ is dependent from $u(t-1)$ and $x(t-2)$ (and so on), then $y(t)$ is dependent from $u(t),\ u(t-1), .., \ u(0)$: this allows us to show that a recurrent neuron (and consequently, a RNN) has a sort of memory due to the correlation among the current output and all the previous inputs from instant 0 up to the current one.

Analysing a bit more in detail the structure of a RNN observed in Figure 1.7, and considering the description proposed by Bianchi et al. in [13], these networks are characterised by three types of layers: the input layer, the hidden layer and the output layer. Also, a loop on the hidden unit is observable. As described, it is possible to unfold the loop in order to transform the RNN into a sort of FFNN, where the same structure is repeated multiple times, pointing out how the previous state $x(t-1)$ is used as input to its successor $x(t)$.

Considering Figure 1.8, and indicating with W_i, W_x, W_o the weight matrices respectively for input u, state x and output y, it is clear how these matrices are always the same throughout the whole RNN: it means that the weights for input $u(i)$ and the weights for input $u(i+1)$ are identical. The same reasoning is followed for the output weights and for the weights associated to the connections between the states of the previous neurons used as inputs for the next ones. In other words, W_i, W_x and W_o never change on the basis of the considered time instant.

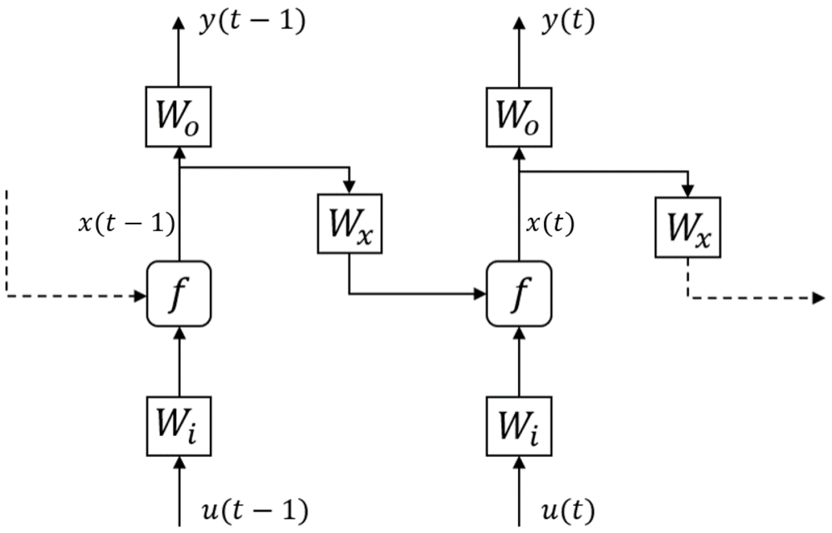


Figure 1.8. Representation of the structure of an unrolled RNN. Notice how W_i, W_x and W_o are always the same even if time instant t changes.

1.3.2 Backpropagation through time

The procedure to train a FFNN has been already described in previous sections, in particular the concept of backpropagation has been introduced in Section 1.2.3. Dealing with a RNN necessarily leads to a change of the tools used to train the network, and of course of the backpropagation method, since we are dealing with a structure where the flow of signals is not unidirectional (the graph of a RNN is cyclic). That is why the so-called *BackPropagation Through Time* (BPTT) is introduced [14]: it consists in unrolling the RNN through time (see Fig. 1.8), and then applying the backpropagation procedure described in Section 1.2.3; let us recall that unrolling a RNN leads to a structure that is essentially a FFNN, with the only difference that the weights matrices W_i, W_x and W_o are replicated across the layers.

Basically, once the unfolding of the RNN is performed, the training behaves almost identically to what has been shown for a FFNN: shortly, a feedforward passage is applied to compute predictions and, consequently, the loss function (indicated with J). Then, its partial derivatives with respect to the weights are computed and backpropagated through the unrolled network: eventually, the gradient descent (Section 1.2.1) is applied to change the parameters of the network.

However, the situation in this case is a bit different from the mathematical point of view: the total loss for a given sequence of inputs (paired with a sequence of outputs of the same length) is given by the sum of the losses through all the time steps, i.e. the overall loss J_{tot} is such that $J_{tot} = \sum_t J_t$, where J_t is the loss evaluated only on the time instant t.

Therefore, as the loss is summed up to compute the total one, the gradients need to be summed up as well. Let us recall the current state $x(i)$ (indicated also as x_i) depends from its input but also from the previous states $x(i-1)$, $x(i-2)$ (i.e. x_{i-1}, x_{i-2}) and so on, hence the gradient of the error with respect to the weights W_x at time instant t does not depend only from the input but also from the gradients of the previous states at all previous time instants.

For example, let us indicate with J_3 the loss at time-step 3, then the following relationships hold:

$$\frac{\partial J_3}{\partial W_o} = \frac{\partial J_3}{\partial y_3}\frac{\partial y_3}{\partial W_o} \tag{1.5}$$

$$\frac{\partial J_3}{\partial W_x} = \frac{\partial J_3}{\partial y_3}\frac{\partial y_3}{\partial x_3}\frac{\partial x_3}{\partial W_x} + \frac{\partial J_3}{\partial y_3}\frac{\partial y_3}{\partial x_3}\frac{\partial x_3}{\partial x_2}\frac{\partial x_2}{\partial W_x} + \frac{\partial J_3}{\partial y_3}\frac{\partial y_3}{\partial x_3}\frac{\partial x_3}{\partial x_2}\frac{\partial x_2}{\partial x_1}\frac{\partial x_1}{\partial W_x} \tag{1.6}$$

Notice we assumed that the first time-step is 1 and not 0. The Eq. (1.6) can be generalized by changing J_3 with J_t and extending the partial derivatives up to time instant t.

1.3.3 Truncated backpropagation through time

Using BPTT algorithm may lead to the same problems that affect FFNNs, e.g. vanishing and exploding gradient (Section 1.3.5), especially if the unrolled network is a deep network, which in other words means the input sequence is particularly long (keep in mind that applying BPTT approach, Section 1.3.2, to train a RNN means unfolding it a number of times equal to the number of time-steps present in the input sequence). This is the main reason why a variation of BPTT is introduced, called *Truncated BackPropagation Through Time* (TBPTT), that relies on the usage of shorter input sequences through which we feed the network. For example, let us imagine the original input sequence is formed by 10000 data samples: the training could be extremely time consuming and computationally tough since the network has to be unrolled 10000 times. However, by dividing such sequence into 50 subsequences of 200 samples each, we can reduce consistently the training complexity and the depth of the unrolled network. Hence, in this way the network will be unfolded for a much lower number of time-steps than the ones with the full sequence.

As a direct consequence, TBPTT may prevent the NN from learning long-term patterns [15] exceeding the length of the subsequence, denoted by l_{batch}. For example, if there exists a specific pattern that could be observed every 2000 data, dividing the original dataset into subsequences smaller than $l_{batch} = 2000$ may prevent the network to capture that peculiarity. However, this problem could be partially limited imposing an overlap between consecutive subsequences. This aspect will be better analysed in Section 4.3.

Summarizing in a formal way, the TBPTT can be seen as a variation of BPTT where the structure of the network, instead of being unrolled an infinite number of times (as could be done in theory with BPTT), is replicated a finite number of times, limiting the probability of vanishing gradient problem and the depth of the network itself [13]. More specifically, two parameters can be introduced, as in [16]:

- k_1, which is the number of time-steps every which the BPTT is performed (processing the sequence one time-step at a time). Notice that $k_1 \leq T_{inp}$, where T_{inp} is the time length of the original input sequence;
- k_2, which is the number of time-steps for which BPTT is performed. It means that, if t is the starting time instant (with t multiple of k_1 due to its previous assumption), BPTT is performed from t to $(t - k_2)$.

Starting from this notation, the TBPTT(k_1, k_2) can assume different meanings based on the values of its two parameters, for example TBPTT(T_{inp}, T_{inp}) (i.e. $k_1 = k_2 = T_{inp}$) is the classical BPTT approach since the whole sequence is analysed before running the algorithm and the updates are performed considering all time-steps in the sequence itself.

In Tensorflow, the implementation of TBPTT can be easily done dividing the input sequence into more subsequences, considering each of them as a separate training batch for the network.

1.3.4 Optimizer

As previously mentioned, the aim of the training is to minimize the loss function J, which means to minimize the mismatch between predicted outputs and real ones. To do that, as discussed in Section 1.2.1, the gradient descent (and its variations) can be used. However, when a deep NN is faced, this method is very slow and not optimal at all. That is why in our code it will be replaced by the so-called *RMSProp optimizer*, that takes usage of *momentum optimization*, introduced by Polyak in [17]: the momentum optimization relies not only on the last computed gradient, but also on a momentum, based on the gradients computed in the previous iterations. Therefore, the equations (taking into account the discussion in [11]) are:

$$\begin{aligned} m &\leftarrow \beta \cdot m + \eta \cdot \nabla_\theta J(\theta) \\ \theta &\leftarrow \theta - m \end{aligned} \tag{1.7}$$

where m is the update term (also called momentum vector) while β is a friction term that ranges from 0 to 1, and it is used to prevent that the moment grows too much. It can be noticed how the m term is used to accumulate the past gradients, since the momentum vector at time instant t (indicates as m_t) is dependent from m_{t-1} (therefore from $\nabla_\theta J(\theta_{t-1})$), which is dependent from m_{t-2} and so on. Thus:

$$m_t = \beta^2 \eta \nabla_\theta J(\theta_{t-3}) + \beta \eta \nabla_\theta J(\theta_{t-2}) + \eta \nabla_\theta J(\theta_{t-1}) + ... = \eta \sum_{\tau=1}^{t-1} \beta^{t-\tau-1} \nabla_\theta J(\theta_\tau)$$

Let us highlight that momentum optimization is based on the usage of a learning rate η that is kept always the same throughout the whole procedure. Therefore, an alternative method called *AdaGrad optimizer* (introduced in [18]) that adapts the learning rate with respect to the parameters is introduced. From a mathematical point of view, we get:

$$\begin{aligned} s &\leftarrow s + \nabla_\theta J(\theta) \circ \nabla_\theta J(\theta) \\ \theta &\leftarrow \theta - \eta \cdot \nabla_\theta J(\theta) \oslash \sqrt{s + \epsilon} \end{aligned} \tag{1.8}$$

where $\oslash$ is the element-wise division, ϵ a small constant for numerical stability (used to avoid a division by zero) and η the learning rate.

The first step of Eq. (1.8) aims to store the square of the gradients into the vector s, and then we use it for the parameters update. It is straightforward to notice that the method, starting from an initial standard value for the learning rate, decays it at every iteration due to the element-wise division by $\sqrt{s + \epsilon}$. In particular, it means that each parameter uses a different learning rate at every time instant, achieving smallest η for the steepest directions (high values of s_i), while bigger learning rates are used for the other ones (where we record low values of s_i). Thus, this method eliminates the manually tuning of η.

The main drawback of AdaGrad optimization is that the vector s (which stores the square of the gradients) gets higher and higher at every iteration (since all the summed elements are greater or equal to zero), leading to a decrease of η, which could become even very close to zero (not useful for the training of deep NNs).

Let us notice by a quick comparison between Eq. (1.4) and Eq. (1.7) that gradient descent completely ignores all the already computed gradients, but it updates the parameters θ just on the basis of the current gradient. Starting from this approach, and considering Eq. (1.8) where all the gradients are taken into account from initial to current time instants, the RMSProp optimizer (Tieleman et al. [19]) is obtained: it implements a variation on which past gradients are considered, storing only the ones of the most recent iterations (and not all the gradients computed up to now). From a mathematical point of view, let us consider Eq. (1.9) where β is the decay rate.

$$\begin{aligned} s &\leftarrow \beta \cdot s + (1-\beta)\nabla_\theta J(\theta) \circ \nabla_\theta J(\theta) \\ \theta &\leftarrow \theta - \eta \cdot \nabla_\theta J(\theta) \oslash \sqrt{s+\epsilon} \end{aligned} \tag{1.9}$$

For the sake of knowledge, a direct alternative to RMSProp optimizer is the so-called *Adam optimizer* (*Adaptive Moment Estimation*), discussed in [20]. More specifically, it can be seen as a blend of RMSProp and momentum methods, as noticeable from Eq. (1.10) where k indicates the number of the iteration (starting from 1), β_1 and β_2 indicate respectively the momentum decay hyperparameter and the scaling decay hyperparameter.

$$\begin{aligned} m &\leftarrow \beta_1 \cdot m + (1-\beta_1)\nabla_\theta J(\theta) \\ s &\leftarrow \beta_2 \cdot s + (1-\beta_2)\nabla_\theta J(\theta) \circ \nabla_\theta J(\theta) \\ m &\leftarrow \frac{m}{1-\beta_1^k} \\ s &\leftarrow \frac{s}{1-\beta_2^k} \\ \theta &\leftarrow \theta - \eta \cdot m \oslash \sqrt{s+\epsilon} \end{aligned} \tag{1.10}$$

Clearly, even Adam optimizer does not require a manual tuning of the learning rate.

1.3.5 Vanishing and exploding gradient problems

Two common problems only mentioned up to now that affect the training of the deeper layers of the networks are the so-called *vanishing* and *exploding gradients*, that will be here briefly discussed, and treated by Aurélien Géron in [11]. Let us recall in few words that backpropagation algorithm works propagating from output layer to input one the error gradient and, subsequently, computing the gradients of the cost function with respect to each parameter of the network and using these gradients to update the weights of the network itself. However, sometimes it may happen that such gradients become smaller and smaller across layers and, once the deeper layers (which are the first layers of the network, since backpropagation algorithm works backwards) are reached, their values are almost zero, which means the weights in that part of the networks are not updated at all (i.e. it is like that portion of the network is never trained): this phenomenon is called *vanishing gradient*.

On the other hand, the opposite case may be observed: the gradients become larger and larger at every layer, hence the deeper layers go through an insanely large weights update. This problem is known as *exploding gradient*.

In [21], some explanations of these two phenomena have been proposed. In particular, a correlation between the choice of the activation function and the initialization procedure for network weights has been shown by the authors, describing how the variance of the output of the layer increases at each layer if the initialization and the activation functions are not selected carefully.

Different solutions have been proposed in [21], such that the usage of a new initialization like the *Xavier initialization* (considering as activation function f the sigmoid σ_g, Fig. 1.3) in order to avoid the neuron activation functions start in saturated regions (i.e. with values of the weights that are too small or too high). In particular, it initializes the weights of the network by drawing them from a normal distribution with zero mean and a variance function of n_{in} and n_{out}, which are respectively the number of input (*fan-in*) and output (*fan-out*) connections of the layer. The results indicated that, considering Xavier initialization, the network maintained near identical variances of its weight gradients across the layers.

An additional solution could be to take the activation function neither as the sigmoid nor the hyperbolic tangent, but as the ReLU function, introduced in Section 1.1.2. With such choice, a new initialization proposed in [22], called *He initialization*, could be considered: it is almost equal to Xavier initialization, with the only difference that the variance is multiplied by a factor two, i.e. $Var_{He} = 2 \cdot Var_{Xavier}$.

However, one of the main tools to avoid vanishing and exploding gradients are the usage of gated units for the network, e.g. LSTM and GRU cells that will be discussed in the Sections 1.3.6 and 1.3.7 respectively.

1.3.6 Long Short-Term Memory cell

A common problem in RNNs is that they may lose gradually stored informations about the first inputs they went through if the training (or even the input sequence) is too long. That is why long memory cells have been proposed during the years, starting from the so-called *Long Short-Term Memory cell* (LSTM cell), introduced by Hochreiter and Schmidhuber in [23]. The basic behaviour of a LSTM cell is to recognize if the received input is important or not, store the important informations contained in it for a long-term period until they are not required anymore, and retrieve such contents when needed.

From a deeper point of view, considering [11], [24] and [25], a LSTM cell has its state split into two different ones: $\xi(t)$ (the short-term state) and $x(t)$ (the long-term state). Going through the LSTM cell, $x(t)$ undergoes three gates called respectively *input gate*, *output gate* and *forget gate*, each one with a specific role in terms of providing the next long-term state $x(t+1)$ and short-term one $\xi(t+1)$, which is likely equal to the cell output $y(t+1)$ (notice a linear transformation might be placed in between). Going into details of the fundamental quantities of a LSTM cell, we analyse:

- $f(t)$, called also forget gate, defined as the output of a sigmoid layer that takes as inputs both $\xi(t-1)$ and $u(t)$, and defines which contents of $x(t-1)$ need to be kept or discarded (totally or only partially since the sigmoid function varies from 0 up to 1);
- $g(t)$, called also candidate cell state, defined as the output of a *tanh* (σ_c) layer that takes as inputs both $\xi(t-1)$ and $u(t)$;
- $i(t)$, called also input gate, defined as the output of a sigmoid layer that takes as inputs both $\xi(t-1)$ and $u(t)$. Together with $g(t)$, they will define which part of the new input will be stored in the cell state;
- $o(t)$, called also output gate, defined as the output of a sigmoid layer that takes as inputs both $\xi(t-1)$ and $u(t)$. It will be useful to define, as it will be shown, $\xi(t)$ and consequently $y(t)$.

Therefore, the equations associated to the three gates plus the $g(t)$ quantity we have just defined are:

$$\begin{aligned} Forget\ gate:&\ f(t)=\sigma_g(W_f\cdot u(t)+U_f\cdot\xi(t-1)+b_f)\\ Input\ gate:&\ i(t)=\sigma_g(W_i\cdot u(t)+U_i\cdot\xi(t-1)+b_i)\\ Output\ gate:&\ o(t)=\sigma_g(W_o\cdot u(t)+U_o\cdot\xi(t-1)+b_o)\\ Candidate\ cell\ state:&\ g(t)=\sigma_c(W_c\cdot u(t)+U_c\cdot\xi(t-1)+b_c)\end{aligned}\tag{1.11}$$

and the resulting equations for a LSTM cell, considering (1.11), can be described by:

$$\begin{aligned} x(t)&=f(t)\circ x(t-1)+i(t)\circ g(t)\\ \xi(t)&=o(t)\circ\sigma_c(x(t))\\ y(t)&=U_o\cdot\xi(t)+b_o\end{aligned}\tag{1.12}$$

where $u\in\mathcal{R}^{n_u}$ is the input, $y\in\mathcal{R}^{n_y}$ is the output, $x\in\mathcal{R}^{n_x}$ is the *hidden state*, $\xi\in\mathcal{R}^{n_x}$ is the *output state*, σ_g is the sigmoid function (Fig. 1.3), σ_c is the *tanh* function (Fig. 1.4), $\circ$ is the element-wise multiplication (Hadamard product), while the terms $W_f,\ W_i,\ W_o,\ W_c\in\mathcal{R}^{n_x\times n_u},\ U_f,\ U_i,\ U_o,\ U_c\in\mathcal{R}^{n_x\times n_x},\ U_o\in\mathcal{R}^{n_y\times n_x}$ are the weighting matrices and $b_f,\ b_i,\ b_o,\ b_c\in\mathcal{R}^{n_x},\ b_o\in\mathcal{R}^{n_y}$ are the biasing vectors.

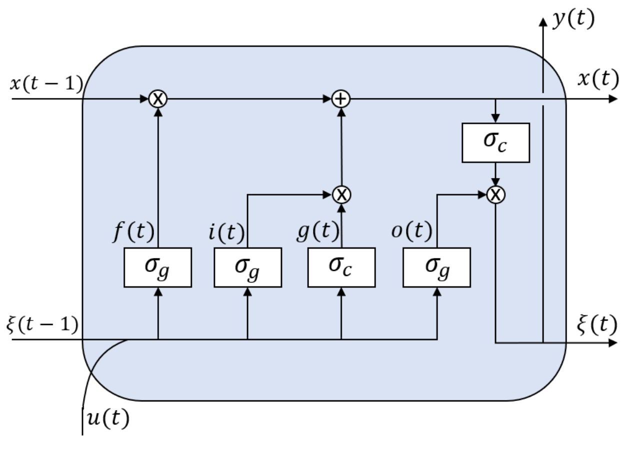


Figure 1.9. Schematic representation of a LSTM cell.

At this point, we are able to have an intuitive idea on how a LSTM cell works, and in particular let us focus on the path that $x(t-1)$ goes through along the cell (see Fig. 1.9):

- the first thing is to remove the useless informations from the vector through the forget state, so a multiplication between $x(t-1)$ and $f(t)$ is performed;
- then, the new informations need to be added to the hidden state, so they are summed to the product (element-wise multiplication) between $i(t)$ and the quantity $g(t)$ previously described;
- at this point, the new hidden state $x(t)$ is obtained, however the output state $\xi(t)$ is still unknown, therefore $x(t)$ goes through the σ_c function and then it is multiplied by $o(t)$ (element-wise multiplication);
- the output $y(t)$ can be either equal to $\xi(t)$ or going through a linear transformation as indicated in Eq. (1.12). Notice in Fig. 1.9 the linear transformation has been neglected for simplicity.

Extending Eq. (1.12) for a multi-layer LSTM network, the result is:

$$x^{(i)+} = f^{(i)} \circ x^{(i)} + i^{(i)} \circ \sigma_c(W_c^{(i)} \cdot x^{(i-1)+} + U_c^{(i)} \cdot \xi^{(i)} + b_c^{(i)}) \tag{1.13}$$

where the superscript $^{(i)}$ and $^{+}$ denote the layer i and the quantity value at the next time instant respectively.

In conclusion, the LSTM cell is capable of memorizing relevant features and informations passed as input for long time thanks to the long-term state recalling them every time it is needed, discarding the useless ones.

1.3.7 Gated Recurrent Unit

A direct alternative to LSTM cell is the so-called *Gated Recurrent Unit cell* (GRU cell), proposed by Kyunghyun Cho et al. [26].

GRU can be seen as an exemplification of LSTM since the single cell takes usage of just two gates instead of three as viewed before. In fact, in this case we deal with *update gate* and *reset gate* only, while the output gate is removed, and a single state vector is considered instead of the two of LSTM cells.

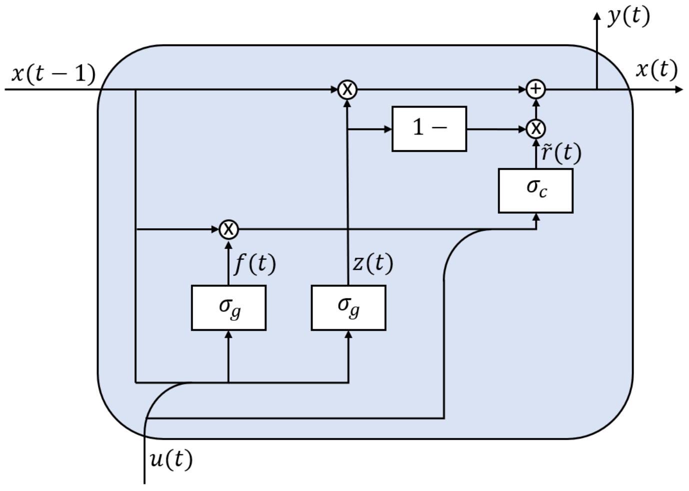


Figure 1.10. Schematic representation of a GRU cell.

Notice the absence of an output gate means that once the state vector is computed, it is used as output too: indeed the output vector is identical to the state one (see Fig. 1.10). However, in some cases a linear output transformation could be applied to $x(t)$ in order to obtain $y(t)$.

Dealing with equations, and being concordant with the notation used in Section 1.3.6, the GRU cell can be described by:

$$\begin{aligned} z(t) &= \sigma_g(W_z \cdot u(t) + U_z \cdot x(t) + b_z) \\ f(t) &= \sigma_g(W_f \cdot u(t) + U_f \cdot x(t) + b_f) \\ \tilde{r}(t) &= \sigma_c(W_r \cdot u(t) + U_r \cdot (f(t) \circ x(t)) + b_r) \\ x(t) &= z(t-1) \circ x(t-1) + (1 - z(t-1)) \circ \tilde{r}(t-1) \end{aligned} \tag{1.14}$$

where $u \in \mathcal{R}^{n_u}$ is the input, $x \in \mathcal{R}^{n_x}$ is the state, $z(t)$ is the update gate, $f(t)$ is the reset gate, $\tilde{r}(t)$ the *candidate hidden state*, σ_g is the sigmoid function (Fig. 1.3), σ_c is the *tanh* function (Fig. 1.4), $\circ$ is the element-wise multiplication (Hadamard product), while the terms $W_z,\ W_f,\ W_r \in \mathcal{R}^{n_x \times n_u}$, $U_z,\ U_f,\ U_r \in \mathcal{R}^{n_x \times n_x}$ are the weighting matrices and $b_z,\ b_f,\ b_r \in \mathcal{R}^{n_x}$ are the biasing vectors. Considering a linear transformation on the output, we would get:

$$y(t) = U_o \cdot x(t) + b_o$$

where $y \in \mathcal{R}^{n_y}$, $U_o \in \mathcal{R}^{n_y \times n_x}$ and $b_o \in \mathcal{R}^{n_y}$.

Extending Eq. (1.14) for a multi-layer GRU network, the result is:

$$\begin{aligned} z^{(i)} &= \sigma_g(W_z \cdot x^{(i-1)+} + U_z \cdot x^{(i)} + b_z) \\ f^{(i)} &= \sigma_g(W_f \cdot x^{(i-1)+} + U_f \cdot x^{(i)} + b_f) \\ \tilde{r}^{(i)} &= \sigma_c(W_r^{(i)} \cdot x^{(i-1)+} + U_r^{(i)} \cdot (f^{(i)} \circ x^{(i)}) + b_r^{(i)}) \\ x^{(i)+} &= z^{(i)} \circ x^{(i)} + (1 - z^{(i)}) \circ \tilde{r}^{(i)} \end{aligned} \tag{1.15}$$

where the superscript $^{(i)}$ denotes the layer i and the superscript $^{+}$ denotes the value at the next time instant. Notice that the input of the i-th layer is provided by the state of the $(i-1)$th layer, where $x^{(1)} = u$ (i.e. the input of the network).

Let us highlight that, when the update gate is close to 1, the new state $x(t)$ is simply the old one (i.e. $x(t-1)$) retrieved, hence the input $u(t)$ is almost ignored. On the other hand, if $z(t) \simeq 0$, the new state $x(t)$ will be quite similar to $\tilde{r}(t)$.

In [28] the discussion of *Input-to-State Stability* (ISS) for nonlinear discrete-time system is treated. In particular, in [27] sufficient conditions for ISS of a GRU network (both for single layer and multi-layer discrete-time networks) are provided. First of all, the ISS property guarantees that the effects of initial conditions progressively vanish, together with the boundedness of the state trajectories (i.e. of the network's state) generated by bounded inputs or bounded disturbances. This allows us to avoid to affect the performances of NNs due to (possible) wrong initialization of the network's parameters.

More formally, let us define with $\mathcal{K}$ the set of functions γ such that they are continuous, strictly increasing and with $\gamma(0) = 0$, with $\mathcal{K}_\infty$ the set of unbounded $\gamma \in \mathcal{K}$ and with $\beta \in \mathcal{KL}$ a function such that $\beta(\cdot, t) \in \mathcal{K}$ for all $t \geq 0$ and $\beta(s, \cdot)$ continuous and strictly decreasing for all $s > 0$. Then, considering the generic single layer GRU network indicated in (1.14), its ISS definition is provided in Definition 1.3.1 (as explained in [27]).

Definition 1.3.1 (ISS) *System* (1.14) *is input-to-state stable if there exist functions $\beta(\|\bar{x}\|_\infty, t) \in \mathcal{KL}$, $\gamma_u(\|u\|_\infty) \in \mathcal{K}_\infty$, $\gamma_b(\|b_r\|_\infty) \in \mathcal{K}_\infty$ such that the following relationship holds*

$$\|x(t, \bar{x}, u, b_r)\|_\infty \leq \ \beta(\|\bar{x}\|_\infty, t) + \gamma_u(\|u\|_\infty) + \gamma_b(\|b_r\|_\infty) \tag{1.16}$$

for any $t \geq 0$, any input u, any initial condition $\bar{x}$ and any value of b_r.

The sufficient condition for the ISS of a single layer GRU network is provided by the following constraint:

$$\|U_r\|_\infty \, \sigma_g\Big(\|W_f \quad U_f \quad b_f\|_\infty\Big) < 1 \tag{1.17}$$

Also, it has been proved that GRU are such that, for any initial state $\bar{x} \in \mathcal{R}^{n_x}$, there exists a finite $\bar{k} \geq 0$ at which $x(k) \in \mathcal{X} \ \forall \ k \geq \bar{k}$, where $\mathcal{X} = [-1, 1]^{n_x}$ is an invariant set of the system. It means that, whatever the initialization of the network state is, and for any input $u : \ u \in [-1, 1]^{n_u}$, the state will be in $[-1, 1]$ from a certain time instant onward. For further details and the proofs of these statements, the interested readers are addressed to [27], where the explanation is extended also to multi-layer GRU networks.

1.4 Conclusions

In Chapter 1 the concept of neural network has been discussed, focusing on the basic element, i.e. the neuron (together with the main types of activation functions f), and analysing both FFNNs and RNNs. Also, the training for both these networks has been described, highlighting the main useful tools, like gradient descent algorithm, backpropagation, BPTT and TBPTT.

The two main problems of RNN training, i.e. vanishing and exploding gradients, were proposed and briefly discussed. At this purpose, we went into details about the gated units introduced to solve such problems, that are the LSTM and GRU cells, focusing on their main mathematical relationships and properties.

Chapter 2

Benchmark: the four-tanks system

In this chapter, it will be introduced the benchmark considered in the Thesis, describing it in terms of physical characteristics and mathematical properties.

In particular, the system is the four-tanks plant proposed by Johansson in [1], and discussed by Alvarado et al. [29]. The physical constraints will be highlighted, describing the values of the system parameters in the real world. More specifically, the mathematical model (together with its linearisation) will be presented, then the steady-state conditions for the nonlinear system, the analysis of the zeros of the transfer matrix and the step (and frequency) responses of the linear system will be studied. It will be also treated the implementation of the plant in Simulink using a MATLAB function.

2.1 Mathematical model of the plant

The system consists in four interconnected water tanks with two input pumps that allow the water to reach all of them. It can be noticed from Figure 2.1 as the input flows to the tanks are regulated by two valves that control the outflow of the corresponding pumps. Another element that can be noticed is that the pumps take the water from a storage tank below the plant, that in our case is considered to have an infinite capacity. In addition, the two upper tanks (whose levels are called for simplicity h_3 and h_4) outlet the water directly in the two tanks below (with levels h_1 and h_2 respectively).

Since it is needed to introduce a mathematical model for the plant, a name is assigned to each variable:

- v_i: voltage used to control the input pump i, for $i \in \{1, 2\}$;
- y_i: output of the tank i, for $i \in \{1, 2\}$;
- h_i: level of the tank i, for $i \in \{1, 2, 3, 4\}$;
- γ_i: opening position of the valve i, for $i \in \{1, 2\}$, $\gamma_i \in [0, 1]$;
- S: cross-section of the outlet hole of tank (the same for each tank by assumption);
- q_i: flow of the pump i, for $i \in \{a, b\}$;

- a_i: discharge constant of the tank i, for $i \in \{1, 2, 3, 4\}$.

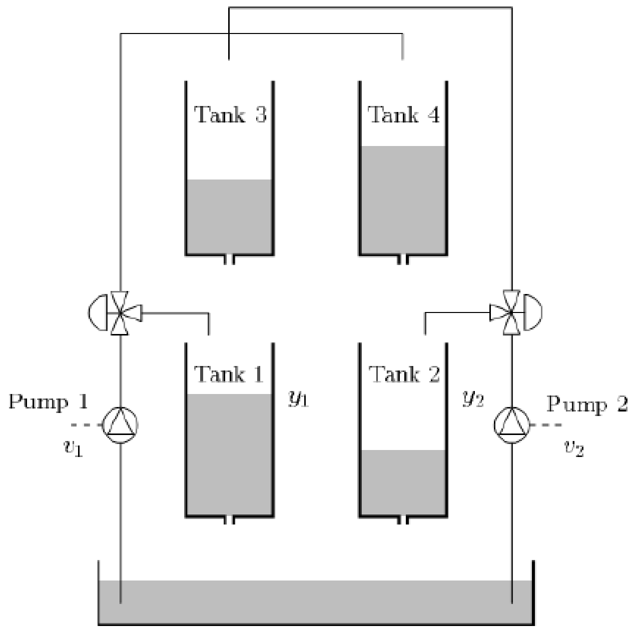


Figure 2.1. Model of the four-tank system proposed by Johansson in [1].

As indicated in [29], the mathematical system that properly describes the evolution of the state variables h_i is the following one:

$$\begin{cases} \dfrac{dh_1}{dt} = -\dfrac{a_1}{S}\sqrt{2gh_1} + \dfrac{a_3}{S}\sqrt{2gh_3} + \dfrac{\gamma_1}{S}q_a \\ \dfrac{dh_2}{dt} = -\dfrac{a_2}{S}\sqrt{2gh_2} + \dfrac{a_4}{S}\sqrt{2gh_4} + \dfrac{\gamma_2}{S}q_b \\ \dfrac{dh_3}{dt} = -\dfrac{a_3}{S}\sqrt{2gh_3} + \dfrac{(1-\gamma_2)}{S}q_b \\ \dfrac{dh_4}{dt} = -\dfrac{a_4}{S}\sqrt{2gh_4} + \dfrac{(1-\gamma_1)}{S}q_a \end{cases} \tag{2.1}$$

Clearly, the model is just an exemplification of the real system, since some aspects (like for example the turbulences inside the tanks) are ignored. In particular, it can be observed that the obtained model is nonlinear due to the presence of square roots in the expressions of all the $\dot{h}_i$.

Because of this non linearity, to analyse the relevant mathematical properties it should be provided a linearised version of the model. To do that, it is required first of all to define a set of nominal values, in addition to the constraints defined by the real plant (e.g. the maximum levels of the tanks, the maximum flows of the input pumps and so on).

Table 2.1 includes the values that will be taken into account for the linearisation and also for the implementation of the model in Simulink. Notice that, for simplicity, the four tanks are directly indicated by their corresponding levels h_i.

Name of the variable	Value	Unit
a_1	1.31e-4	m^2
a_2	1.51e-4	m^2
a_3	9.27e-5	m^2
a_4	8.82e-5	m^2
h_{min}	0	m
$h_{1,max}$	1.36	m
$h_{2,max}$	1.36	m
$h_{3,max}$	1.3	m
$h_{4,max}$	1.3	m
S	0.06	m^2
γ_1	0.3	
γ_2	0.4	
h_1^0	0.65	m
h_2^0	0.66	m
h_3^0	0.65	m
h_4^0	0.66	m
q_a^0	1.63	m^3/h
q_b^0	2.00	m^3/h
$q_{a,max}$	3.26	m^3/h
$q_{b,max}$	4	m^3/h
q_{min}	0	m^3/h
g	9.81	m/s^2

Table 2.1. Table that summarize the values of the parameters used for the plant and for the linearisation.

By defining the linearisation variables x_i $(i = 1, 2, 3, 4)$ and u_j $(j = 1, 2)$ as it follows:

$$\begin{aligned} x_i &= h_i - h_i^0 \\ u_k &= q_k - q_k^0, \quad k \in \{a, b\} \end{aligned} \tag{2.2}$$

it is obtained the linearised model in the form:

$$\begin{cases} \dot{x} = \dfrac{dx}{dt} = Ax + Bu \\ y = Cx \end{cases} \tag{2.3}$$

where $x = [x_1,\ x_2,\ x_3,\ x_4]^T$, $y = [x_1,\ x_2]^T$ and $u = [u_1,\ u_2]^T$. The matrices A, B and C have the following forms:

$$A = \begin{bmatrix} -\frac{1}{T_1} & 0 & \frac{1}{T_3} & 0 \\ 0 & -\frac{1}{T_2} & 0 & \frac{1}{T_4} \\ 0 & 0 & -\frac{1}{T_3} & 0 \\ 0 & 0 & 0 & -\frac{1}{T_4} \end{bmatrix}$$

$$B = \begin{bmatrix} \frac{\gamma_1}{S} & 0 \\ 0 & \frac{\gamma_2}{S} \\ 0 & \frac{1-\gamma_2}{S} \\ \frac{1-\gamma_1}{S} & 0 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

where $T_i = \frac{S}{a_i\sqrt{\frac{2h_i^0}{g}}}$ (for $i = 1,\ 2,\ 3,\ 4$) is the time constant of the tank i. As observable in matrix C, the outputs y considered are only the levels of the lower tanks h_1 and h_2.

Taking into account the values in Table 2.1 and from the structure of the matrix A, it is clear that the model has four stable real poles (i.e. $-\frac{1}{T_i}$, $i = 1, .., 4$). The transfer functions computed by the matrices are:

$$G(s) = \begin{bmatrix} \frac{\gamma_1 c_1}{1+sT_1} & \frac{(1-\gamma_2)c_1}{(1+sT_1)(1+sT_3)} \\ \frac{(1-\gamma_1)c_2}{(1+sT_4)(1+sT_1)} & \frac{\gamma_2 c_2}{1+sT_2} \end{bmatrix} \tag{2.4}$$

with $c_1 = T_1/S$, $c_2 = T_2/S$.

2.1.1 Steady-state conditions of the nonlinear system

Taking into account the nonlinear model proposed in System (2.1), and a stationary operating point (h_1^0, h_2^0, q_a^0, q_b^0), at the steady-state the last two equations become:

$$\begin{aligned} \frac{a_3}{S}\sqrt{2gh_3^0} &= \frac{(1-\gamma_2)}{S}q_b^0 \\ \frac{a_4}{S}\sqrt{2gh_4^0} &= \frac{(1-\gamma_1)}{S}q_a^0 \end{aligned} \tag{2.5}$$

Therefore, replacing Eq. (2.5) in System (2.1), the first two equations become:

$$\begin{aligned} \frac{a_1}{S}\sqrt{2gh_1^0} &= \frac{a_3}{S}\sqrt{2gh_3^0} + \frac{\gamma_1}{S}q_a^0 = \frac{(1-\gamma_2)}{S}q_b^0 + \frac{\gamma_1}{S}q_a^0 \\ \frac{a_2}{S}\sqrt{2gh_2^0} &= \frac{a_4}{S}\sqrt{2gh_4^0} + \frac{\gamma_2}{S}q_b^0 = \frac{(1-\gamma_1)}{S}q_a^0 + \frac{\gamma_2}{S}q_b^0 \end{aligned} \tag{2.6}$$

It follows that there always exists an unique input (q_a^0, q_b^0) giving the stationary point (h_1^0, h_2^0) if and only if the matrix

$$\begin{bmatrix} \gamma_1 & (1-\gamma_2) \\ (1-\gamma_1) & \gamma_2 \end{bmatrix}$$

is non-singular. It means that:

$$\gamma_1\gamma_2 - (1-\gamma_2)(1-\gamma_1) = \gamma_1 + \gamma_2 - 1 \neq 0 \quad \Rightarrow \quad \gamma_1 + \gamma_2 \neq 1.$$

Assuming the singularity is provided, i.e. $\gamma_1 + \gamma_2 = 1$, it can be noticed that the flows through tank 1 and tank 2 are respectively given by:

$$flow_{tank\,1} = (1-\gamma_2)q_b + \gamma_1 q_a = \gamma_1(q_a + q_b) \tag{2.7}$$

$$flow_{tank\,2} = (1-\gamma_1)q_a + \gamma_2 q_b = (1-\gamma_1)(q_a + q_b) \tag{2.8}$$

Analysing Equations (2.7) and (2.8), it is clear how the two flows are dependent, therefore even the corresponding levels h_1, h_2 must be.

2.1.2 Multivariable (invariant) zeros of the linearised system

Let us notice that the positions of the three-way valves determine the location of a multivariable zero of the linearised model [1]. Taking into account the transfer matrix (2.4) previously obtained, the zeros of $G(s)$ are given by the roots of the numerator of its determinant, i.e.:

$$\det(G(s)) = \frac{c_1 c_2}{\gamma_1\gamma_2 \prod_{i=1}^{4}(1+sT_i)}\left[(1+sT_3)(1+sT_4) - \frac{(1-\gamma_1)(1-\gamma_2)}{\gamma_1\gamma_2}\right] \tag{2.9}$$

In particular, the zeros (indicated as z_1, z_2) are given by:

$$\begin{aligned}(1+sT_3)(1+sT_4) - \frac{(1-\gamma_1)(1-\gamma_2)}{\gamma_1\gamma_2} &= s^2 T_3 T_4 + s(T_3+T_4) + \left(1 - \frac{(1-\gamma_1)(1-\gamma_2)}{\gamma_1\gamma_2}\right) = 0 \\ &= as^2 + bs + c = 0\end{aligned}$$

$$\Rightarrow \; z_{1,2} = \frac{-(T_3+T_4) \pm \sqrt{(T_3+T_4)^2 - 4\cdot T_3T_4\left(1 - \frac{(1-\gamma_1)(1-\gamma_2)}{\gamma_1\gamma_2}\right)}}{2\cdot T_3T_4} \tag{2.10}$$

It is straightforward to notice that $G(s)$ has two finite zeros related to the two parameters γ_1, γ_2, which are both limited in the range $[0, 1]$: one zero is always placed in the left half-plane, while the other one can be either in the left or in the right half-plane, based on the values of γ_1, γ_2.

More specifically, the position of the second zero is strictly related to the sign that $\eta = 1 - \frac{(1-\gamma_1)(1-\gamma_2)}{\gamma_1\gamma_2}$ assumes: if $\eta < 0$ it will be in the right half-plane, while for $\eta > 0$ it will be in the left half-plane. Therefore, if $\gamma_1 + \gamma_2 = 1$ the system will have a zero in the origin, if $0 < \gamma_1 + \gamma_2 \leq 1$ it will be non-minimum phase (i.e. $G(s)$ is causal and stable while its inverse $G(s)^{-1}$ is unstable), if $1 < \gamma_1 + \gamma_2 \leq 2$ it will be minimum phase (i.e. both $G(s)$ and $G(s)^{-1}$ are stable). In Figure 2.2 a graphical representation of the situation is provided.

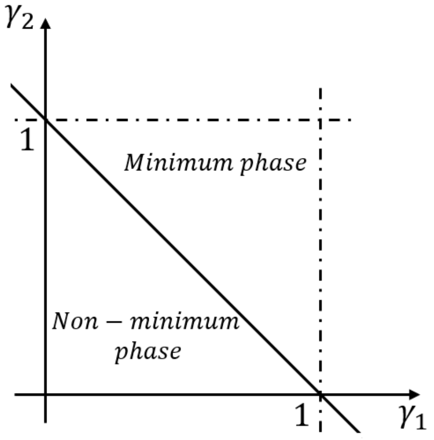


Figure 2.2. Graphical representation of minimum and non-minimum phase behaviours of the system as function of γ_1, γ_2.

Taking into account the two operating points P^-, P^+ taken from [1] (summarised in Table 2.2), and using Eq. (2.10), the obtained zero values are written in Table 2.3.

Variable	P^-	P^+	Unit
$(h_1^0,\ h_2^0)$	$(12.4, 12.7)$	$(12.6, 13.0)$	cm
$(h_3^0,\ h_4^0)$	$(1.8, 1.4)$	$(4.8, 4.9)$	cm
$(v_1^0,\ v_2^0)$	$(3, 3)$	$(3.15, 3.15)$	V
$(k_1,\ k_2)$	$(3.33, 3.35)$	$(3.14, 3.29)$	cm^3/Vs
$(\gamma_1,\ \gamma_2)$	$(0.7, 0.6)$	$(0.43, 0.34)$	
$(T_1,\ T_2)$	$(62, 90)$	$(63, 91)$	
$(T_3,\ T_4)$	$(23, 30)$	$(39, 56)$	

Table 2.2. Summary of the values of the two operating points P^-, P^+.

Variable	P^-	P^+
$(z_1,\ z_2)$	(-0.018, -0.06)	(0.013, -0.057)

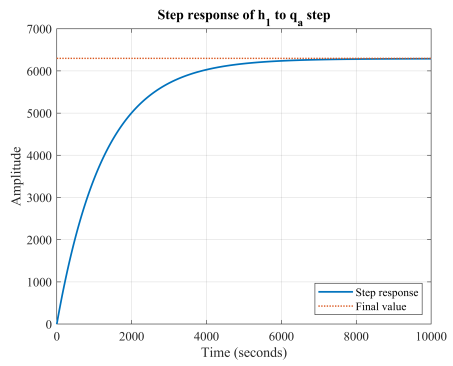
Table 2.3. Table containing the values of the two zeros z_1, z_2 considering the operating points P^-, P^+.

Let us notice that, as expected, the operating point P^+ is characterised by a zero in the right half-plane (indeed, $\gamma_1^+ + \gamma_2^+ = 0.77 < 1$). On the other hand, the operating point P^- has both the zeros in the left half-plane ($\gamma_1^- + \gamma_2^- = 1.3 > 1$). Notice also the position of the poles are given by the T_i terms, therefore if we ensure that $T_i > 0\ \forall\ i$, all the poles are located in the left half-plane.

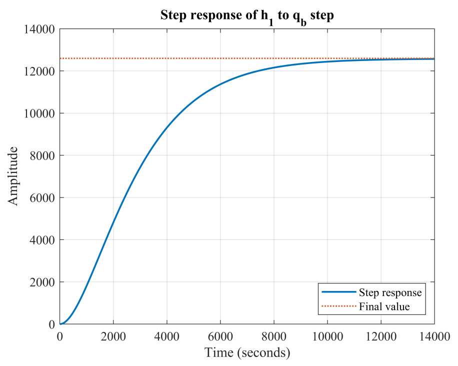
2.1.3 Step responses and Bode diagrams

Dealing with the linearised system at the equilibrium point ($h_1^0 = 0.65\ m,\ h_2^0 = 0.66\ m$) introduced in Table 2.1 together with the other parameters, we can study its frequency characteristics and step responses. From Fig. 2.3, it can be observed that the step responses of h_1, h_2 are almost identical for q_a step and q_b step, with a difference only on the final stationary value (let us point out we consider two unitary steps), while the settling period is very similar. Considering for simplicity only the q_a step response, the settling time of the linearised system is around $0.4 \cdot 10^4$ s for h_1 and almost the double for h_2. Notice that neither overshoots nor undershoots are observed.

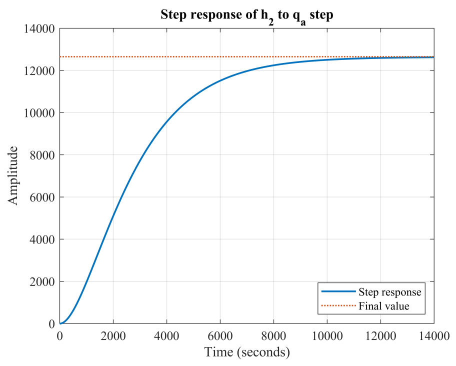
The frequency responses of the elements of the transfer matrix $G(s)$ are reported in Fig. 2.4.



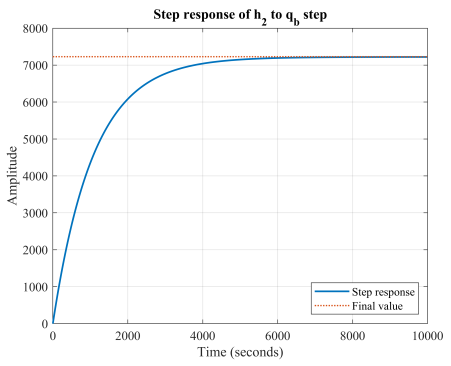
(a)



(b)

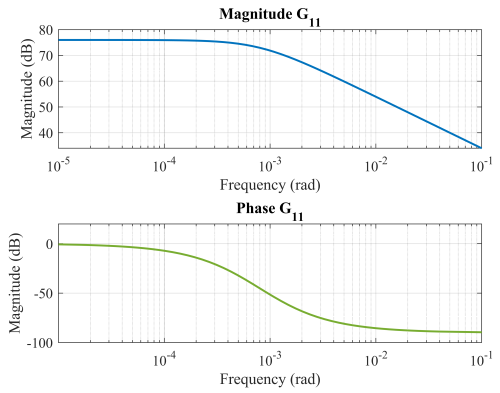


(c)

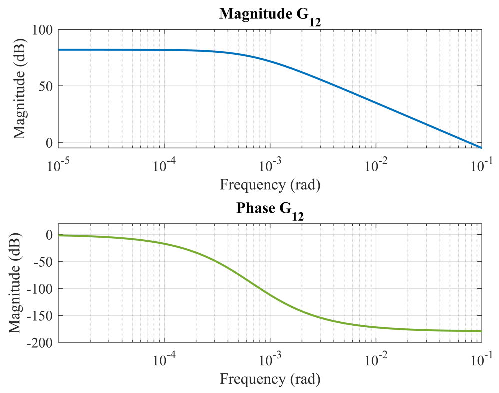


(d)

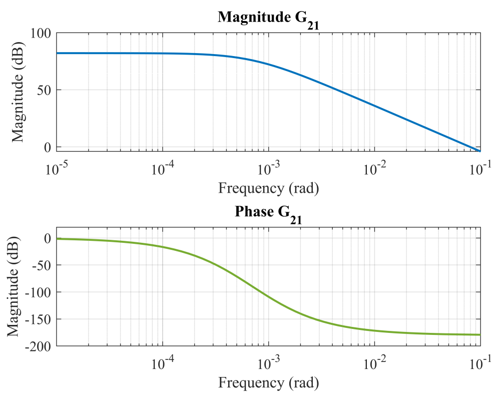
Figure 2.3. (a) Linearised system response of h_1 with respect to q_a step; (b) Linearised system response of h_1 with respect to q_b step; (c) Linearised system response of h_2 with respect to q_a step; (d) Linearised system response of h_2 with respect to q_b step.



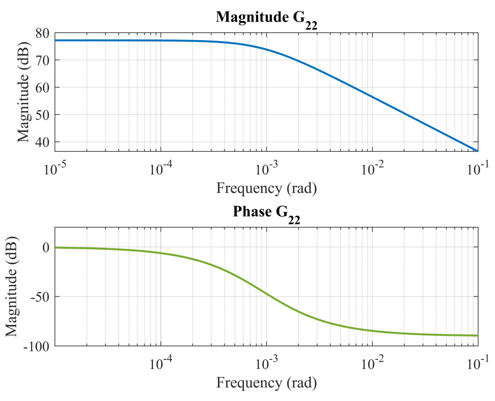
(a)



(b)



(c)



(d)

Figure 2.4. (a) Bode diagram of $G_{11}(s)$; (b) Bode diagram of $G_{12}(s)$; (c) Bode diagram of $G_{21}(s)$; (d) Bode diagram of $G_{22}(s)$.

2.2 Implementation of the plant in Simulink

At this point, the model is ready to be implemented in MATLAB and Simulink, in order to simulate it. Notice the implementation has been done taking into account always the parameters defined in Table 2.1 and Eq. (2.1). Let us point out that some quantities previously defined have a measurement unit that must be changed in order to avoid numerical problems: in particular, all the flows of the two pumps (e.g. $q_{a,max}$, $q_{b,max}$ and so on) were defined in the Section 2.1 in m^3/h. Hence, they need to be changed in m^3/s dividing their values by 3600.

In order to perform the implementation, it has been considered the following MATLAB function:

$$dx = \texttt{fourtankfun}(u,\ a_1,\ a_2,\ a_3,\ a_4,\ S,\ \gamma_a,\ \gamma_b) \tag{2.11}$$

where $dx = [\dot{h}_1,\ \dot{h}_2,\ \dot{h}_3,\ \dot{h}_4]^T$ and $u = [h_1,\ h_2,\ h_3,\ h_4,\ q_a,\ q_b]^T$.

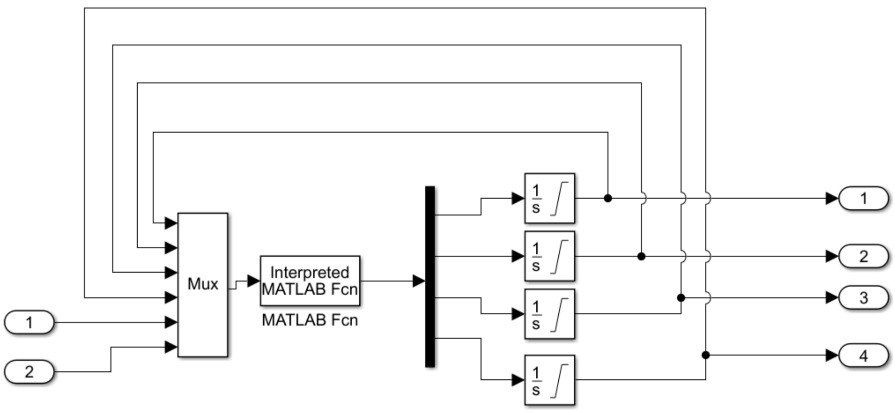


Figure 2.5. Simulink block of the Function 2.11 introduced before.

Notice from Fig. 2.5 that the elements h_i are obtained through the integration of the outputs of the function $\dot{h}_i$. Input ports 1 and 2 indicate respectively q_a and q_b. The integral actions are all initialized at the corresponding level h_i^0, including a saturation such that $h_i \in [0,\ h_{i,max}]$ for $i = 1, 2, 3, 4$.

In this function, once the MATLAB code containing all the values in Table 2.1 has been run, the only two things that need to be specified are the inputs q_a and q_b: the characteristics of these two signals will be discussed in Chapter 3.

2.3 Conclusions

In Chapter 2, the benchmark adopted in the Thesis has been introduced and described: firstly, an overview of the four-tanks system was proposed, then the main mathematical relationships was discussed, highlighting the non-linearity of the model.

Then, its linearisation was introduced, together with a series of properties like the steady-state conditions of the nonlinear system and the position of the zeros as function of γ_1, γ_2. Also the step and frequency responses of the linearised system have been shown.

The real system parameters (expressed in Table 2.1) were presented, noticing how some physical constraints characterise the system, e.g. the maximum levels of the tanks. Also, the implementation of the system in Simulink using a MATLAB function has been discussed.

Chapter 3

Neural network modeling of the four-tanks system and design of experiments

In this chapter, the main elements adopted for the construction of the neural network model of the four-tanks system will be presented and described, highlighting the choices for the number of units and layers of the NN, together with the different types of cells used.

A key part of the training is represented by the cost function adopted, therefore its description will be presented, together with the choice of the optimizer and the discussion of the initialization procedure of the parameters.

Also, the choice of the design of experiments will be discussed (starting from the system function in Figure 2.11 and the model presented in Section 2.1), specifying the main features of the input signals q_a and q_b (such as the presence of noise, the sampling of the signals, etc.), how the normalization is performed and how the batches for training and validation are collected.

3.1 Structure of the neural network

The first thing we have to describe is the structure of the network, focusing on the choice of the cost function for the training, the optimizer adopted, the number of layers (and units) used, the type of basic cell (GRU, LSTM) and so on. Notice that some alternatives are proposed, especially related to the NN topologies. The aspect of the variable initialization has to be treated as well, since it is a critical point as anticipated in Chapter 1.

In the next chapter, all these elements will be fundamental in order to perform the training experiments.

Let us point out that the structure of a NN is not easy to be selected, especially from the point of view of the number of units and layers required for a proper training (there are no precise rules for their exact number computations), therefore the structures proposed in this Thesis are just some possibilities that could be adopted.

3.1.1 Basic elements of the structure

Let us start to discuss about the characteristics that our NN will have in terms of type of cell, number of layers and units. Basically, the approach that will be followed is characterised by the two types of cells introduced in Sections 1.3.7 (GRU cell) and 1.3.6 (LSTM cell), focusing more on the first one. The GRU cells defined in our environment allows the user to add a final linear transformation for the output. Let us recall that in that case the expression is:

$$y(t) = U_o \cdot x(t) + b_o \tag{3.1}$$

Notice that the Eq. (3.1) is used when the GRU cells are composing the output layer of the network, i.e. in the case of multiple layers the output transformation is enabled only in the last one (unless particular requirements on intermediate layers' outputs). It means that, recalling Eq. (1.15) for multi-layer GRU networks, and considering m layers for our NN, the output is provided by:

$$y(t) = U_o \cdot x^{(m)}(t) + b_o \tag{3.2}$$

where $x^{(m)}(t)$ is the state of layer m.

By assumption, in a single NN only one type of cell is considered, so no mix of GRU and LSTM cells is taken into account in the same structures.

Focusing on the number of units per layer, the assumption of keeping it the same value through all the layers of the same network is taken. In particular, for our training experiments this number will vary from a minimum of 5 up to a maximum of 30. Notice that the number of units is strictly related to the training of the network since a too high value may lead to a difficulty in the procedure (let us highlight that every time a unit is added, additional parameters to be tuned are introduced in the network, increasing its complexity), while a too low value may lead to the impossibility to correctly train the network (the available parameters are not enough to mimic the system behaviour).

A similar approach is followed for the number of layers, where just two cases are considered: one layer and two layers. An interesting test that will be performed will compare two different NNs with an equal overall number of units, but divided in different layers (for example, the performances of a NN with 30 units and 1 layer are compared with the ones of a NN with 15 units and 2 layers).

3.1.2 Cost function and optimizer choices

In Section 1.2 the basics of the training of a generic FFNN have been discussed. Since we are dealing with a RNN (due to the presence of GRU and LSTM cells), let us keep in mind what has been said in Section 1.3.2. In particular, let us recall that the training procedure consists in an iterative adjustment of networks' weights so as to minimize a loss function, which still needs to be defined.

For all the trainings, the loss function adopted is a variation of the mean squared error, indicated with the name MSE *washout* (or J^*) because it computes the MSE on two sliced tensors, i.e. y (which represents the value of the real outputs) and $\hat{y}$ (which represents the outputs of the NN), where the first *washout* (also indicated as

T_w) elements are removed from both of them. This is done because the first temporal elements of the outputs might be subject to settling periods, so not relevant for performances evaluation.

However, the size of T_w is kept the same for all the cases and equal to 15, i.e. the first 15 temporal elements of y and $\hat{y}$ are removed before the MSE is evaluated (hence $T_w = 15$).

In Eq. (3.3) the procedure of MSE *washout* is summarized:

$$J^*(y, \hat{y}) = \frac{\sum_{i=T_w}^{n} \left(y_i - \hat{y}_i\right)^2}{n - T_w} \tag{3.3}$$

where n is the number of the elements of the two original vectors, and $\hat{y}_i,\ y_i$ indicates the values of the two quantities at time i.

Also, a penalizing term to J^* expression is added when dealing with GRU networks. It means that the cost function used to perform the minimization during the training is:

$$\begin{aligned} J &= J^* + \mu \\ \mu &= \|U_r\|_2 \cdot 10^{-4} \end{aligned} \tag{3.4}$$

The regularization term μ is introduced in order to reduce as much as possible the oscillations of the MSE value over the epochs. Using this additional term is not mandatory and removing it does not lead to problems in training or worse performances of the NN, however it helps containing the (normal) oscillations of the cost function value. The choice of using the 2-norm of U_r instead of the ones of other weight matrices is related to the fact that U_r is directly involved in the sufficient condition for GRU input-to-state stability (see Eq. (1.17)). Let us notice that a regularization term is added for every layer of the network: if, for example, a network with 3 layers is considered, the resulting cost function is given by:

$$\begin{aligned} J &= J^* + \mu^{(1)} + \mu^{(2)} + \mu^{(3)} \\ \mu^{(i)} &= \|U_r^{(i)}\|_2 \cdot 10^{-4}\ , \quad i = 1,\ 2,\ 3 \end{aligned}$$

where $\mu^{(i)}$ indicates the regularization term for layer i.

Focusing on the choice of the optimizer, the discussion of possible alternatives to gradient descent algorithm when deep NNs are faced was treated in Section 1.3.4. In particular, for our network structure the RMSProp optimizer presented in Section 1.3.4 has been chosen (recall Eq. (1.9)), and the corresponding values of the parameters are $\beta = 0.9$, $momentum = 0.15$, $\eta = 0.002$ and $\epsilon = 10^{-7}$. These parameters are fixed for all the training operations that will be performed in Chapter 4.

3.1.3 Initialization of the variables

The last thing that has to be discussed is how the variables of the network (weights and biases) are initialized. This represents an important point since initializing the quantities in a too wide range may lead to vanishing and/or exploding gradient problems (Section 1.3.5). In fact, since we are dealing with a training algorithm that is iterative, the starting point (imposed by the initialization) affects the convergence or divergence of the algorithm itself (as discussed by Goodfellow et al. in [9]).

Also, let us notice that initializing the weights with too high values may lead to activation function saturation, corresponding to a loss of gradients through such saturated units.

In our case, the *variance scaling* initializer is considered: it draws the samples from a truncated normal distribution with zero mean and standard deviation $\sigma = \sqrt{\frac{scale}{n}}$, where *scale* is a positive parameter tuned by the user that indicates the scaling factor (in our case equal to 0.5) and n is the number of input elements in the weight tensor. It can be noticed that such initializer is able to adapt the standard deviation of the normal distribution to the size of the weight tensor it has to initialize; about *scale* parameter, its only constraint is to ensure that it is a positive quantity, and as noticeable from the expression of σ, it acts as a scaling factor on the standard deviation increasing or decreasing it (for example we could take *scale* equal to 1 or 2).

Let us recall that other types of initialization procedures (e.g. Xavier and He initializations) were proposed in Section 1.3.5.

3.2 Design of experiments

In this section, the design of experiments is faced, more specifically the collection of the inputs: it is a fundamental part to ensure that the training of the NN is well-performed, since a poor (i.e. non informative) dataset structure may lead to a (partially or totally) wrong learning of the network. In particular, the discussion will be mainly based on the features of q_a and q_b, starting from a bunch of hypothesis and assumptions. Afterwards, the outputs h_1, h_2, h_3, h_4 will be collected and the obtained dataset will be normalized.

Starting from the normalized dataset, the batches (single sets of data) will be defined to allow the effective training and validation phases.

3.2.1 Preliminary assumptions and starting point

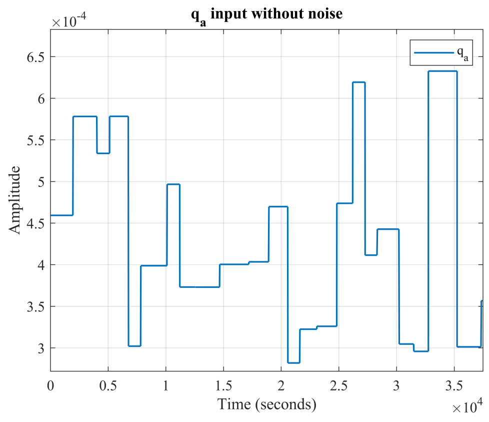
A key part of the training of a NN is the design of experiment and pre-processing of the measured data. It has been discussed in the previous chapter the model of our four-tanks system through which, with a proper set of inputs, it is possible to obtain the desired outputs (i.e. the levels of the four tanks). Starting from this idea, the first thing to be explained is the generation of the inputs q_a and q_b: the approach is to mimic the behaviour of an experiment performed in the real world, where the input flows are continuously changed and characterized by disturbances (noise).

Therefore, the definition of q_a and q_b consists in the generation of two signals viewed as the composition of multiple steps in series, such that each of them has a random time length, a random amplitude and a random starting time instant. However, the signals cannot be generated in a completely random fashion as some constraints need to be accounted:

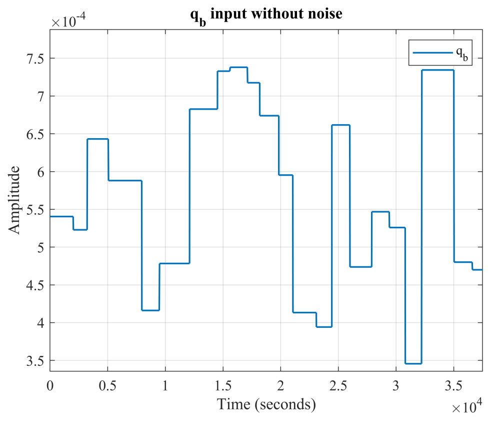
- all the steps have a minimum length of 1000 seconds: this ensures that their effects are significant on the levels of the tanks, but these ones do not reach the stationary condition every time;

- the length of the overall experiment has to be fixed: in our case a time length of 37500 seconds is considered, in order to ensure a number of data equal to 1500 per experiment (for each quantity), as it will be discussed later considering the sampling time. Clearly, different values for the length of the experiment could be selected, in order to increase or decrease the number of samples;
- the starting time instants of the steps are randomly selected in the range [1, 37500], still guaranteeing the minimum length of each one;
- the first step is set at the first time instant to avoid a zero start of the signals that may lead to a drop of the tanks' levels (if the first step is too far from the beginning);
- the amplitudes of the steps are limited at the $\pm$ 40% of their nominal values q_a^0, q_b^0. This will guarantee that the signals will vary in a sufficiently large interval avoiding as much as possible that the tanks' levels reach their saturation limits (this specific case will be treated separately in Chapter 4).

Considering these assumptions, an example of the signals q_a and q_b is shown in Figures 3.1a and 3.1b respectively.



(a)



(b)

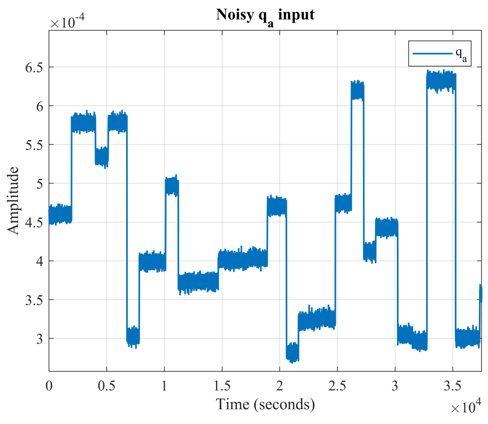
Figure 3.1. (a) q_a input before sampling without noise; (b) q_b input before sampling without noise.

3.2.2 Noise, signals sampling and thresholds check

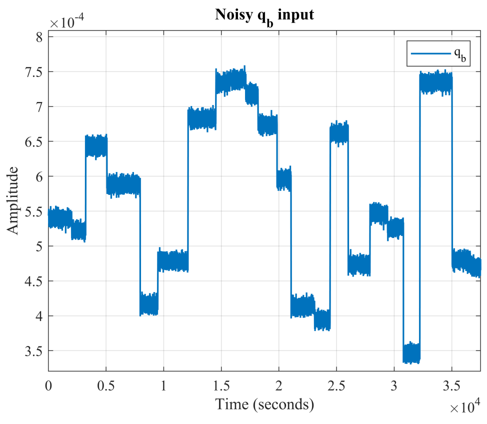
In real world, it is almost impossible to have a perfect input signal (especially a flow rate like in our case) without any disturbance acting on the system: here the idea is to try to add a replicated *White Noise* (WN) to the inputs to mimic the disturbances present in the real world. At this purpose, let us try to simplify a bit the problem, still guaranteeing the randomness of the noise: the approach consists in placing the noise all over the time length of each step, for its whole duration. It is expected that our NN reacts to such disturbances, still guaranteeing a good tracking of the reference signals.

About the amplitude of the replicated WN, the choice has to be properly selected since a too wide range for such value may lead to a distortion of the inputs, that is not what we are interested in. Therefore, a good way to proceed is to take, for every time instant of the signals previously defined, a random value from the standard normal distribution (notice we may have also negative values) and multiply it by the 1% of q_a^0 or q_b^0, on the basis of the input considered. The resulting quantity is then summed to the current value of the input at the same instant of time.

By adding the noise to the two signals indicated in Fig. 3.1a and Fig. 3.1b, they will change as in Figures 3.2a and 3.2b. As it can be noticed, the signals are not distorted by the WN if compared to the original ones in Figures 3.1a, 3.1b.



(a)



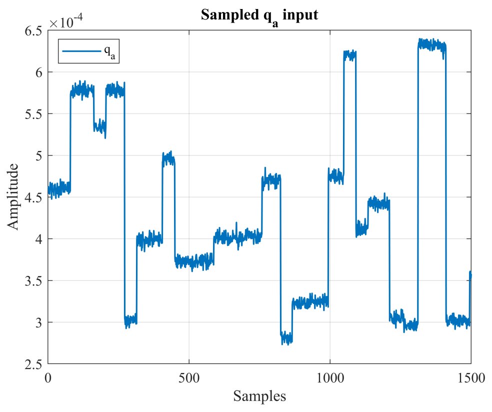
(b)

Figure 3.2. (a) q_a input signal with WN before sampling; (b) q_b input signal with WN before sampling.

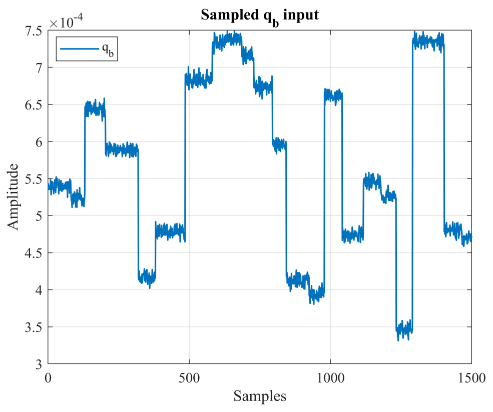
Up to now, the generated inputs are noisy signals with a time length of 37500 seconds, but it is necessary to sample them using a specific value for the sampling time T_c. In our case, and from now on, T_c will be always equal to 25 seconds, however different possibilities could be adopted, e.g. 30 s, 20 s and so on. It can be observed that, retrieving the step responses of the linearised system in Figure 2.3, the faster settling time observed is around 4000 s (for the step response of h_1 to q_a step), therefore the associated time constant is $\tau = \frac{4000}{5} = 800\ s$, which corresponds to $\omega = \frac{1}{800} = 0.00125\ rad$. On the other hand, taking into account the sampling time $T_c = 25\ s$, the associated frequency is $\omega_n = \frac{\pi}{Tc} = 0.1256\ rad$, so an attenuation of almost 40 dB (i.e. two decades) is observed.

Therefore, by sampling both inputs q_a and q_b at every 25 time instants, the two resulting signals are characterized by $\frac{37500}{25} = 1500$ samples, and, correspondingly, all the outputs of the system (i.e. h_1, h_2, h_3 and h_4) have the same size. Recalling the physical constraints of the system and the specific values introduced in Table 2.1, it is mandatory to guarantee that $q_a \in [q_{min},\ q_{a,max}]$ and $q_b \in [q_{min},\ q_{b,max}]$, that is why once the two signals are generated a final check must be performed in order to ensure the boundaries fulfilment.

After the sampling is performed, the resulting input signals, from Fig. 3.2a and Fig. 3.2b, are indicated in Figures 3.3a and 3.3b respectively.



(a)



(b)

Figure 3.3. (a) q_a input signal with WN after sampling; (b) q_b input signal with WN after sampling.

3.2.3 Output data collection

Once q_a and q_b have been defined, we need to use them as inputs to the system in order to generate h_1, h_2, h_3 and h_4. In order to do that, the Simulink block in Figure 3.4 is taken into account.

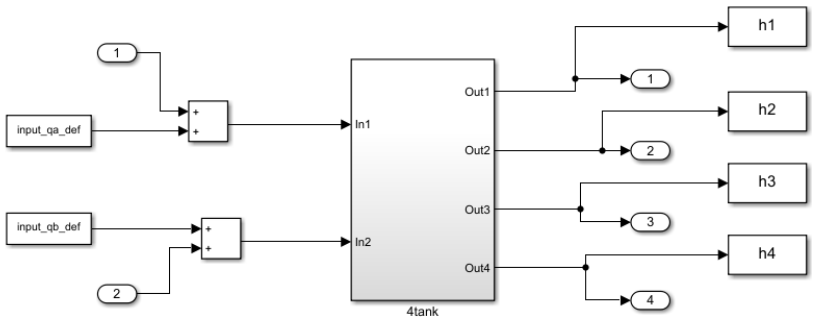


Figure 3.4. Simulink block used for the outputs generation.

From the scheme it can be noticed that the function defined in Section 2.2 is run, fed by q_a and q_b. Also, in a similar way to what has been done for the inputs, even the outputs need to fulfil physical constraints: in fact h_i needs to be greater or equal to zero for $i = 1,\ 2,\ 3,\ 4$ (negative values for tank level have no meaning); however, it has to be guaranteed each tank does not exceed its corresponding maximum value. These two conditions can be wrapped up as:

$$h_i \in [0,\ h_{i,max}]\ \ \forall\ i \in [1,\ 4]$$

At this point, all the inputs and outputs of the system are available, all with the same number of samples (1500 as anticipated) and the overall experiment size has to be discussed: to train well the NN, a quite huge variety of inputs and outputs needs to be collected, so just one experiment with 1500 samples per variable is not enough. The idea, for the moment, is to execute a set of 20 experiments, from which the training and validation datasets are extracted, as it will be shown later on. It means we will deal with 20 vectors for q_a and for q_b, that are associated correspondingly to 20 output vectors h_1, h_2, h_3, h_4 with 1500 samples each. In Figures 3.5a, 3.5b, 3.5c and 3.5d the levels of the tanks are shown, taking as inputs the two signals in Fig. 3.3a and 3.3b.

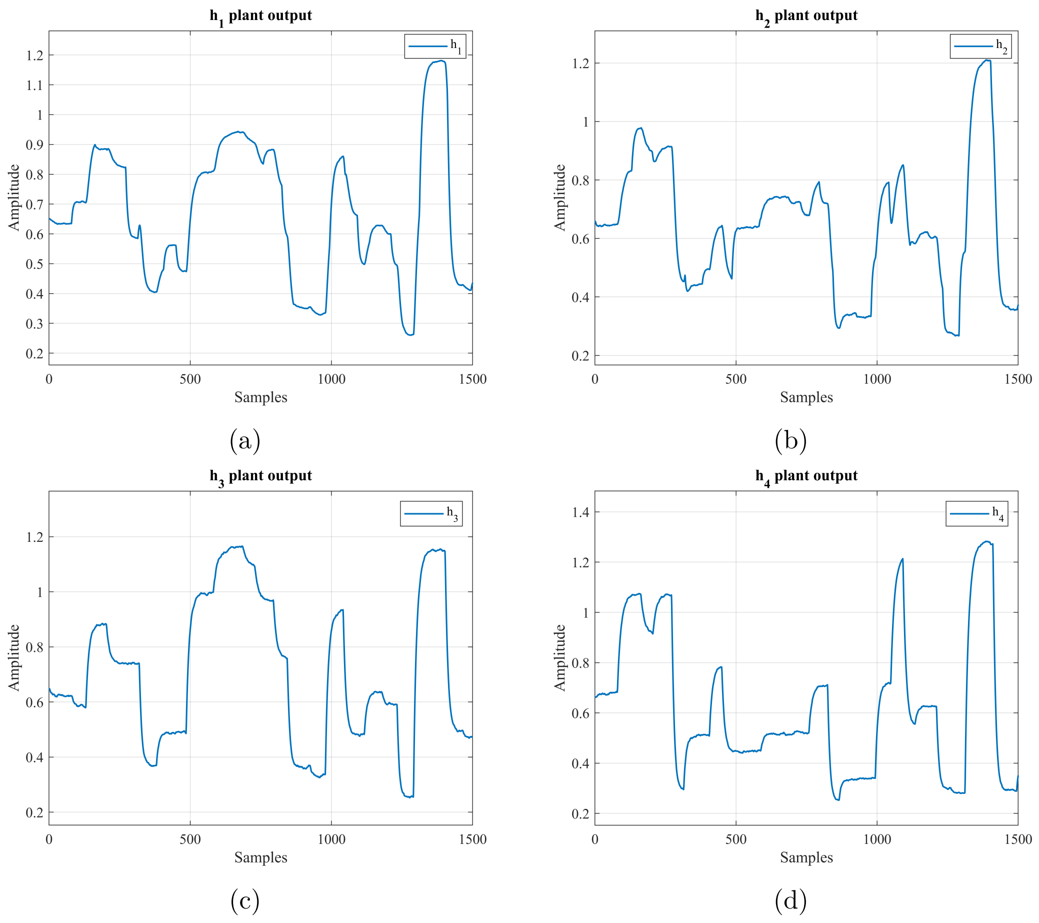


Figure 3.5. (a) h_1 signal; (b) h_2 signal; (c) h_3 signal; (d) h_4 signal.

It is easy to notice that the presence of the WN in q_a and q_b perturbs the tanks levels, in fact in the previous four figures it can be observed very small fast variations of the levels (e.g. for Fig. 3.5c around the sample 500).

3.2.4 Data normalization

The data obtained up to now is not ready to be used yet. The reason is that the signals are not normalized: in fact, neural networks work better with normalized datasets. The approach that will be followed is a sort of standardization (or *Z-score normalization*), i.e. indicating with Z the set of observations considered, with Z_{mean} its mean values and with Z_{max} the maximum value of $|Z - Z_{mean}|$, the normalized observation Z_{norm} will be:

$$Z_{norm} = \frac{Z - Z_{mean}}{Z_{max}} \tag{3.5}$$

Notice that the main difference between our normalization of Eq. (3.5) and common standardization scaling is that the latter provides a new set of observations Z_{norm}

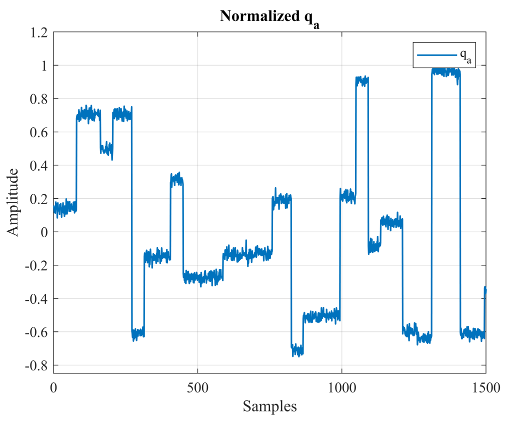
such that $Z_{norm} \in [0,\ 1]$ (using as denominator the standard deviation σ of Z), while in our case we get $Z_{norm} \in [-1,\ 1]$.

It is important to point out that, since we are dealing with a set of 20 experiments, the normalization of the dataset (in terms of both inputs and outputs) cannot be performed one experiment at a time. In other words, if we normalize all the data associated to a single experiment independently from all the others, the resulting dataset will not be useful for the training of the network. The procedure can be summarized as it follows:

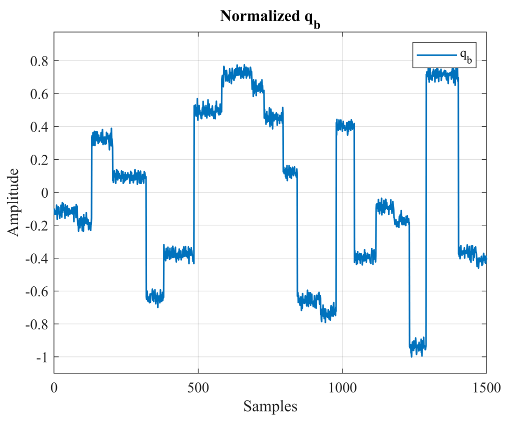
1. select a variable in the set $\{q_a,\ q_b,\ h_1,\ h_2,\ h_3,\ h_4\}$ and let us call it Φ;
2. put all the observations over the 20 experiments of Φ into a single vector, called ψ (which will have a size of $20 \cdot 1500 = 30000$ samples);
3. compute the mean value of ψ (indicated as ψ_{mean}) and subtract it from the vector ψ itself, obtaining $\tilde{\psi} = \psi - \psi_{mean}$;
4. compute the maximum of the absolute value of $\tilde{\psi}$, i.e. $\psi_{max} = \max|\tilde{\psi}|$;
5. normalize the values of Φ as it follows (element-wise):
$$\Phi^i_{norm} = \frac{\Phi^i - \psi_{mean}}{\psi_{max}} \quad \forall\, i = 1, ..., \ 20$$
where the superscript ^{i} indicates the value of the quantity of the i-th experiment. Therefore, this procedure is done for all the values of Φ over all the experiments performed;
6. repeat the procedure from point 1 by selecting a different variable until all of them are normalized.

Keep in mind that all the variables $\{q_a,\ q_b,\ h_1,\ h_2,\ h_3,\ h_4\}$ have a size of $(1500,\ 20)$, so the batches for the training and validation are not formed yet.

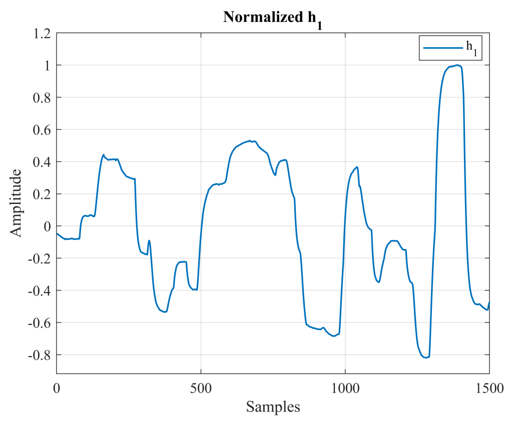
Taking as references for simplicity only the signals of Figures 3.3a, 3.3b, 3.5a, 3.5b, 3.5c and 3.5d, after the normalization, the new signals will be in the range [-1, 1]. From a graphical point of view, the corresponding results are shown in Figures 3.6a, 3.6b, 3.6c, 3.6d, 3.6e, 3.6f.



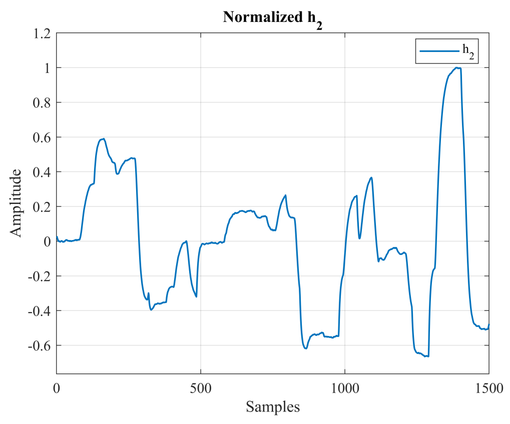
(a)



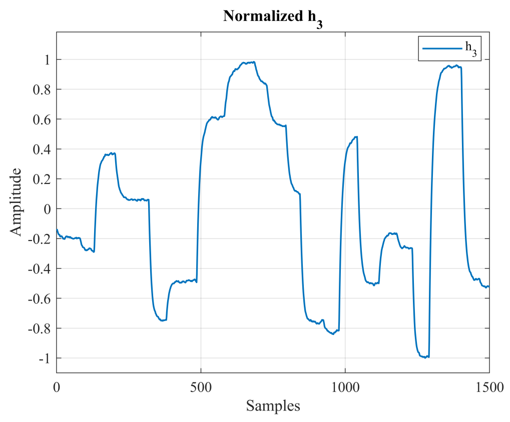
(b)



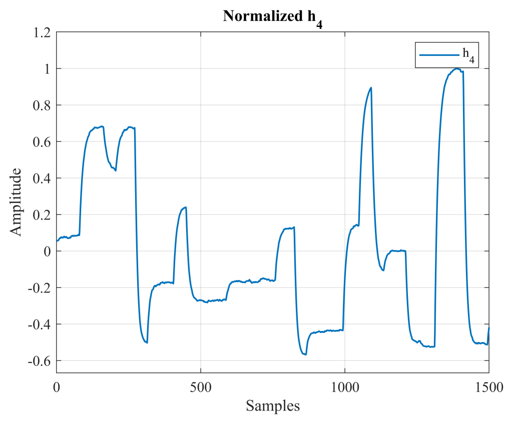
(c)



(d)



(e)



(f)

Figure 3.6. (a) q_a signal; (b) q_b signal; (c) h_1 signal; (d) h_2 signal; (e) h_3 signal; (f) h_4 signal.

3.2.5 Batches creation for training and validation

At this point, all the inputs and outputs of the system are known and available, however the training and validation datasets have to be defined. The first one is needed to perform the training of the NN, while the second one for the performance evaluation.

As previously said, we will consider 20 experiments (each one characterised by two q_a, q_b signals and the corresponding four tanks' levels h_i), where a single experiment corresponds to a single batch of the dataset: more specifically, the choice will be to take the first 19 batches for training dataset and the 20th one for validation dataset. Indeed, since the validation dataset is used in order to check the performances of our NN, it has to consider a completely new set of data never seen during the training. By observing if the associated validation MSE (indicated as J^* as in Eq. (3.3), and computed as the washout mean squared error between the predictions of our RNN and the real values of the outputs of validation dataset) is low or high, the performances of the network are verified. It is clear that, if we enclose even the 20th batch in the training set, the validation MSE at the end of each epoch would be meaningless since we may get overfitting on such data. It is important to notice that the validation dataset is such that the corresponding inputs q_a, q_b are noise-free: this guarantees that the value of the validation MSE is not influenced in any way by the presence of the WN acting on the signals, but it is strictly related to the network performances only.

Now the datasets are ready to be used, noticing that for each epoch all the nineteen batches will be used, passing them once at a time; let us highlight that the order of batches through which they are passed to the NN is re-shuffled at every epoch to avoid overfitting.

3.3 Conclusions

In Chapter 3, the main elements of the NN structure have been introduced, treating the design of experiments too. In particular, we started from the basic elements like the number of units and layers. Then, the choice of the cost function for the training has been discussed, focusing on the importance of the optimizer to speed up the procedure and on the initialization of the parameters.

In the second part of the chapter the main characteristics of the inputs $q_a,\ q_b$ were proposed, discussing the presence of noise, the sampling of the signals and a set of assumptions. Then, the corresponding recording of the outputs was introduced, together with the normalization of the variables within the interval $[-1, 1]$, ending with the batches creation procedure for training and validation operations.

Chapter 4

Training of the neural network model

In this chapter the training of the network is performed: different NN structures will be considered and the results will be presented and compared in terms of how much the predictions provided by the model are accurate and of the computational cost required (average time to complete the training).

In order to validate the obtained results, the tests for each set-up will be executed three times each, due to the (small) variability that characterises the training of a NN. This also guarantees that, even if all the tests are executed on the same PC under the same conditions, the time required for the training operations (extremely related to the computer hardware, state of the battery, etc.) is more reliable; let us notice that the initialization of the parameters is different for every test.

More specifically, this chapter will be characterised by a first section where the common BPTT approach is adopted for system with $h_1,\ h_3$ as outputs, a second section where the dataset (described in Section 3.2) is subject to some changes in order to study the saturation of the tanks' levels, a third section where the TBPTT approach is implemented and a fourth one for the training of the system with h_1 and h_2 as outputs.

In all these cases, the assumptions about the NN structure and the design of experiments discussed in Chapter 3 will be valid unless explicitly specified.

Two different types of system will be taken into account:

- MIMO system considering both the input pumps and only two of the four levels h_i (MIMO 2x2), dealing with three alternatives:
  - only the left-hand side tanks h_1 and h_3 are taken into account, together with q_a and q_b (Figure 4.1a);
  - only the right-hand side tanks h_2 and h_4 are taken into account, together with q_a and q_b (Figure 4.1b);
  - only the lower tanks $h_1,\ h_2$ are taken into account, together with q_a and q_b (Figure 4.1c). Notice this represents the system that is controlled in the real world;
- MIMO 2x4 system considering all the input pumps and all the tanks' levels h_i, for $i = 1,\ 2,\ 3,\ 4$ (Figure 4.1d).

Let us notice that the left-hand side and right-hand side systems are identical from the point of view of the training operation, so only one the left-hand side case will be analysed.

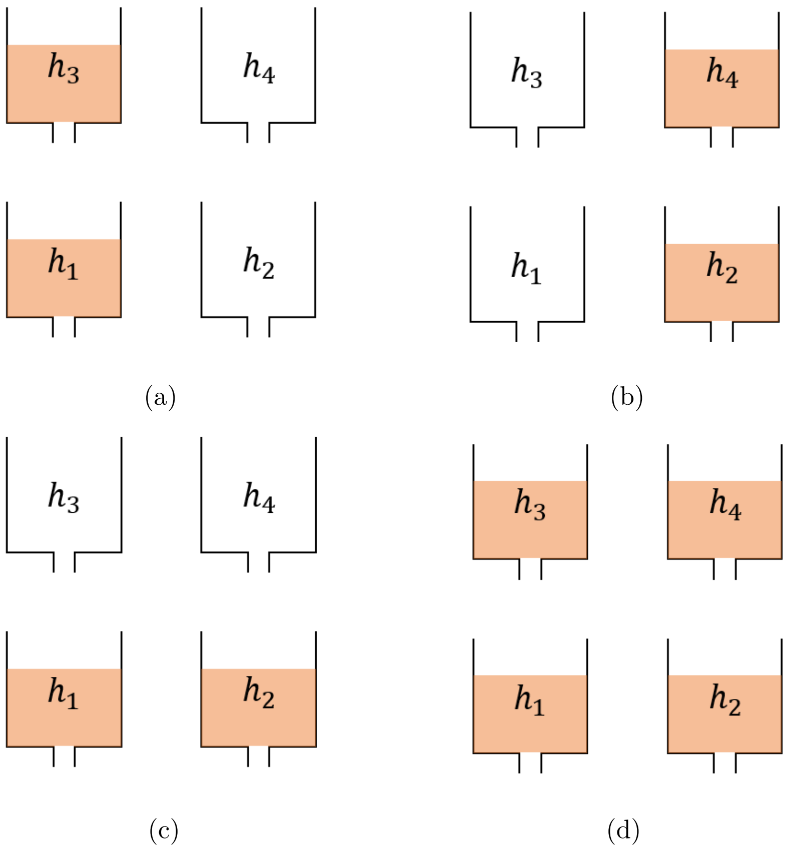


Figure 4.1. (a) Scheme of h_1, h_3 system; (b) Scheme of h_2, h_4 system; (c) Scheme of h_1, h_2 system; (d) Scheme of h_1, h_2, h_3, h_4 system.

4.1 Training experiments with BPTT approach

For the first training operations, the common BPTT approach will be used (Section 1.3.2). Let us specify that all of them are performed over 300 epochs (a sufficiently high number to guarantee a good training). As it will be discussed in the chapter, different trials will be made, every time changing a bit the network structure.

The following NN topologies will be considered:

- GRU with 5 units and 1 layer (for both two-tanks and four-tanks systems);
- GRU with 30 units and 1 layer (for both two-tanks and four-tanks systems);
- GRU with 15 units per layer, and 2 layers (for two-tanks system);
- LSTM with 5 units and 1 layer (for two-tanks system);
- LSTM with 30 units and 1 layer (for two-tanks system).

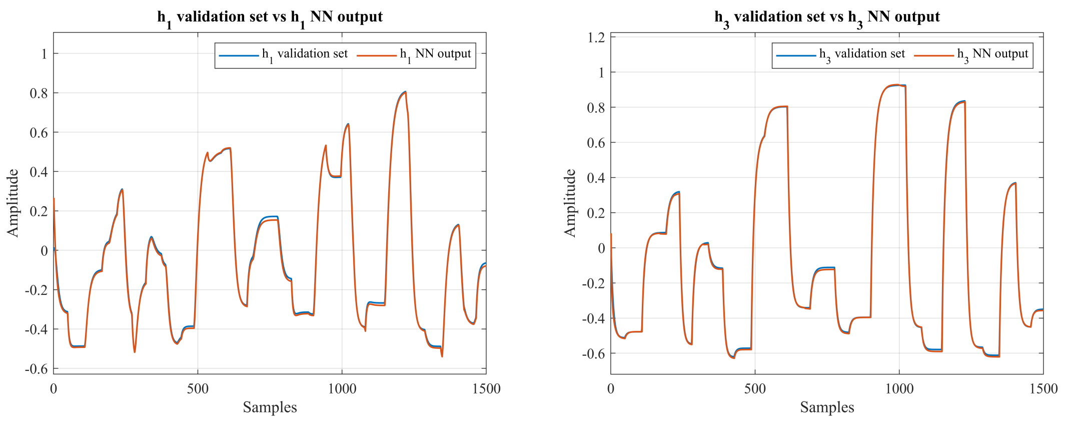
Let us recall that, as explained in Chapter 3, the training dataset is composed by 20 batches (i.e. experiments) with 1500 samples each (for every considered quantity), where the first 19 batches are used for the training and the last one for the validation.

4.1.1 GRU with 1 layer and 5 units

The first training performed takes into account a NN with GRU cells, 5 units and a single layer. The system treated is the two-tanks system described in Fig. 4.1a (i.e. only h_1 and h_3 are studied). As previously said, the test is repeated three times, and the results are given by the mean values of the three trials; more specifically, the Eq. (4.1) is used as reference for the computations:

$$\begin{aligned} J^*_{avg} &= \frac{J^*_1 + J^*_2 + J^*_3}{3} = 4.2 \cdot 10^{-5} \\ t_{avg} &= \frac{t_1 + t_2 + t_3}{3} \simeq 25\ min \end{aligned} \tag{4.1}$$

where J^*_i indicates the value of the loss function without any regularization term of the test i (see Eq. (3.3)), and t_i the time occurred to complete the training operation of test i. From now on, only the two quantities J^*_{avg} and t_{avg} will be taken into account.



(a) (b)

Figure 4.2. (a) h_1 predicted by the NN (5 units, 1 layer) versus validation signal; (b) h_3 predicted by the NN (5 units, 1 layer) versus validation signal.

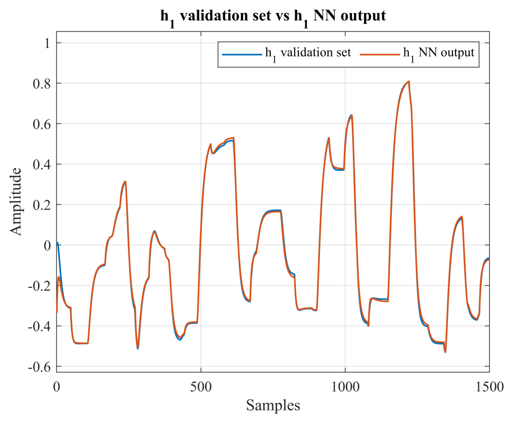
Notice from Figures 4.2a, 4.2b the predicted outputs of the NN are very close to the real ones (as expected from J^*_{avg} value), so the network approximates the real plant very well.

Now let us try to analyse if the NN structure described before is sufficient and able to train the four-tanks system (Figure 4.1d). Hence, we still consider a GRU network with 5 units and only one layer, obtaining the following results:

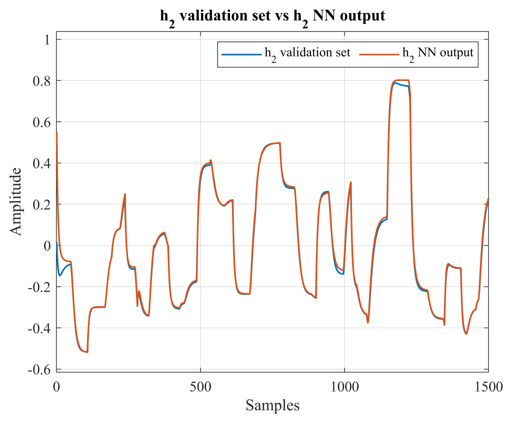
$$\begin{aligned} J^*_{avg} &= 9.7 \cdot 10^{-5} \\ t_{avg} &\simeq 35\ min \end{aligned}$$

Clearly, the value of the error increases due to the presence of four quantities to be estimated instead of two as before, however such value is still very low and satisfactory.

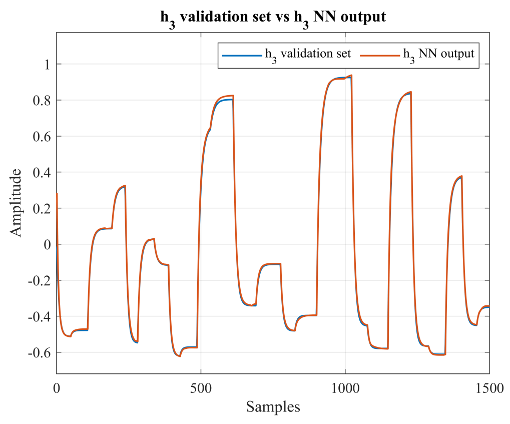
For the same reason, even t_{avg} increases too. Analysing Figures 4.3a, 4.3b, 4.3c, 4.3d all the four levels h_i are well predicted, therefore the structure with only 5 units still guarantees a good training.



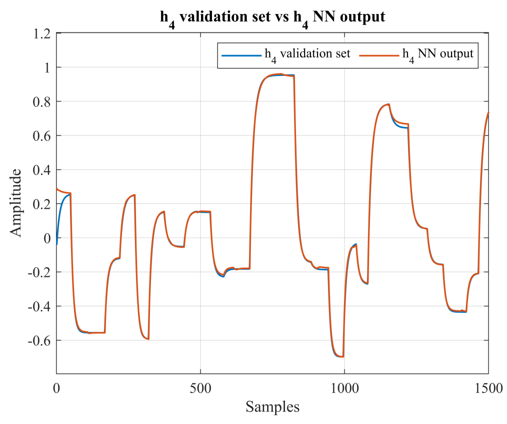
(a)



(b)



(c)



(d)

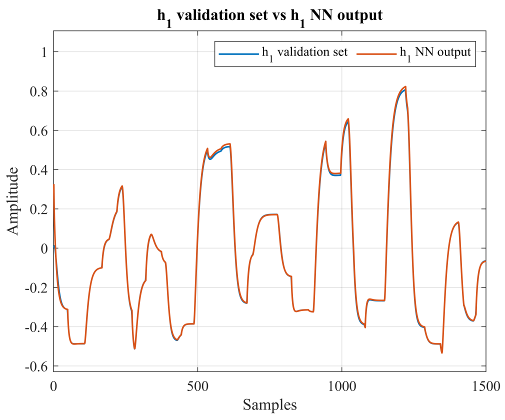
Figure 4.3. (a) h_1 predicted by the NN (5 units, 1 layer) versus validation signal; (b) h_2 predicted by the NN (5 units, 1 layer) versus validation signal; (c) h_3 predicted by the NN (5 units, 1 layer) versus validation signal; (d) h_4 predicted by the NN (5 units, 1 layer) versus validation signal.

4.1.2 GRU with 1 layer and 30 units

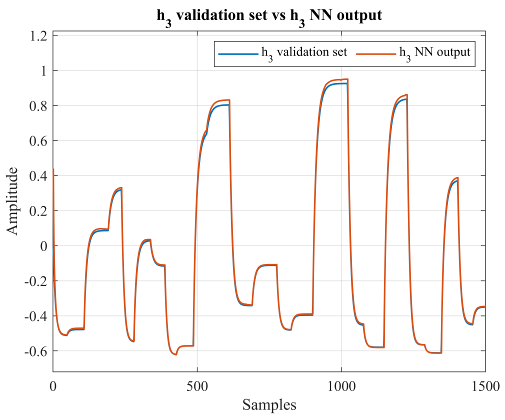
Now the number of units of the single layer is increased to 30, checking if such an increase leads to a better training (lower value of J^*_{avg}), keeping the required time as low as possible; let us recall that adding units means dealing with a higher number of parameters to be tuned by the network, however it does not necessarily imply neither a slower training nor a worse/better performance.

Considering Equation (4.1) and the two-tanks system with only h_1, h_3 as levels, the quantities obtained are:

$$J^*_{avg} = 2.9 \cdot 10^{-5}$$
$$t_{avg} \simeq 30\ min$$



(a)



(b)

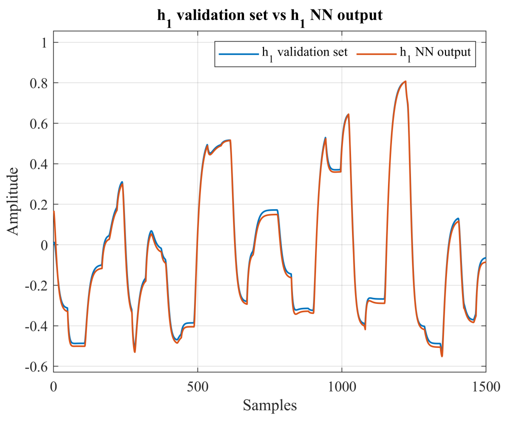
Figure 4.4. (a) h_1 predicted by the NN (30 units, 1 layer) versus validation signal; (b) h_3 predicted by the NN (30 units, 1 layer) versus validation signal.

By a quick comparison between Figures 4.2 and 4.4 (and the associated values too), the results in terms of accuracy of predictions are both satisfactory, with a sightly better performance of the configuration with 30 units: probably this is due to the higher number of parameters that can be tuned by the network. However, the best configuration can be identified in the one with 5 units since, even if it has a slightly higher J^*_{avg}, it has a lower number of units (and consequently of network parameters) so the risk of overfitting is reduced. About t_{avg}, the increase of units leads to a (small) increase of the average training time, but not so relevant to be considered a problem.

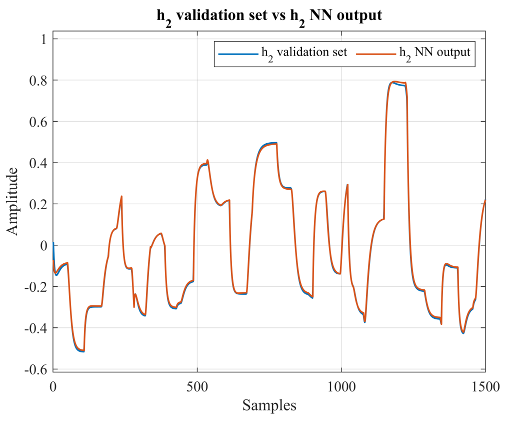
Now, as discussed for the previous case with only 5 units, we apply the same NN structure to the four-tanks system, expecting a lower value of J^*_{avg} as for the two-tanks system than the one achieved with the previous topology. Therefore we obtain:

$$J^*_{avg} = 4.8 \cdot 10^{-5}$$
$$t_{avg} \simeq 40 \; min$$

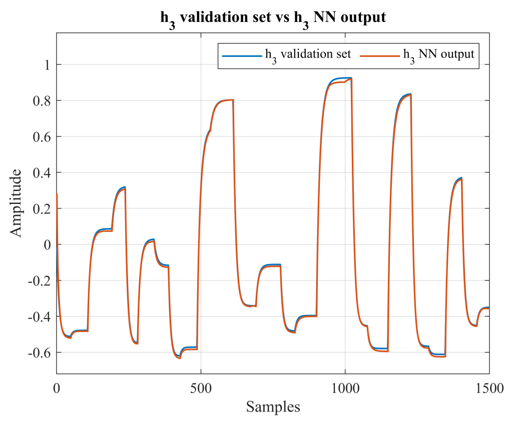
As expected, the average validation MSE decreases of almost one half if compared to the NN with only five units, guaranteeing a better accuracy of the predictions $\hat{y}$. Considering the mean training time, the increasing is limited as for the two-tanks system. In Figures 4.5a, 4.5b, 4.5c and 4.5d the corresponding results are shown.



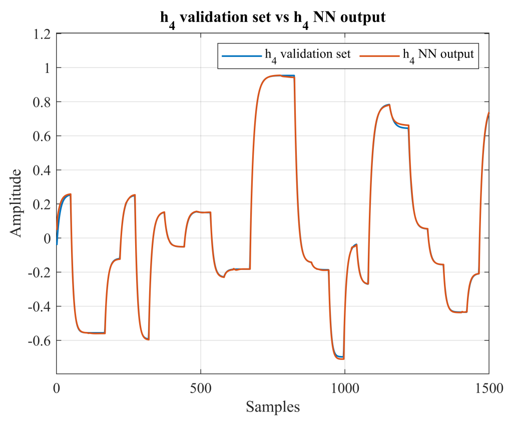
(a)



(b)



(c)



(d)

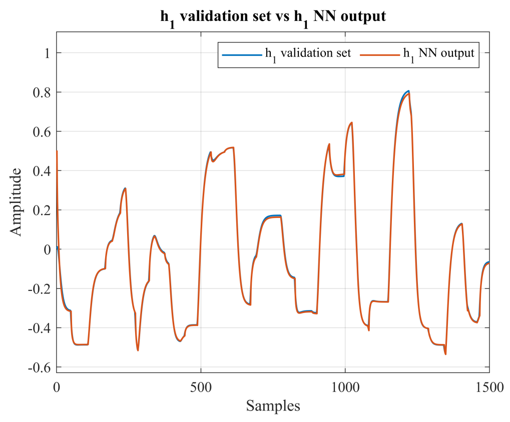
Figure 4.5. (a) h_1 predicted by the NN (30 units, 1 layer) versus validation signal; (b) h_2 predicted by the NN (30 units, 1 layer) versus validation signal; (c) h_3 predicted by the NN (30 units, 1 layer) versus validation signal; (d) h_4 predicted by the NN (30 units, 1 layer) versus validation signal.

4.1.3 GRU with 2 layers and 15 units

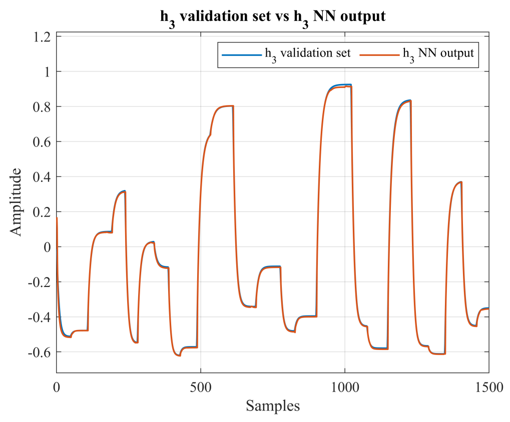
Let us check if keeping the same number of units, but dividing them over more than one layer leads to a better performance of the training. At this purpose, we consider a GRU network with 2 layers and 15 units per layer: this means we will have an equivalent number of 30 units in the NN, so a comparison with previous experiment of Section 4.1.2 can be done. Also, the two-tanks system in Figure 4.1a is used.

After the three tests, the obtained results are:

$$J^*_{avg} = 1.5 \cdot 10^{-4}$$
$$t_{avg} \simeq 50\ min$$



(a)



(b)

Figure 4.6. (a) h_1 predicted by the NN (2 layers, 15 units each) versus validation signal; (b) h_3 predicted by the NN (2 layers, 15 units each) versus validation signal.

As expected, considering the required time, this configuration leads to a worse performance, since training a network with more layers requires a higher computational cost than in the case of a NN with an unique layer. In addition, even if from Figures 4.6a, 4.6b it is difficult to notice the difference with the configuration with 1 layer and 30 units, the accuracy of predictions is lower: checking the J^*_{avg} of both cases, there is almost an order of magnitude of difference. Therefore, considering the two-tanks system with h_1, h_3, the NN configuration with 30 units and a single layer ensures better performances than splitting the units over two layers as done before.

4.1.4 LSTM with 1 layer and 5 units

Considering the previous three training operations with GRU cells, the best network structure obtained is the one with single layer and 5 units. Starting from this, let us check if, under the same conditions, the LSTM cells work better than the GRU ones. At this purpose, we consider a LSTM network with one layer and 5 cells, obtaining:

$$J^*_{avg} = 9.2 \cdot 10^{-5}$$
$$t_{avg} \simeq 30\ min$$

Analysing the differences between this training and the one of Section 4.1.2 with GRU cells and same architecture, for the considered system and for the collected data the proposed GRU cells are associated to a lower J^*_{avg} than LSTMs. It means that GRUs work better, guaranteeing a higher accuracy. We conjecture that this is due to the higher complexity of LSTMs: as explained in Section 1.10, GRU cells can be seen as an exemplification of LSTMs, therefore in our specific environment they mimic better the considered system and are easier to be trained.

In Figures 4.7a, 4.7b the results with LSTM cells are shown: they are still accurate, however the differences among predictions and real values can be noticed in some points, especially at the peaks of h_1 and h_3.

Let us highlight that, even if in this specific case GRU cells work better than LSTMs, this is not a general rule for every plant: for other types of benchmarks, it is likely that the latters have better performances.

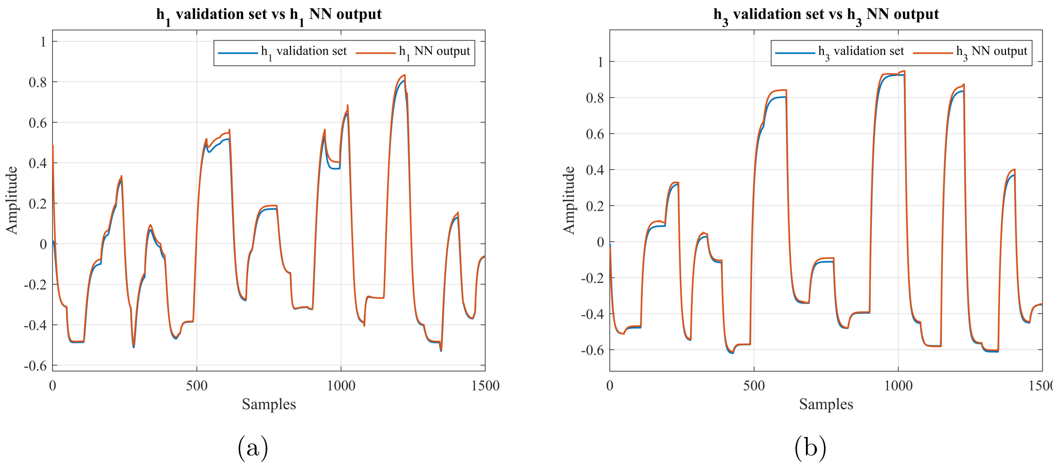


Figure 4.7. (a) h_1 predicted by the NN (5 LSTM units, 1 layer) versus validation signal; (b) h_3 predicted by the NN (5 LSTM units, 1 layer) versus validation signal.

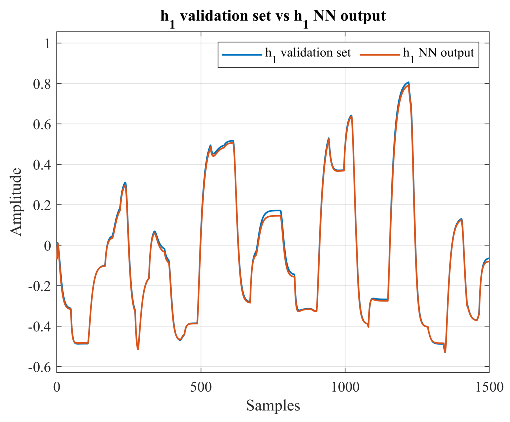
4.1.5 LSTM with 1 layer and 30 units

Observing what has been achieved with 5 units in Section 4.1.4, let us try to increase the number of units up to 30 checking the corresponding results, expecting a higher precision.

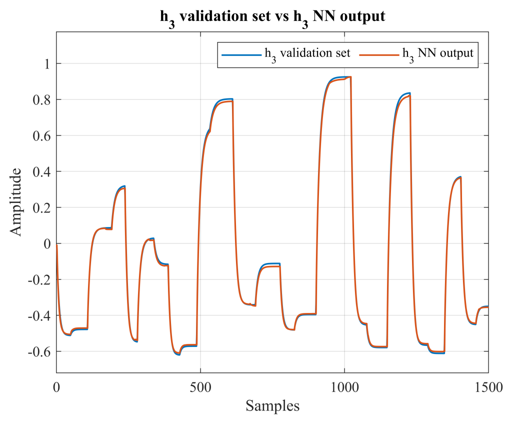
Considering Eq. (4.1), the obtained values are:

$$J^*_{avg} = 4.8 \cdot 10^{-5}$$
$$t_{avg} \simeq 40\ min$$

As already observed for GRU cells, even in this case with LSTMs the increase of the number of units corresponds to a lower J^*_{avg}, i.e. a higher accuracy of the predictions. Indeed, comparing Figures 4.7a, 4.7b and Figures 4.8a, 4.8b it can be observed how the mismatches between predicted and real values are considerably reduced. About the time required for training operations, they are quite similar showing that the increase of units does not affect t_{avg} in a relevant way as for GRUs.



(a)



(b)

Figure 4.8. (a) h_1 predicted by the NN (30 LSTM units, 1 layer) versus validation signal; (b) h_3 predicted by the NN (30 LSTM units, 1 layer) versus validation signal.

In Table 4.1, the summary of all the trainings performed up to now is proposed.

Considered tanks	Type of cell	n. of layers	n. of units	J^*_{avg}	t_{avg}
$h_1,\ h_3$	GRU	1	5	4.2e-05	25 min
$h_1,\ h_2,\ h_3,\ h_4$	GRU	1	5	9.7e-05	35 min
$h_1,\ h_3$	GRU	1	30	2.9e-05	30 min
$h_1,\ h_2,\ h_3,\ h_4$	GRU	1	30	4.8e-05	40 min
$h_1,\ h_3$	GRU	2	15	1.5e-04	50 min
$h_1,\ h_3$	LSTM	1	5	9.2e-05	30 min
$h_1,\ h_3$	LSTM	1	30	4.8e-05	40 min

Table 4.1. Table that summarizes the training experiments performed in Section 4.1, considering the treated tanks, the value of J^*_{avg}, the approximated t_{avg} of the training and the network architecture. Notice that J^*_{avg} and t_{avg} are obtained respectively as the mean value of the J^* of the 3 tests and as the mean value of the time length of the 3 tests.

4.2 Training experiments with saturated dataset

In the previous training sessions we considered a pretty standard dataset that allowed us to check if the network performs well or not. Since the obtained results are good, some additional trials can be done, introducing interesting variations to the input dataset described in Section 3.2. More specifically, the amplitudes of the q_a, q_b signals are increased up to $\pm 90\%$ of their nominal values $q^0_{a,b}$ in order to highlight the effect of the tanks' levels saturations (instead of the $\pm 40\%$ variations considered up to now). This test allows us to check if the NN is able to mimic the behaviour of the tank when it has to match the saturation. Let us notice that we still ensure $0 \leq q_a \leq q_{a,max}$, $0 \leq q_b \leq q_{b,max}$.

In Figure 4.9, the behaviour of h_3 tank is shown as an example, highlighting how the tank's level reaches the corresponding maximum value $h_{3,max}$ in different points.

Notice that, even if this variation of the dataset is implemented, the BPTT principle will be still followed. Also, this change will be tested on the two-tanks system considering h_1, h_3 due to a better visibility of the results (only two outputs), with a network structure characterised by GRU cells, 5 units and a single layer (the procedure could be easily tested on four-tanks MIMO system and/or with different architectures too). All the assumptions made for NN definition (such as the optimizer, loss function etc.) presented in Section 3.1 are still valid.

At this point, let us proceed with the training, doing it for the same number of epochs as in previous cases, i.e. 300. The results in Figures 4.10a, 4.10b show the low capability of the NN to follow the imposed constraints: it is clear how the predictions fail to match the saturation levels of the tank (especially for h_3), while for all the other parts the match is acceptable (but not particularly good). The corresponding value of the average J^* is:

$$J^*_{avg} = 3.4 \cdot 10^{-4}$$

Comparing such value with the one achieved with the same structure and non-saturated training set (i.e. $J^*_{avg} = 4.2 \cdot 10^{-5}$ from Tab. 4.1), a huge decrease of prediction accuracy is recorded due to both the saturation of the tanks' levels and the higher range of values spaced by the input signals (let us recall they are now inside the range $[-0.9 \cdot q^0_{a,b}, +0.9 \cdot q^0_{a,b}]$).

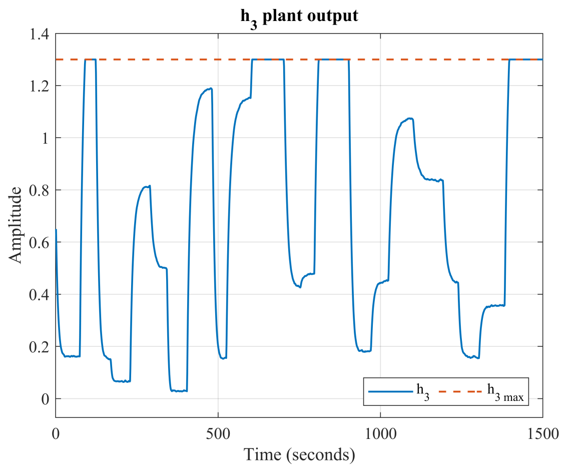
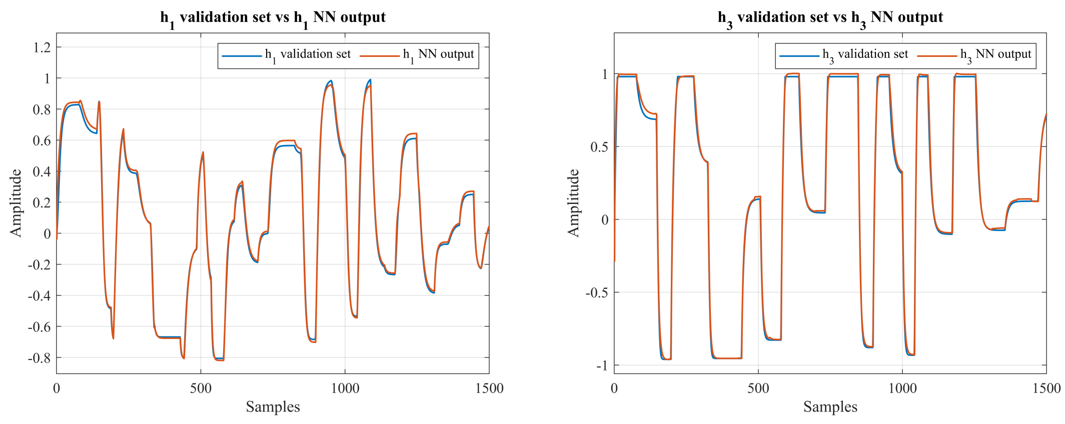


Figure 4.9. Example of h_3 level when it reaches its maximum value $h_{3,max}$ (the upper limit is highlighted in red).



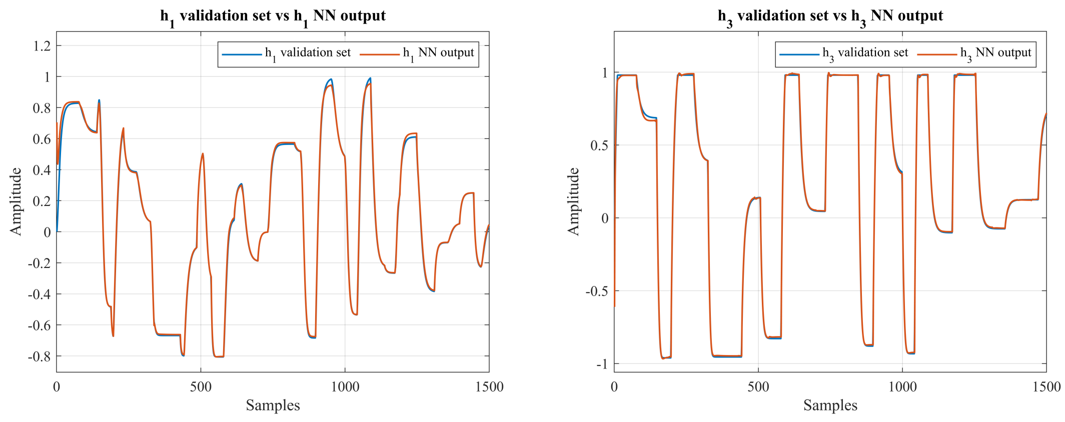
(a) (b)

Figure 4.10. (a) h_1 predicted by the NN (5 units, 1 layer) versus validation signal; (b) h_3 predicted by the NN (5 units, 1 layer) versus validation signal.

Considering now the same saturated dataset with a GRU NN characterised by only one layer and 30 units, we are expecting a higher precision in terms of accuracy, according to the trainings performed with the common non-saturated dataset in Section 4.1. In terms of values, the corresponding J^*_{avg} is equal to:

$$J^*_{avg} = 1.8 \cdot 10^{-4}$$

Comparing the two values of the average cost function, it decreased of almost one half with respect to the one recorded with the previous structure (one layer and 5 units only), thanks to the increase of units in the same layer. In Figure 4.11b the predictions look more accurate than the ones of Figure 4.10b in the saturated areas, showing once again how the presence of more units leads to more accurate results for this kind of system. However, let us point out how the matching of the dataset is not perfect and still shows some problems at saturation.



(a) (b)

Figure 4.11. (a) h_1 predicted by the NN (30 units, 1 layer) versus validation signal; (b) h_3 predicted by the NN (30 units, 1 layer) versus validation signal.

In Table 4.2, the results of the trainings performed with the saturated dataset are summarised.

Considered tanks	Type of cell	n. of layers	n. of units	J^*_{avg}
$h_1,\ h_3$	GRU	1	5	3.4e-04
$h_1,\ h_3$	GRU	1	30	1.8e-04

Table 4.2. Table that summarize the training experiments performed in Section 4.2 considering only the two-tanks system with $h_1,\ h_3$, the corresponding network architecture and the value of J^*_{avg}.

4.3 Training experiments with TBPTT

Let us recall that up to now the dataset (described in Section 3.2) used for all the training operations, independently from how the input signals are built in terms of noise, amplitude etc., consists in a set of 20 experiments with 1500 samples each for every quantity taken into account. Reasoning from a practical point of view, performing such a high number of experiments and collecting all these samples is difficult, therefore a more suitable approach is the TBPTT introduced in Section 1.3.3. Summarizing the idea, a single long experiment is performed, from which (shorter) batches for training and validation are extracted. In addition to ease the collection of data, this approach also guarantees a faster training of the network, since the unfolding procedure is performed for a limited number of time-steps, on the basis of the length of each subsequence.

Thus, a change of the dataset has to be introduced: we collect 12000 samples for each variable once the sampling of the original signals is done, hence the time length of each signal is $12000 \cdot T_c = 12000 \cdot 25 = 300000\ s$. Notice that, since it is required to generate a training set and a validation set, and the latter is characterised by no noise for our choice (in this way we avoid that the value of the performance index is influenced by the WN), the idea adopted is to collect 10000 (noisy) samples for the training and 2000 (noise-free) samples for the validation process. Therefore, starting from the training set, the batches for training are created as follows:

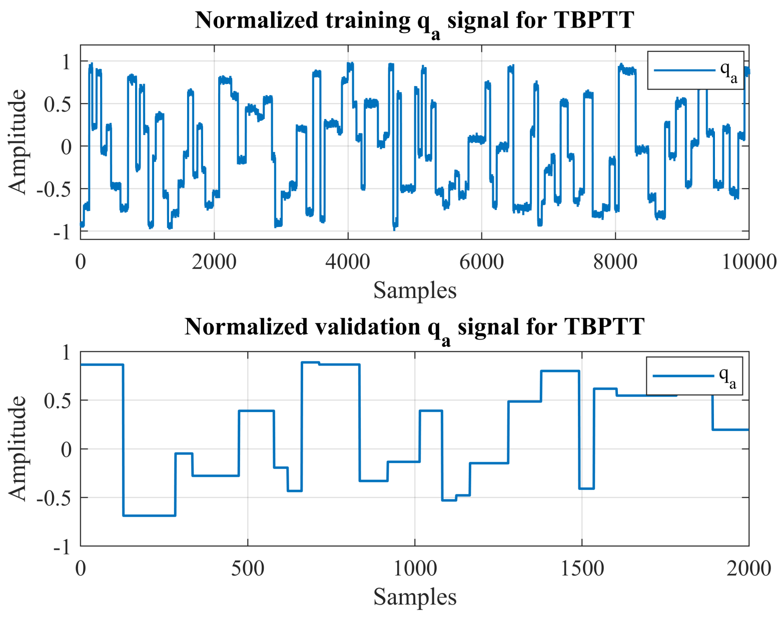
1. the length of the batches l_{batch} is decided (in our case, it will be equal to 400 time-steps, and the same for all the batches), together with the overlap term, indicated as l_{over}. The latter indicates the number of common samples in two consecutive batches, and it will be the 10% of the batch length, i.e. $l_{over} = 40$. Notice this helps to prevent that the NN does not learn data patterns longer than the established batch size since we do not consider just new data, but also a part of the one contained in the previous batch;

2. the batches are extracted sequentially from the dataset, imposing that the last l_{over} elements of the batch i are equal to the first l_{over} elements of the batch $(i + 1)$. Hence, the first l_{batch} elements of the dataset form the first batch, then the elements from $(l_{batch} - l_{over})$ to $(2 \cdot l_{batch} - l_{over})$ of the dataset form the second one, and so on. In our case, the number of batches created for the training will be 27, given by $n_{batch} = \left\lfloor \dfrac{10000 - l_{batch}}{l_{batch} - l_{over}} \right\rfloor + 1$;

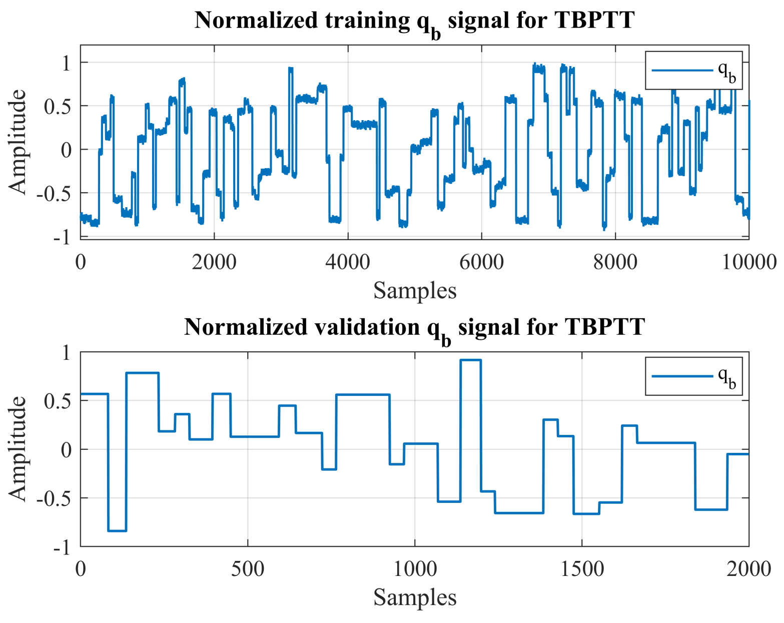
3. the last 1500 elements out of the 2000 samples of noise-free dataset are used for validation, so the performances of the NN are still evaluated on the basis of 1500

data like in Sections 4.1 and 4.2, even if the training is done with subsequences of 400 samples.

From the point of view of input signals, the features described in Section 3.2 are still ensured (with the changes previously mentioned), and the structure of the network presented in Section 3.1 is still valid. An example of the generic inputs q_a and q_b (for training and validation) are respectively indicated in Figures 4.12a and 4.12b.



(a)



(b)

Figure 4.12. (a) Example of normalized q_a input used for TBPTT approach; (b) Example of normalized q_b input used for TBPTT approach.

As previously said, it is clear from these two images that the 2000 samples of the validation signals are noise-free. Notice that the noise length of each step in the training signals is equal to the whole time length of the step itself, in order to be able to make a comparison with the results of Section 4.1. At this point, the trainings are ready to be executed, focusing on:

- GRU with 5 units and 1 layer (for both two-tanks and four-tanks systems);
- GRU with 30 units and 1 layer (for both two-tanks and four-tanks systems);
- GRU with 15 units per layer and 2 layers (for two-tanks system).

Even in this case, each training is repeated three times, and the quantities J^*_{avg}, t_{avg} are obtained with Eq. (4.1). At every epoch the batches order is reshuffled in order to avoid overfitting. Also, notice that the number of samples for the training passed from $1500 \cdot 19 = 28500$ to 10000 for each variable, so lower values for average training times are expected not only for the choice of the truncated approach.

4.3.1 GRU with 1 layer and 5 units

Let us start focusing on the best architecture selected with the training experiments in Section 4.1, i.e. a NN with GRU cells, one layer and 5 units, considering the two-tanks system with h_1, h_3 as outputs.

Recalling what was previously said during the introduction to TBPTT approach, this method should guarantee a huge improvement of training in terms of time required, since the unfolding of the RNN is limited. Checking the obtained results, we have:

$$J^*_{avg} = 5.3 \cdot 10^{-5}$$
$$t_{avg} \simeq 15\ min$$

It is clear that the time required for training is reduced by almost a factor two (15 minutes versus 25 minutes), as expected: it can be for sure associated to truncated BPTT that speeds up a lot the training procedure, but also partially to the reduction of the samples used for training. On the other hand, the accuracy of predictions decreases a bit.

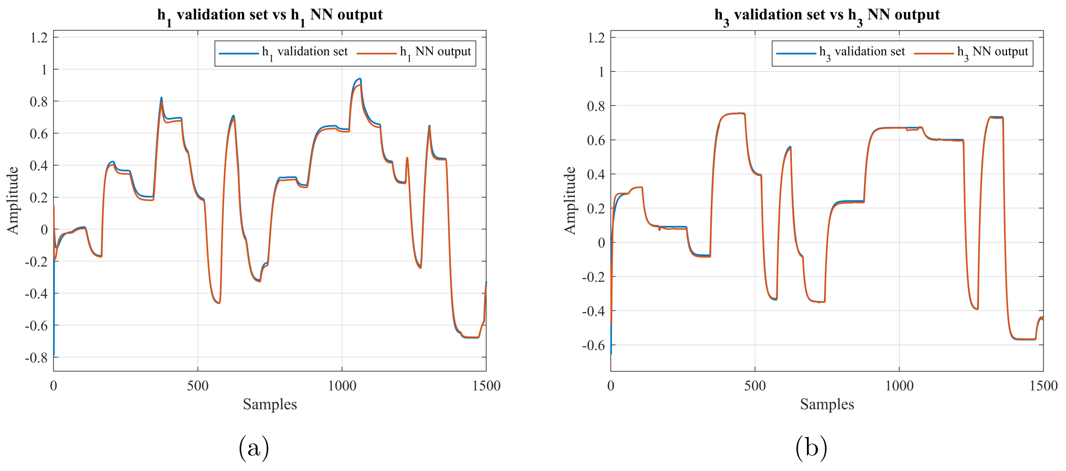
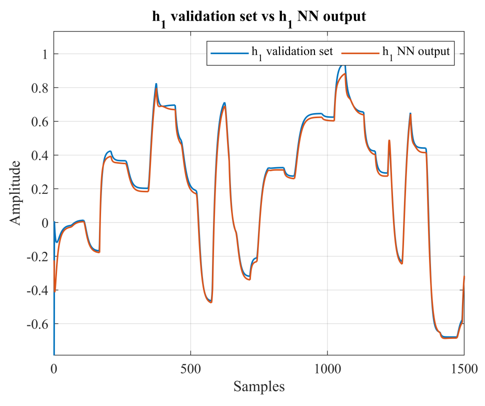


Figure 4.13. (a) h_1 predicted by the NN (5 units, 1 layer) versus validation signal; (b) h_3 predicted by the NN (5 units, 1 layer) versus validation signal.

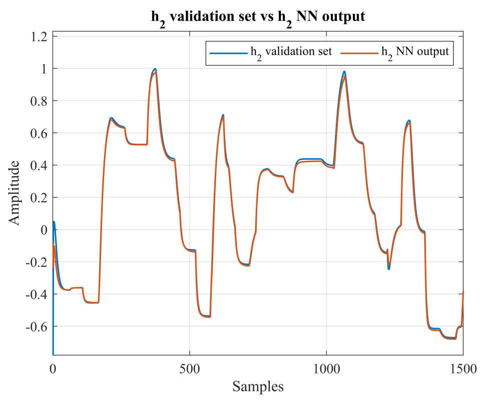
Dealing with the four-tanks MIMO system (all the four outputs are considered), and keeping the same NN structure, a similar behaviour is obtained:

$$J^*_{avg} = 1.4 \cdot 10^{-4}$$
$$t_{avg} \simeq 15\ min$$

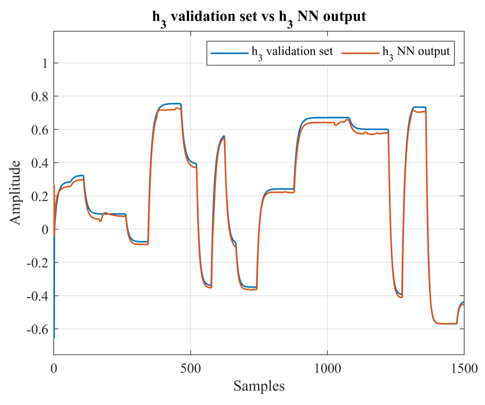
The time shows no increase, while J^*_{avg} increases consistently due to the presence of four tanks' levels to be estimated; however, its value is still very low and considered acceptable, therefore the network structure can be left the same (no changes are required neither in the units nor in the number of layers). Considering now the comparison with the BPTT approach and same architecture, the t_{avg} term shows a clear improvement in terms of required time, while J^*_{avg} increases a bit, passing from $J^*_{avg} = 9.7 \cdot 10^{-5}$ to $J^*_{avg} = 1.4 \cdot 10^{-4}$. In Figures 4.13 and 4.14 the results of the two trainings are shown. We can compare Figures 4.13b and 4.14c that refer to the h_3 level, noticing how in the first case with the two-tanks system the accuracy of $\hat{y}$ is much higher: this denotes once again how the ease of training with a reduced number of tanks is higher than the one of the four-tanks system. About h_1, the two neural networks behave in a similar way, with a slightly better performance of the first one.



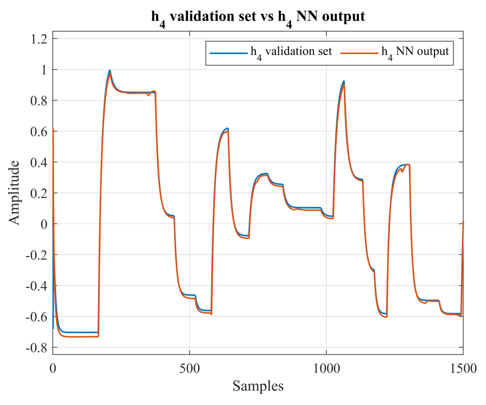
(a)



(b)



(c)



(d)

Figure 4.14. (a) h_1 predicted by the NN (5 units, 1 layer) versus validation signal; (b) h_2 predicted by the NN (5 units, 1 layer) versus validation signal; (c) h_3 predicted by the NN (5 units, 1 layer) versus validation signal; (d) h_4 predicted by the NN (5 units, 1 layer) versus validation signal.

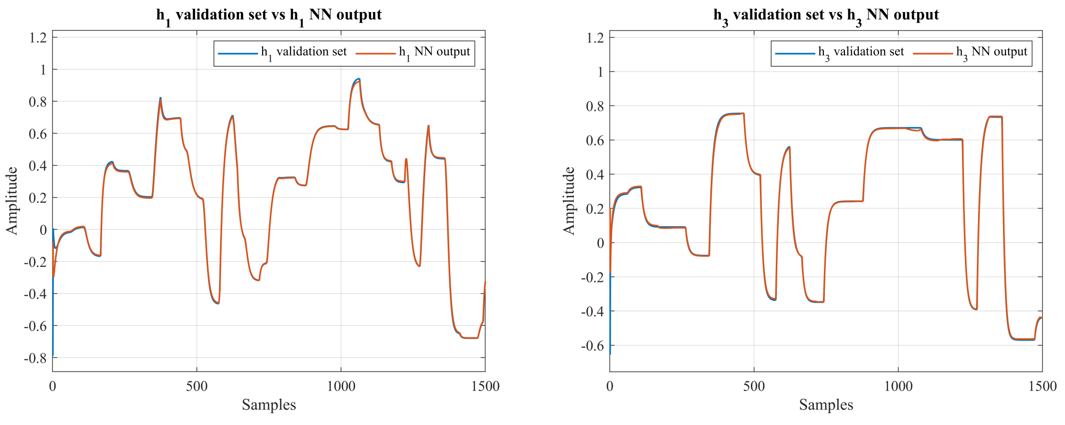
4.3.2 GRU with 1 layer and 30 units

Since with common BPTT the average J^* obtained with 30 units and a single layer was lower than the one with only 5 units, let us try to check if it holds even with TBPTT. The final results are:

$$J^*_{avg} = 4.5 \cdot 10^{-5}$$
$$t_{avg} \simeq 16 \; min$$

As expected, the J^*_{avg} decreases due to the higher number of units, leading to better performances of the NN, but not so remarkably as with the BPTT approach. Considering Figures 4.15a, 4.15b it can be noticed how the reduction of the average cost function corresponds to a visible improvement of the performances if compared to Figures 4.13a, 4.13b, especially for h_1 where almost no mismatches are observable anymore.

Also, t_{avg} shows a very limited increase: indeed, due to the action of TBPTT the average training time has been affected in a very limited way.



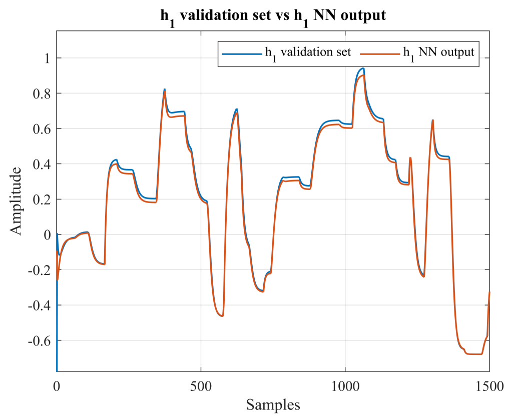
(a) (b)

Figure 4.15. (a) h_1 predicted by the NN (30 units, 1 layer) versus validation signal; (b) h_3 predicted by the NN (30 units, 1 layer) versus validation signal.

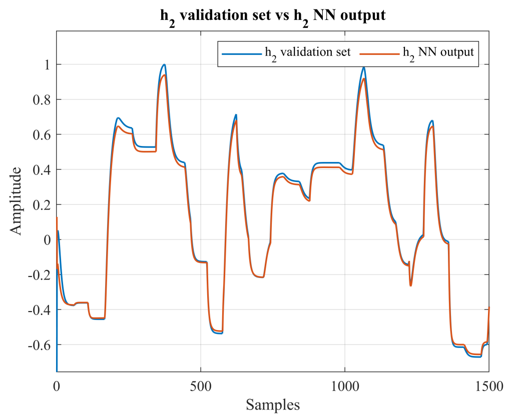
For the sake of knowledge, let us consider all the four tanks as outputs, dealing with the same RNN structure. In Section 4.3.1, where a RNN with 5 units only has been treated, the switching from two-tanks system to four-tanks system led to higher value of both J^*_{avg} and t_{avg}. In this case, the corresponding results considering 30 units are:

$$J^*_{avg} = 6.0 \cdot 10^{-5}$$
$$t_{avg} \simeq 20 \; min$$

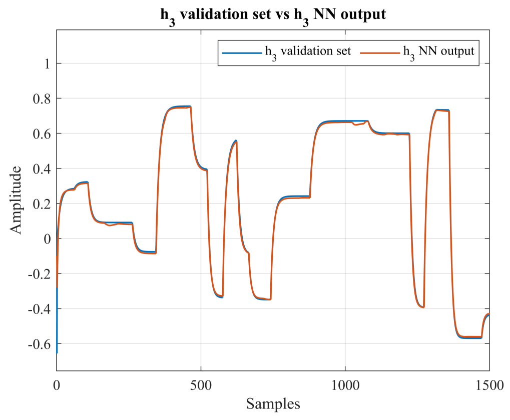
As expected, the same behaviour is observed: the average cost function switched from $4.5 \cdot 10^{-5}$ to $6.0 \cdot 10^{-5}$, with an increase of the required training time too. However, let us notice that the increase of J^*_{avg}, if compared to the one recorded for 5 units NN case in Section 4.3.1, is much more limited and acceptable. In Figures 4.16a, 4.16b, 4.16c and 4.16d the results of the NN performances are shown.



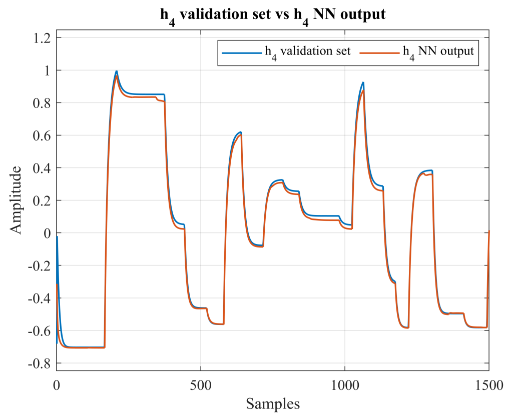
(a)



(b)



(c)



(d)

Figure 4.16. (a) h_1 predicted by the NN (30 units, 1 layer) versus validation signal; (b) h_2 predicted by the NN (30 units, 1 layer) versus validation signal; (c) h_3 predicted by the NN (30 units, 1 layer) versus validation signal; (d) h_4 predicted by the NN (30 units, 1 layer) versus validation signal.

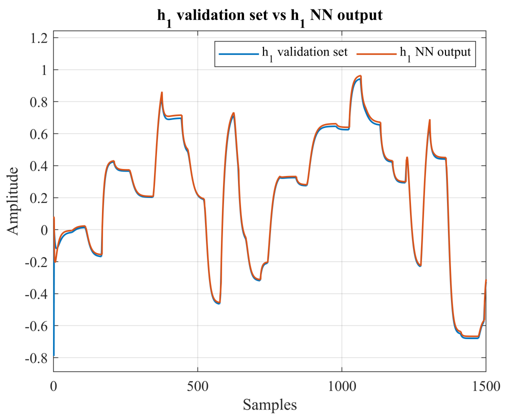
4.3.3 GRU with 2 layers and 15 units each

From Section 4.1, it is known how adding layers to the network structure (keeping the same number of overall units) leads to worse performances in our specific case, and it is not so convenient from this point of view: let us try to check if it holds true even with the truncated BPTT approach.

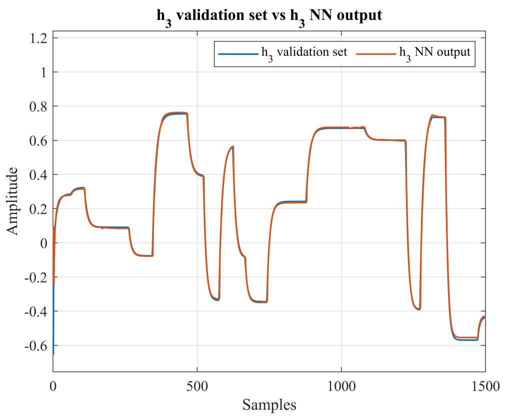
Let us consider the two-tanks system with h_1, h_3 in Figure 4.1a and a RNN characterised by GRU cells and 2 layers with 15 units each. Then, the obtained results are:

$$J^*_{avg} = 1.4 \cdot 10^{-4}$$
$$t_{avg} \simeq 30\ min$$

The expected behaviour is shown: compared to the case of GRU with one layer and 30 units, the average training time increases (almost doubled) and the same for J^*_{avg}, indicating worse results in terms of accuracy. In Figures 4.17a and 4.17b the corresponding h_1, h_3 levels predicted by the RNN are compared to the validation ones.



(a)



(b)

Figure 4.17. (a) h_1 predicted by the NN (2 layers, 15 units each) versus validation signal; (b) h_3 predicted by the NN (2 layers, 15 units each) versus validation signal.

In Table 4.3, the results of the training operations with TBPTT are summarized. Analysing its contents, the best compromise between the average J^* value and the time required for training is once again the network configuration with 5 units and an unique layer for the two-tanks system with $h_1,\ h_3$, while for the four-tanks one the configuration with 30 units and a single layer is preferable.

Considered tanks	Type of cell	n. of layers	n. of units	J^*_{avg}	t_{avg}
$h_1,\ h_3$	GRU	1	5	5.3e-05	15 min
$h_1,\ h_2,\ h_3,\ h_4$	GRU	1	5	1.4e-04	15 min
$h_1,\ h_3$	GRU	1	30	4.5e-05	16 min
$h_1,\ h_2,\ h_3,\ h_4$	GRU	1	30	6.0e-05	20 min
$h_1,\ h_3$	GRU	2	15	1.4e-04	30 min

Table 4.3. Table that summarize the training operations performed in Section 4.3 considering TBPTT approach and the treated tanks, the corresponding network architecture, the value of J^*_{avg}, the approximated time length of the training. Notice that J^*_{avg} and t_{avg} are obtained respectively as the mean value of J^* of the 3 tests and as the mean value of the time length of the 3 tests, as indicated in Eq. (4.1).

4.4 Training of two-tanks system with $h_1,\ h_2$

In all the training cases analysed up to now, we have never considered the two-tanks MIMO system with $h_1,\ h_2$ as outputs (Figure 4.1c). Indeed, recalling the expressions of the benchmark in Eq. (2.3), the only two outputs are the levels of the lower tanks. This represents a challenge for the NN, since we are completely disregarding any information about the levels $h_3,\ h_4$ which influence the lower ones as explained in Chapter 2.

In order to mimic as much as possible the behaviour of the real system, we will consider for the tests a training set with fully noisy $q_a,\ q_b$ and with amplitudes such that the saturations of the tanks are reached, considering the same dataset as in

Section 4.2, hence 20 experiments where a single experiment is formed by 1500 samples for each quantity (with $q_{a,b} \in [-0.9 \cdot q^0_{a,b},\ +0.9 \cdot q^0_{a,b}]$). Let us notice that BPTT approach will be used, with a number of epochs equal to 300.

For the RNN structure, two alternatives are considered:

- GRU with 1 layer and 5 units, since it has been shown to be the best found structure for two-tanks system at the moment;
- GRU with 2 layers and 15 units each, in order to check if an increase of units and layers corresponds to better performances.

4.4.1 GRU with 1 layer and 5 units

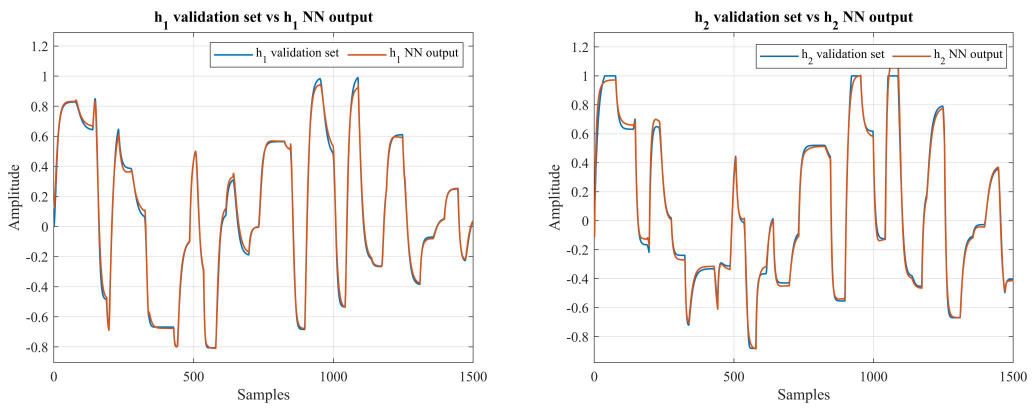
Considering the first structure, i.e. a RNN with GRU cells, one layer and 5 units, the results using Eq. (4.1) are:

$$J^*_{avg} = 8.4 \cdot 10^{-4}$$
$$t_{avg} \simeq 30\ min$$

As expected, the higher complexity of the training task results in a lower accuracy compared to values in Tab. 4.2, providing a not so good value for J^*_{avg}; more specifically, it almost doubled showing how the lack of prediction accuracy is not only related to the saturated dataset, but also due to the training complexity. Notice also that the average training time increases a bit with respect to the one required for two-tanks system with $h_1,\ h_3$ and the same structure.

We cannot compare the obtained J^*_{avg} and the one achieved for the $h_1,\ h_3$ system with the same NN structure in Tab. 4.1, since they are referring to two different datasets (the latter has been trained with a non-saturated dataset, so the accuracy of $\hat{y}$ is higher for sure). However, the poor value of the validation MSE suggests us that five units are not sufficient to compensate the missing informations about upper tanks levels, therefore other RNN structures have to be considered.

In Figures 4.18a, 4.18b the comparison between predicted and validation signals is shown, highlighting how the mismatches are distributed all over the lengths of $h_1,\ h_3$.



(a) (b)

Figure 4.18. (a) h_1 predicted by the NN (5 units, 1 layer) versus validation signal; (b) h_2 predicted by the NN (5 units, 1 layer) versus validation signal.

4.4.2 GRU with 2 layers and 15 units

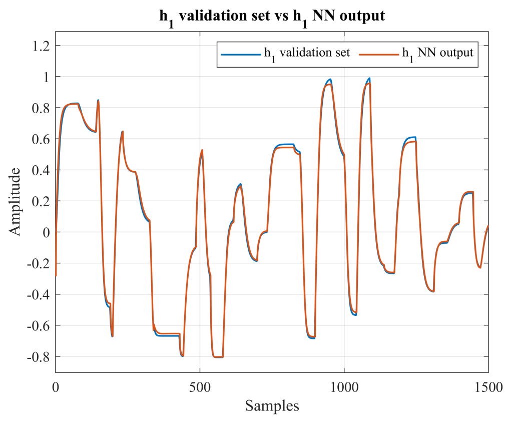
Let us try to increase both the layers and units up to 2 and 15 per layer respectively, in order to check if the complexity of the training is partially compensated by a higher number of parameters, spread over more layers. Considering Eq. (4.1), we get:

$$J^*_{avg} = 4.9 \cdot 10^{-4}$$
$$t_{avg} \simeq 50\ min$$

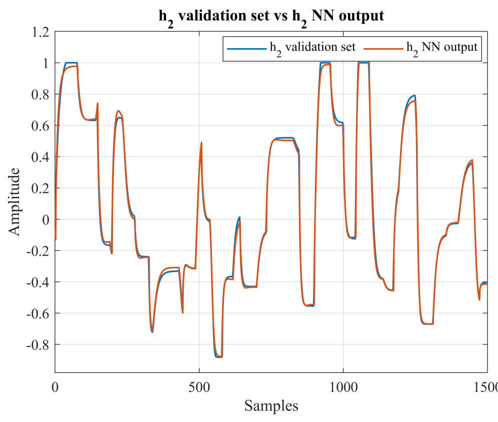
Making a comparison with the previous case of one layer with 5 units in Section 4.4.1, adding parameters to the network increases the training accuracy of almost a factor two. As consequence, the training time doubles (mainly due to the usage of a multi-layer NN). However, since we are interested in the most accurate model possible in order to replicate the system, this latter solution is preferred over the first one, disregarding the cons about the training time, and it will be used as NN model for the control scheme. To this purpose, increasing the number of epochs of the training surely leads to better result since the validation MSE depicts a decreasing trend over the epochs.

In Figures 4.19a, 4.19b the signals comparisons are shown: more accurate predictions are observable than the ones in Figures 4.18a, 4.18b.

The summary of the results about the training operations of Section 4.4 is indicated in Table 4.4.



(a)



(b)

Figure 4.19. (a) h_1 predicted by the NN (2 layers, 15 units each) versus validation signal; (b) h_2 predicted by the NN (2 layers, 15 units each) versus validation signal.

Considered tanks	Type of cell	n. of layers	n. of units	J^*_{avg}	t_{avg}
$h_1,\ h_2$	GRU	1	5	8.4e-04	30 min
$h_1,\ h_2$	GRU	2	15	4.9e-04	50 min

Table 4.4. Table that summarize the training experiments performed in Section 4.4 considering the two-tanks system with $h_1,\ h_2$, the corresponding network architecture, the value of J^*_{avg} and the average time length of the training. Notice that J^*_{avg} and t_{avg} are obtained respectively as the mean value of J^* values of the 3 tests and as the mean value of the time length of the 3 tests, as indicated in Eq. (4.1).

4.5 Final remarks

Focusing on the training operations performed in Section 4.1, and taking as reference Table 4.1, it is clear how the network defined considering GRU cells is such that it provides good results for almost all the configurations considered: in particular, the J^*_{avg} was always around 10^{-5} for most of them. However, even with LSTM cells the results were interesting, but worse than the ones considering GRUs under the same conditions: recalling what has been discussed in Section 1.3.7, GRU cells are a sort of exemplification of LSTM ones, so the lower the number of parameters to be tuned the higher the training ease. That is why GRUs are preferable in our case.

As discussed along the chapter, the best configuration discovered for the two-tanks plant with $h_1,\ h_3$ is the one with GRU cells, 5 units and a single layer. It provides a J^*_{avg} that is not the best one (its value with 30 units configuration is a bit lower), however having a lower number of parameters helps to avoid overfitting more likely, together with a (slightly) lower t_{avg}. It has been noticed also that such configuration works very well even with the four-tanks system, treating all the four levels h_i. Moreover, it has been observed how the same number of units spread over multiple levels affects the performances of the network if the systems in Figures 4.1a, 4.1b or 4.1d are considered. In fact comparing the behaviour of a single layer with 30 units and the one of two layers each one with 15 units a worse performance (in terms of both accuracy and time required) from the latter with these specific plants is recorded.

Concerning the trainings performed in Section 4.2, the aim was to check if, using a saturated training dataset (where the inputs and tanks' levels reach their maximum values), the NN is able to predict that behaviour matching the saturation zones. It has been observed that the predictions $\hat{y}$ fail to match perfectly the saturations of reference signals: more specifically, the NN with only 5 units (Figures 4.10a, 4.10b) showed worse performances than the ones of 30 units network (Figures 4.11a. 4.11b), both graphically and numerically (higher value of J^*_{avg}, as reported in Tab. 4.2). Therefore, increasing the number of units led to higher accuracy, as observed even with a non-saturated dataset, still showing limited mismatches in the maximum value areas.

Taking into account the training experiments performed using the TBPTT approach (Section 4.3), it has been shown how the average training time t_{avg} values decreased relevantly (TBPTT ensured a huge advantage in terms of velocity, scoring a training time significantly lower than the one of BPTT), while the validation MSE values indicated a slightly increasing trend: the first phenomenon is associated to the truncation of RNN unfolding to a lower number of time-steps. The second aspect may have two causes: the decrease of training samples with respect to Section 4.1 (which affects also the training time), and the impossibility of the network to learn patterns higher than subsequence length (in our case 400 samples). The overlapping between consecutive sequences helps in such problem (together with making smoother transitions between consecutive subsequences), but it does not represent a final solution: better results in terms of prediction accuracy could be achieved by increasing the length of the sequences, or increasing their number, which is equivalent to require a higher number of training samples.

However, with TBPTT similar conclusions to Section 4.1 are obtained comparing the different network structures (taking as reference the values in Table 4.3): keeping GRU cells, and passing from 5 units to 30 units, allowed us to achieve more accurate predictions of the NN, but the required time increased too. Comparing Figures 4.13a, 4.13b with Figures 4.15a and 4.15b, the difference between the $\hat{y}$ precision is observable: the matching of the 30 units NN signals is almost perfect, especially if we compare the h_1 levels. On the other hand, considering the 5 units RNN the predictions are close to real values too, so there is not the necessity to increase the network parameters. The comparison of the network's behaviours with the same number of units, but different layers, indicated exactly what was already observed, i.e. an increase of the training time and a lower prediction accuracy. Hence, implementing truncated BPTT did not lead to benefits in spreading the same number of units over multiple layers with the considered systems.

Considering Section 4.4 where the training of the two-tanks system with outputs h_1 and h_2 has been performed, the results highlight how the complexity of the training requires an higher number of units (and layers). In contrast to what has been shown up to now, the best selected architecture for this kind of system is characterised by GRU cells, two layers and 15 units each; more specifically, this architecture led (as expected) to slower training (i.e. higher t_{avg}), but a much better performance in terms of accuracy (the J^*_{avg} was almost a half smaller than the case with only one layer and five units), showing how a reduced number of units does not allow us to mimic the behaviour of this specific plant with high precision.

4.6 Conclusions

In Chapter 4, the training of the plant with different configurations has been treated (let us notice that the main elements of the network, e.g. the cost function adopted, the initialization procedure, the optimizer and so on, were kept always the same for all the training operations). We started applying the BPTT approach, focusing on the performances in terms of accuracy and average training time considering different NN structures, all with good results.

Then, additional training experiments were performed dealing with a saturated dataset, showing how the RNNs depict a lack of accuracy for saturated areas (especially the ones with a low number of units).

Afterwards, TBPTT was applied considering a non-saturated dataset, pointing out the gains in terms of training time, with J^*_{avg} values close to the ones recorded with BPTT.

Finally, the two-tanks system with h_1 and h_2 as outputs was trained with the same saturated dataset used for the additional training experiments, privileging the multi-layer RNN structure due to the high complexity of the training.

For final comments about the obtained results, take as reference Section 4.5.

Chapter 5

Internal Model Control

In this chapter, the *Internal Model Control* (IMC) design strategy is presented, together with its properties, starting from its application to a SISO, linear plant. Then, its extension to MIMO and nonlinear systems will be introduced, pointing out the main changes with respect to the SISO case.

The proposed scheme, where RNNs are used both as system model and as IMC regulator, is detailed, focusing on how the controller training is performed. A final variation of the "classical" scheme is described, adding an integral action (with and without anti wind-up scheme on the controller input) in order to guarantee zero steady-state errors for the tracking of constant reference signals.

At the end of the chapter, the control scheme will be tested for the four-tanks system considering first h_1, h_3 as outputs and then h_1, h_2, dealing with four different setpoints (two for each considered system).

5.1 IMC scheme

The IMC scheme was developed for the first time by Morari et al. in [30] and is a model based control technique, where the controller contains (implicitly or explicitly) a model of the plant. Let us assume we are dealing with a SISO, linear system (for the general discussion of IMC, continuous-time domain is considered). Take into account Figure 5.1 that depicts the feedback scheme with IMC: $G_c(s)$ is the transfer function of the controller, $G_p(s)$ the one of the plant, $G_m(s)$ the one of the model and d is the disturbance acting on the output y. It can be observed how the feedback signal is not simply the output of the real system as usual: indeed, the model error e_{IM} used as feedback is given by $y - \hat{y}$, where $\hat{y}$ is the output of our model.

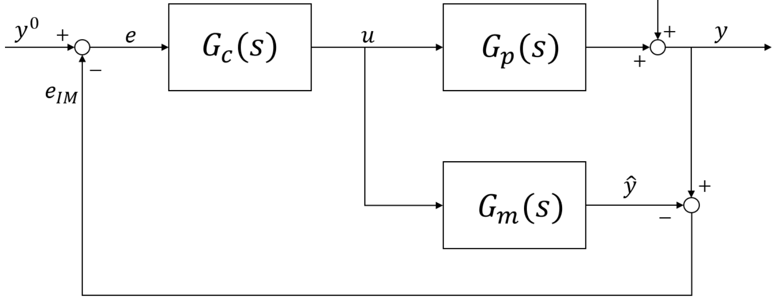


Figure 5.1. Block scheme of IMC; let us notice how the feedback signal is given by $y - \hat{y}$.

Starting from this scheme, a common feedback structure of IMC can be obtained (Figure 5.2), placing the controller and the model into a single transfer function. It follows:

$$\tilde{G}_c(s) = \frac{G_c(s)}{1 - G_c(s)G_m(s)} \tag{5.1}$$

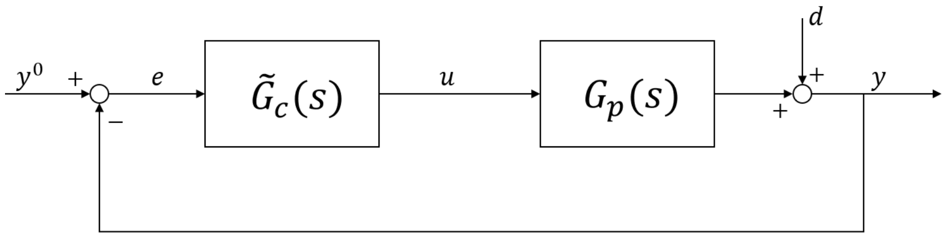


Figure 5.2. Common feedback control scheme with output disturbance.

The main advantage of the block scheme in Fig. 5.1 is that the controller is easier to be tuned if compared to the one in Eq. (5.1). Also, from a mathematical point of view it can be shown that the system from $d(s)$ to the model error is:

$$e_{IM}(s) = \frac{d(s)}{1 + G_c(s)(G_p(s) - G_m(s))} \tag{5.2}$$

Assuming that the model $G_m(s)$ is identical to the plant ($G_p(s) = G_m(s)$), the feedback error would be only $d(s)$.

In order to highlight the main properties of IMC, let us write some of the transfer functions obtained from Fig. 5.1. More specifically:

$$open - loop \;\; TF: \;\; L(s) = G_c(s)\Big(G_p(s) - G_m(s)\Big) \tag{5.3}$$

$$u(s) = \frac{G_c(s)}{1 + L(s)} \Big(y^0(s) - d(s)\Big) \tag{5.4}$$

$$y(s) = \frac{G_c(s)G_p(s)}{1 + L(s)} \Big(y^0(s) - d(s)\Big) + d(s) \tag{5.5}$$

Considering these equations, the following three properties can be stated (for the proofs and further details, the interested readers are addressed to [30], where discrete-time domain is considered).

Property 5.1.1 (Dual stability) *Let us assume that $G_p(s) = G_m(s)$, i.e. the model is perfect. Then if the process $G_p(s)$ and the controller $G_c(s)$ are (input-output) stable, the closed-loop system is stable.*

Property 5.1.2 (Perfect control) *Let us assume that $G_c(s) = G_m^{-1}(s)$, i.e. the controller is equal to the inverse of the model, and that the closed-loop system is stable. Then, if there are no disturbances, the perfect control is achieved ($y = y^0$).*

Property 5.1.3 (Offset-free) *Let us assume that the steady-state gain of the controller is equal to the one of the inverse of the model ($G_c(0) = G_m^{-1}(0)$), and that the closed-loop system is stable. Then for asymptotically constant setpoints ($\lim_{t\to\infty} y^0(t) = \bar{y}^0$) and asymptotically output disturbances ($\lim_{t\to\infty} d(t) = \bar{d}$) the output of the control scheme is offset free ($\lim_{t\to\infty} y(t) = \bar{y}^0$).*

The resulting closed-loop transfer function $T(s)$ from y^0 to y is given by:

$$T(s) = \frac{G_c(s)G_p(s)}{1+L(s)} = \frac{G_c(s)G_p(s)}{1+G_c(s)(G_p(s)-G_m(s))} \tag{5.6}$$

Focusing on Property 5.1.1 and on Eq. (5.3), the reason why $G_p(s)$ and $G_c(s)$ need to be input-output stable with $G_p(s) = G_m(s)$ is straightforward: more specifically, from Equations (5.3), (5.4) and (5.5) necessary and sufficient conditions for asymptotic stability of IMC scheme is that the roots of (5.6) must have $\mathrm{Re}(\cdot) < 0$. In the case of ideal modelling (i.e. $G_p(s) = G_m(s)$), it turns out that G_p and G_m must be asymptotically stable.

It is important to notice that, if we suppose to consider an open-loop unstable plant $G_p(s)$, the IMC scheme is not able to stabilize it: the only way to proceed should be to cancel out with the controller the unstable dynamics of the plant, which is infeasible.

Considering Property 5.1.2, it can be noticed how the ideal controller leads to ideal performances of the IMC scheme. On the other hand, Property 5.1.3 describes how the control scheme guarantees (under suitable conditions) no offset for the output with constant inputs, even in the absence of an additional explicit integral action.

Clearly, in general it is not possible to adopt a perfect controller in real cases, due to multiple reasons, such as a strictly proper model, an unstable zero or time delays in the model or plant [32]. Therefore, the design of the controller is performed in two steps:

1. the model is factorized, i.e. $G_m(s) = G_m(s)^- G_m(s)^+$, where $G_m(s)^+$ contains all the unstable zeros and time delays;

2. a *Low-Pass Filter*, LPF, $F(s)$ is introduced in order to make the controller feasible and equal to
$$G_c(s) = \Big(G_m(s)^-\Big)^{-1} F(s) \tag{5.7}$$

The introduction of a properly selected LPF can guarantee that $G_c(s)$ is proper and can provide some robustness (see [30]). Notice that the filter must be characterised by unitary gain, i.e. $F(0) = 1$, to maintain Property 5.1.3. For further details about its design, the readers are addressed to [32].

Introducing the LPF, the block scheme is changed from Fig. 5.1 to Fig. 5.3. Placing $F(s)$ in the feedback path of the control scheme, the new closed-loop transfer function becomes:

$$\tilde{T}(s) = \frac{G_c(s)G_p(s)}{1+G_c(s)F(s)(G_p(s)-G_m(s))} \tag{5.8}$$

Thus the characterstics equation to be studied to ensure the stability is:

$$\frac{1}{G_c(s)} + F(s)\Big(G_p(s) - G_m(s)\Big) = 0 \tag{5.9}$$

It can be observed that the strong advantage to introduce the LPF is that $F(s)$ can be selected in order to ensure that all the roots of Eq. (5.9) have $\mathrm{Re}(\cdot) < 0$.

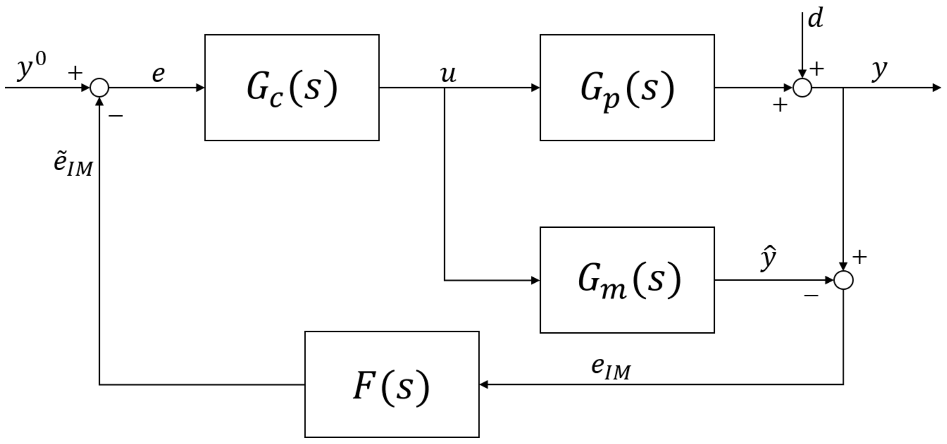


Figure 5.3. IMC scheme with a low-pass filter on the feedback branch.

5.1.1 IMC of MIMO systems

As treated in [33], the IMC scheme (Section 5.1) can be extended to MIMO systems. Clearly, a change of notations is required, more specifically in this case we talk about transfer matrices and not about transfer functions. Equations (5.4) and (5.5) become respectively:

$$\mathbf{u}(s) = \Big(1 + \mathbf{L}(s)\Big)^{-1}\mathbf{G}_c(s)(\mathbf{y}^0(s) - \mathbf{d}(s)) \tag{5.10}$$

$$\mathbf{y}(s) = \Big(1 + \mathbf{L}(s)\Big)^{-1}\mathbf{G}_c(s)\mathbf{G}_p(s)\Big(\mathbf{y}^0(s) - \mathbf{d}(s)\Big) + \mathbf{d}(s) \tag{5.11}$$

where $\mathbf{G}_c(s)$, $\mathbf{G}_p(s)$ are the controller and plant transfer matrices, $\mathbf{L}(s)$ is the open-loop transfer matrix and $\mathbf{u}(s)$, $\mathbf{y}^0(s)$, $\mathbf{d}(s)$, $\mathbf{y}(s)$ are the vectors containing the inputs, the setpoints, the disturbances and the outputs respectively.

As explained by Morari et al. in [32], the properties of the IMC scheme for SISO system (Properties 5.1.1, 5.1.2 and 5.1.3) are still valid in the MIMO case, adapting them to transfer matrices.

Also, recalling what has been said in Section 5.1 about the possibility to obtain from the IMC scheme a common feedback scheme, let us notice that this reasoning can be applied even with MIMO plants: in Equation (5.12) the new controller for the equivalent scheme in Figure 5.2 is provided.

$$\tilde{\mathbf{G}}_c(s) = \Big(\mathbf{I} - \mathbf{G}_c(s)\mathbf{G}_m(s)\Big)^{-1}\mathbf{G}_c(s) \tag{5.12}$$

The resulting closed-loop transfer matrix is:

$$\mathbf{T}(s) = \Big(\mathbf{I} + \mathbf{L}(s)\Big)^{-1}\mathbf{L}(s) \tag{5.13}$$

with $\mathbf{L}(s) = \tilde{\mathbf{G}}_c(s)\mathbf{G}_p(s)$. Analysing Eq. (5.13) it can be observed that in the case of the equivalent feedback scheme, in order to ensure the stability of the control scheme Eq. (5.14) has to be studied, even with $\tilde{\mathbf{G}}_c(s)$, $\mathbf{G}_p(s)$ stable.

$$\det\Big(\mathbf{I} + \mathbf{G}_p(s)\mathbf{G}_c(s)\Big) = 0 \tag{5.14}$$

More specifically, the roots of Eq. (5.14) need to have all real parts lower than zero. On the other hand, the advantage of the classic IMC scheme is that, ensuring the conditions of Property 5.1.1, the stability issue is immediately solved.

About the design of the controller, the procedure is equal to the one presented in Section 5.1, with the factorization of $\mathbf{G}_m(s)$ into $\mathbf{G}_m^+(s)$, $\mathbf{G}_m^-(s)$ and the introduction of a low-pass filter $\mathbf{F}(s)$ [33]. Implementing the LPF, the new IMC scheme is represented in Figure 5.4.

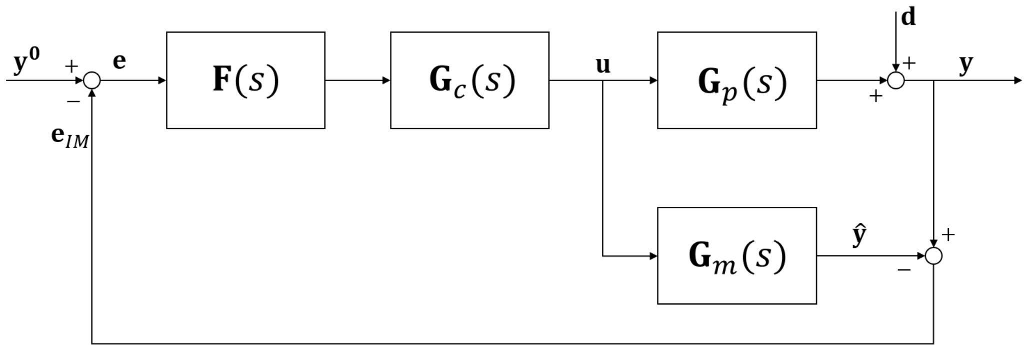


Figure 5.4. IMC scheme with low-pass filter and MIMO plant.

By a comparison between Figures 5.3 and 5.4, it can be noticed how the LPF in the MIMO case is placed immediately before the controller, instead on the feedback branch. Let us recall that $\mathbf{F}(s)$ allows us to provide robustness to the control scheme with respect to modelling errors and mismatches, filtering possible instantaneous variations of the reference signal too.

5.1.2 IMC with nonlinear systems

Up to now, the considered SISO and MIMO plants were both linear. The extension of the internal model control technique to nonlinear plants, treated in [34], needs to be considered since we deal with a nonlinear model of the benchmark (Chapter 2).

Starting from the control scheme, in Figure 5.5 the IMC scheme for nonlinear systems is introduced, where C, P, M are respectively the controller, the plant and the model (notice the double lines for the blocks means that they are nonlinear).

Focusing on the three properties observed for the SISO IMC, in this case some changes are introduced.

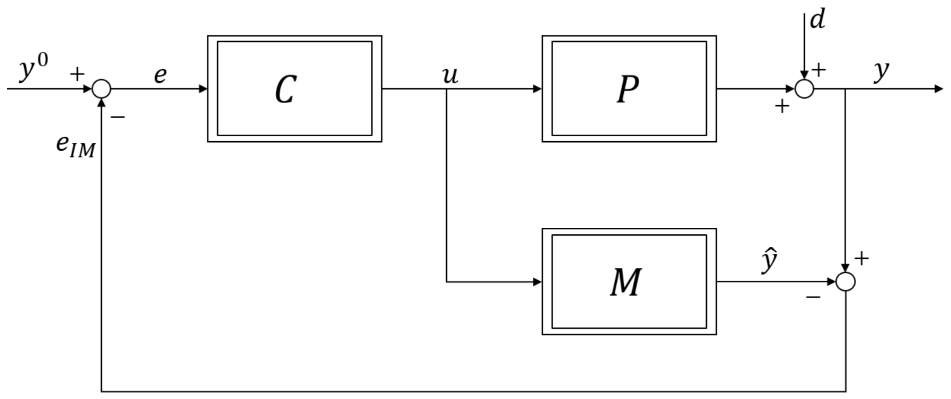


Figure 5.5. IMC scheme in the nonlinear plant case.

Property 5.1.4 (Stability) *Let us assume that C and P are (input-output) stable and $M = P$, i.e. the model is identical to the plant. Then, the closed-loop scheme is (input-output) stable.*

Property 5.1.5 (Perfect control) *Let us assume that the right inverse of the model, called M^r, exists such that $C = M^r$ and that the closed-loop system is (input-output) stable. Then, the perfect control is achieved ($y = y^0$).*

Property 5.1.6 (Offset-free) *Let us assume that the right inverse of the steady-state model, called M^r_∞, exists such that $C = M^r_\infty$ and that the closed-loop system is (input-output) stable. Then, for asymptotically constant setpoints ($\lim_{t\to\infty} y^0(t) = \bar{y}^0$) the output of the control scheme is offset free ($\lim_{t\to\infty} y(t) = \bar{y}^0$).*

In order to correctly understand Properties 5.1.5 and 5.1.6, some quantities need to be defined, in particular the definitions of right and left inverses together with the explanation of steady-state operator are discussed.

- Let us consider a generic operator M, a generic $u(t)$ (that belongs to the domain of M) and a generic $y(t)$ (that belongs to the range of M). Then, the right and left inverses of M, called respectively M^r and M^l, are defined such that:

$$M^l M u = u$$
$$M M^r y = y$$

- Let us consider a generic operator M which is input-output stable. Taking a signal $u(t)$ in the domain of M such that $\lim_{t\to\infty} u(t) = u_\infty < \infty$ and considering $y_\infty = \lim_{t\to\infty} Mu(t) < \infty$, the steady-state operator M_∞ is such that:

$$y_\infty = M_\infty u_\infty$$

Even with nonlinear systems, a low-pass filter could be introduced in order to face modelling errors and to introduce robustness against them; at this purpose, take as reference Fig. 5.4 with the filter placed in series with the controller. Clearly in this case the LPF may be nonlinear. An additional advantage of a filter in the IMC scheme is that it helps to reduce the effects of instantaneous variation of the setpoints, together with the reduction of noisy signals consequences.

5.2 IMC with neural networks

Up to now, the discussion of internal model control has been focused on general linear and nonlinear frameworks, not specifically for NNs. In this section, we will treat the application of IMC scheme with neural networks as model of both the plant and the controller.

Considering Figure 5.6, it can be observed how the controller and the model are replaced by two recurrent neural network structures. It means that two different trainings need to be performed: one for the model (already described in Section 4.4) and one for the controller. Notice also the presence of the LPF with unitary gain.

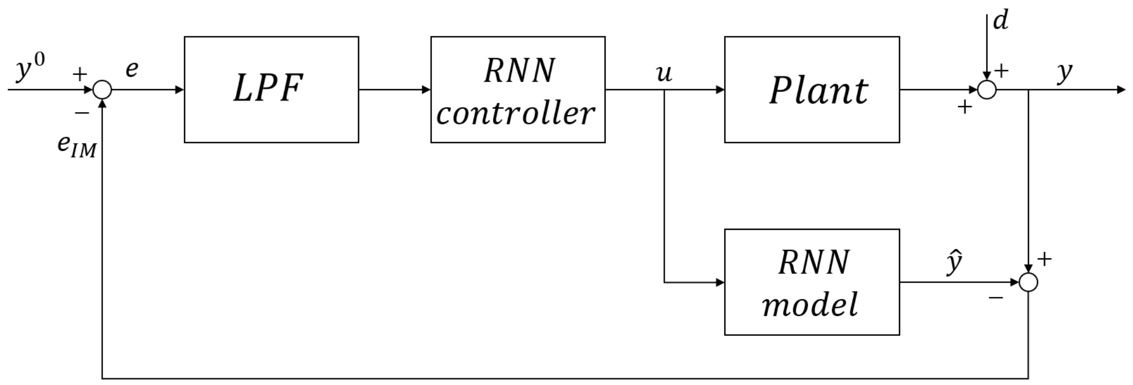


Figure 5.6. IMC scheme with neural networks as controller and model.

Starting from the discussion in [35], where a nonlinear IMC using NNs (in particular FFNNs) in discrete-time domain is treated, we can apply the procedure for RNNs, still dealing with the discrete-time domain.

The usage of a RNN controller allows us to avoid all the controller design procedure observed in the previous IMC cases, where the model was factorized and the LPF was designed to ensure the controller feasibility. Indeed, the only task to be faced is the training of such RNN (together with the one of the model, already deeply discussed), dealing with its training dataset and the main features.

Let us recall that performing the training allows us to change the parameters θ of the RNN in order to make the model as close as possible to the plant, while the controller training aims to guarantee its closeness to the plant inverse. At this purpose, the goal of the controller is to provide the signal u in Fig. 5.6 to be fed to both the plant and the model.

It is important to notice that Properties 5.1.1, 5.1.2 and 5.1.3 are still valid under the corresponding assumptions.

5.2.1 Training of the recurrent neural network controller

Going back to our benchmark and starting from the assumption that the RNN model has been already trained, let us proceed with the training of the controller. Based on the principle of IMC, it is expected to implement a controller that consists in the inverse of the NN model of the four-tanks as previously mentioned. Since it is a tricky operation to be performed mathematically, let us proceed in the following way: from a practical point of view, we are interested in a NN such that, receiving as input the setpoint y^0 (that contains the desired levels h_1^0, h_2^0 or h_1^0, h_3^0), returns as output the signals q_a, q_b that allow the plant to reach y^0. A schematic summary for h_1, h_2 levels as controlled variables is indicated in Figure 5.7.

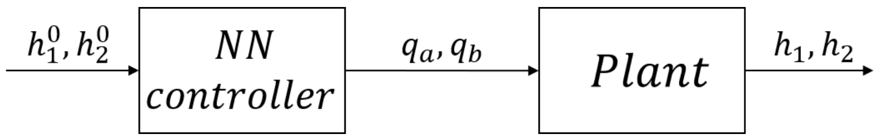


Figure 5.7. Summary of how the neural network controller in the IMC scheme works.

Analysing the procedure in [35], the training of the controller is performed considering the cascade of two NNs: the first one is a new NN (representing the controller), while the latter is the NN model. The key point is that the model is fixed throughout the training, i.e. its parameters (weights, biases) cannot be changed during the controller training. On the other hand, the weights of the first RNN are changed in order to minimize (with respect to θ) the cost function:

$$J(\theta) = J^*(\hat{y},\ \tilde{y}^0_{tr}) \tag{5.15}$$

where $\tilde{y}^0_{tr}$ is the filtered training batch. Let us recall that J^* was defined in Section 3.1.2.

The cost function takes into account the training batch and the output of the model, instead of the output of the controller u and the expected control variable u^*. This choice is related to the fact that a small difference between $q_{a,b}$ and the expected $q^*_{a,b}$ might lead to different levels on the tanks since the plant is nonlinear. Also, let us point out that different inputs might lead to the same levels. The training procedure of the controller is depicted schematically in Fig. 5.8.

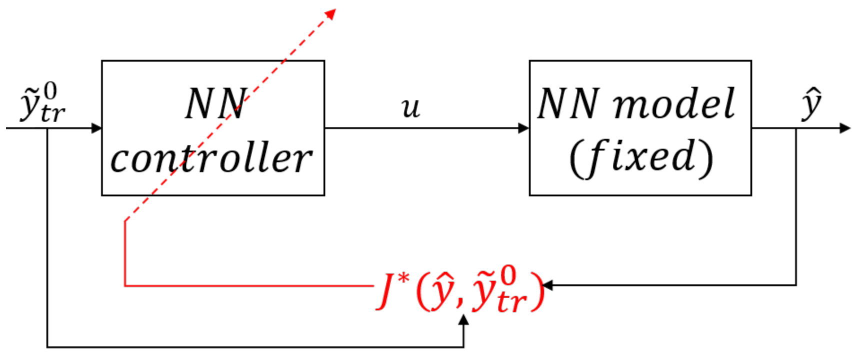


Figure 5.8. Scheme of neural network controller training for IMC; let us notice the updating signal for the controller's parameters is provided by the MSE washout of $(\hat{y}, \tilde{y}^0_{tr})$.

From a practical point of view, let us notice that in our case the operation of training is performed considering a single RNN structure: it means that, instead of taking into account two separated networks, we train only one RNN blending the layers of the model NN (with fixed parameters already estimated) and the layers of the controller NN. More specifically, we consider a single NN where the first n_c layers represent the controller, and the others n_m layers are the ones of the already trained and fixed model (with $n_t = n_c + n_m$ indicating the total number of layers of this new network). Therefore, only the first n_c layers' parameters are updated through the training procedure minimizing (5.15) as described before, where $\hat{y}$ is the output of the last layer n_t.

In Figure 5.9 the schematic representation of the unfolded RNN (with GRU units) previously described and used for the controller training operation is introduced: it can be noticed how the states of the layer n_c (i.e. the last one associated to the controller) are directly used as inputs to the layer $(n_c + 1)$ (i.e. the first one of the model structure). Let us specify that the superscript indicates the number of the layer and the subscript indicates the reference time instant, while the unfolding is limited for simplicity only from $(t - 1)$ to $(t + 1)$ time instants.

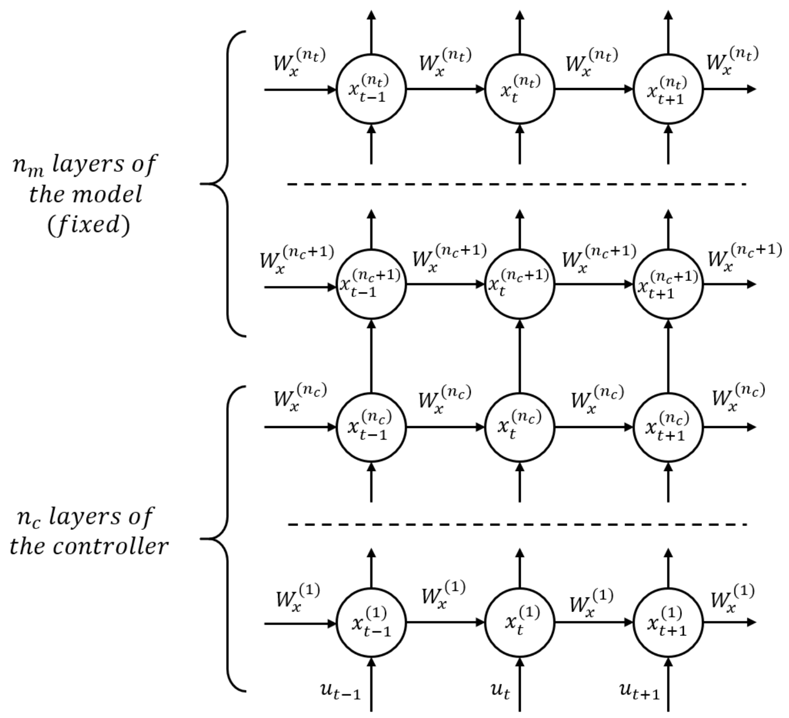


Figure 5.9. Scheme of the unfolded GRU neural network used for controller training.

Let us consider the following example to better understand what we have just explained, supposing we deal with a pre-trained model consisting of 2 layers ($n_m = 2$) and 15 units each; we could consider for the controller 3 layers ($n_m = 3$) and 10 units each. Therefore, the final RNN used for the training operation of the controller will be characterised by a total of 5 layers ($n_t = 5$).

From the point of view of the implementation of the controller in the IMC scheme, even if the training just described provides us a single trained RNN consisting of both the controller and the model together, they will be implemented separately (as indicated in Fig. 5.6): the RNN model is already available since it was trained independently, while the first n_c layers of the RNN previously discussed (indicated in Fig. 5.9) are extracted to form a separate NN, i.e. the RNN controller.

Focusing in particular on the training of the controller-model NN, the assumptions made in Section 3.1 are valid. So, a key point is represented by the training data of the RNN controller: while for the model the datasets are collected directly from the plant, avoiding any problem of feasibility, in this case this procedure is not possible (we are assuming to have available a limited quantity of data from the real world), therefore the training set has to be completely generated from scratch. The procedure followed is quite similar to the one observed for the generation of q_a, q_b in Section 3.2, with a relevant difference: generating random steps of the reference signals h_1^0, h_2^0 in terms of amplitude may lead to couples (h_1, h_2) that are infeasible for the real system, i.e. they cannot be reached by h_1, h_2 whatever inputs q_a, q_b it receives (the same may happen for h_1, h_3). Therefore, pointing out that the equations associated to the RNN model are known since it consists in a multi-layer GRU network (see Section 1.3.7 and Eq. (1.15)), a check of the generated data is performed feeding each couple of reference signals to the System (5.16) (where a four-layers GRU network has been considered since a RNN of this type will be adopted later on, with $i = 2,\ 3,\ 4$):

$$\begin{cases} \bar{z}_1 = \sigma_g(W_z^{(1)} \cdot u + U_z^{(1)} \cdot \bar{x}_1 + b_z^{(1)}) \\ \bar{f}_1 = \sigma_g(W_f^{(1)} \cdot u + U_f^{(1)} \cdot \bar{x}_1 + b_f^{(1)}) \\ \tilde{\bar{r}}_1 = \sigma_c(W_r^{(1)} \cdot u + U_r^{(1)} \cdot (\bar{f}_1 \circ \bar{x}_1) + b_r^{(1)}) \\ \bar{x}_1 = \bar{z}_1 \circ \bar{x}_1 + (1 - \bar{z}_1) \circ \tilde{\bar{r}}_1 \\ ------------------- \\ \bar{z}_i = \sigma_g(W_z^{(i)} \cdot \bar{x}_{i-1} + U_z^{(i)} \cdot \bar{x}_i + b_z^{(i)}) \\ \bar{f}_i = \sigma_g(W_f^{(i)} \cdot \bar{x}_{i-1} + U_f^{(i)} \cdot \bar{x}_i + b_f^{(i)}) \\ \tilde{\bar{r}}_i = \sigma_c(W_r^{(i)} \cdot \bar{x}_{i-1} + U_r^{(i)} \cdot (\bar{f}_i \circ \bar{x}_i) + b_r^{(i)}) \\ \bar{x}_i = \bar{z}_i \circ \bar{x}_i + (1 - \bar{z}_i) \circ \tilde{\bar{r}}_i \end{cases} \tag{5.16}$$

More specifically, we analyse if there exists a solution to (5.16): if not, it means that the couple of values considered is not feasible for the plant, requiring a change in the defined signals. Let us point out that System (5.16) describes the steady-states equations of the considered RNN, where the first two layers are associated to the controller, and the last two to the pre-trained fixed model.

Ensuring the feasibility of the data (i.e. the feasibility of each couple of the two signals h_i^0 defined), the training set is generated, consisting of 30 experiments with 1500 samples each (for each defined h_i^0). More specifically, a single reference signal is characterised by a series of steps (as done for q_a, q_b in Section 3.2) which are filtered by a LPF in order to remove instantaneous variations of the signals, as it will be described later.

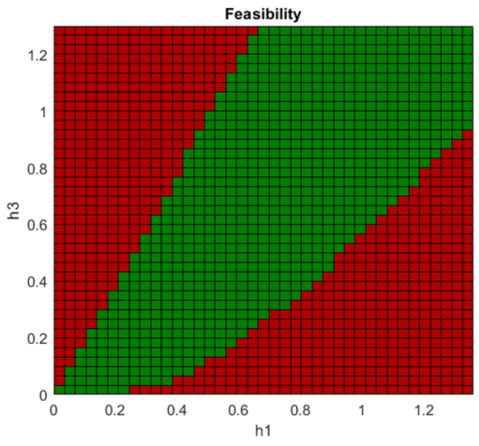
Once the training set is generated, it has to be normalized in the range [-1,1] as explained in Section 3.2.4. Let us point out that the mean and max values used for the normalization of the controller training dataset are the same used for the normalization of the NN model training dataset, with a small change: for q_a and q_b, the corresponding mean and max values are such that they are all equal to $q_{a,max}/2$ or $q_{b,max}/2$, on the basis of the signal considered (for both the training set of the model and of the controller). In order to correctly understand this choice, let us notice that the output of the controller u (which contains the values $q_{a,b}$ computed by the controller in the range [-1,1]) before it can be fed to the plant has to be denormalized in the range $[0, q_{a,b\,max}]$. Fixing the normalization constants to $q_{a,b\,max}/2$ allows us to guarantee that the variable u, after the denormalization, is able to reach both the minimum and maximum values (i.e. 0 and $q_{a,b\,max}$ respectively).

Let us consider the following example, focusing on the q_a signal only for simplicity: we assume that our maximum normalization constant is such that $\psi^{q_a}_{my,max} = \psi^{q_a}_{my,mean} = q_{a,max}/2$. When the controller output is $u = 1$ (i.e. the maximum admissible value, which is $q_{a,max}$), once the signal is denormalized to be fed to the plant, its value for q_a will be equal to:

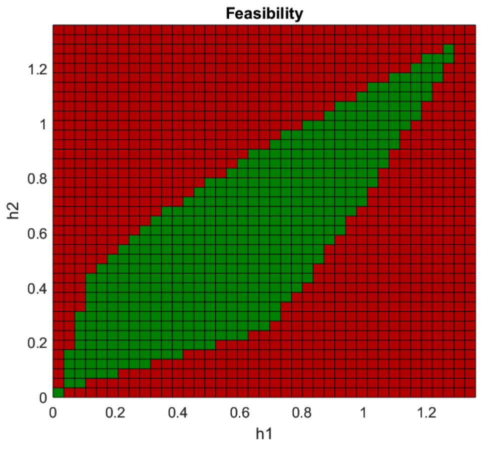
$$u_{denorm} = (u \cdot \psi^{q_a}_{my,max}) + \psi^{q_a}_{my,mean} = \psi^{q_a}_{my,max} + \psi^{q_a}_{my,mean} = q_{a,max}$$

In the same way, for $u = -1$ (i.e. the minimum admissible value) the result is:

$$u_{denorm} = (u \cdot \psi^{q_a}_{my,max}) + \psi^{q_a}_{my,mean} = -\psi^{q_a}_{my,max} + \psi^{q_a}_{my,mean} = 0$$



(a)



(b)

Figure 5.10. (a) Plant feasibility for h_1, h_3 couples; (b) Plant feasibility for h_1, h_2 couples.

To give an idea about how the levels of the tanks can be selected in order to guarantee their feasibility, in Figures 5.10a and 5.10b the feasible couples for h_1, h_3 and h_1, h_2 respectively are indicated in green, while in red all the points that the plant cannot reach; let us highlight that these two images are obtained solving directly the equations of the plant in (2.1), and not the ones of the NN model in order to neglect a (possible) mismatch between them due to the not perfect training. In particular, the green points describe the feasible steady-states for the plant: indeed, considering h_1, h_2 case, the couple $(h_{1,max}, h_{2,max})$ is not feasible even if the plant can reach it without problems with suitable inputs since it is not an equilibrium point. Hence, it is expected that all the steady-states (i.e. couples) of the training set used for the controller are inside the green area, together with the ones of the setpoints for IMC.

The final IMC scheme is depicted in Figure 5.11, where it can be noticed how the setpoint y^0 is filtered before it is passed to the controller to filter its possible instantaneous variations, as already mentioned. Let us specify that the LPF is the same used for the filtering of the training set for the NN controller.

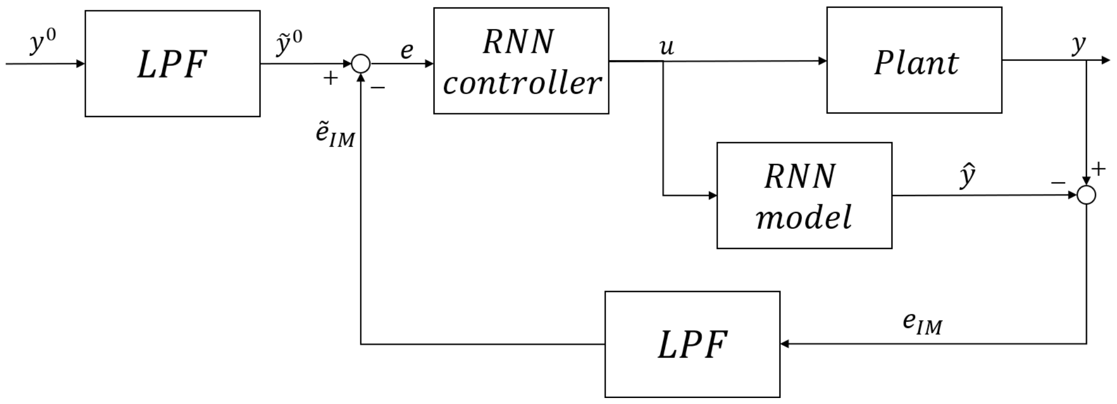


Figure 5.11. Final IMC scheme adopted for the control of the plant; let us notice that two blocks (i.e. RNNs) are used for the implementation of the controller and of the model.

5.2.2 Additional remarks

It is important to point out that, from the beginning of the discussion about NNs, the sample time T_c has been used: indeed, in the simulation of the control scheme we have to ensure that the RNN controller and the RNN model are triggered every T_c (equal to 25) seconds.

Also, as previously mentioned, the output of the controller u is in the range [-1,1], and since the plant accepts only inputs such that $u \in [0, q_{a,b\,max}]$, we need to denormalize it (using the constants retrieved from the model training set as explained earlier) before the signal can be passed to the plant. In the same way, the plant output y (which is in the range $[0, h_{i,max}]$) has to be normalized in the range [-1,1] before it can be used to compute the feedback error e_{IM}.

5.2.3 Performances with h_1, h_3

Let us consider the case with h_1^0, h_3^0 as reference levels, and the control scheme in Figure 5.11. We take into account a RNN structure for the model with 2 layers with 15 units each, while for the controller 2 layers are used, the first one with 15 units and the latter with only 2 units (in order to guarantee the compatibility with the dimensions of u without adding a linear transformation on the output of the last controller layer).

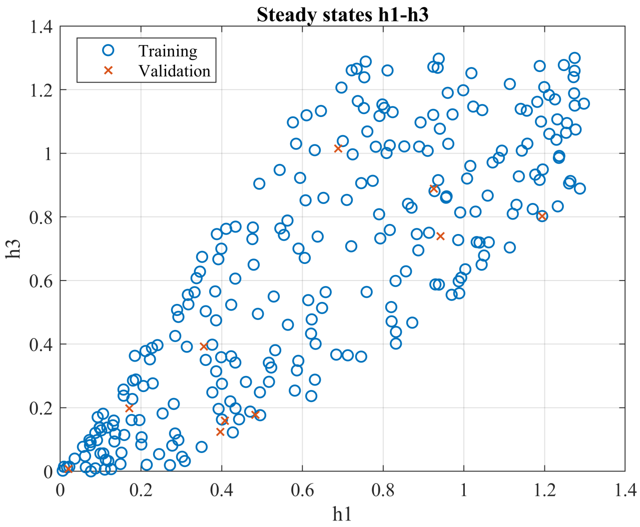
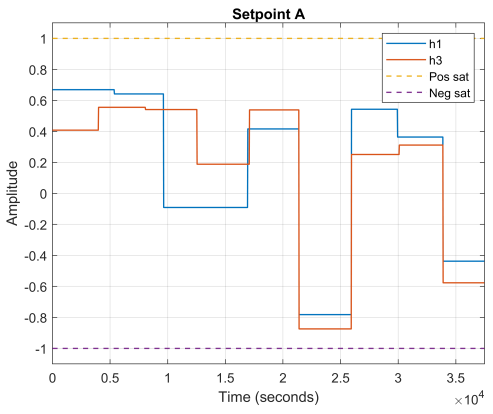


Figure 5.12. Steady-states couples for h_1, h_3 in controller training and validation datasets.

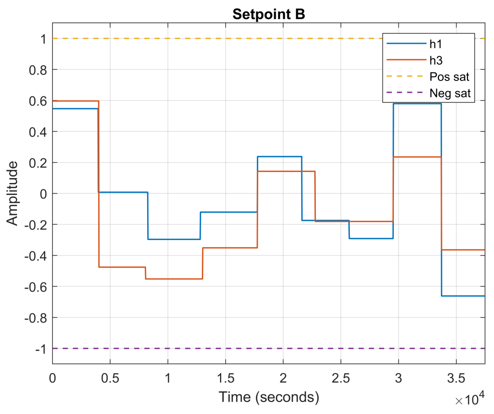
In Figure 5.12, the steady-state couples of the training set used for the controller training are indicated. Performing a comparison with Fig. 5.10a, it is clear how all the points are feasible since they are within the green area. Therefore, the training performances of the model and of the controller are:

$$Model: \; J^*_{avg} = 1.5 \cdot 10^{-4}$$
$$Controller: \; J^*_{avg} = 1.6 \cdot 10^{-4}$$

To check the behaviour of the control scheme, two setpoints A and B are respectively introduced in Figures 5.13a and 5.13b.



(a)

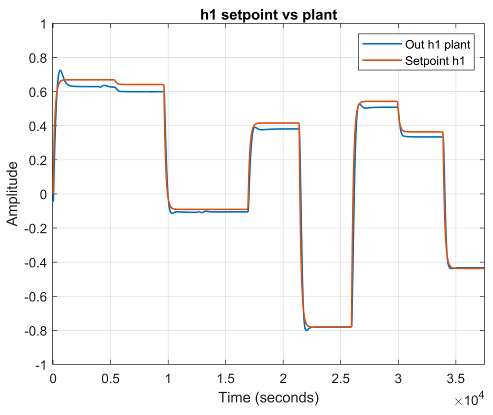


(b)

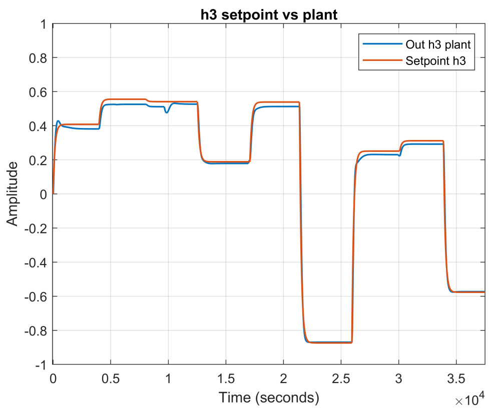
Figure 5.13. (a) Setpoint A; (b) Setpoint B.

Let us describe the results with these two trajectories:

- starting from setpoint A, the corresponding outputs of the control scheme previously introduced are provided in Figures 5.14a and 5.14b. It can be noticed that for both h_1, h_3 the plant outputs show steady-state offsets for most of the step couples. This is due to the absence of an integral action and a wrong estimate of the (local) gain, and once again it highlights how a perfect training of the model-controller system is not feasible in real world. On the other hand, the tracking of the setpoints is quite good, since the plant outputs follow the reference signals correctly without showing undesired behaviours.



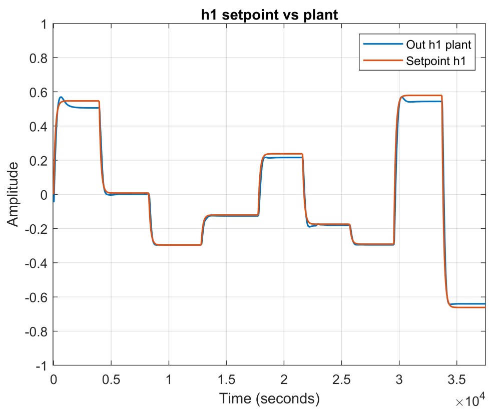
(a)



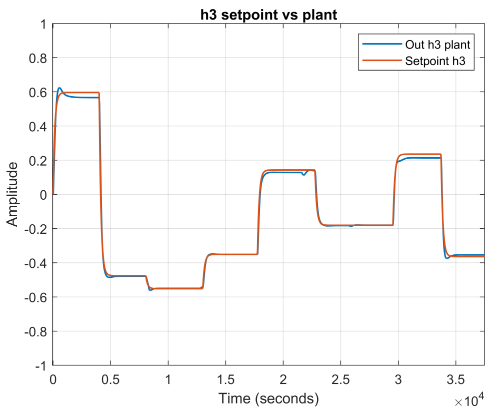
(b)

Figure 5.14. (a) Setpoint A vs output h_1 of the plant; (b) Setpoint A vs output h_3 of the plant.

- Let us now consider setpoint B (Figure 5.13b). Once again, the outputs of the control scheme, introduced in Figures 5.15a and 5.15b, depict the presence of steady-state mismatches between reference trajectories and current outputs of the real system. Even in this case, the tracking performances are good, showing that the trained model and controller are well-behaving as for the previous setpoint, i.e. no undesired behaviours are observed, with a very high accuracy of the plant outputs.



(a)



(b)

Figure 5.15. (a) Setpoint B vs output h_1 of the plant; (b) Setpoint B vs output h_3 of the plant.

Notice that increasing the training data in both the number of data and feasible steady-states couples (h_1, h_3) might lead to better tracking, reducing the offsets in amplitude and number too.

5.2.4 Performances with h_1, h_2

Let us consider now the system with outputs h_1 and h_2. As discussed before for h_1, h_3, the model is characterised by 2 layers and 15 units each, while the controller is provided by 2 layers, the first one with 15 units and the latter with 2 units. The training performances of the model and of the controller are:

$$\begin{aligned} Model: \ J^*_{avg} &= 2.1 \cdot 10^{-4} \\ Controller: \ J^*_{avg} &= 1.4 \cdot 10^{-3} \end{aligned}$$

Introducing in Figure 5.16 the steady-states of the dataset used for the controller training, it is clear how all the points are in the admissible range of values (see Fig. 5.10b). Let us notice that the blue circles denote the couples of the signals used for training, while the red crosses indicate the steady-states in the validation set.

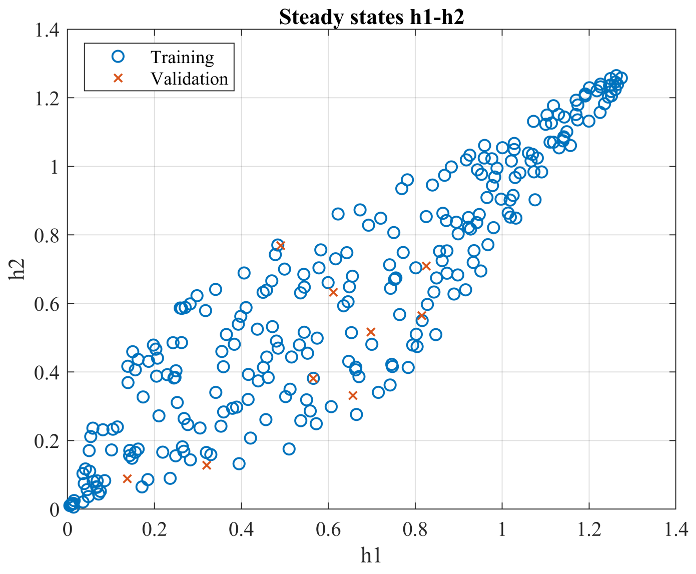


Figure 5.16. Steady-states couples for h_1, h_2 in controller training and validation datasets.

Even in this case, in order to check if the control scheme well-performs, two setpoints C and D are introduced respectively in Figures 5.17a and 5.17b.

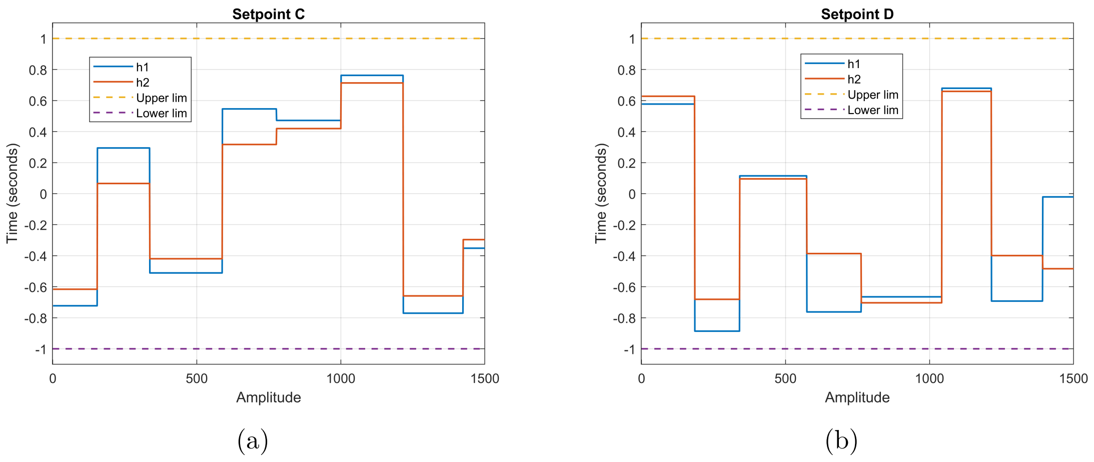
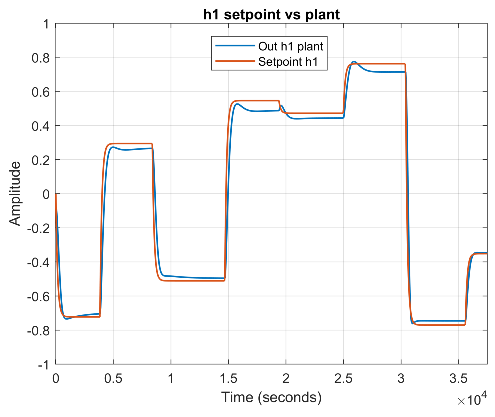
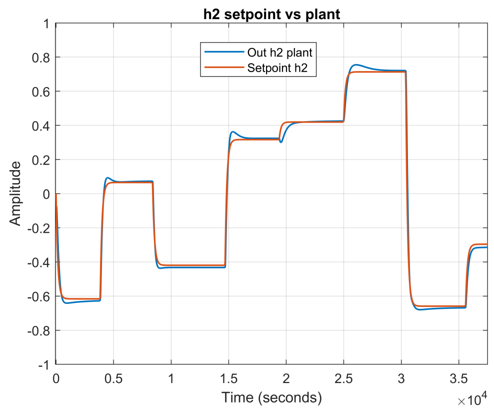


Figure 5.17. (a) Setpoint C; (b) Setpoint D.

- Let us consider trajectory C in Figure 5.17a. Considering Figures 5.18a and 5.18b where the responses of the plant h_1, h_2 are shown, the behaviour is almost the same of the one observed for the h_1, h_3 system: steady-state offsets throughout the whole signals are visible, despite a good tracking of the trajectory in terms of shapes of the signals. Due to the worse precision of the controller training than the system with h_1, h_3 the mismatches, especially for h_1, are more relevant.



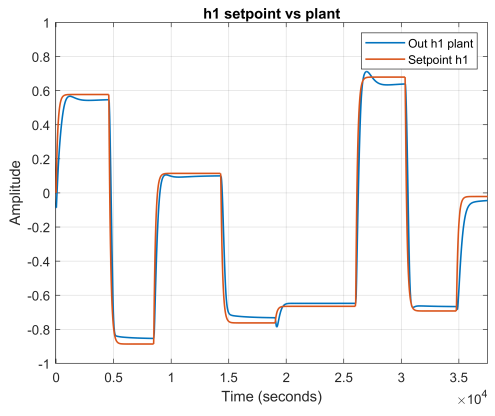
(a)



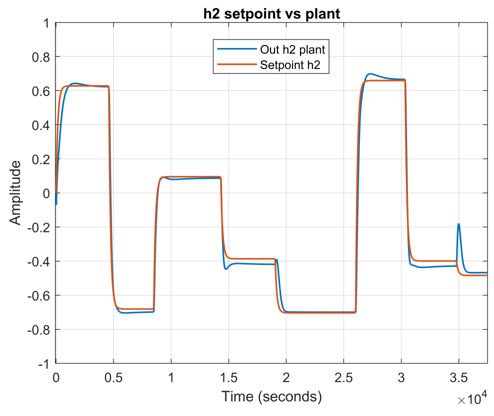
(b)

Figure 5.18. (a) Setpoint C vs output h_1 of the plant; (b) Setpoint C vs output h_2 of the plant.

- Switching to trajectory D, the results are expected to be almost identical to the case of setpoint C. Hence, analysing Figures 5.19a, 5.19b the mismatches are once again present in the plant outputs, highlighting the not so perfect training of the NNs. Also, in the h_2 case, an overshoot can be observed at $t = 3.5 \cdot 10^4\ s$.



(a)



(b)

Figure 5.19. (a) Setpoint D vs output h_1 of the plant; (b) Setpoint D vs output h_2 of the plant.

In this case, the main reason of the relevant mismatches for trajectories C and D is the low training performances of the controller: it can be observed how, between the J^*_{avg} values for the controllers of the two systems analysed, there is an order of magnitude of difference.

5.3 IMC with recurrent neural networks and integral action

Analysing the performances of the control scheme in Sections 5.2.3 and 5.2.4, it can be noticed how the results show in some cases steady-state mismatches between $\tilde{y}^0$ and y. This is due to the fact that perfect training of the model, i.e. finding a model which exactly matches the plant, is not achievable in practice, and the same for the ideal controller, which cannot be trained in order to retrieve the ideal (exact) values of q_a, q_b necessary to reach $\tilde{y}^0$.

Also, the absence of a controller able to fix model-plant mismatches, e.g. a PID, suggests us to introduce an integral action acting on the tracking error in the scheme indicated in Figure 5.6. The resulting control scheme is introduced in Figure 5.20, where the disturbance d can represent a real disturbance or the effect of a model mismatch.

Let us try to explain why the introduction of an integral action is required in the IMC scheme in Figure 5.20 (under suitable assumptions), assuming for simplicity that we are in the continuous-time domain, and all the transfer functions are SISO (as we will observe, the analysis could be extended to MIMO case too). Let us indicate with $I(s)$ the integral action ($I(s) = \frac{\mu}{s}$), with $F(s)$ the LPF ($F(s) = \frac{1}{\tau s+1}$), with P the plant, with C the RNN controller and with M the RNN model.

Let us assume that $P = M$ and $C = k \cdot P^{-1}$, with $k \in [\underline{k}, \overline{k}]$, $\underline{k} \leq 1$, $\overline{k} \geq 1$. So, we are considering that the controller is almost equal to the inverse of the plant with a small gain error (e.g. $k \in [0.9, 1.1]$). Starting from the scheme in Figure 5.21, the presence of a mismatch k between the controller and the plant inverse leads to a static gain in the control scheme (as in Fig. 5.22).

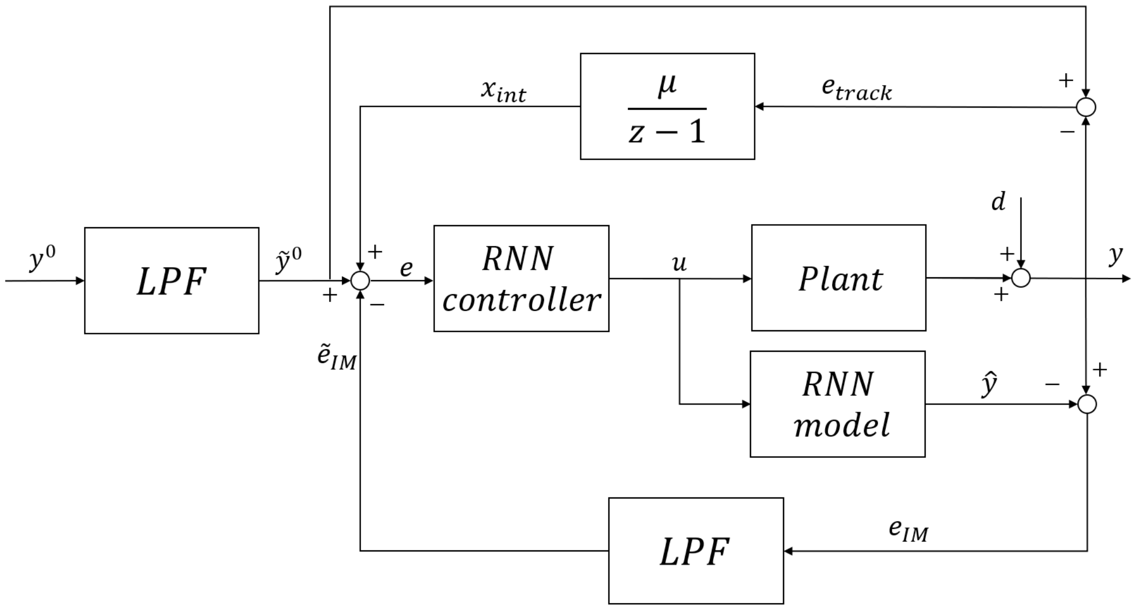


Figure 5.20. IMC scheme with RNN controller and model, and integral action.

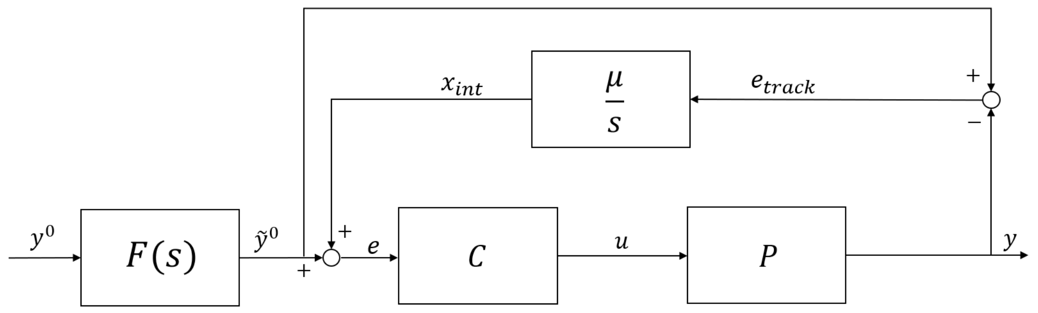


Figure 5.21. IMC scheme with integral action and perfect modelling (i.e. plant=model); notice no output disturbance is considered.

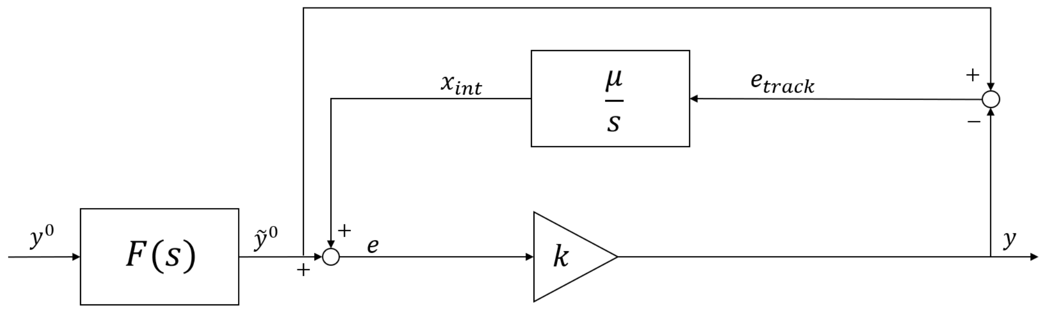


Figure 5.22. IMC scheme with integral action, perfect modelling and non-ideal controller; notice that the controller is equal to the model times a constant k.

Therefore, the resulting output signal $y(s)$ is equal to:

$$y(s) = \Big(\tilde{y}^0(s) + \big(\tilde{y}^0(s) - y(s)\big)I(s)\Big)CP$$
$$y(s)\big(1 + kI(s)\big) = \big(\tilde{y}^0(s) + \tilde{y}^0(s)I(s)\big)k$$

Thus we get:

$$y(s) = \frac{k\big(1 + I(s)\big)}{1 + kI(s)}\tilde{y}^0(s) = S_1(s)\tilde{y}^0(s) \tag{5.17}$$

$$S_1(s) = k\frac{1 + \frac{\mu}{s}}{1 + \frac{k\mu}{s}}\tilde{y}^0(s) = k\frac{s+\mu}{s+k\mu}\tilde{y}^0(s) = \frac{\frac{s}{\mu}+1}{\frac{s}{k\mu}+1}\tilde{y}^0(s) \tag{5.18}$$

From Eq. (5.18) we can observe how the DC gain of the transfer function is unitary (i.e. $S_1(0) = 1$), which means that perfect tracking is achieved for $t \to \infty$: at the steady-state the signal y is such that $y(s) = \tilde{y}^0(s)$. Therefore, in this case the presence of an integral action allows us to remove the steady-state error due to the non-ideal controller design. Let us point out that, in the absence of $I(s)$, we would get $y(s) = k\tilde{y}^0(s)$ for all values of s.

Notice once again that all the achieved results are strictly related to the (strong) assumptions introduced, like the presence of a perfect model.

At this point, let us go back to the control scheme in Fig. 5.20, focusing more specifically on the integral action introduced: we deal with a discrete-time integrator with sample time equal to T_c and gain $\mu = 5 \cdot 10^{-4}$. Therefore, the expression of the integrator is:

$$I(z) = \frac{\mu}{z-1}\,, \qquad \mu = 5 \cdot 10^{-4}$$

It is clear from Figure 5.20 how the integrator removes the mismatches between $\tilde{y}^0$ and y, as previously mentioned.

Considering now the input of the controller e, the setpoint $\tilde{y}^0$ is influenced by other two terms, i.e.:

$$e(t) = \tilde{y}^0(t) + x_{int}(t) - \tilde{e}_{IM}(t) \tag{5.19}$$

Recalling that the training of both the controller and the model were performed using normalized datasets in the range [-1,1], we have to ensure that even the controller input e is inside this range at every time instant, otherwise we could have an undesired behaviour of the controller, since it does not know exactly what to do with data outside the prescribed range. Hence, an additional implementation aspect has to be considered, introducing an anti wind-up action on the integrator. Let us highlight how this implementation has been done in order to guarantee that the output of the integral action x_{int} is always such that:

$$-1 \leq \tilde{y}^0(t) + x_{int}(t) - \tilde{e}_{IM}(t) \leq +1 \quad \forall\, t \tag{5.20}$$

The adopted implementation of the integral action added to Scheme 5.11 is provided in Figure 5.23, where the two limits of the saturation action are equal to:

$$\begin{aligned} e^{low}(t) &= -1 - \tilde{y}^0(t) - x_{int}(t) + \tilde{e}_{IM}(t) \\ e^{up}(t) &= +1 - \tilde{y}^0(t) - x_{int}(t) + \tilde{e}_{IM}(t) \end{aligned}$$

and the resulting control scheme is indicated in Figure 5.24. It can be noticed how the anti wind-up control is designed in order to be triggered when the range [e^{low}, e^{up}] is exceeded, considering as feedback signal the difference between the input and the output of the saturation.

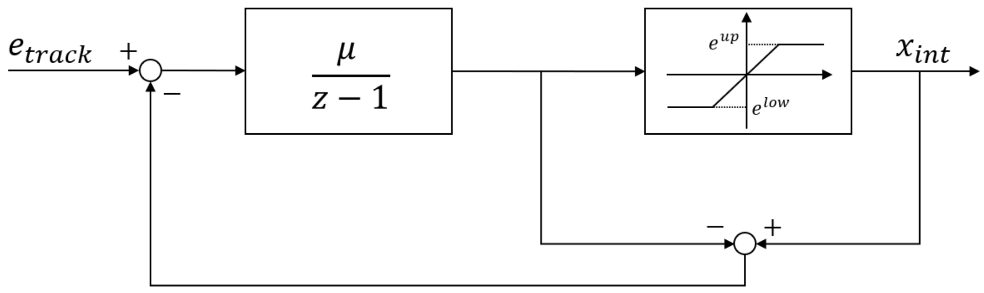


Figure 5.23. Anti wind-up scheme for the integral action; notice the limits of saturation are based on the controller input e.

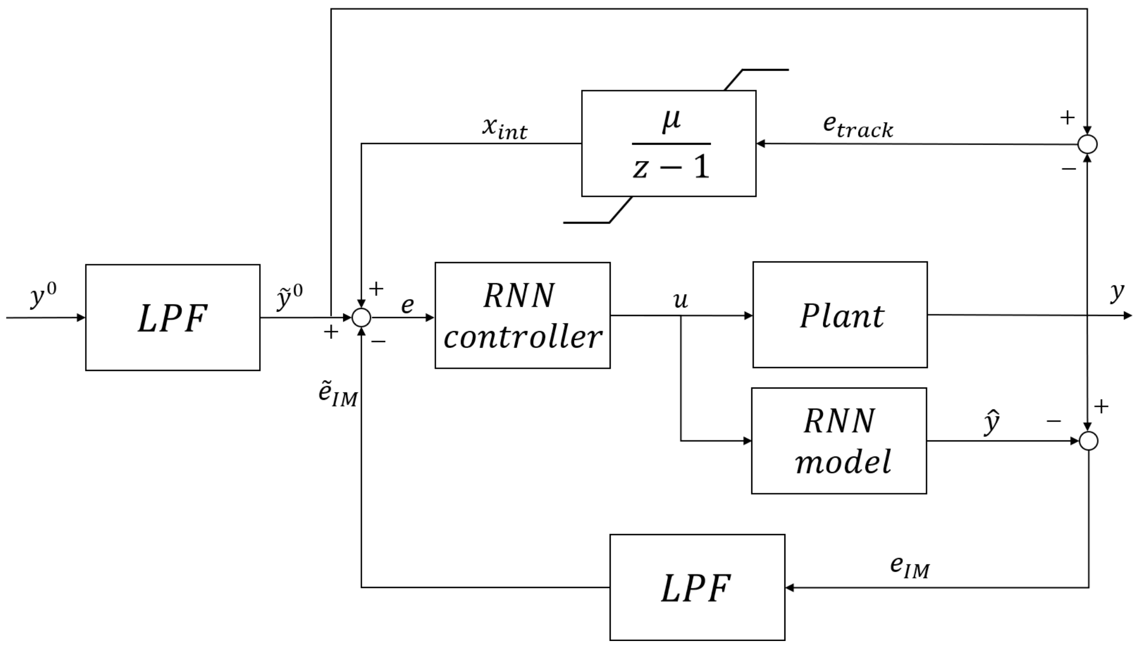


Figure 5.24. IMC scheme with integral action and anti wind-up scheme.

5.3.1 Performances with h_1, h_3

Recalling the setpoints A and B introduced in Figures 5.13a and 5.13b, we are going to assess if the introduction of the integral action is able to ensure zero-error output regulation.

- Starting from setpoint A, in Figures 5.25a and 5.25b the responses of the control scheme are represented. As expected, the mismatches for every step have been removed with the introduction of the integrator, ensuring almost a perfect reference tracking. Let us notice how for the first step of the trajectory, the offset is reduced slowly, however for a longer time length the output-reference difference would be completely cleared.

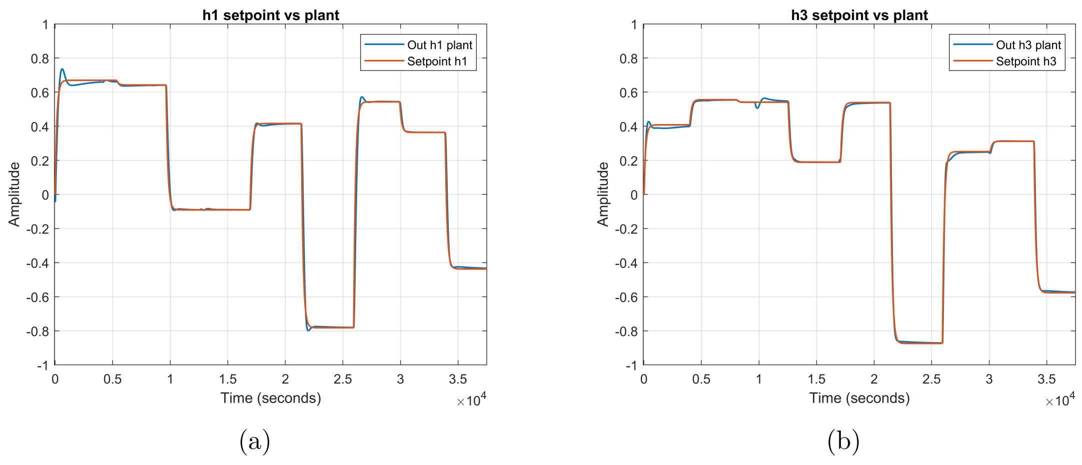
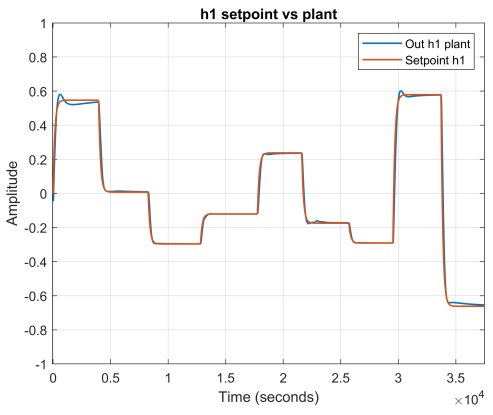
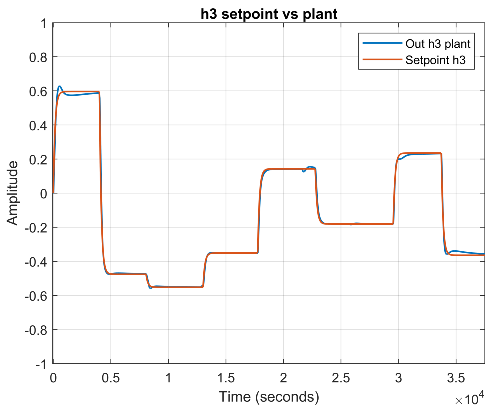


Figure 5.25. (a) Setpoint A vs output h_1 of the plant; (b) Setpoint A vs output h_3 of the plant.

- Considering setpoint B, the results are almost identical to what has been observed for trajectory A. Indeed, no steady-state errors are observed in Figures 5.26a and 5.26b.



(a)



(b)

Figure 5.26. (a) Setpoint B vs output h_1 of the plant; (b) Setpoint B vs output h_3 of the plant.

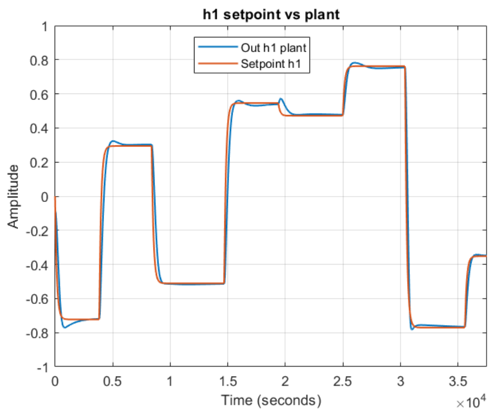
Analysing the obtained results, for the $h_1,\ h_3$ system the control scheme with integral action performs very well, removing all the steady-state errors: the tracking of the assigned trajectories is almost perfect, and the errors due to the (normal) imperfect training of the RNNs are cleared.

5.3.2 Performances with $h_1,\ h_2$

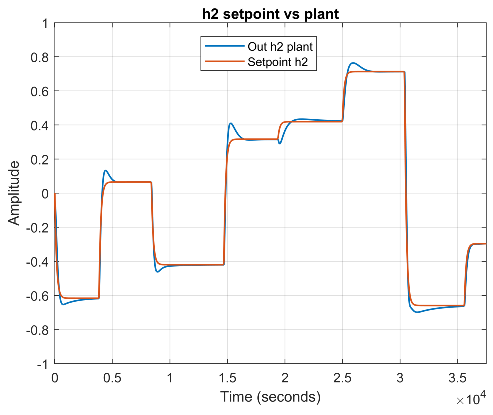
Let us focus now on the system with $h_1,\ h_2$ as reference output levels. The results without integral action have been described in Section 5.2.4, where the steady-state errors were highlighted.

We expect that the introduction of the integrator allows us to guarantee perfect tracking of the imposed trajectories, as for $h_1,\ h_3$ system in Section 5.3.1.

- Let us consider setpoint C (Fig. 5.17a): as said before, the offsets are almost removed, however especially for h_1 the tracking is not perfect; exact zero offsets may be achieved for longer steps or faster integral action (i.e. higher gain μ of $I(z)$). Also, focusing on h_2, the new scheme leads to higher overshoots amplitudes than classic IMC scheme (let us compare Fig. 5.18b and Fig. 5.27b). Notice that speeding up the integral action may lead to faster tracking, but on the other hand the peaks of the overshoots will be increased, maybe leading to oscillations on plant outputs.



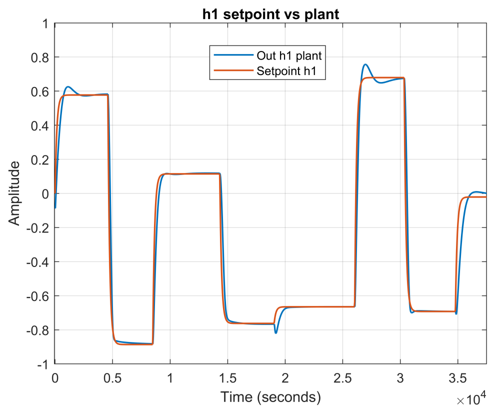
(a)



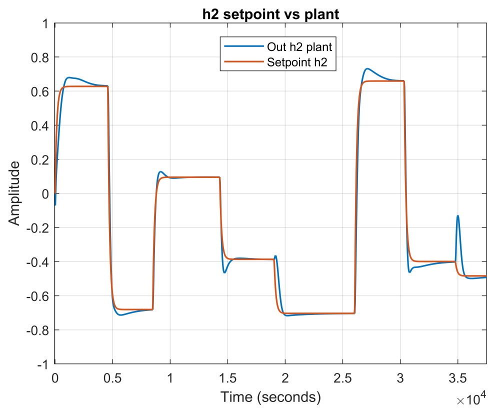
(b)

Figure 5.27. (a) Setpoint C vs output h_1 of the plant; (b) Setpoint C vs output h_2 of the plant.

- We consider now the setpoint D. No differences with respect to trajectory C are expected, indeed looking at Figures 5.28a and 5.28b, the results are very similar: almost zero offsets is achieved, and the overshoots are increased in terms of amplitude.



(a)



(b)

Figure 5.28. (a) Setpoint D vs output h_1 of the plant; (b) Setpoint D vs output h_2 of the plant.

Taking into account what has been observed, the results for h_1, h_2 are worse in terms of precision than the ones saw for h_1, h_3 system. The main cause is the lower accuracy of the controller training due to its higher complexity (no informations about upper tanks are available, therefore a direct correlation between q_a, q_b and h_1, h_2 is not easy to be found). However, doing a comparison between Sections 5.2.4 and 5.3.2, strong improvements have been registered in the tracking of the trajectories with the implementation of an integral action, showing how the control scheme well performs.

Let us point out that the obtained results of the IMC plus integral action scheme in terms of response time were expected: in continuous time the introduction of the integrator in the loop introduces a phase loss of -90°, which means worse settling time is achieved (even if we are considering discrete-time domain). The tuning of the integral gain μ can partially limit such problem as already mentioned, however we may observe an increase of the output oscillations in the tracking task (up to possible instability).

5.4 Conclusions

In Chapter 5 the Internal Model Control scheme has been presented, introducing its properties for a common SISO, linear system. Then, its main variations for MIMO and nonlinear systems were treated, focusing on the usage of NNs (more specifically, RNNs) in the scheme as controller and model.

In particular, in Section 5.2.1 the training of the RNN controller was highlighted, focusing on the procedure starting from a pre-trained RNN model representing the plant and the feasibility of the generated training dataset.

Also, the introduction of an integrator in the classic scheme has been proposed, discussing its zero steady-state offset property.

Four trajectories (two for $h_1,\ h_3$ system and two for $h_1,\ h_2$ system) have been introduced for the performance evaluations, and for each one a comparison between IMC scheme with and without integral action has been done in Sections 5.2 and 5.3 respectively, showing how the control architecture well performs in both these cases.

Chapter 6

Conclusions

In Chapter 1, an introduction to neural networks has been presented: first of all, an explanation of the common feed-forward NNs was provided, highlighting how the neurons work. Starting from the behaviour of a single neuron, and considering that the flow of the inputs in a FFNN is unidirectional, the corresponding final outputs of such network has been computed through the so-called feed-forward passage. To be able to perform operations with NNs, they have to be trained properly: in order to train a FFNN, the backpropagation algorithm has been considered. It takes usage of the gradient descent algorithm (or its main variation, the stochastic gradient descent) together with the feed-forward and backward passages: the final result is a fully trained FFNN. Also, it has been discussed how the main goal of the training operation is the minimization of a given cost function (e.g. the mean square error between the network predictions and the expected values) through the regulation of the NN's parameters, i.e. its weights and biases.

At this point, a second type of NNs has been introduced: the so-called recurrent neural networks. The main advantages discussed are their capability to memorise useful patterns observed in the analysed data, computing the outputs not only on the basis of the current inputs, but also on the basis of the older ones: indeed a correlation between the current output $y(t)$ of the RNN and the current and older inputs $u(t),\ u(t-1), .., \ u(0)$ (and outputs $y(t-1), .., \ y(0)$ too) has been shown. This led us to consider this kind of NNs instead of the FFNNs. In order to explain the training of RNNs, the unfolding operation together with the backpropagation through time were discussed, focusing on how the unfolding of RNNs leads essentially to FFNNs. Also, the truncated version of BPTT was introduced, exploiting its advantages in terms of computational cost due to the limited unfolding of the network (i.e. lower required average training time). At the end of the chapter, the optimizers to speed up the training operations were briefly shown, exploiting the main training problems that may arise (vanishing and exploding gradients). The gated units used to build RNNs were described, together with their main characteristics, highlighting their memorization capabilities.

In Chapter 2, the benchmark considered, i.e. the four-tanks system, has been introduced. Firstly, a mathematical model was proposed, allowing us to exploit its nonlinear properties. Then, its linearisation was discussed, focusing on the corresponding computed transfer matrix. Starting from this, the analysis of the steady-state conditions for the system showed how, considering a generic stationary operating point, there exists an unique input providing it only if the sum of the two opening position of the valves is different from 1, i.e. $\gamma_1 + \gamma_2 \neq 1$. Then, it has been showed how the positions of the three-way valves determine the location of a multivariable zero of the linearised system, leading to the stability or instability of the transfer matrix inverse. Also, the step responses and Bode diagrams of the linearised system were presented.

In Chapter 3 the main elements for RNN structure (with GRU or LSTM cells) has been discussed. In particular, we faced the choice of the cost function (i.e. the MSE washout), of the optimizer and discussed the initialization of the parameters. The latter may lead to problems in the training procedure, as discussed in the chapter, therefore it has to be performed carefully. In the second section, the design of experiments has been performed: the input signals q_a, q_b were generated from scratch as a sequence of steps (with random amplitudes and lengths), starting from a set of preliminary assumptions such as their maximum amplitude. In order to mimic the behaviour of real signals, white noise was added to the generated inputs over their whole time length. Also, the sampling of the signals has been treated: it is a key point, since it has been showed how the sampling period $T_c = 25\ s$ led to an attenuation of almost 40 dB. Let us point out that the normalization of the final collected dataset is mandatory in order to guarantee a correct (i.e. informative) training dataset: its discussion focused on the importance to normalize all the data collected over multiple experiments with the same corresponding values. This procedure allows us to ensure the usage of the same normalization scale for all the signals. Then, the batches for validation and training phases were created, considering a single experiment as a single batch.

In Chapter 4, the training of the NN model has been performed: more specifically the goal of this chapter was to provide a RNN such that its behaviour is as equal as possible to the one of the real plant. Due to the peculiarities of our benchmark, two types of system were considered: the first one with h_1, h_3 as outputs, while the second one with h_1, h_2 as outputs. Different networks were considered, changing every time the structure (in terms of adopted cells, number of units and number of layers). The obtained results showed how, for the system with outputs h_1 and h_3, the best selected architecture was the RNN with GRU cells, 5 units and only one layer. Indeed, this network guaranteed the best compromise between the required average training time and the accuracy of the predictions. Comparing LSTM cells with GRUs (considering the same number of units and layers), a lower precision of the outputs from the former has been observed, showing the higher ease of training of the GRU cells (they can be seen as an exemplification of LSTMs, since they are composed by only two gates instead of three). Also, some trials with a saturated dataset were performed in order to check if the trained NN was able to match the saturation zones: the results depicted the difficulty of the network to predict such areas, showing a not so high accuracy of the NN outputs (the increasing of the number of units helped in such problem, however it does not represent a final solution). Since the training operations were always performed with the BPTT approach, the truncated one was applied: huge improvements of the required training times were recorded with all the examined structures due to both the reduction of training samples and the limited unfolding of the RNNs. On the other hand, a lower accuracy of predictions was observed since considering subsequences of reduced length $\tilde{T}$ may prevent the NN to observe patterns in the data longer than $\tilde{T}$.

About the system with h_1 and h_2 as outputs, different outcomes were noticed: the increase of the number of both units and layers led to a remarkable improvement in terms of predictions accuracy. Thus, the best selected architecture was the one characterised by two layers and 15 units each in contrast with the previous system's trainings. The main reason to this outcome is that, disregarding all the informations about the upper tanks' levels, the training is much more tricky, i.e. the value of validation MSE will be higher if the number of units is not high enough.

In Chapter 5, the internal model control scheme has been presented and discussed, underlying its application to SISO, MIMO and nonlinear systems. In particular, we focused on the usage of RNN models and RNN controllers in the IMC scheme: a long discussion about the training procedure of the controller was faced, pointing out the theoretical and practical approaches. The recorded performances of such scheme were interesting: the tracking of the imposed setpoints (for both the systems mentioned in the previous chapter) was good, only some steady-state mismatches were observed; this is expected since the perfect controller cannot be obtained in practice. Starting from these results, the implementation of a discrete-time integral action in the regulator structure in order to remove the achieved offsets was discussed, highlighting the behaviour of the resulting IMC scheme under ideal hypothesis (like perfect modelling). For both the systems with $h_1,\ h_3$ and $h_1,\ h_2$ as outputs, the introduction of the integral term removed all the steady-state mismatches between the IMC outputs and the imposed setpoints. On the other hand, the effects of the additional integral term were reflected on the increase of the settling time required by the signals to follow the setpoints, and on the increase of their overshoots (in terms of amplitude). Since the RNN controller works only with input signal in the range [-1,1], an anti wind-up scheme was added to the integral term, acting on the controller input: this guarantees it is always in the correct range of values, so no undesired behaviours are observed.

In conclusion, this Thesis showed how the introduction of RNNs in IMC scheme may lead to very good results in terms of reference signals tracking. More specifically, their usage as model of the plant and as controller allowed us to notice only small steady-state offsets between the control scheme's outputs and the setpoints (local gain errors). The implementation of an integral action to solve this problem ensured zero errors at the steady-states, however a slowdown of the tracking procedure has been recorded. Thus, the final control scheme showed very good performances, allowing us to think about a possible usage of neural networks for the control of other types of systems. Clearly, some cons can be pointed out: the trainings of both model and controller require a lot of data to be available (if the training dataset size is too small, it will be non-informative), and the corresponding network structures are difficult to be immediately defined since there are no fixed rules at this purpose. Also, if the training set is not well-defined, for example it does not cover all the possible admissible values, the resulting performances of the NNs would be affected considerably.

Bibliography

[1] K. H. Johansson, "The quadruple-tank process: A multivariable laboratory process with an adjustable zero," *IEEE Transactions on control systems technology*, vol. 8, no. 3, pp. 456–465, 2000.

[2] T. L. Fine, *Feedforward neural network methodology.* Springer Science & Business Media, 2006.

[3] M. A. Nielsen, *Neural networks and deep learning.* Determination press San Francisco, CA, USA:, 2015, vol. 2018.

[4] F. Rosenblatt, *The perceptron, a perceiving and recognizing automaton Project Para.* Cornell Aeronautical Laboratory, 1957.

[5] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *ICML*, 2010.

[6] Q. V. Le, N. Jaitly, and G. E. Hinton, "A simple way to initialize recurrent networks of rectified linear units," *arXiv preprint arXiv:1504.00941*, 2015.

[7] D. Stathakis, "How many hidden layers and nodes?" *International Journal of Remote Sensing*, vol. 30, no. 8, pp. 2133–2147, 2009.

[8] A. Thomas, "An introduction to neural networks for beginners," Technical report in Adventures in Machine Learning, Tech. Rep., 2017.

[9] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016, http://www.deeplearningbook.org.

[10] H. B. Curry, "The method of steepest descent for non-linear minimization problems," *Quarterly of Applied Mathematics*, vol. 2, no. 3, pp. 258–261, 1944.

[11] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* O'Reilly Media, 2019.

[12] L. Bottou, "Stochastic gradient descent tricks," in *Neural networks: Tricks of the trade.* Springer, 2012, pp. 421–436.

[13] F. M. Bianchi, E. Maiorino, M. C. Kampffmeyer, A. Rizzi, and R. Jenssen, "An overview and comparative analysis of recurrent neural networks for short term load forecasting," *arXiv preprint arXiv:1705.04378*, 2017.

[14] P. J. Werbos, “Backpropagation through time: what it does and how to do it,” *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[15] C. Tallec and Y. Ollivier, “Unbiased online recurrent optimization,” *arXiv preprint arXiv:1702.05043*, 2017.

[16] I. Sutskever, *Training recurrent neural networks.* University of Toronto Toronto, Canada, 2013.

[17] B. T. Polyak, “Some methods of speeding up the convergence of iteration methods,” *USSR Computational Mathematics and Mathematical Physics*, vol. 4, no. 5, pp. 1–17, 1964.

[18] J. Duchi, E. Hazan, and Y. Singer, “Adaptive subgradient methods for online learning and stochastic optimization.” *Journal of machine learning research*, vol. 12, no. 7, 2011.

[19] T. Tieleman and G. Hinton, “Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude,” *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[20] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[21] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[22] K. He, X. Zhang, S. Ren, and J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification,” in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[23] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[24] E. Terzi, M. Farina, and R. Scattolini, “Model predictive control design for dynamical systems learned by long short-term memory networks,” *arXiv preprint arXiv:1910.04024*, 2019.

[25] D. Saccani and T. Bonetti, “Modeling and learning-based hybrid predictive control with recurrent neural networks of the cooling station of a large business center,” 2019.

[26] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using rnn encoder-decoder for statistical machine translation,” *arXiv preprint arXiv:1406.1078*, 2014.

[27] F. Bonassi, M. Farina, and R. Scattolini, “On the stability properties of gated recurrent units neural networks,” *arXiv preprint arXiv:2011.06806*, 2020.

[28] Z.-P. Jiang and Y. Wang, “Input-to-state stability for discrete-time nonlinear systems,” *Automatica*, vol. 37, no. 6, pp. 857–869, 2001.

[29] I. Alvarado, D. Limon, D. M. De La Peña, J. M. Maestre, M. Ridao, H. Scheu, W. Marquardt, R. Negenborn, B. De Schutter, F. Valencia *et al.*, "A comparative analysis of distributed mpc techniques applied to the hd-mpc four-tank benchmark," *Journal of Process Control*, vol. 21, no. 5, pp. 800–815, 2011.

[30] C. E. Garcia and M. Morari, "Internal model control. a unifying review and some new results," *Industrial & Engineering Chemistry Process Design and Development*, vol. 21, no. 2, pp. 308–323, 1982.

[31] B. A. Francis and W. M. Wonham, "The internal model principle of control theory," *Automatica*, vol. 12, no. 5, pp. 457–465, 1976.

[32] M. Morari, "Internal model control-theory and applications," *IFAC Proceedings Volumes*, vol. 16, no. 21, pp. 1–18, 1983.

[33] C. E. Garcia and M. Morari, "Internal model control. 2. design procedure for multivariable systems," *Industrial & Engineering Chemistry Process Design and Development*, vol. 24, no. 2, pp. 472–484, 1985.

[34] C. G. Economou, M. Morari, and B. O. Palsson, "Internal model control: Extension to nonlinear system," *Industrial & Engineering Chemistry Process Design and Development*, vol. 25, no. 2, pp. 403–411, 1986.

[35] I. Rivals and L. Personnaz, "Nonlinear internal model control using neural networks: Application to processes with delay and design issues," *IEEE transactions on neural networks*, vol. 11, no. 1, pp. 80–90, 2000.

Ringraziamenti

Innanzitutto, vorrei ringraziare il Professor Scattolini per l'opportunità datami, per la fiducia nei miei confronti e per le continue indicazioni che hanno permesso la realizzazione di questa tesi.

Un ringraziamento speciale va anche al Dott. Bonassi, il quale ha saputo guidarmi sin dall'inizio con pazienza e attenzione, dimostrandosi sempre disponibile. Grazie ai suoi suggerimenti, mi ha spronato a dare sempre il massimo, aiutandomi a migliorare continuamente.

Ringrazio la mia famiglia, per avermi sostenuto in questo lungo viaggio. Mia mamma, per avermi insegnato a non accontentarmi mai e a credere sempre in me stesso. Mio padre, per avermi fatto capire il valore del sacrificio e del duro lavoro. Mio fratello, per essere stato un punto di riferimento negli anni e un esempio da seguire. Mia nonna, per tutto l'affetto e gli insegnamenti che mi ha dato e che porterò sempre nel cuore.

Un grazie va anche ai miei compagni di corso, che con me hanno condiviso questi cinque lunghi anni, regalandomi tanti momenti che ricorderò per sempre. In particolare un grazie va ad Enea, grande amico e grande persona ancor prima che collega.

Ringrazio infine i miei amici per essere stati sempre presenti, per avermi sempre sostenuto e donato attimi di felicità e spensieratezza che mi hanno permesso di raggiungere questo grande traguardo.