POLITECNICO
DI MILANO

Politecnico di Milano
Dipartimento di Matematica "F. Brioschi"

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

École Polytechnique Fédérale de Lausanne
Chair of Computational Mathematics and Simulation Science

Master of Science in
Computational Science and Engineering

Reduced order modeling of nonlinear problems using neural networks

Candidate
Stefano Ubbiali

Supervisors
Prof. Jan S. Hesthaven
Prof. Matteo Zunino

A.Y. 2016 - 2017

⚠ • non tutte le eq. devono avere il numero: solo quelle che "citi": se fai \eqref allora ok, se no viene numero!

• quando introduci una sigla, dopo usala ☺

• "on the other hand" lo puoi usare solo se prima metti "on one hand"

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction to neural networks

Let us start this dissertation by introducing the key components of an artificial neural network and discussing the way it can be configured for a specific application. Please note that this chapter is not meant to provide a comprehensive overview on neural networks, rather to investigate some aspects and concepts functional to the following chapters. For further reading, we refer the reader to, e.g., [14, 16, 27], from which we retrieved many of the informations provided in this chapter.

Throughout this work we confine the attention to the most-spread neural network paradigm - the *feedforward* neural network, presented in Section 1.2.2 - employing the well-known *backpropagation of error* as learning rule, derived in Section 1.2.3. Actually, in the numerical experiments we carried out and whose results will be discussed in Chapter 3, we mainly refer to a variant of backpropagation : the Levenberg-Marquardt algorithm.

Before moving to the description of technical neural networks, let us provide a brief excursus on their biological counterparts. The goal is to highlight the basic features of the human nervous system, focusing on the working principles of neurons and the way informations are processed, thus to extract the key concepts which should be taken over into a mathematical, simplified representation.

1.1 Biological motivation

The information processing system of a vertebrate can coarsely be divided into the *central nervous system* (CNS) and the *peripheral nervous system* (PNS). The former consists of the *brain* and the *spinal cord*, while the latter mainly comprises the *nerves*, which transmit informations from all other parts of the body to the CNS (*sensory nerves*) and viceversa (*motor nerves*). When an output stimulus hits the sensory cells of an organ sense, these generate an electric signal, called *sensory signal*, which is transfered to the central nervous system via the *sensory nerves*. Within the CNS, informations are stored and managed to provide the muscles with a suitable *motor signal*, broadcast through the *motor nerves* and finally converted by the effectors into a system output [16].

Hence, both the central and peripheral nervous system are directly involved in the information processing workflow. At the cellular level, this is accomplished through a huge amount of modified cells called *neurons*. These processing elements continuosly communicate each other by means of electric signals, traveling through a thick net of

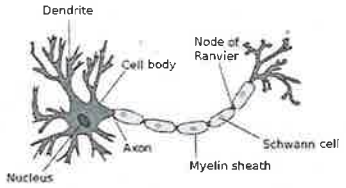

Figure 1.1. Simplified representation of a biological neuron, with the components discussed in the text; retrieved from [27].

connections. For instance, in a human being each neuron is linked in average with $10^3 - 10^4$ other neurons. As detailed in the next paragraph, a neuron is characterized by a rather simple structure, specifically designed to rapidly collect input signals and generate an output pulse whenever the accumulated incoming signal exceeds a threshold : the *action potential*. In other terms, a neuron acts as a switch, establishing a typically nonlinear input-output mapping [27].

From a simplifying perspective, a neuron consists of three main components: the *dendrites*, the *nucleus* or *soma*, and the *axon*. Dendrites are tree-like networks of nerve fibers receiving input signals from many sources and conveying them directly to the nucleus of the neuron. Here, input signals are accumulated and thresholded, as mentioned before. The possible output pulse is then broadcast to the cells contiguous the neuron through the axon - a unique, slender fiber constituing an extension of the soma and splitting in many branches at the opposite extremity [44]. To ease the electrical conduction of the signal, the axon is isolated through a myelin sheath which consists of Schwann cells (in the PNS) or oligodendrocytes (in the CNS). However this insulating film is not continuous, rather presents gaps at regular intervals called *nodes of Ranvier*, which allow the signal to be conducted in a saltatory way.

The signal coming from the axon of another neuron or from another cell is transfered to the dendrites of a neuron through a particular connection called *synapsis*[1]. A synaptic may be either electrical or chemical. In the former, the presynaptic side, i.e., the sender axon, and the postsynapic side, i.e., the receiver dendrite, are directly in contact, so that the potential can simply travel by electrical conduction. Conversely, a chemical synapsis consists of a synaptic *cleft*, physically separating the presynaptic side from the postsynaptic side. Then, to let the action potential reach the postsynaptic side, at the presynaptic side the electrical pulse is converted into a chemical signal. This is accomplished by releasing some chemical substances called *neurotransmitters*. These neurotransmitters then cross the cleft and bind to the receptors dislocated onto the membrane of the postsynaptic side, where the chemical signal is re-converted into an electrical potential. However, neurotransmitters do not simply broadcast the action potential. Indeed, we can distinguish between excitatory and inhibitory neurotransmitters, respectively amplifying or modulating

[1] For the sake of completeness, we mention that there exist synapses directly connecting the axon of the sender neuron with either the soma or the axon of the receiver. Actually, a synapsis may also connect the axon of a neuron with the dendrite or soma of the same neuron (autosynapsis). However, for our purposes we can confine the attention to the axon-dendrite synapsis.

the signal. Hence, the pulse outgoing a neuron is preprocessed within the synapsis before reaching the target cell. In other terms, a neuron gets in input many *weighted* signals, which should then be collected.

Different studies have unveiled the tight correlation between the synapses the neurons establish among each other and the tasks a neural network can address [14]. That is, the set of interneuron connection strengths represent the information storage, i.e., the knowledge, of a neural network [27]. Knowledge is acquired through a *learning* or *training* process, entailing adjustments at the synaptic level to adapt to environmental situations. The adjustments may not only involve the modification of existing synapses, but also the creation of new synaptic connections. Hence, the nervous system is a distributed memory machine whose evolutionary structure is shaped by experience.

As mentioned above, a biological neural network acquaints itself with problems of a specific class through a learning procedure. During the learning, the network is exposed to a collection of situations, giving it the possiblity to derive a set of tools which will let it provide reasonable solutions in similar circumstances. In other terms, the cognitive system should be able to *generalize*. Furthermore, after a successful training, a neural network should also show a discrete level of *fault tolerance* against external errors, e.g. noisy inputs. It worths notice here that the nervous system is also naturally fault tolerant against *internal* errors. Indeed, in case a neuron or a (relatively small) group of neurons got damaged or died, the other processing nodes would take care of its tasks, so that the overall cognitive capabilities would be only slightly affected [27].

1.2 Artificial neural networks

Inspired by the biological information processing system discussed so far, an artificial neural network (ANN), usually simply referred to as "neural network", is a computational model capable to learn from observational data, i.e., by example, thus providing an alternative to the algorithmic programming paradigm [34]. Exactly as its original counterpart, it consists of a collection of processing units, called (artificial) neurons, and directed weighted synaptic connections between the neurons themselves. Data travel among neurons through the connections, following the direction imposed by the synapses themselves. Hence, an artificial neural network is an *oriented graph* to all intents and purposes, with the neurons as *nodes* and the synapses as oriented *edges*, whose weights are adjusted by means of a *training* process to configure the network for a specific application [44].

Formally, a neural network could be defined as follows [27].

Definition 1.1 (Neural network). *A* neural network *is a sorted triple* $(\mathcal{N}, \mathcal{V}, w)$*, where* $\mathcal{N}$ *is the set of* neurons*, with cardinality* $|\mathcal{N}|$*,* $\mathcal{V} = \{(i,j), 1 \le i, j \le |\mathcal{N}|\}$ *is the set of* connections*, with* (i,j) *denoting the oriented connection linking the sending neuron* i *with the target neuron* j*, and* $w : \mathcal{V} \to \mathbb{R}$ *is the* weight function*, defining the weight* $w_{i,j} = w((i,j))$ *of the connection* (i,j)*. A weight may be either positive or negative, making the underlying connection either excitatory or inhibitory, respectively. By convention,* $w_{i,j} = 0$ *means that neurons* i *and* j *are not directly connected.*

In the following, we dive deeper into the structure and training of a neural network, starting by detailing the structure of an artificial neuron.

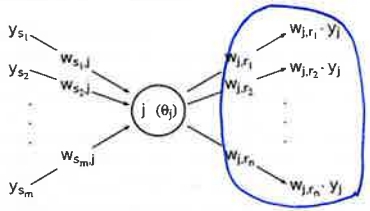

Figure 1.2. Visualization of the generic j-th neuron of an artificial neural network. The neuron accumulates the weighted inputs $\{w_{s_1,j}\, y_{s_1}, \dots, w_{s_m,j}\, y_{s_m}\}$ coming from the sending neurons $s_1, \dots, s_m$, and fires y_j, sent to the target neurons $\{r_1, \dots, r_n\}$ through the synapsis $\{w_{j,r_1}, \dots, w_{j,r_n}\}$. The neuron threshold θ_j is reported within its body.

1.2.1 Neuronal model

As its name may suggest, an artificial neuron represents a simplified model of a biological neuron, retaining its main features discussed in Section 1.1. To introduce the components of the model, let us consider the neuron j represented in Figure 1.2. Suppose that it is connected with m sending neurons $s_1, \dots, s_m$, and n receiving (target) neurons $r_1, \dots, r_n$. Denoting by $y_\Omega(t) \in \mathbb{R}$ the scalar output fired by a generic neuron Ω at time t, neuron j gets the weighted inputs $w_{s_k,j}\, y_{s_k}(t)$, $k = 1, \dots, m$, at time t, and sends out the output $y_j(t + \Delta t)$ to the target neurons $r_1, \dots, r_n$ at time $t + \Delta t$. Please note that in the context of artificial neural networks the time is discretized by introducing the timestep Δt. This is clearly not plausible from a biological viewpoint; on the other hand, it dramatically eases the implementation. In the following, we will avoid to specify the dependence on time unless strictly necessary, thus to lighten the notation.

An artificial neuron j is completely characterized by three functions: the propagation function, the activation function, and the output function. These will be defined and detailed hereunder in the same order they get involved in the data flow.

Propagation function. The propagation function f_{prop} converts the vectorial input $\mathbf{p} = [y_{s_1}, \dots, y_{s_m}]^T \in \mathbb{R}^m$ into a scalar u_j often called *net input*, i.e.,

$$u_j = f_{prop}(w_{s_1,j}, \dots, w_{s_m,j}, y_{s_1}, \dots, y_{s_m}). \tag{1.1}$$

A common choice for f_{prop} (used also in this work) is the weighted sum, adding up the scalar inputs multiplied by their respective weights:

$$f_{prop}(w_{s_1,j}, \dots, w_{s_m,j}, y_{s_1}, \dots, y_{s_m}) = \sum_{k=1}^{m} w_{s_k,j}\, y_{s_k}. \tag{1.2}$$

The function (1.2) provides a simple, yet effective way of modeling the accumulation of different input electric signals within a biological neuron; this motivates its popularity.

Activation or transfer function. At each timestep, the *activation state* a_j, often shortly referred to as *activation*, quantifies at which extent neuron j is currently active or excited. It results from the activation function f_{act}, which combines the net input u_j with a threshold $\theta_j \in \mathbb{R}$ [27]:

$$a_j = f_{act}(u_j; \theta_j) = f_{act}\Big(\sum_{k=1}^{m} w_{s_k,j}\, y_{s_k}; \theta_j\Big), \tag{1.3}$$

where we have employed the weighted sum (1.2) as propagation function. From a biological perspective, the threshold θ_j is the analogous of the action potential mentioned in Section 1.1. Mathematically, it represents the point where the absolute value (f'_{act}) of the derivative of the activation function is maximum. Then, the activation function reacts particularly sensitive when the net input u_j hits the threshold value θ_j [27]. Furthermore, noting that θ_j is a parameter of the network, one may like to adapt it through a training process, exactly as can be done for the synaptic weights, as we shall see in Section 1.2.3. However, θ_j is currently incorporated in the activation function, making its runtime access somehow cumbersome. This is typically overcome by introducing a *bias neuron* in the network. A bias neuron is a continuously firing neuron, with constant output $y_b = 1$, which gets directly connected with neuron j, assigning the *bias weight* $w_{b,j} = -\theta_j$ to the connection. As can be deduced by Figure 1.3, θ_j is now treated as a synaptic weight, while the neuron threshold is set to zero. Moreover, the net input becomes

$$u_j = \sum_{k=1}^{m} w_{s_k,j}\, y_{s_k} - \theta_j, \tag{1.4}$$

i.e., the threshold is now included in the propagation function rather than in the activation function, which we can now express in the form

$$a_j = f_{act}\Big(\sum_{k=1}^{m} w_{s_k,j}\, y_{s_k} - \theta_j\Big). \tag{1.5}$$

Let us point out that this trick can be clearly applied to all neurons in the network which are characterized by a non-vanishing threshold: just connect the neuron with the bias, weighting the connection by the opposite of the threshold value. However, for ease of illustration, in the following we shall avoid to include the bias neuron in any graphical representation of a neural network.

Conversely to the propagation function, there exist various choices for the activation function, as the Heaviside or binary function, which assumes only 0 or 1, according to whether the argument is negative or positive, respectively:

$$f_{act}(v) = \begin{cases} 0, & \text{if } v < 0, \\ 1, & \text{if } v \geq 0. \end{cases} \tag{1.6}$$

Neurons characterized by such an activation function are usually named McCulloch-Pitts neurons, after the seminal work of McCulloch and Pitts [16], and their employment is usually limited to single-layer perceptrons implementing boolean logic (see Section 1.2.2). In addition, note that (1.6) is discontinuous, with a vanishing derivative everywhere except that in the origin, thus not admissible for the backpropagation training algorithm presented in Section 1.2.3.

Among continuous activation maps, sigmoid functions have been widely used for the realization of artificial neural networks due to their graceful combination of linear and nonlinear behaviour [16]. Sigmoid functions are s-shaped and monotically increasing, and assume values in a bounded interval, typically $[0, 1]$, as the logistic function,

$$f_{act}(v) = \frac{1}{1 + e^{-v/T}} \quad \text{with } T > 0, \tag{1.7}$$

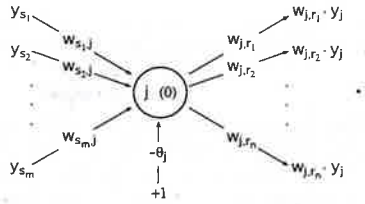

Figure 1.3. Visualization of the generic j-th neuron of an artificial neural network. The neuron accumulates the weighted inputs $\{w_{s_1,j}\, y_{s_1}, \ldots, w_{s_m,j}\, y_{s_m}, -\theta_j\}$ coming from the sending neurons $s_1, \ldots, s_m$, b, respectively, with b the bias neuron. The neuron output y_j is then conveyed towards the target neurons $\{r_1, \ldots, r_n\}$ through the synapsis $\{w_{j,r_1}, \ldots, w_{j,r_n}\}$. Observe that, conversely to the model offered in Figure 1.2, the neuron threshold is now set to 0.

or $[-1, 1]$, as the hyperbolic tangent,

$$f_{act}(v) = \frac{e^v - e^{-v}}{e^v + e^{-v}}. \tag{1.8}$$

Both functions are displayed in Figure 1.4. Note that the logistic function resembles the Heaviside function as T decreases.

Output function. Finally, the output function f_{out} is in charge of calculating the scalar *output* $y_j \in \mathbb{R}$ based on the activation state a_j of the neuron:

$$y_j = f_{out}(a_j) \tag{1.9}$$

Typically, f_{out} is the identity function, so that activation and output of a neuron coincides, i.e., $y_j = f_{out}(a_j) = a_j$. Let us point out that while the input $\mathbf{p} = [y_{s_1}, \ldots, y_{s_m}]^T \in \mathbb{R}^m$ of the neuron is generally vectorial, i.e., $m > 1$, the output is scalar. The output y_j could then either be sent to other neurons, included the outputting neuron itself (autosynapsis), or constitute a component of the overall output vector of the network, as for the neurons in the output layer of a feedforward neural network (see Section 1.2.2).

It worths mention here that, as the activation function, also the output function is usually *globally* defined, i.e., all neurons in the network (or at least a group of neurons) are equipped with the same output function.

The neural model presented so far actually refers to the so called *computing* neuron, i.e., a neuron processing input informations to provide a response. However, in a neural network one may also distinguish *source* neurons, supplying the network with the respective components of the activation pattern (input vector) [16]. The role of source neurons will be clearer in the next section, where we will introduce the multilayer feedforward neural network. Here, we just mention that such a neuron receives a scalar and unweighted input, which is simply forwarded to the connected neurons, no computation is performed.

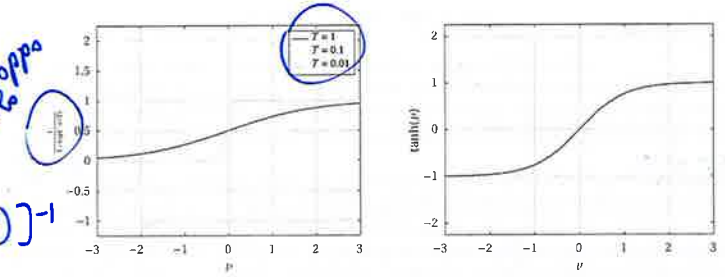

Figure 1.4. Left: logistic function (1.7) for three values of the parameter T; note that as T decreases, the logistic function resembles the Heaviside function. Right: hyperbolic tangent.

1.2.2 Network topologies: the feedforward neural network

The way neurons are interconnected within a network defines the *topology* of the network itself, i.e., its design. In literature, many network architectures have been proposed, sometimes tailored to a specific application or task. In this section, we expand our investigation for the two most common network topologies: the feedforward and the recurrent neural network.

Feedforward neural network. In a feedforward neural network, also called *perceptron* [42], neurons are arranged into *layers*, with one *input layer* of M_I source neurons, K *hidden layers*, each one consisting of H_k computing neurons, $k = 1, \dots, K$, and an *output layer* of M_O computing neurons. As characteristic property, neurons in a layer can only be connected with neurons in the next layer towards the output layer. Then, an *activation pattern* $\mathbf{p} \in \mathbb{R}^{M_I}$, supplied to the network through the source nodes in the first layer, provides the input signal for the neurons in the first hidden layer. For each hidden layer, its output signal gives the input pattern for the following layer. In this way, informations travel towards the last layer of the network, i.e., the output layer, whose outputs constitute the components of the overall output $\mathbf{q} \in \mathbb{R}^{M_O}$ of the network, response to the input pattern $\mathbf{p}$[2]. Hence, a feedforward network establishes a map between the *input space* $\mathbb{R}^{M_I}$ and the *output space* $\mathbb{R}^{M_O}$. This makes this network architecture particularly suitable for, e.g., classification and continuous function approximation.

Feedforward networks can be classified according to the number of hidden neurons they present, or, equivalently, the number of layers of trainable weights. Single-layer perceptrons (SLPs) just consist of the input and output layer, no hidden layers. Note that the layer which the name refers to is the output layer, the input layer is not accounted for since it does not perform any calculation. Despite their quite simple structure, the range of application of single-layer perceptrons is rather limited. Indeed, consider a binary input vector, supplied to an SLP provide with a unique output neuron, equipped with a binary

[2] Please note that while the output of a single neuron is denoted with the letter y, we use the letter $\mathbf{q}$ (bolded) to indicate the overall output of the network. Clearly, for the j-th output neuron the output y_j coincides with the correspondent entry of $\mathbf{q}$, i.e., $q_j = y_j$, $j = 1, \dots, M_O$.

activation function. The network acts then as a classifier, splitting the input space, i.e., the unit hypercube by means of a hyperplane. Therefore, only *linearly separable* data can be properly represented [27]. On the other hand, the share of linearly separable sets decreases as the space dimension increases, making single-layer perceptrons seldom attractive.

Conversely, multi-layer perceptrons (MLPs), providing at least one hidden layer, are universal function approximators, as stated by Cybenko [8, 9]. In detail:

(i) a multi-layer perceptron with one layer of *hidden neurons* and differentiable activation functions can approximate any *continuous* function [9];

(ii) a multi-layer perceptron with two layers of *hidden neurons* and differentiable activation functions can approximate *any function* [8].

Therefore, in many practical applications there is no reason to employ MLPs with more than two hidden layers. Considering again the binary classifier discussed above, one can represent any convex polygon by adding one hidden layer, and any arbitrary set by adding two hidden layers; further, increasing the number of hidden neurons would not improve the representation capability of the network. However, we should point out that (i) and (ii) do not give any practical advice on both the number of hidden neurons and the number of samples required to teach the network.

An instance of a three-layer (i.e., two hidden layer plus the output layer) feedforward network is offered in Figure 1.5. In this case, we have $M_I = 3$ input neurons (denoted with the letter i), $H_1 = H_2 = 6$ hidden neurons (letter h), and $M_O = 4$ output neurons (letter o). In particular, we point out that it represents an instance of a *completely linked* perceptron, since each neuron is directly connected with all neurons in the following layer.

Finally, let us just mention that, although we have previously stated that in a feedforward neural network a synapses can only connect pairs of neurons in contiguous layers, recent years have seen the development of different variants. For instance, *shortcut connections* skip one or more layers, while *lateral connections* takes place between neurons within the same layer. However, throughout this work we shall be faithful to the original definition of the perceptron.

Recurrent neural network. In recurrent networks any neuron can bind with any other neuron, but autosynapses are forbidden, i.e., the output of a neuron can be input into the same neuron at the next time step. If each neuron is connected with all other neurons, then the network is said *completely linked*. As a consequence, one can not distinguish neither input from output neurons any more: neurons are all equivalent. Then, the input of the network is represented by the initial *network state*, which is the set of activation states for all neurons in the network. Similarly, the overall network output is given by the final network state. So, communication between a recurrent neural network and the surrounding environment takes place through the states of the neurons. Examples of recurrent networks are the Hopfield networks [21], inspired by the behaviour of electrically charged particles within a magnetic field, and the self-organizing maps by Kohonen [26], highly suitable for cluster detection.

As mentioned in the introductory chapter, in this work we refer to neural networks for the approximation of the unknown map ϕ between the parameters $\boldsymbol{\mu}$ of a parametrized partial differential equation, and the coefficients $\boldsymbol{\alpha}$ of the corresponding reduced basis

solution. To accomplish this task, we rely on a collection of samples $\{\boldsymbol{\mu}_i, \boldsymbol{\alpha}_i\}_{i=1}^{N_{tr}}$ generated through a high-fidelity model. Although a detailed explanation will be provided later in Chapter 2, what is worth notice here is that we are concerned with a *continuous function approximation* problem. Then, motivated by what previously said in the section, a multilayer feedforward neural network turns out as the most suitable network architecture for our purposes.

We are now left to investigate the way the weights of a perceptron can be *trained* to meet our purposes.

1.2.3 Training a multilayer feedforward neural network

As widely highlighted so far, the primary characteristic of a neural network lies in the capability to *learn* from the environment, storing the acquired knowledge through the network internal parameters, i.e., the synaptic and bias weights. Learning is accomplished through a training process, during which the network is exposed to a collection of examples, called *training patterns*. According to some performance measure, the weights are then adjusted by means of a well-defined set of rules. Therefore, a learning procedure is an *algorithm*, typically iterative, modifying the neural network parameters to make it knowledgeable of the specific environment it operates in [16]. Specifically, this entails that after a successfull training, the neural network has to provide reasonable responses for unknown problems of the same class the network was acquainted with during training. This property is known as *generalization* [27].

Training algorithms can be firstly classified based on the nature of the training set, i.e., the set of training patterns. We can then distinguish three *learning paradigms*.

Supervised learning. The training set consists of a collection of *input patterns* (i.e., input vectors) $\{\mathbf{p}_i\}_{i=1}^{N_{tr}}$, and corresponding desired responses $\{\mathbf{t}_i\}_{i=1}^{N_{tr}}$, called *teaching inputs*. Then, $\mathbf{t}_i$ is the output the neural network should desirably provide when it gets fed with $\mathbf{p}_i$. As we shall see, any training procedure aims to minimize (in some appropriate norm) the discrepancy between the *desired* output $\mathbf{t}_i$ and the *actual* output $\mathbf{q}_i$ given by the network as response to $\mathbf{p}_i$.

Unsupervised learning. Although supervised learning is a simple and intuitive paradigm, there exist many tasks which require a different approach to be tackled. Consider, for instance a cluster detection problem. Due to lack of prior knowledge, rather than telling the neural network how it should behave in certain situations, one would like the neurons to *independently* identify rules to group items. Therefore, a training pattern just consist of an input pattern. Since no desired output is provided, such a pattern is referred to as *unlabeled*, as opposed to the *labeled* examples involved in the supervised learning paradigm.

Reinforcement learning. Reinforcement learning is the most plausible paradigm from a biological viewpoint. After the completion of a series of input patterns, the neural network is supplied with a boolean or a real saying whether the network is wrong or right. In the former case, the *feedback* or *reward* may also indicate to which extent the network is wrong [27]. Conversely to the supervised and unsupervised paradigms, reinforcement learning focuses on finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge). Hence, this paradigm particularly suits problems involving a trade-off

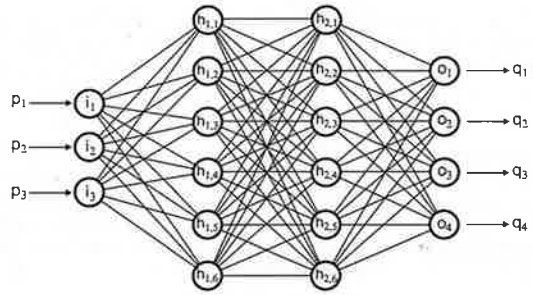

Figure 1.5. A three-layer feedforward neural network, with 3 input neurons, two hidden layers each one consisting of 6 neurons, and 4 output neurons. Within each connection, information flows from left to right.

between a long-term and a short-term reward [24].

Clearly, the choice of the learning paradigm is task-dependent. In particular, function approximation (i.e., what we are interested in) perfectly fits the *supervised learning* paradigm. Indeed, consider the nonlinear unknown function $\boldsymbol{f}$,

$$\boldsymbol{f} : \mathbb{R}^{M_I} \to \mathbb{R}^{M_O}$$
$$\mathbf{x} \mapsto \mathbf{y} = \boldsymbol{f}(\mathbf{x}),$$

and a set of labeled examples $\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^{N_{tr}}$. The task implies to approximate $\boldsymbol{f}$ over a domain $V \subset \mathbb{R}^{M_I}$ up to a user-defined tolerance ϵ, i.e.,

$$||\boldsymbol{F}(\mathbf{x}) - \boldsymbol{f}(\mathbf{x})|| < \epsilon \quad \forall \mathbf{x} \in V,$$

where $\mathbf{F} : \mathbb{R}^{M_I} \to \mathbb{R}^{M_O}$ is the actual input-output map established by the neural network, and $||\cdot||$ is some suitable norm on $\mathbb{R}^{M_O}$. Surely, as necessary condition the neural system must satisfy to well approximate $\boldsymbol{f}$ for each input in the domain V, the system should provide accurate predictions for the input patterns, i.e.,

$$\boldsymbol{F}(\mathbf{x}_i) \approx \mathbf{y}_i, \quad \forall i = 1, \dots, N_{tr}.$$

Then, we could train the network through a supervised learning algorithm, employing $\{\mathbf{x}_i\}_{i=1}^{N_{tr}}$ as input patterns and $\{\mathbf{y}_i\}_{i=1}^{N_{tr}}$ as teaching inputs. That is,

$$\mathbf{p}_i = \mathbf{x}_i \text{ and } \mathbf{t}_i = \mathbf{y}_i, \quad \forall i = 1, \dots, N_{tr}.$$

As defined in the first part of the section, a training algorithm involves:

(a) a set of well-defined rules to modify the synaptic and bias weights;

(b) a *performance function*, quantifying the current level of knowledge of the surrounding environment.

Regarding (a), a plethora of weight updating techniques have been proposed in literature, sometimes tailored on specific applications. Nevertheless, most of them relies on the well-known Hebbian rule, proposed by Donal O. Hebb in 1949 [17]. Inspired by neurobiological considerations, the rule can be stated in a two-steps fashion [16]:

(i) if two neurons on either side of a synapse (connection) are activated simultaneously (i.e., synchronously) then the strength of that synapse is selectively increased;

(ii) if two neurons on either side of a synapse are activated asynchronously, then that synapse is selectively weakened (or eliminated).

Actually, (ii) was not included in the original statement; nevertheless, it provides a natural extension to [16]. In mathematical terms, consider the synapsis between a sending neuron i and a target neuron j. Then, at the t-th iteration (also called *epoch*) of the training procedure, the weight $w_{i,j}(t)$ of the connection (i,j) is modified by the quantity

$$\Delta w_{i,j}(t) \sim \eta\, y_i(t)\, a_j(t), \tag{1.10}$$

where $\eta > 0$ is the *learning rate*, and we have exploited the fact that $y_i = a_i$ since the output function is usually represented as the identity. Hence, at the subsequent iteration $t+1$ the synaptic weight is simply given by

$$w_{i,j}(t+1) = w_{i,j}(t) + \Delta w_{i,j}(t). \tag{1.11}$$

Many of the supervised learning rules proposed in literature, included the backpropagation of error (derived late), can be recast in the following form, which turns out as a generalization of the Hebbian rule (1.10) [27]:

$$\Delta w_{i,j} = \eta\, h(y_i, w_{i,j})\, g(a_j, t_j), \tag{1.12}$$

with the functions h and g dependent on the specific learning rule, and t_j the j-th component of the teaching input $\mathbf{t}$. Note that all variables involved in (1.12) are supposed to be evaluated at time t, i.e., the correction $\Delta w_{i,j}$ is time-dependent. In addition, let us remark that (1.12) represents a *local* and *interactive* mechanism, since it involves both activity of neurons at the end-points of the synapse.

The second ingredient required to define a training algorithm is the performance or error function E. This function somehow measures the discrepancy between the neural network knowledge of the surrounding environment, and the actual state of the environment itself. In other terms, the larger the performance function, the farer the neural network representation of the world is from the actual reality, i.e. the farer we are from the application goal. Therefore, every learning rule aims to *minimize* the performance E as much as possible. For this purpose, E should be intended as a scalar function of the free parameters, i.e., the weights of the network, namely

$$E = E(\mathbf{w}) \in \mathbb{R}. \tag{1.13}$$

Recalling the notation and assumptions introduced in Definiton 1.1, here $\mathbf{w} \in \mathbb{R}^{|\mathcal{V}|}$ is a vector collecting the weights $\{w_{i,j} = w((i,j))\}_{(i,j)\in\mathcal{V}}$, with $\mathcal{V}$ the set of admissible connections in

the network[3]. Thus, Equation (1.13) implies that the point over the error surface reached at the end of a successful training process provides the *optimal* configuration $\mathbf{w}_{opt}$ for the network.

The steps a generic supervised training procedure should pursue are listed by Algorithms 1.1 and 1.2 for online and offline learning, respectively. *Online* learning means that the weights are updated after the exposition of the network to each training pattern. In other terms, each epoch involves only one training pattern. Conversely, in an *offline* learning procedure the modifications are based on the entire training set, i.e., the weights are adjusted only after the network has been fed with all input patterns and the corresponding errors have been accumulated. Therefore, we should distinguish between the *specific error* $E_{\mathbf{p}}(\mathbf{w})$, specific to the activation pattern $\mathbf{p}$, and the *total error* $E(\mathbf{w})$ accounting for all specific errors, namely

$$E(\mathbf{w}) = \sum_{\mathbf{p}\in P} E_{\mathbf{p}}(\mathbf{w}), \tag{1.14}$$

with P the training set. For instance, we could think of the specific error as the Mean Squared Error (MSE):

$$E_{\mathbf{p}}(\mathbf{w}) = \frac{1}{M_O} \sum_{j=1}^{M_O} \left(t_{\mathbf{p},j} - q_{\mathbf{p},j}\right)^2, \tag{1.15}$$

where we have added the subscript $\mathbf{p}$ to the components of the teaching input $\mathbf{t}$ and the actual output $\mathbf{q}$ to remark they refer to the input pattern $\mathbf{p}$. Accordingly, we could provide the following definition for the accumulated MSE:

$$E(\mathbf{w}) = \sum_{\mathbf{p}\in P} E_{\mathbf{p}}(\mathbf{w}) = \sum_{\mathbf{p}\in P} \frac{1}{M_O} \sum_{j=1}^{M_O} \left(t_{\mathbf{p},j} - q_{\mathbf{p},j}\right)^2. \tag{1.16}$$

So far, we have not yet discussed how the update $\Delta w_{i,j}$ for the weight $w_{i,j}$ can be practically computed at each iteration. Namely, we have still to rigourously define the functions h and g involved in the generalized Hebbian rule (1.12). To this end, let us recall that any given operation carried out by the neural network can be thought as a point over the error surface $E(\mathbf{w})$. Therefore, to increase the performance of the network, we need to iteratively move toward a (local) minimum of the surface [16]. For this purpose, we may employ a *steepest descent* technique, thus following the direction of the anti-gradient

$$\Delta w_{i,j} = -\eta \frac{\partial E(\mathbf{w})}{\partial w_{i,j}}, \tag{1.17}$$

where $\eta > 0$ being the learning rate, modulating the size of the step. Its role will be clearer later in the chapter. Among the others, the *backpropagation of error* [33] is surely the most-known supervised, gradient-based training procedure for a multi-layer feedforward neural network whose neurons are equipped with a *semi-linear*, i.e., continuous and differentiable, activation function[4]. The derivation of the backpropagation algorithm is provided in the following.

[3] Please note that while in Definiton 1.1 $\mathcal{V}$ denoted the set of all *possible* connections, here $\mathcal{V}$ disregards those connections which are not consistent with the network topology in use. For instance, a feedforward neural network can not be endowed with connections oriented towards the input layer, then such connections will not be included in $\mathcal{V}$. In this way, we reduce the size of $\mathbf{w}$ - which is a practical advantage.

[4] Therefore, backpropagation cannot apply with a binary activation function.

Algorithm 1.1 Backbone of any supervised online learning algorithm; the procedure ends when all training patterns yield an error below a defined threshold.

Input: neural network $(\mathcal{N}, \mathcal{V}, \mathbf{w}_0)$, training set $P = \{\mathbf{p}_i, \mathbf{t}_i\}_{i=1}^{N_{tr}}$,
metric $d(\cdot,\cdot) : \mathbb{R}^{Mo} \times \mathbb{R}^{Mo} \to \mathbb{R}$, tolerance ϵ, maximum number of epochs T
Output: trained neural network $(\mathcal{N}, \mathcal{V}, \mathbf{w}_{opt})$

$t = 0$, $i = 1$, $k = 0$
$\mathbf{w}(0) = \mathbf{w}_0$
while $t < T$ and $k < N$ **do**
  evaluate output vector $\mathbf{y}_{\mathbf{p}_i}(t)$, correspondent to input pattern $\mathbf{p}_i$
  $E_{\mathbf{p}_i}(\mathbf{w}(t)) = d(\mathbf{y}_{\mathbf{p}_i}(t), \mathbf{t}_i)$
  if $E_{\mathbf{p}_i}(\mathbf{w}(t)) < \epsilon$ **then**
    $k \leftarrow k + 1$
  else
    $k = 0$
    compute weight update $\Delta\mathbf{w}(t)$ based on $E_{\mathbf{p}_i}(\mathbf{w}(t))$
    $\mathbf{w}(t+1) = \mathbf{w}(t) + \Delta\mathbf{w}(t)$
  end if
  $t \leftarrow t + 1$, $i \leftarrow i \pmod N + 1$
end while
$\mathbf{w}_{opt} = \mathbf{w}(t)$

Backpropagation of error

Let us consider the generic synapse (i, j) of a multi-layer feedforward neural network, connecting the *predecessor* neuron i with the *successor* neuron j. As mentioned before, suppose both i and j present a semi-linear activation function. By widely exploiting the chain rule, the backpropagation of error provides an operative formula for the evaluation of the antigradient of the error function $E(\mathbf{w})$ in an arbitrary point $\mathbf{w}$, thus allowing to compute the update $\Delta w_{i,j}$ for the weight $w_{i,j}$ according to (1.17). For this purpose, we shall need to distinguish whether the successor neuron j is either an output or an inner neuron. To improve the clarity of the following mathematical derivation, we shall decorate any variable concerning a neuron with the subscript $\mathbf{p}$, thus to explicitly specify the input pattern presented.

Let $S = \{s_1, \dots, s_m\}$ the set of m neurons sending their output to j through the synapses $\{(s_1, j), \dots, (s_m, j)\}$. Recalling the definition (1.14) of the accumulated error, via the chain rule we can formulate the derivative of $E(\mathbf{w})$ with respect to the weight $w_{i,j}$ as follows:

$$\frac{\partial E(\mathbf{w})}{\partial w_{i,j}} = \sum_{\mathbf{p}\in P} \frac{\partial E_{\mathbf{p}}(\mathbf{w})}{\partial w_{i,j}} = \sum_{\mathbf{p}\in P} \frac{\partial E_{\mathbf{p}}(\mathbf{w})}{\partial u_{\mathbf{p},j}} \frac{\partial u_{\mathbf{p},j}}{\partial w_{i,j}} \tag{1.18}$$

Let us focus on the product

$$\frac{\partial E_{\mathbf{p}}(\mathbf{w})}{\partial u_{\mathbf{p},j}} \frac{\partial u_{\mathbf{p},j}}{\partial w_{i,j}} \tag{1.19}$$

occurring in Equation (1.18). Assuming the propagation function be represented as the

Algorithm 1.2 Backbone of any supervised offline learning algorithm; the procedure to compute the accumulated error is provided as well.

Input: neural network $(\mathcal{N}, \mathcal{V}, \mathbf{w}_0)$, training set $P = \{\mathbf{p}_i, \mathbf{t}_i\}_{i=1}^{N_{tr}}$,
metric $d(\cdot,\cdot) : \mathbb{R}^{Mo} \times \mathbb{R}^{Mo} \to \mathbb{R}$, tolerance ϵ, maximum number of epochs T
Output: trained neural network $(\mathcal{N}, \mathcal{V}, \mathbf{w}_{opt})$

$t = 0$
$\mathbf{w}(0) = \mathbf{w}_0$
$E(\mathbf{w}(0)) =$ OFFLINEERROR$(\mathcal{N}, \mathcal{V}, \mathbf{w}(0), P, d)$
while $t < T$ and $E(\mathbf{w}(t)) > \epsilon$ **do**
  compute weight update $\Delta\mathbf{w}(t)$ based on $E(\mathbf{w}(t))$
  $\mathbf{w}(t+1) = \mathbf{w}(t) + \Delta\mathbf{w}(t)$
  $E(\mathbf{w}(t+1)) =$ OFFLINEERROR$(\mathcal{N}, \mathcal{V}, \mathbf{w}(t+1), P, d)$
  $t \leftarrow t + 1$
end while
$\mathbf{w}_{opt} = \mathbf{w}(t)$
function $E(\mathbf{w}) =$ OFFLINEERROR$(\mathcal{N}, \mathcal{V}, \mathbf{w}, P, d)$
  $E(\mathbf{w}) = 0$
  for $i = 1, \dots, N_{tr}$ **do**
    evaluate output vector $\mathbf{y}_{\mathbf{p}_i}$, correspondent to input pattern $\mathbf{p}_i$
    $E(\mathbf{w}) \leftarrow E(\mathbf{w}) + d(\mathbf{y}_{\mathbf{p}_i}, \mathbf{t}_i)$
  end for
end function

weighted sum, so that the net input is given by (1.4). The second factor in (1.19) reads:

$$\frac{\partial u_{\mathbf{p},j}}{\partial w_{i,j}} = \frac{1}{\partial w_{i,j}} \left(\sum_{s\in S} w_{s,j}\, y_{\mathbf{p},s} - \theta_j \right) = y_{\mathbf{p},i}\,. \tag{1.20}$$

We then denote the opposite of the first term in (1.19) by $\delta_{\mathbf{p},j}$, i.e.,

$$\delta_{\mathbf{p},j} = -\frac{\partial E_{\mathbf{p}}(\mathbf{w})}{\partial u_{\mathbf{p},j}}\,. \tag{1.21}$$

$\delta_{\mathbf{p},j}$ is often referred to as the *sensitivity* of the specific error $E_{\mathbf{p}}$ with respect to neuron j. Plugging (1.21) and (1.20) into (1.18), then embedding the resulting equation into (1.17), the weight update can be cast in the form:

$$\Delta w_{i,j} = \eta \sum_{\mathbf{p}\in P} \delta_{\mathbf{p},j}\, y_{\mathbf{p},i}\,. \tag{1.22}$$

We now proceed to derive an operative formula for $\delta_{\mathbf{p},j}$. Consider the case j is an output neuron. Supposing a specific error of the form (1.15) and an identity output function, it

follows:

$$\begin{aligned}\delta_{\mathbf{p},j} &= -\frac{\partial E_{\mathbf{p}}(\mathbf{w})}{\partial y_{\mathbf{p},j}}\frac{\partial y_{\mathbf{p},j}}{\partial u_{\mathbf{p},j}} \\ &= -\frac{1}{\partial y_{\mathbf{p},j}}\frac{1}{M_O}\sum_{k=1}^{M_O}\left(t_{\mathbf{p},k}-y_{\mathbf{p},k}\right)^2\frac{\partial a_{\mathbf{p},j}}{\partial u_{\mathbf{p},j}} \\ &= \frac{2}{M_O}\left(t_{\mathbf{p},j}-y_{\mathbf{p},j}\right)\frac{\partial f_{act}(u_{\mathbf{p},j})}{\partial u_{\mathbf{p},j}} \\ &= \frac{2}{M_O}\left(t_{\mathbf{p},j}-y_{\mathbf{p},j}\right)f'_{act}(u_{\mathbf{p},j}),\end{aligned} \tag{1.23}$$

It worths mention here that (1.23) implies the derivative f'_{act} of the activation function f_{act} with respect to its argument. This motivates the requirement of a differentiable transfer function.

On the other hand, Equation (1.23) does not apply when j lies within a hidden layer, as no teaching input is provided for a hidden neuron. In that case, let us denote by $R = \{r_1, \ldots, r_n\}$ the set of n neurons receveing the output generated by j, i.e., the *successors*. We then point out that the output of any neuron can directly affect only the neurons which receive the output itself, i.e., the successors [27]. Hence:

$$\begin{aligned}\delta_{\mathbf{p},j} &= \frac{\partial E_{\mathbf{p}}(\mathbf{w})}{\partial u_{\mathbf{p},j}} \\ &= \frac{\partial E_{\mathbf{p}}(u_{\mathbf{p},r_1},\ldots,u_{\mathbf{p},r_n})}{\partial u_{\mathbf{p},j}} \\ &= \frac{\partial E_{\mathbf{p}}(u_{\mathbf{p},r_1},\ldots,u_{\mathbf{p},r_n})}{\partial y_{\mathbf{p},j}}\frac{\partial y_{\mathbf{p},j}}{\partial u_{\mathbf{p},j}} \\ &= \sum_{k=1}^{n}\frac{\partial E_{\mathbf{p}}}{\partial u_{\mathbf{p},r_k}}\frac{\partial u_{\mathbf{p},r_k}}{\partial y_{\mathbf{p},j}}\frac{\partial y_{\mathbf{p},j}}{\partial u_{\mathbf{p},j}}.\end{aligned} \tag{1.24}$$

Applying the definition of sensitivity for neuron r_k, $k = 1, \ldots, n$, under the same assumptions of Equation (1.23) we can further develop (1.24):

$$\begin{aligned}\delta_{\mathbf{p},j} &= \sum_{k=1}^{n}\left(-\delta_{\mathbf{p},r_k}\right)\frac{1}{\partial y_{\mathbf{p},j}}\left(\sum_{l=1}^{m_k} w_{s_l,r_k}y_{\mathbf{p},s_l}-\theta_{r_k}\right)\frac{\partial f_{act}(u_{\mathbf{p},j})}{\partial u_{\mathbf{p},j}} \\ &= -\sum_{k=1}^{n}\delta_{\mathbf{p},r_k}\,w_{j,r_k}\,f'_{act}(u_{\mathbf{p},j}).\end{aligned} \tag{1.25}$$

Here, we have supposed that r_k receives input signals from m_k neurons $\{s_1, \ldots s_{m_k}\}$.

Let us now collect and summarize the results derived so far. At any iteration t of the backpropagation learning algorithm, the weight $w_{i,j}(t)$ of a generic connection (i, j) linking neuron i with neuron j is corrected by an additive quantity $\Delta w_{i,j}(t)$, i.e.,

$$w_{i,j}(t+1) = w_{i,j}(t) + \Delta w_{i,j}(t).$$

Assume the accumulated mean squared error (1.16) as performance function. Understanding the dependence on time for the sake of clarity, the weight update $\Delta w_{i,j}$ reads:

$$\Delta w_{i,j} = \eta\sum_{\mathbf{p}\in P} y_{\mathbf{p},i}\,\delta_{\mathbf{p},j},$$

where $\eta > 0$, while $\delta_{\mathbf{p},j}$ is given by

$$\delta_{\mathbf{p},j} = \begin{cases} f'_{act}(u_{\mathbf{p},j})\sum_{r\in R}\delta_{\mathbf{p},r}\,w_{j,r}, & \text{if } j \text{ inner neuron}, \quad (1.26a) \\ \frac{2}{M_O}f'_{act}(u_{\mathbf{p},j})\left(t_{\mathbf{p},j}-y_{\mathbf{p},j}\right), & \text{if } j \text{ output neuron}. \quad (1.26b)\end{cases}$$

Some relevant remarks about the overall algorithm should be pointed out. First, observe that Equation (1.26a) defines $\delta_{\mathbf{p},j}$ for a hidden node j by relying on neurons in the following layer, whereas Equation (1.26b) only involves variables concerning the neuron (namely, $u_{\mathbf{p},j}$ and $y_{\mathbf{p},j}$) and the exact output $t_{\mathbf{p},j}$. Therefore, the coupled equations (1.26a)-(1.26b) implicitly set the order in which the weights must be adjusted: starting from the output layer, update all the connections ending in that layer, then move backwards to the preceeding layer. In this way, the error is *backpropagated* from the output down to the input, leaving traces in each layer of the network [27, 46].

The weight updating procedure detailed above corresponds to the offline version of the backpropagation of error, since it involves the total error E. On the other hand, the online algorithm readly comes from Equation (1.22), simply dropping the summation over the elements of the training set P.

Although intuitive and very promising, backpropagation of error suffers of all those drawbacks that are peculiar to gradient-based techniques. For instance, we may get stuck in a local minimum, whose level is possibly far from the global minimum of the error surface E. Furthermore, since the step size dictated by the gradient method is given by the norm of the gradient itself, minima close to steepest gradients are likely to be missed due to a large step size. This motivates the introduction in (1.17) of the learning rate $\eta \in (0, 1]$, acting as a reducing factor and thus enabling a keener control on the descent. As suggested by Kriesel [27], in many applications reasonable values for η lies in the range $[0.01, 0.9]$. In particular, a time-dependent learning rate usually enables a more effective and more efficient training procedure. At the beginning of the process, when the network is far from the application goal, one often needs to span a large extent of the error surface, thus to identify a region of interest. Then, the learning rate should be large, i.e., close to 1, thus to speed up the exploration. However, as we approach the optimal configuration, we may want to progressively reduce the learning rate, using the delta rule, to fine-tune the weights of the neural network.

Nevertheless, selecting an appropriate value for η is still more an art than a science. Furthermore, for sigmoidal activations functions as the ones shown in Figure 1.4, the derivative is close to zero far from the origin. This results in the fact that it becomes very difficult to move neurons away from the limits of the activation, which could extremely extend the learning time [27]. Hence, different alternatives to backpropagation have been proposed in literature, either by modifying the original algorithm (as, e.g., the resilient backpropagation [41] or the quickpropagation [12]), or pursuing a different numerical approach to the optimization problem. The latter class of algorithms includes the Levenberg-Marquardt algorithm, which we shall extensively use in our numerical tests.

Levenberg-Marquardt algorithm

While backpropagation is a steepest descent algorithm, the Levenberg-Marquardt algorithm [29] is an approximation to the Newton's method [13]. As for backpropagation, the learning

procedure is driven by the loss function $E = E(\mathbf{w})$, $\mathbf{w} \in |\mathcal{V}|$. Applying the Newton's method for the minimization of E, at each iteration the *search direction* $\Delta\mathbf{w}$ is found by solving the following linear system:

$$\nabla^2 E(\mathbf{w})\Delta\mathbf{w} = -\nabla E(\mathbf{w}), \tag{1.27}$$

where $\nabla E(\mathbf{w})$ and $\nabla^2 E(\mathbf{w})$ denotes, respectively, the gradient vector and the Hessian matrix of E with respect to its argument $\mathbf{w}$. Assume now that the loss function is represented as the accumulated mean squared error,

$$E(\mathbf{w}) = \sum_{\mathbf{p}\in P} \frac{1}{M_O} \sum_{j=1}^{M_O} \left(t_{\mathbf{p},j} - y_{\mathbf{p},j}\right)^2 = \sum_{\mathbf{p}\in P} \frac{1}{M_O} \sum_{j=1}^{M_O} e_{\mathbf{p},j}(\mathbf{w})^2; \tag{1.28}$$

with $e_{\mathbf{p},j}$ the j-th component of the error vector $\mathbf{e_p} = \mathbf{t_p} - \mathbf{y_p}$ corresponding to the input pattern $\mathbf{p}$. Then, introducing the Jacobian $J_\mathbf{p}$ of the specific error vector $\mathbf{e_p}$ with respect to $\mathbf{w}$, i.e.,

$$J_\mathbf{p}(\mathbf{w}) = \begin{bmatrix} \frac{\partial e_{\mathbf{p},1}}{\partial w_1} & \frac{\partial e_{\mathbf{p},1}}{\partial w_2} & \cdots & \frac{\partial e_{\mathbf{p},1}}{\partial w_{|\mathcal{V}|}} \\ \frac{\partial e_{\mathbf{p},2}}{\partial w_1} & \frac{\partial e_{\mathbf{p},2}}{\partial w_2} & \cdots & \frac{\partial e_{\mathbf{p},2}}{\partial w_{|\mathcal{V}|}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial e_{\mathbf{p},M_O}}{\partial w_1} & \frac{\partial e_{\mathbf{p},M_O}}{\partial w_2} & \cdots & \frac{\partial e_{\mathbf{p},M_O}}{\partial w_{|\mathcal{V}|}} \end{bmatrix} \in \mathbb{R}^{M_O \times |\mathcal{V}|} \tag{1.29}$$

simple computations yield:

$$\nabla E(\mathbf{w}) = \sum_{\mathbf{p}\in P} \frac{2}{M_0} J_\mathbf{p}(\mathbf{w})^T \mathbf{e_p} \in \mathbb{R}^{|\mathcal{V}|} \tag{1.30}$$

and

$$\nabla^2 E(\mathbf{w}) = \sum_{\mathbf{p}\in P} \frac{2}{M_O} \left[J_\mathbf{p}(\mathbf{w})^T J_\mathbf{p}(\mathbf{w}) + S(\mathbf{w})\right] \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}, \tag{1.31}$$

with

$$S(\mathbf{w}) = \sum_{\mathbf{p}\in P} \frac{2}{M_O} \sum_{j=1}^{M_O} e_{\mathbf{p},j} \nabla^2 e_{\mathbf{p},j}.$$

Assuming $S(\mathbf{w}) \approx 0$, inserting (1.30) and (1.31) into (1.27) we get the linear system

$$\left[\sum_{\mathbf{p}\in P} J_\mathbf{p}(\mathbf{w})^T J_\mathbf{p}(\mathbf{w})\right] \Delta\mathbf{w} = -\sum_{\mathbf{p}\in P} J_\mathbf{p}(\mathbf{w})^T \mathbf{e_p}(\mathbf{w}). \tag{1.32}$$

The Levenberg-Marquardt modification to the Newton's method reads [13, 29]:

$$\left[\sum_{\mathbf{p}\in P} J_\mathbf{p}(\mathbf{w})^T J_\mathbf{p}(\mathbf{w}) + \mu I\right] \Delta\mathbf{w} = -\sum_{\mathbf{p}\in P} J_\mathbf{p}(\mathbf{w})^T \mathbf{e_p}(\mathbf{w}), \tag{1.33}$$

with $\mu \geq 0$ and I the identity matrix of size $|\mathcal{V}| \times |\mathcal{V}|$. Note that if $\mu = 0$ we recover the Newton's method (1.32), while for $\mu \gg 1$ the search direction $\Delta\mathbf{w}$ approaches the antigradient of E, i.e., we recover the backpropagation algorithm. Then, the Levenberg-Marquardt algorithm

can be seen as an interpolation between the Newton's method and the steepest descent method, aiming to retain the advantages of both techniques.

The Levenberg-Marquardt training algorithm proceeds as follows. At each epoch t of the training procedure, we solve the (potentially large) linear system (1.33). Whenever the step $\Delta\mathbf{w}(t)$ leads to a reduction in the performance function, i.e., $E(\mathbf{w}(t) + \Delta\mathbf{w}(t)) < E(\mathbf{w}(t))$, the parameter μ is reduced by a factor $\beta > 1$. Conversely, if $E(\mathbf{w}(t) + \Delta\mathbf{w}(t)) > E(\mathbf{w}(t))$ the parameter is multiplied by the same factor β. This reflects the idea that far from the actual minimum we should prefer the gradient method to the Newton's method, since the latter may diverge. Yet, once in a neighborhood of the minimum, we switch to the Newton's method so to exploit its faster convergence [29].

The key step in the algorithm is the computation of the Jacobian $J_\mathbf{p}(\mathbf{w})$ for each training vector $\mathbf{p}$. Suppose that the k-th element w_k of $\mathbf{w}$ represents the weight $w_{i,j}$ of the connection (i, j), for some i and j, with $1 \leq i, j \leq |\mathcal{N}|$. Then, the (h,k)-th entry of $J_\mathbf{p}(\mathbf{w})$ is given by:

$$\frac{\partial e_{\mathbf{p},h}}{\partial w_k} = \frac{\partial e_{\mathbf{p},h}}{\partial w_{i,j}}, \quad 1 \leq h \leq M_O, 1 \leq k \leq |\mathcal{V}|. \tag{1.34}$$

We recognize that the derivative on the right-hand side of Equation (1.34) is intimately related with the gradient of the performance function. Therefore, we can follow the very same steps performed to derive the backpropagation, namely:

$$\begin{aligned} \frac{\partial e_{\mathbf{p},h}}{\partial w_{i,j}} &= \frac{\partial e_{\mathbf{p},h}}{\partial u_{\mathbf{p},j}} \frac{\partial u_{\mathbf{p},j}}{\partial w_{i,j}} \\ &= \frac{\partial e_{\mathbf{p},h}}{\partial y_{\mathbf{p},j}} \frac{\partial y_{\mathbf{p},j}}{\partial u_{\mathbf{p},j}} \frac{\partial u_{\mathbf{p},j}}{\partial w_{i,j}} \\ &= \frac{\partial e_{\mathbf{p},h}}{\partial y_{\mathbf{p},j}} f'_{act}(u_{\mathbf{p},j})\, y_{\mathbf{p},i} \\ &= \delta_{\mathbf{p},h,j}\, y_{\mathbf{p},i}, \end{aligned} \tag{1.35}$$

with

$$\delta_{\mathbf{p},h,j} := -\frac{\partial e_{\mathbf{p},h}}{\partial u_{\mathbf{p},j}} = -\frac{\partial e_{\mathbf{p},h}}{\partial y_{\mathbf{p},j}} f'_{act}(u_{\mathbf{p},j}). \tag{1.36}$$

For the computation of the derivative $\partial e_{\mathbf{p},h} / \partial y_{\mathbf{p},j}$ occurring in (1.36), further assume that within the set of neurons $\mathcal{N}$ items are ordered such that the output neurons come first, i.e.,

$$j \text{ output neuron} \quad \Leftrightarrow \quad 1 \leq j \leq M_O.$$

We can then distinguish three cases:

(i) j output neuron, $j = h$: since $e_{\mathbf{p},h} = e_{\mathbf{p},j} = t_{\mathbf{p},j} - y_{\mathbf{p},j}$, then

$$\frac{\partial e_{\mathbf{p},h}}{\partial y_{\mathbf{p},j}} = -1; \tag{1.37}$$

(ii) j output neuron, $j \neq h$: the output of an output neuron can not influence the signal fired by another output neuron, hence

$$\frac{\partial e_{\mathbf{p},h}}{\partial y_{\mathbf{p},j}} = 0; \tag{1.38}$$

(iii) j inner neuron: letting R be the set of successors of j, similarly to (1.25) the chain rule yields

$$\frac{\partial e_{\mathbf{p},h}}{\partial y_{\mathbf{p},j}} = \sum_{r\in R} \frac{\partial e_{\mathbf{p},h}}{\partial u_{\mathbf{p},r}} \frac{\partial u_{\mathbf{p},r}}{\partial y_{\mathbf{p},j}} = -\sum_{r\in R} \delta_{\mathbf{p},h,r}\, w_{j,r}\,. \tag{1.39}$$

Ultimately, at any iteration of the learning algorithm the entries of the Jacobian matrix $J_{\mathbf{p}}$ are given by

$$\frac{\partial e_{\mathbf{p},h}}{\partial w_k} = \delta_{\mathbf{p},h,j}\, y_{\mathbf{p},i}\,, \quad 1\le h\le M_O, 1\le k\le |\mathcal{V}|, w_k = w_{i,j} \text{ for some } i \text{ and } j\,,$$

with

$$\delta_{\mathbf{p},h,j} = \begin{cases} f'_{act}(u_{\mathbf{p},j}) \sum_{r\in R} \delta_{\mathbf{p},h,r}\, w_{j,r}\,, & \text{if } j \text{ inner neuron}\,, \quad (1.40a)\\ f'_{act}(u_{\mathbf{p},j})\delta^K_{jh}\,, & \text{if } j \text{ output neuron}\,. \quad (1.40b)\end{cases}$$

where δ^K_{jh} is the Kronecker delta. The steps to be performed at each iteration of the Levenberg-Marquardt algorithm are summarized in Algorithm 1.3.

Let us finally remark that a trial and error approach is still required to find satisfactory values for μ and β; as proposed in [29], a good starting point may be $\mu = 0.01$, with $\beta = 10$. Moreover, the dimension of the system (1.33) increases nonlinearly with the number of neurons in the network, making the Levenberg-Marquardt algorithm poorly efficient for large networks [13]. However, it is more efficient than backpropagation for networks with a few hundreads of connections, besides producing much more accurate results. We shall gain further insights into this topic in Chapter 3.

Algorithm 1.3 An iteration of the Levenberg-Marquardt training algorithm.

function $[\mathbf{w}+\Delta\mathbf{w}, E(\mathbf{w}+\Delta\mathbf{w}), \mu]$ = LMITERATION($\mathcal{N}, \mathcal{V}, \mathbf{w}, P, E(\mathbf{w}), d, \mu$)
  $\beta = 10$
  for $i = 1, \dots, N_{tr}$ **do**
    evaluate output vector $\mathbf{y}_{\mathbf{p}_i}$, correspondent to input pattern $\mathbf{p}_i$
    $\mathbf{e}_{\mathbf{p}_i} = \mathbf{y}_{\mathbf{p}_i} - \mathbf{t}_{\mathbf{p}_i}$
    for $h = 1, \dots, M_0$, $k = 1, \dots, |\mathcal{V}|$ **do**
      compute $(J_{\mathbf{p}_i})_{h,k}$ according to (1.35), (1.40a) and (1.40b)
    end for
  end for
  assemble and solve $\left[\sum_{i=1}^{N_{tr}} J^T_{\mathbf{p}_i} J_{\mathbf{p}_i} + \mu I\right] \Delta\mathbf{w} = -\sum_{i=1}^{N_{tr}} J^T_{\mathbf{p}_i} \mathbf{e}_{\mathbf{p}_i}$
  $E(\mathbf{w}+\Delta\mathbf{w})$ = OFFLINEERROR($\mathcal{N}, \mathcal{V}, \mathbf{w}+\Delta\mathbf{w}, P_{tr}, d$)
  if $E(\mathbf{w}+\Delta\mathbf{w}) < E(\mathbf{w})$ **then**
    $\mu \leftarrow \mu/\beta$
  else
    $\mu \leftarrow \mu * \beta$
  end if
end function

1.2.4 Practical considerations on the design of artificial neural networks

We conclude this introductory chapter on neural networks by discussing the major concerns regarding their design and implementation, mainly focusing on multi-layer perceptrons. As we shall see, most of these issues are still open questions in many research fields, and as such they should be tackled pursuing a *trial-and-error* approach.

In Section 1.2.2, we reported two fundamental results ((i) and (ii)) promoting three-layers feedforward neural networks as universal function approximators. Yet, these statements do not provide any information regarding the number of neurons and training patterns required to approximate a function up to a desired level of uncertainty. In other terms, (i) and (ii) are not operative, and therefore one has to rely on emprirical considerations and greedy approaches in the design and training of a neural network.

Clearly, the larger the training set, the better, and one could perform a sensitivity analysis on the amount of teaching inputs required to get satisfactory predictions. However, in real-life applications training data often come from an external source of information, on which we do not have any influence, and as a result the number of available teaching patterns is fixed. A relevant example is provided by recommender systems, which seek to build a predictive model based on a user's past behaviour (e.g., the items he/she has purchased) and similar choices made by other users.

The critical point for any network paradigm, designed either for continuous regression, classification or cluster-detection, is the choice of the right number of neurons it should be equipped with. As a rule of thumb, the network should feature as *few* free parameters as possible but as many as *necessary* [27]. In this respect, recall that the computing and processing power of a network is determined by its neurons, while the weighted synapses represent the information storage. Then, too few neurons (i.e., synapses) do not endow the network with the necessary representation capability, leading to poor results, i.e., large values for the performance function. On the other hand, an oversized network would rather precisely align with the teaching inputs, but is likely to fail on patterns it has not been exposed to during the training. Indeed, being the degrees of freedom of the underlying problem fewer than the network paremeters, the latter can be tuned to lower the error function at will. In other words, once the training phase is over, the system has successfully *memorized* the training patterns but is not capable of *generalizing* what it has learnt to similar (yet different) situations. In this circumstance, we say that the network *overfits* the training data.

In the decades, many expedients have been proposed to avoid overfitting. For instance, one could alter the data through additive (white) noise, thus preventing the network to perfectly fit the training set. Another well-known practice is *regularization*, which consists in correcting the error function by a regularizing term $L(\mu)$, namely

$$E(\mathbf{w}) + \lambda L(\mathbf{w})\,.$$

Here, L is a functional increasing with the components of $\mathbf{w}$, thus penalizing large values for the weights, and so preventing the network from heavily relying on individual data points [30]. The positive coefficient λ tunes the level of regularization introduced in the model: for $\lambda = 0$, we recover the original model, while for $\lambda \gg 1$ we sacrifice the performance on the training dataset for the sake of a more flexible system. As typical in neural networks, a suitable value for λ has to be found empirically.

In our numerical experiments, overfitting is prevented upon resorting to *cross-validation* combined with an *early stopping* technique [25]. Data are split in three subsets: *training*, *validation*, and *testing* data sets, denoted respectively by P_{tr}, P_{va} and P_{te}. The former consists of data which are actually used to train the network. Specifically, in the Levenberg-Marquardt algorithm these data are used to build up the linear system (1.33) yielding the weights and biases update $\Delta\mathbf{w}$. Whereas, validation samples are used to monitor the error performed by the model *during* the training, but are not involved in the training itself. Finally, the testing data set is used to assess the performance of the system once the learning phase is over, and it is thus useful to compare different models, e.g., neural networks with different number of layers and/or neurons per layer [30]. For further details on the way such subsets have been generated for our test cases, we refer the reader to the upcoming chapters.

At the beginning of the learning stage, the error on both the training and validation data set typically decreases. However, while the error yielded by the training samples should keep lowering as time advances (provided a well-chosen minimization technique), at a certain point the validation error may start increasing, meaning that the network is likely to overfit the data. Therefore, we *early stop* the procedure whenever the validation error keeps rising up for K_{ea} consecutive iterations. The optimal configuration $\mathbf{w}_{opt}$ is then that one yielding the minimum of the validation (and not the training) error curve.

Once in possession of an effective training procedure, one can basically pursue two different yet complementary strategies to determine a suitable number of neurons for a given application:

(a) moving from a (relative) small amount of neurons, progressively augment the size of the network until the error on the test data set starts increasing;

(b) consider a sufficiently large number of neurons, then possibly adopt a *pruning* strategy (e.g., the Optimal Brain Surgeon technique proposed by Hassibi & Stork [19]) to iteratively remove the *less relevant* connection, erasing a neuron whenever it gets isolated from the rest of the network.

As we shall see in Section 2.6, in this work we resort to the first approach for the sake of implementation-wise convenience. In particular, we first consider perceptrons with a single hidden layer, and we then add another computing layer whenever necessary[5].

Lastly, let us point out that the final network configuration resulting from any learning strategy is affected by the initial values assigned to the synaptic and bias weights. As a good practice, the weights should be initialized with random values averaging zero. In this respect, a random uniform sampling over $[-0.5, 0.5]$ or a standard Gaussian distribution may be adequate choices. Then, to limit the dependence of the results on the initial configuration $\mathbf{w}_0$, we train each network topology several times, say K_{mr}, employing different $\mathbf{w}_0^k$, $k = 1, \ldots, K_{mr}$, finally keeping the configuration yielding the minimum error on the test data set. This approach is usually referred to as *multiple-restarting*. The definitive training procedure used in this project, based on the Levenberg-Marquardt algorithm and keeping into consideration all the observations made in this section, is given in Algorithm 1.4.

[5] Recall that for perceptrons equipped with differential activation functions, two hidden layers are sufficient to approximate *any* function.

Algorithm 1.4 The complete training algorithm adopted in our numerical tests.

function $[\mathbf{w}_{opt}, E_{opt}]$ = TRAINING($(\mathcal{N}, \mathcal{V})$, P_{tr}, P_{va}, P_{te}, d, K_{ms}, ϵ, T, K_{ea})
  $E_{opt} = \infty$
  for $j = 1, \ldots, K_{ms}$ **do**
    $t = 0$, $k = 0$, $\mu = 0.01$
    randomly generate $\mathbf{w}(0)$
    $E_{tr}(\mathbf{w}(0))$ = OFFLINEERROR($\mathcal{N}, \mathcal{V}, \mathbf{w}(0), P_{tr}, d$)
    while $t < T$ and $E_{tr}(\mathbf{w}(t)) > \epsilon$ and $k < K_{ea}$ **do**
      $[\mathbf{w}(t+1), E_{tr}(\mathbf{w}(t+1)), \mu]$ = LMITERATION($\mathcal{N}, \mathcal{V}, \mathbf{w}(t), P_{tr}, E(\mathbf{w}(t)), d, \mu$)
      $E_{va}(\mathbf{w}(t+1))$ = OFFLINEERROR($\mathcal{N}, \mathcal{V}, \mathbf{w}(t+1), P_{va}, d$)
      if $E_{va}(\mathbf{w}(t+1)) > E_{va}(\mathbf{w}(t))$ **then**
        $k \leftarrow k + 1$
      else
        $k = 0$
      end if
      $t \leftarrow t + 1$
    end while
    $\mathbf{w}_{opt}^{(j)} = \mathbf{w}(t-k)$
    $E_{te}(\mathbf{w}_{opt}^{(j)})$ = OFFLINEERROR($\mathcal{N}, \mathcal{V}, \mathbf{w}_{opt}^{(j)}, P_{te}, d$)
    if $E_{te}(\mathbf{w}_{opt}^{(j)}) < E_{opt}$ **then**
      $\mathbf{w}_{opt} = \mathbf{w}_{opt}^{(j)}$
      $E_{opt} = E_{te}(\mathbf{w}_{opt}^{(j)})$
    end if
  end for
end function

Chapter 2

Reduced basis methods for nonlinear partial differential equations

In this chapter, we combine a reduced basis (RB) method with neural networks for the efficient resolution of parametrized nonlinear partial differential equations (PDEs) defined on variable shape domains. As illustrative yet relevant instances of nonlinear PDEs, the nonlinear Poisson equation and the stationary incompressible Navier-Stokes equations are used as expository vehicles. The latter feature a quadratic nonlinearity laying in the convective term, while we consider instances of the Poisson equation where the nonlinearity stems from a solution-dependent viscosity (or diffusion coefficient). Throughout this work, we shall confine the attention to one- and two-dimensional differential problems. However, the proposed RB procedure can be almost effortlessy extended to higher dimensions, and to other classes of either linear or nonlinear PDEs as well.

Before diving into the mathematical foundation of the proposed RB method, it worths provide an overview of the whole numerical procedure. In doing so, we seek to provide some insights on the rational behind reduced-order modeling, and to point out the motivations for resorting to neural networks, clarifying their role within the algorithm. Furthermore, we shall start setting the notation which will be used throughout the chapter.

Let $\boldsymbol{\mu}_{ph} \in \mathscr{P}_{ph} \subset \mathbb{R}^{P_{ph}}$ and $\boldsymbol{\mu}_g \in \mathscr{P}_g \subset \mathbb{R}^{P_g}$ be respectively the *physical* and *geometrical* parameters characterizing the differential problem at hand. The former address material properties (e.g. the viscosity in the Poisson equation), source terms and boundary conditions, while the latter define the shape of the computational domain $\widetilde{\Omega} = \widetilde{\Omega}(\boldsymbol{\mu}_g) \subset \mathbb{R}^d$, $d = 1, 2$. Furthermore, assume both $\mathscr{P}_{ph}$ and $\mathscr{P}_g$ compact (i.e., closed and bounded) subsets of $\mathbb{R}^{P_{ph}}$ and $\mathbb{R}^{P_g}$, respectively, and denote by $\boldsymbol{\mu} = (\boldsymbol{\mu}_{ph}, \boldsymbol{\mu}_g) \in \mathscr{P} = \mathscr{P}_{ph} \times \mathscr{P}_g \subset \mathbb{R}^P$, $P = P_{ph} + P_g$, the overall *input vector parameter*. For a given $\boldsymbol{\mu}$ in the parameter space $\mathscr{P}$, we then seek the corresponding solution $\widetilde{u} = \widetilde{u}(\boldsymbol{\mu})$ of the underlying PDE in a suitable Hilbert space $\widetilde{V}(\boldsymbol{\mu}_g)$, defined over the computational domain $\widetilde{\Omega}(\boldsymbol{\mu}_g)$.

In the context of RB methods, when dealing with domains undergoing geometrical transformations, it is convenient to introduce a parametric map

$$\Phi : \Omega \times \mathscr{P}_g \to \widetilde{\Omega},$$

enabling the formulation and resolution of the differential problem over a fixed, parameter-independent, reference domain Ω [32], such that

$$\widetilde{u}(\boldsymbol{\mu}) = u(\boldsymbol{\mu}) \circ \Phi(\boldsymbol{\mu}_g).$$

Here, $u(\boldsymbol{\mu})$ represents the solution over the reference domain Ω, and lies in a suitable Hilbert space V. Then, $u(\boldsymbol{\mu})$ can be regarded as a map linking the parameter domain $\mathscr{P}$ with V, i.e.,

$$u : \mathscr{P} \to V. \tag{2.1}$$

The map (2.1) defines the *solution manifold* $\mathscr{M} = \{u = u(\boldsymbol{\mu}) : \boldsymbol{\mu} \in \mathscr{P}\} \subset V$, consisting of solutions to the parametrized PDE for any admissible input parameter [20]. In Section 2.2, we detail how to recover the formulation over the fixed domain for the differential equations in consideration. In addition, we present therein an effective way to build the volumetric parametrization $\Phi(\boldsymbol{\mu}_g)$ given the boundary parametrization of $\widetilde{\Omega}(\boldsymbol{\mu})$ [23].

In real life applications, differential equations often do not admit an analytical solution in closed form. Hence, one has typically to resort to some numerical schemes, e.g., the finite element (FE) method, providing a *high-fidelity* approximation $u_h(\boldsymbol{\mu})$ of $u(\boldsymbol{\mu})$. The discrete solution u_h is sought in a finite-dimensional space $V_h(\boldsymbol{\mu}_g) \subset V(\boldsymbol{\mu}_g)$, and can be identified through the associated degrees of freedom $\mathbf{u}_h(\boldsymbol{\mu}) \in \mathbb{R}^M$, yielded by a nonlinear algebraic system of the form:

$$\boldsymbol{G}_h(\mathbf{u}_h(\boldsymbol{\mu}); \boldsymbol{\mu}) = \mathbf{0} \in \mathbb{R}^M. \tag{2.2}$$

For the FE method, $\mathbf{u}_h$ collects the nodal values of the corresponing solution u_h, and therefore represents the algebraic counterpart of u_h. In the following, we shall refer indiscriminately to both u_h and $\mathbf{u}_h$ as the *full-order* or *truth* solution.

Many applications entail the repeated resolution of a parametrized PDE in various settings and over different geometrical configurations, namely, for several parameter values. Notable examples include parametric sensitivity analysis, optimal control, topology optimization, and uncertainty quantification. From a geometrical standpoint, one has then to span the *discrete solution manifold* $\mathscr{M}_h = \{u_h = u_h(\boldsymbol{\mu}) : \boldsymbol{\mu} \in \mathscr{P}\} \subset V_h$. To this end, at the algebraic level this would imply the resolution of a possibly large number of nonlinear systems as (2.2). As the dimension M of the discrete space V_h increases, this direct approach becomes prohibitive in the practice, both in terms of CPU time and storage, and even on massive parallel workstations [39].

In this scenario, *reduced order modeling* (ROM) (also known as *order model reduction*) aims at replacing the computationally expensive discrete model, encoded by the large-scale system (2.2), with a reduced problem of significant smaller dimension L, with L *independent of* M. The variegate ROM methods proposed in literature differ for the way the reduced system is assembled starting from the full one. In particular, *reduced basis* (RB) methods supply a *reduced space* $V_{\mathrm{rb}} \subset V_h$, approximating the discrete solution manifold $\mathscr{M}_h$ by means of L well-chosen *basis functions* $\{\psi_1, \dots, \psi_L\} \subset V_h$, namely

$$V_{\mathrm{rb}} = \mathrm{span}\{\psi_1, \dots, \psi_L\} \quad \text{and} \quad \dim V_{\mathrm{rb}} = L.$$

Then, given $\boldsymbol{\mu} \in \mathscr{P}$ a *reduced solution* u_L is sought in the reduced space V_{rb}. Denoting by $\mathbb{V} = [\boldsymbol{\psi}_1 | \dots | \boldsymbol{\psi}_L] \in \mathbb{R}^{M \times L}$ the matrix collecting the nodal evaluations of the basis functions, the (algebraic) reduced solution has the form

$$\mathbf{u}_L = \mathbb{V}\, \mathbf{u}_{\mathrm{rb}} = \sum_{i=1}^{L} u_{\mathrm{rb}}^{(i)} \boldsymbol{\psi}_i, \tag{2.3}$$

so that

$$u_L(\boldsymbol{x}; \boldsymbol{\mu}) = \sum_{i=1}^{L} u_{\mathrm{rb}}^{(i)}(\boldsymbol{\mu}) \psi_i(\boldsymbol{x}). \tag{2.4}$$