Università degli Studi Roma Tre

Scuola Dottorale in Ingegneria dell'elettronica biomedica,
dell'elettromagnetismo e delle telecomunicazioni
XXVIII Ciclo

Softcomputing Techniques on Embedded Systems for Industrial Engineering and Information Technology

Gabriele Maria Lozito

Tutor:
Prof. Francesco Riganti Fulginei

Coordinatore del Dottorato:
Prof. Alessandro Salvini

Contents

Introduction

Engineering problems are seldom solved by exact solutions.

Even simple tasks like direct measurement of a quantity involves interaction with complex nonlinear systems that can strive very quickly from the ideal representation given by theory. For this reason, several techniques (based on numerical approaches) have been developed to give an approximate solution to engineering problems that cannot be solved (either for their nature or for practical reasons) by exact techniques. These techniques falls under the name of "Soft Computing". A notable example that can be used to introduce the concept behind Soft Computing is the representation of systems through numerical models. Creation, identification and simulation of models are three tasks that are often faced in all engineering fields. Since models are a representation of reality, it is difficult to classify them a priori regardless of the object that they are representing. However, a meaningful classification of lies in the black-white box models. A white box model is a mathematical representation of a phenomena, or system, completely based on theoretical knowledge. Laws determining this model behavior are exact, and usually, comes from physics considerations. These models are defined by a low number of parameters, where each parameter can be interpreted in

physical terms. For this reason, these models are often identified by physical measurements. However, since knowledge of physical laws is indeed limited, this model can only be used to represent ideal systems. A black box model, on the other hand, completely disregard the knowledge of the inner mechanisms that regulates the system, and aims at emulating its behavior through data observation. This kind of model features a high number of parameters, has higher computational costs, and to be identified, requires a large quantity of data from the system itself. Moreover, by observing the model parameters no knowledge can be acquired on the system, and for this reason, they are determined through numerical techniques. The real advantage of black-box models over white-box ones is that this technique can be used on the majority of real engineering problems where the complexity of the system makes a white box model unusable. A very common black-box model is an Artificial Neural Network (ANN). This technique will be deeply analyzed in this work, but for now, we will just describe it as a black box featuring an input-output architecture, which can be internally tuned by numerical algorithms to emulate a system behavior.

This very versatile tool has a range of collateral research topics that has shown the interest of the academic community in the last decades. The most common application is, indeed, mathematical modeling of numerical data coming from observations of phenomenons, but more complex ANN have been created for tasks that go beyond their original scope. Among these tasks we find clustering and classification, decision making, signal processing, forecasting and optimum predictors.

Apart from very peculiar ANN that are based on uncommon operations

(e.g. the Convolutional Deep Neural Networks used by Google for image classification [Krizhevsky et al., 2012]) the intrinsic nature of the ANNs proposed in literature is similar. Numerical input data is processed by a combination of linear and non-linear operations, that can be either static or dynamic (i.e. representable as a input-state-output non-linear dynamic system) to produce a numerical output. These operations may carry an onerous computational cost, but from the complexity point of view, are in general trivial. In the majority of the cases, it will reduce to simple matrix-based operations and few non-linear function evaluations. On the other hand, the "tuning" of this black-box model is rather complex. This process, addressed as "training", can be classified as a non-linear optimization problem. The number of techniques used for training an ANN is very high, and in this work, a selection of the most interesting ones has been proposed. On the matter of non-linear optimization, a full chapter will be dedicated to analyze inverse problems and the most common techniques used in literature for the solution of these problems. All these techniques can be combined for the training of an ANN (a very common technique is the use of a Genetic Algorithm to precondition an ANN [Leung et al., 2003]).

The second part of this work will be centered on the problem of ANNs implementation in embedded environments. This is of great interest to the engineer since an ANN can be used for several tasks that are usually posed in embedded systems: filtering, control, prediction and simulation. The study of the implementation for such a general-purpose tool on embedded platforms must start from the environment itself. Several platforms with different characteristics are interesting for the implementation of this technique. The

platforms differs from each other in terms of raw computational capabilities, cost, programming tools and functionalities implemented (i.e. networking, interfaces, AD/DA conversion). Indeed, computational capabilities are limited on an embedded unit, as is the memory available. Smart management of available resources is paramount for a successful implementation of these techniques in such environment. This can be done in several ways, but as will be shown, a great deal of the computational costs associated to the ANN lies in its non-linear component. Several techniques to tame such problem will be shown in this part of work.

The last part of this work will present the applications that have been implemented. The first one is a control-based application, focused on solving the problem of Maximum Power Point Tracking found in Photovoltaic devices. The second one regards the estimation and prediction of the solar irradiance, a quantity critical for energy generation assessment and storage management. The third one is a Biomechanics problem, involving the estimation of the muscular forces of a cyclist during the cycling activity. All the applications will be analyzed in terms of chosen platform, implemented algorithms and code, computational costs and techniques used for the optimization of the performance.

Following this last part, conclusions and final remarks on future developments will close this work.

Part I

Soft-computing for Engineering Applications

Chapter 1

Neural Networks and Data Processing

1.1 Supervised Artificial Neural Networks

In machine learning theory, supervised learning is defined as the inference of a function from organized training data. The training data (also known in ANN literature as *training set*) comes from examples from the function that needs to be inferred. Generally, each sample from the set is a pair of input object (that can be either a vector or a scalar) and a desired output (that can be as well either a vector or a scalar). Through an algorithm the ANN is tuned, and in an optimal scenario, the tuned ANN will be able to generalize the function correctly. According to the vector/scalar nature of the input/output, ANN can be defined as SISO (Single Input Single Output), MISO (Multiple Input Multiple Output) and MIMO (Multiple Input Multiple Output). SIMO ANN are seldom used (it is more practical to train separate

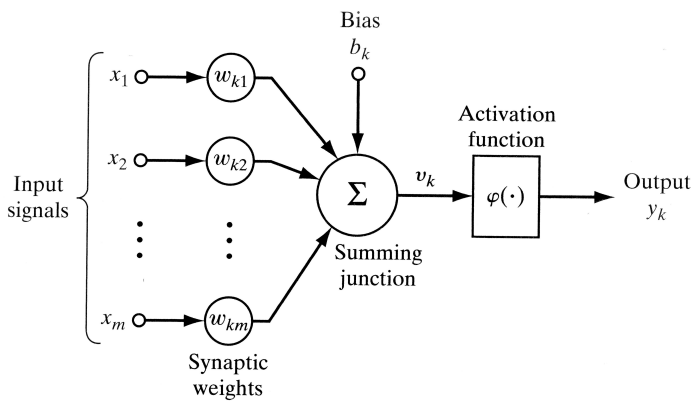


Figure 1.1: The Artificial Neuron

networks for each output). In this section, several aspects of this kind of ANN will be analyzed: the basic theory, a selection of useful architectures, the most important training algorithms available.

1.1.1 What is an Artificial Neural Network

An ANN is a mathematical model able to represent nonlinear relationships for systems whose inputs and outputs can be represented as a numerical quantity. The paradigm at the base of an ANN is of biological inspiration. The neural system of most of the evolved mammals is composed by elementary units called neurons. Each neuron has a small computational capability, limited to consider the cumulative effect of all the impulses coming from other neurons, and according to a particular threshold, fire (or not) an impulse to the other neurons itself. A neural system for a primate is composed, approximatively, by 10^13 neurons and 60×10^{18} connections. The extreme complexity of this network allows an advanced routing for signals that creates complex

behaviors like the ones present in a sentient being. This concept is at the base of the connectivism hypothesis, stating that "knowledge is distributed across a network of connections, and therefore learning consists of the ability to construct and traverse those networks" (Downes, Stephen). Inspired by such system, the mathematical tool known as ANN mimics the elementary behavior of a biological neuron, and tries to expand its computational capabilities by creating a complex network of connections between the neurons. The mathematical unit of an ANN is the Artificial Neuron, which is shown in Figure 1.1. The input-output relation of the k-th neuron of the network is given by:

$$u_k = \sum_{j=1}^{m} w_{kj} x_j \tag{1.1}$$

$$y_k = \phi(u_k + b_k) \tag{1.2}$$

Where x_j are the input signals, w_{kj} represent the weight of the j-th input, b_k is a bias term for the k-th neuron. With ϕ a generic function that maps the "weighted sum of inputs" (i.e. u_k, also referred in literature as net) to the output of the neuron is expressed. This is called the Activation Function (or AF), and extended research on different proposals for this function can be found in literature. In the simplest case, this function is a purely linear relationship, propagating the net value directly in output. However, if a neural network were composed only by units featuring linear AFs, the modeling capabilities of an ANN would be limited to linear systems. To expand these capabilities, usually the ϕ function is a non-linear squashing function. Commonly used functions are the hyperbolic tangent, or the sigmoid:

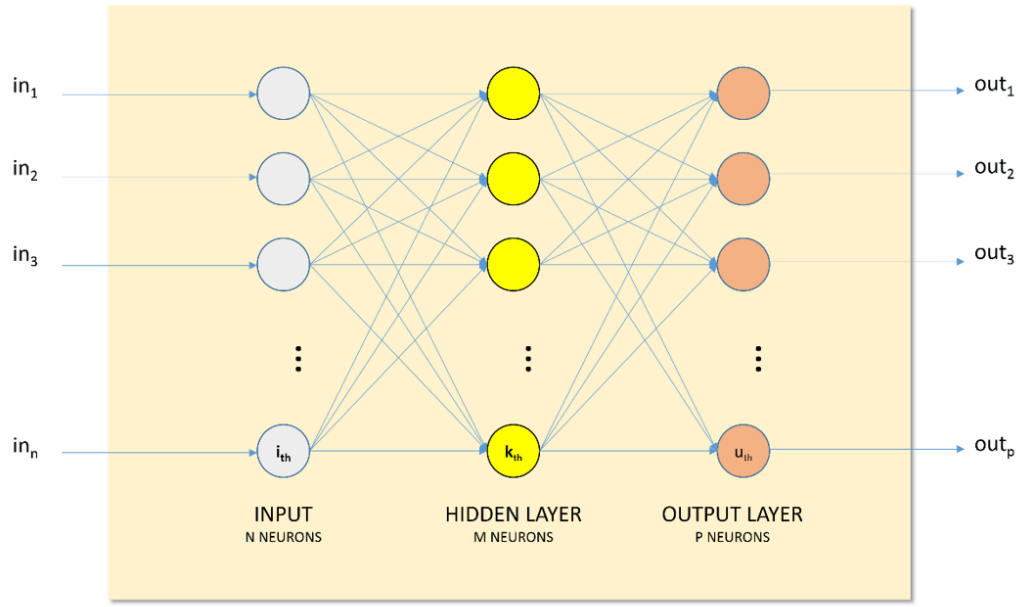


Figure 1.2: A Feed Forward Artificial Neural Network

$$\phi(v) = \frac{1}{1 + exp(-av)} \tag{1.3}$$

Given this definition of artificial neuron, an ANN is a set of different neurons connected according to some criterion. The simplest criterion for connection that gives useful results is the Feed Forward (FF) connection, which will be explained in the following paragraph.

1.1.2 The Feed Forward Architecture and the Universal Approximation Theorem

The paradigm is structured with the following rules:

- Neurons are organized in layers.

- Each layer contains a finite number of neurons.
- Each neuron in the network is connected to all (**and only**) neurons of the subsequent layer.

In a FF network all the connections points towards the exit of the network. No connection bypass the layers, and no connection goes backwards. An example can be seen in Figure 1.2. Neurons in the input layer are dummy units (they perform no action) but are usually represented in literature to express the "number of inputs" of the network. Neurons in the Hidden Layer are the ones featuring the non-linear AF, and are the one responsible for the non-linear mapping capabilities of the ANN. More than a hidden layer can be present. Neurons in the output layer are usually linear, and provide a normalization for the Hidden Layer output. This is needed because the squashing function has bounded outputs. The degrees of freedom of an ANN are the weights and the biases of its neurons. An ANN with a higher number of neuron will have more degrees of freedom than one with fewer neurons. By changing the values of these quantities, an ANN is able to behave differently. By considering an arbitrarily large ANN, thus with arbitrarily numerous degrees of freedom, it is intuitive that it could approximate any given function. This concept was formally proven in 1989 by Cybenko FF networks. The proof itself is beyond the scope of this work, however, the conditions required for an ANN to be an universal approximator include some assumptions on the AF for the neurons in the hidden layer: the function should be bounded, monotonically increasing and continuous. Under this assumption, it can be proved that the superposition of such functions (i.e.

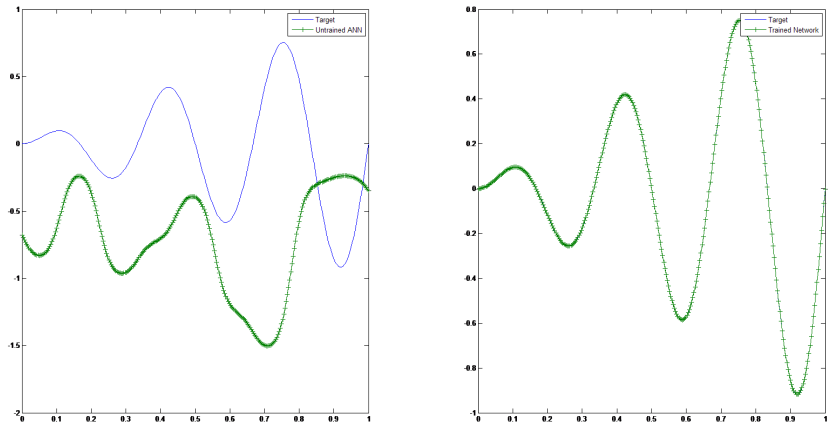


Figure 1.3: Difference between an initialized (left) and a trained (right) ANN.

the output of the ANN itself) can approximate a function with an arbitrarily low error:

$$F(x) = \sum_{i=1}^{N} v_i \phi(w_i^T x + b) \tag{1.4}$$

$$|F(x) - f(x)| < \epsilon \tag{1.5}$$

Where $F(x)$ is the linear superposition of the i-th weighted (v_i) activation function (ϕ) centered in b and scaled by w_i , and $f(x)$ is the desired interpolated function. The assumptions on the AF, as will be described in the following chapters, will be very important when different AFs will be considered for alternative implementations.

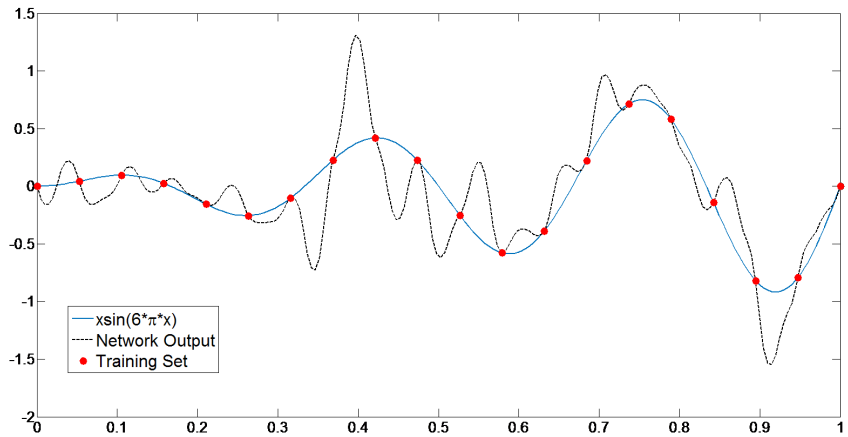


Figure 1.4: An over-fitted ANN

1.1.3 Training and Generalization

Weights and biases for the neurons of an ANN cannot be set directly. An ANN is a black box model, thus there is no explicit connection between the phenomenon and the model parameters. For this reason, algorithmic techniques were developed to modify automatically the weights and the biases of the neurons according to an error minimization criterion. These algorithms are called "Training Algorithms", and are based on a process defined as "Supervised Learning". This process involves gathering a wide set of examples, defined as couples of **input** and **output** quantities for a specific system, called **training set**. The output samples are often defined as targets, since they represent the desired behavioral response for the ANN. The procedure goes as follows:

1. The ANN is initialized with a defined architecture and random weights/biases.

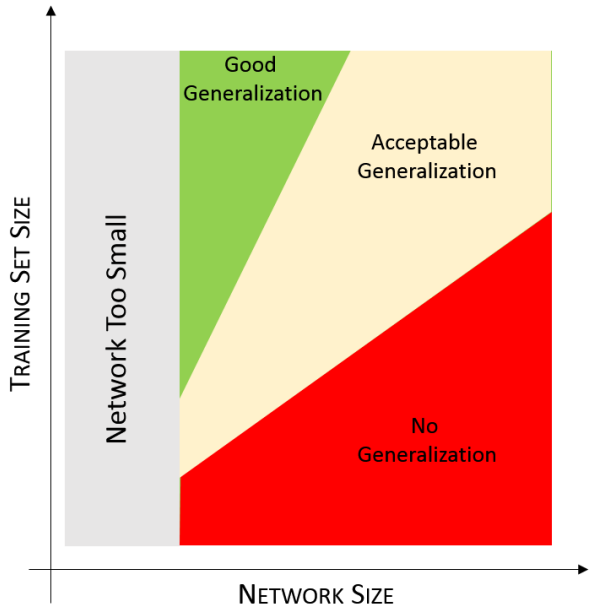


Figure 1.5: Generalization capabilities of an ANN according to the network size and the training set size

2. An input sample from the training set is sent into the ANN, and the output is computed.

3. The error between the output and the target relative to that input is computed.

4. Using the error, the training algorithm create a correction matrix for the weights and the biases of the ANN.

5. Accumulate both error and the correction matrix.

6. If available, a new sample is chosen (go to 2), else, go to 7.

7. Update the weights and biases with the accumulated correction matrix.

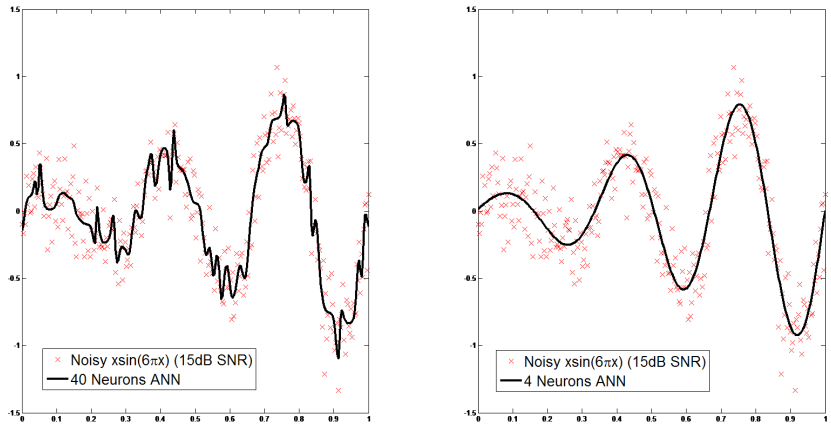


Figure 1.6: Filtering properties of a properly sized ANN

An epoch has passed.

8. If convergence is reached, (i.e. the accumulated error is low enough) or the number of epochs (i.e. full sweeps of the training sets) reached the maximum value, exit, else, repeat from 2.

As the ANN is initialized with random weights and biases, the output for a given input is meaningless and random. After the training procedure is over, the output for a given sample has been optimized to be as close as possible to the target. An example of this behavioral change can be seen in Figure 1.3. In this case, the input vector is composed by 100 samples linearly spaced between 0 and 1. The target vector is composed by 100 samples of the function $y(x) = x\sin(6\pi x)$ where x is a sample of the input vector. In Figure 1.3 (left), the target vector and the untrained ANN output is shown. As it can be seen, the output is completely random. In Figure 1.3 (right), the output of the trained ANN is shown. After 659 epochs (i.e. after showing the

training algorithm all the 100 samples from the training set for 659 times) convergence was reached. This particular ANN featured 10 neurons in the hidden layer. An apparent misconception on the size of an ANN is that the only upper boundary in it lies in the computational capabilities required to train the network and compute the output. The idea is that the larger the ANN, the better modelling capabilities will exhibit. This idea is well supported by the evidence that, given a generic problem (i.e. a traning set the ANN has to train on using a training algorithm), the larger the ANN is, the quicker convergence is reached. However, reaching convergence quickly by increasing the size of an ANN comes to a cost in terms of generalization capabilities. Generalization is the capability of an ANN to exhibit correct behaviour when new inputs (i.e. inputs that do not belong to the training set) are presented to the network itself. Indeed, generalization is the real desired characteristic of the ANN. Data from the training set is already known. What the ANN is useful for is filling the gaps between this data, providing a consistent behaviour with what was shown during the training procedure. Unfortunately, when an ANN is over-sized (i.e. too many neurons are included) the large number of degrees of freedom causes the undesired effect of "overfitting". An over fitted ANN is a network where the training procedure managed to tune the weights of the ANN towards a solution that allows a fine representation of the training set points. However, if some of these points are removed from the training set before the training procedure, and are shown to the ANN afterwards as a validation set, the error on these points is very high. This is because the ANN aims at reducing the error on the training set at the limit, by exploiting all the degrees of freedom that are

given by the large number of neurons. This, at the cost of introducing strong distortions outside the training set points. This problem is especially true if the training set size is small. The behaviour can be seen in Figure 1.4. In this case, an ANN with a large number of neurons (40) is used on a rather small dataset (20 points) belonging the same sinusoidal function used before. The desired output is shown with the full blue line. The network output after the training is shown with the dotted black line. The training set is shown with the red dots. The ANN behaviour on the training set points is perfect (in fact, after just 10 epochs, an error of 10^{-18} was reached), however as soon as the input deviates from the points used in training, the modelling performance drops rapidly. This is due to a simple concept:

- The training of an ANN is a multimodal non-linear optimization problem.
- The number of local minima in this problem is proportional to the number of degrees of freedom given to the ANN.

Demonstrating this statement formally is beyond the purpose of this work, however it can be seen by studying the surface of the error function that, for every added degree of freedom, a new local minimum appears. This means that an oversized ANN could reach a global optimum where it would model correctly the behavior of the system. However, the restricted knowledge given by the small traning set makes it impossible to recognize a local minimum from a global optimum. Since the enlargement of the training set is not always an option, the rule of thumb to size an ANN that preserves generalization capabilities, is to use as few neurons as possible. Indeed, given a

particular problem, a minimum number of neurons to represent the behavior correctly is needed. Below that number of neurons, the network is just too small to represent the relationship. However, as soon as that critical number of neuron is reached, increasing the numbers of neurons further will only apparently reduce the complexity of the training procedure, at the cost of losing generalization capabilities. If the training procedure is too difficult, and the number of neurons needs to be increased, the only way to maintain the generalization capabilities of the ANN is to increase the size of the training set as well. A schematic representation of the different scenarios that can be faced when training an ANN can be found in Figure 1.5. The figure represent four possible scenarios, according to the size of the ANN and the size of the training set. The grey area in the left represent a network too small to model the system correctly. In this case, increasing the training set will not yield any advantage: the number of neuron must be increased. The red area represent an example of over fitted network: the number of neurons has been increased too much and the size of the training set is too small to discern the global optimum from the local optima efficiently. The yellow area is an acceptable area, where the error on the validation set is slightly higher than the one on the training set. The green area is the desired one, where the two error are comparable. An ANN trained to lie in the green area will provide the best generalization capabilities at the lowest computational costs. The obvious drawback is that, compared to a network lying in the yellow area, this network takes much more time to be trained, both because the training set is larger, and because there are fewer convergence minima. A side advantage of a smaller network, compared to a larger one, is noise rejection.

The lower the number of hidden neurons, the greater the ANN capability to erase the input noise. The reason lies in the degrees of freedom that would be required to represent noise. A smaller ANN will be able to follow the general trend of a function, discarding the high frequency noise. A bigger ANN will try to learn the noise as well, with undesired results. In Figure 1.6, two examples of training are shown for the same training set. The function is always the same, $y(x) = x\sin(6\pi x)$ with 300 samples between 0 and 1, but this time a white Gaussian noise was added, with a Signal to Noise Ratio of $15dB$. As can be seen, the complex ANN to the left gives poor results since the additional degrees of freedom favors the ANN in following the noise. On the other hand, the simpler ANN to the right discards the noise in favor of following the function general trend. Indeed, correctly sizing an ANN will give the added benefit of restricting the band of the signal to the useful and meaningful one.

1.1.4 Fields of Application

Supervised ANN fields of application are very wide due to the flexibility of such technique. A very common use is to exploit their generalization capabilities for control purposes: an ANN can be used in place of a feedback loop to control a particular non-linear system in a direct way. This is very useful in embedded systems. As shown in Figure 1.7, a classic feedback control system measure the output of the system through a Sensor, compares it with the Reference, and adjusts the controller output until the error is minimum. An ANN can be trained to predict the correct system input for each reference, thus removing the need for a feedback loop and simplifying the control sys-

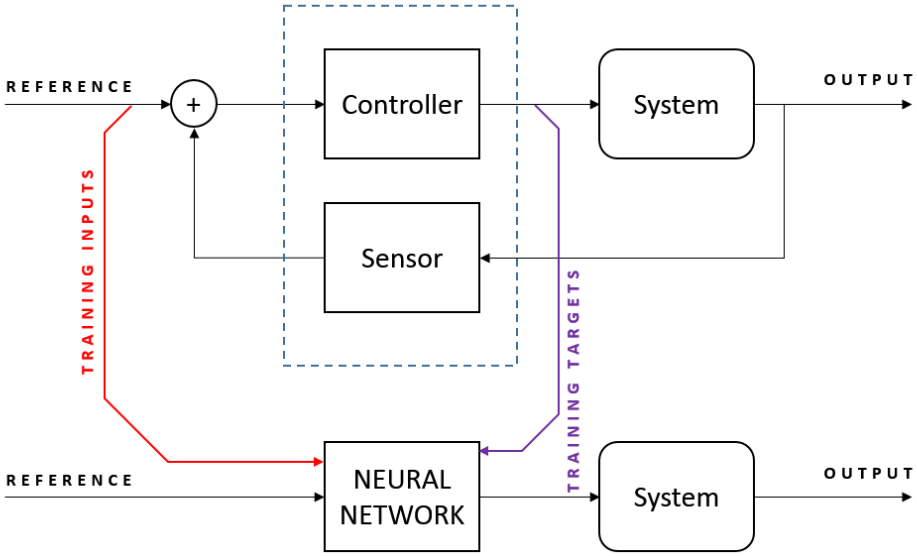


Figure 1.7: Using an ANN as forward controller

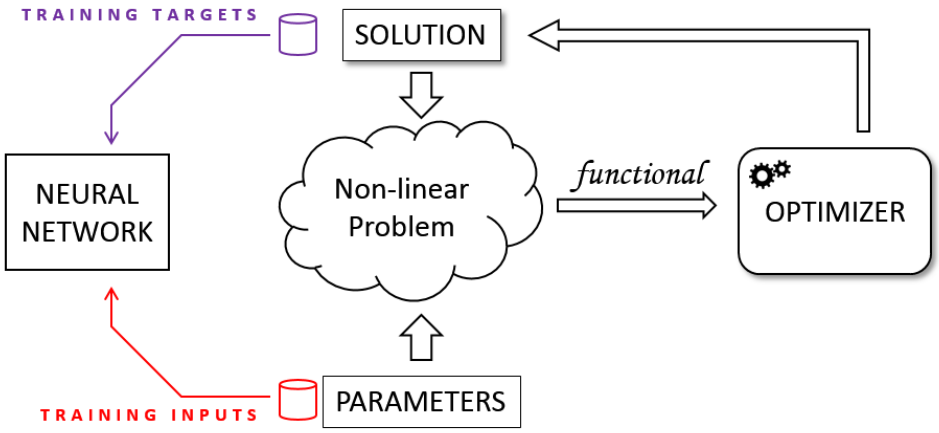


Figure 1.8: Using an ANN as an optimum predictor

tem. The drawback, of course, is that this ANN controller is specific for the particular system, and must be re-trained if the system itself changes. Neural Networks can also be used as simple interpolators for acquired data. In this case, the advantage over classic interpolation techniques lies in the robustness of the ANN towards noisy and unevenly spaced data. A particular set of ANN features delayed connections that creates a dynamic behavior for the network. This can be useful both for non-linear filtering of data, and for time series prediction. Particular classes of ANN are used for this last task, with extraordinary results even on difficult dynamic problems like chaotic time series prediction. Artificial Neural Networks can be also useful as optimum solution predictor for optimization problems. As will be shown in the next chapters, non-linear optimization problem requires complex algorithms to be solved. To be solved, a problem must be representable through a functional f that express the goodness of the solution found so far.

$$Fitness = f(solution, parameters) \tag{1.6}$$

The fitness is a numerical value expressing the goodness of the solution. The functional depends on both the solution and the parameters of the problem. The parameters of the problem are assigned, and for that particular problem, the best solution must be found. Classically, the solution is found through a trial-and-error approach performed by the optimizer. However, once the problem has been solved several times, a set of different parameters, with relative optimal solution, can be accumulated. Such database can be used to train an ANN, as shown in Figure 1.8. Such network will be able to predict the optimal solution to an optimization problem given the prob-

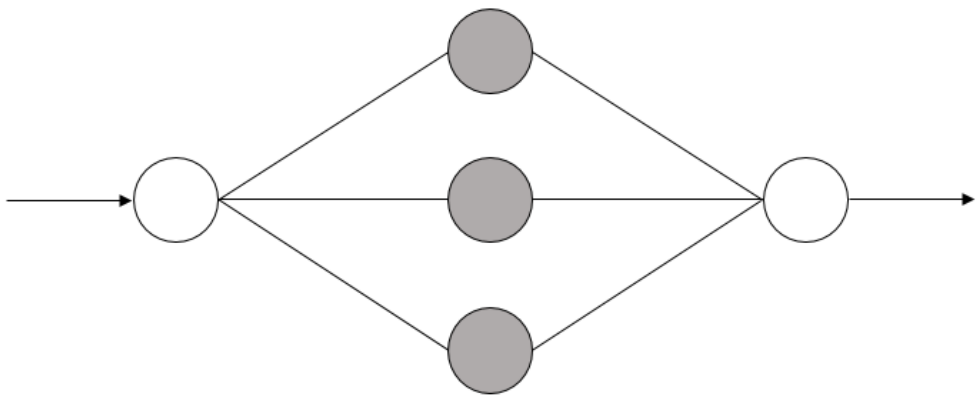


Figure 1.9: Feed Forward Neural Network

lem parameters. This is very useful in embedded systems because an ANN computation time is known and defined, whereas an optimization algorithm is generally an iterative approach.

1.1.5 Practical Architectures

This subsection will examine several architectures that are interesting for practical applications in engineering problems. Indeed, the number of architectures for ANN that can be found in literature is very high. However, often these architectures are mainly thought for theoretical purposes, and are rather difficult to use in practical scenarios. Each architecture has advantages and drawbacks, and the proper one should be chosen according to the problem at hand.

Feed Forward (Multilayer Perceptron)

Although it has been described earlier, this architecture is recalled here for the sake of completeness. The connection criterion is based on the assumption that every neuron of a layer is connected to ALL and ONLY the neurons of the next layer.The architecture is simple, computation is fast and memory footprint is acceptably small. Due to the simple nature of the connections, computing the gradient of the error with respect to the network weights is very easy (i.e. can be done with simple backpropagation). Considering an arbitrarily large number of neurons, the network has been proven a universal interpolator. However, a drawback for this architecture lies in its efficiency. Indeed complex problem can be solved by increasing the number of neurons, but this comes with a cost of increased memory footprint and computational time. For this particular architecture, the number of neurons required for a specific problem is, in general, higher than other architectures. A very common benchmark can be the classic parity-N problem: N Boolean inputs are sent in a XOR logic port that gives the output. Even the simple second order problem is not linearly separable, thus making it a suitable test bench for a non-linear classifier like an ANN. It has been seen that a parity-N problem, to be solved by a Feed Forward network requires at least $N + 1$ neurons.

Bridged Neural Networks

Several problems involve an output that is the linear combination of the input and an unknown additional signal that can be, indeed, function of the signal itself.

$$F(x) = a \times x + b \times g(x) \tag{1.7}$$

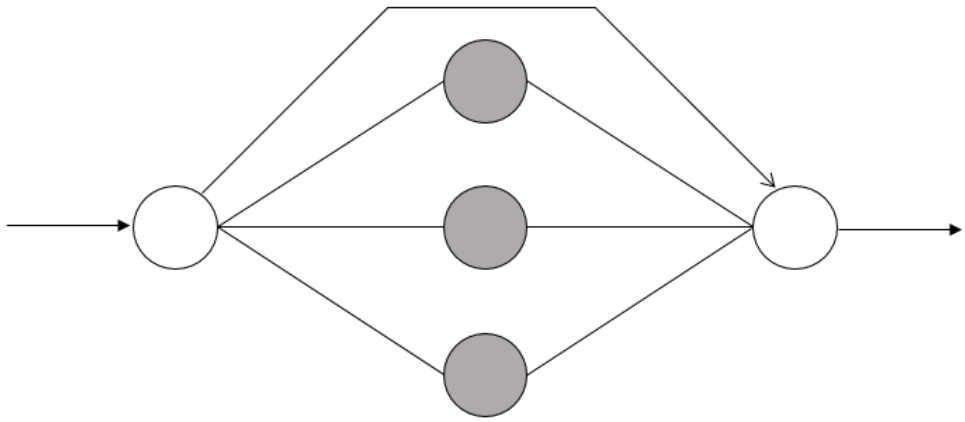


Figure 1.10: Bridge Neural Network

Interpolating such problem with a Feed Forward ANN can be rather inefficient, since the input is always going through the hidden layer where it is "distorted" by the Activation Function. An efficient solution to this problem would be to create a bypass connection directly from the input to the output layer, which performs a linear combination. This architecture is called a Bridged Neural Network. This architecture can be rather problem-specific, and since the connections are less regular than the Feed Forward, computing the gradient in this case can be rather difficult, and advanced training algorithms that are not architecture specific (e.g. the Neuron-by-Neuron, or the modified Levemberg-Marquardt) are needed. This architecture can be tested as well against the parity-N problem. For this particular network, the problem is solved with $N/2 + 1$ neurons.

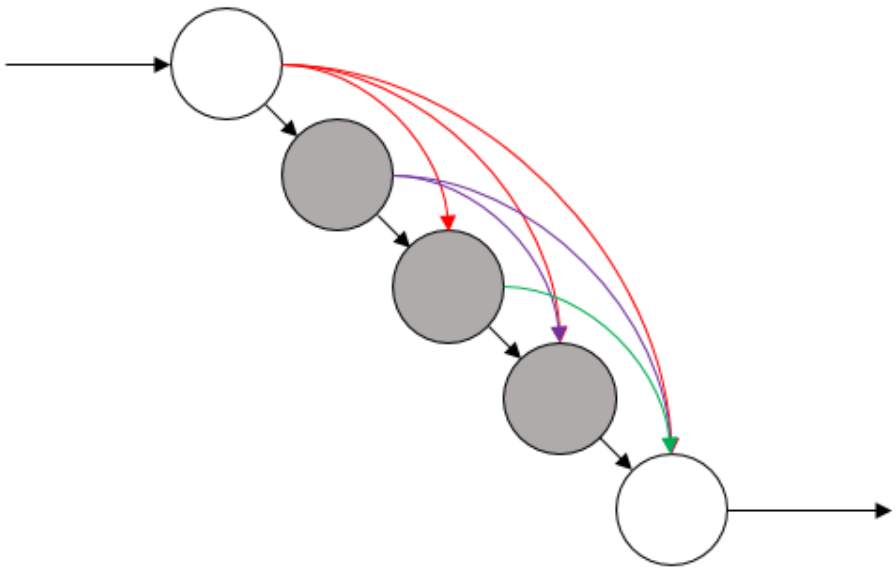


Figure 1.11: Fully Connected Cascade

Fully Connected Cascade

A particular case of Bridged Neural Network is the Fully Connected Cascade, or FCC. Such network is composed by an arbitrarily long series of hidden layers, each with a SINGLE neuron inside. Each neuron is connected to EVERY subsequent layer, thus creating a "cascade" from input to output.Apart from the advantages coming from the direct bridge from input to output (as in ordinary Bridged ANN), the FCC solves the parity-N problem with the lowest number of neuron when compared with both the MLP and the Bridged ANN. The solution can be obtained with $log_2(N + 1)$ neurons. As can be seen from Figure 1.12 and 1.13, the number of neurons for a FCC is lower, and rises more slowly, than the Bridged ANN and the MLP. A low number of neurons is very important to reduce computational costs if the FCC is to be

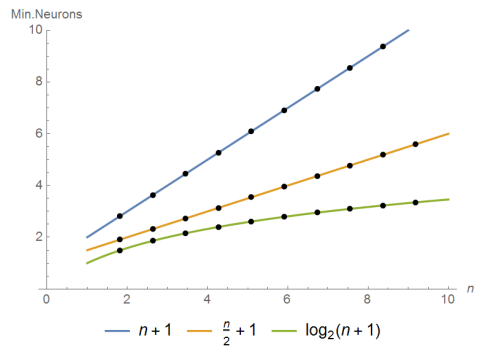


Figure 1.12: Neurons required for a parity-N problem

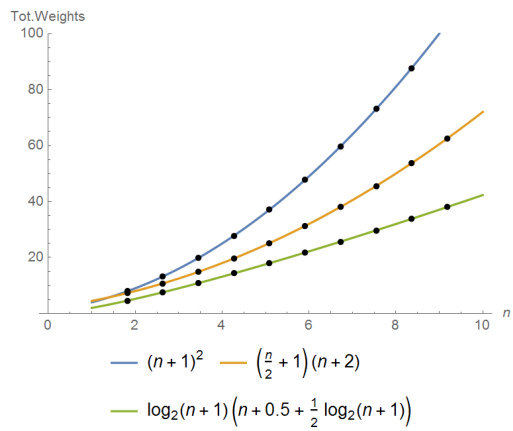


Figure 1.13: Weights required for parity-N problem

implemented in an embedded environment, since fewer neurons means fewer evaluation of the Activation Function. However, analogously as the Bridged ANN, training algorithms for this kind of ANN are more difficult than the ones available for the MLP.

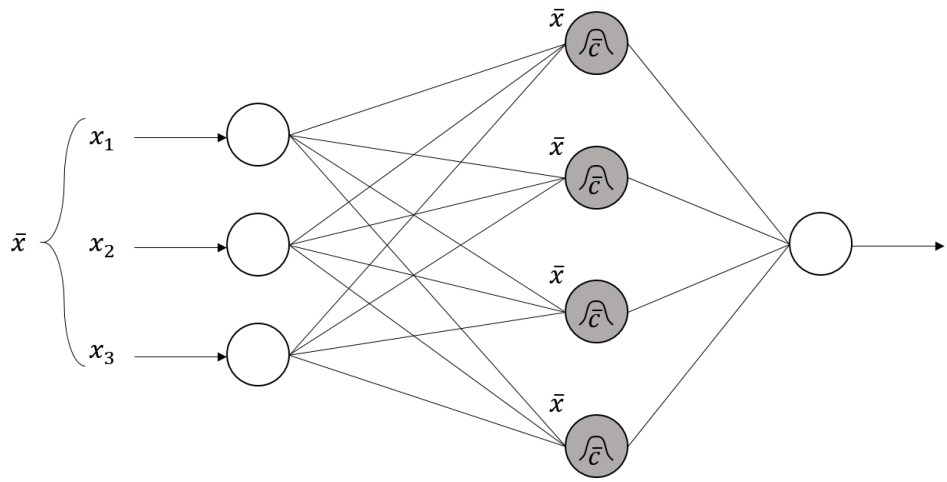


Figure 1.14: Radial Basis Function

Radial Basis Function Network

Radial Basis Function, or Radial Basis Neural Networks, often shorted as RBF, are not a particular architecture, but rather a Feed Forward Neural Network using a very specific kind of neuron in the hidden layer. These neurons features three differences from classic ones: the inputs are not linearly combined (in fact, all weights are unitary), the AF is a radial function, and the argument of the AF is the norm of the distance between the vector of the inputs and the vector of the AF center. The distinctive quality of a radial function is that it decrease (or increase) monotonically as the argument diverges from a central value. The function is characterized by the center, the shape of the curve, and a scale factor that express the width of the curve. A classic example, often used in literature, is the Gaussian function:

$$AF(\bar{x}) = exp\left(\frac{||\bar{x} - \bar{c}||}{r^2}\right) \tag{1.8}$$

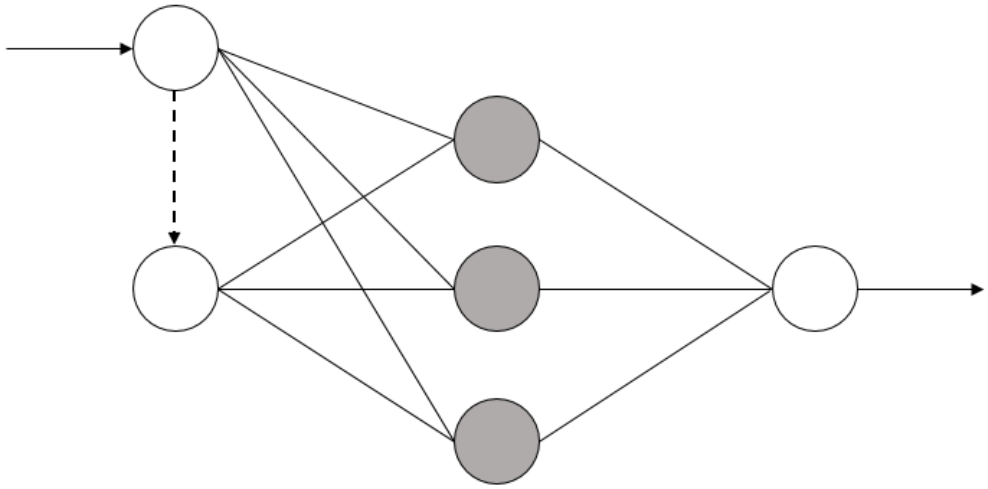


Figure 1.15: Time Delay Neural Network

Where $(\bar{x})$is the input vector (the same for all the neurons of the RBF), $(\bar{c})$ the centrum for the specific neuron, and r^2 is the squared radious of the gaussian curve. In fact, this particular ANN maps a set of gaussian curves (one for each neuron) in the multi-dimensional space of the input vector, and combines them linearly in the output layer. To train an RBF it is critical to choose the right number of neurons in the hidden layer. For this reason, sizing of the network is considered part of the training procedure as well. Techniques like incremental building and pruning are often used to size the RBF correctly. Then, the neurons are centered and scaled accordingly using classic backpropagation based algorithms.

Time Delay Neural Networks

The architectures shown by now shares the characteristic of being static, or in other words, constitute models where the present output(s) depends

solely on the present input(s). As it is well known, several systems needs to be represented by dynamic models, where the output is dependent on both the present inputs and the history of either past inputs or past outputs (also known as "state" of the system). To introduce this behavior to ANN, some kind of memory must be introduced in the architecture. The simplest possible form of memory is introducing a delayed propagation of the signal. This delay can appear either in form of feedback loops (internal or external), or in front of the ANN, between the inputs and the network. This second case is very useful due to the simplicity of implementation, and is commonly referred as Time Delay Neural Networks (TDNN). As shown in figure, the input entering the ANN goes through a delay line (slashed), and delayed replicas of inputs are sent to the ANN as independent ones. The network itself has no dynamic properties (in fact, often simple MLP are used), and the dynamic capabilities are reliant on the length of the delay line. The main advantage of the TDNN is that since the delay lines are present at the input only, every training algorithm compatible with a static ANN can be used to train a TDNN as well. This is because there are no weights that needs to be updated between the delay lines.

Fully-Recurrent Neural Network

If the delay line is inside the ANN, rather the in front of it, the ANN is by definition, dynamic. Several architectures have been proposed for a dynamic ANN, yet the simplest are usually the most useful in practical applications. A possible connection criterion is to connect the output of every neuron of the ANN to the input of all the neurons of the ANN (including the same neuron,

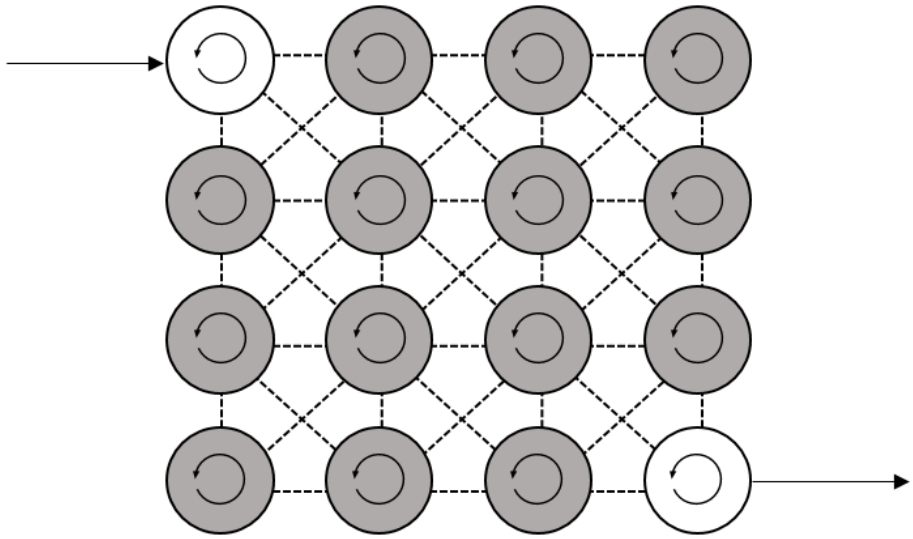


Figure 1.16: Fully Recurrent Neural Network

thus creating a self-feedback). This architecture is called Fully-Recurrent NN (FRNN) and features extreme dynamic flexibility. A known variation of such architecture is the Hopfield Network, which differs from this one since it lacks the self-feedback property. The dynamic propagation of the signal inside the FRNN (and in any other dynamic ANN) advances in time on a discrete scale: the output of all neurons is computed at a particular time step, and that output is used as input for the connected units at the following one. Training of a FRNN, and in general, of a dynamic ANN, is a complex task that requires specific algorithms. Simper network can be computed through a dynamic expansion of the classic Error Backpropagation algorithm (called Backpropagation through Time), but a FRNN should be trained with advanced techniques, like for instance the Real-Time Recurrent Learning algorithm. Both algorithms will be thoroughly explained in the

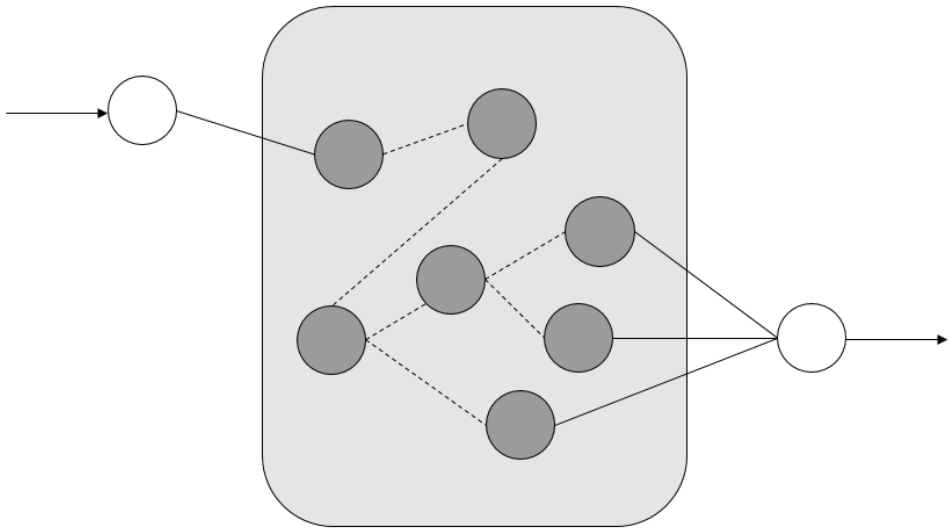


Figure 1.17: Echo-State Network

relative sections.

Echo-State Neural Networks

The dynamic capabilities of an ANN lies in the memory created by the delay lines found in the network connections. Analogously as the TDNN, it has been seen that the delayed connections can be useful even if unweighted. Still, the TDNN dynamic capabilities are very limited. An evolution of such architecture is the Echo-State Neural Network (ESN). This architecture is composed by two sections. The first one is called reservoir, and is composed by a set of neurons with the following characteristics: neurons are randomly connected, connections are sparse (about 1%), neurons are non-linear, a random quota of connections are delayed, and a random quota of neurons is

sent to the next section. The second section is an output layer composed by linear neurons. During usage, an ESN behaves analogously to a recurrent ANN with a particular connection criterion. The main difference however lies in the training of such network. Training experience on FRNN shown that the most important weight change, towards error minimization, happens in the output neurons (i.e. the linear ones). An ESN exploits such idea and leave open for change only the weights of the second section (i.e. the output neurons, also referred in literature as read-outs). Since the neurons under training are all linear, algorithms for ESN training are all based on linear regression techniques. This is very useful in case of online training. The network dynamic capabilities are proportional to the reservoir size. Practical implementation of such network in embedded systems benefits from the reduced computational costs in terms of training. The main drawback however, is the very high memory cost associated to the reservoir. Compared to a FRNN, an ESN need a larger number of neurons due to the sparse connection criterion.

1.2 Training Algorithms

Different algorithms are available in literature to train ANNs. The final goal of a training algorithm is to modify the ANN configuration (i.e. weights and biases) towards the maximum similarity with the function generating the training set. This is done by minimizing the error on a set of sample points that are called, conveniently, training set. Thus, training is a minimization problem: weights and biases are changed with the goal of getting an error as

low as possible on a set of points. This is a classic Non Linear Least Squares Problem. Solving such problem involves, generally, two major stages. The first one is to identify some kind of derivative of the error with respect to the network weights. Either this can be the simple gradient, a second-degree Hessian matrix, or the Jacobian matrix where the error is considered in vector form R^n where n is the size, in samples, of the training set. The second one is the update rule, which can go from simple gradient descent techniques to hybrid approaches like to one used in the Levemberg Marquardt algorithm. As for the architectures, the number of training algorithms in literature is high, and in this section, a selection of practical-oriented techniques will be described.

1.2.1 Error Backpropagation with Gradient Descent

The most intuitive way of minimizing the error operating on the network weights consist in expressing the derivative of the network error with respect to weight changes. Computing such derivative (and in general, the gradient) requires a major fundamental property that is enforced for ANN training: the transparency of the weight changes with respect to the ANN output. A perturbation on the weights must be visible in the output of the ANN. This concept, straightforward at first sight, implies that the ANN activation function must be *soft*. Computation of the error derivative for an ANN is a generalization of the computation for a single layer network.

- n neurons in the layer.
- m inputs to the layer.

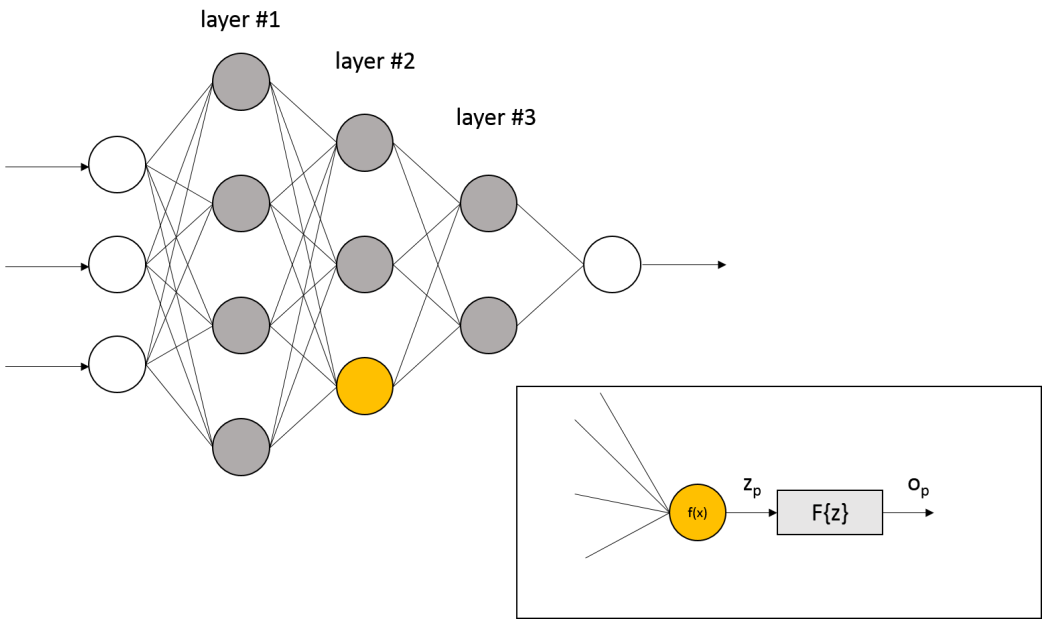


Figure 1.18: A FFNN with multiple hidden layers. Hidden neurons are connected undirectly to the output of the ANN through a generic non-linear function $F(z)$

- j is the index for the neurons.
- i is the index for the inputs.

$$net_j = \sum_{i=1}^{m} w_{i,j} x_i \tag{1.9}$$

$$o_j = f(net_j) = tanh(\frac{knet_j}{2}) \tag{1.10}$$

$$f'(o_j) = 0.5k(1 - o_j^2) \tag{1.11}$$

where k is the slope of the activation function for null *net* values, the error for the j-th neuron can e expressed as:

$$Error = \frac{1}{2}(o_j - d_j)^2 \tag{1.12}$$

Where d_j is the desired j-th output, and o_j j-th neuron output. The derivative of such error with respect to the weights of the neuron is:

$$\frac{\partial Error}{\partial w_{ij}} = (o_j - d_j)\frac{\partial f(net_j)}{\partial net_j} x_i \tag{1.13}$$

Then for each weight of the neuron, a derivative of the error with respect to weight change has been determined. Combined, this information gives the *gradient* of the error. The simplest approach to minimize the error would be to move in the direction of gradient descent. Thus updating the $\bar{w}$ as follows:

$$\bar{w}_{k+1} = \bar{w}_k - (\alpha \nabla w) = \bar{w}_k + \Delta w \tag{1.14}$$

Where α is a learning constant, and the components of the term Δw are, in fact:

$$\Delta w_{ij} = \alpha x_i (d_j - o_j) f'_j = \alpha x_i \delta_j \tag{1.15}$$

The term δ_j, expressing the product of the error and the slope of the AF, gives the name to this training algorithm for single neurons, and is called *delta learning rule*. This rule can be used as is for ANN composed by a single layer of neurons (thus with no connections between themselves). In practical cases however, the simplest structure interesting to be trained are ANN composed by three layers (input, hidden and output). This requires a generalization of the delta rule to multilayer architectures. The basic process consist in calculating the error at the output of the ANN, and then, *back-propagating* it through the network weights to every neuron in the network. This generalization of the delta rule is called *Error Backpropagation* (EBP). In a multilayer ANN like the one shown in Figure 1.18 all the hidden neurons are connected to the output through the rest of the network in front of them. For this reason, the error computed by Eq. 1.15 is valid only for the output neurons. Inner neurons see in front of themselves a generic non-linear function composed by the path towards the output. For the sake of simplicity, we will consider an ANN with a single output, and a single pattern.

The error for this ANN is:

$$E = \frac{1}{2}(o - d)^2 \tag{1.16}$$

The derivative of the error E with respect to the weight connecting the i-th neuron output to the j-th neuron input is:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E(o_j)}{\partial o_j} \times \frac{\partial o_j}{\partial net_j} \times \frac{\partial net_j}{\partial w_{ij}} = (d-o)F' \times f' \times o_i = \delta_j o_i \tag{1.17}$$

The computation of the δ_j term involves the evaluation of the $(d-o)F' \times f'$ term that is not very straightforward. However, expressing it as the derivative product:

$$\delta_j = \frac{\partial E(o_j)}{\partial o_j} \times \frac{\partial o_j}{\partial net_j} \tag{1.18}$$

And considering the total derivative:

$$\frac{dE(o_j)}{do_j} = \sum_{l \in L} \left(\frac{\partial E}{\partial net_l} \frac{\partial net_l}{\partial o_j} \right) = \sum_{l \in L} \left(\frac{\partial E}{\partial o_l} \frac{\partial o_l}{\partial net_l} w_{jl} \right) = \sum_{l \in L} (\delta_l w_{jl}) \tag{1.19}$$

Where L is the set of neurons in front of the j-th one. The third term of this recursive equation express the fundamental EBP equation: the δ term for a particular inner neuron can be computed by knowing the δ terms of the neurons in front of it. Thus, by starting from the output of the ANN, it is possible to propagate the δ term by combining Eq. 1.18 and Eq. 1.19:

$$\delta_j = \sum_{l \in L} (\delta_l w_{jl}) \times f' \tag{1.20}$$

Analogously as the delta rule training for single neurons, the update rule is given by Eq. 1.15.

By now, two main assumptions have been made.

- The ANN has a single output.
- The ANN is trained by a single element from the dataset.

The first assumption is not mandatory, but is a general good practice to keep the ANN with single outputs. The EBP by its nature is scalable to multiple outputs by combining *delta* terms from different outputs linearly. On the other hand, the second assumption requires more attention. Updating the weights after each sample, in the way described in this paragraph, is defined as *online training*. A different, and more converging approach, would be to compute a cumulative weight change matrix Δw_T obtained by combining the partial weight change matrices Δw_p computed for each p-th sample in the dataset, and only after the final sample has been used, compute the weight update. Thus, over a dataset of P samples:

$$\Delta w_T = \sum_{p=1}^{P} \Delta w_p \tag{1.21}$$

This approach is defined as *batch training*. Both approaches have advantages and drawbacks. On-line learning is used for dynamic environments that provide a continuous stream of new patterns, and is useful if the ANN is supposed to learn continuously an unknown function that changes over time. In general, on-line training does not converge to a single point in weight space, but oscillates around the minimum of the error function. Batch learning on the other hand will converge more stably to a local minimum since each update is performed based on all patterns. The two methods are not mutually exclusive. An ANN could be previously trained in batch mode, and then it can be updated on the run through on-line training. Or small segments of live acquired data could be packed in a single "mini-batch".

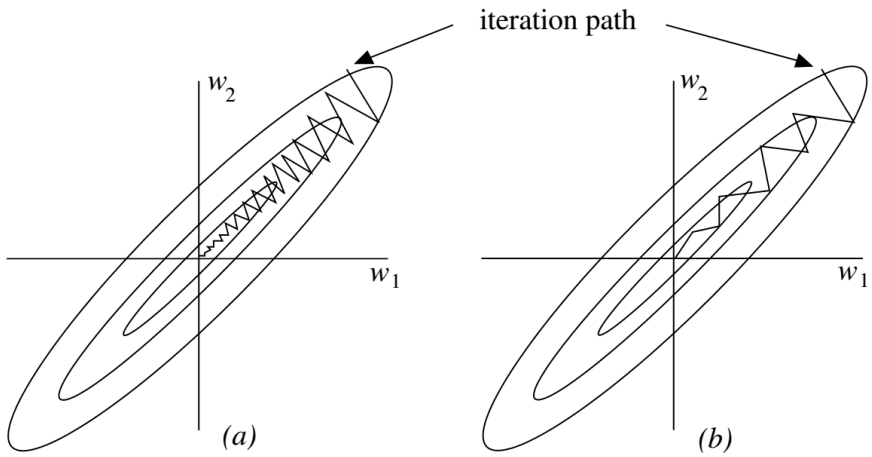


Figure 1.19: Effect of momentum on training of a ANN. Trajectory oscillates without momentum (left) and converges much faster with momentum (right)

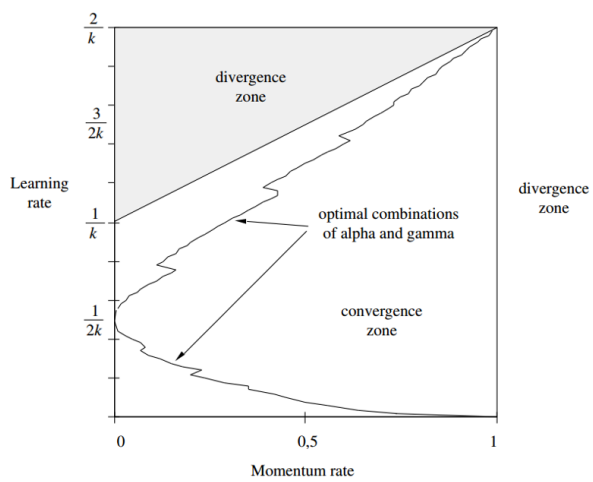


Figure 1.20: Possible combinations of α and γ for the minimization of a kx^2 function.

Momentum

A very common setup that may happen during error minimization is an error function shaped like a narrow valley. A bidimensional example (thus with only 2 parameters to be optimized) can be seen in Figure 1.19. Indeed, the optimal approach would be to point the search direction near the center of the ellipsoid, however the search direction pointed by the gradient is different. A solution to this problem is to introduce an inertial coefficient called *momentum*. At every iteration the correction matrix for weights is computed, but instead of being applied directly, it is averaged with the values of the previous directions. Since the step-size is proportional to the learning constant α, so will be the amplitude of the oscillations, as can be seen from Figure 1.19 (left). An inertial component can be added simply to the weight update procedure:

$$\Delta w_{k+1} = (1 - \gamma)\Delta w_k + \gamma \Delta w_{k-1} \tag{1.22}$$

The γ term is called momentum constant. Fine tuning the learning constant α and the momentum constant γ can yield faster convergence rate as shown in Figure 1.19 (right). The optimal combination between learning and momentum is problem specific and is generally found by trial and error. However [Rojas, 2013] propose an analysis of a simple quadratic function minimization of the family kx^2, which is representative of the error minimization for a linear neuron. Indeed, as can be seen in Figure 1.20, an optimal combination of the two constants exist, and learning rates too elevated can not be compensated by any momentum constant, causing divergence.

Weigths Initialization

Common sense suggest that weight initialization can be a critical stage in ANN training. A common technique when using sigmoidal AF is to set the weigts of the ANN with uniform probability in an interval $[-a, a]$. Since the expected value of such weights is zero, it is sensible to assume that the total input of each node of the network will be zero as well. The derivative of the sigmoid AF reaches the maximum value of 0.25 in this point. From a first analysis, this is a good thing, since the backpropagation of the *delta* parameter is proportional to the derivative of the AF f' (as in Eq. 1.20). However, if the weights are zero, or very small, this will lead to a reduction of the backpropagated error from the output to the hidden layers. Thus, a very small value of a can lead to very slow convergence. On the other hand, large values can quickly drive the AF near the saturation points where the derivative is zero. An almost unitary value can give a good trade-off between minimizing the two factors [Rojas, 2013].

Resilient Backpropagation

A modified version of the EBP, referred as Resilient Backpropagation (or *Rprop*) was proposed by [Riedmiller and Rprop, 1994,Riedmiller and Braun, 1992]. Each weight has a correction factor associated to it, which is randomly initialized in the beginning of the training procedure. After each epoch, the gradient is computed, and the sign of the partial derivative for each weight is noted. For each weight, if the sign of the partial derivative is the same of the previous iteration, the correction factor is multiplied by $\eta_+ > 1$, else, by $\eta_- < 1$. The correction factor is carried on epoch by epoch, and is applied to

the weights after being multiplied by the sign of the partial derivative. This approach has the great advantage of not suffering from learning slow-down due to elevated *net* values (i.e low AF derivatives). Empirical values for η_+ and η_- are, respectively, 1.2 and 0.5.

Weights Decay

Sizing an ANN correctly yields the best generalization capabilities, as shown in previous chapters of this work. Indeed, obtaining such delicate balance is difficult considering the strong influence that the specific problem has on the training outcome. Limiting the number of neurons through an algorithmic approach [Hassibi and Stork, 1993,LeCun et al., 1989] is a technique referred as *pruning* the ANN, but it's not the only approach possible. A possible way to contain the network complexity (and thus maximizing the generalization capabilities) is to limit the magnitude of the weights [Moody et al., 1995]. This can be achieved by adding a decay term that can prevent the weights from growing too large unless that trend presents itself regularly. Adding a cost function that penalizes large weights can perform such task easily:

$$E(\mathbf{w}) = E_0(\mathbf{w}) + \frac{1}{2}\lambda \sum_i w_i^2 \tag{1.23}$$

Where E_0 is a user choice way of measuring error (e.g. sum of squared errors) and λ is a parameter proportional to the penalization for large weights, and $\mathbf{w}$ is of course the weight vector/matrix. If the gradient descent is used as learning rule, the time trend of the weights will be expressed by the following differential relationship:

$$\dot{w}_i \propto -\frac{\partial E_0}{\partial w_i} - \lambda w_i \tag{1.24}$$

The term $\dot{w}_i$ is expressed as a derivative over time of the network weights. It has been shown in [Moody et al., 1995] that weight decay can suppress irrelevant components of the weight vector, and minimize the effect of added noise on the target vector (effectively exploiting at full ANN noise filtering capabilities).

1.2.2 Conjugate-Gradient-Method based Algorithms

Classic EBP based approaches uses gradient descent as learning rule. Although this approach can be refined with several tweaks to reduce oscillation and increase convergence rate, this is not the best strategy to achieve fast convergence. In fact, even if the direction for fastest function decrease is the negative gradient, this is seldom the optimal route to the minimum. An example of such behavior has been already presented in Figure 1.19. Indeed, it is theoretically possible to move towards the optimum direction with a reduced set of *A-orthogonal*, or *conjugate* directions. In this section the Conjugate-Gradient-Method (CGM) for ANN training will be explained. However, to fully understand how it works on a *non-linear* problem, an introduction on how this method works on linear problems is mandatory.

Conjugate Gradient on Linear Problems

A very intuitive introduction to this method is given in [Shewchuk, 1994], where the Conjugate-Gradient-Method (CGM) is used to solve a simple linear problem:

$$\mathbf{A}x = b \tag{1.25}$$

The solution of the problem can be found, assuming that A is positive-definite and symmetric, through the scalar quadratic function:

$$f(x) = \frac{1}{2}x^T\mathbf{A}x - b^T x + c \tag{1.26}$$

Some quantities must be defined for the following explanation: the *error* $e_i = x_i - x$ express the distance from the solution. The *residual* $r_i = b - \mathbf{A}x_i$ is the projected error in the b space. It can be seen that $r_i = -\mathbf{A}e_i = -f'(x_i)$ where $f'(x_i)$ is the gradient of the function in the x_i point. Minimizing Eq. 1.26 consist in finding a series of $x(i)$ that converges to the function minimum (which will be unique if A is not singular). Each new point is found by adding a movement vector to the previous one. The vector is composed by two parts: the direction and the step size. Solution through gradient descent would use the gradient as the direction (that, for a linear problem, points in the same direction of the residuals), and the step size necessary to zero the directional derivative of the error:

$$\alpha_i = \frac{r_i^T r_i}{r_i^T \mathbf{A} r_i} \tag{1.27}$$

$$x_{i+1} = x_i + \alpha_i r_i \tag{1.28}$$

This approach, as shown in Fig. 1.19, can give several oscillations. A better combination of search direction and step size is to use *Conjugate Directions Method* (CDM). Two conditions are imposed:

- The directions are $A - orthogonal$, thus $d_i^T \mathbf{A} d_j = 0$ for $i \neq j$

- The step size *alpha* will ensure $A - orthogonality$ between the current direction and the vector at the next iteration, thus $d_i^T \mathbf{A} e_{i+1} = 0$. This is equivalent to imposing a line search in the direction d_i for the minimum vale of the function.

A set of conjugate directions can be obtained by Gram-Schmidt Conjugation. Given a set of linearly independent vectors $u_0, u_1, ..., u_{n-1}$, the i-th direction d_i can be constructed from u_i and subtracting any components that are not *A-orthogonal* to the previous directions.

$$d_i = u_i + \sum_{k=1}^{i-1} \beta_{ik} d_k \tag{1.29}$$

$$b_{ij} = -\frac{u_i^T \mathbf{A} d_j}{d_j^T \mathbf{A} d_j} \tag{1.30}$$

The step size α, on the other hand, can be obtained by:

$$\alpha_i = \frac{d_i^T r_i}{d_i^T \mathbf{A} d_i} \tag{1.31}$$

This algorithm can be consistently improved on the front of computational costs by using the residuals as Gram-Schmidt base for direction conjugation (thus using $u_i = r_i$). The reasons of this choice are several. First, the residuals are, by definition, orthogonal to the previous search direction, and for this reason, they will produce a new direction unless the residual is zero (i.e. the problem is solved.) Moreover, each residual is a combination of the previous residual and the former search direction. In other words, the space $\mathcal{D}_i$ where the previous search direction d_{i-1} lies can be combined with the transformed subspace $A\mathcal{D}_i$ to create the subspace of the next search di-

rection $\mathcal{D}_{i+1}$. A subspace obtained by repeatedly applying a matrix is called a *Krylov Subspace*.

Since $A\mathcal{D}_i \in \mathcal{D}_{i+1}$, the fact that r_{i+1} is perpendicular to $\mathcal{D}_{i+1}$ implies that it is *A-orthogonal* to $\mathcal{D}_i$: the base for the new search direction to be used in Gram-Schmidt conjugation is already A-orthogonal to all former directions (the space $\mathcal{D}_i$) except the actual one d_i. It is no longer necessary to store all the previous search vectors to ensure A-orthogonality. This reduce the complexity per iteration from $\mathcal{O}(n^2)$ to $\mathcal{O}(m)$, where m is the non-zero entries of A. The final relations defining this algorithm then are:

$$d_0 = r_0 = b - \mathbf{A}x_0 \tag{1.32}$$

$$\alpha_i = \frac{r_i^T r_i}{d_i^T \mathbf{A} d_i} \tag{1.33}$$

$$x_{i+1} = x_i + \alpha_i d_i \tag{1.34}$$

$$r_{i+1} = r_i - \alpha_i \mathbf{A} d_i \tag{1.35}$$

And Gram-Schmidt reduction is performed via:

$$\beta_{i+1} = \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i} \tag{1.36}$$

$$d_{i+1} = r_{i+1} + \beta_{i+1} d_i \tag{1.37}$$

This algorithm is commonly referred as *Conjugate-Gradients-Method*, or, CGM.

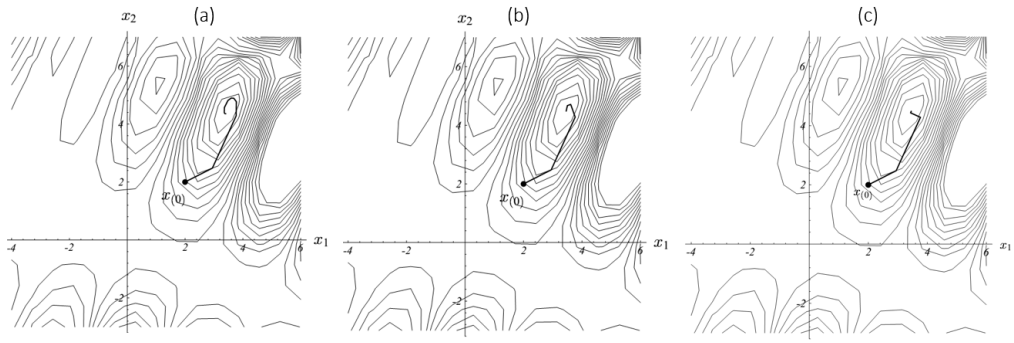


Figure 1.21: Convergence of the CGM on a non-linear problem. a) Fletcher-Reeves b) Polak-Riebere c) Powell

Non-Linear CGM

By now the use of CGM to solve a linear problem $\mathbf{A}x = b$ through the minimization of the quadratic expression Eq. 1.26 has been examined. This method can be used to solve non-linear problems as well, by modifying some key concepts.

- The residual is always set as the negation of the gradient: $r_i = -f'(x_i)$
- Direction search is performed through modified Gram-Schmidt conjugation.
- The step size is found through numerical line search methods.

Several solutions exist for the first point. The most common one, known as Fletcher-Reeves formula [Dai and YUAN, 1996,Fletcher and Reeves, 1964], suggest to find β as:

$$\beta_{i+1} = \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i} \tag{1.38}$$

and converges if the starting point is sufficiently closed to the desired minimum. A faster convergence, although at risk of stability, can be obtained with the Polak-Riebere formula [Polak, 1971] (and modified in [Zhang et al., 2006, Grippo and Lucidi, 1997]):

$$\beta_{i+1} = \frac{r_{i+1}^T (r_{i+1} - r_i)}{r_i^T r_i} \tag{1.39}$$

One last improvement that can be applied to CGM for non-linear problems is the periodic restart. Indeed, CGM can only generate n conjugate directions in a n-dimensional space. So it makes sens to restart it every n iterations, especially if n is small. Restarting the algorithm means reset the actual search direction as the negative of the gradient. A quantitative method to decide when the restart should occour was proposed by [Powell, 1977]. The restart will occour if there is little orthogonality between the current and the former residuals, or:

$$|r_{k-1}^T r_k| \geq 0.2||r_k||^2 \tag{1.40}$$

If this condition is satisfied, the search direction is reset to the negative gradient.

1.2.3 Weight Perturbation & Node Perturbation

A very straightforward strategy to avoid analytic computation of the gradient function is *weight perturbation* [Flower and Jabri, 1993, Jabri and Flower, 1992]. An ANN is initialized with an initial weight matrix $\mathbf{w}$. The error for that particular weight configuration $E(\mathbf{w})$ is computed. A small perturbation

β is then applied to the (i,j)-th weight, chosen at random, creating the a new point in the weight space **w'**. Error in the new point $E(\mathbf{w'})$ is computed, again. The update rule for the i-th weight is:

$$\Delta w_{i,j} = -\alpha \frac{E(\mathbf{w'}) - E(\mathbf{w})}{\beta} \tag{1.41}$$

This discrete approximation can be very useful in embedded applications, since the learning algorithm can be implemented with very low computational complexity. An interesting alternative, known as *node perturbation*, proposed by [Rojas, 2013] is to perturb the output o_i of the i-th neuron by Δo_i instead of a random weight. It is possible then to compute the difference $E - E'$ between the old and the new error. If the difference is positive (i.e. $E' < E$) then the new output of the neuron $o_i + \Delta o_i$ is desirable to train the network. In case of sigmoid AF, the weighted neuron input to achieve such output is:

$$\sum_{k=1}^{m} w'_k x_k = s^{-1}(o_i + \Delta o_i) \tag{1.42}$$

Where s^{-1} is the inverse AF. The new weights can be obtained from the following equation:

$$w'_k = w_k \frac{s^{-1}(o_i + \Delta o_i)}{\sum_{k=1}^{m} w_k x_k} \tag{1.43}$$

Weights are updated in proportion to their relative size. To break this systematic symmetry, a stochastic factor can be included, or the two techniques (weight perturbation and node perturbation) can be applied alternatively.

1.2.4 Levenberg–Marquardt

The Levenberg–Marquardt algorithm, proposed in [Marquardt, 1963] aims to solve non-linear least squares problems through a stable and fast approach. This algorithm is extremely popular in ANN training since it is very well suited for small and medium sized networks, and the partial derivatives of the gradient can be found analytically through the EBP technique. Differently from the EBP training however, this algorithm merges the first order approach of the steepest descent with the quadratic approximation of the Gauss-Newton algorithm. The Gauss-Newton algorithm approximates the error function with a quadratic one, and if such approximation is accurate, can converge quite rapidly. However, this is seldom the case, and afar from such approximation the Gauss-Newton algorithm can be very unstable. The steepest descent on the other hand features a slower yet more stable convergence. This algorithm merges the two approaches and try to take advantage of both by using a combined training process: when the search is being performed in areas with complex curvatures the method use a steepest descent approach, and as soon as a quadratic approximation is suitable, it switches to Gauss-Newton for faster convergence. Few definitions are needed to follow the non-linear least square problem at hand:

- p is the index for the points of the training set, from 1 to P.
- m is the index of output neurons, from 1 to M.
- i and j are both used as weights indexes, from 1 to N.
- k is the index for the epochs.

We recall the definition of total error on multiple points as:

$$E = \frac{1}{2} \sum_{p=1}^{P} \sum_{m=1}^{M} e_{p,m}^2 \tag{1.44}$$

Where:

$$e_{p,m} = d_{p,m} - o_{p,m} \tag{1.45}$$

The error $e_{p,m}$ is expressed as difference between desired $d_{p,m}$ and actual $o_{p,m}$ output, on the $p-th$ point for the $m-th$ output neuron. The gradient of such error with respect to the network weights is:

$$g = \left[\frac{\partial E}{\partial w_1} \frac{\partial E}{\partial w_2} ... \frac{\partial E}{\partial w_N} \right]^T \tag{1.46}$$

Newton & Gauss-Newton algorithms

The Newton algorithm search the minimum of a function through quadratic approximation. The formal proof of this algorithm is beyond the scope of this work, and only the operative formulas will be given. The Newton algorithm requires the computation of the *Hessian* matrix $\mathbf{H}$, whose entries are the mixed second derivatives of the total error Eq. 1.44 with respect to different weights:

$$\mathbf{H} = \begin{pmatrix} \frac{\partial^2 E}{\partial w_1^2} & \frac{\partial^2 E}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_1 \partial w_N} \\ \frac{\partial^2 E}{\partial w_2 \partial w_1} & \frac{\partial^2 E}{\partial w_2^2} & \cdots & \frac{\partial^2 E}{\partial w_2 \partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial^2 E}{\partial w_N \partial w_1} & \frac{\partial^2 E}{\partial w_N \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_N^2} \end{pmatrix} \tag{1.47}$$

The update rule for the Newton algorithm is:

$$w_{k+1} = w_k - \mathbf{H}_k^{-1} g_k \tag{1.48}$$

This requires the computation of the inverse Hessian matrix, whose elements are very complicate to obtain. Indeed, it is possible to approximate the Hessian matrix and simplify the procedure. We define the *Jacobian* matrix $\mathbf{J}$ as:

$$\mathbf{J} = \begin{pmatrix} \frac{\partial e_{1,1}}{\partial w_1} & \frac{\partial e_{1,1}}{\partial w_2} & \dots & \frac{\partial e_{1,1}}{\partial w_N} \\ \frac{\partial e_{1,2}}{\partial w_1} & \frac{\partial e_{1,2}}{\partial w_2} & \dots & \frac{\partial e_{1,2}}{\partial w_N} \\ \dots & \dots & \dots & \dots \\ \frac{\partial e_{1,M}}{\partial w_1} & \frac{\partial e_{1,M}}{\partial w_2} & \dots & \frac{\partial e_{1,M}}{\partial w_N} \\ \dots & \dots & \dots & \dots \\ \frac{\partial e_{P,1}}{\partial w_1} & \frac{\partial e_{P,1}}{\partial w_2} & \dots & \frac{\partial e_{P,1}}{\partial w_N} \\ \frac{\partial e_{P,2}}{\partial w_1} & \frac{\partial e_{P,2}}{\partial w_2} & \dots & \frac{\partial e_{P,2}}{\partial w_N} \\ \dots & \dots & \dots & \dots \\ \frac{\partial e_{P,M}}{\partial w_1} & \frac{\partial e_{P,M}}{\partial w_2} & \dots & \frac{\partial e_{P,M}}{\partial w_N} \end{pmatrix} \tag{1.49}$$

The $\mathbf{J}$ takes into account the partial derivatives of the error, for each output, and for each pattern, with respect to the weight changes. Indeed it can be seen, by applying Eq. 1.46 to Eq. 1.44 that $g = \mathbf{J}e$. The approximation proposed by Newton-Gauss method is:

$$\mathbf{H} \approx \mathbf{J}^T \mathbf{J} \tag{1.50}$$

That involves considering as zero some mixed derivatives from the Newton method. Then, the update rule of the Newton method Eq. 1.48 can be modified into:

$$w_{k+1} = w_k - (\mathbf{J}_k^T \mathbf{J}_k)^{-1} \mathbf{J}_k e_k \tag{1.51}$$

Which is the update rule of the Newton-Gauss algorithm. The great advantage of this algorithm is that only first order partial derivatives must be computed. Still, this algorithm is limited to quadratically approximate functions, and tends to diverge when more functions more complexly shaped are involved. In mathematical terms, this manifests as a non-invertible $\mathbf{J}^T\mathbf{J}$ matrix.

Levenberg–Marquardt Algorithm

In order to ensure that the $\mathbf{J}^T\mathbf{J}$ matrix is invertible, a modification to the Hessian approximation is performed:

$$\mathbf{H} \approx \mathbf{J}^T\mathbf{J} + \mu\mathbf{I} \tag{1.52}$$

Where $\mathbf{I}$ is an identity matrix and μ is a called *combination coefficient*. Indeed, the diagonal elements of the approximated $\mathbf{H}$ are now all greater than zero, thus the inversion is always possible. The new update rule for the *Levemberg-Marquardt* (LM) is then:

$$w_{k+1} = w_k - (\mathbf{J}_k^T \mathbf{J}_k + \mu\mathbf{I})^{-1} \mathbf{J}_k e_k \tag{1.53}$$

The μ coefficient is responsible from switching from Steepest Descent to Newton-Gauss behavior. Large values of μ approximate the update towards simple steepest descent (the term $\mathbf{J}^T\mathbf{J}$ is negligible) and is used in the first part of the solution search. Gradually μ is reduced during the training to

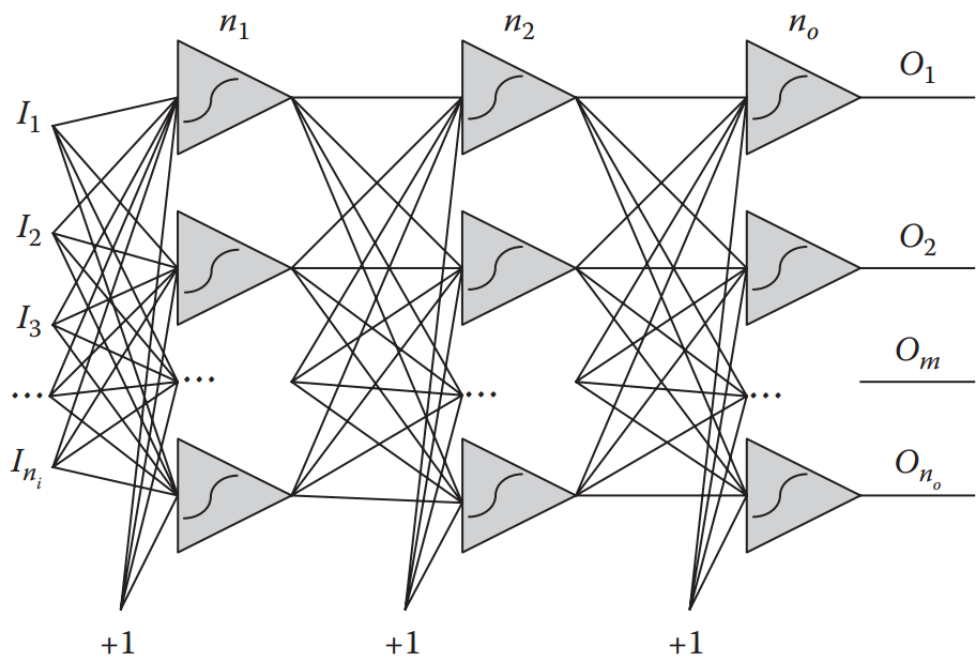


Figure 1.22: A simple multi-layer Feed Forward ANN

speed up convergence in the proximity of the solution. In practical algorithms, a check on the error trend is performed: if the error goes down from one iteration to the other, the quadratic approximation on the total error is accurate and the coefficient μ can be reduced. On the other hand, if the error increases, it means that the curvature of the function is not efficiently approximated by a quadratic, and the μ coefficient is increased to steer the direction towards gradient descent.

Jacobian Computation

Analogously as gradient based methods, all the work revolves around calculating the partial derivatives of the error with respect to the network weights.

In EBP it used to compute the gradient of the total error, in LM is used to compute the Jacobian (which takes into account all the patterns at the same time). The elements of the Jacobian can be computed through:

$$\frac{\partial e_m}{\partial w_{i,j}} = y_j \delta_{m,j} \tag{1.54}$$

The term $\delta_{m,j}$ is defined at the output of the ANN and is back-propagated through a simple iterative process. Take into account the ANN shown in Fig. 1.22. During forward propagation, the outputs y_j and the slopes s_j of all neurons are calculated. For output m, the error and the initial δ is computed:

$$e_m = d_m - o_m \tag{1.55}$$

$$\delta^3_{m,j} = s^3_j \tag{1.56}$$

The superscript ^{3} is used to identify the layer (in this case the third, output one). The δ is then propagated, through the weights, to the previous layers:

$$\delta^2_{m,j} = s^2_j \times \sum_{i=1}^{nn^3} \delta^3_{m,i} w_{j,i} \tag{1.57}$$

$$\delta^1_{m,j} = s^1_j \times \sum_{i=1}^{nn^2} \delta^2_{m,i} w_{j,i} \tag{1.58}$$

Where nn^2 and nn^3 are the neurons in the second and third layer respectively. Put in words, Eq. 1.57 and 1.58 express that: the δ parameter is characteristic of a neuron, for a given *m-th* output. It is propagated backward

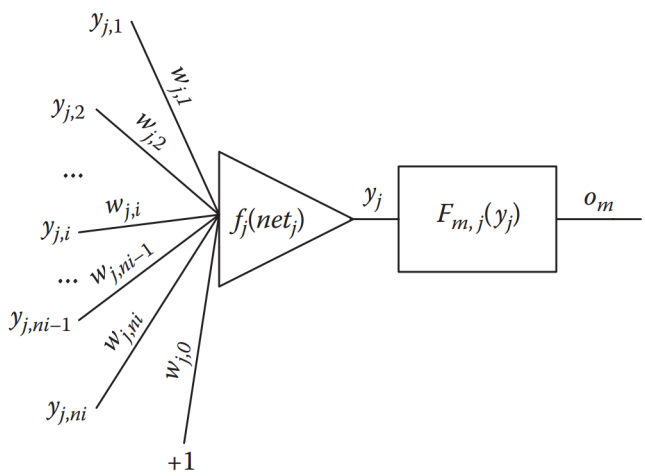


Figure 1.23: Schematic representation of a Neuron and the concept of node. It can be either $y_{j,i}$, meaning the $i-th$ input of neuron $j-th$, or y_j, meaning the output of neuron $j-th$. The $F_{m,j}(y_j)$ is the non-linear relationship between the neuron output y_j and the network output o_m

.

to the previous neurons by multiplying it by the network weights. Converging connections are summed, and the δ of the neuron is found by multiplying the value by the slope. Once all δ values of the ANN have been computed, the Jacobian elements can be computed by Eq. 1.54.

1.2.5 Neuron-by-Neuron Algorithm

The recently developed Neuron-by-Neuron Algorithm (NBN) [Wilamowski, 2009, Wilamowski et al., 2008] is based on the LM algorithm, but tries to overcome some bottlenecks that are present in it. Two main difference are found: first, the back-propagation procedure is generalized for arbitrarily connected ANN. Second, an alternative forward-only training with critical

optimizations in terms of computational costs is introduced. An extract of the work presented in [Wilamowski, 2009] is discussed in this section. First, let us introduce some indices that are necessary for the discussion.

- p is the index of patterns, from 1 to np, were np is the number of patterns (i.e. data points in the training set).
- m is the index of output neurons, from 1 to no, where no is the number of output neurons.
- j and k are the index of neurons, from 1 to nn, where nn is the number of neurons.
- i is the index of neuron inputs, from 1 to ni, where ni is the number of inputs and may vary for different neurons.

The total error over the training set is:

$$E = \frac{1}{2} \sum_{p=1}^{np} \sum_{m=1}^{no} e_{p,m}^2 \tag{1.59}$$

Where, as for the LM algorithm:

$$e_{p,m} = o_{p,m} - d_{p,m} \tag{1.60}$$

To understand the following discussion correctly, the concept of *node* must be introduced. Given a Neuron as the one shown in Figure 1.23 it can be seen that with $y_{j,i}$ and y_j we can either refer to one of the specific inputs of the neuron, or the output of it. Indeed, such neuron can exist anywhere inside the ANN. It could be in the input layer (and in this case,

its inputs would be the network inputs) or in the output layer (and in this case, its outputs would be the network outputs). The update rule for the NBN algorithm is the same used for the LM (Eq. 1.53), thus is necessary to define all the elements of the *Jacobian* matrix $\mathbf{J}$ (Eq. 1.49). In the NBN Algorithm, a different concept is introduced to define $\delta_{m,j}$. This quantity could be interpreted as the gain between the *net* input of neuron j and the network output m. Indeed this concept could be extended, and the notation $\delta_{k,j}$, as the gain between neurons j and k:

$$\delta_{k,j} = \frac{\partial F_{k,j}(y_j)}{\partial net_j} = \frac{\partial F_{k,j}(y_j)}{\partial y_j}\frac{\partial y_j}{\partial net_j} = F'_{k,j}s_j \tag{1.61}$$

Where k and j are neuron index, and the term $F_{k,j}(y_i)$ is the non-linear function between the output nodes of neurons k and j. Of course, $\delta_{k,k} = s_k$. All the $\delta_{k,j}$ can be organized in a $\boldsymbol{\delta}$ matrix. If no backward connections exist in the network, the matrix will be triangular. The procedure to define $\boldsymbol{\delta}$ will be initially given for a Fully Connected Network. It can be seen that any arbitrarily connected ANN can be seen as such by removing some connections (i.e. setting to zero some weights). The procedure goes as follows: For each j-th neuron, set $\delta_{j,j} = s_j$. For connections between neurons, use the following equation:

$$\delta_{k,j} = \delta_{k,k}\sum_{i=j}^{k-1} w_{i,k}\delta_{i,j} \tag{1.62}$$

In case the ANN has 4 neurons, the $\boldsymbol{\delta}$ could be as follows:

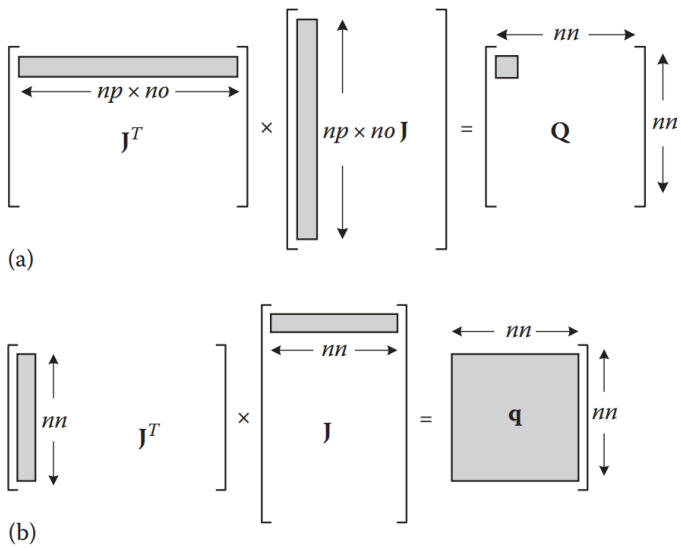


Figure 1.24: Different ways to perform matrix-matrix multiplication for the $\mathbf{J}^T\mathbf{J}product$

.

$$\boldsymbol{\delta} = \begin{pmatrix} s_1 & 0 & 0 & 0 \\ \delta_{2,1} & s_2 & 0 & 0 \\ \delta_{3,1} & \delta_{3,2} & s_3 & 0 \\ \delta_{4,1} & \delta_{4,2} & \delta_{4,3} & s_4 \end{pmatrix} \tag{1.63}$$

The $\boldsymbol{\delta}$ is directly used to compute the $\mathbf{J}$. The last no rows of the matrix are used for computation. Supposed that the ANN has two outputs ($no = 2$). The last two rows of $\boldsymbol{\delta}$ are processed in te following way:

$$\begin{pmatrix} \delta_{3,1} \times \boldsymbol{y_1} & \delta_{3,2} \times \boldsymbol{y_2} & s_3 \times \boldsymbol{y_3} & 0 \times \boldsymbol{y_4} \\ \delta_{4,1} \times \boldsymbol{y_1} & \delta_{4,2} \times \boldsymbol{y_2} & \delta_{4,3} \times \boldsymbol{y_3} & s_4 \times \boldsymbol{y_4} \end{pmatrix} \tag{1.64}$$

Where $\mathbf{y}_j$ is the vector of j-th neuron inputs (variable width according to the specific neuron). The matrix shown in Eq. 1.64 covers no of the $no \times np$ rows of $\mathbf{J}$. This procedure can be repeated for each *p-th* pattern to compute the full Jacobian. Computing the Jacobian row-wise has a major advantage that is exploited by the NBN algorithm. The $\mathbf{J}^T\mathbf{J}$, necessary to approximate the Hessian matrix, can be carried on in two ways. The first one is the classic row-by-column, shown in Figure 1.24 (a), where basically a single element of $\mathbf{H}$ is computed. Since the Jacobian *must* be computed row-wise, and a complete column is necessary to compute the product, a complete Jacobian is needed to start computing the product. On the other hand, computing the product column-by-row, as shown in Figure 1.24 (b), generates a partial matrix q, that must be summed element-wise to the other partial matrices to obtain the final Hessian. In the end, the number of operations is the same. However, column-by-row product requires a single *row*, and rows starts to be available as soon as the Jacobian matrix starts to be computed. Two advantages comes from this. First, the computation is faster. Second, the Jacobian rows relative to previous patterns does not need to be stored. In fact, this algorithm can be virtually used with an arbitrarily large training set without memory issues.

1.2.6 Real-Time Recurrent Learning

All the training algorithms shown so far are useful to train *static* ANN. If for some reasons the ANN features a dynamic behavior (i.e. feedback ando/or delayed connections are present inside the architecture) specific algorithms to train recurrent ANN should be used. Indeed, it is possible to represent

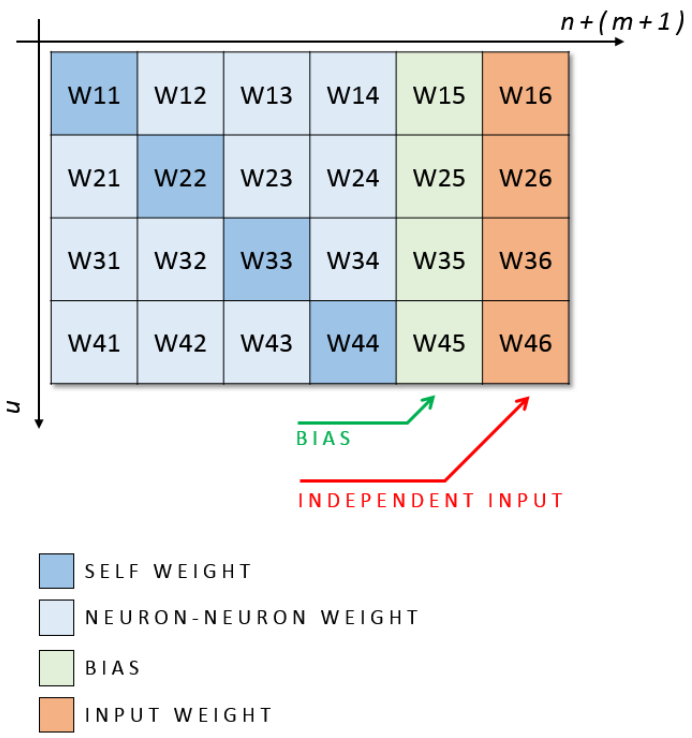


Figure 1.25: Weight matrix **W** of a recurrent ANN to be trained with RTRL Algorithm.

.

any dynamic ANN by a larger, unfolded, static ANN. This is the base of the Backpropagation Through Time algorithm (BTT), that allows to extend the previously shown algorithms to the dynamic case [Werbos, 1990]. The problem involved in such technique however is to define an optimal depth for the network unfolding (i.e. how many times replicate the recurrent layers), that is arbitrary. A shallow unfolding yields scarce long-term learning, whereas a too deep one is affected by cumbersome computational costs and gradient vanishing problems. A different approach, proposed by [Williams and Zipser, 1989], allows an online training for a generic dynamic ANN without need to unfold it into a static one. The idea at the base of the Real-Time Recurrent Learning algorithm (RTRL) is to compute, over a period of time, the gradient of the error with respect to the network weights, for each time step. In concept, it is very similar to a EBP approach. However in this case, the partial derivatives of the error must take into account the effect of weight change over a period of time. Some basic declarations must be done to understand the discussion: the network has n units arbitrarily connected, and m of these units are connected to external independent inputs. The vector $y(t)$ denotes the outputs of the units at time t, and $x(t)$ the external input signals to the network at time t. The concatenation of these two vectors gives z_k, where $k \in U \cup I$. The set I denotes indices k for which z_k is an external input, and U the ones for which it is the output of a network unit. The weights of the ANN are in the matrix $\mathbf{W}$, with a unique weight between every pair of units and to the input lines. The weight matrix has dimensions $n \times (n + m + 1)$ and is shown in Figure 1.25. Given the *net* value at a time t as:

$$net_k(t) = \sum_{l \in U \cup I} w_{k,l} z_l(t) \tag{1.65}$$

The output for the *k-th* unit at the next time-step is:

$$y_k(t+1) = f_k(net_k(t)) \tag{1.66}$$

We now define the *error* of the network for specific units (i.e. the output ones) for which a desired value is defined at a particular time. Note that the output units are not bound to be the same over time. The set of index for which the unit is an output is denoted by $T(t)$.

$$e_k(t) = \begin{cases} d_k(t) - y_k(t) & k \in T(t) \\ 0 & otherwise. \end{cases} \tag{1.67}$$

Then, the error at a particular time is:

$$E(t) = \frac{1}{2} \sum_{k \in U} e_k(t)^2 \tag{1.68}$$

And the error over a period of time from t_0 to t_1 is:

$$E_{TOT}(t_0, t_1) = \sum_{t=t_0}^{t_1} E(t) \tag{1.69}$$

The goal is to adjust the weight matrix $\mathbf{W}$ over this trajectory to minimize the error, thus, towards the negative gradient $\nabla \mathbf{W} E_{TOT}$. Since the error accumulates linearly, so does the gradients and the weight adjustment matrices.

$$\Delta w_{i,j} = \sum_{t=t_0}^{t_1} \Delta w_{i,j}(t)$$
$$\Delta w_{i,j}(t) = -\alpha \frac{\partial E(t)}{\partial w_{i,j}} \tag{1.70}$$

Where α is a learning rate. The partial derivative of the error can be computed from Eq. 1.68:

$$-\frac{\partial E(t)}{\partial w_{i,j}} = \sum_{k \in U} e_k(t) \frac{\partial y_k(t)}{\partial w_{i,j}} \tag{1.71}$$

The partial derivative in the sum can be obtained by differentiating Eq. 1.65 and Eq. 1.66:

$$\frac{\partial y_k(t+1)}{\partial w_{i,j}} = f'_k(net_k(t)) \left[\sum_{l \in U} w_{k,l} \frac{\partial y_l(t)}{\partial w_{i,j}} + \delta_{i,k} z_j(t) \right] \tag{1.72}$$

Where $\delta_{i,k}$ is equal to 1 for $i = k$ and 0 otherwise. Since the initial state of the network is independent from the weights, we can assume:

$$\frac{\partial y_k(t_0)}{\partial w_{i,j}} = 0 \tag{1.73}$$

This term express the sensitivity of the output of the *k-th* neuron with respect to a change in the value of $w_{i,j}$, at a time t, taking into account the effect of such a change in the weight over the entire network trajectory from t_0 to t. The weight $w_{i,j}$ does not have to be connected to unit k. Thus this algorithm is non-local. Eq.1.72 is obviously recursive, and with the initial conditions given by Eq.1.73 the problem can be formulated in terms of a dynamic system evolution:

$$\frac{\partial y_k(t_0)}{\partial w_{i,j}} = p_{i,j}^k(t) \tag{1.74}$$

$$\begin{cases} p_{i,j}^k(t+1) = f_k'(net_k(t)) \left[\sum_{l \in U} w_{k,l} p_{i,j}^l(t) + \delta_{i,k} z_j(t)\right] \\ p_{i,j}^k(t_0) = 0 \end{cases} \tag{1.75}$$

Thus the algorithm computes, at each time step, the quantities $p_{i,j}^k(t)$, and then through the error $e_k(t)$ te weight correction factor:

$$\Delta w_{i,j}(t) = \alpha \sum_{k \in U} e_k(t) p_{i,j}^k(t) \tag{1.76}$$

Analogously to EBP, in this algorithm the corrections can be made either by accumulating several weight correction matrices (batch) or in real time (on-line). Indeed the training will not follow directly the negative gradient of the error, but this may be an advantage if the phenomenon under study changes dynamically.

Chapter 2

Inverse Problems and Optimization

This chapter will explain the concept of Inverse Problem in engineering applications (although the discussion given here is easily applicable to other scientific areas) and how these problems can be solved through a process called Optimization. The concepts common to all optimization techniques will be shortly presented, and then, an overview of the most important techniques found in literature will be examined.

2.1 What is, and how can we solve, an Inverse Problem

To understand what an Inverse Problem is, it is better to understand first what is a Direct Problem. A Direct Problem is the estimate of the measurement of a phenomenon given the knowledge of its model. Suppose our idea

is to estimate the electric field in three dimensional space by knowing the charge distribution. If the charge distribution is known, we can of course use the simple Coulomb's Law in its vector form:

$$E(r,\rho) = \frac{1}{4\pi\epsilon_0}\int \frac{r-r'}{|r-r'|^3}\rho(r')dV' \tag{2.1}$$

This allows us to define the electric field $E(r)$ in all space. Suppose now, that the charge distribution $\rho(r')$ is unknown, but a set of $\{E_k, r_k\}, k = 1, 2...n$ field values in points r_k are known from experimental measurement. An Inverse Problem starts from such knowledge and tries to estimate the generating model (i.e. the charge distribution). An important difference that arise quickly is that the direct problem has a single solution, whereas the inverse one has several. This is intuitive: a charge distribution will give a definite and unique field, but the same field can be generated by several equivalent charge distributions. Indeed it is possible to *assume* a distribution charge given some observations on the experimental measurements. With this assumed distribution, that we will call $\tilde{\rho}(r')$, we can compute the Electric Field in the same points where we have the measurements $E(r_k, \tilde{\rho})$. Then we can compute the cost function:

$$f(\tilde{\rho}) = \frac{1}{n}\sum_{k=1}^{n}(E(r_k,\tilde{\rho}) - E_k)^2 \tag{2.2}$$

This cost function is a merit figure of the error: the lower, the better our model represent the system. However, this does not implies that when the error is zero we have $\tilde{\rho}(r') = \rho(r')$, because we could have found one of the several charge distributions that generates an equivalent electric field. Still we found a model that we can use to compute directly the electric field, and

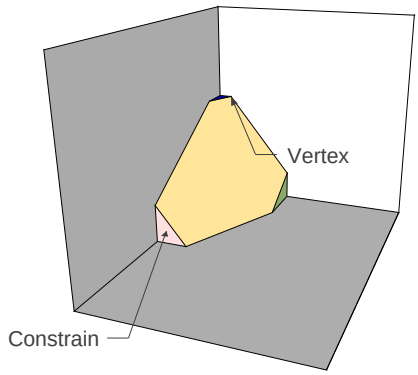


Figure 2.1: A simplex for a problem in 3 variables

.

this is an *optimal* solution. To find such solution, the only information that we have is how low is the error of the current one. The process of refining the current solution (i.e. moving it in the solution space) towards better ones by following a criterion of error minimization is called *optimization*. Finding the minimum of Eq. 2.2 can be very difficult if the nature of $f(\tilde{\rho})$ is non linear, as will be shown in the next sections.

2.1.1 A glimpse of Linear Optimization - The Simplex Algorithm

To understand the implications of non-linear optimization, some basic properties of linear optimization should be given. The simplex theorem is a numerical method to solve linear optimization problems, like the following:

$$\begin{cases} maximize & (c^T x) \\ with & \mathbf{A}x \leq b \\ and & x \geq 0 \end{cases} \tag{2.3}$$

Where x represents the vector of variables (to be determined), c and b are vectors of (known) coefficients and $\mathbf{A}$ is a (known) matrix of coefficients. The expression to be maximized is called the *objective function*. The inequalities define a convex polytope over which the objective function needs to be optimized, like the one shown in 2.1. It can be proved that if the function and the constraints are linear, the solution will lie on one of the vertex of the polytope. For this reason, the algorithm will start from a random vertex of the simplex and will check, every time, if it is the optimal one. To find the vertex of the simplex, we search for the so called *admissible base solutions* (ABS). Suppose we have a problem with m constraints and n variables. The coefficient matrix $\mathbf{A}$ will be $m \times n$. First, a *base* for matrix $\mathbf{A}$ must be found.

- $\mathbf{B}_{m\times m}$ is a sub-matrix of $\mathbf{A}_{m\times n}$ composed by random m columns linearly independent.
- $\mathbf{N}_{m\times(n-m)}$ is the matrix obtained with the remainders.

Thus we can represent the original matrix as:

$$\mathbf{A} = [\mathbf{B}|\mathbf{N}] \tag{2.4}$$

In other words we have rewritten the matrix with m independent column on the left. Of course, the unknowns vector should be rewritten as well:

$$x = \left[\frac{x_B}{x_N}\right] \tag{2.5}$$

Where x_B and x_N are the in-base and out-of-base unknowns. By putting to zero the out-of-base unknowns, the system is now satisfied for a set of *base solutions*. We rewrite the system as:

$$\begin{cases} x_B = \mathbf{B}^{-1} \times b \\ x_N = 0 \end{cases} \tag{2.6}$$

If $\mathbf{B}^{-1}b > 0$ then $\mathbf{B}$ is an admissible base, since it ensure the non-negativity of the unknowns. We then proceed to discard all solutions outside the admissibility set. The remaining solutions $\left[\frac{x_B}{[0]}\right]$ are the admissible solutions, and corresponds to the vertex of the polytope. Indeed, if the dimensionality of the problem was low, an exhaustive search of all the possible solutions could quickly yield the optimal one. However, the number of vertex is proportional to both the number of variables and the number of constraints. For this reason, an exhaustive search is generally unadvised. The simplex algorithm propose a method to move from one vertex to the other and check if the solution is optimal. The linear problem, rewritten with the matrix split is:

$$\begin{cases} maximize & (c_B^T x_B + c_N^T x_N) \\ with & x_B = \mathbf{B}^{-1}b - \mathbf{B}^{-1}\mathbf{N}x_n \geq 0; \\ and & x_n \geq 0 \end{cases} \tag{2.7}$$

By substituting the second equation into the cost function we have:

$$\left(c_B^T \mathbf{B}^{-1} b - c_B^T \mathbf{B}^{-1} \mathbf{N} x_N + c_N^T x_N\right) \tag{2.8}$$

And by isolating x_N we have:

$$\begin{cases} maximize & \left(c_B^T \mathbf{B}^{-1} b + (c_N^T - c_B^T \mathbf{B}^{-1} \mathbf{N}) x_N\right) \\ with & x_B = \mathbf{B}^{-1} b - \mathbf{B}^{-1} \mathbf{N} x_n \geq 0; \\ and & x_n \geq 0 \end{cases} \tag{2.9}$$

We now define a reduced cost vector function:

$$\gamma = (c_N^T - c_B^T \mathbf{B}^{-1} \mathbf{N}) \tag{2.10}$$

In the hypothesis of being on a ABS, we have:

$$\begin{cases} x_B = \mathbf{B}^{-1} b \\ x_N = 0 \end{cases} \tag{2.11}$$

To understand if the ABS is the optimal one, it is sufficient to check the sign of γ. By moving slightly from the ABS, we perturb the out-of-base variables. The variation of the cost function is going to be $\gamma \times x_N$. If γ is positive, the variation of the functional will be as well. This means that the perturbed solution is better than the actual one, and for this reason, the actual one is not the global optimum. The algorithm proceed to move to the next solution by incrementing an out-of-base unknown. Indeed, the in-base unknowns will vary as well since they are linked by the relation:

$$x_B = \mathbf{B}^{-1} b - \mathbf{B}^{-1} \mathbf{N} x_n \tag{2.12}$$

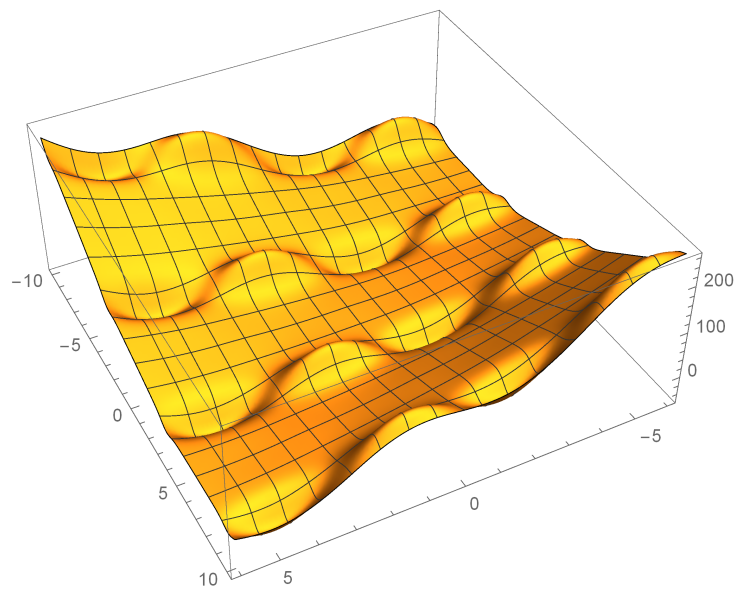


Figure 2.2: Bird Function: $f(x,y) = sin(x)e^{(1-cos(y))^2} + cos(y)e^{(1-sin(x))^2} + (x-y)^2$

.

The increment continues until one of the in-base unknowns reaches zero. Geometrically, this is equivalent to moving on one of the edges of the polytope. Once the unknown reaches zero, it is switched with the out-of-base unknown, and the procedure to check if it is an optimal one is repeated.

2.1.2 Global Optimization

In literature the term *Global Optimization* is used to identify the research area involving the development of techniques for minimization of non-linear functions. The term *global* points out the most obvious problem involved in non-linear minimization: the possibility of multiple minima existence. A non-linear function like the one shown in Figure 2.2 features multiple local minima

that may be very far from the optimal solution. Several complications may arise when non-linear functions are involved:

- The function may be non-convex, thus multiple minima may exist.
- There is no way to determine if the solution found is the global minimum.
- The function may diverge for some values, or its derivative may become zero (flat areas) in a region, or may present discontinuities.
- If a priori knowledge of the problem is unavailable, the search domain for the minimum is unbounded.

Indeed, searching for the minima of a function with such premises is very difficult. The non-convexity of the function limits the use of simple gradient/Jacobian based approaches, since if initialized near a local minima, the optimization procedure will converge there, instead of searching the domain for the global optimum. The solution space for a non-linear problem is a large and difficult area to explore, due to discontinuities and divergences. For this reason, a set of robust search techniques have been developed specifically for non-linear problems. These techniques are often inspired to natural phenomenons, and for this reasons, several are addressed as *meta-heuristics*. A complete and exhaustive discussion on all the global optimization techniques found in literature is beyond the purpose of this work. However, the techniques with the highest practical implementability will be analyzed in this chapter.

2.1.3 Exploration *vs* Exploitation

The choice of a particular algorithm to solve a problem over one other is, indeed, dependent on the nature of the problem itself. A not so intuitive evolution of such concept is that, given the totality of problems, any elevated performance over one class of problems is offset by performance over another class, for a given algorithm. This is the enunciate of the *No Free Lunch Theorem*, and was formally put in a mathematical framework by [Wolpert and Macready, 1997]. The practical implication of this theorem is that, given a fixed computational resource, there is no algorithm able to solve efficiently all optimization problems. This is because every algorithm search capabilities features two mutually exclusive characteristics: **exploration** and **exploitation**. Explorative algorithms (i.e. that features an high exploration) have a very large mobility in the solution space. They are able to investigate several areas at once, and are thought to avoid being trapped into local minima. Highly explorative algorithms can investigate the whole domain and give an approximation of the optimal solution easily. However, the convergence of these algorithms is very slow, if not absent at all. On the other hand, exploitation algorithms (also known as local search algorithms) are less resilient to avoid local minima, but converge rapidly towards the solution. To solve a difficult optimization problem correctly, both characteristics are needed. The No Free Lunch Theorem prove that both traits can not be found in the same algorithm, and for this reason, hybrid ones are being used for practical optimization problems.

2.2 Evolutionary Algorithms

Evolutionary Algorithms are a subset of Global Optimization Algorithms that borrows some concepts directly from natural evolution theory. All algorithms represent the current solution(s) as an individual (population). Such individual is subject to several mechanisms (often named "operators" in literature) like *reproduction*, *selection*, and *mutation*. The basic idea is to mimic the natural evolution in its ability to solve problems. Evolutionary Algorithms exhibit very high exploration capabilities, and are well suited for parallel implementation.

2.2.1 Genetic Algorithms

Genetic Algorithms (GA) are the most common Evolutionary Algorithm that is found in literature. The mechanism at the base of a GA is the *natural selection*. Each k-th individual represent a solution s_k to the non-linear problem at hand, with cost function $f(s_k)$. To each solution, we can associate the relative cost function of the problem. The inverse of such cost function is the *fitness* of the individual g . The algorithm goes as follows:

1. An initial population of individuals is created randomly.
2. Each individual is evaluated for his fitness.
3. According to a selection criterion, a set of individuals is extracted from the population, with probability of extraction proportional to the fitness.

4. Extracted individuals are coupled in pairs. From each pair, two new individuals are born with mixed traits from the original ones.

5. A low percentage of the population is randomly mutated.

6. Repeat from 2 until convergence or halt criterion is reached.

We will now explain in detail the steps. First, the concept of *individual* must be discussed. Indeed each element of the population is nothing more than a solution. However, the way this solution is expressed influences strongly the cross-over and mutation steps. Several codings techniques are possible for a GA, however the two most used in literature are *binary* and *continuous*.

Binary GA

Suppose the solution space is R^N. Each solution has N parameters (addresed as *chromosomes*). In *binary* GA, each solution is coded in binary with M digits (addressed as *alleles*). The k-th individual of the population is then:

$$s_k = \begin{matrix} 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & (Chromosome^1) \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & (Chromosome^2) \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & (Chromosome^3) \end{matrix} \tag{2.13}$$

In this case, we used a 8-bit coding precision for the variables. This means that each variable of the problem can assume values between 0 and 255. This may be inconvenient, especially if the problem is real-valued. For this reason, for fitness evaluation the chromosomes are normalized to the search domain of the problem. Addressing with s_k^u the u-th parameter of the problem,

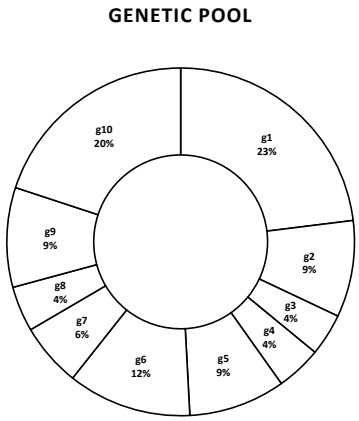


Figure 2.3: Genetic pool for selection. Each individual has a chance to be extracted from the pool proportional to its fitness.

.

$$ns_k^u = lower_u + \frac{s_k^u}{255}(upper_u - lower_u) \tag{2.14}$$

Where ns_k^u is a normalized, real-valued parameter, that can be passed to the cost function, and $upper_u$ and $lower_u$ are the u-th boundaries for the search domain in the solution space. As it can be seen, if s_k^u is zero, the normalized parameter will assume the lowest admissible value. If it is 255, it will assume the highest admissible value. The fitness $g_k(ns_k)$ is computed for each *k-th* individual. Then, the *selection* process begins. The idea at the base of this stage is to choose individuals with a better fitness to create the next generation of individuals. However, choosing only the best individuals and discarding the worst one would reduce the exploration capabilities of the

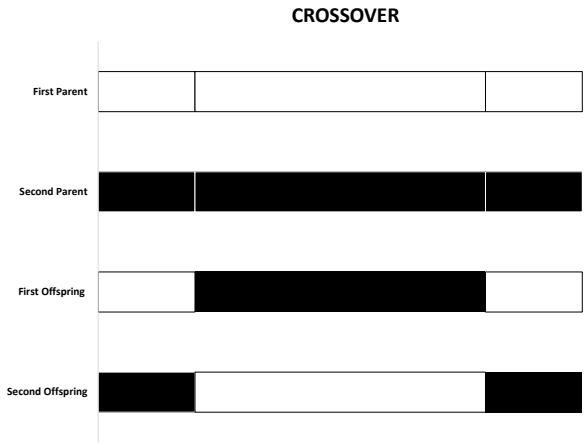


Figure 2.4: Two-Point crossover operator.

.

algorithm, and force it to converge, sooner or later, on the current optimum. Natural selection on the other hand does not exclude, a priori, the worst individuals, it just makes it unlikely that they will reproduce. This behavior is included in the GA as well. As can be seen in Figure 2.3 each individual is included in a roulette-like system, where the probability of being extracted is proportional to its fitness. This means that even individuals with very low fitness can be extracted for the next stage of the algorithm. A number of couples are created (the same individual can be extracted more than once, but not twice in the same couple). Each couple generate offsprings according to some *crossover* operator. Several techniques exist in literature, but the two-point operator is the most common. The technique is shown in Figure 2.4. Two random points are used split both parents chromosomes. The three

segments are then switched, and the offspring is created. After crossover is completed, the new population is subject to random occurrences of *mutation*. This ensure to maintain genetic diversity from a generation to the next one, and is crucial to evolve the solution. The probability of mutation however should be low, to avoid turning the algorithm in a random search one. The most used technique for mutation is the bitswitch:

$$\begin{array}{cccccccc} 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & \mathbf{0} & 1 & 0 & 1 & 1 & 0 \end{array} \tag{2.15}$$

Each individual has a probability of having one of its alleles switched. Mutation is the last stage of the algorithm, after which, it is repeated from the *selection* until the stop criterion (either convergence or maximum number of iterations) is reached. Indeed, it is not guaranteed that, from a generation to the next one, the algorithm will evolve toward better solutions. For this reason, practical implementations always keep note of the *best* solution found so far, even if by chance such individual was removed from the genetic pool.

Real-Coded GA

Real-coded genetic algorithms are less common and more difficult to treat than Binary Coded ones. Still, it is much more intuitive to see a chromosome as a vector of floating point numbers that points to the solution in the solution space. An example of individual is:

$$s_k = \begin{bmatrix} s_k^1 \\ s_k^2 \\ ... \\ s_k^N \end{bmatrix} = \begin{bmatrix} 2.145928e+0 \\ 1.112341e+3 \\ ... \\ 1.302941e-2 \end{bmatrix} \tag{2.16}$$

Indeed, the use of real coding for the chromosomes avoid the requirement to define a search space through lower and upper boundaries (although it is still strongly advisable). It is possible to explore the solution space taking into advantage the variable resolution of floating point coding, instead of being stuck with the uniform sampling of the binary coding. In a Real-Coded GA the selection operator is identical to the one used in the binary version. The crossover between parents however is different, because using floating-point coding, it is meaningless to operate at bit level. The operations are then carried out at chromosome level. The most common crossover technique is an arithmetic combination:

$$\begin{cases} s_k^u = \lambda s_w^u + (1-\lambda) s_v^u & \\ with & k \in offsprings \\ and & w, v \in parents \end{cases} \tag{2.17}$$

The term λ is a random factor between 0 and 1. Mutation in this case is a random number added (or subtracted) to one of the chromosomes. A very detailed discussion on the topic can be found in [Herrera et al., 1998].

Elitism and Adaptive GA

Since the goal of an optimization algorithm is to find the fittest solution to a problem, it is intuitive that conserving the best solution found so far has sense. Indeed, for a GA, this does not force the algorithm to actually conserve the individuals. However, doing so, is an interesting strategy that has advantages and drawbacks. The term elitism define a technique involving the direct copy, in the next generation, of the fittest candidates. Those individuals undergo the selection operator, and if selected, contribute through crossover to the individuals in the next generation. However,along generating offspring, they are copied in the next generation as well. This process has a very strong effect of keeping the solution in the neighborhood of the current optimum, thus enhancing convergence. The opposite strategy is possible as well: the *worst* individuals of the population are carried on. In this case, the elitism try to conserve the greater genetic diversity through iterations, keeping intact an attraction factor very far from where the solutions are already clustering. Another strategy to maximize the efficiency of GA is to introduce variable probabilities for crossover and mutation according to the state of advancement of the algorithm. [Zhang et al., 2007] propose an Adaptive GA that adjust the two probabilities according to the degree of clustering of the solutions around the best and the worst individuals in the search domain, using a Fuzzy Logic controller.

2.2.2 Differential Evolution

Differential Evolution [Storn and Price, 1997] is similar to a Real-Coded GA with simpler rules for recombination of individuals. Each *k-th* individual s_k is a solution in the R^N space. In general, the s_k^u parameters of the solutions are coded with floating-point precision, thus individuals of the population are the same as in Real Coded GA (Eq. 2.16). The population is initialized randomly in the search space. Then, the algorithm goes as follows:

- For each i-th individual of the population, pick three other individuals j-th, k-th and p-th. The four individuals must be different.
- Pick a random parameter index $R \in 1, ..., N$
- For each q-th parameter different from the *R-th*, check against a crossover rate CR. If crossover happens, the new parameter is $s_j^q + F \times (s_k^q - s_p^q)$, else, is the original s_i^q one.
- For the q-th parameter euqal to the *R-th*, perform the previous operation without the crossover rate (i.e. will always occur for at least a parameter).
- Check if the new position for *i-th* individual is better than the previous one. If it is, update it, else, discard it.

The choice for the F and CR parameters is, in general, empirical. Authors [Storn and Price, 1997] suggest to choose a population NP between 5N and 10N, with at least 4 individuals to ensure a high enough number of vector to combine. The factor F can start at 0.5, and may be increased if convergence is too fast. The CR should be around 10%.

2.3 Swarm Intelligence

Swarm intelligence is a class of meta-heuristics algorithm that mimics some social mechanics found in nature. In nature, swarms are a set of individuals that through independent *and* collective behavior manage to solve a particular problem. The most notable example is the search for food. Swarm algorithms observe and imitate such mechanics. Similar to evolutionary algorithms, the solutions are still represented by individuals (this time, addressed differently according to the specific algorithm). In this case however, the rules for position updating are less abstract than genetic operators, and are easier to visualize like mechanical interactions or a social behavior. As a rule of thumb, Swarm Intelligence algorithm are less explorative than evolutionary algorithms, but more than local search ones.

2.3.1 Particle Swarm Optimization

Analogously to the GA for Evolutionary Algorithms, the Particle Swarm Optimization (PSO) is the most commonly known swarm intelligence algorithm. The original algorithm proposed in [Eberhart et al., 1995] is based on a population of particles that are characterized by a position (i.e. the current solution represented by the individual) and a *speed*, representing the movement vector that will be applied to the particle at the next iteration. Three factors influence the speed:

- The former velocity of the particle, through an *inertial* parameter ω.
- The attraction towards the best solution found by the particle so far, through the *cognitive* parameter λ.

- The attraction towards the best solution found by the swarm so far, through the *social* parameter γ

For each *k-th* individual, position x_k and speed v_k is given by:

$$v_k[t] = \omega v_k[t] + \lambda \left(pb_k[t] - x_k[t]\right) + \gamma \left(gb[t] - x_k[t]\right) \tag{2.18}$$

$$x_k[t+1] = x_k[t] + v_k[t] \tag{2.19}$$

Where $pb_k[t], gb[t]$ are the positions of the best solution found by the *k-th* particle so far (addressed as *personal best*) and the position of the best solution found by the swarm so far (addressed as *global best*). Indeed, positive values for λ, γ will exhibit an attraction of the particles towards their *personal best* and *global best*. In general, the three parameters are not constant, but are rather defined as a random number between minimum and maximum values.

$$\begin{cases} \omega \in [\omega_{min}, \omega_{max}] \\ \lambda \in [\lambda_{min}, \lambda_{max}] \\ \gamma \in [\gamma_{min}, \gamma_{max}] \end{cases} \tag{2.20}$$

The randomness in the parameters grants local minima escape capabilities to the algorithm.

2.3.2 Flock Of Starlings Optimization

An interesting evolution of the classic PSO was introduced in [Fulginei and Salvini, 2010], by including to the dynamics of the algorithm some observations made in ornithology. It has been observed that particular flocks of

birds (starlings) moves in an organized pattern that is able to explore very large areas yet keeping the flock compact. The reason is that every member of the flock observe a set of about 7 other birds in the flock, and mimic their movements. Including this behavior to the PSO is very simple. It is sufficient to add one last addend to the $v_k[t]$:

$$v_k[t] = \omega v_k[t] + \lambda \left(pb_k[t] - x_k[t]\right) + \gamma \left(gb[t] - x_k[t]\right) + \sum_{i \in F_k} h_{k,i} v_i[t] \tag{2.21}$$

In the last term, the set F_k is the set of indexes correspondent to the birds observed by the k-th bird, and $h_{k,i}$ are the coupling coefficients with their velocity. Indeed, the size of F_k can be as small as another bird, and as large as the whole flock. Compared to the PSO, the FSO presents better exploration capabilities for very large solution spaces.

2.3.3 Continous Flock Of Starlings Optimization

Despite the exploration capabilities, the FSO is still a discrete algorithm whose behavior is controlled by coefficients in a way that is not directly apparent. Indeed, by modifying the $\omega, \lambda, \gamma, h$ parameters it is possible to influence the algorithm general conduct, but strict control of convergence, divergence and oscillation is impossible. A big novelty on this behalf was introduced with the Continuous Flock of Starlings algorithm (CFSO). In this algorithm, the update equations for the FSO were considered as state equations for a dynamic system, and were integrated in the Laplace domain.

$$s \begin{bmatrix} \mathbf{V}(s) \\ \mathbf{X}(s) \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{1} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{V}(s) \\ \mathbf{X}(s) \end{bmatrix} + \begin{bmatrix} \mathbf{F}(s) \\ \mathbf{0} \end{bmatrix} + \begin{bmatrix} \mathbf{v}(0) \\ \mathbf{x}(0) \end{bmatrix} \tag{2.22}$$

$$\mathbf{X}(s) = (s^2 1 - \mathbf{A}_{1,1} s - \mathbf{A}_{1,2})^{-1} \cdot \{(s1 - \mathbf{A}_{1,1}) \cdot \mathbf{x} + \mathbf{v} + \mathbf{F}\} \tag{2.23}$$

$$\mathbf{A}_{1,1} = \omega \cdot 1 + \mathbf{H} \tag{2.24}$$

$$\mathbf{A}_{1,2} = -\mu \cdot \mathbf{1} \tag{2.25}$$

$$\Im_k^j(t) = \lambda \times pb_k^j(t) + \gamma \times gb^j(t) \tag{2.26}$$

In Eq. 2.22, $\mathbf{V}$ and $\mathbf{X}$ are two vectors expressing the velocities and positions for the different members of the flock. The $\mathbf{F}$ is a forcing term that takes into account both the personal and the global bests. The sub-matrix $\mathbf{A}_{1,1}$ combines the contribution for both inertia (ω) and the particle-to-particle interaction ($\mathbf{H}$). The sub-matrix $\mathbf{A}_{1,2}$ accounts the cognitive and social coefficients (given that $\mu = \lambda + \gamma$). The integration in Laplace domain for this system can be done under some assumptions for the $\mathbf{H}$ matrix. However, expressing the vector $\mathbf{F}$ as a Laplace transform can be rather tricky. Indeed, in Eq. 2.26 we can see the forcing vector in the time domain. The $pb_k^j(t)$ and $gb^j(t)$ functions are not known a priori, since they change according to the different points found by the algorithm during the exploration. For this reason, it is not possible to express them as Laplace transform. To solve this

problem, the approach proposed in the algorithm suggests evaluating these two functions on a time-window (TW) reference. Each time window has a τ length. Inside the time-window, it is possible to assume that the value assumed by the global and personal best for each particle does not change, and is equal to the one evaluated at the beginning of the time-window. Under this assumption, it is possible to perform the Laplace integration on a time-window basis, thus effectively tracing the trajectory for the particles in closed form for each TW. Updating the particle position by using closed forms for trajectories has major advantages in terms of supervising the algorithm global behavior. Indeed, the set of particles is now a dynamic system completely defined in the Laplace domain. t is then possible to analyze the poles of the system to understand its stability conditions.

$$s_{1,2} = \frac{1}{2}(\tilde{\omega} - \tilde{h} \pm \sqrt{\left(\tilde{\omega} - \tilde{h}\right)^2 - 4\tilde{\mu}} \tag{2.27}$$

$$s_{3,4} = \frac{1}{2}(\tilde{\omega} + (N-1)\tilde{h} \pm \sqrt{\left(\tilde{\omega} + (N-1)\tilde{h}\right)^2 - 4\tilde{\mu}} \tag{2.28}$$

Equations 2.27 and 2.28 express the poles for the $\mathbf{X}$(s) for a particular case of $\mathbf{H}$ where all the birds are connected (*Fully Connected CFSO*). The $\sim$ symbol on the ω, μ, h parameters expresses a parameter that is valid only under the current time window. Under the assumption that $\left(\tilde{\omega} - \tilde{h}\right)^2 - 4\tilde{\mu} \neq 0$ and $\left(\tilde{\omega} + (N-1)\tilde{h}\right)^2 - 4\tilde{\mu} \neq 0$ all poles are simple (either complex or real) with single multiplicity. We can thus apply the inverse Laplace Transform and obtain the expression for the single particle trajectory. Stability analysis can be performed by observing the real and imaginary parts of the poles:

$$\tilde{\omega} < \tilde{h} < -\frac{\tilde{\omega}}{N-1} \; or \; \tilde{\mu} > 0 \tag{2.29}$$

$$-2\sqrt{\tilde{\mu}} + \tilde{\omega} < \tilde{h} < 2\sqrt{\tilde{\mu}} + \tilde{\omega} \; or \; \tilde{h} < \frac{2\sqrt{\tilde{\mu} - \tilde{\omega}}}{N-1} \tag{2.30}$$

Asymptotic stability can be obtained by condition Eq. 2.29, whereas oscillation can be obtained by condition Eq. 2.30. Indeed, if the poles are real part negative, the resulting trajectories in the time domain will tend to a convergence towards an equilibrium point. If poles are real part positive, a fast divergence will occur. In both cases, if the poles are complex conjugates, the behaviors will be combined with oscillations.

2.3.4 Firefly Algorithm and Glowworm Swarm Algorithm

The firefly algorithm (FA) is a swarm intelligence algorithm especially suited for multiobjective optimization problems [Yang, 2010]. A multi-objective optimization problem, put in simple terms, is a problem where multiple cost function are defined. The algorithm is based on the mating process of fireflies. Fireflies are attracted to other ones (regardless of gender) on a brightness based criterion: a firefly is attracted to its brighter mates, and will move towards them. However, perceived brightness from a fly to the other decreases with distance: this creates the tendency to organize individuals in clusters. If the brightness of the individual is related to the cost function, the clusters will lie in optima.

The pseudoalgorithm goes as follows:

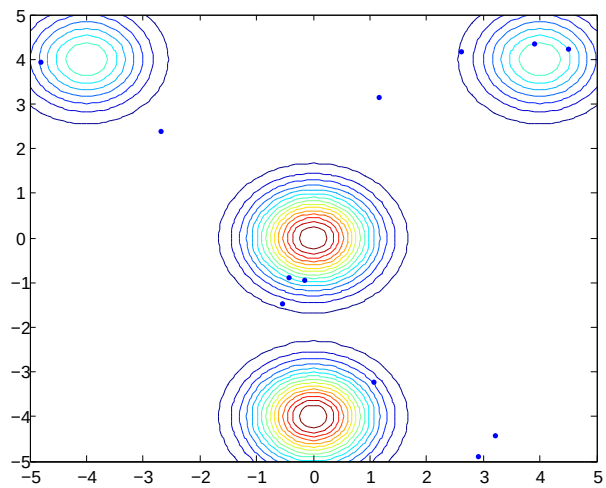


Figure 2.5: Starting conditions for the Firefly algorithm.

.

1. A random population is created, and the brightness for each individual is computed through cost function evaluation.

2. Each individual observe his mates one by one: if the perceived brightness (i.e. scaled by the distance) is higher than his, he moves towards him.

3. If no individual has a higher brightness, it moves randomly.

The resulting behavior can be seen in Fig 2.5 and 2.6. Individuals starts scattered and then creates groups in regions where minima are found. The position update equation used in step 2 is the following, for a couple of j and i fly (i moving towards j).

$$x_i[t+1] = x_i[t] + \beta e^{-\gamma r_{i,j}} \times (x_j[t] - x_i[t]) + \alpha[t]\epsilon[t] \tag{2.31}$$

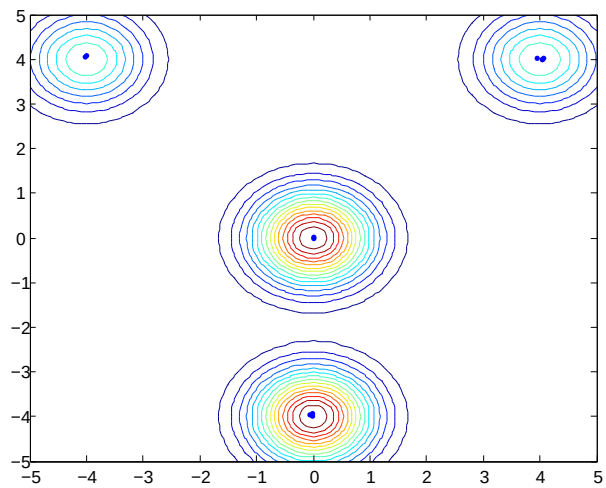


Figure 2.6: Ending conditions for the Firefly algorithm. Individuals clustered in the various minima

.

where $\alpha[t]$ is a parameter controlling the step size, while $\epsilon[t]$ is a vector drawn from a Gaussian or other distribution. The γ should be related to the scales of design variables. Ideally, the β term should be order one, which requires that γ should be linked with scales. For example, one possible choice is to use $\gamma = 1/\sqrt{L}$ where L is the average scale of the problem. In case of scales vary significantly, γ can be considered as a vector to suit different scales in different dimensions. Similarly, $\alpha[t]$ should also be linked with scales. For example, $\alpha[t] \leftarrow 0.01L\alpha[t]$. in actual implementation by most researchers, the motion of the fireflies is gradually reduced by an annealing-like randomness reduction via $\alpha[t] = \alpha_0\delta[t]$ where $0 < \delta < 1$ (e.g., $\delta = 0.97$) , though this value may depend on the number of iterations.[2] In some difficult problem, it may be helpful if you increase $\alpha[t]$ at some stages, then reduce it when necessary. This non-monotonic variation of $\alpha[t]$ will enable

the algorithm to escape any local optima when in the unlikely case it might get stuck if randomness is reduced too quickly. An interesting variation of the AF algorithm is the Glow Worm Optimization (GWO) [Krishnanand and Ghose, 2005]. This algorithm shares basically the same movement principle of the AF: brighter worms attract darker ones, and the brightness is related to the cost function. However, a concept of *neighborhood* is introduced: a worm will not be attracted by distant individuals if he is surrounded by enough other worms. The practical difference with AF is that fireflies are clustered by distance, glow worms by density.

2.4 Local Search Techniques

Local search techniques are used for problems with a very narrow search domain. These techniques can converge quickly to the optimum, but if used in the wrong way, may suffer from local minima entrapment. These techniques should never be used if it is suspected that the cost function, in the search domain, is strongly non-convex.

2.4.1 Linear Techniques

Indeed, all linear techniques proposed by now (GD, CGM, LM) can be used as optimization tools for a non-linear problem. This freedom of course should be limited only to local searches, where a complex non-linear function can be safely approximated as linear or quadratic. The advantage of linear techniques is the degree of convergence, which can be faster than any metaheuristic. However, most of these techniques requires the computation of the

cost function derivative (either to compute the gradient, or the jacobian/hessian). It is pretty common for non-linear optimization problem to be defined with a non-analytical cost function. It is then impossible to compute its derivative. In these cases, the gradient and the jacobian are computed iteratively by approximating the cost function with a linear one (computing the 2-point derivative).

2.4.2 Bacterial Chemotaxis Algorithm

The BCA has been proposed in [Müller et al., 2002] and is an evolutionary algorithm based on the emulation of the motion of a real bacterium looking for food (i.e. the optimum of a cost function). A mathematical description of a 2D bacterium motion can be developed by assuming an assigned speed v and by the determination of suitable probabilistic distributions of the motion duration τ and of the direction ϕ shown by each individual. The virtual bacterium motion follows the following rules:

- The path of a bacterium is a sequence of straight-line trajectories joined by instantaneous turns, each trajectory being characterized by speed, direction, and duration.
- All trajectories have the same constant speed.
- When a bacterium turns, its choice of a new direction is governed by a probability distribution, which is azimuthally symmetric about the previous direction. In two dimensions, this means that the probability to turn left or right is the same.

- The angle between two successive trajectories is governed by a probability distribution.
- The duration of a trajectory is governed by an exponentially decaying probability distribution.
- The probability distributions for both the angle and the duration are independent of parameters of the previous trajectory.

The velocity, as stated above, is constant and assigned: $v = const$. The step size is determined by τ. The duration of the trajectory can be computed from the distribution of a random variable with an exponential probability density function:

$$P(X = \tau) = \frac{1}{T} e^{-\tau/T} \tag{2.32}$$

Where the expected value is $\mu = E(X) = T$ and the variance $\sigma^2 = T^2$. The time T is given by:

$$T = \begin{cases} T_0, & for\ \frac{f_{pr}}{l_{pr}} \geq 0 \\ T_0 \left(1 + b \left| \frac{f_{pr}}{l_{pr}} \right| \right), & for\ \frac{f_{pr}}{l_{pr}} < 0 \end{cases} \tag{2.33}$$

Where T_0 is the minimal mean time, f is the cost function, f_{pr} is difference between the actual and the previous function value, l_{pr} is the vector connecting the previous and the actual position in the parameter space and b is a dimensionless parameter. The new direction for left or right turning can be computed by:

$$P(X = \alpha, v = \mu) = \frac{1}{\sigma\sqrt{2\pi}} exp\left[-\frac{(\alpha - v)^2}{2\sigma^2}\right] \tag{2.34}$$

$$P(X = \alpha, v = -\mu) = \frac{1}{\sigma\sqrt{2\pi}} exp\left[-\frac{(\alpha - v)^2}{2\sigma^2}\right] \tag{2.35}$$

Given an expected value $\mu(degrees) = E(X) = 62$, a std. deviation $\sigma(degrees) = \sqrt{Var(X)} = 26$ and $\alpha \in [0, 180]$. The choice of a right or left direction as referring to the previous trajectory is determined using a uniform probability density distribution, thereby yielding a probability density distribution for the angle α.

$$P(X = \alpha) = \frac{1}{2}\left[P(X = \alpha, v = \mu) + P(X = \alpha, v = \mu)\right] \tag{2.36}$$

The new position can now be computed. Indeed, this algorithm was presented in a 2-D domain, but can be generalized to a higher dimensionality easily.

2.4.3 Simplex Downhill Algorithm

The method developed by [Nelder and Mead, 1965], known as Simplex Downhill (SD) is based on a geometrical evaluation of the cost function, defined in $\mathbb{R}^N$, in $N + 1$ points that forms the vertex of a *N-dimensional* simplex. The simplex adapts itself to the local landscape, and contracts on to the final minimum. It is a derivative-free method for local search that exhibit a fast convergence and a considerable resistance to local minima entrapment. The algorithm can be implemented very easily:

1. A set of at least *N+1* x_k vertex are randomly initialized in the solution space.

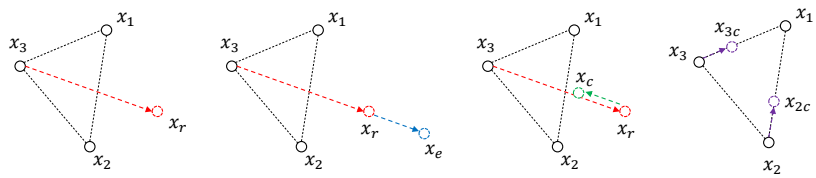


Figure 2.7: From left to right, the various operations of the SD algorithm: Reflection, Expansion, Contraction, Reduction.

.

2. The cost function is evaluated in each x_k position, and the indexes k are re-ordered so that x_1 is the best solution, and x_{N+1} is the worst.

3. The centroid x_o(geometric mean) position of all points, except x_{N+1} is computed.

4. Compute **reflected** point $\mathbf{x}_r = \mathbf{x}_o + \alpha(\mathbf{x}_o - \mathbf{x}_{n+1})(\alpha > 0)$

5. If the reflected point is better than the second worst, but not better than the best, i.e.:$f(\mathbf{x}_1) \leq f(\mathbf{x}_r) < f(\mathbf{x}_n)$, then obtain a new simplex by replacing the worst point $\mathbf{x}_{n+1}$ with the reflected point $\mathbf{x}_r$, and go to step 2.

6. If the reflected point is the best point so far, $f(\mathbf{x}_r) < f(\mathbf{x}_1)$, then compute the **expanded** point $\mathbf{x}_e = \mathbf{x}_r + \gamma(\mathbf{x}_r - \mathbf{x}_o)(\gamma > 0)$. If the expanded point is better than the reflected point, $f(\mathbf{x}_e) < f(\mathbf{x}_r)$ then obtain a new simplex by replacing the worst point $\mathbf{x}_{n+1}$ with the expanded point $\mathbf{x}_e$, and go to step 2. Else obtain a new simplex by replacing the worst point $\mathbf{x}_{n+1}$ with the reflected point $\mathbf{x}_r$, and go to step 1. Else (i.e.

reflected point is not better than second worst) continue at step 5.

7. Here, it is certain that $f(\mathbf{x}_r) \geq f(\mathbf{x}_n)$. Compute contracted point $\mathbf{x}_c = \mathbf{x}_o + \rho(\mathbf{x}_{n+1} - \mathbf{x}_o))(0 < \rho < 0.5)$. If the contracted point is better than the worst point, i.e. $f(\mathbf{x}_c) < f(\mathbf{x}_{n+1})$ then obtain a new simplex by replacing the worst point $\mathbf{x}_{n+1}$ with the contracted point $\mathbf{x}_c$, and go to step 2. Else go to step 8.

8. For all but the best point, replace the point with $\mathbf{x}_i = \mathbf{x}_1 + \sigma(\mathbf{x}_i - \mathbf{x}_1)$ for all i $\in \{2, \ldots, n+1\}$. Go to step 2.

Indeed, the size of the original simplex can influence dramatically the final outcome of the problem. The algorithm lacks effective qualities to escape a local minima (being deterministic by nature), and for this reason it is listed in this work as a "local search" technique. Launching this algorithm in the neighborhood of a good solution is critical to ensure the correct convergence towards the optimum.

2.4.4 Simulated Annealing

The Simulated Annealing (SA) algorithm [Aarts and Korst, 1988, Hwang, 1988] is included in this work for the sake of completeness, although it can not be considered the best approach for a local search. In general, (SA) is similar to a Random Walk method: the solutions are progressively found by a random movement in their neighborhood. What characterize the SA is the decision process of moving to the next candidate solution or not. The algorithm is inspired by the natural process of annealing that exist in crystal lattice. This process "heals" the defects that naturally occurs in crystal

lattice through successive heating and cooling stages. The fewer the defects, the better the solution. Initially, the heating/cooling process is performed at high temperature, and it is probable that the new lattice configuration (i.e. the new solution) may be worse than the previous one. Gradually, the temperature is lowered, and the reconfiguration occurs with progressively increasing selectivity (i.e. the algorithm becomes more greedy). The pseudo-algorithm goes as follows:

1. Create a set of randomly distributed solutions s_k
2. Evaluate the fitness $f(s_k)$ for each solution.
3. Generate, for each solution, a neighbor solution s_k^n and compute its fitness $f(s_k^n)$.
4. According to the switch probability $P\left(f(s_k), f(s_k^n), T\right)$, move to the candidate solution or not.
5. Decrease the temperature T. Go to 2.

The core of the algorithms are, indeed, the choice of the probability function $P\left(f(s_k), f(s_k^n), T\right)$ and the method to generate the neighbor solution s_k^n. Indeed, the probability of switching states should be proportional to the temperature T, so that the algorithm will progressively become more and more convergent. It should be related as well to the cost function evaluations. Yet, as long as the temperature is high, the probability of an inconvenient move should be greater than zero. In this way, we can ensure that the algorithm will not be stuck in local minima. The choice of the neighbor is very problem specific, but a common practice is to reduce the distance between s_k, s_k^n

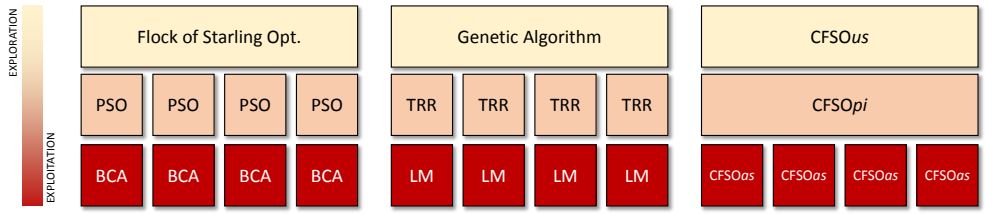


Figure 2.8: From left to right: MeTEO, r-MeTEO and CFSO^3 parallel strategies. Wider blocks indicate the Master node, smaller square the Slave nodes.

.

with the temperature, thus emphasizing the gradual convergence even further. Indeed the SA can exhibit, by dimensioning its parameter accordingly, either exploration or exploitation capabilities. However, using this algorithm to explore the solution space can be rather costly since no information is exchanged from one solution to the other. On the other hand however, this algorithm very easy to implement, and can be an interesting simple alternative to more complex local search techniques.

2.5 Hybrid Algorithms

Practical non-linear optimization problems can not be solved by a single algorithm. Several strategies have been proposed (and reviewed in this work) to make explorative algorithms more convergent and local search algorithm less prone to local minima entrapment. The best results however are achieved by combining different algorithms altogether, creating an *hybrid* algorithm. The basic strategy for a working hybrid algorithm is to follow a 2-3 stage approach:

- **Candidate Region Identification**: Use an algorithm with high exploration capabilities to identify areas were minimum (both local and global) may exist.
- **Local Search Pre-Positioning** *(optional)*: Use an intermediate algorithm in the region to find an optimal starting position for the local search, where the cost function is as convex as possible.
- **Local Search Exploitation**: Use a powerful local algorithm (even linear), initialized in the region previously identified, to converge towards optimum.

A dramatic advantage of hybrid algorithms lies in their natural predisposition for parallel implementation. Indeed, it is possible to use a single Master node to explore coarsely the solution space and perform region identification, and then deploy through Slave nodes local search algorithms that work in parallel in the candidate regions. Three strategies will be discussed in the following: MeTEO, r-MeTEO and CFSO[3]. The three strategies are summarized in Fig 2.8.

2.5.1 MeTEO and r-MeTEO

The algorithm proposed in [Fulginei et al., 2012] has been called MeTEO to point out its *Me*tric-*T*opological and *E*volutionary inspiration. In fact, it is based on a hybridization of two heuristics coming from swarm intelligence: the FSO (topological swarm), the PSO (metric swarm) and the BCA, that although not having a collective behavior that shows its better performances

in local searches. The present approach uses the FSO to explore the solution space, the PSO to investigate subspaces in which the global optimum could be present and finally the BCA to refine solutions. A parallel strategy is implemented: the FSO is permanently running, and each time it finds a possible solution, the PSO is launched. After that, it is transformed in BCA, and so on. A final important computational strategy completes the present approach: the fitness function is deliberately made worse just in those suitable narrow regions in which the FSO has decided to launch PSO–BCA. This fitness modification (FM) aims to prevent FSO from coming back in an already explored subspace. The FM is inspired to the famous Tabu-search algorithm [Glover, 1989] and in particular to the Tabu list. Since the global optimum is coincident with the smallest value achievable for the fitness functions, the FM has to ensure that the new fitness function must never indicate again a suspected regions if it has been already detected by the FSO. In fact, the FM consists in adding to the past fitness function a positive Gaussian function centered into the best co-ordinates found by FSO at the current iteration. It is important to remark that the effect of the FM acts just on the Master node, i.e. it is valid just for FSO, whereas the fitness function holds the original starting expression for both PSO and BCA working on the Slave nodes. A reduced version of this parallel hybrid strategy was implemented in [Lozito and Salvini, 2014] for a problem of model identification that did not require all the computational power of the original algorithm. This new strategy, called r-MeTEO, is composed by three algorithms: the first one is a GA, running on a Master, used for region identification. Then, on the slaves, two algorithms in cascade are used: the Trust-Region Reflective (TRR) and

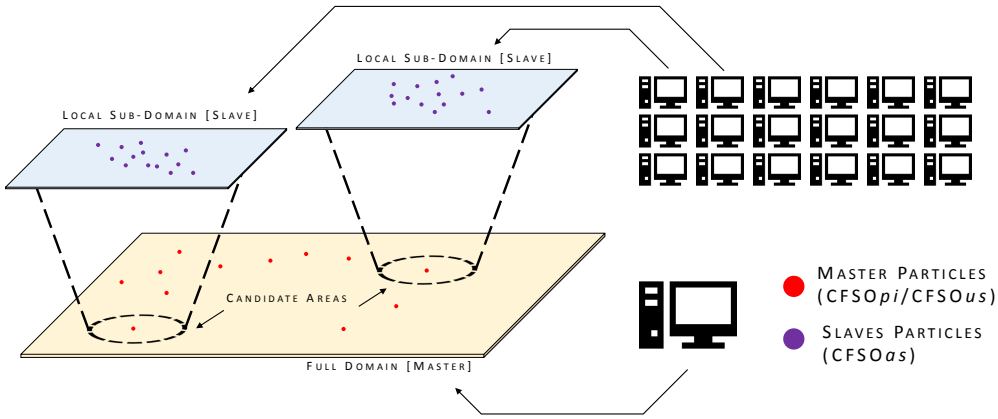


Figure 2.9: Conceptual representation of the CFSO^3 algorithm: the original domain for the solution space (yellow) is explored by the master using a CFSOpi algorithm. Once a candidate area is found by a particle, a slave is dispatched to explore the subdomain (light blue) using CFSOas algorithm.

.

the Levemberg-Marquardt. The former is a simple algorithm to pre-position the LM in a suitable area. The LM on the other hand is specifically thought for nonlinear least squares, and performs the final optimization task. Indeed r-MeTEO is more exploitation-oriented than exploration-oriented, but for the problem at hand (i.e. the model identification), where the search domain was pretty much well known, a robust and deep convergence was the most desirable attribute for the optimizer.

2.5.2 CFSO^3

It is understandable that the CFSO can exhibit both exploration and exploitation capabilities according to the poles of the flock. Both divergence,

oscillation and convergence can be obtained. Each of these behaviors can be used to create a hybrid strategy that features both exploration and exploitation capabilities.This strategy is thought to run on parallel architecture by the algorithm called CFSO^3 [Laudani et al., 2013b]. The CFSO^3 algorithm runs on master-slave parallel cluster. The master runs the CFSO configured to exhibit oscillating behaviors (CFSOpi). Its purpose is to explore the solution space and find candidate areas where a further investigation can be performed by the slaves. They run the CFSO with asymptotically convergent particles (CFSOas) that is initialized near the candidate area identified by the master. Each time the master identifies a new area, a slave is assigned to explore it. In the meantime, to increase the exploration capabilities of the master algorithm further, the poles are temporarily switched to an unstable configuration (CFSOus) to escape from the local minimum (the found candidate area). The strategy is depicted in Fig 2.9.

Part II

Implementation of Soft Computing Techniques on Embedded Platforms

Chapter 3

Embedded Soft Computing: Platforms

Several soft computing techniques can be used with great advantage in engineering applications featuring embedded devices. Indeed, the computational costs involved in the use of a Soft Computing technique can be rather convenient if compared, for example, to hard computing one in applications like modeling and control. However, embedded applications usually require a real-time approach to the problem (also known as *"time critical"*). Several embedded devices have very limited computational capabilities (if compared to high-level environments like a PC) and for this reason, the implementability of the aforementioned techniques is strictly limited to its ability in solving the problem at hand in an acceptable time frame.This concept is, of course, common to hard-computing techniques as well. But whereas for hard-computing techniques any possible tweak or speed-up is going to be strictly related to the problem at hand (since white-box models are specific),

soft-computing techniques shares their advantages across different scenarios. A faster, more accurate, or less memory-occupying algorithm can usually be used on several different problems. This flexibility is especially present for ANN, that are the main focus of the following work. When embedded implementation is involved, three possible environments can be considered. The first one are devices running an operative system like the Intel MiniPC or the linux based Raspberry Pi. The second ones are microcontroller units running a real-time OS or no operative system at all. The third ones are complex digital circuits implemented on a Field-Programmable Gate Array (FPGA). The three solutions will be discussed in this chapter, with advantages and disadvantages.

3.1 High Level Devices

High level devices are all-in-one boards generally mounting a powerful microprocessor unit, defined System-on-Chip. The boards may rely on external memory for boot and non-volatile storage, or may present an on-board flash as well. Since most of these devices are used as single board computers, they usually feature convenient interfaces like USB, HDMI for video output and Ethernet/Wi-Fi networking. The implementation capabilities for this devices are the closest to the one that can be achieved on a standard PC. In terms of raw computational speed, the resources of these systems are very high, almost close to a low-end PC. However, most of these resources are used for OS management, and code execution is subject to kernel distribution of computational resources. It is not possible to grant the real-time

execution of code. For this reason, the choice of this class of devices is only used if there is a strong necessity for accessory resources (networking, USB interfacing) that would make the implementation on a lower level device inconvenient. Coding on these platforms can be performed using (almost) all the options available on a normal PC. Algorithms can be implemented at low level if a C compiler is available for the CPU architecture (and usually is), or more powerful interpreted languages can be used. Matlab is still mostly a prerogative of personal computers, however it is possible to compute `.exe` binaries that runs natively on specific boards. Mathematica code requires its specific kernel to run, however, it has been recently included in the OS distribution *"New Out Of Box Software"* or `NOOBS` for the Raspberry Pi (see below). One last very common programming language that is widely used for embedded computation is Python. This is a very simple and readable interpreted language that allows the creation of complex programs with few code lines thanks to the powerful libraries included in it. Indeed, the Python interpreter must be installed on the device. Regardless of the coding platform used, several interesting options exists in terms of board choice. A comprehensive discussion on all the possible devices is beyond the purpose of this work, thus, a selection of the most interesting ones is going to be presented.

3.1.1 Raspberry Pi

Raspberry Pi is a small, cheap ARM-based PC for education and hobbyists that Runs Debian GNU/Linux from an SD card. The size of the board (schematic can be seen in Figure 3.1) is about the one of a credit card, although some smaller releases have been proposed. Most recent releases are

Table 3.1: Raspberry Pi Characteristics

Operating System	Linux (e.g. Raspbian), RISC OS, FreeBSD, NetBSD, Plan 9, Inferno, AROS
CPU	700 MHz single-core ARM1176JZF-S (model A, A+, B, B+, CM)
Memory	256 MB (model A, A+, B rev 1) 512 MB (model B rev 2, B+, CM) 1GB (Pi 2)
Storage	SDHC slot (model A and B), MicroSDHC slot (model A+ and B+), 4 GB eMMC IC chip (model CM)
Graphics	Broadcom VideoCore IV
Power	1.5 W (model A), 1.0 W (model A+), 3.5 W (model B), 3.0 W (model B+) or 0.8 W (model Zero)
Price	US$25 (model A, B+), US$20 (model A+), US$35 (RPi 1 model B, RPi 2 model B), US$30 (CM)

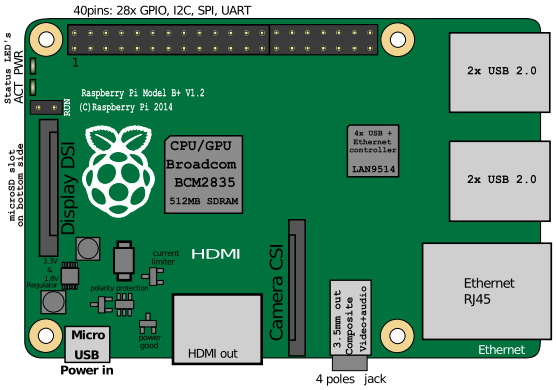


Figure 3.1: Raspberry Pi 2 Board Overview.

.

based on the ARMv7 quad core processor, and mount up to 1GB of RAM on board (see Table 3.1 for details). The performances of the Raspberry Pi are comparable to the one of an old Pentium II PC. The default operative frequency is 700MHz, and by benchmark analysis, the first generation of the Pi had an equivalent performance of 0.041 GFLOPS.By itself, the CPU performance is comparable to a 300MHz Pentium II processor. The GPU has 1Gpixel/s with 24GFLOPS processing capabilities. These can be used for parallel computing using CUDA/OpenCL programming language. This device is very well suited for implementation of parallel clusters, especially due to the very low power consumption. A cluster of 64 Raspberry Pi Model-B achieved a result of 1.14 GFLOPS with a total consumption of 216W. For scientific computation, a very important advantage related to this board is that the Mathematica environment from Wolfram is natively installed in the OS. The Mathematica language is completely embedded with the device, and it is

possible to control directly low level functions of the board (e.g. the GPIO interface) from inside the Mathematica environment itself. Computation can be performed efficiently using Python scripts as well. The libraries `NumPy` and `SciPy` contains functions to efficiently implement linear algebra (useful for ANN implementation) and several algorithms for numerical computation. Packages in `SciPy` includes:

- Clustering algorithms
- Physical and mathematical constants
- Fast Fourier Transform routines
- Integration and ordinary differential equation solvers
- Interpolation and smoothing splines
- Input and Output
- Linear algebra
- N-dimensional image processing
- Orthogonal distance regression
- Optimization and root-finding routines
- Signal processing
- Sparse matrices and associated routines
- Spatial data structures and algorithms

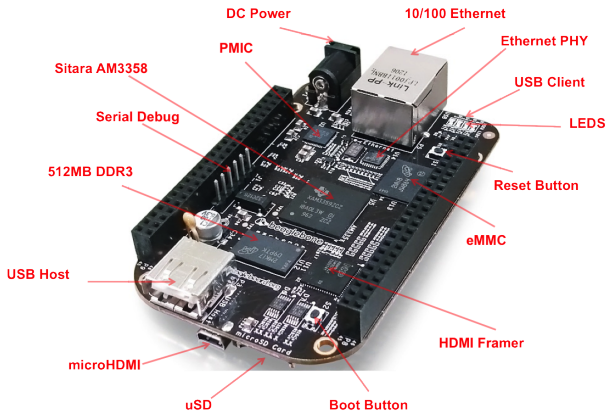


Figure 3.2: BeagleBone Black Overview.

.

- Special functions
- Statistical distributions and functions
- C/C++ integration

3.1.2 BeagleBoard

BeagleBoard is the name of a family of devices based on the Texas Instruments microprocessor Sitara™ARM Cortex-A8 processor. Sitara processors acts as bridges between the peripheral richness of a microcontroller unit and the speed and performance of a microprocessor. The A8 series includes among its peripherals an LCD controller, CAN, Gb EMAC switch, 2x USB w/PHY, integrated industrial protocols and touch screen control while retaining a clock speed up to 1GHz. A co-processor is present as well to manage real-time tasks. The **P**rogrammable **R**eal-Time **U**nit Subsystem and **I**ndustrial **C**ommunication **S**ub**S**ystem (PRU-ICSS) consists of dual 32-

bit RISC cores (Programmable Real-Time Units, or PRUs), data and instruction memories, internal peripheral modules, and an interrupt controller (INTC). The programmable nature of the PRU-ICSS, along with their access to pins, events and all SoC resources, provides flexibility in implementing fast real-time responses, specialized data handling operations, custom peripheral interfaces, and in offloading tasks from the other processor cores of the system-on-chip (SoC). Several operating systems are compatible with the Sitara processor: Linux, Android, Windows CE, StarterWare and RT-OS from Texas Instruments. By default, BeagleBoards mount a Linux distribution (Debian) out of the box. Numerical computation is aided by the presence of the NEON floating-point accelerator, which is a particular family of SIMD (Single Instruction Multiple Data) processor. SIMD processors where introduced to accommodate the need of processing large amounts of data that is less than word-sized. 16-bit data is common in audio applications, and 8-bit data is common in graphics and video. Most of the embedded MCU operates on 32-bit microprocessors, and for this reason, part of the computation units remains unused when processing this kind of data. Still, even if unused, they continue to consume power. SIMD technology try to make a better use of the available resources by using a particular architecture that supports an instruction set able to operate on multiple data elements of the same type and size at the same time. To give a quantitative example, an hardware that would normally add two 32-bit values, is able to perform four parallel additions of 8-bit values in the same amount of time as well. The ARMv7 architecture introduced the Advanced SIMD extension that defines groups of instructions that operates on vectors stored in 64-bit D, doubleword, reg-

Table 3.2: BeagleBoard Characteristics

Operating System	Linux, Minix, FreeBSD, OpenBSD, RISC OS, Symbian, Android
CPU	Sitara ARM Cortex-A8 with frequency 600 MHz to 1 GHz
Memory	128 MB to 512 MB
Storage	SDHC slot , MicroSDHC slot
Graphics	PowerVR SGX
Power	2.0 W
Price	US$95 to $149

isters and 128-bit Q, quadword, vector registers. The implementation of the Advanced SIMD extension used in ARM processors is called NEON, and this is the common terminology used outside architecture specifications. NEON technology is implemented on all current ARM Cortex-A series processors. NEON instructions are executed as part of the ARM or Thumb instruction stream. This simplifies software development, debugging, and integration compared to using an external accelerator. Traditional ARM or Thumb instructions manage all program flow and synchronization. Analogously to the Raspberry Pi, several options are available in terms of OS (with multiple unix distributions) and coding platforms. A breakout of the main characteristics for the board can be found in Table 3.2.

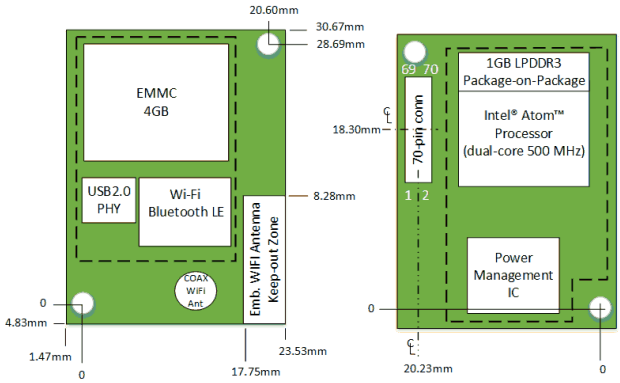


Figure 3.3: Intel Edison Overview.

.

3.1.3 Intel Edison

Intel Edison is an interesting alternative for high level programming since it features a processor (Intel Atom) compatible with x86 instruction set. The board is extremely small as can be seen from Fig. 3.3 and has been thought for a direct integration with the *Arduino Uno* microcontroller envinronment through specific breakout boards. Intel Edison's small compute package enables connectivity with Wi-Fi and Bluetooth LE, and also has LPDDR2 and NAND flash storage, as well as a wide array of flexible and expandable I/O capabilities. The board's main SoC is a 22 nm Intel Atom "Tangier" (Z34XX) that includes two Atom Silvermont cores running at 500 MHz and one Intel Quark core at 100 MHz (for executing RTOS ViperOS). The SoC has 1 GB RAM integrated on package. There is also 4 GB eMMC flash on board, Wi-Fi, Bluetooth 4 and USB controllers. The board has 70-pin dense connector (Hirose DF40) with USB, SD, UARTs, GPIOs. The price of the device is around 50$. It runs Yocto Linux with development support for Arduino IDE,

Eclipse (C, C++, Python), Intel XDK (NodeJS, HTML5), and Wolfram. If the module is not to be embedded on an *Arduino Uno* board, Intel release a breakout-board for easy access to the IO array and USB connectors.

3.2 Microcontrollers

Microcontroller units (MCUs) are highly integrated devices that enclose in a single chip a CPU and all the important sub-system needed for its operation. A MCU usually feature a CPU, several storage elements (RAM, ROM, FLASH, EEPROM), efficient digital I/O arrays, ADC and (seldom) DAC systems, timer modules and communication interfaces. The idea behind a MCU is to embed as much electronics as possible in a single device to increase system integration and reduce power consumptions. In general, CPU mounted on MCU are not as fast as the ones present on single-boards computer, and architectures are either RISC (ARM) based. Some CISC based MCUs exist (a notable example are the PIC series from Microchip Technologies) but the complexity of the instruction supported is still very low if compared to a x86 architecture. MCU do not run, natively, with an Operative System. It can be implemented, and there are some notable examples of very efficient Real-Time Operative Systems. However, since the operations required by a MCU are, in general, of lower-level, direct sequential execution of a program is, in general, the preferred method of MCU programming. This requires the use, in general, of a low-level programming language. Even if this may be a drawback in terms of easiness of programmability, it gives a direct control on the device real-time behavior. The most common languages used to program

a MCU are ASSEMBLY and C. ASSEMLBY is the lowest level programming language available, since each line of code has, in general, a one-to-one correspondence with a CPU instruction. This makes ASSEMLBY language architecture-specific (a RISC machine will have a completely different instruction set from a CISC machine). The main advantage of ASSEMLBY programming is, obviously, the complete control over the CPU operations. The disadvantage is the verbosity of the programs. Mathematical operations and memory structures are non-existent at CPU instruction level, and must be implemented by the programmer on the fly. Programming in C on the other hand has almost all the advantages of a low-level programming language, but with the bare minimum facilities introduced by the ANSI library. Indeed, programming a MCU in C requires a specific compiler for that architecture, and a set of libraries that makes addressing the MCU resources (peripherals, memory areas, system configuration strings) easily. In general, all the MCU adopt a memory-map approach for their peripherals. This means that, for example, to enable a particular function on the device ADC, a n-bit word will need to be written in a specific memory area of the MCU. This memory area is non-existent in fact, and is just used to give the programmers a mnemonic reference to enable and disable digital circuits. This reference is often given by `#define` directives in some header libraries given by the MCU producers. Those libraries are fundamental for a comfortable programming of the device, since modern MCU packs a very high number of peripherals with a 32-bit address space. The advantage of programming in C is the level of abstraction introduced by the libraries and the complex functions and structures that are inherently implemented in it.

This is the main disadvantage as well, since the abstraction layer makes it difficult to control directly the CPU behavior at instruction level. In practical terms, this is seldom a problem, however, for some very time-critical applications this issue may arise. Fortunately, modern compilers fully support *inline assembler*. This is a feature that allows ASSEMLBY code to be directly embedded in C code, allowing the programmer to use directly processor instructions when needed. As we did in the previous chapter, we are going to cover some examples of MCU boards suitable for implementation of soft-computing techniques for practical engineering applications.

3.2.1 High-End Microcontroller Units

High-end devices MCU are general purpose controllers based on an ARM CPU. Among different producers the CPU may be similar (at the moment of the redaction of this work, Cortex M4 are very diffused) and what makes a substantial difference is the quantity and the quality of the peripherals included in the MCU. Another key concept is the availability of development boards. Historically MCU programming required a stand-alone JTAG interface and some accessory electronics (a quartz/RC oscillator and a power supply, at least). Modern high-end MCU are sold with self-sufficient development boards that can be connected to a PC through USB out of the box. A very versatile family of development boards mounting high-end MCUs is the one proposed by Texas Instruments. These boards mounts the TM4C129 micro controller units with different peripherals added according to the model of the board. Both the MCU and the boards mounting it (referred as *Launchpads*™) are thought for Internet-of-Things (IoT) applications and real-time

control. The TM4C129 family are based on a Cortex M4-F CPU that integrates Ethernet MAC and physical (PHY) interface. Clock frequency is up to 120MHz and several peripherals for communication and control are available, as shown in Fig. 3.4. The device is extremely well suited for applications that involves heavy computational costs, due to the generous SRAM present and the complementary Floating Point Unit embedded in the CPU. An important addition are worth of note for this MCU (other aspects will be mentioned in the application sections) is the presence of a factory ROM where all the libraries involving the control functions for the MCU peripherals are stored. This ROM is used to avoid writing in the program memory of the MCU the routines needed to interact with the unit architecture, thus saving space for the program itself. This is a very precious characteristic since often soft-computing based algorithms have large memory footprints (e.g. the weights for the ANN). Different Launchpads model exists, with different optional characteristics: the most interesting ones are the *connected* models that includes Ethernet or Wi-Fi connectivity directly on board. All the Launchpad form factor is very small, making it simple to use it both as a development board to test code, and as a prototype board to include in a larger circuit. Several IDE are available for the Launchpad systems. All includes an editor, a compiler and a debugger environment. Texas Instruments provides its IDE as well, Code Composer Studio (CCS), that is based on a modified version of the Eclipse environment. CCS is free to download and can be used for free on Launchpads (and other development boards from Texas Instruments). For stand-alone devices (i.e. not on Launchpads) CCS has a memory limit (64kbytes) or requires a pay license.

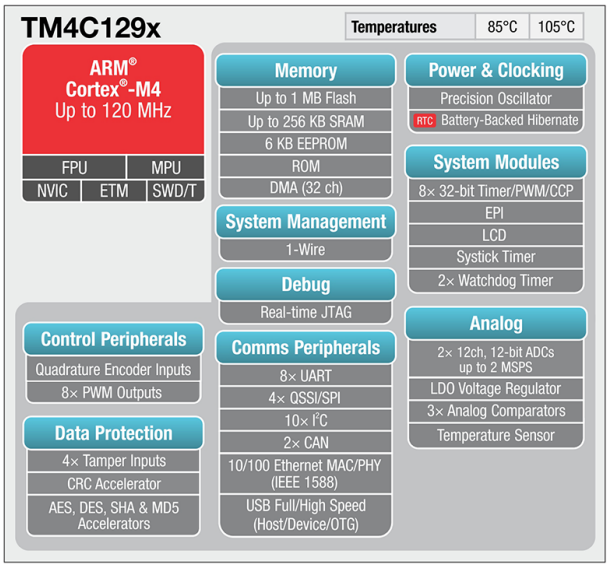


Figure 3.4: TM4C129 family MCU overview.

.

3.2.2 Digital Signal Processors

DSP are microcontroller units that sacrifices some versatility in terms of peripherals and interfaces in spite of a higher performance in real-time applications. They are architecturally designed to excel at mathematical operations and data movement. DSP for more demanding applications are, in general, multi-cores with high-speed interfaces and efficient internal fabrics allowing many devices to work together effectively. Low-end DSP have single cores, but with architectures engineered to work in conjunction with powerful libraries that speeds up numerical computations. A notable example is the C28x architecture from Texas Instruments, used for TMS320C28x DSP chips. For these devices, the `IQmath` library is used to speed up difficult arithmetic operations. Several of these DSPs mounts a CPU with added

FPU that speeds up the computation of floating point operations. Still, for control applications sometimes speed is more desirable than precision, and moving to a Fixed-Point arithmetic may be desirable. The `IQmath` library is collection of highly optimized and high precision mathematical Function Library for C/C++ programmers to seamlessly port the floating-point algorithm into fixed point code. `IQmath` uses the internal hardware of the DSP in the most efficient way to operate with 32 bit fixed-point numbers. Taking into account that all process data usually do not exceed a resolution of 16 bits, the library gives enough headroom for advanced numerical calculations.

3.2.3 Low-End Microcontroller Units

Low-End devices feature processors with low computational capabilities, but counterbalance this with reduced costs, smaller size and lower power consumption. Two interesting devices will be presented. The first one is the PIC18F6627 Microcontroller mounted on the SBC65EC Modtronix board as shown in Figure 3.5. The MCU belongs to the lowcost low-power PIC18/8-bit family, featuring a 4 kB RAM memory and a 96 kB reprogrammable flash memory; 12 AD converters; and SPI, I2C, and RS232/RS485 interfaces. The computational capabilities of this device are very limited, but the high resolution AD converter, coupled with the very low power consumption (as low as 0.2 μA in sleep mode) makes it suitable for battery-powered applications. In particular, the SBC65EC Modtronix board allows to communicate with the MCU through a web-server interface. The second device is the MSP430FRxx by Texax Instruments. This is an Ultra-Low Power device that is based on the use of FRAM (Ferroelectric RAM) for memory and storage. FRAM is a

Figure 3.5: SBC65EC Modtronix prototype board, mounting a PIC18F6627 Microcontroller Unit.

.

random access memory, meaning that each bit is read and written individually. This non-volatile memory is similar in structure to DRAM, which uses one transistor and one capacitor (1T-1C), but FRAM stores data as a polarization of a ferroelectric material (Lead-Zirkonate-Titanate). As an electric field is applied, dipoles shift in a crystalline structure to store information. The use of crystal polarization as opposed to charge storage enables state retention, lower voltage requirements (as low as 1.5V) and fast write speeds when compared against Flash, EEPROM and SRAM technologies used in typical MCUs. For this device, power consumptions can get as low as 0.02 μA.

3.3 FPGA

Field Programmable Gate Arrays, or FPGA, are re-arrangeable digital circuits. An FPGA is composed by a large quantity of programmable logic blocks that can be configured and arranged with different connections to

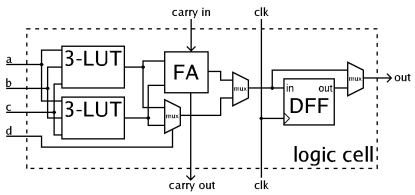


Figure 3.6: An example logic cell for an FPGA.

.

create a logic circuit that performs the desired operation. The blocks are flexible and can be used for different purpose. They are in general composed by a variable number of logic cells, like the one shown in Fig. 3.6. The configuration is dynamic, and is performed through a hierarchical set of connection that forms a matrix wiring the blocks together. Modern FPGA devices embed some "hard blocks" inside their architecture. These blocks are non-editable and are usually reserved for standard interfaces, processor cores, memory controllers and sometimes even ADC/DAC units to implement a mixed/signal (analog and digital) application.

3.3.1 HDL Languages Flow

FPGAs are not "programmed" in the classical sense of the term, since there is no processing unit performing operations in an FPGA. By the term "programming", when referred to an FPGA, the process of describing the implemented logic function is intended. The complexity of the logic function that can be implemented in an FPGA depends on the size of the FPGA itself. It can be as simple as a combinatorial function and as complex as a fully functioning CPU with added blocks for memory. The FPGA configuration is generally specified using a hardware description language (HDL), that is a

specialized computer language used to describe the structure and behavior of electronic circuits, and most commonly, digital logic circuits. Programming the FPGA through graphical interface is very common as well, especially when standard digital circuits and configurations are involved, or when the design requires a direct visualization. The most common HDL programming languages for FPGA are VHDL and Verilog. Both are widely supported by hardware producers in their IDE up to a point where the choice of the language is merely a matter of preference. For this work, all codes have been written in VHDL. VHDL programs does not describe directly the hardware that is going to be implemented on the FPGA, but it rather describe the desired "behaviour" that should occur in the device. To do this, it uses two constructs: entities and architectures. An entity is hollow box that can contain a logic circuit that should behave in some way: inputs and outputs of the box are defined. An example entity suitable for a 2-input combinatorial logic circuit can be:

```
entity MY_COMBINATORIAL is
port (
A : in std_logic;                    -- First input
B : in std_logic;                    -- Second input
O  : out std_logic);                 -- Output
end entity MY_COMBINATORIAL;
```

After the hollow box has been created, it is necessary to define the "wiring" that connects the interfaces to achieve the desired behavior. This is defined in the architecture:

```
architecture LOGIC of MY_COMBINATORIAL is
```

```
begin
O <= A and B;     -- O = A and B.
end architecture LOGIC;
```

The behavior described in the architecture is a simple AND port. When this VHDL code is compiled, this entity is translated into a Register Transfer Level (RTL) representation. RTL is a level of abstraction where a behavioral code is translated in a logic circuit composed by combinational logic and/or registers. This circuit can be simulated like any other logic circuit. To be implemented on silicon however, two more steps needs to be performed. The first one is the translation at GDL (Gate Description Level), where the RTL circuit is implemented using the logic components available in the FPGA (i.e. the one present in the logic blocks). This step, differently from the RTL translation, is device-specific. The RTL translation, along with the GDL one, are referred as Synthesis. The result is a *netlist* that describes a set of real components and their connection. Last step to be performed is the *Place and Route*. This steps maps the *netlist* in the FPGA internal circuitry. This last step is critical for two reasons. The first one is that a poor place and route strategy may result in a poor use of the FPGA resources (i.e. it is a problem very similar to PCB layout routing). The second one is related to signal propagation in the circuit. Gate delays should be accounted to ensure synchronous propagation of signals and clocks in the circuit.

3.3.2 Soft Processors

Modern FPGAs have enough resources to implement fully functional microprocessors as part of their architecture, along with the memory required to

run software. A microprocessor implemented inside an FPGA is usually referred as *Soft Processor*. The cost to implement this core in terms of FPGA resources is, in general, high. However the advantages of having an object able to process complex instructions sequentially simplify the application development considerably. Without a processor, the only way to create a circuit able to perform an "algorithm" (which is the principal task found in Soft Computing) is to implement, from zero, a Finite State Machine. This machine can be implemented to be extremely fast, however, it will lack the flexibility of a CPU executing C code. As we will see in the next sections, the software nature of Soft Processors makes them eligible for architectural tweaking, by adding or modifying processor instructions to better suit the application needs.

Chapter 4

Computational Costs of an ANN

In this chapter, the problem of the computational costs associated to the embedded implementation of an ANN are going to be analyzed. Two aspects are going to be covered. The first aspect is going to be the optimal choice of the AF of an ANN. This will be analyzed in spite of two aspects, the easiness of training and the reduction of the computational costs associated with the calculus of the ANN output. The second aspect is going to be the use of hardware accelerators, implemented in FPGA, to compute the ANN output.

4.1 The choice for a suitable AF

The mapping capabilities of a ANN are strictly related to the nonlinear component found in the AF of the neurons. Indeed, without the presence of a nonlinear activation function, the ANN would be a simple linear interpolator.

The nonlinear part of a NN is completely separated from the linear combination of the weighted inputs, thus opening a large number of possibilities for the choice of an activation function. The choice of a suitable activation function for a Feed Forward ANN, and in general, for a ANN, is subject to different criterions. The most common considered criterion are training efficiency and computational cost. The former is especially important in the occurrence that a ANN is trained in a general-purpose computing environment (e.g., using Matlab); the latter is critical in embedded systems (e.g., microcontrollers and FPGA) where computational resources are inherently limited. The following sections will be an extract from the work [Laudani et al., 2015], which is a comprehensive review on the topic. More details on the AFs presented, including a comparison of the results given by the cited authors in their works, can be found in the original paper.

4.1.1 AF for Easy Training

The commonly used back-propagation algorithm for ANN training suffers from slow learning speed. One of the reasons for this drawback lies in the rule for the computation of the ANN's weights correction matrix, which is calculated using the derivative of the activation function for the ANN's neurons. The universal approximation theorem [Cybenko, 1989] states that one of the conditions for the ANN to be a universal approximator is for the activation function to be bounded. For these reasons, most of the activation functions show a high derivative near the origin and a progressive flattening moving towards infinity. This means that, for neurons having a sum of weighted inputs very large in magnitude, learning rate will be very slow. A

detailed comparison between different simple activation functions based on exponentials and logarithms can be found in [Kamruzzaman and Aziz, 2002], where the authors investigate the learning rate and convergence speed on a character recognition problem and the classic XOR classification problem, proposing the use of the inverse tangent as a fast-learning activation function. The authors compare the training performance, in terms of Epochs required to learn the task, of the proposed inverse tangent function, against the classic sigmoid and hyperbolic tangent functions, and the novel logarithmic activation function finding a considerable performance gain. In [Ma and Khorasani, 2005], the proposed activation function is derived by Hermite orthonormal polynomials. The criterion is that every neuron in the hidden layer is characterized by a different AF, which is more complex for every neuron added. Through extensive simulations, the authors prove that such network shows great performance in comparison to analogous ANN with identical sigmoid AFs. In [Hara and Nakayamma, 1994], the authors suggest the combination of sigmoid and sinusoidal and Gaussian activation function, to exploit their independent space division properties. The authors compare the hybrid structure in a multifrequency signal classification problem, concluding that even if the combination of the three activation functions performs better than the sigmoid (in terms of convergence speed) and the Gaussian (in terms of noise rejection), the sinusoidal activation function by itself still achieves better results. Another work investigating an activation function based on sinusoidal modulation can be found in [Lee and Moraga, 1996], where the authors propose a cosine modulated Gaussian function. The use of sinusoidal activation function is deeply investigated and the authors

present a comprehensive comparison between eight different activation functions on eight different problems. Among other results, the Sinc activation function is proved as a valid alternative to the hyperbolic tangent, and the sinusoidal activation function has good training performance on small ANNs.

Fuzzy and Adaptive techniques

Other strategies adopted for fast convergence involve the use of Fuzzy Logic. In [Soria-Olivas et al., 2003], the authors define the hyperbolic tangent transfer using three different membership functions, defining in fact the classical activation function by means of the fuzzy logic methodology. The main advantage during the training phase is a low computational cost, achieved since weight updating is not always necessary.

One last strategy that is used to achieve fast convergence is based on adaptive AF, which requires for some assumptions on it to be overlooked in favor of a more efficient training procedure. Indeed, it has been seen that a spectral similarity between the activation function and the desired mapping gives improved performance in terms of training. The extreme version of this approach consists in having an activation function that is modified during the training procedure itself, creating in fact an ad hoc transfer function for neurons. The training algorithm for such networks requires taking into consideration the activation function adaptation, as well as the weights tuning. The authors in [Wu et al., 1997] propose a simple BP-like algorithm to train a NN with trainable AF and compare the training performance with a classic sigmoid activation function on both XOR problem and a nonlinear mapping.

4.1.2 AF for Fast Computation

The computational cost of an ANN can be split into two main contributions. The first one is a linear cost, deriving from the operations needed to perform the sum of the weighted inputs of each neuron. The second one, nonlinear, is related to the computation of the activation function. In a computational environment, those operations are carried out considering a particular precision format for numbers. Given that the AF is usually limited to a small co-domain, the use of integer arithmetic is unrecommended. Fixed-point and floating-point arithmetic are the most commonly used to compute the ANN elementary operations. The linear part of the ANN is straightforward: operations of products and sums are carried out by multipliers and adders (usually found in the floating-point unit (FPU) of an embedded device). The nonlinear part, featuring transcendental expressions, is carried out through complex arithmetic evaluations (IEEE 754 is the reference standard) that, in the end, still use elementary computational blocks like adders and multipliers. Addition and product with floating-point precision are complex and long operations, and the computational block that executes these operations often features pipeline architectures to speed up the arithmetic process. Although a careful optimization of the linear part is required to completely exploit pipeline capabilities, the ratio between the two costs shows, usually, that the linear quota of the operations is negligible when compared to the nonlinear part. In embedded environment, computational resources are scarce, in terms of both raw operations per second and available memory (or resources, for synthesizable digital circuits like FPGAs and ASICs). Since embedded applications usually require real-time interaction, the development of NN

applications in embedded environments shows the largest contributions in terms of fast and light solutions for AFs computation. Three branches of approaches can be found in literature: PWL (piecewise linear) interpolation, LUT (Lookup-Table) interpolation, and higher order/hybrid techniques.

Zero Order (LUT) Techniques

Approximation by LUT is the simplest approach that can be used to reduce the computational cost of a complex function. The idea is to store inside the memory (i.e., a table) samples from a subdomain of the function and access those instead of calculating the function. The table either can be preloaded in the embedded device (e.g., in the flash memory of a microcontroller) or could be calculated at run-time with values stored in the heap. Both alternatives are valid, and the choice is strictly application dependent. In the first case, the static approach occupies a memory section that is usually more available. In the second case, the LUT is saved in the RAM memory, which is generally smaller than the flash; however, in this case the LUT is adjustable in case a finer (or coarser) version is needed. A variation of the simple LUT is the RA-LUT (Range Addressable LUT), where each sample corresponds not only to a specific point in the domain, but to a neighborhood of the point.

First Order (PWL) Techniques

Approximation through linear segments of a function can be easily carried out in embedded environment since, for every segment, the approximated value can be computed by one multiplication and one addition. In [Tisan et al., 2009] four different PWL techniques (three linear and one quadratic

that will be discussed shortly) are analyzed considering the hardware resources required for implementation, the errors caused by the approximation, the processing speed, and the power consumption. The techniques are all implemented using the System Generator, a Simulink/Matlab toolbox released by Xilinx. The first technique implemented by the authors is called A-Law approximation, which is based on a PWL approximation where every segment has a gradient expressed as a power of two, thus making it possible to replace multipliers with adders [Myers and Hutchinson, 1989]. The second technique is the Alippi and Storti-Gajani approximation [Alippi and Storti-Gajani, 1991], based on a segmentation of the function in specific breakpoints where the value can be expressed as sum of power of two numbers. The third technique, called PLAN (Piecewise Linear Approximation of a Nonlinear Function), uses simple digital gate design to perform a direct transformation from input to output [Amin et al., 1997]. In this case, the shift/add operations, replacing the multiplications, were implemented with simple gate design that maps directly the input values to sigmoidal outputs. All the three techniques are compared together and against the classic LUT approach. The authors conclude that, overall, the best results are obtained through PLAN approximation.

Higher Order Techniques

Hybrid techniques try to combine both LUT and PWL approximations to obtain a solution that yields a compromise between the accuracy of the PWL approximation and the speed of the LUT. Higher order techniques push the boundaries and try to represent the AF through higher order approxima-

tion (e.g., polynomial fitting). In [Lozito et al., 2014b] authors propose a piecewise II-degree polynomial approximation of the activation function for both the sigmoid and the hyperbolic tangent AFs. Performance, resources required, and precision degradation are compared to full-precision and RA-LUT solutions. In [Kwan, 1992] a simple 2nd-order AF, which features an origin transition similar to the hyperbolic tangent, is proposed. The digital complexity of this function is in the order of a binary product, since one of the two products required to obtain a 2nd-order function is performed by a binary shift. A similar 2nd-order approximation is proposed in [Zhang et al., 1996].

4.1.3 Weights Transformation

Different papers shown in this survey pointed out advantages and drawbacks of using an AF instead of another one. Two very common AFs that are found in almost any comparison are the sigmoid activation function and the hyperbolic tangent activation function. Considering an embedded application, as the one suggested in [Lozito et al., 2014b], where the activation function is directly computed by a floating-point arithmetic chain of blocks, using a sigmoid AF instead of a hyperbolic tangent AF allows synthesizing the chain with less arithmetic units. However, as shown in several papers, the low derivative of the sigmoid AF makes it a poor candidate for training purposes when compared to the hyperbolic tangent. In this final note of the section, a set of transformation rules, for a single layer ANN with arbitrary inputs and outputs, is proposed. The rules allow modifying weights and biases of a NN so that changing the hidden layer AF does not change the NN output.

$$Sigmoid\ to\ Tanh \begin{cases} Hidden\ Bias & \left[B_H^L\right] = 2 \times \left[B_H^T\right] \\ Hidden\ Weights & \left[W_H^L\right] = 2 \times \left[W_H^T\right] \\ Output\ Bias & \left[B_O^L\right] = \left[B_O^T\right] - \left[W_O^T\right][1] \\ Output\ Weights & \left[W_O^L\right] = 2 \times \left[W_O^T\right] \end{cases} \tag{4.1}$$

$$Tanh\ to\ Sigmoid \begin{cases} Hidden\ Bias & \left[B_H^T\right] = 0.5 \times \left[B_H^L\right] \\ Hidden\ Weights & \left[W_H^T\right] = 0.5 \times \left[W_H^L\right] \\ Output\ Bias & \left[B_O^T\right] = \left[B_O^L\right] + 0.5\left[W_O^L\right][1] \\ Output\ Weights & \left[W_O^T\right] = 0.5 \times \left[W_O^L\right] \end{cases} \tag{4.2}$$

Eq. 4.1 and 4.2 allow an easy translation of weights and biases to switch between sigmoid and hyperbolic function AFs ([1] denotes a $1-by-N$ vector, where N is the number of output neurons). A possible strategy to exploit these relations would be to create a tanh based ANN in a general-purpose environment, like Matlab. Then, after the ANN has been trained, translate the weights in sigmoid form to obtain a ANN that features a simpler AF.

4.2 Hardware Accelerators

As shown in the previous section, it is imperative to optimize the computation of the AF. Of the several approaches presented investigated, two different implementations are investigated: a high level solution to create a neural network on a soft processor design, with different strategies for enhancing

the performance of the process; a low level solution, achieved by a cascade of floating point arithmetic elements. Comparisons of the achieved performance in terms of both time consumptions and FPGA resources employed for the architectures are presented. The following is an extract from the work [Lozito et al., 2014b], additional details can be found in the original paper.

4.2.1 Nios II/f Soft Processor with Custom Instruction

The first solution attempted to implement the network on FPGA makes usage of the soft core processor Nios II/f, released by Altera as a crypted core. This core can be synthesized with as low as 1600 logic elements (LE) and supports a maximum frequency of 140 MHz. After synthesis and programming on the FPGA device, the soft core itself can be programmed and debugged in C using a JTAG tool chain running inside an Eclipse environment. This soft core processor supports hardware integer multiplication and division, and up to 255 custom instructions definable by the designer. These custom instructions can be defined at RTL level using VHDL or Verilog code, and are synthesized as parallel blocks of the internal Nios II Arithmetic Logic Unit (ALU) as shown in Fig. 4.1.

When a custom instruction is called from the instruction memory of the Nios II, the operands are transferred in the custom logic and, according to the type of custom instruction (combinatorial or sequential) the result is collected after a definite number of clock cycles. The design is based on the Nios II/f core, modified to have a Floating Point ALU and two Custom Instructions. The system works with a 100MHz clock, which is replicated by means of a PLL with a phase shift of 3 ns to control an external 8 Mb SDRAM. As shown

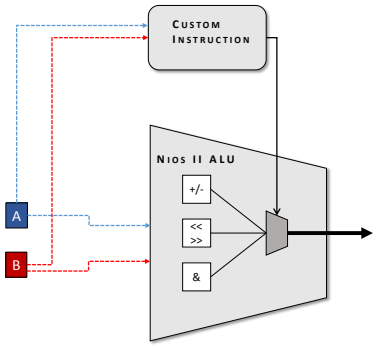


Figure 4.1: Synthesis of a custom logic instruction in parallel to the Nios II ALU.

in Fig. 4.2, the processor was equipped with a standard JTAG interface for programming and a Performance Counter to determine the execution time of the implemented code. The Floating Point ALU was the standard block from the library released by Altera as a part of the Quartus II environment. Two Activation Function LUT(s) were created in VHDL (one for the Tansig and one for the Logsig) and imported into the design as user-made custom instructions.

For the LUT implementation, the AF was not sampled with a uniform and constant spacing between the sampling points. This is because the activation function assumes almost constant values near the saturation points, making it wasteful to choose a fine sampling in their proximity. On the other hand, near the origin, the slope of the function is very high, and a finer sampling may help in reducing quantization error. In different works, only two kinds of spacing are used: a fine one, near the origin, and a wide one, near the saturation branches. In this approach, a different technique is used: the

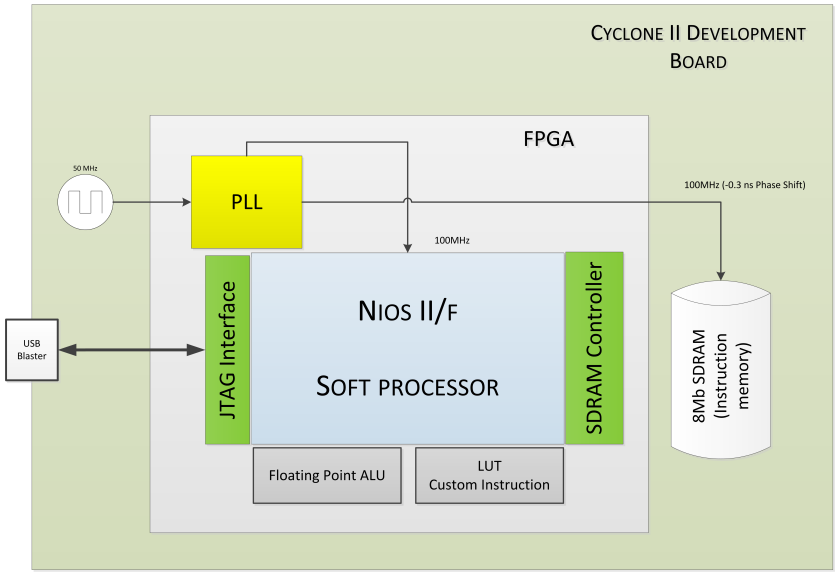


Figure 4.2: Implemented Soft Processor and Peripherals.

distance between a sample and the following one is inversely proportional to the slope of the function in the sampled point. This yields a finer sampling near the origin, gradually getting wider near the saturation points. The Logsig function was sampled with 256 values between 16 and +16, while the Tansig, being an odd function, was sampled for positive arguments only, with 256 values between 0,2 and. Using these values, a VHDL combinatorial code was written and simulated in Altera ModelSim environment for RTL analysis. The implemented block has a single floating point input, that is split in sign, exponent and mantissa. Through the use of a suitable IF-THEN-ELSE chain the input value addresses a specific entry in the LUT, that is propagated as output. If the input value magnitude is bigger than the saturation values, a suitable constant value is propagated as output. Since the Tansig, near the origin, can be approximated to the bisector of the first

quadrant, values smaller than 0.2 are directly propagated in output (thus approximating the function linearly). The synthesis result of this IF-THEN-ELSE structure is a very long chain of comparators. Propagation of the signal through this chain can be long, so a tunable delay of 4 clock cycles was introduced to ensure result stability (the delay is controlled by a simple counter that can be modified to suit the size of the LUT). As a term of comparison with this hardware based technique, a polynomial interpolator was implemented as well. Indeed, The basic operations of floating point math are greatly fastened by the presence of a Floating Point ALU, and other than speeding up the Multiplier-Accumulation part of the FFNN, this hardware module can be used to compute a polynomial approximation of the activation function A group of second-degree polynomials was chosen to fit the activation functions. The coefficients of the polynomials were determined in Matlab environment through the use of the Curve Fitting Tool. Both the functions were fitted only for positive arguments. For the Logsig polynomial fitting, a function (denoted as 5PY-L) composed by the superposition of 5 second-degree polynomials, has been implemented. Even if the Logsig function is not odd, a partial symmetry is present. This was exploited for its negative arguments: first, the value of the function is calculated considering the absolute value of the input; then, if the input is negative, the calculated value is subtracted by the value of 1. For the Tansig polynomial fitting, two functions, composed by 4 and 5 second-degree polynomials have been implemented, respectively denoted as 4PY-T and 5PY-T. This time, since the Tansig is an odd function, the argument is considered in absolute value, and the sign is directly propagated to the output.

Table 4.1: Nios II/f test results on FFNN with Logsig activation functions.

Function	**MSE**	**Average time/sample**
Floating Point	0.0000 (ref)	650μs
LUT (Logsig)	0.1598	17.5μs
5PY-L	0.0075	185μs

Table 4.2: Nios II/f test results on FFNN with Tansig activation functions.

Function	**MSE**	**Average time/sample**
Floating Point	0.0000 (ref)	715μs
LUT (Tansig)	0.0053	17.5μs
4PY-T	0.0039	142μs
5PY-T	0.0018	174μs

The design was used to simulate a FFNN trained on the function $y = x^2$, and was tested on a vector of 2048 linearly spaced inputs between 5 and +5. The results in Tab. 4.1 and Tab. 4.2 show the performance in terms of mean squared error (MSE) and execution time of the different solutions proposed above. As a reference for execution time, the performance of a FFNN featuring a full precision software implementation of the activation function is shown in both tables.

4.2.2 NN Core Implementation

In the following section a solution based on low level architecture is presented. The proposed design was used for the implementation of the same

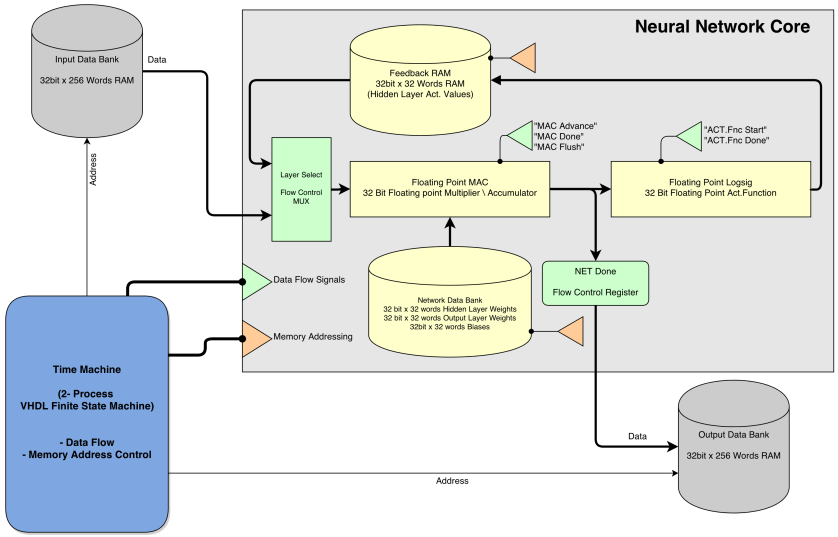


Figure 4.3: NN Core schematic diagram.

FFNN previously described. The proposed design (shown in Fig. 4.3) is an arithmetic core composed by high performance floating point arithmetic blocks developed by Altera, whose data flow is controlled by a Finite States Machine (FSM) written in VHDL.

The arithmetic core is composed by 3 blocks: a multiplier-accumulator (MAC), an activation function, and a feedback RAM. These three blocks constitute a suitable base to build a Neural Network. The first block computes, for each neuron, the weighted sum of the inputs. The second block has the results of the first block as inputs, and computes the activation values for the hidden layer. The third block, receiving the output from the activation function block, stores the values from the hidden layer. These values are then sent through a MUX back into the MAC block for the output layer computation. Both input and output data of the FFNN are stored in RAM blocks that are accessible through JTAG interface using the Quartus II software. The whole

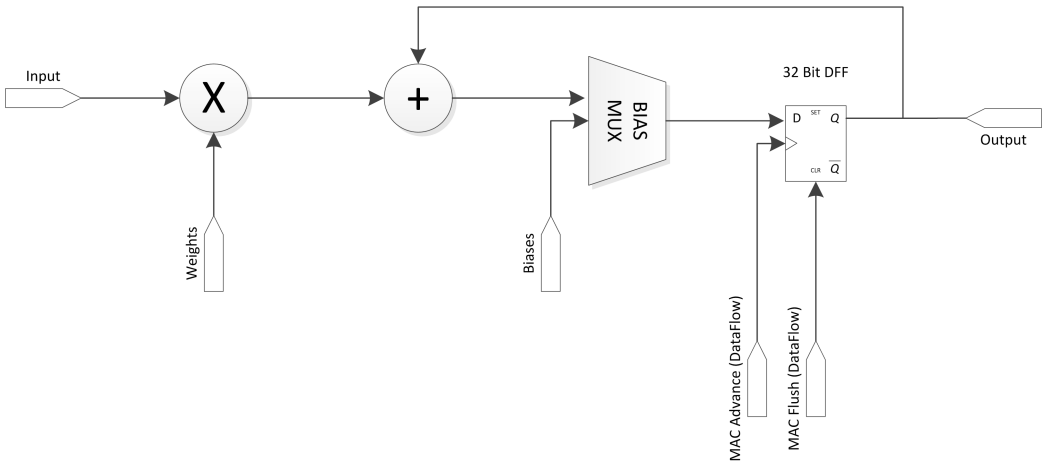


Figure 4.4: MAC block diagram.

Core and the data banks are controlled by a free running 2-Process Finite State Machine "Time Machine" using data flow control signals and address registers. Internal data flow of the core is regulated by a number of 32-bit wide MUXes and DType Flip Flops (DFFs). The design was implemented on a EP2C20F484C7 Cyclone II FPGA mounted on a DE2 – Development Board. After synthesis and fitting the full design occupied about 5000 logic elements (LE) and all the 52 hardware multipliers present on the FPGA.

The computation of the arithmetic core begins by loading the first sample from the Input Data Bank into the MAC block. The core contains into its internal memory the weights and biases of the FFNN. This memory is addressed directly by the Time Machine control block. Since the MAC is computing the hidden layer, each neuron will have a bias value that must be added to the weighted input. This bias value is preloaded into the 32-bit DFF accumulator using the Bias MUX. Inputs and weights are multiplied and the results are added to the preloaded bias, as shown in Fig. 4.4.

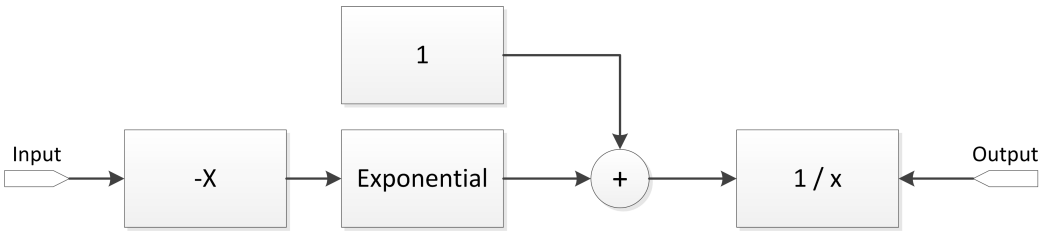


Figure 4.5: Logsig block diagram.

Since the hidden layer has only one input, the MAC is done for the first neuron, and the result is propagated to the next block, where the activation function is computed. In this section, a logical not is operated on the MSB of the input, changing its sign. The result is sent to an exponential arithmetic block whose output is connected to an adder that sums the result to the constant value of 1 (see Fig. 4.5). The result is then inverted and the activation value of the first neuron is finally written in the Feedback RAM. This operation is repeated for the 10 neurons, filling the RAM with the activation values of the hidden layer. Then, the Time Machine switches the Layer Select MUX so that the MAC block is now connected to the Feedback RAM. The bias of the output neuron is preloaded in the accumulator, and the MAC computes the weighted sum of all the activation values from the hidden layer. This is the output result of the network, and is saved in the Output Data Bank.

Data processing from input to output needs to be managed by some sort of control block, responsible for synchronizing the data-flow and, were needed, perform memory addressing. In a traditional programming language, like C, a popular approach to create such controller is to use a finite state machine (FSM). In its simplest form, a FSM is a set of code blocks, each identifying a

particular function (e.g. "load data from RAM", "sum input A and input B", "transpose array C"), inside a switch/case structure. If the FSM is the sole controller of the system, the switch/case structure is confined in an endless loop. The variable controlling the switch is updated at the end of each code block, ensuring that every time the switch/case is evaluated the FSM will execute a specific code block (i.e. will be in a known and definite state). This rather simple approach is not as straightforward in HDL languages, since the code is not executed by a processor, thus not inherently sequential. Hardware, emulating the processor sequential behaviour, must be created.

A possible approach is to create an instruction counter whose value is increased at every clock edge. By using a net of comparators, when a particular value is assumed by the instruction counter, specific logic functions (states) are executed. Creating the FSM in this way grant an important advantage: since the instruction counter is updated on clock edge, the FSM can work synchronously with the other elements in the design. This is very important when some blocks in the design have definite input-output delays, since the FSM can be programmed to remain in a "wait" state until the output is ready to be propagated to the next block. In VHDL this architecture can be defined by the use of two code blocks (processes), one sequential and one combinatorial. The first one is responsible for the instruction counter increase at every clock edge, and is synthesized with a counter register. The second one is responsible for decoding the instruction counter into actual logic signals, and is synthesized with a network of comparators. The cycle of operations performed by the FSM is obviously limited, once the last operation is performed (i.e. the last output value has been loaded in the Output

Table 4.3: Best performance comparison.

Function	**MSE**	**Avg. time/sample**	**F.Time (2048 Samples)**
NN Core (50 MHz clock)	0.0000 (ref)	$154\mu s$	$315.4ms$
NN Core (100 MHz clock)	0.0000	$78\mu s$	$159.8ms$
5PY-L	0.0075	$185\mu s$	$378.8ms$
LUT (Tansig)	0.0054	$17.5\mu s$	$35.79ms$

Data Bank), the FSM will reset and start over. With a 50 MHz clock, the computation of a single sample takes about $150\mu s$.

4.2.3 Solution Comparison

In the Tab. 4.3, a comparison of the best performances among solutions is presented. At full precision, the NN Core design provides a quite lower computation time than the Nios II design. Moreover, by doubling the clock frequency through a PLL (thus using the same frequency used for the Nios II designs, 100 MHz) the computation time drops at $78\mu s/sample$. However, if full precision is not needed (and the choice of a particular activation function is not mandatory), implementing a FFNN based on a Tansig activation function yields the lowest computation time, using the Nios II design. In particular, implementing a LUT yields the best results in terms of precision over computation time.

Table 4.4: Nios II/f design main resources usage by entity.

Entity	**LC Comb.**	**LC Reg.**	**DSP**
Nios II CPU	2382	1799	4
FPU	5125	3783	7
LUT (Tansig)	1815	4	0
LUT (Logsig)	1617	4	0

Table 4.5: NN Core design main resources usage by entity.

Entity	**LC Comb.**	**LC Reg.**	**DSP**
MAC Block	1015	620	7
Logsig Block	2784	1874	45
FSM	205	130	0

In Tab. 4.4 and Tab. 5 the resources, in terms of dedicated Combinatorial and Register logics (LC Comb. and LC Reg.) are shown. The high level solution is expensive in terms of resources usage, peaking with 15098 logic elements (LE) if both the LUT(s) are implemented as custom instructions. This is generally not necessary, since only one of the activation functions is used in the network. By excluding the Logsig LUT from the synthesis the LE usage drops to 12699 LE. The low level solution, although completely saturating the DSP blocks of the FPGA, is contained in 5037 LE.

4.3 Linear Acceleration for MISO ANN

In this section an efficient approach to compute the result from a multiple-input-single-output Neural Network using floating-point arithmetic on FPGA is described. The following is an extract of [Laudani et al., 2014d]. More details can be found in the original paper. The proposed algorithm focus on optimizing pipeline delays by splitting the "Multiply and accumulate" algorithm into separate steps using partial products. It is a revisit of the classical algorithm for NN computation, able to overcome the main computation bottleneck in FPGA environment. The proposed algorithm can be implemented into an architecture that fully exploits the pipeline performance of the floating-point arithmetic blocks, thus allowing a very fast computation for the neural network. Consider the implementation on an embedded system of a simple SISO Feed Forward Neural Network, the performance of the system has two main bottlenecks. The first one lies in the number of floating point multiply-and-accumulate (MAC) operations. The second one in the complexity of the non-linear activation function. The latter problem has been addressed throughly in this work, focusing on optimizing the performance of a NN in embedded system by reducing the complexity of the activation function. The optimization of the MAC operations however is often overlooked. On a low-end microcontroller unit, the floating-point operations are carried out in software, leaving the code performance to the capacity of the assembly compiler libraries. Many high-end microcontrollers on the other hand figure a Floating Point Unit with dedicated multipliers that can speed up the MAC part of the neural network computation. Several comparative studies have been made on this topic, however, few options

are available to the programmer to optimize the code itself.

Working on a FPGA embedded system a different approach can be followed to enhance the Neural Network performance. The MAC algorithm is usually carried out by using a multiplier, an adder and a one-word register to store the partial MAC results . Since floating-point arithmetic is more complex than integer arithmetic, computation blocks usually trade performance for footprint (in terms of logic elements used) by adding a pipeline. Since the process of multiply-and-accumulate is inherently sequential, the adder must wait for a new multiplication result each time it updates the cumulative sum, yielding poor results in terms of throughput. The burden of this problem grows exponentially if the Neural Network has multiple inputs (MISO) or more hidden layers

4.3.1 Understanding the MAC Bottleneck

In HDL language, the multiplier and adder for floating point numbers are usually implemented with an arithmetic logic block that shows a pipeline delay. The delay of the blocks is the time (expressed in clock cycles) between the first input getting in and the firstoutput going out. After this initial delay has expired, the data will output with the same cadence (even clockwise) it was input. Suppose a multiplier block with 3 clock cycles delay. If data is presented at $t = 0, 1, 2, 3, 4$, the result of the multiplication can be collected at $t = 3, 4, 5, 6, 7$. Now suppose a "multiply-and-accumulate" block as the one shown in Fig. 4.6, where the multiplier and the adder have both 10 clock cycles delay. The first two factors are loaded in the multiplier at $t = 0$. At $t = 10$, the output is available at the input of the adder, which

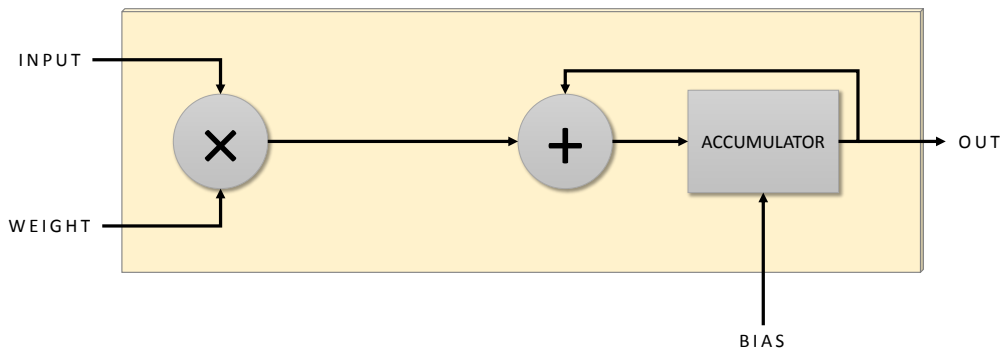


Figure 4.6: A "multiply-and-accumulate" block with input, weights and biases.

can start the computation and complete it after 10 cycles ($t = 20$). Once the computation is completed, the adder can update the 32-Bit accumulator. Since only after the accumulator has been updated the next sum can begin, exploiting the blocks pipeline is impossible: if two data were sent in the multiplier at $t = 0, 1$ the second product would be ready at $t = 11$. At this time, even if the adder could start a pipelined sum, one of the operands (the one from the accumulator) would not be ready for another nine clock cycles. In conclusion, the adder delay limits the throughput of the MAC.

4.3.2 A more efficient approach for MAC

The solution to this problem is twofold: first, the multiplier timing must be decoupled from the adder; second, a particular approach for the adder must be followed to allow a synchronization between the end of a sum and the beginning of a new one. The first problem can be easily solved by storing the partial products of the neurons in a buffer memory. Since the results

are all stored, the data can be sent into and collected from the multiplier at full speed. The synchronization of the adder is a more complex matter. For each of the m neurons in the layer, a total of n partial products must be summed. The total sum of n numbers requires $n-1$ operations, with a variable number of operations requiring the result of a previous sum. For easiness, we will suppose the worst case where all the operations but the first one will require a previous result.

- $n_1 + n_2 = n_{12}$
- $n_{12} + n_3 = n_{123}$
- $n_{123} + n_4 = n_{1234}$
- ... and so on.

Supposing the adder has a 10 cycles delay, between each sum, at least 10 cycles must pass. However, during this delay the adder can compute the sums for the other neurons.

- $n_1 + n_2 = n_{12} \quad Neuron_1$
- $n_1 + n_2 = n_{12} \quad Neuron_2$
- $n_1 + n_2 = n_{12} \quad Neuron_3$
- ...
- $n_1 + n_2 = n_{12} \quad Neuron_9$
- $n_1 + n_2 = n_{12} \quad Neuron_{10}$

- $n_{12} + n_3 = n_{123} \quad Neuron_1$
- $n_{12} + n_3 = n_{123} \quad Neuron_2$
- ... and so on.

Obviously, this calls in some considerations on the pipeline delay size. If the pipeline delay d is equal to m,the system will be inherently synchronized. If d is larger than m, the computation time for the sum will be the same as the one required for a layer of d neurons. If d is smaller than m, the system could go out of synchrony, since the results from the previous computation would arrive too early. It is possible however to increase the pipeline depth of any block by adding a cascade of delay registers to the output. In conclusion, the larger between d and m determines the computation time for the sum.

4.3.3 MISO NN Core

The algorithm proposed in the previous section was implemented in FPGA environment. The architecture, shown in Fig. 4.7, is composed by a cascade of high performance arithmetic blocks developed by Altera, whose dataflow is controlled by a dedicated ControlBlock running a Finite State Machine coded in VHDL. The proposed architecture can compute a 10 inputs MISO NN with 10 neurons in the hidden layer. Four main blocks can be identified in the architecture: the MAC block, the Activation Function, the Control Block, and the Memory block. Each of these blocks will be discussed.

MAC The MAC block is responsible for the execution of the enhanced "multiply-and-accumulate" algorithm previously show. Input data and weights

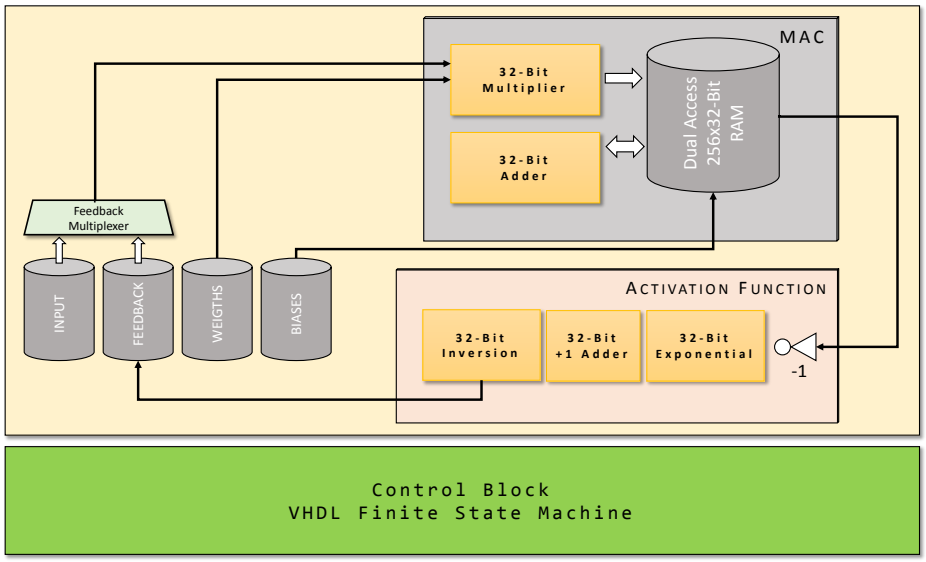


Figure 4.7: The MISO NN Core as implemented on FPGA.

are fed to the multiplier, which stores the results in the 256x32-Bit RAM. As it can be seen, whereas the connection between RAM and multiplier is unidirectional, the connection with the adder is bidirectional. The reason lies in the "-and-accumulate" optimization: the adder must be able to read the partial products while writing the results in the RAM itself. Amongst other reasons, this is why the RAM must be of dual access type. As it can be seen, the Adder is isolated from the outside architecture even if the biases for the neuron could be sent directly as input. Instead, when needed, the biases are (pre)loaded in the RAM.

Activation Function This block computes the activation function of the ANN. The first operation needed is a sign change, achieved through a logical not on the MSB of the input, representing the sign of the floating-point

value. The result is fed to an exponential arithmetic block whose output is connected to an adder that sums the result to the constant value of one. The result of the sum is then inverted and the result (the activation value of the neuron) is propagated to the output. Since in this case the operations are independent, the three pipelines were used in simple cascade without the need for a buffer RAM. A note on the choice for the activation function is necessary: this particular function was chosen, instead of the more common tansig activation function, for the easiness of implementation in FPGA environment: the arithmetic for a tansig activation would require an additional adder and multiplier. However, as shown before in this work, it is possible to transform an ANN from tansig to logsig based by simple weight adaptation relationships.

Memory The memory block is a set of 32-Bit RAM units. The blocks holds the input values, the weights and the biases of the NN. These blocks are accessible by JTAG interface for online programming. The Feedback block is used to store the values from layer to layer: it is connected via multiplexer to the multiplier along with the NN input. As the first layer computing is completed, the second layer receives as input the results from the previous layer.

Control Block A control block is required to manage data processing from input to output. The block needs to synchronize the dataflow according to the pipeline delays, thus sending and recovering data at specific times. Along with this, it is responsible for memory addressing. The data flow of the entire system is organized by the control block. The following routines are executed

in overlap to maximize the throughput; The type of operation performed can be of either loading a pipeline with operands, or recovering the results from it. The only exception, beside the initialization, is the READ_PP routine, where the pipeline is loaded and unloaded continuously.

For the Hidden Layer we have:

- LOAD_MULT_F: The input RAM and the weights RAM are connected to the multiplier. All the inputs and weights are sent to the multiplier clock-wise.

- LOAD_PP: The output of the multiplier is connected to Port A of the dual access RAM. The partial products are stored in the RAM.

- READ_PP: This routine computes the sum of the partial products. Port A of the RAM is connected to the input of the Adder. Port B of the RAM is connected to the outputof the Adder. The accumulated result is written in base 10 memory positions: neuron 1 in 0x000, neuron 2 in 0x00A, neuron 3 in 0x014.

- LOAD_BS: Port B of the RAM is connected to the biases RAM. The biases are loaded in base 10+1 positions: neuron 1 in 0x001, neuron 2 in 0x00B, neuron 3 in 0x015.

- SUM_BS: This routine sums the biases to the accumulated partial products. Note that after this routine is completed, the base 10 positions of the dual access RAM will have the net values of the neurons.

- LOAD_AF: Port A of the RAM is connected to the input of the activation function chain. The netvalues of the neurons are sent through

the chain clock-wise.

- WRITE_AF: The Feedback RAM is connected to the output of the activation function chain. The results from the activation function are stored in the RAM.

And, for the outut layer:

- LOAD_MULT_F2: Analogous to LOAD_MULT_F, but with the output layer weights.
- LOAD_PP: As above.
- READ_PP: As above.
- LOAD_BS_F2: Analogous to LOAD_BS, but with the output layer biases.
- SUM_BS: As above.

4.3.4 SISO-MISO-MIMO scalability

It can be seen that the output layer routines are as time-consuming as the hidden layer routines. This should be not expected if the network is a MISO architecture. For example, a 10-10-1 network would require 100 products and sums for the hidden layer, but only 10 products and sums for the output layer. In this architecture, both the hidden layer and the output layer performs the same number (100) of operations, but in the output layer, 90 of the 100 weights are null.This may appear as a waste of time resources, but two aspects must be discussed. First, only the multiplier routines throughput

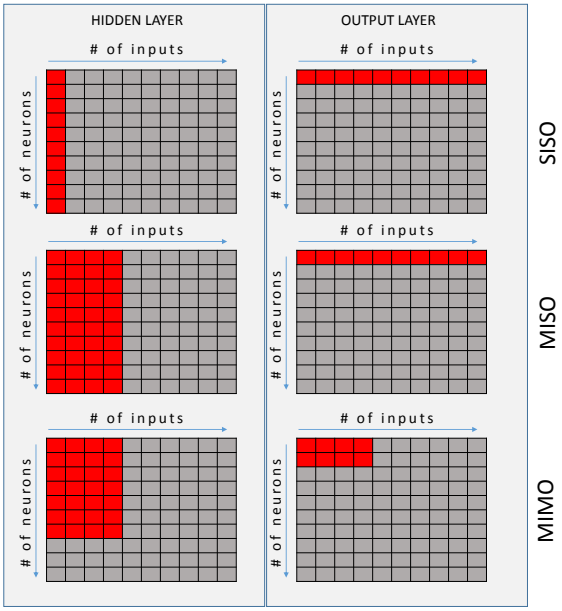


Figure 4.8: Memory arrangement for architecture tuning.

could be increased. The adder routines throughput, as explained above, are limited by the adder pipeline delay. The delay cannot be reduced because of the hidden layer size. Another adder, with a specific pipeline delay for the output layer could be implemented, but at the cost of incrementing the used resources. Second, by computing the output layer with the same routines used in the hidden layer, the architecture can be used for both a MISO NN and a MIMO NN. In fact, by modifying the weights and biases null elements, this architecture can work as a SISO, MISO or MIMO. In Fig. 4.8, three of the possible configurations for the network are shown. Each couple of tables represents the RAM containing the weights of the NN.

The memory content was rearranged on rows of 10 elements. The red elements are non-null values, the gray elements are null values. The Hidden Layer table is filled analogously as the weight matrix (or column, in case of single input), with the conceptual difference that the matrix size corresponds to the size of the non-zero area in the table. In the Output Layer table, the width of the non-zero area is bound to the height of the Hidden Layer table (i.e. the number of hidden neurons), while its width is the number of outputs desired for the network. The Bias memory arrangement is simpler and does not require a graphic explanation. The column vector of the biases for the Hidden Layer goes from 0x000 to 0x00A, while the one for the Output Layer goes from 0x00B to 0x014.

4.3.5 Performance

To evaluate the performance of the proposed architecture, the three configurations (SISO, MISO, and MIMO) were implemented in different environments, profiling the code execution. All the networks had 10 neurons in the hidden layer. The MISO architecture had 10 inputs and 1 output. The MIMO architecture had 10 inputs and 10 outputs.Four different environments were chosen for the validation: Matlab, an x86 architecture, Nios II Soft Processor for FPGA and a Cortex M4A ARM Microcontroller.

Matlab The NNs were computed by taking advantage of the matrix arithmetic functions implemented in Matlab language. Matlab was running on a Windows 7 64-bit Core i3 machine with 2 GB of RAM.

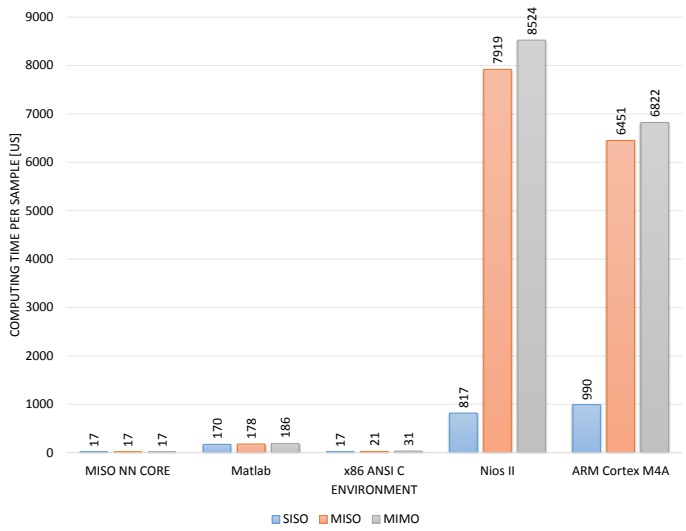


Figure 4.9: Performance comparison between different programming environments.

C code on x86 The network was implemented using nested for loops and single precision floating point functions from the ANSI C library. The code was compiled using Tiny GCC Compiler and was executed on the same machine as above.

FPGA Nios II Soft Processor This is the same soft-processor used in the previous section for AF acceleration. The processor has both hardware multipliers/dividers and a dedicated Floating Point Unit. The clock for this processor was 100 MHz. The same C code used for x86 architecture was used for this test.

Cortex M4A ARM Microcontroller The LM4F120XL microprocessor by Texas Instruments was used as deployment platform. This microcontroller

runs at 80MHz and, like the Nios II, has a dedicated Floating Point Unit. As above, the same C code for the x86 architecture wasused.

MISO NN Core The computation time for the MISO NN Core is 17us on a 50MHz clock. The computation time is the same for both SISO, MISO and MIMO architectures. The resource footprint for the design is 5492 Logic Elements, about 100kbit of memory, and 52 9-bit multipliers. The results are shown as a comparative histogram in Fig. 4.9.

Part III

Applications

Chapter 5

MPPT for Photovoltaic Devices

The use of Photo-voltaic (PV) devices for centralized and/or distributed power generation requires a control system to correctly and efficiently harvest the electrical power generated by the panels, because of the intrinsic non linearity of the current-voltage characteristic (I/V) of the panel. The maximum power point (MPP) on this curve (V_{MPP}, I_{MPP}) lies near the knee of the curve: from this point any higher voltage level causes a current dropping and a lower power extraction, as a consequence. Then, the control system device tries to keep the device working point as close as possible to the point of maximum power. It performs this task by controlling the input resistance of the DC/DC converter that is (usually) connected to the output of the PV device. On the other hand, the calculation of the V_{MPP}, I_{MPP} point it is not trivial since the I/V characteristic of a particular panel depends on (among constructive parameters) environmental conditions such as temperature (T) and irradiance (G) as shown in 5.1. The relationship between the I/V characteristic of a panel and the parameters is still an open problem. Even if

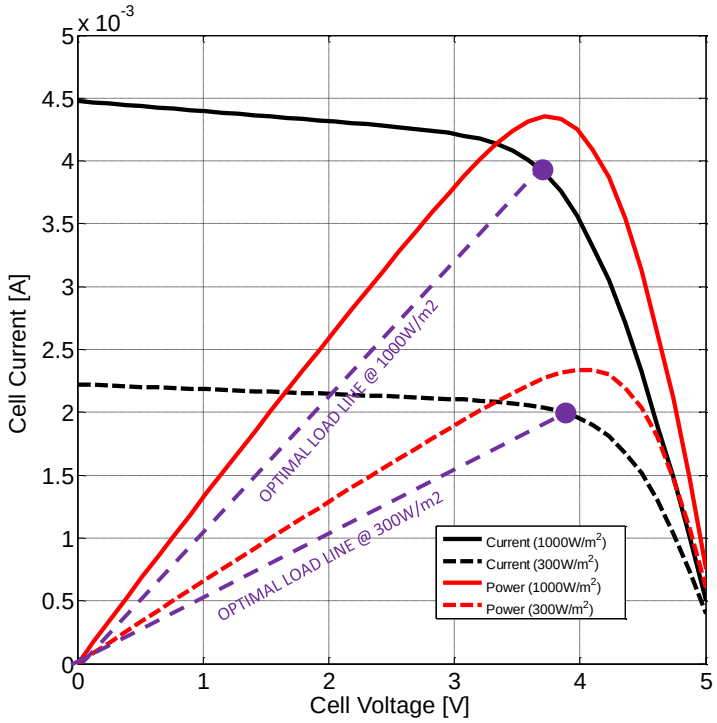


Figure 5.1: Current vs Voltage and Power vs Voltage characteristics with two different environmental conditions (irradiance and temperature).

a circuital model of the device is currently adopted, the characteristic (and the V_{MPP}, I_{MPP} point) can be calculated analytically, anyway. The model generates a set of characteristics which are function of the environmental conditions and, whereas the temperature is easy to be measured, the irradiance is complex to be measured correctly.

Different approaches proposed both in scientific literature and in industrial applications try to avoid the model identification by tracking the optimum power point iteratively, changing the working point until an optimum condition is reached. Common examples are the Perturb and Observe (PO)

method and the Incremental Conductance (IC) method. Both of these methods do not require an a priori knowledge of the irradiance value. As a drawback, their step-like control causes bad convergence on the MPP, oscillations and a non fixed computational time. A completely different approach can be followed by using a NN (Neural Network) for the MPP traking (MPPT). This approach, recently proposed in literature [Mahamad and Saon, 2014, Vincheh et al., 2014, Liu et al., 2013, Punitha et al., 2013, Carrasco et al., 2013], uses a set of different curves generated by the mathematical model for different temperature and irradiance conditions. This set is used to train a NN. Also this approach requires a temperature measurement, but needs just a single-iteration for the solution to the MPPT problem. Moreover the solution is robust versus abrupt changes in climatic conditions, whereas the iterative PO and IC usually fail to converge. In this chapter, two approaches to implement an ANN based MPPT controller will be presented. The first one is based on a high-end device (an ARM microcontroller) and use both a MLP architecture and a FCC. The second is implemented on a low-end 8-bit device.

5.1 The One-Diode Model for PV Devices

The basic component of a complex PV device is the PV cell. One of the most commonly used model to represent it is a circuit composed by a single diode, as the one shown in Fig. 5.2. Different models, with different parameters, exists in literature. Among these, the one featuring a single diode and five control parameters is by far the most used one. This model is addressed in literature as **One-Diode Model** or **Five Parameters Model**. By com-

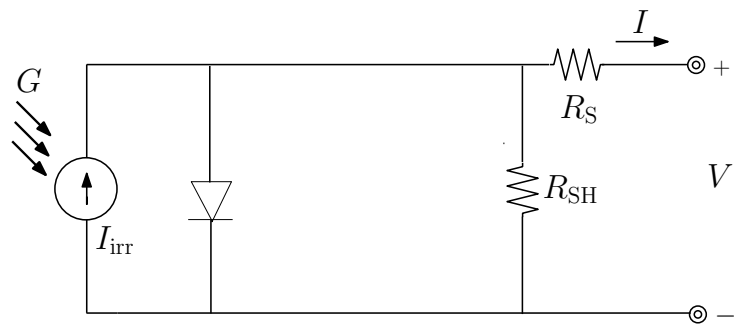


Figure 5.2: Circuital "one diode" model for a PV device.

bining a number of cells in series, a PV module is done. By connecting a number of PV modules in series/parallel, a PV Array is obtained. The I/V characteristic of the cell is shown in Eq. 5.1

$$I = I_{Irr} - I_0 \left[exp\left(\frac{q\left(V + IR_S\right)}{nkT} \right) - 1 \right] - \frac{V + IR_S}{R_{SH}} \tag{5.1}$$

where I_{IRR} is the irradiance current, I_0 is the saturation reversal current of the diode, R_S is the series resistance, R_{SH} is the shunt resistance, n is the ideality factor, T is the temperature, k is the Boltzmann constant and V and I are the current and voltage of the cell. Considering N_S cells in series and N_P modules in parallel, a simple extension of this equation makes it valid for modules and arrays.

$$I = I'_{Irr} - I'_0 \left[exp\left(\frac{q\left(V_A + I_A R'_S\right)}{N_S nkT} \right) - 1 \right] - \frac{V_A + I_A R'_S}{R'_{SH}} \tag{5.2}$$

Were the new parameters shown in Eq. 5.2 are defined in Table 5.1 and V_A and I_A are the voltage and the current of the array respectively.

In Eq. 5.1, the variables $n; R_S; I_{irr}; I_0, R_{SH}$ can be expressed as a function of temperature, irradiance, and their reference value at SRC (Standard Reference Conditions, $T_{ref} = 25°C$ and $S_{ref} = 1000W/m^2$). The following

Table 5.1: Symbol equivalence for PV Arrays

PV Module Parameter	PV Array Parameter
$N_P I_{IRR}$	I'_{IRR}
$N_P I_0$	I'_0
$N_S R_S / N_P$	R'_S
$N_S R_P / N_P$	R'_P

equations can be used to update the $n; R_S; I_{irr}; I_0, R_{SH}$ parameters by their temperature and irradiance dependence.

$$\begin{cases} n = n_r ef \\ R_S = R_{S,ref} \\ I_{irr} = \frac{S}{S_{ref}} \left[I_{irr,ref} + \alpha_T (T - T_{ref}) \right] \\ I_0 = I_{0,ref} \left[\frac{T}{T_{ref}} \right]^3 exp \left[\frac{E_{g,ref}}{kT_{ref}} - \frac{E_g}{kT} \right] \\ E_g = 1.17 - 4.73 \times 10^{-4} \times \frac{T^2}{T+636} \\ R_{SH} = \left(\frac{S_{ref}}{S} \right) R_{SH,ref} \end{cases} \tag{5.3}$$

where $E_{g,ref}$ is the band gap energy at $T_{ref} = 298.16K$.

5.1.1 Model Identification

The problem of identification for the circuital model of a PV device is an actually open problem in the scientific community. The goal is, indeed, to find a particular model behaving like the device it is representing. Two strategies have been proposed in literature to solve such problem, one us-

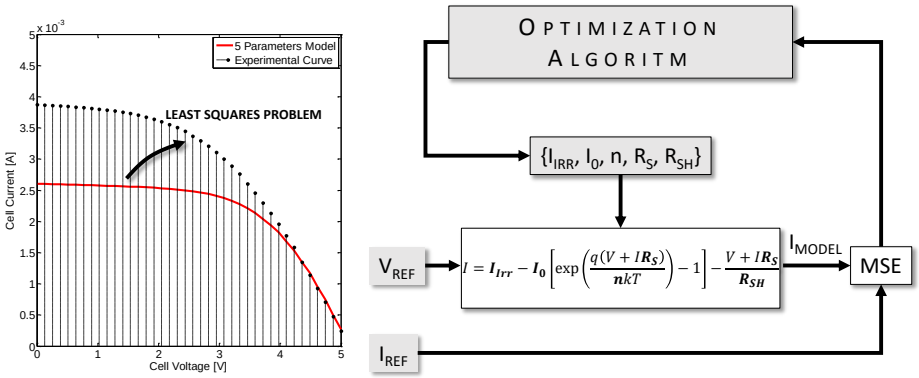


Figure 5.3: Identification of the 5 Parameters Model through experimental curves.

ing experimental curves, the other using values from the device datasheet. Identification from experimental curves basically consists in solving a Least Squares problem, where we take into account the discrepancy between the response (current) of the model given an excitation (voltage), and the corresponding response from the experimental data. A schematic representation of the problem can be seen in Fig. 5.3. The problem is non-linear, multi-modal, and prone to finding non physical solutions. This means that the quintuplet of parameters $n; R_S; I_{irr}; I_0, R_{SH}$ found by the optimization algorithm may lack a physical meaning (i.e. they could be negative). Still, this technique is widely used in literature. Indeed, either the experimental data is taken at SRC, or the conditions of T and S should be known so that 5.3 can be used to derive the SRC model. The second technique of identification make use of datasheet values to create a system of non-linear implicit equations constraining the model to a single quintuplet of parameters. The values used by the datasheet are:

- Open circuit voltage V_{OC}.
- Short circuit current I_{SC}.
- Maximum power voltage and current V_{MPP}, I_{MPP}.
- Temperature coefficient α of open circuit voltage.

The first three values are used to impose, respectively, the passage of the model curve by the three points of open circuit, short circuit and maximum power. Then, a fourth equation is added, considering this time the power versus voltage curve $P(V)$. In this equation, a null derivative for the power

versus voltage curve is imposed in the point of maximum power. This equation is necessary because imposing that the model passes by V_{MPP}, I_{MPP} does not guarantee that such point will be the one of maximum power. For the last equation, several options are available. The curve may be constrained to pass on an additional arbitrary point, or the derivative near open circuit may be imposed. A common approach however is to use the temperature coefficient α to force a particular temperature behavior at open circuit. It has been seen [Laudani et al., 2014b, Laudani et al., 2013a] that the first three equations (Open Circuit, Short Circuit and Maximum Power Point passage) can be re-formulated as explicit relationships expressing three of the parameters $(I_{irr}; I_0, R_{SH})$ in function of the other two $(n; R_S)$. The advantages are several:

- The problem dimensionality is reduced to two parameters from the original five.
- The problem becomes convex.
- The $(n; R_S)$ parameters can be searched in a closed domain (*the original five parameter problem had a semi-open domain* $\{n; R_S; I_{irr}; I_0, R_{SH}\} \in \mathbb{R}^5 \geq 0$)

This method can be applied, with slight differences, to model identification through experimental curves. More details can be found in [Laudani et al., 2014a].

5.2 MPPT through MLP and FCC Neural Networks on ARM devices

The following is an extract of [Lozito et al., 2014a] where the problem of implementing the MPPT control system through a Neural approach has been discussed for high-end devices. For this work, two architectures have been used and the obtained results compared in terms of accuracy and computational costs. The choice of a NN architecture usually falls on the most common and simple network for non-linear classification and interpolation, the Multi Layer Perceptron (MLP). As shown in previous section, in this architecture, the neurons are organized in subsequent layers, with connections made according to the Feed Forward configuration (no connections between neurons of the same layer, and each neuron of a layer is connected to all the units of the next one). By definition, the MLP can have an arbitrary number of hidden layers. However, by choosing a to implement just a single hidden layer, the structure of the NN is univocally determined by knowing that layer size. Even if a MLP with a single hidden layer (arbitrarily large) can interpolate any given function, complex problems may require a very high number of neurons to be correctly solved, with risk of oversizing the network and losing in generalization capabilities [Fulginei et al., 2013]. For this reason, the problem at hand was approached with the FCC architecture as well. As previously shown, the FCC is composed by a sequence of single-neuron layers, and the output of each neuron is propagated both to the next layer, and to all the subsequent layers down to the exit neuron. The FCC enjoys the same property of the single hidden layer MLP: by knowing the

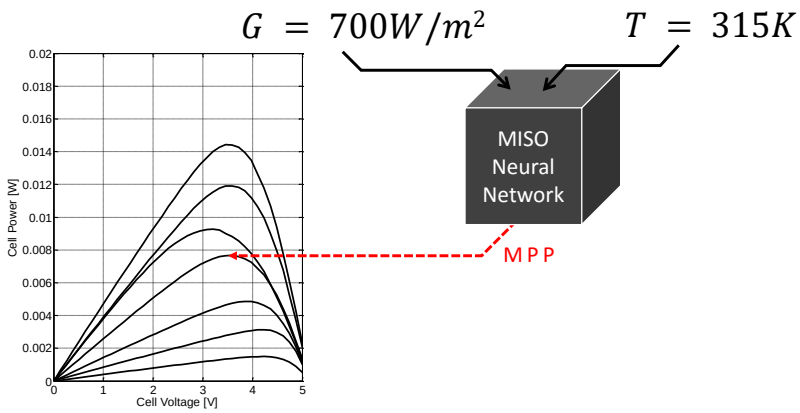


Figure 5.4: Estimating the maximum power point by means of temperature and irradiance.

number of neurons, the network architecture is univocally defined. Instead of training this network with the classical EB (error backpropagation) or LM (Levember-Marquardt) algorithms, the faster Neuron-by-Neuron algorithm was used.

5.2.1 Training Dataset and Neural Network Generation

Since the purpose of the ANN is to predict an optimal value V_{OPT} as close as possible to V_{MPP} for any given condition of G and T, the training pattern for the network should ideally have as inputs G and T, and as output the V_{MPP} specific for those environmental conditions, as shown in Fig. 5.4.

This is because, for each G and T, a characteristic I/V curve exist, featuring a specific V_{MPP}, I_{MPP} point. Such an approach is correct, but the

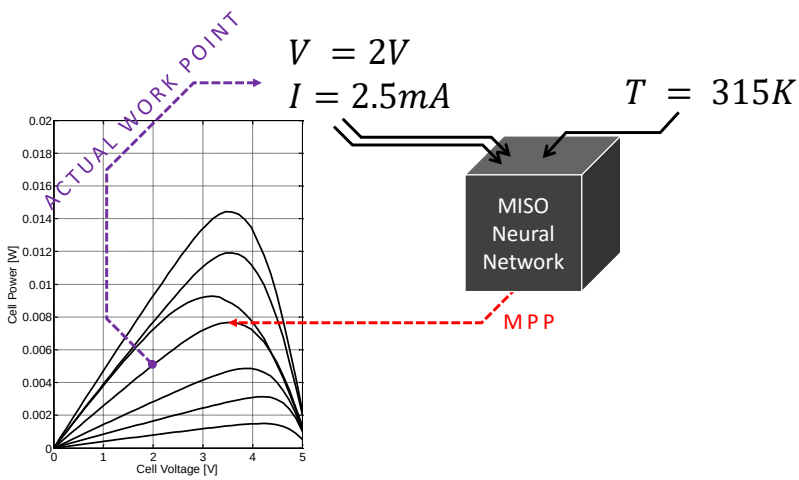


Figure 5.5: Estimating the maximum power point by means of the actual work point (V, I) and temperature (T).

network would require the irradiance measurement G, which is supposedly unavailable. However, instead of using both environmental values, by knowing just the temperature and the actual working point, a specific I/V characteristic (and thus, V_{MPP} point) is still defined. By using this approach, the NN configuration is composed by 3 inputs (the actual voltage, current and temperature) and one output (the predicted optimal voltage V_{OPT}).

By using an identified model such as the One-Diode, a set of I/V curves for different values of G and T were generated. Then, for each curve, the V_{MPP} point was calculated, and associated to each point of the I/V curve. The resulting dataset was used to train a set of FCC and MLP networks, with different numbers of neurons. Among all the trained NN, the smallest networks yielding a training error of $1e-6$ were chosen. For the MLP, a single hidden layer network with 20 Neurons in the hidden layer was chosen.

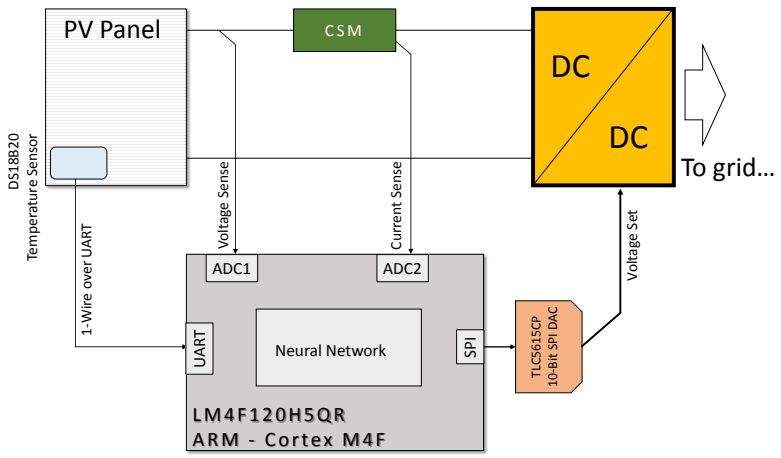


Figure 5.6: MPPT system overview.

For the FCC, a network of 10 Neurons (that is 10 layers) achieved the same results.

5.2.2 Functional Overview

The proposed system can be seen in Fig. 5.6 . The PV device is connected to a DC/DC link via a CSM (Current Shunt Monitor) that allows a precise and noise-free sensing of the load absorbed current. The voltage can be measured directly at the output of the PV device (obviously a level adapter stage must be considered to scale down the PV voltage to the ADC dynamic range). For the temperature, an external thermometer DS18B20, mounted near the PV device can be used to monitor in real time the panel temperature. This particular thermometer communicates via 1-Wire interface, which is not implemented as a standard interface on the LM4F120H5QR microcontroller. However, a fast enough UART port (113kbps) can be configured to emulate

the 1-Wire interface. The three digital values are processed inside the microcontroller unit: the natural values of current, voltage and temperature are calculated in floating point precision, they are scaled on a range between -1 and +1, and they are sent as input to the ANN. The output of the NN is de-normalized, converted into an SPI control string and sent to the DAC. The TLC5615CP is a 10-Bit SPI controlled DAC, and is used to generate the voltage reference used by the DC/DC to modulate the input resistance so that the PV module works near the MPP.

5.2.3 Microcontroller Code Implementation

The LM4F120H5QR Microcontroller, mounted on the Stellaris Launchpad ®LM4F120XL board, can be programmed using CCS (Code Composer Studio) IDE via ICDI (In-Circuit-Debug-Interface). The CCS development environment allows an easy implementation of the Matlab algorithm in C language, automatically translated and optimized by a dedicated compiler in assembly language. The code is structured as a FSM (Finite State Machine) featuring the following routines.

- Initialization: This routine configures the microcontroller central system and enables the TIMER0, SPI, ADC, UART and FPU (Floating Point Unit) peripherals.

- SenseInputs: This routine performs three tasks: first, it reads the analog values of current and voltage from the ADC. Then, it interrogates the DS18B20 thermometer for a temperature reading. Last, it converts the sensors data into floating-point real values.

- ProcessNN: This routine process the data through the neural network: data is first normalized between +1/-1, then it is elaborated by the NN, and the output is de-normalized.

- SPIControl: This routine performs a preliminary check on the data generated by the ProcessNN routine to ensure the voltage is inside the dynamic range of the DAC. Then, it translate the voltage value in a control string and sends it through SPI interface to the DAC.

5.2.4 Neural Network Optimization

For the present work, both the FCC and the MLP architecture were implemented on the microcontroller unit to understand which solution could give the best results in terms of computational speed, occupied memory and prediction accuracy. To retain a computational accuracy, the data going through the ProcessNN is coded using floating-point precision. Then, all the network weights and non linear functions must be calculated using this format. The LM4F120H5QR microcontroller is built around a fast ARM Cortex M4F that is capable of a great performance on integer arithmetics. However, to achieve similar performances on floating-point arithmetics, a mathematical co-processor must be used. The FPU (Floating-Point Unit) included in the microcontroller performs 32-bit instructions for single precision (float) data-processing operations. It supports add, subtract, multiply, divide, square roots and multiply-accumulate in combined instructions (Fused MAC). The computational cost of a Neural Network can be split in two main contributes. The first contribute comes from the linear part of the network: the sum of the

weighted inputs for each neuron. The second lies in the activation function, which is usually non-linear. The Fused MAC allows a very quick computation for the linear part of both the FCC and the MLP. However, the non linear part, even considering the FPU acceleration, can still build up to a considerable quote of the computational time. Different solutions, with different advantages and drawbacks, were considered for the activation function implementation:

- Full Precision: the activation function is computed explicitly, yielding the slowest and most accurate result
- Polynomial Interpolation: By taking advantage of the hardware multipliers present in the FPU, the activation function can be approximated by a polynomial. This solution is faster than the Full Precision, but less accurate.
- Lookup Table: A sampled version of the activation function is loaded in the microcontroller memory, and is addressed as needed. This solution is the fastest, and the precision can be scaled according to the memory availability.

To determine the precision reduction introduced by the Polynomial and LUT implementations, both the options were investigated prior implementation on the MCU. The two NN previously identified were implemented in C code and compiled for x86 architecture, and thoroughly tested on all their dynamic range. In table 5.2 the results from the precision comparison between the Full Precision solution and the Polynomial / LUT options are shown for both the MLP and FCC.

Table 5.2: Precision degradation using different activation function.

Architecture	Act.Function	MSE [V]
MLP (20 Neurons)		
	F.Precision	0 (ref)
	Polynomial	1.66e-4
	LUT (256)	1.05e-2
FCC (10 Neurons)		
	F.Precision	0 (ref)
	Polynomial	3.29e-4
	LUT (256)	8.41e-3

5.2.5 Prototype implementation

To test the approach practical feasibility a prototype board, featuring a low power solar cell was implemented. A schematic representation of the board is shown if Figure 5.7. The circuit is powered by a 9V battery, and a second 5V line is derived using a TPS7250Q LDO Voltage Regulator. This secondary line powers the TLC5615CP DAC and a voltage reference branch composed by two forward diodes and a current limiting resistance. The three operational amplifiers LM258 are powered directly by the 9V line. The circuit is built around a low power Mono-Si solar cell KXOB22-01X83 rated 4V / 3.8mA. The two operational amplifiers shown on the right of the cell in Figure 5.7 decouples the cell itself from the ADC input of the MCU. The top unit gain is negative, since the LM4F120H5QR ADC voltage reference is 3.3V, and the maximum rated output from the cell is 4.7V. The bottom

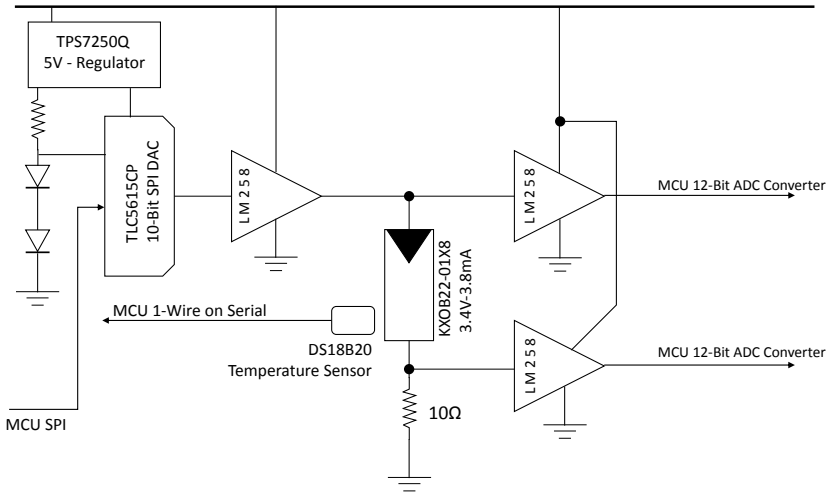


Figure 5.7: Test prototype circuit.

unit gain on the other hand is positive: the maximum rated current from the cell is 3.8mA, so the maximum voltage across the 10 Ω sensing resistor is 38mV, which needs to be amplified to be correctly monitored by the MCU's ADC. On the left part of the circuit, the work point tracking system is implemented. The temperature of the cell is monitored by the DS18B20 1-Wire Temperature Sensor, which is connected to the MCU GPIO. The TLC5615CP DAC is directly connected to SPI interface, and uses as reference voltage the one generated across the two forward diodes in the reference branch. The voltage output of the DAC is buffered and used to set the work point on the KXOB22-01X83.

Table 5.3: Code profiling and memory footprint for different implementations of FCC and MLP Neural Networks.

NN	Act.Function	Memory	Cycles
MLP (20 Neurons)			
	F.Precision	4730	56770
	Polynomial	3054	14179
	LUT (256)	5010	4404
FCC (10 Neurons)			
	F.Precision	4646	23052
	Polynomial	3166	7890
	LUT (256)	4822	2078

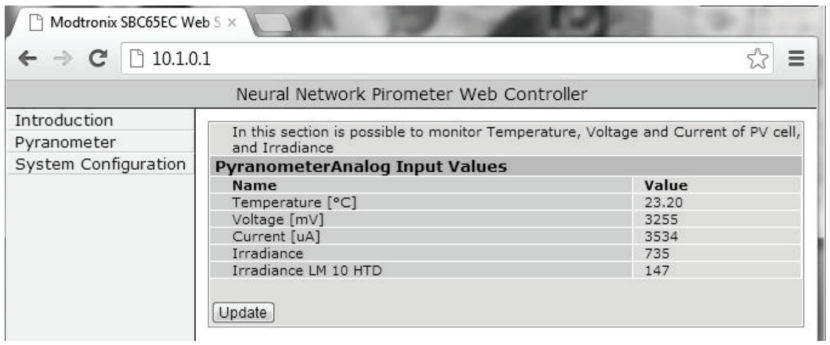


Figure 5.8: A screenshot of web-server internet page, showing temperature, voltage, current measured and irradiance predicted by neural system. .

5.2.6 Code profiling and memory footprint

All the service routines for initialization and peripheral control were implemented using the TI Peripheral Drivers Library. Since a pre-compiled version of these routines exists in the LM4F120H5QR Read-Only Memory, the footprint for system control is negligible and below 2kb of memory. The NN routine, on the other hand, is consistent on both the required memory and the computational time. In table 5.3 the required memory and the clock cycles per sample of both the FCC and the MLP architectures is shown. Both architectures were implemented with full precision activation function, a II degree five-pieces polynomial, and a 256 values Lookup Table.

5.3 MPPT on a low-cost device

The following section is an extract of [Laudani et al., 2014c] where the maximum power point tracking algorithm, based on the neural approach, is embedded in a low-cost 8-bit microcontroller. The obtained device can correctly

track the maximum power point even under abrupt changes in solar irradiance and improves the dynamic performance of the power converter that connects photovoltaic power plants into the ac grid. This neural system is at the same time quick and low cost since it is implemented in a cheaper than 5$ microcontroller, but also evaluates the MPP by just one computational step with a very high accuracy. The low cost micro-controller based system also allows us to have direct interface with the neural system and provides for a straightforward calibration and interfacing via RS232, RS485 or Ethernet with the rest of control system of the PV power plant.

The approach followed is the same as shown in the previous section using a MLP network, but this time, it is implemented on a different, low-end device. In particular, the NN algorithms, designed by using MATLAB programming environment, have been adapted to run in an embedded system (SBC 65 EC by Modtronix), built around the microcontroller PIC18F6627 (Microchip Technology Inc.). This microcontroller belongs to the low–cost low–power PIC18/8–bit family, featuring a 4 kB RAM memory and a 96 kB reprogrammable flash memory; 12 10-bits AD converters; and SPI, I2C, and RS232/RS485 interfaces. Furthermore, the microcontroller can be interfaced to a personal computer through a RS232/RS485 or an Ethernet cable. Choosing an inexpensive microcontroller is essential for a low–cost implementation, then an 8–bit microcontroller has been utilized. In addition, the SBC 65 EC can be used also as web server, allowing to manage the operations of the system by means of a remote internet connection. Specifically, the developed software for the microcontroller consists of a set of initializing functions and a main process. The latter controls the peripheral units,

manages the communication protocols, acquires the input data (current and voltage of PV array terminal by using analog/digital converter embedded in microcontroller and temperature from an one–wire sensor Maxim Integrated model DS18B20), executes the implemented NN routines, and outputs the optimal voltage reference.

All the routines of the program have been written in C language, developed by means of MPLABX IDE and compiled by means of Microchip C 18 compiler. It returns a HEX code file that is downloaded into the microcontroller by using a simple Ethernet connection. The routines have been written taking care not to consume too memory and power of the microcontroller, and also the neural network configuration has been studied in order to be extremely efficient and accurate with a low number of layers and neurons. The whole set of NN routines occupies less than 15kB in terms of source code and less than 1.5k bytes for data. The results obtained by using the microcontroller were compared to those returned by the MATLAB implementation and none remarkable differences have been noted. Lastly, it is important to emphasize that the aforementioned features of the embedded system allows a very easy way to reprogram the microcontroller and also, if needed, to change the weight and the biases of NN in order to recalibrate the device or to include new customized features. In addition, the web server implemented in SBC 65 EC allows remote controlling operations of the PV plant. Indeed, it is possible to check the status of voltage, current, temperature and predicted irradiance of the PV system and to store all these data for further elaborations. An example of screenshot of the web server page is shown in figure 6.

Chapter 6

Solar Irradiance Assessment

Solar irradiance is the power per unit area produced by the Sun in the form of electromagnetic radiation. Irradiance may be measured in space or at the Earth's surface after atmospheric absorption and scattering. Total solar irradiance (TSI), is a measure of the solar radiative power per unit area normal to the rays, incident on the Earth's upper atmosphere, that accounts for an average of $1366W/m^2$. Roughly half of this power lies in the infrared part of the spectrum. The Sun's rays are attenuated as they pass through the atmosphere, leaving maximum normal surface irradiance at approximately $1000W/m^2$ at sea level on a clear day. When $1367W/m^2$ are arriving above the atmosphere (as when the earth is one astronomical unit from the sun), direct sun is about $1050W/m^2$, and global radiation on a horizontal surface at ground level is about $1120W/m^2$. This quantity can be a theoretical assessment for a clear day, unfortunately, climatic conditions (e.g. the movement of clouds) makes the solar irradiance a difficult quantity to estimate in practical scenarios. Indeed, solar irradiance is a critical quantity to assess in PV ap-

plications. Knowing solar irradiance value allows an optimized management of photovoltaic (PV) power plants in terms of produced energy. Indeed, the operating point at which a PV array delivers its maximum power changes as a function of the solar irradiance, temperature and shading conditions. As shown in the previous chapter, if the solar irradiance and cell temperature are sensed, one might theoretically compute the Maximum Power Point and directly act on PV modules by forcing them to operate at that point. Unfortunately, although sensing temperature is easy, the measurement of solar irradiance is expensive: for this reason, the irradiance sensors are seldom utilized in photo-voltaic power plants. Instead, indirect methods are implemented to maximize efficiency. PV modules are the main building blocks of PV power plants, which usually span over a large geographical area with non uniform irradiance. Indeed, one of the most important issues related to a good working of a PV plant is the partial shading, a non-uniform irradiance that significantly decreases the power delivered by solar photo-voltaic arrays [Velasco-Quesada et al., 2009]. In addition, a sensor in situ mounted on the PV panel can also account for its inclination and can generate real time irradiance data for MPPT control algorithms. Obviously, as the solar sensors on the market are not cheap, using these devices for each PV module would be too much expensive for the energy producers.

In the following, an extract of the work [Oliveri et al.,] will be presented, where two circuit architectures for the estimation of the solar irradiance based on simple measurements are proposed. They are thought to be part of a centralized system implemented on FPGA (Field Programmable Gate Array) for sensing and monitoring of solar irradiance in a whole PV plant. The

FPGA centralized architecture could allow for a real time irradiance mapping by exploiting information coming from several low-cost measuring circuits suitably allocated on the PV modules. Validations on real irradiance data collected by the U.S. Department of Energy's National Renewable Energy Laboratory are presented. Following the presentation of this method for the indirect estimation of solar irradiance, a method for prediction of this quantity by means of a dynamic ANN will be briefly shown.

6.1 FPGA based Solar Irradiance Virtual Sensors

In this section two circuit architectures implementing virtual sensors for the estimation of the solar irradiance based on simple measurements are proposed. Thanks to their small size and high speed, these architectures can constitute the main building blocks of a centralized system implemented on FPGA (Field Programmable Gate Array) for sensing and monitoring solar irradiance in a whole PV plant. Indeed, owing to its high performance, the FPGA centralized architecture could allow for a real time irradiance mapping by exploiting information coming from several low-cost measuring circuits suitably allocated on the PV modules (see Fig. 6.1 for a schematic representation)

The first architecture implements a neural-network-based virtual sensor. The neural network approach is widely adopted for virtual sensors for PV applications. In this case the chosen architecture implements a recently proposed low–cost and extremely accurate virtual solar sensor based on a neural

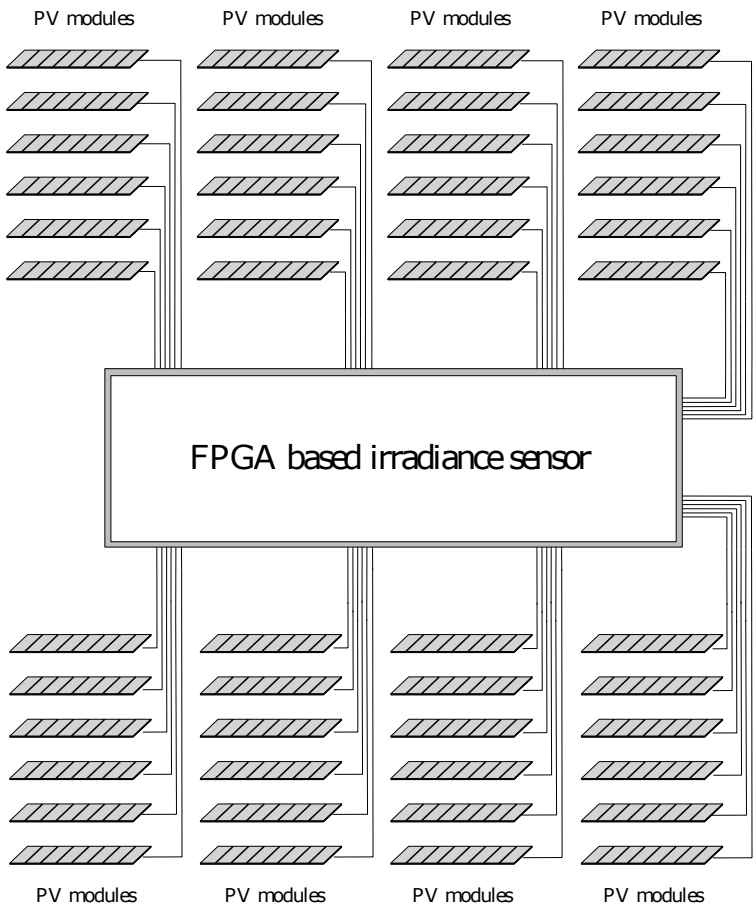


Figure 6.1: Sketch of the centralized measurement of solar irradiance for a whole PV plant by using suitable measuring circuits installed on PV modules.

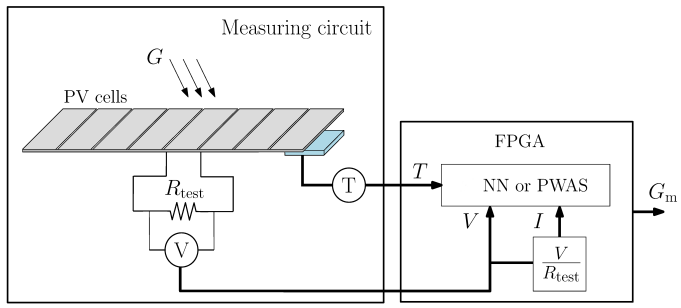


Figure 6.2: Schematic of the proposed solar irradiance measurement.

network (NN) algorithm [Mancilla-David et al., 2014], appositely tailored for the estimation of the irradiance from the knowledge of voltage, current and temperature of a PV cell. The second architecture implements a novel virtual irradiance sensor, based on piecewise-affine functions defined over uniform simplicial partitions (PWAS functions). This solution generally exhibits a lower accuracy with respect to the NN-based approach, but allows for a reduction of both circuit complexity and latency.

6.1.1 Measurement Circuit

The indirect measurement of the irradiance is based on the operating point of the device (i.e. current and voltage) and its temperature. In order to sense V , I and T , a suitable measuring circuit must be used. The solution proposed in [Mancilla-David et al., 2014] (sketched in Fig.) consists of a low-cost PV element (composed by a series of few solar cells), a temperature sensor and a known testing resistor Rtest , suitably chosen according to the PV cell I – V characteristic. The aim of this circuit is to measure the current

I , the voltage V across the cell terminals and the cell temperature, T. These three quantities are fed through analog to digital converters (ADCs) into the virtual sensor implemented in FPGA, which computes the estimated solar irradiance.

6.1.2 Virtual Sensors

Two different irradiance virtual sensors are considered . The first one (which relies on a NN) has been proposed and successfully applied in [16], but a FPGA implementation was not available. The second one is a novel approach based on PWAS functions. PWAS functions have been recently used for the design of virtual sensors, being of interest for their very efficient circuit implementation. Both virtual sensors do not require the knowledge of a model, but are fully dependent on measured data (i.e. they are both black-box models). A training set must be therefore created to properly set the weights defining the shape of the function that relates the virtual sensor inputs to its output. The input pattern $x = [V, I, T]^T$ of both proposed virtual sensors consists of measurements of voltage, current and temperature, while the output $y = G_m$ is the estimated irradiance. The operating points $\{V; I; T\}$, needed for the training process, can be obtained by means of an analytical approach or by using a set of suitable measurements. In the former case, the curves are obtained on the basis of the One-Diode model identified through datasheet values method. In this case the resistance R_{test} is considered to be ideal. To obtain a higher accuracy, a better calibration of the solar sensor is required. In this case, the operating points $\{V; I; T\}$ should be experimentally generated in laboratory under controlled environmental conditions. This would

guarantee a higher accuracy since the calibration procedure can take into account also the effects of parasitic components, inaccurate datasheet information, the dependence on temperature T of the whole measuring circuit, variability in the components (e.g., R_{test}), etc. Here, the first strategy (successfully applied in [Mancilla-David et al., 2014] with very accurate results) is used, with the resistance R_{test} equal to 921Ω. The five parameters used for modeling the adopted solar cell (IXYS KXOB22-01X8, rated at $3.4V, 3.8mA$ at MPP under standard testing conditions) and obtained by using datasheet information, are the following: $I_{irr;ref} = 4.50 \times 10^{-3}A$, $I_{0,ref} = 8.14 \times 10^{-15}A$, $R_{S,ref} = 212.34\Omega$, $R_{sh,ref} = 9k\Omega$ and $n_{ref} = 0.849$. In any case, in NN and PWAS approaches, the three quantities V , I , T are considered as independent variables, even if I and V are proportional each other for each measuring circuit. This makes the solution more general and easily adaptable if experimental training data are used (where V and I are measured independently). By using the I-V relation of the One-Diode model and considering $I = V/R_{test}$, a suitable training set is created with operating points $\{V; I; T\}$ obtained by N different irradiance-dependent I–V curves for a constant temperature (equally spaced in the range $[260 - 310]K$), and by M different temperature-dependent I–V curves at a constant irradiance (equally spaced in the range $[200 - 1500]W/m^2$). In this way, $n_{train} = N \times M$ training patterns are created. Since each operating point $\{V; I; T\}$ corresponds to a certain value of the irradiance G that lights up the solar cell, after the training procedure, the virtual sensor is able to generalize the existing relationship between the current operating point and the corresponding irradiance G.

NN-Based Sensor

The choice of using a NN for computing solar irradiance is due to its versatility. Unlike other approaches requiring data distributed over regular grids, NNs can perform multidimensional fitting by using any data distribution. Moreover, the NN is a paradigm that can be realized by using simulation software, analog circuits, optical microsystems and, as in our case, FPGA. A feed–forward NN (FFNN) with a single hidden layer should be chosen in order to decrease as much as possible the computational cost. For choosing the architecture and for the training of the NN-based virtual sensor, the procedure described in [Fulginei et al., 2013] was followed, by using Levenberg–Marquardt as training algorithm and by assuming satisfactory a target MSE on the training set of $6 \times 10 - 3$. By following the theoretical approach described in [Fulginei et al., 2013], starting from initial guessed values for n_h hidden layer neurons and n_{train} training set size, the iterative process yielded $n_h = 8$ and $n_{train} = 50$.

PWAS-Based Sensor

A PWAS function is a piecewise-affine function defined over a domain partitioned into simplices. Consider a hyper-rectangular domain $\mathcal{D} = \{x \in \mathbb{R}^n : a_i \leq x_i \leq b_i\}$, being x_i, $i = 1, \ldots, n$ the components of point x. Given a set of $n+1$ vertices $v_0, v_1, \ldots, v_n \in \mathbb{R}^n$, a simplex in $\mathbb{R}^n$ is a convex combination of the vertices, i.e. it is the set of points:

$$\left\{ x \in \mathbb{R}^n : x = \sum_{i=0}^{n} \mu_i v_i \right\} \tag{6.1}$$

where $0 \leq \mu_i \leq 1, i = 0, \ldots, n$ and $\sum_{i=0}^{n} \mu_i = 1$. For $n = 1$ a simplex is a segment, for $n = 2$ a triangle and for $n = 3$ a tetrahedron. In a general case, it is a n-dimensional hyper-triangle.

The main advantage of using PWAS functions is that they can be implemented very efficiently in digital circuits (e.g., FPGAs), or application specific integrated circuits, ASICs. Low-complexity virtual sensors based on PWAS functions have been recently proposed in [**?**] along with a circuit architecture suitable for FPGA implementation. Here this approach is applied for the estimation of solar irradiance.

Any continuous PWAS function can be defined as a linear combination of PWAS α-basis functions as follows

$$f_{PWAS}(x) = \sum_{i=1}^{N_v} w_i \alpha_i(x) \tag{6.2}$$

being

$$\alpha_i(v_j) = \begin{cases} 1 & \text{if } j = i \\ 0 & \text{if } j \neq i, \end{cases}$$

where v_j, $j = 1, \ldots, N_v$,[1] denote the vertices of the simplicial partition, and the coefficients w_i, $i = 1, \ldots, N_v$, determine uniquely f_{PWAS}. The PWAS-based virtual sensor expresses the quantity to be estimated $y \in \mathbb{R}$ as a PWAS function of the input variables x, i.e., $y = f_{PWAS}(x)$. In order to train the virtual sensor (i.e., to find a suitable function f_{PWAS}) a set of n_{train} noisy measurements $x^{(k)}$ and $y^{(k)}$ ($k = 1, \ldots, n_{train}$) of x and y, respectively, is necessary. Starting from these measurements, an optimization problem

[1] Being $\alpha(x)$ a PWAS function, its value outside the partition vertices can be easily obtained by linear interpolation.

can be defined, whose solution provides the weights vector w. A possible choice, based on the 2-norm, is the following:

$$\min_{w} \left\{ \sum_{k=1}^{n_{train}} \left[y^{(k)} - f_{PWAS}(x^{(k)}) \right]^2 + \sigma w^T \Gamma w \right\} \tag{6.3}$$

being σ and Γ the Tikhonov regularization parameter and matrix, respectively, and f_{PWAS} defined as in Eq. (6.2). The training process of the PWAS-based virtual sensor consists in finding the weights vector w by minimizing cost function (6.3), which results being a quadratic function of w. Prior to this operation, it is necessary to choose the simplicial partition of the PWAS function domain. The parameters to be set for training the PWAS-based virtual sensor are the N_v weights w_i. For this application a uniform simplicial partition made up of $N_v = 2592$ vertices was employed, with 80 subdivisions along V, 1 subdivision along I, and 15 subdivisions along T. A first-order Tikhonov regularization has been employed with $\sigma = 10^{-6}$. The dimension of the training set is $n_{train} = 1500$ ($N = 50$ and $M = 30$).

6.1.3 Circuit implementation

Two digital circuit architectures are described, suitable for the FPGA implementation of both NN- and PWAS-based virtual sensors. Since the proposed circuits are thought to be part of a centralized system for sensing the irradiance in a whole PV plant, the fastest the circuit computation, the higher the number of panels that can be monitored per second. In order to maximize speed, therefore, a fixed point data representation is used and computation parallelism is exploited.

Implementation of FFNN based virtual sensor

Several circuit architectures have been proposed in the literature for the FPGA implementation of neural networks. These architectures mainly differ in data representation (fixed/floating point) and computation of the activation function. The reader is referred to for a comprehensive survey. One of the main issues related to the digital circuit implementation of neural networks is the computation of the nonlinear activation function. Several methods are adopted in the literature: polynomial approximations, CORDIC (COordinate Rotation DIgital Computer algorithms), rational approximations, table-driven methods. For our application a look-up-table (LUT) was combined with first-order polynomials. The values $f_{i,j} = f_i(v_j)$ have been computed in 2^9 points v_j, uniformly distributed over the range [0 6] (only positive values are considered since f_i is an odd function). The activation function in a generic point v is then computed by linear interpolation as follows:

$$f_i(v) = \begin{cases} \frac{f_{i,j+1}-f_{i,j}}{v_{j+1}-v_j}(v - v_j) + f_{i,j}, & \text{if } 0 \leq v \leq 6 \\ w_i^o, & \text{if } v > 6 \\ -w_i^o, & \text{if } v < -6 \\ -\left[\frac{f_{i,j+1}-f_{i,j}}{v_{j+1}-v_j}(v - v_j) + f_{i,j}\right], & \text{if } -6 \leq v < 0 \end{cases} \tag{6.4}$$

where $v_j \leq |v| < v_{j+1}$. Samples $f_{i,j}$ are stored in a LUT.

All hidden neurons perform their computations at the same time. The block scheme of the proposed circuit architecture is shown in Fig. 6.3.

A finite state machine **CONTROL_FSM** receives the input pattern x

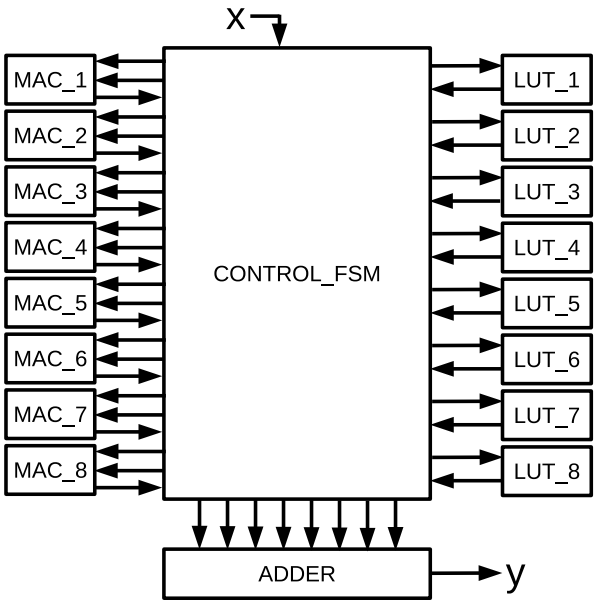


Figure 6.3: Block scheme of the circuit architecture implementing the NN-based virtual sensor.

and is used to send the correct inputs and manage the outputs of 8 MAC blocks, 8 LUTs and one adder. The MAC blocks evaluate in parallel the expressions $h_i(x)$, $i = 0, \ldots, n_h - 1$. Based on the results of this computations, two addresses (j and $j+1$) are generated for each neuron, to access a corresponding LUT containing values $f_{i,j}$ and $f_{i,j+1}$. Function $f_i(h_i)$ can be therefore computed by resorting to (6.4), evaluated again by the same MAC blocks. Notice that a multiplication by the weights w_i^o is not necessary, since the samples $f_{i,j}$ stored in the LUTs already comprise the coefficients. A final sum is performed by block **ADDER** in order to compute output y, representing the estimated irradiance G_m.

Implementation of PWAS based virtual sensor

Two digital circuit architectures (a serial and a parallel one) are proposed in [Storace and Poggi, 2011]. The architectures are described in VHDL language, can be used for any PWAS function with uniform simplicial partition and are suitable for FPGA implementation. In [Oliveri and Storace, 2012] the architectures have been improved and modified in order to make them suitable for control applications, where a constant latency and a fixed sampling time are needed. Moreover an arbitrary non-uniform simplicial partition can be handled.

Equation (6.2) could be used, in principle, to evaluate the PWAS function at any point $x \in \mathbb{R}^n$. Nevertheless, a more efficient strategy, from a circuit implementation standpoint, consists in (i) finding the $n+1$ vertices of the simplex containing x and (ii) linearly interpolating the values of the function at these vertices, as follows:

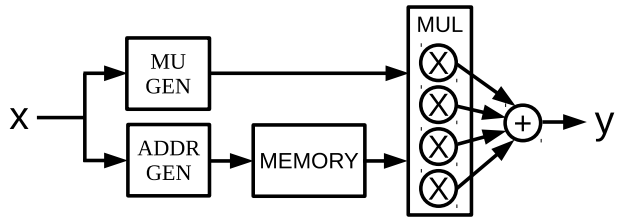


Figure 6.4: Block scheme of the circuit architecture implementing the PWAS-based virtual sensor.

$$f_{PWAS}(x) = \sum_{i=1}^{n+1} \mu_i f_{PWAS}(v_i) \tag{6.5}$$

Interpolation coefficients μ_i (the same as in (6.1)) depend on the position of x within the simplex [Storace and Poggi, 2011]. Due to the regularity of the simplicial partition, the point location problem solution is trivial if Kuhn lemmas are employed [Storace and Poggi, 2011].

A block scheme of the proposed architecture is shown in Fig. 6.4.

The circuit architecture consists of a memory storing the values of the PWAS function at the partition vertices. Block **MU GEN** is responsible of computing, based on input pattern x, the interpolation coefficients μ_i in Eq. (6.5). Block **ADDR GEN** generates instead the addresses of the memory cells storing the values of the function at the vertices of the simplex containing x. Values $f_{PWAS}(v_i)$ are therefore provided by the memory and expression (6.5) can be finally evaluated by means of 4 multipliers and one adder.

A detailed description of the architecture is available in [Oliveri and Storace, 2012].

6.1.4 Results

The proposed irradiance virtual sensors have been validated on a test set constituted by real irradiance data collected by the U.S. Department of Energy's National Renewable Energy Laboratory at the Hawaii's Honolulu International Airport in the island of Oahu. The solar irradiance data composing the validation set is shown in Fig. 6.5. Data were collected over 698 seconds taking measurements every second on a partially cloudy day. In particular, the solar irradiance profile ranges between about 1100 W/m^2 and 250 W/m^2, reaching this lower values in the last 150 seconds. Since no data was available for the temperature profile of the PV modules, the arbitrary profile shown in Fig. 6.6 has been used.

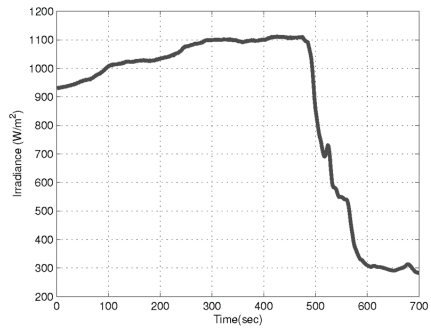


Figure 6.5: Actual irradiance at Hawaii's Honolulu International Airport used for the validation set.

It is worth noticing that this choice does not affect the performance. For each pair T-G, indeed, there exists a unique I-V characteristic. Therefore, once the model has been correctly identified, it can be used to generate, for any T, different pairs I-V corresponding to the same value of irradiance

G. Moreover, the use of an arbitrary temperature profile also constitutes an effective validation procedure.

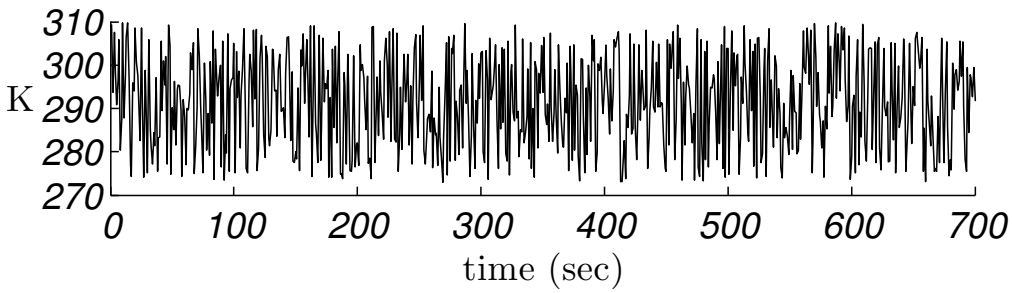


Figure 6.6: Arbitrary temperature profile used for validating the virtual sensor.

The circuit architectures for the implementation of the designed NN- and PWAS-based virtual sensors have been implemented on a low-cost (about 10$) FPGA (Xilinx Spartan3 XC3S200). This device is equipped with 12 18×18 bit multipliers, therefore a resolution of 18 bits has been chosen. Table **??** summarizes the latency, number of operations per second (by considering a clock frequency of 50 MHz) and resources occupation of the proposed architectures.

A post place and route simulation of the VHDL code has been performed, which takes into account the effects of data quantization.

Fig. 6.7 shows the percentage relative error between the irradiance data contained in the test set and the estimations performed with the NN- and PWAS-based virtual sensors both in MATLAB (double precision) and by VHDL simulation (18 bits fixed precision). Given y the actual irradiance (contained in the test set) and $\hat{y}$ the estimated one, the percentage relative error has been computed as

Table 6.1: Latency and resources occupation of the circuit architectures implementing the NN- and the PWAS-based virtual sensors.

	NN-based sensor	PWAS-based sensor
latency (clock cycles)	10	3
% occupied slices	80	15
% block RAMs	66%	66%
mult.	8	4
computations per second	5 million	16 million

Table 6.2: Mean Squared Error.

	NN-based sensor	PWAS-based sensor
MATLAB	$7.74 \cdot 10^{-3}$	$9.16 \cdot 10^{-1}$
VHDL	$6.38 \cdot 10^{-2}$	$9.60 \cdot 10^{-1}$

$$e = 100\frac{|y - \hat{y}|}{|y|} \tag{6.6}$$

It can be noticed, from Tab. 6.1, that the PWAS-based architecture is less resource-demanding and more than three times faster than the NN-based solution, at the cost of a lower accuracy in the estimation.

Notice also (Fig. 6.7) that the accuracy of the PWAS-based sensor is very high when estimating low values of irradiance (from about $t = 500$ seconds). This happens because, when low values of solar radiation (lower than 300 W/m^2) strike the PV cell, the dependence of G with respect to V, I and T for the adopted cell is almost linear, as results from extrapolated data: therefore a PWAS function is more suitable to approximate it, with respect

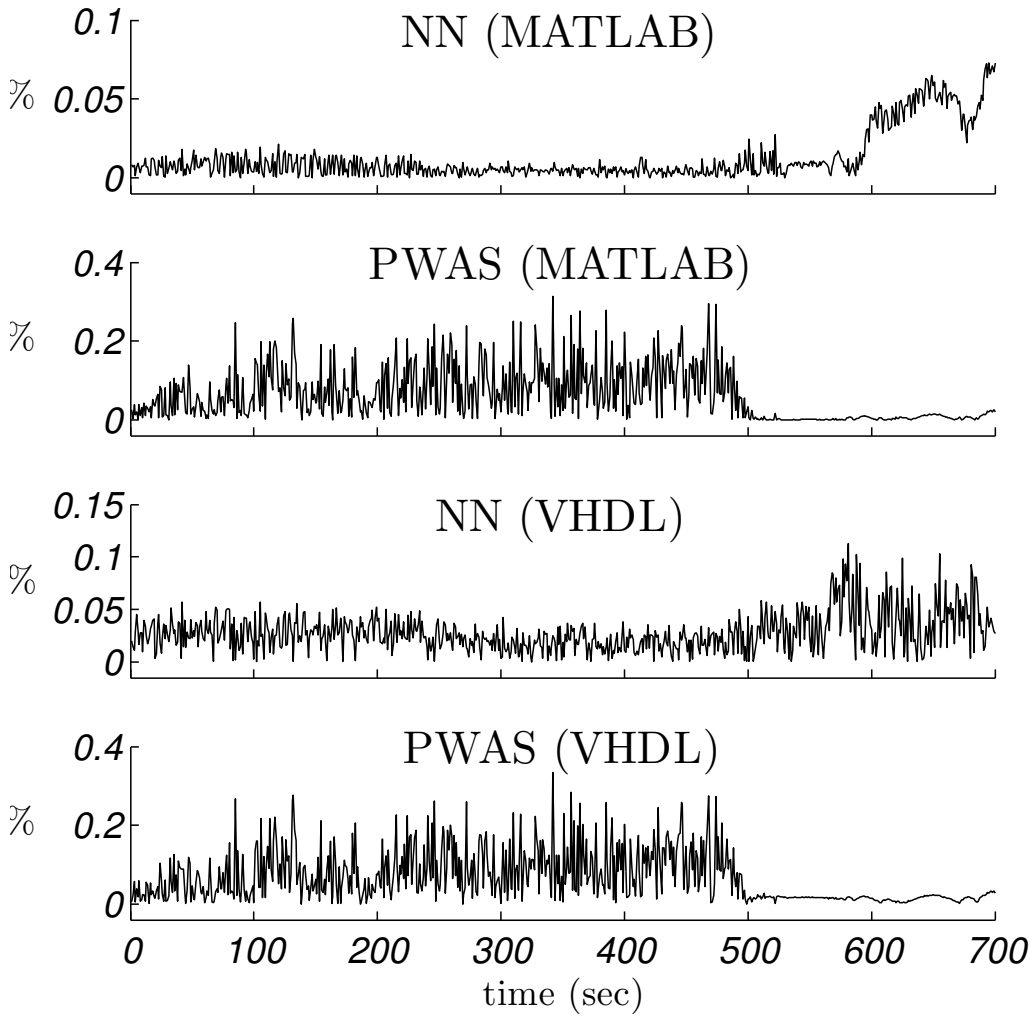


Figure 6.7: Percentage relative error with both PWAS- and NN-based sensor, in MATLAB and by VHDL post place and route simulation simulation.

to a neural network with non-linear characteristics.

Table 6.2 shows also that the architecture implementing the NN-based sensor is more sensitive to the quantization error, with respect to the PWAS-based approach, for which the MSE is almost unchanged if double or fixed precision is adopted. In order to perform a further comparison, the two proposed virtual sensors have been implemented in the microcontroller LM4F120H5QR by Texas Instruments, which has a cost and a clock frequency similar to the XC3S200 FPGA. This microcontroller is built around an ARM Cortex M4F that provides high performance for integer arithmetics. To achieve similar performances on floating-point arithmetics, a mathematical co-processor must be used. The performances in terms of required memory and computation times are shown in Table 6.3, for a clock frequency of 50 MHz. The number of computations per second is quite high also in this case, mainly for PWAS, even if it is about a thousand times lower than for the FPGA implementation. It is worth noticing that the accuracy is the same as achieved with MATLAB and an increment in the performance of the NN based virtual sensor can be obtained if the nonlinear activation function is computed through a polynomial approximation or a lookup table (in the current implementation it is evaluated exactly). In these cases the gain in computation time is paid with a larger memory employment and entails lower accuracy. With respect to the FPGA implementation, the main limitation of microcontroller is not the computation time but instead its sequential management of the operations, which makes almost impossible the interaction with more than one physical sensor at a time. Indeed the bottleneck is not the response of the microcontroller or FPGA in the computation of results but the management

Table 6.3: Latency and resources occupation of the implementation of the virtual sensors on TI LM4F120H5QR microcontroller.

	NN-based sensor	PWAS-based sensor
latency (clock cycles)	57965	1653
occupied Bytes	453	1324
Reserved Bytes SRAM	318 (0.99%)	1288 (4.02%)
computations per second	850	30250

and acquisition times of the sensors. For instance, the external thermometer DS18B20, mounted close to the PV device, can be used to monitor in real time the panel temperature. This device communicates with the main unit by means of a one-wire interface extremely simple to implement, but it requires about half a second to measure a temperature. Thus, it is not efficient to manage a large amount of physical sensors by using microcontroller-based virtual sensors.

This problem does not affect FPGA, since several circuit architectures can be replicated in the same board (according to the circuit complexity and the board size) or pipelined in order to increase parallelism. Of course many other issues would arise in the implementation of the whole system, such as, for example, the management of the communication between the measuring circuits installed on the PV cells and the central FPGA. Moreover the actual performances of the monitoring system will be influenced by the delays in data acquisition and transmission, such as those previously cited for the temperature sensors DS18B20 or for the analog to digital conversion and trasmission of PV output voltage on a dedicated bus. With the very

simple FPGA mentioned in this work, it is not possible to replicate the architecture several times, since the RAM occupation of a single circuit is 66%; nevertheless, a pipeline can be created which allows computing the value of the irradiance in one clock cycle (assuming that the pipeline is always full, i.e., data are acquired at a frequency of 50MHz). A multiplexer can be implemented in the FPGA so that many sensors can be connected to the board and their outputs are read in sequence. The number of sensors which can be managed by the same FPGA depends on many factors, such as the size of the FPGA, the number of input pins, the analog to digital converters (number of bits, serial/parallel output). Therefore it strongly depends on the specific application.

6.2 Prediction of Solar Irradiance Trough Fully-Recurrent ANN

For some applications having the information of the solar irradiance with some anticipation (that might be in the order of hours or days) is critical. Forecasting solar irradiance is strictly related to two different necessities: negotiation on the energy market and managing storage elements. The former is related to the determination of the price for electricity on Day-Ahead and Infra-Day markets, according to offer and demand price laws. The latter is used in control systems to decide whether to store excess energy (i.e. not used by the installation) in a storage device or re-inject it into the distribution network to sell it back to the service provider. The choice can be done either to achieve the maximum reliability of the system (this is especially true for

scenarios where power distribution is irregular and unreliable) or minimum energy cost. Regardless of the desired goal, it is clear that the knowledge of the availability for the solar resource is critical for this kind of assessment. Prediction of solar irradiance is a generic time-series forecasting problem.

$$G[t+M] = f(G[t], G[t-1], G[t-2]...G[t-N]) \tag{6.7}$$

Where $G[t]$ is the irradiance sampled at time t, f is a multivariate function, M is the forecast horizon and N is the forecast memory. Indeed, the problem can be reduced to find the f relationship between the inputs and the outputs data which allows the forecast of the quantity at the desired time $[t+M]$ based on the observed data at the times $[t-1, t-2, t-3]$. Numerical instruments like recurrent neural networks (RNN), wavelet-networks, and wavelet-networks-fuzzy are very suitable for this kind of applications. This model is also referred as *auto-regressive*, since the quantities used for the estimation are previous values of the *same* quantity. This is the approach followed in this application, but it is not the sole alternative. It is possible to consider *exhogenous* inputs as well: i.e. quantities that are correlated to the one predicted and may help the ANN in the prediction process. In [Mellit and Pavan, 2010] the authors presents a very good applicative paper, explaining in detail how to predict irradiance by mean daily solar irradiance, mean daily air temperature and the day of the month.

In the method herein proposed, the forecasting is obtained using a Recurrent Neural Network, trained through the use of the Real-Time Recurrent Learning (RTRL) algorithm described in Sec. 1.2.6. This algorithm can train a generic RNN, and was designed for optimal execution on serial machines.

The final implementation of the RTRL algorithm was done in embedded environment, to explore the feasibility of a compact and economic solution for irradiance forecasting. The algorithm, originally implemented in Matlab environment, was translated in C and compiled on both x64 architecture and ARM. The embedded platform used was an ARM Cortex M4-F microcontroller by Texas Instruments, described in Sec. 3.2.1.

6.2.1 Implementation of RNN Architecture

For this forecast problem, a fully RNN was used: the basic rule for this network is that every neuron has a weighted directed connection to every other neuron *and* itself. Some nodes features an independent input (i.e. not an output from another neuron), and are defined as **input** neurons. Other neurons have a supervised value at given times, and are defined as **output** neurons. All other neurons are **hidden** neurons. The network architecture is the following:

- The network has n units, with $(m + 1)$ external input lines.
- Of the external input lines, the first m are independent inputs of the RNN, and the last is a constant value used for neuron biasing.
- At any given time, we define the vector $\mathbf{X}(t)$ as the m-tuple of inputs, the vector $\mathbf{Y}(t)$ as the n-tuple of units outputs. The concatenation of $\mathbf{X}(t)$ and $\mathbf{Y}(t)$ is $\mathbf{Z}(t)$, a vector ($n + m + 1$ long).

$$\mathbf{X}(t) = \begin{bmatrix} x_1 \\ x_2 \\ ... \\ x_m \\ 1 \end{bmatrix}_{1,(m+1)} \qquad \mathbf{Y}(t) = \begin{bmatrix} y_1 \\ y_2 \\ ... \\ y_m \end{bmatrix}_{1,n} \qquad \mathbf{Z}(t) = \begin{bmatrix} \mathbf{X}(t) \\ \mathbf{Y}(t) \end{bmatrix}_{1,n+m+1} \tag{6.8}$$

Given the full connectivity of the architecture, the$z_k(t)$ vector represent the input of every neuron in the network. The weights between the neurons are described by the W matrix. The matrix has n rows and $n + (m + 1)$ columns. The generic element $W_{i,j}$ represent the weigth of the connection between the j-th element of $z_k(t)$ and the i-th neuron. The sum of the weighted inputs of each i-th neuron is then computed by the product:

$$net_i(t) = \sum_{j=1}^{(n+m+1)} W_{i,j} z_j(t) \tag{6.9}$$

or, in matrix form:

$$[\mathbf{net}] = [\mathbf{W}][\mathbf{Z}] \tag{6.10}$$

The output of the i-th neuron at the next time step is given by:

$$y_i(t + 1) = f_i(net_i(t)) \tag{6.11}$$

Where f_i is the neuron activation function, which can be either linear or non-linear. Note that the input vector y is not updated until the next time step. For the purpose of this work, we will assume a non-linear (tangent

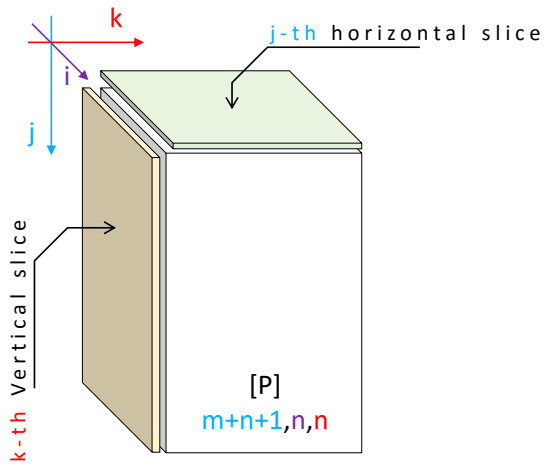


Figure 6.8: P Matrix for RTRL algorithm.

sigmoid) activation function for the hidden and input neurons, and a purely linear activation function for the output neurons.

6.2.2 RTRL Implementation

In this section, the embedded implementation of the RTRL algorithm will be explained. To better understand the following pseudo-code, a table summarizing the different arrays used in the code will follow.

In order to enhance the dynamic capabilities of the RNN even further, two tapped-delay lines were implemented. The first one creates delayed replicas of the input, and sends it to the RNN as independent inputs. The second one feedbacks delayed replicas of the output, and sends it as input to the RNN as well. Both delay lines can be disabled independently. The bookkeeping

Table 6.4: RTRL Pseudo-code arrays.

Name	Size	Description
E	1-by-n	Error Vector
dFdY	n-by-n	Activation function derivative matrix, rows are all equal, column elements represent, for each neuron, the derivative of the activation function calculated for the neuron's weighted inputs
W	n-by-(n+m+1)	Weight matrix
W_N	n-by-n	Left part of the W matrix, relative only to the neuron connections weights.
Z	1-by-(n+m+1)	State vector
EYE_Z	n-by-n	Matrix obtained by the product of an unitary matrix and the scalar value of Z[j]
P	(n+m+1)-by-n-by-n	Tri-dimensional matrix containing the states of the dynamic system describing the weights changes. As shown in Fig. 6.8, the matrix in the algorithm is updated every iteration through horizontal slicing, and is used for weight update through vertical slicing.
DELTA_W	n-by-(m+n+1)	Weights update matrix

for the actual input vector requires a periodic shift, discarding the oldest sample. The pseudo-code for the algorithm is the following:

- For every new input:
  - Calculate RNN output.
  - Create the dFdY matrix.
  - Extract the W_N matrix from the W matrix.
  - For every j-th horizontal slice of P:
    - Create the EYE_Z matrix (n-by-n matrix).
    - Multiply W_N by the j-th slice of P.
    - Sum the previous product to EYE_Z, element-wise.
    - Multiply the previous sum by dFdY, element-wise.
    - Overwrite the j-th slice of P with the previous product result.
  - For every k-th non-null value of E:
    - Extract the k-th vertical slice of P
    - Multiply the slice by the scalar value of E[k] and alpha.
    - Accumulate the result in the weigth connection matrix DELTA_W
  - Sum DELTA_W to W, element-wise.
  - Bookkeep input vector.

The P matrix, that stores the sensitivities for all the weights of the RNN for different time-steps, is represented in Fig.6.8.

6.2.3 Simulation Performance

The pseudo-code proposed was implemented in Matlab and C programming languages, and was used to create, train and validate a RNN for the 1-h forecast of solar irradiance. Since no real-time elaboration of the ANN was required (data is sent hour-wise), code profiling was limited to the occupied memory. Indeed, allocation of the P matrix, and all the service arrays shown in Tab 6.4 requires considerable memory, and for this reason, code was implemented on a high level MCU. The LM4F120H5QR Microcontroller, mounted on the Stellaris Launchpad LM4F120XL board was used. Since no time-related optimization are required, all calculations have been performed using full-precision Floating Point arithmetics. The full code occupy $10kB$ of FLASH memory and $24kB$ of RAM with a network composed by 6 hidden neurons and a delay-tapped input of 12 samples. Fig. 6.9 shows the error magnitude and forecasting results for the RNN when used to predict solar irradiance, hourly, on a monthly period.

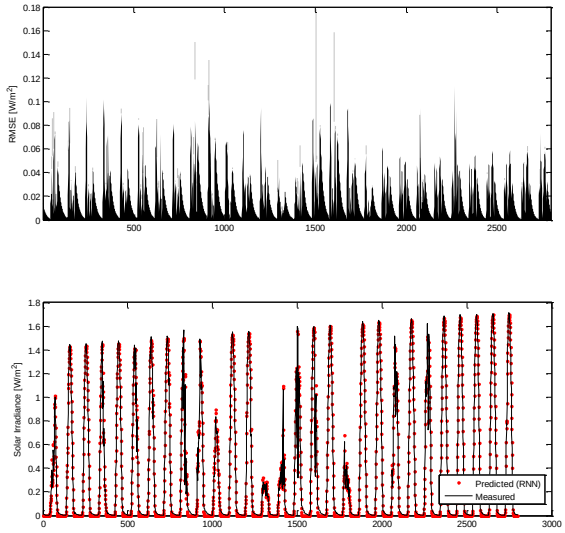


Figure 6.9: Relative error and simulation performance for an RTRL trained RNN predicting solari irradiance with 24h forecast

Chapter 7

Inverse Biomechanics

In biomechanics, internal forces exerted during the execution of motor tasks can be estimated by combining a biomechanical model, able to predict the forces acting on each involved joint, with the design of an optimization criterion to determine the contribution of each muscle to the overall force. This approach has been applied in a variety of application fields, ranging from the analysis of gait and running, to the study of upper limb movements. An particular segment of biomechanics is interested in the study of muscular forces exerted during athletic activity. In particular, in this section, the work towards the implementation of an embedded system for the estimation of muscular forces of a cyclist during cycling activity will be presented. In the biomechanics of cycling it is important to evaluate how the athlete executes the required motor task, in order to have objective parameters that quantify the performance. This aspect can be analyzed in terms of power exerted while cycling, using different techniques, or investigating the role of muscle activity while performing the task. To this aim, it is possible to propose an

inverse dynamics approach to predict muscle force patterns by the measurement of the external forces exerted on the pedal. These predictions were compared against muscle activity, as estimated from surface Electromyography (sEMG) data. The use of standard optimization algorithms, to solve the equations that describe the inverse dynamics of cycling, can represent a limit to the development of a real time device. Being able to estimate muscle forces in real time can be used on the field to assess and monitor athlete performance. The purpose of this study was to develop a new optimization algorithm based on artificial Neural Networks (NN), in order to reduce the computational complexity of the deterministic one, while maintaining the quality of estimation. The study was carried out on two levels. First, a biomechanical model based on ANN was implemented in a high level environment. Second, the model was implemented in C language and simulated by a MCU for real time estimation of muscle forces. The following section are extracts from the works [Cecchini et al., 2014] (the high level implementation) and [Lozito et al., 2015] (the MCU implementation). More information and details can be found in the original papers.

7.1 Neural Networks for Muscle Forces Prediction in Cycling

This section documents a system based on Artificial Neural Networks to predict muscle force patterns of an athlete during cycling. Two independent inverse problems must be solved for the force estimation: evaluation of the kinematic model and evaluation of the forces distribution along the limb. By

solving repeatedly the two inverse problems for different subjects and conditions, a training pattern for an Artificial Neural Network was created. Then, the trained network was validated against an independent validation set, and compared to evaluate agreement between the two alternative approaches using Bland-Altman method. The obtained neural network for the different test patterns yields a normalized error well below 1% and the Bland-Altman plot shows a considerable correlation between the two methods. The new approach proposed herein allows a direct and fast computation for the inverse dynamics of a cyclist, opening the possibility of integrating such algorithm in a real time environment such as an embedded application.

7.1.1 The biomechanical model

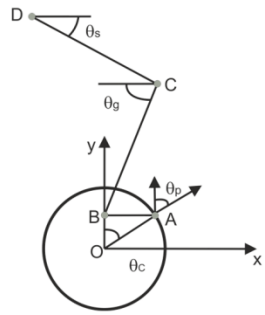
A biomechanical model was identified to reproduce the cycling task and used for the muscle forces estimation of the lower limb. It has been modeled as a three-joint (i.e.ankle, knee, and hip) system, actuated by nine muscles and built in three steps:

1. Definition of a kinematic model to evaluate the position of every segment of the leg involved in the gesture;

2. Definition of the inverse dynamics to evaluate the muscular torque for every joint;

3. Calculation of the muscular forces through the data obtained with the two previous steps.

Regarding the third step, a cost function, based on a physiological criterion, was minimized to predict muscular force patterns. This optimization



Segment	Body part	Symbol
AB	foot	*f*
BC	shin	*s*
CD	thigh	*t*

Figure 7.1: Kinematic chain of lower limb and angle between body segment.

was obtained by using a feed forward NN. An additional one was used to solve the equation associated with the first step, as described in the following.

Restricting the analysis to the sagittal plane, the kinematic model of the lower limb is composed of constrained rigid elements and mechanical elements of the bicycle, used to transmit the motion to the wheel. By modeling each body segment and each mechanical element as a segment (Figure 7.1) it is possible to define a kinematic chain with five elements and two degrees of freedom,so the position of each member in the sagittal plane is determined by the length of each segment and two of the following angles:

1. Θ_C angle between the bicycle's frame and the crank;

2. Θ_P angle between the crank and the pedal;

3. Θ_G knee angle identified as in Fig. 7.1;

4. Θ_S hip-saddle angle identified as in Fig. 7.1.

The length of each segment of the model was determined by direct measurement. Once kinematic data and pedal forces are obtained, ankle, knee

and hip joint moments are calculated using the inverse dynamics. Afterwards, these data are used to implement the three equilibrium equations at each joint, involving the following muscles, that represent the minimum set to be involved in the model: (1) Tibialis anterior (TA); (2) Soleus (SO); (3) Gastrocnemius (GA); (4) Vastii (VA); (5) Rectus femoris (RF); (6) Short head of biceps femoris (BFs); (7) Long head of Biceps Femoris (BFl); (8) Iliacus (IL); (9) Gluteus Maximum (GLM). The relation between the muscular moments and the muscular forces at each joint j is given by the equation:

$$\sum_{i=1}^{N_j} F_j \times d_{i,j} = M_j \tag{7.1}$$

where M_j represents the muscular moment at the j-th joint, N_j is the number of muscles acting on the j-th joint, F_i is the muscular force exerted by the i-th muscle and $d_{i,j}$ is the effective moment arm of the i-th muscle from the j-th joint.As the number of equations is not sufficient to calculate muscular force values, these were calculated by minimizing the cost function:

$$U = \sum_{i=1}^{p} \left(\frac{F_i}{PCSA_i} \right)^3 \tag{7.2}$$

given the constraints:

$$0 < F_i < F_{i,max} \tag{7.3}$$

here, given p total number of muscles, and being $PCSA_i$ and $F_{i,max}$ respectively the physiological cross sectional area and the maximum force value for the i-th muscle. The cubic exponent used in the Eq. 7.2, guarantees the best trade off between the muscular contractile force and the maximum du-

ration of the contraction. This cost function relies on the co-activation of all the muscles involved in the gesture.

7.1.2 The Neural Implementation

In this model, NNs are used in two critical steps:

1. Calculation of the relative rotational angle between the frame of the bicycle and the thigh, Θ_S;

2. Estimation of muscle forces.

In the first step, considering the coordinates of the points (A, B, C, D) respect to the reference system centered in O, as reported in Fig. 7.1, the neural network is necessary since the angle Θ_S is defined in an implicit transcendental equation:

$$\frac{X^2 + Y^2 + t^2 - s^2}{2t} = Xcos(\Theta_S) + Ysin(\Theta_S) \tag{7.4}$$

given that:

$$\begin{cases} X = X_B + X_D \\ Y = Y_B - Y_D \\ X_B = X_A - fsin(\Theta_C - \Theta_P) \\ Y_B = Y_A + fsin(\Theta_C - \Theta_P) \end{cases} \tag{7.5}$$

Both the segments t and s are constants, whereas the terms X and Y are the time-varying unknowns. The equation can be solved for Θ_S using numerical methods (i.e. optimization algorithms), but even if this is an

assessed solution that gives accurate results, it is iterative and consequently slow. To solve this problem, for the calculation of the angle Θ_S, a MISO NN with two inputs, four hidden neurons and a single output, has been used. The inputs were the angle between the frame of the bicycle and the crank Θ_C and the angle between the frame of the bicycle and the pedal Θ_P. For this first step, training data set was composed of 7050 samples from three different subjects, and 118,000 samples from a different new subject composed the testing dataset. It is important to highlight that since the problem is analytical it is not necessary to use measured input data to train the NN, as long as it belongs to a sensible function domain. In this work being the measured data set oversampled with respect to its frequency content and covering the whole angular domain, it was more practical to use it instead of synthetic data. The comparison between the old method and the new one was done through direct error estimation. This is justified considering in this first phase the NN approach as an approximation of the quasi-analytical solution.

The second step, relative to the estimation of muscle forces, can be approached by numerically solving the implicit non-linear system described by the Eq. 7.1, while minimizing Eq. 7.2 considering the boundaries in Eq. 7.3. This is done by solving an inverse problem of bounded function minimization, which can be approached by numerical optimization techniques, such as the ones shown in Sec. 2. Each time the problem must be solved, it has three known parameters (the muscular moments) and nine unknowns (the muscle forces). The problem was generalized using nine different MISO NNs, one for each unknown. The inputs were the three parameters (muscular moments of

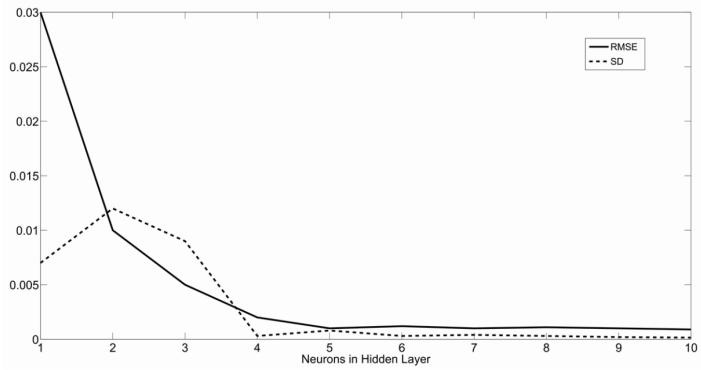


Figure 7.2: Root Mean Square Error (solid) and Standard Deviation (dashed) Error plot function of the number of neurons (from 1 to 10).

ankle, knee and hip), while the output was a specific unknown (a muscular force). In this case, the training set was composed by 2360 samples from a single subject. Test data was composed by 118,000 samples from a different new subject.

The choice of the network size was made on statistical considerations: by testing the NN repeatedly with increasing number of neurons in the hidden layer, the error progressively diminished. However, over a specific number of neurons, the differential increase in performance was negligible, so the last point with relevant increase in performance was taken as the optimal size. In Fig. 7.2, an example of mean and variance for network performance (RMSE) can be seen. A low standard deviation for the chosen point confirms that the performance of the network was not due to chance. The NNs were trained by the LM algorithm. For this second step, results obtained using the NN approach were compared to those obtained in the traditional approach considering that both can be considered muscular forces estimators, neither

of them being able to provide an exact solution. Indeed neither the former nor the latter can be considered the gold standard. The comparison can thus be performed through a Bland-Altman plot, considering the average between the two estimates as the hypothesized true value. Details on such comparison can be found in [Cecchini et al., 2014].

7.1.3 Experimental Validation

A previously acquired set of data was used to validate the approach proposed in this section. These data were obtained by pedaling on a cycling simulator for sessions about 50 minutes long with a pedaling cadence fixed at 70 rounds per minute (rpm). The cycling simulator was equipped with a system to control the power exerted by the participant. Force data were acquired (2000Samples/s sampling frequency, 12-bit A/D converter) by a homemade instrumented pedal mounted on the cycling simulator. With this system, it was possible to measure force components exerted on the pedal, the angular displacement of the pedal, Θ_P, and the angular displacement of the crank, Θ_C. These data were used as input for the biomechanical model described above. The experimental protocol was used for the estimation of the muscular forces using the two different techniques and its validation was:

- Training and validation set of NNs using data obtained previously by a deterministic optimization algorithm.
- Plot analysis between the signals obtained by NNs and signals obtained by the optimization algorithm, and the evaluation of the RMSE and RMSE Standard Deviation.

Table 7.1: Parameters of Neural Networks (NN) used to obtain the results.

Output	Topology	Inputs	Hidden Neurons	Training Set	Validation Set
Θ_S Angle	1 MISO	2	4	7050	118000
Muscle Forces	9 MISO	3	15	2360	118000

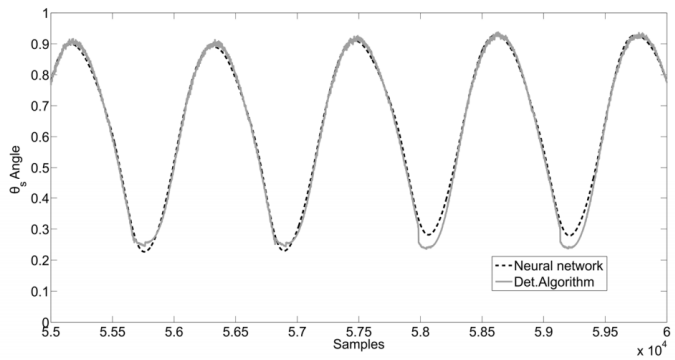


Figure 7.3: Angle Θ_S signals, obtained by the deterministic optimization algorithm (solid gray) and by the neural network (dashed black).

- Validation of the experimental protocol analyzing Bland-Altman plots extrapolating 1180 random samples from each muscle forces signals.

To obtain the results, two NN were implemented on the basis of the parameters reported in Tab 7.1.

The Θ_S angle estimated signals, for one of the subject, obtained using deterministic algorithm optimization and using neural network are both shown in Fig. 7.3.

Table 7.2: Results of RMS Error percentage and SD Error for the calculation of Θ_S angle.

Subject	RMS Error %	Std. Deviation Error
1	0.085	2.68×10^{-4}
2	0.134	7.35×10^{-4}
3	0.077	4.53×10^{-4}

In the Tab. 7.2 the correlation of the two signals, for three different subjects, is shown in terms of Root Mean Square Error.

NN, for all the three subjects, makes a RMS error well below 1%, and a SD error below 10^{-3}, showing a good convergence in the research of the optimal solution. Regarding the second step, 1180 random samples out of 118000 have been taken for the analysis of the results. In the Fig. 7.4, the Bland-Altman plot for the Rectus Femoris is shown. While the $1.96\sigma_{diff}$ boundaries are not negligibly narrow, they can be reasonably attributed to the intrinsic noise affecting the deterministic algorithm (more on this matter will be discussed below). The distribution is acceptably symmetric around the mean axis, excluding the possibility of a systematic measurement error. There is no apparent pattern in the error distribution apart from a border effect, consisting in a clustering of samples on the left side around the axis origin.

This is due to boundary conditions used in the model: muscular forces cannot have negative value, because of obvious physiological reasons. Aside from the error, a consideration can be made on the networks performance predicting muscular forces. Indeed, the number of unknowns in (2) is higher

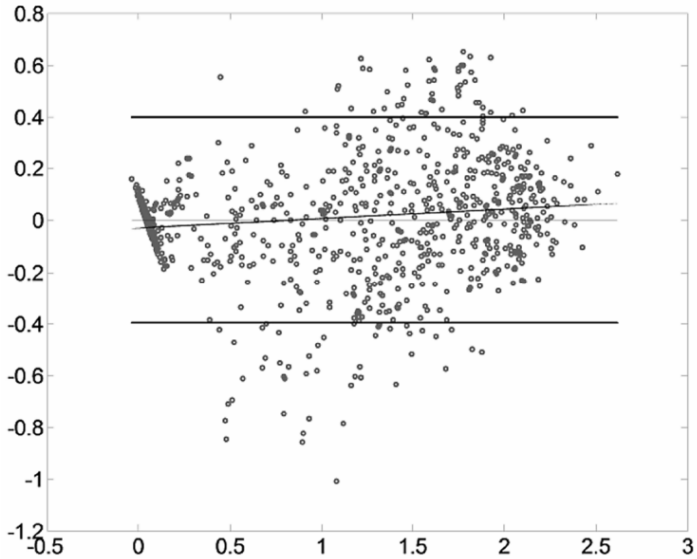


Figure 7.4: Bland-Altman Results for the rectus femoris muscle.

than the number of equations (9 unknowns for 3 equations), so the numerical optimization of such problem is multi-modal, yielding a very noisy solution. The NN however performs an average of the different solutions, smoothing effectively the results. This effect is highlighted in Figure 7.5: the NN completely filters unnatural "high" frequency components in the muscular activity signals. In terms of computational complexity, the computational time required to elaborate a full sample, thus to compute the nine muscular forces starting from the pedals data, is about 26 ms on a Core i7 Machine, against the 2,200 ms required to compute the deterministic original approach on the same machine, yielding a considerable speed up.

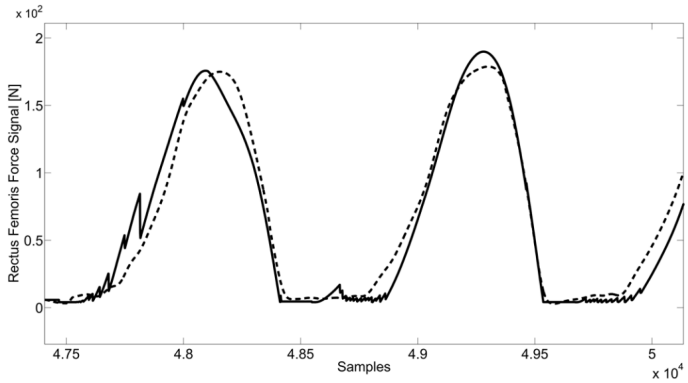


Figure 7.5: Muscle forces of rectus femoris obtained with the neural network (dashed) and with the deterministic algorithm optimization (solid). Sampling frequency 1000 Samples/s.

7.2 Embedded System for Real-time Estimation of Muscle Forces

In this section, the possibility of implementing a neural solution in an embedded environment for the muscular force estimation is investigated. When implementing this algorithm in an embedded environment, the limited computational capabilities calls for a trade-off among precision, memory footprint, and computational cost. As shown in Chapter 4, Different studies tested the embedded implementation of NNs to achieve optimal results, either by re-arranging the operations required to compute the linear part of the NN to fully exploit pipelining, or by speeding up the costly non-linear activation function through different numerical approximation.

Another issue worth being addressed in this implementation is the possibility of solving in real time the unknowns. In the original proposed algorithm

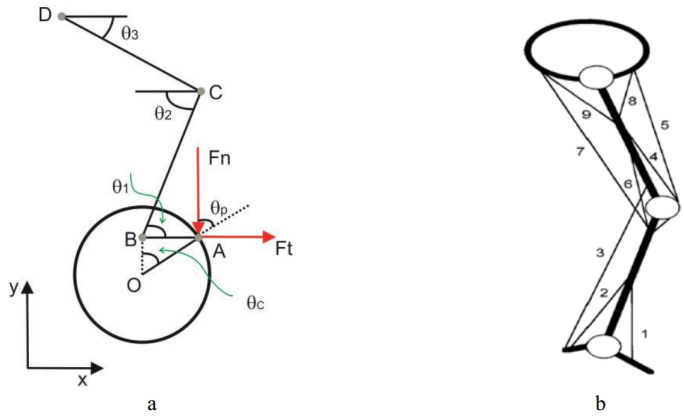


Figure 7.6: Kinematic chain (left) and correspondent muscular model (right) of the lower limb while cycling.

data were processed in batch, allowing heavy filtering for noisy signals. In a real-time approach, only a small time-window for the signal is available, thus excluding the possibility of intensive filtering. The inverse model requires, to be computed, several II order numerical differentiations, that naturally introduce an amplification for high frequency noise. Different techniques are used in the literature to obtain a noise-rejecting differentiator that can be applied easily in embedded environment.

In the first part of this section, the real-time implementation of the model will be explained from a systemic point of view, with special attention to the neural estimator and the differentiation techniques. Then, the embedded implementation will be presented along with the performance evaluators considered.

7.2.1 Real-Time Inverse Model Overview

In order to simplify its implementation and subsequent test-debug procedure, the proposed model can be solved considering two different sections: the first, addressed as "kinematic section", is related to the determination of the complete kinematics, considering as inputs the actual angles measured as explained above; the second, addressed as "dynamic section", aims to the determination of the joint reactions and the joint moments, using current and past data obtained from the kinematic section, and of the muscular forces. As displayed in Figure 7.6, the input data provided to the whole model are the angles Θ_C and Θ_P and the pedal force components Fn and Ft. The kinematic section of the model receives as input only the angles and, using trigonometric equations, computes the x, y positions for the leg joints. For the computation of the other angles, instead of solving an inverse trigonometry problem, a neural network was used. The input of the neural network is the cosine of both Θ_C and Θ_P angles while the output is Θ_3. The other angles Θ_1 and Θ_2 are calculated as a function of Θ_3. The network is composed by a single neuron for reasons that will be explained in the next section.

In the dynamic section, the joint reactions and the moments must be computed to determine the muscular forces of the leg. The mechanical model, summarized above, is a II order one, which requires a numerical solution for the acceleration resolution of several elements composing it. To compute the second derivative of a quantity with respect to time, at least the current value, and the previous two samples of the actual quantity, must be known. Since this is a real-time model, a buffer system that holds the previous values

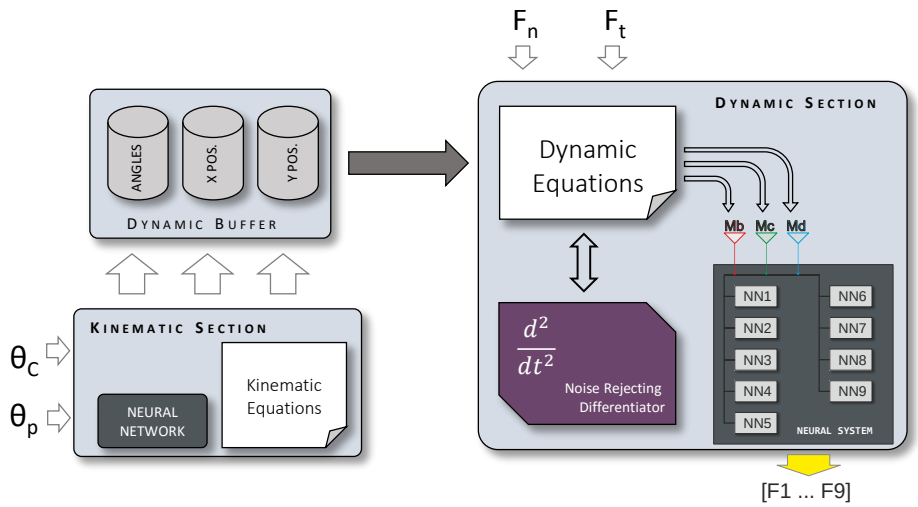


Figure 7.7: Inverse biomechanical model overview.

of the quantity must be interposed between the kinematic and the dynamic parts of the model. The buffer system is composed by a set of three bi-dimensional arrays, two for the x, y positions, and one for the angles. Every time the model is computed, the array acts like a shift register, discarding the oldest sample and replacing it with the new one.

Given the experimental nature of the data a noise-rejecting differentiator, which works on more than 3 points, was implemented. More information on the differentiator will follow. Once the joint reactions are computed, the muscular moments can be easily calculated. The final computation of muscular forces is demanded to the Neural System, which receives the three muscular moments Mb, Mc and Md as inputs.

7.2.2 Neural Estimator

The neural estimator implemented in the model was refined starting from the one implemented in the previous section, and is composed by two parts: a Multiple-Input-Single-Output (MISO) NN, to calculate the Θ_3 angle, and a Neural System of nine MISO NN, to assess the muscular forces (one for each force). The main purpose of the original NNs in was to obtain the maximum performance through heavily filtered signals, and for this reason, the criterion to size the NNs was maximizing the accuracy with the smallest number of neurons (to preserve generalization capabilities). Conceiving the problem in an embedded environment, two considerations must be done: first, since the computational capabilities of a MCU is limited, some accuracy should be traded for performance, to ensure the real-time capabilities of the system; then, as it will be shown, a serious problem affecting the embedded implementation of the model lies in the strong noise introduced by the process. From this perspective, a simpler and less accurate NN, with less neurons, can actually enhance the model performance by introducing a natural low-pass filtering of data. For these reasons, the NNs used for the embedded implementation of the model were reduced in complexity. The NN used to compute the Θ_3 angle has a single nonlinear (tangent sigmoid) neuron in the hidden layer, and uses as input the cosine of Θ_P and Θ_C angles. All the NNs used to compute the nine muscular forces have four nonlinear (tangent sigmoid) neurons in the hidden layer, and use the three muscular moments Mb, Mc and Md as inputs. Networks were trained and validated in Matlab® environment. To enhance the embedded performance of the NN, a speedup for the activation function of the hidden neurons was obtained by computing

it through a 2nd degree polynomial interpolating function, as suggested in Chapter 4.

7.2.3 Noise-Rejecting Differentiator

The dynamic section of the model requires the computation of the second derivative for unfiltered, noisy signals. The magnitude impulse response of an ideal second degree differentiator is $|H(\omega)| = \omega^2$ exalting the high frequency components of the signal, and lowering the signal to noise ratio (SNR) for signals affected by white Gaussian noise. In the high-level implementation of the model, the problem of noise was solved by using heavy low-pass filters on the signals. However, the filter lengths and the real-time nature of the present implementation discourage the use of intermediate filtering during the process. An alternative approach is to use differentiating filters, shaped to have a response proportional to the low-pass filtered second derivative of the excitation. The filters order N is variable, must be odd and larger than 5. From N, the coefficients and the equations for the filtering can be easily derived through Eq. 7.6 and Eq. 7.7.

$$f(x_0) \approx \frac{1}{2^{N-3}h^2}\left(s_0 f_{-M} + \sum_{k=1}^{M} s_k (f_{-M+k} + f_{-M-k})\right) \tag{7.6}$$

Where $M = (N-1)/2$ and the $\{s\}_{k=0}^{M}$ coefficients can be calculated by a recursive algorithm for $k = [M-1, ..., 0]$

$$\begin{cases} s_{M+1} = 0 \\ s_M = 1 \\ s_k = \frac{[(2N-10)s_{k+1}-(N+2k+3)s_{k+2}]}{(N-2k-1)} \end{cases} \tag{7.7}$$

The differentiator was implemented in the system in the form of a library, where the coefficients are precomputed at the beginning of the program execution using a separate function.

Theoretically, the order of the filter could be changed in real time by re-computing the coefficients. However, the order of the filter determines the length of the filter itself, i.e. the number of points needed in the dynamic buffer to compute the second derivative.Since the memory for the buffer is allocated statically (through a series of #define directives) the order of the filter can be modified at compile time, not run-time. Obviously, increasing the filter length yields a smoother signal at the cost of heavily degrading the real-time algorithm performance.

7.2.4 Workbench

The model was initially developed in C and tested in x86 Windows environment using the simple CodeBlocks IDE. In this environment, a set of libraries was created for the model: the NNs and the noise-rejecting differentiator. To implement the project in embedded environment, a powerful Cortex M4-F ARM microcontroller was used, the LM4F120H5QR device, mounted on the Stellaris LaunchPad (Texas Instruments). This microcontroller has a maximum clock frequency of 80MHz, 256KB of Flash / 32Kb SRAM / 2KB EEPROM, dual 12-Bit ADC, a dedicated Floating Point Unit,

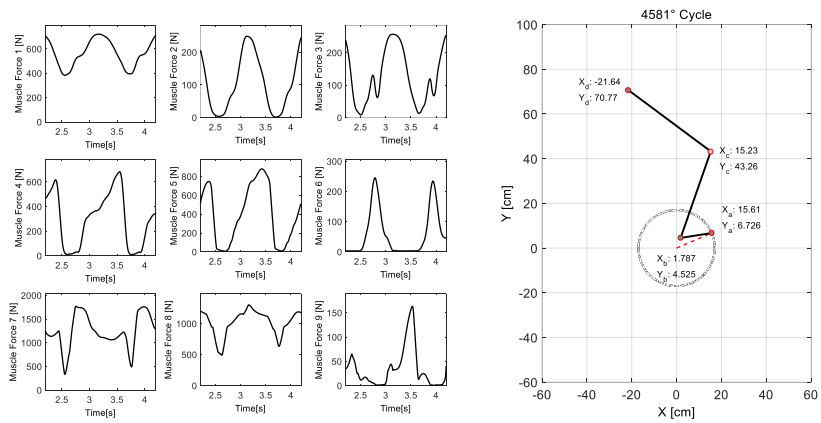


Figure 7.8: Matlab interface and real-time control utility.

and the board implements a RS232 interface, through the programming port (In-Circuit Debug Interface ICDI), that can be used for communication. The free IDE Code Composer Studio was used to program the microcontroller board. To test the real-time performance of the model, a control program was created in Matlab, to send a stream of data to the MCU, to recover the results, to assess the error introduced by the microcontroller, and to show the results. The graphical interface of the Matlab utility is shown in Fig. 7.8. Code profiling,however, had to be performed on the MCU itself: the RS232 interface implemented in Matlab library is a harsh bottleneck, introducing a considerable delay between samples, which should not be accounted for when evaluating code performance. For this reason, execution times were measured by clock-cycles using the debug utility of Code Composer Studio.

7.2.5 Performance

The algorithm was validated on the test bench previously illustrated and the code was profiled both in terms of precision, memory footprint and clock cycles required for computation. The code was built with different compile-time parameters allowing fine-tuning of performance. In this investigation, two parameters were changed:

- Order (i.e. number of samples used) of the II order differentiator;
- Activation function of the NNs.

In the following table, for differentiators of variable length, the computational cost of the algorithm, in terms of clock cycles and maximum sampling, is shown. Indeed, the actual microcontroller works with a clock of 50MHz, but the TM4C1294NCPDTI model, lately released by Texas Instruments, belonging to the same family of microcontrollers, has a 120MHz internal clock, and yields higher performance.

As displayed in Tab. 7.3, the computational cost for the model rises quickly as soon as the order of the differentiator grows. This is due to the increased number of operations required to compute the dynamic section of the model. However, in terms of overall error,the best results are obtained with an order of 5 or 7. Even if a higher order filter better removes the noise of the signal, the system responds slowly and in proximity of quick variations the error is considerable. Indeed, an order higher than 9 should be considered only if more data processing is needed for which noisiness is less desirable than inaccuracy.

Table 7.3: Computational cost for different filtering orders.

Order	C.Cycles	Period (50MHz)	Period (120MHz)	Max.Freq. (50MHz)	Max.Freq. (120MHz)
5	125632	0.002512	0.001046	397.988	955.171
7	172991	0.003459	0.001441	289.032	693.678
9	191415	0.003828	0.001595	261.213	626.910
11	212707	0.004254	0.001772	235.065	564.156

The second parameter that was investigated was the activation function used for the hidden neurons of the NN. Two alternatives were used: the full-precision activation function, computed using math.h C library, and a polynomial interpolation of the activation function, pre-computed in Matlab, composed by a fit of five 2nd order polynomials (see Section 4.1 for more information). The second solution is obviously a trade-off between precision and performance. By the combination of this parameter and the order, two final solutions are proposed, one for best performance, one for highest precision. Percent Root Mean Square Error(%RMSE) was calculated as the average RMSE (on the 9 forces) between the output in embedded environment and the output obtained from the original algorithm, on a set of 20.000 samples. This comparison shows, in quantitative terms, the degradation introduced by transposing the algorithm from a batch Matlab implementation to a real-time, embedded implementation.

As it can be seen in Tab. 7.4 and Tab. 7.5 , error for the Performance configuration is almost 3 times higher than the Precision one, whereas the speedup factor is less than 2. The Performance configuration should only be

Table 7.4: Performance configuration.

2nd Degree Differentiator Order	5
Activation Function	Polynomial
% RMSE	10.3%
Clock Cycles	125632
Cutoff Frequency (50MHz Clock)	397 Hz
Cutoff Frequency (120MHz Clock)	955 Hz
Flash Memory Occupation	4.13kB

Table 7.5: Precision configuration.

2nd Degree Differentiator Order	7
Activation Function	Full-Precision
% RMSE	3.61%
Clock Cycles	212636
Cutoff Frequency (50MHz Clock)	235 Hz
Cutoff Frequency (120MHz Clock)	564 Hz
Flash Memory Occupation	3.59kB

used if a coarse and rapid guess of the forces is needed. Both configurations largely satisfy minimum requirements for frequency, since for the biomechanics of the studied gesture, the force information is contained between 0 and 40 Hz. Such a large gap in terms of computational time can be used either to increase precision further, through intermediate filtering, or to include additional data elaboration, like sEMG correlation algorithms.

Conclusions and Future Developments

The work proposed on this PhD thesis is composed by two contributions.

In the first one, an analysis of the most relevant soft-computing techniques, and in particular, the ones involving embedded environments, was analyzed. Of the very wide spectrum of techniques that are defined by the name of Soft Computing, two where studied in depth: Optimization Algorithms and Neural Networks. Both were studied considering the analytical formulation of the techniques, implementations in a high-level programming environment, and the practical implementation on embedded devices.

In the second one, the knowledge obtained by the state-of-the-art research led to development and implementation of new methodologies to efficiently implement those techniques in embedded environment. Most of the proposed methods involve searching for optimal trade-offs between the scarce computational resources available on devices, and the performance degradation coming from numerical approximations.

Some techniques developed in this work were implemented and/or tested on real devices, but they were never used on a practical application (i.e.

Section 4.2 and Section 4.3). Other techniques, on the other hand, were used for real engineering problem scenarios. In the first case, the goal was simply to obtain an efficient computational procedure to execute an algorithm. In the second case on the other hand the algorithms implementations had to be tuned to meet project specifications. This required an additional study, concerning the engineering problem involved.

The engineering fields that have been studied for the application of SC techniques are two. The first, and most important one, is the study of Photovoltaic systems (i.e. Chapter 5 and Chapter 6): the circuital models that represent the PV devices, the problem of estimating and predicting solar irradiance, and the issue of tracking the optimal work point under variable environmental conditions. The second one is the study of the inverse biomechanics models used for the force estimation in cycling (i.e. Chapter 7): a literature-known model was first modified by replacing some elements with neural estimators, and was later implemented in real-time on a microcontroller unit.

Several aspects remain open for future development. On the purely algorithmic front, the optimized procedure for ANN computation presented in Section 4.3 could benefit greatly from an integration of pruning techniques for optimal network sizing [Hassibi and Stork, 1993, LeCun et al., 1989]. On the matter of Maximum Power Point Tracking for PV devices, the actual challenge is to create a control system able to face the partial shading problem. This can be done either by optimization algorithms like the ones proposed in Chapter 2 [Kazmi et al., 2009] [Lei et al., 2011], or by active reconfiguration of the solar array to exclude shaded panels. Both techniques requires opti-

mization algorithms efficiently implemented for embedded devices. On the matter of irradiance sensing and prediction, recent developments concerning the circuital model of the PV device(s) propose an analytic approach to compute the solar irradiance. For this reason, it becomes critical to implement the circuit model equations directly on the control device. Other than creating very accurate and fast irradiance sensors, this can be used to implement tools like PV panel simulators and I-V curves tracers. On the matter of biomechanics, the solution implemented by now is a gray-box approach, combining the computational flexibility of an ANN with a very complex (and heavy) dynamic system. A comparative analysis between this approach, and the complete simulation of the system through a dynamic ANN could be the critical step to achieve higher accuracy without increasing dramatically the computational costs. Combining this approach with a real-time learning algorithm, like the one proposed in Section 1.2.6, and on-line sEMG measurements, could give a flexible tool that learns the muscle response of the athlete during the activity itself.

Bibliography

[Aarts and Korst, 1988] Aarts, E. and Korst, J. (1988). Simulated annealing and boltzmann machines.

[Alippi and Storti-Gajani, 1991] Alippi, C. and Storti-Gajani, G. (1991). Simple approximation of sigmoidal functions: realistic design of digital neural networks capable of learning. In *Circuits and Systems, 1991., IEEE International Sympoisum on*, pages 1505–1508. IEEE.

[Amin et al., 1997] Amin, H., Curtis, K. M., and Hayes-Gill, B. R. (1997). Piecewise linear approximation applied to nonlinear function of a neural network. In *Circuits, Devices and Systems, IEE Proceedings-*, volume 144, pages 313–317. IET.

[Carrasco et al., 2013] Carrasco, M., Mancilla-David, F., Fulginei, F. R., Laudani, A., and Salvini, A. (2013). A neural networks-based maximum power point tracker with improved dynamics for variable dc-link grid-connected photovoltaic power plants. *International Journal of Applied Electromagnetics and Mechanics*, 43(1).

[Cecchini et al., 2014] Cecchini, G., Lozito, G. M., Schmid, M., Conforto, S., Fulginei, F. R., and Bibbo, D. (2014). Neural networks for muscle forces prediction in cycling. *Algorithms*, 7(4):621–634.

[Cybenko, 1989] Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314.

[Dai and YUAN, 1996] Dai, Y. and YUAN, Y.-x. (1996). Convergence properties of the fletcher-reeves method. *IMA Journal of Numerical Analysis*, 16(2):155–164.

[Eberhart et al., 1995] Eberhart, R. C., Kennedy, J., et al. (1995). A new optimizer using particle swarm theory. In *Proceedings of the sixth international symposium on micro machine and human science*, volume 1, pages 39–43. New York, NY.

[Fletcher and Reeves, 1964] Fletcher, R. and Reeves, C. M. (1964). Function minimization by conjugate gradients. *The computer journal*, 7(2):149–154.

[Flower and Jabri, 1993] Flower, B. and Jabri, M. (1993). Summed weight neuron perturbation: An o (n) improvement over weight perturbation. In *Advances in Neural Information Processing Systems (NIPS92*. Citeseer.

[Fulginei et al., 2013] Fulginei, F. R., Laudani, A., Salvini, A., and Parodi, M. (2013). Automatic and parallel optimized learning for neural networks performing MIMO applications. *Advances in Electrical and Computer Engineering*, 13(1):3–12.

[Fulginei and Salvini, 2010] Fulginei, F. R. and Salvini, A. (2010). The flock of starlings optimization: influence of topological rules on the collective behavior of swarm intelligence. In *Computational Methods for the Innovative Design of Electrical Devices*, pages 129–145. Springer.

[Fulginei et al., 2012] Fulginei, F. R., Salvini, A., and Pulcini, G. (2012). Metric-topological–evolutionary optimization. *Inverse Problems in Science and Engineering*, 20(1):41–58.

[Glover, 1989] Glover, F. (1989). Tabu search-part i. *ORSA Journal on computing*, 1(3):190–206.

[Grippo and Lucidi, 1997] Grippo, L. and Lucidi, S. (1997). A globally convergent version of the polak-ribiere conjugate gradient method. *Mathematical Programming*, 78(3):375–391.

[Hara and Nakayamma, 1994] Hara, K. and Nakayamma, K. (1994). Comparison of activation functions in multilayer neural network for pattern classification. In *Neural Networks, 1994. IEEE World Congress on Computational Intelligence., 1994 IEEE International Conference on*, volume 5, pages 2997–3002. IEEE.

[Hassibi and Stork, 1993] Hassibi, B. and Stork, D. G. (1993). *Second order derivatives for network pruning: Optimal brain surgeon*. Morgan Kaufmann.

[Herrera et al., 1998] Herrera, F., Lozano, M., and Verdegay, J. L. (1998). Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis. *Artificial intelligence review*, 12(4):265–319.

[Hwang, 1988] Hwang, C.-R. (1988). Simulated annealing: theory and applications. *Acta Applicandae Mathematicae*, 12(1):108–111.

[Jabri and Flower, 1992] Jabri, M. and Flower, B. (1992). Weight perturbation: An optimal architecture and learning technique for analog vlsi feedforward and recurrent multilayer networks. *Neural Networks, IEEE Transactions on*, 3(1):154–157.

[Kamruzzaman and Aziz, 2002] Kamruzzaman, J. and Aziz, S. M. (2002). A note on activation function in multilayer feedforward learning. In *Neural Networks, 2002. IJCNN'02. Proceedings of the 2002 International Joint Conference on*, volume 1, pages 519–523. IEEE.

[Kazmi et al., 2009] Kazmi, S., Goto, H., Ichinokura, O., and Guo, H. J. (2009). An improved and very efficient mppt controller for pv systems subjected to rapidly varying atmospheric conditions and partial shading. In *Power Engineering Conference, 2009. AUPEC 2009. Australasian Universities*, pages 1–6. IEEE.

[Krishnanand and Ghose, 2005] Krishnanand, K. and Ghose, D. (2005). Detection of multiple source locations using a glowworm metaphor with applications to collective robotics. In *Swarm Intelligence Symposium, 2005. SIS 2005. Proceedings 2005 IEEE*, pages 84–91. IEEE.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

[Kwan, 1992] Kwan, H. K. (1992). Simple sigmoid-like activation function suitable for digital hardware implementation. *Electronics letters*, 28(15):1379–1380.

[Laudani et al., 2014a] Laudani, A., Fulginei, F. R., and Salvini, A. (2014a). High performing extraction procedure for the one-diode model of a photovoltaic panel from experimental I–V curves by using reduced forms. *Solar Energy*, 103:316 – 326.

[Laudani et al., 2014b] Laudani, A., Fulginei, F. R., Salvini, A., Lozito, G. M., and Coco, S. (2014b). Very fast and accurate procedure for the characterization

of photovoltaic panels from datasheet information. *International Journal of Photoenergy*, 2014. Article ID 946360.

[Laudani et al., 2014c] Laudani, A., Fulginei, F. R., Salvini, A., Lozito, G. M., and Mancilla-David, F. (2014c). Implementation of a neural mppt algorithm on a low-cost 8-bit microcontroller. In *Power Electronics, Electrical Drives, Automation and Motion (SPEEDAM), 2014 International Symposium on*, pages 977–981. IEEE.

[Laudani et al., 2014d] Laudani, A., Lozito, G. M., Fulginei, F. R., and Salvini, A. (2014d). An efficient architecture for floating point based miso neural neworks on fpga. In *Computer Modelling and Simulation (UKSim), 2014 UKSim-AMSS 16th International Conference on*, pages 12–17. IEEE.

[Laudani et al., 2015] Laudani, A., Lozito, G. M., Fulginei, F. R., and Salvini, A. (2015). On training efficiency and computational costs of a feed forward neural network: a review. *Computational intelligence and neuroscience*, 2015:83.

[Laudani et al., 2013a] Laudani, A., Mancilla-David, F., Riganti-Fulginei, F., and Salvini, A. (2013a). Reduced-form of the photovoltaic five-parameter model for efficient computation of parameters. *Solar Energy*, 97(0):122 – 127.

[Laudani et al., 2013b] Laudani, A., Riganti Fulginei, F., Salvini, A., Schmid, M., and Conforto, S. (2013b). Cfso 3: a new supervised swarm-based optimization algorithm. *Mathematical Problems in Engineering*, 2013.

[LeCun et al., 1989] LeCun, Y., Denker, J. S., Solla, S. A., Howard, R. E., and Jackel, L. D. (1989). Optimal brain damage. In *NIPs*, volume 89.

[Lee and Moraga, 1996] Lee, S.-W. and Moraga, C. (1996). A cosine-modulated gaussian activation function for hyper-hill neural networks. In *Signal Processing, 1996., 3rd International Conference on*, volume 2, pages 1397–1400. IEEE.

[Lei et al., 2011] Lei, P., Li, Y., and Seem, J. E. (2011). Sequential esc-based global mppt control for photovoltaic array with variable shading. *Sustainable Energy, IEEE Transactions on*, 2(3):348–358.

[Leung et al., 2003] Leung, F. H., Lam, H.-K., Ling, S.-H., and Tam, P. K. (2003). Tuning of the structure and parameters of a neural network using an improved genetic algorithm. *Neural Networks, IEEE Transactions on*, 14(1):79–88.

[Liu et al., 2013] Liu, Y.-H., Liu, C.-L., Huang, J.-W., and Chen, J.-H. (2013). Neural-network-based maximum power point tracking methods for photovoltaic systems operating under fast changing environments. *Solar Energy*, 89:42–53.

[Lozito et al., 2014a] Lozito, G. M., Bozzoli, L., and Salvini, A. (2014a). Microcontroller based maximum power point tracking through fcc and mlp neural networks. In *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, pages 207–211. IEEE.

[Lozito et al., 2014b] Lozito, G.-M., Laudani, A., Fulginei, F. R., and Salvini, A. (2014b). Fpga implementations of feed forward neural network by using floating point hardware accelerators. *Advances in Electrical and Electronic Engineering*, 12(1):30.

[Lozito and Salvini, 2014] Lozito, G. M. and Salvini, A. (2014). An empirical investigation on the static jiles-atherton model identification by using different set of measurements. In *AEIT Annual Conference-From Research to Industry: The Need for a More Effective Technology Transfer (AEIT), 2014*, pages 1–6. IEEE.

[Lozito et al., 2015] Lozito, G. M., Schmid, M., Conforto, S., Fulginei, F. R., and Bibbo, D. (2015). A neural network embedded system for real-time estimation of muscle forces. *Procedia Computer Science*, 51:60–69.

[Ma and Khorasani, 2005] Ma, L. and Khorasani, K. (2005). Constructive feed-forward neural networks using hermite polynomial activation functions. *Neural Networks, IEEE Transactions on*, 16(4):821–833.

[Mahamad and Saon, 2014] Mahamad, A. K. and Saon, S. (2014). Development of artificial neural network based mppt for photovoltaic system during shading condition. *Applied Mechanics and Materials*, 448:1573–1578.

[Mancilla-David et al., 2014] Mancilla-David, F., Riganti-Fulginei, F., Laudani, A., and Salvini, A. (2014). A neural network-based low-cost solar irradiance sensor. *Instrumentation and Measurement, IEEE Trans. on.*

[Marquardt, 1963] Marquardt, D. W. (1963). An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial & Applied Mathematics*, 11(2):431–441.

[Mellit and Pavan, 2010] Mellit, A. and Pavan, A. M. (2010). A 24-h forecast of solar irradiance using artificial neural network: Application for performance prediction of a grid-connected pv plant at trieste, italy. *Solar Energy*, 84(5):807–821.

[Moody et al., 1995] Moody, J., Hanson, S., Krogh, A., and Hertz, J. A. (1995). A simple weight decay can improve generalization. *Advances in neural information processing systems*, 4:950–957.

[Müller et al., 2002] Müller, S. D., Marchetto, J., Airaghi, S., and Kournoutsakos, P. (2002). Optimization based on bacterial chemotaxis. *Evolutionary Computation, IEEE Transactions on*, 6(1):16–29.

[Myers and Hutchinson, 1989] Myers, D. and Hutchinson, R. (1989). Efficient implementation of piecewise linear activation function for digital vlsi neural networks. *Electronics Letters*, 25:1662.

[Nelder and Mead, 1965] Nelder, J. A. and Mead, R. (1965). A simplex method for function minimization. *The computer journal*, 7(4):308–313.

[Oliveri et al.,] Oliveri, A., Cassottana, L., Laudani, A., Riganti Fulginei, F., Lozito, G., Salvini, A., and Storace, M. Two fpga-oriented high speed irradiance virtual sensors for photovoltaic plants.

[Oliveri and Storace, 2012] Oliveri, A. and Storace, M. (2012). Hardware-in-the-loop simulations of circuit architectures for the computation of exact and approximate explicit mpc control functions. In *Electronics, Circuits and Systems (ICECS), 2012 19th IEEE International Conference on*, pages 380–383. IEEE.

[Polak, 1971] Polak, E. (1971). *Computational methods in optimization*. Academic press.

[Powell, 1977] Powell, M. J. (1977). Restart procedures for the conjugate gradient method. *Mathematical programming*, 12(1):241–254.

[Punitha et al., 2013] Punitha, K., Devaraj, D., and Sakthivel, S. (2013). Artificial neural network based modified incremental conductance algorithm for maximum power point tracking in photovoltaic system under partial shading conditions. *Energy*, 62:330 – 340.

[Riedmiller and Braun, 1992] Riedmiller, M. and Braun, H. (1992). Rprop-a fast adaptive learning algorithm. In *Proc. of ISCIS VII), Universitat*. Citeseer.

[Riedmiller and Rprop, 1994] Riedmiller, M. and Rprop, I. (1994). Rprop-description and implementation details.

[Rojas, 2013] Rojas, R. (2013). *Neural networks: a systematic introduction*. Springer Science & Business Media.

[Shewchuk, 1994] Shewchuk, J. R. (1994). An introduction to the conjugate gradient method without the agonizing pain.

[Soria-Olivas et al., 2003] Soria-Olivas, E., Martín-Guerrero, J. D., Camps-Valls, G., Serrano-López, A. J., Calpe-Maravilla, J., and Gómez-Chova, L. (2003). A low-complexity fuzzy activation function for artificial neural networks. *IEEE Transactions on Neural Networks*, 14(6):1576–1579.

[Storace and Poggi, 2011] Storace, M. and Poggi, T. (2011). Digital architectures realizing piecewise-linear multivariate functions: Two fpga implementations. *International Journal of Circuit Theory and Applications*, 39(1):1–15.

[Storn and Price, 1997] Storn, R. and Price, K. (1997). Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359.

[Tisan et al., 2009] Tisan, A., Oniga, S., Mic, D., and Buchman, A. (2009). Digital implementation of the sigmoid function for fpga circuits. *Acta Technica Napocensis, Electronics and Telecommunications*, 50(2):15–20.

[Velasco-Quesada et al., 2009] Velasco-Quesada, G., Guinjoan-Gispert, F., Piqué-López, R., Román-Lumbreras, M., and Conesa-Roca, A. (2009). Electrical pv array reconfiguration strategy for energy extraction improvement in

grid-connected pv systems. *Industrial Electronics, IEEE Transactions on*, 56(11):4319–4331.

[Vincheh et al., 2014] Vincheh, M., Kargar, A., and Markadeh, G. (2014). A hybrid control method for maximum power point tracking (mppt) in photovoltaic systems. *Arabian Journal for Science and Engineering*, pages 1–11.

[Werbos, 1990] Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560.

[Wilamowski, 2009] Wilamowski, B. M. (2009). Neural network architectures and learning algorithms. *Industrial Electronics Magazine, IEEE*, 3(4):56–63.

[Wilamowski et al., 2008] Wilamowski, B. M., Cotton, N. J., Kaynak, O., and Dundar, G. (2008). Computing gradient vector and jacobian matrix in arbitrarily connected neural networks. *Industrial Electronics, IEEE Trans. on*, 55(10):3784–3790.

[Williams and Zipser, 1989] Williams, R. J. and Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280.

[Wolpert and Macready, 1997] Wolpert, D. H. and Macready, W. G. (1997). No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on*, 1(1):67–82.

[Wu et al., 1997] Wu, Y., Zhao, M., and Ding, X. (1997). Beyond weights adaptation: a new neuron model with trainable activation function and its supervised learning. In *Neural Networks, 1997., International Conference on*, volume 2, pages 1152–1157. IEEE.

[Yang, 2010] Yang, X.-S. (2010). *Nature-inspired metaheuristic algorithms*. Luniver press.

[Zhang et al., 2007] Zhang, J., Chung, H. S.-H., and Lo, W.-L. (2007). Clustering-based adaptive crossover and mutation probabilities for genetic algorithms. *Evolutionary Computation, IEEE Transactions on*, 11(3):326–335.

[Zhang et al., 2006] Zhang, L., Zhou, W., and Li, D.-H. (2006). A descent modified polak–ribière–polyak conjugate gradient method and its global convergence. *IMA Journal of Numerical Analysis*, 26(4):629–640.

[Zhang et al., 1996] Zhang, M., Vassiliadis, S., and Delgado-Frias, J. G. (1996). Sigmoid generators for neural computing using piecewise approximations. *Computers, IEEE Transactions on*, 45(9):1045–1049.

List of Figures