Roberto Lopez
rlopez@cimne.upc.edu

International Center for Numerical Methods in Engineering

OpenNN: Open Neural Networks Library
Version 0.9 (beta) manual.

Preface

Neural networks have found a wide range of applications, which include function regression, pattern recognition, time series prediction, optimal control, optimal shape design or inverse problems. A neural network can learn either from data sets or from mathematical models.

`OpenNN` is a comprehensive class library which implements neural networks in the C++ programming language. This software tool can be used for the whole range of applications mentioned above. `OpenNN` also provides a workaround for the solution of function optimization problems. The library has been released as the open source GNU Lesser General Public License.

This manual is organized as follows: Chapter 1 provides some guidelines for installing the software and using some basic data structures, such as vectors and matrices. In Chapter 2, a brief introduction to the principal concepts of neural networks is given. Also, in Chapter 3 the most general software model of `OpenNN` is presented. Chapters 4, 5 and 6 state the learning problem for neural networks and provide a collection of related algorithms. In Chapters 7, 8, 9, 10 and 11 the most important learning tasks for neural networks are formulated and several practical applications are also presented. Finally, Chapter 12 explains how to solve function optimization problems by means of `OpenNN`.

Contents

Chapter 1

Preliminaries

1.1 Building OpenNN

OpenNN has been designed for portability. This means that the library can be built on any operating system with little effort. For Windows, project files of Visual C++ are also included. Regarding Linux, simple makefiles are included in the distribution. There should be no problem in building OpenNN on other operating systems, since it has been written in ANSI C++.

Windows

Compiling OpenNN on Windows is easy. The library comes with project files for the latest version of Microsoft Visual C++ Express Edition. When working with another compiler is needed, a project for it must be created.

Microsoft Visual C++ 2010 Express Edition is a free, lightweight, easy-to-use, and easy-to-learn tools for the hobbyist, novice, and student developer. It can be downloaded at

http://www.microsoft.com/express

OpenNN includes the opennn.sln solution file for that compiler in the build/visual_studio folder.

To open the OpenNN project just double click on that file. A similar window than that depicted in Figure 1.1 should come up.

Pressing Ctrl+F5 will compile, build and run the test suite application. A MS-DOS console should appear with the following message:

```
...

OpenNN test suite results:
Tests run: tests_run
Tests passed: tests_run
Tests failed: 0
Test OK
```

This guarantees that OpenNN has been compiled properly, toguether with all the libraries included. On the other hand, many practical applications can be found in the same solution.

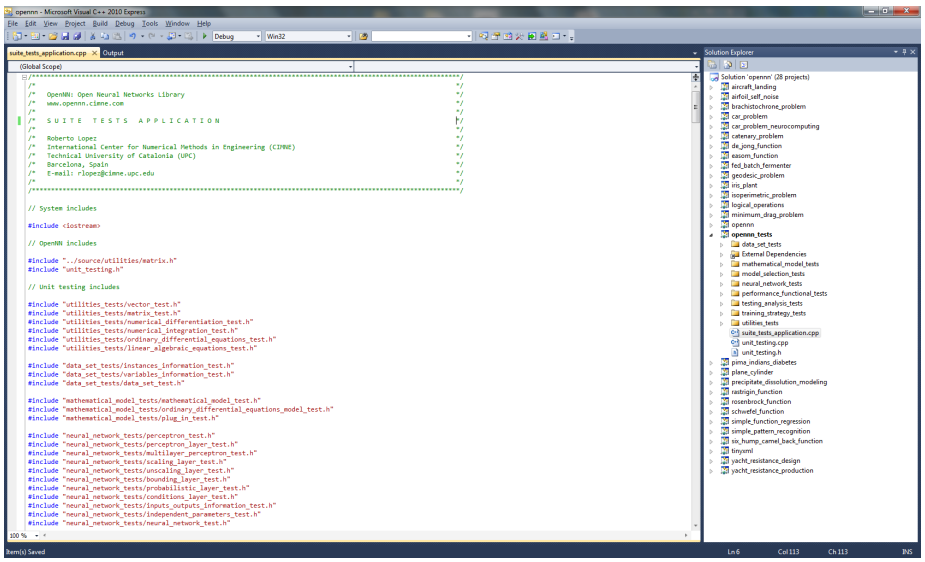

Figure 1.1: Microsoft Visual C++ 2010 solution view.

Note that project files of other versions than Visual C++ 2010 Express Edition are not guaranteed to be opened. In that case, and in order to use OpenNN, a new solution should be created.

Linux

Compilation of OpenNN in Linux is straight-forward, since simple makefiles are here provided. In order to do that, the following steps must be performed:

1. Extract the OpenNN.zip file to the installation folder.

To install OpenNN from the download location DOWNLOAD_DIRECTORY into the installation location INSTALLATION_DIRECTORY use the following commands:

```
>cd DOWNLOAD_DIRECTORY
>unzip OpenNN.zip INSTALLATION_DIRECTORY
```

You can specify any name for the installation folder. The name OpenNN will be used here.

2. Run the test suite makefile.

The folder \OpenNN\build\make contains a makefile for a test suite of the whole library. To run that makefile type the following commands on the terminal:

```
>cd \OpenNN\build
>make -f opennn_tests_makefile
```

This compiles all the classes included in OpenNN and builds a test suite for them. To verify the installation, run the test suite executable:

```
>./opennn_tests
```

If nothing has been wrong, the following message should appear on the terminal:

```
...
OpenNN test suite results:
Tests run: tests_run
Tests passed: tests_run
Tests failed: 0
Test OK
```

3. Run an example makefile.

The folder \OpenNN\build also contains makefiles for all the examples included in the distribution. To run the simple function regression example, type the following commands on the terminal:

```
>cd \OpenNN\build\make
>make -f simple_function_regression_makefile
```

Once all the classes have been compiled and the application has been built, you can run the example:

```
>./simple_function_regression
```

Read the application code to see what the simple function regression example does.

4. Removing a OpenNN Installation

To remove an `OpenNN` installation, enter the following command on the terminal:

```
>rm -rf \$OpenNN
```

This will delete the whole `OpenNN` folder.

1.2 OpenNN namespace

Each set of definitions in the `OpenNN` library is 'wrapped' in the namespace OpenNN. In this way, if some other definition has an identical name, but is in a different namespace, then there is no conflict.

The **using** directive makes a namespace available throughout the file where it is written [10]. For the OpenNN namespace the following sentence can be written:

```
using namespace OpenNN;
```

1.3 Vector template

The Vector class is a template, which means that it can be applied to different types [10]. That is, we can create a Vector or **int** numbers, MyClass objects, etc.

The Vector in `OpenNN` is derived from the vector in the Standard Template Library.

Members

The onl member of the Vector class is:

- A double pointer to some type.

That two class members are declared as being private.

File format

Vector objects can be serialized or deserialized to or from a data file which contains the member values. The file format is as follows.

```
element_0 element_1 ... element_N
```

Constructors

Multiple constructors are defined in the Vector class, where the different constructors take different parameters.

The easiest way of creating a vector object is by means of the default constructor, wich builds a vector of size zero. For example, in order to construct an empty Vector of **int** numbers we use

```
Vector<int> v;
```

The following sentence constructs a Vector of 3 **double** numbers.

```
Vector<double> v(3);
```

If we want to construct Vector of 5 **bool** variables and initialize all the elements to $false$, we can use

```
Vector<bool> v(5, false);
```

It is also possible to construct an object of the Vector class and at the same time load its members from a file. In order to do that we can do

```
Vector<int> v('Vector.dat');
```

The file 'Vector.dat' contains a first row with the size of the vector and an aditional row for each element of the vector.

The following sentence constructs a Vector which is a copy of another Vector,

```
Vector<MyClass> v(3);
Vector<MyClass> w(v);
```

Operators

The Vector class also implements different types of operators for assignment, reference, arithmetics or comparison.

The assignment operator copies a vector into another vector,

```
Vector<int> v;
Vector<int> w = v;
```

The following sentence constructs a vector and sets the values of their elements using the reference operator. Note that indexing goes from 0 to $n-1$, where n is the Vector size.

```
Vector<double> v(3);
v[0] = 1.0;
v[1] = 2.0;
v[2] = 3.0;
```

Sum, difference, product and quotient operators are included in the Vector class to perform arithmetic operations with a scalar or another Vector. Note that the arithmetic operators with another Vector require that they have the same sizes.

The following sentence uses the vector-scalar sum operator,

```
Vector<int> v(3, 1.0);
Vector<int> w = v + 3.1415926;
```

An example of the use of the vector-vector multiplication operator is given below,

```
Vector<double> v(3, 1.2);
Vector<double> w(3, 3.4);
Vector<double> x = v*w;
```

Assignment by sum, difference, product or quotient with a scalar or another Vector is also possible by using the arithmetic and assignent operators. If another Vector is to be used, it must have the same size.

For instance, to assign by difference with a scalar, we migh do

```
Vector<int> v(3, 2);
v -= 1;
```

In order to assign by quotation with another Vector, we can write

```
Vector<double> v(3, 2.0);
Vector<double> w(3, 0.5);
v /= w;
```

Equality and relational operators are also implemented here. They can be used with a scalar or another Vector. For the last case the same sizes are assumed.

An example of the equal to operator with a scalar is

```
Vector<bool> v(5, false);
bool is_equal = (v == false);
```

The less than operator with another Vector can be used as follows,

```
Vector<int> v(5, 2.3);
Vector<int> w(5, 3.2);
bool is_less = (v < w);
```

Methods

Get and set methods for each member of this class are implemented to exchange information among objects.

The method size returns the size of a Vector.

```
Vector<MyClass> v(3);
int size = v.size();
```

On the other hand, the method set sets a new size to a Vector. Note that the element values of that Vector are lost.

```
Vector<bool> v(3);
v.set(6);
```

If we want to initialize a vector at random we can use the initialize_uniform or initialize_normal methods,

```
Vector<double> v(5);
v.initialize_uniform();
Vector<double> w(3);
w.initialize_normal();
```

The Vector class also includes some mathematical methods which can be useful in the development of neural networks algorithms and applications.

The calculate_norm method calculates the norm of the vector,

```
Vector<double> v(5, 3.1415927);
double norm = v.calculate_norm();
```

In order to calculate the dot product between this Vector and another Vector we can do

```
Vector<double> v(3, 2.0);
Vector<double> w(3, 5.0);
double dot = v.dot(w);
```

We can calculate the mean or the standard deviation values of the elements in a Vector by using the calculate_mean and calculate_standard_deviation methods, respectively. For instance

```
Vector<double> v(3, 4.0);
double mean = v.calculate_mean();
double standard_deviation = v.calculate_standard_deviation();
```

Finally, utility methods for serialization or loading and saving the class members to a file are also included. In order to obtain a std::string representation of a Vector object we can make

```
Vector<bool> v(1, false);
std::string vector_string = v.to_string();
```

To save a Vector object to a file we can do

```
Vector<int> v(2, 0);
v.save('Vector.dat');
```

The first row of the file Vector.dat is the size of the vector and the other rows contain the values of the elements of that vector.

If we want to load a Vector object from a data file we could write

```
Vector<double> v;
v.load('Vector.dat');
```

Where the format of the Vector.dat file must be the same as that described above.

1.4 Matrix template

As it happens with the Vector class, the Matrix class is also a template [10]. Therefore, a Matrix of any type can be created.

Members

The Matrix class has three members:

- The number of rows.
- The number of columns.
- A double pointer to some type.

That members are private. Private members can be accessed only within methods of the class itself.

File format

The member values of a matrix object can be serialized or deserialized to or from a data file. The format is as follows.

```
element_00 ... element_0M
...
element_N0 ... element_NM
```

Constructors

The Matrix class also implements multiple constructors, with different parameters.

The default constructor creates a matrix with zero rows and zero columns,

```
Matrix<MyClass> m;
```

In order to construct an empty Matrix with a specified number of rows and columns we use

```
Matrix<int> m(2, 3);
```

We can specify the number of rows and columns and initialize the Matrix elements at the same time by doing

```
Matrix<double> m(1, 5, 0.0);
```

To build a Matrix object by loading its members from a data file the following constructor is used,

```
Matrix<double> m('Matrix.dat');
```

The format of a matrix data file is as follows: the first line contains the numbers of rows and columns separated by a blank space; the following data contains the matrix elements arranged in rows and columns. For instance, the next data will correspond to a Matrix of zeros with 2 rows and 3 columns,

```
2 3
0 0 0
0 0 0
```

The copy constructor builds an object which is a copy of another object,

```
Matrix<bool> a(3,5);
Matrix<bool> b(a);
```

Operators

The Matrix class also implements the assignment operator,

```
Matrix<double> a(2,1);
Matrix<bool> b = a;
```

Below there is an usage example of the reference operator here. Note that row indexing goes from 0 to rows_number−1 and column indexing goes from 0 to columns_number−1.

```
Matrix<int> m(2, 2);
m[0][0] = 1;
m[0][1] = 2;
m[1][0] = 3;
m[1][1] = 4;
```

The use of the arithmetic operators for the Matrix class are very similar to those for the Vector class. The following sentence uses the scalar difference operator,

```
Matrix<double> a(5, 7, 2.5);
Matrix<double> b = a + 0.1;
```

Also, using the arithmetic and assignment operators with the Matrix class is similar than with the Vector class. For instance, to assign by sum with another Matrix we can write

```
Matrix<double> a(1, 2, 1.0);
Matrix<double> b(1, 2, 0.5);
a += b;
```

The not equal to operator with another Matrix can be used in the following way,

```
Matrix<std::string> a(1, 1, 'hello');
Matrix<std::string> b(1, 1, 'good bye');
bool is_not_equal_to = (a != b);
```

The use of the greater than operator with a scalar is listed below

```
Matrix<double> a(2, 3, 0.0);
bool is_greater_than = (a > 1.0);
```

Methods

As it happens for the Vector class, the Matrix class implements get and set methods for all the members.

The get_rows_number and get_columns_number methods are very useful,

```
Matrix<MyClass> m(4, 2);
int rows_number = m.get_rows_number();
int columns_number = m.get_columns_number();
```

In order to set a new number of rows or columns to a Matrix object, the set_rows_number or set_columns_number methods are used,

```
Matrix<bool> m(1, 1);
m.set_rows_number(2);
m.set_columns_number(3);
```

A Matrix can be initialized with a given value, at random with an uniform distribution or at random with a normal distribution,

```
Matrix<double> m(4, 2);
m.initialize(0.0);
m.initialize_uniform(-0.2, 0.4);
m.initialize_normal(-1.0, 0.25);
```

A set of mathematical methods are also implemented for convenience. For instance, the dot method computes the dot product of this Matrix with a Vector or with another Matrix,

```
Matrix<double> m(4, 2, 1.0);
Vector<double> v(4, 2.0);
Vector<double> dot_product = m.dot(v);
```

Finally, string serializing, printing, saving or loading utility methods are also implemented. For example, the use of the print method is

```
Matrix<bool> m(1, 3, false);
m.print();
```

Chapter 2

Neural networks basis

In this Chapter we formulate the learning problem for neural networks and describe some learning tasks that they can solve.

2.1 Learning problem

Any application for neural networks involves a neural network itself, a performance functional, and a training strategy. The learning problem is then formulated as to find a neural network which optimizes a performance functional by means of a training strategy.

Neural network

A neuron model is a mathematical model of the behavior of a single neuron in a biological nervous system. The most important neuron model is the so called perceptron. The perceptron neuron model receives information in the form of numerical inputs. This information is then combined with a set of parameters to produce a message in the form of a single numerical output.

Most neural networks, even biological neural networks, exhibit a layered struc- ture. In this work layers are the basis to determine the architecture of a neural network. A layer of perceptrons taks a set of inputs in order to produce a set of outptus.

A multilayer perceptron is built up by organizing layers of perceptrons in a network architecture. In this way, the architecture of a network refers to the number of layers, their arrangement and connectivity. The characteristic network architecture in `OpenNN` is the so called feed-forward architecture. The multilayer perceptron can then be defined as a network architecture of perceptron layers. This neural network represents a parameterized function of several variables with very good approximation properties.

In order to solve practical applications, different extensions must be added to the multilayer perceptron. Some of them include scaling, unscaling, bounding, probabilistic or conditions layers. Therefore, the neural network in `OpenNN` is composed by a mutlilayer perceptron plus some additional layers.

Performance functional

The performance functional plays an important role in the use of a neural network. It defines the task the neural network is required to do and provides a measure of the quality of the representation that the neural network is required to learn. The choice of a suitable performance functional depends on the particular application.

A performance functional in `OpenNN` is composed of three different terms: objective, regularization and constraints. Most of the times, a single objective term will be enough, but some applications will require regularize solutions or will be defined by constraints on the solutions.

Learning tasks such as function regression and pattern recognition will have performance functionals measured on data sets. On the other hand, learning tasks such ase optimal control or optimal shape design will have performance functionasl measured on mathematical models. Finally, the performance functionals in another learning tasks, such ase inverse problems, will be measured in both data sets and mathematical models.

Training strategy

The procedure used to carry out the learning process is called training (or learning) strategy. The training strategy is applied to the neural network to in order to obtain the best possible performance. The type of training is determined by the way in which the adjustment of the parameters in the neural network takes place.

The most general training strategy in `OpenNN` will include three different training algorithms: initialization, main and refinement. Most applications will only need one training algorithm, but some complex problems might require the combination of two or three of them.

A generally good training strategy includes the quasi-Newton method. However, noisy problems might require an evolutionary algorithm. The first cited training algorithm is several orders of magnitude faster than the second one.

Learning activity diagram

The learning problem for neural networks is formulated from a variational point of view. Indeed, learning tasks lie in terms of finding a function which causes some functional to assume an extreme value. Neural networks provide a general framework for solving variational problems.

Figure 2.1 depicts an activity diagram for the learning problem. The solving approach here consists of three steps. The first step is to choose a suitable neural network which will approximate the solution to the problem. In the second step the variational problem is formulated by selecting an appropriate performance functional. The third step is to solve the reduced function optimization problem with a training strategy capable of finding an optimal set of parameters.

2.2 Learning tasks

Learning tasks for neural networks can be classified according to the source of information for them. There are basically two sources of information: data sets

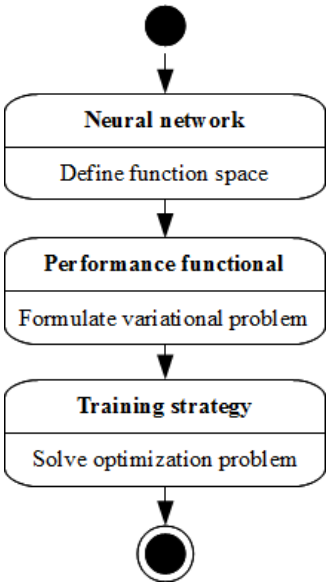

Figure 2.1: Learning problem for neural networks.

and mathematical models. In this way, some classes of learning tasks which learn from data sets are function regression, pattern recognition or time series prediction. On the other hand, learning tasks in which learning is performed from mathematical models are optimal control or optimal shape design. Finally, in inverse problems the neural network learns from both data sets and mathematical models.

Function regression

Function regression is the most popular learning task for neural networks. It is also called modelling. The function regression problem can be regarded as the problem of approximating a function from a data set consisting of input-target instances [20]. The targets are a specification of what the response to the inputs should be [5]. While input variables might be quantitative or qualitative, in function regression target variables are quantitative.

Performance measures for function regression are based on a sum of errors between the outputs from the neural network and the targets in the training data. As the training data is usually defficient, a regularization term might be required in order to solve the problem correctly.

An example is to design an instrument that can determine serum cholesterol levels from measurements of spectral content of a blood sample. There are a number of patients for which there are measurements of several wavelengths of the spectrum. For the same patients there are also measurements of several cholesterol levels, based on serum separation [9].

Pattern recognition

The learning task of pattern recognition gives raise to artificial intelligence. That problem can be stated as the process whereby a received pattern, characterized by a distinct set of features, is assigned to one of a prescribed number of classes [20]. Pattern recognition is also known as classification. Here the neural network learns from knowledge represented by a training data set consisting of input-target instances. The inputs include a set of features which characterize a pattern, and they can be quantitative or qualitative. The targets specify the class that each pattern belongs to and therefore are qualitative [5].

Classification problems can be, in fact, formulated as being modelling problems. As a consequence, performance functionals used here are also based on the sum squared error. Anyway, the learning task of pattern recognition is more difficult to solve than that of function regression. This means that a good knowledge of the state of the technique is recommended for success.

A typical example is to disinguish hand-written versions of characters. Images of the characters might be captured and fed to a computer. An algorithm is then seek to which can distinguish as reliably as possible between the characters [5].

Optimal control

Optimal control is playing an increasingly important role in the design of modern engineering systems. The aim here is the optimization, in some defined sense, of a physical process. More specifically, the objective of these problems is to determine the control signals that will cause a process to satisfy the physical constraints and at the same time minimize or maximize some performance criterion [22] [2].

The knowledge in optimal control problems is not represented in the form of a data set, it is given by a mathematical model. These objective functionals are often defined by integrals, ordinary differential equations or partial differential equations. In this way, and in order to evaluate them, we might need to apply Simpon methods, Runge-Kutta methods or finite element methods. Optimal control problems often include constraints.

As a simple example, consider the problem of a rocket launching a satellite into an orbit around the earth. An associated optimal control problem is to choose the controls (the thrust attitude angle and the rate of emission of the exhaust gases) so that the rocket takes the satellite into its prescribed orbit with minimum expenditure of fuel or in minimum time.

Optimal shape design

Optimal shape design is a very interesting field for industrial applications. The goal in these problems is to computerize the development process of some tool, and therefore shorten the time it takes to create or to improve the existing one. Being more precise, in an optimal shape design process one wishes to optimize some performance criterium involving the solution of a mathematical model with respect to its domain of definition [7].

As in the previous case, the neural network here learns from a mathematical model. Evaluation of the performance functional here might also need the integration of functions, ordinary differential equations or partial differential

equations. Optimal shape design problems defined by partial differential equations are challenging applications.

One example is the design of airfoils, which proceeds from a knowledge of computational fluid dynamics [13] [27]. The performance goal here might vary, but increasing lift and reducing drag are among the most common. Other objectives as weight reduction, stress reinforcement and even noise reduction can be obtained. On the other hand, the airfoil may be required to achieve this performance with constraints on thickness, pitching moment, etc.

Inverse problems

Inverse problems can be described as being opposed to direct problems. In a direct problem the cause is given, and the effect is determined. In an inverse problem the effect is given, and the cause is estimated [23] [34] [31]. There are two main types of inverse problems: input estimation, in which the system properties and output are known and the input is to be estimated; and properties estimation, in which the the system input and output are known and the properties are to be estimated. Inverse problems can be found in many areas of science and engineering.

This type of problems is of great interest from both a theoretical and practical perspectives. From a theoretical point of view, the neural network here needs both mathematical models and data sets. The aim is usually formulated as to find properties or inputs which make a mathematical model to comply with the data set. From a practical point of view, most numerical software must be tuned up before being on production. That means that the particular properties of a system must be properly estimated in order to simulate it well.

A typical inverse problem in geophysics is to find the subsurface inhomogeneities from collected scattered fields caused by acoustic waves sent at the surface and a mathematical model of soil mechanics.

Tasks companion diagram

As we have said, the knowledge for a neural network can be represented in the form of data sets or mathematical models. The neural network learns from data sets in function regression and pattern recognition; it learns from mathematical models in optimal control and optimal shape design; and it learns from both mathematical models and data sets in inverse problems. Please note that other possible applications can be added to these learning tasks.

Figure 2.2 shows the learning tasks for neural networks described in this section. As we can see, they are capable of dealing with a great range of applications. Any of that learning tasks is formulated as being a variational problem. All of them are solved using the three step approach described in the previous section. Modelling and classification are the most traditional; optimal control, optimal shape design and inverse problems can also be very useful.

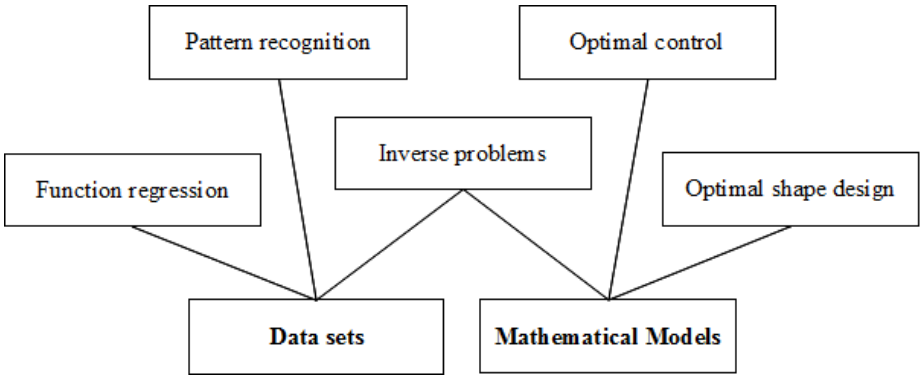

Figure 2.2: Learning tasks for neural networks.

Chapter 3

Software model basis

In this Chapter we present the software model of `OpenNN`. The whole process is carried out in the Unified Modeling Language (UML). The final implementation is written in the C++ Programming Language.

3.1 Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a general purpose visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system [32].

In order to construct a model for `OpenNN`, we follow a top-down development. This approach to the problem begins at the highest conceptual level and works down to the details. In this way, to create and evolve a conceptual class diagram, we iteratively model:

1. Classes.
2. Associations.
3. Compositions.
4. Derived classes.
5. Members.
6. Methods.

3.2 Classes

In colloquial terms a concept is an idea or a thing. In object-oriented modeling concepts are represented by means of classes [36]. Therefore, a prime task is to identify the main concepts (or classes) of the problem domain. In UML class diagrams, classes are depicted as boxes [32].

Through all this work, we have seen that general problemss can be solved with three elements: a neural network, a performance functional and a training strategy. The characterization in classes of these three concepts for `OpenNN` is as follows:

NeuralNetwork The class representing the concept of neural network is called NeuralNetwork.

Performance functional The class which represents the concept of performance functional is called PerformanceFunctional.

Training strategy The class representing the concept of training strategy is called TrainingStrategy.

Figure 3.1 depicts a starting UML class diagram for the conceptual model of `OpenNN`.

Figure 3.1: Conceptual diagram for `OpenNN`.

3.3 Associations

Once identified the main concepts in the model it is necessary to aggregate the associations among them. An association is a relationship between two concepts which points some significative or interesting information [36]. In UML class diagrams, an association is shown as a line connecting two classes. It is also possible to assign a label to an association. The label is typically one or two words describing the association [32].

The appropriate associations among the main concepts of `OpenNN` are next identified to be included to the UML class diagram of the system:

Neural network- Performance functional A neural network *has assigned* a performance functional.

Performance functional - Training strategy A performance functional *is improved by* a training strategy.

Figure 3.2 shows the above UML class diagram with these associations aggregated.

3.4 Composition

Classes are usually composed of another classes. The higher level classes manage the lower level ones.

Regarding `OpenNN`, the three main concepts described above are quite high level structures. This means that the neural network, performance functional and training algorithm classes are composed by different elements. In the next chapters the composition of the high level objects is explained in some detail.

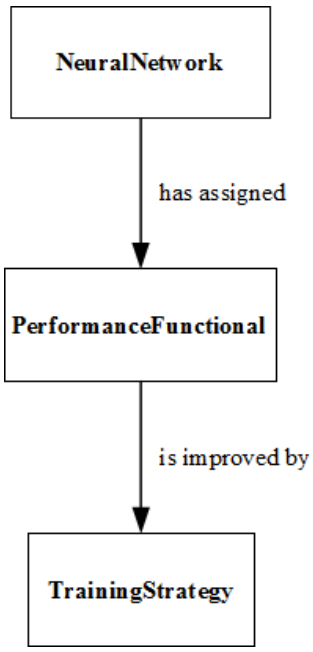

Figure 3.2: Aggregation of associations to the conceptual diagram.

3.5 Derived classes

In object-oriented programming, some classes are designed only as a parent from which sub-classes may be derived, but which is not itself suitable for instantiation. This is said to be an *abstract class*, as opposed to a *concrete class*, which is suitable to be instantiated. The derived class contains all the features of the base class, but may have new features added or redefine existing features [36]. Associations between a base class an a derived class are of the kind *is a* [32].

Some `OpenNN` classes are abstract, and concrete classes are derived from them. In the next chapters we will describe the intheritance of the main components of `OpenNN`: the neural network, the performance functional and the training strategy.

3.6 Members and methods

A member (or attribute) is a named value or relationship that exists for all or some instances of a class. A method (or operation) is a procedure associated with a class [36]. In UML class diagrams, classes are depicted as boxes with three sections: the top one indicates the name of the class, the one in the middle lists the attributes of the class, and the bottom one lists the operations [32].

The main members and methodss of the different `OpenNN` classes are described throughout all this manual.

Chapter 4

Neural network

The class of neural network implemented in `OpenNN` is based on the multilayer perceptron. That model is extended here to contain scaling, unscaling, bounding, probabilistic and conditions layers. A set of independent parameters associated to the neural network is also included here for convenience.

4.1 Basic theory

The neural network implemented in `OpenNN` is based on the multilayer perceptron. That classical model of neural network is also extended with scaling, unscaling, bounding, probabilistic and conditions layers, as well as a set of independent parameters.

Perceptron

A neuron model is the basic information processing unit in a neural network. They are inspired by the nervous cells, and somehow mimic their behaviour. The perceptron is the characteristic neuron model in the multilayer perceptron.

Following current practice [38], the term perceptron is here applied in a more general way than by Rosenblatt, and covers the types of units that were later derived from the original perceptron. Figure 4.1 is a graphical representation of a perceptron [20].

Here we identify three basic elements, which transform a vector of inputs into a single output [4]:

(i) A set of parameters consisting of a bias and a vector of synaptic weights.

(ii) A combination function.

(iii) An activation function or transfer function.

Perceptron layer

Most neural networks, even biological neural networks, exhibit a layered structure [38] [9]. In this work layers are the basis to determine the architecture of a neural network.

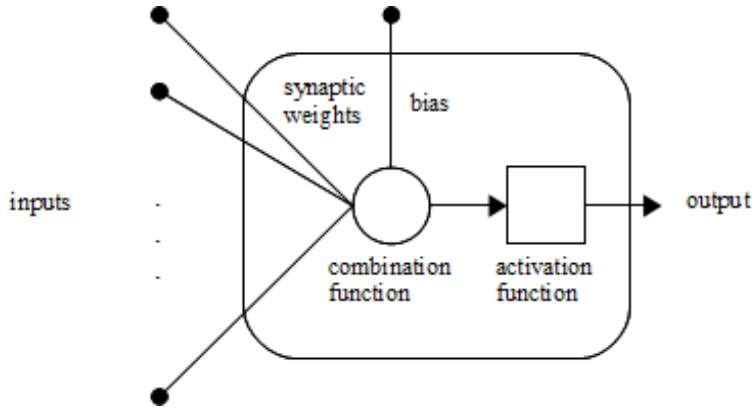

Figure 4.1: Perceptron neuron model.

A layer of perceptrons is composed by a set of perceptrons sharing the same inputs. The architecuture of a layer is characterized by the number of inputs and the number of perceptrons. Figure 4.2 shows a general layer of perceptrons.

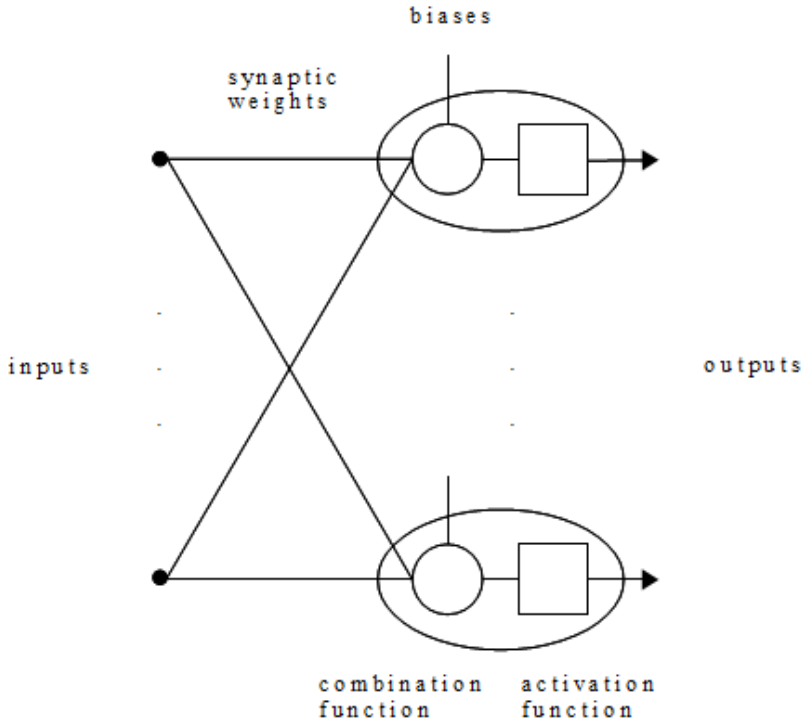

Figure 4.2: Layer.

Here we identify three basic elements, which transform a vector of inputs into a vector of outputs:

(i) A set of layer parameters.

(ii) A layer combination function.

(iii) A layer activation function.

Multilayer perceptron

Layers of perceptrons can be composed to form a multilayer perceptron. Most neural networks, even biological ones, exhibit a layered structure. Here layers and forward propagation are the basis to determine the architecture of a multilayer perceptron. This neural network represent an explicit function wich can be used for a variety of purposes.

The architecture of a multilayer perceptron refers to the number of neurons, their arrangement and connectivity. Any architecture can be symbolized as a directed and labeled graph, where nodes represent neurons and edges represent connectivities among neurons. An edge label represents the parameter of the neuron for which the flow goes in [4].

Thus, a neural network typically consists on a set of sensorial nodes which constitute the input layer, one or more hidden layers of neurons and a set of neurons which constitute the output layer.

There are two main categories of network architectures: acyclic or feed-forward networks and cyclic or recurrent networks [33]. A feed-forward network represents a function of its current input; on the contrary, a recurrent neural network feeds outptus back into its own inputs.

As it was said above, the characteristic neuron model of the multilayer perceptron is the perceptron. On the other hand, the multilayer perceptron has a feed-forward network architecture.

Hence, neurons in a feed-forward neural network are grouped into a sequence of layers of neurons, so that neurons in any layer are connected only to neurons in the next layer.

The input layer consists of external inputs and is not a layer of neurons; the hidden layers contain neurons; and the output layer is also composed of output neurons. Figure 4.3 shows the network architecture of a multilayer perceptron.

A multilayer perceptron is characterized by:

(i) A network architecture.

(ii) A set of parameters.

(iii) The layers activation functions.

Communication proceeds layer by layer from the input layer via the hidden layers up to the output layer. The states of the output neurons represent the result of the computation [38].

In this way, in a feed-forward neural network, the output of each neuron is a function of the inputs. Thus, given an input to such a neural network, the activations of all neurons in the output layer can be computed in a deterministic pass [5].

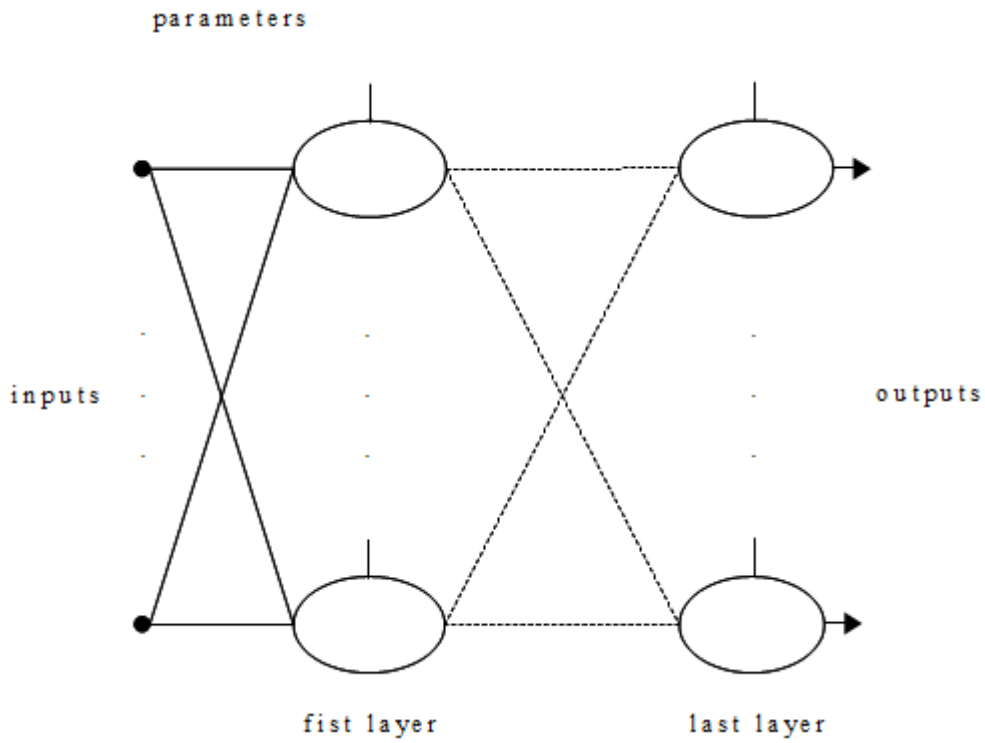

Figure 4.3: Multilayer perceptron.

Scaling layer

In practice it is always convenient to scale the inputs in order to make all of them to be of order zero. In this way, if all the neural parameters are of order zero, the outputs will be also of order zero. On the other hand, scaled outputs are to be unscaled in order to produce the original units.

In the context of neural networks, the scaling function can be thought as an additional layer connected to the input layer of the multilayer perceptron. The number of scaling neurons is the number of inputs, and the connectivity of that layer is not total, but one-to-one.

The scaling layer contains some basic statistics on the inputs. They include the mean, standard deviation, minimum and maximum values. Two scaling methods very used in practice are the minimum-maximum and the mean-standard deviation methods.

Unscaling layer

Also, scaled outputs from a multilayer perceptron are to be unscaled in order to produce the original units. In the context of neural networks, the unscaling function can be interpreted as an unscaling layer connected to the outputs of the multilayer perceptron.

The unscaling layer contains some basic statistics on the outputs. They

include the mean, standard deviation, minimum and maximum values. Two unscaling methods very used in practice are the minimum-maximum and the mean-standard deviation methods.

Bounding layer

Lower and upper bounds are an essential issue for that problems in which some variables are restricted to fall in an interval. Those problems could be intractable if bounds are not applied.

An easy way to treat lower and upper bounds is to post-process the outputs from the neural network with a bounding function. That function can be also be interpreted as an additional layer connected to the outputs.

Probabilistic layer

A probabilistic function takes the outputs to produce new outputs whose elements can be interpreted as probabilities. In this way, the probabilistic outputs will always fall in the range $[0, 1]$, and the sum of all will always be 1. This form of postprocessing is often used in patter recognition problems.

The probabilistic function can be interpreted as an additional layer connected to the output layer of the network architecture.

Note that the probabilistic layer has total connectivity, and that it does not contain any parameter. Two well-known probabilistic methods are the competitive and the softmax methods.

Conditions layer

If some outputs are specified for given inputs, then the problem is said to include conditions.

A conditions layer will be connected to the outputs of the multilayer perceptron. It will take both the input and the output values from that neural network to produce new outptus satisfying the conditions.

Treatment of conditions is quite a difficult task. Here a particular and a homogeneous solutions are first derived. Then, both the inputs and the outputs from the multilayer perceptron are processed with the particular and homogeneous functions in order to hold the conditions.

Independent parameters

If some information not related to input-output relationships is needed, then the problem is said to have independent parameters. They are not a part of the neural network, but they are associated to it.

The independent parameters are grouped together in a vector.

Neural network

A neural network defines a function which is of the following form.

$$outputs = function(inputs).$$

The most important element of an OpenNN neural network is the multilayer perceptron. That composition of layers of perceptrons is a very good function approximator.

Many practical applications require, however, extensions to the multilayer perceptron. OpenNN presents a neural network with some of the most standard extensions. They include the scaling, uscaling, bounding, probabilistic or conditions layers.

For instance, a function regression problem might require a multilayer perceptron with scaling and unscaling layers. On the other hand, an optimal control problem may need a multilayer perceptron with a conditions layer.

Finally, some problems might require the use of other adjustable parameters than those belonging to the multilayer perceptron. That kind of parameters are called independent parameters.

Some basic information related to the input and output variables of a neural network includes the name, description and units of that variables. That information will be used to avoid errors such as interchanging the role of the variables, misunderstanding the significance of a variable or using a wrong units system.

4.2 Software model

As we have seen, the OpenNN neural network is composed by a multilayer perceptron plus some other kinds of layers. In this section we study the software model of the NeuralNetwork class.

Classes

The characterization in classes of the concepts studied in the previous section is as follows:

Perceptron The class which represents the concept of perceptron neuron model is called Perceptron.

PerceptronLayer The class representing a layer of perceptrons is called PerceptronLayer.

MultilayerPerceptron The class which represents a feed-forward architecture of perceptron layers is called MultilayerPerceptron.

Scaling layer The class which represents a layer for scaling variables is called ScalingLayer.

Unscaling layer The class which represents an unscaling layer is called UnscalingLayer.

Bounding layer The class representing a layer of bounding neurons is called BoundingLayer.

Conditions layer The class wich applies input-output conditions is called ConditionsLayer.

Independent parameters A class containing parameters not belonging to the multilayer perceptron is called IndependentParameters.

Neural network The class which aggregates all the different neural network concepts is called NeuralNetwork.

Associations

The appropriate associations between all the neural networks classes in the system are next identified to be included to the association diagram:

Perceptron layer - perceptron A layer of perceptrons is composed of perceptrons.

Multilayer perceptron - perceptron layer A multilayer perceptron is composed of layers of perceptrons.

Neural network - Multilayer perceptron A neural network very probably contains a multilayer perceptron.

Neural network - Scaling layer A multilayer perceptron usually contains a scaling layer.

Neural network - Unscaling layer A multilayer perceptron usually contains an unscaling layer.

Neural network - Bounding layer A multilayer perceptron sometimes contains a bounding layer.

Neural network - Probabilistic layer A multilayer perceptron sometimes contains a probabilistic layer.

Neural network - Conditions layer A multilayer perceptron might contain a conditions layer.

Neural network - Independent parameters A multilayer percetpron might contain a set of independent parameters.

Figure 4.4 depicts an association diagram for the neural network class.

Derived classes

The next task is then to establish which classes are abstract and to derive the necessary concrete classes to be added to the system.

The neural network class in `OpenNN` will be intensively used by any application. Therefore, for performance reasons, all the composing classes have been designed to be concrete.

Let us then examine the classes we have so far:

Perceptron The class Perceptron is concrete, and can implement different activation functions.

Perceptron layer The class PerceptronLayer is also concrete, since it is defined as a vector of perceptrons.

Multilayer perceptron The class MultilayerPerceptron is a concrete class and is itself suitable for instantiation. This class is implemented as a vector of layers of perceptrons.

Scaling layer The class ScalingLayer is concrete, and implements the minimum-maximum and mean-standard deviation scaling methods.

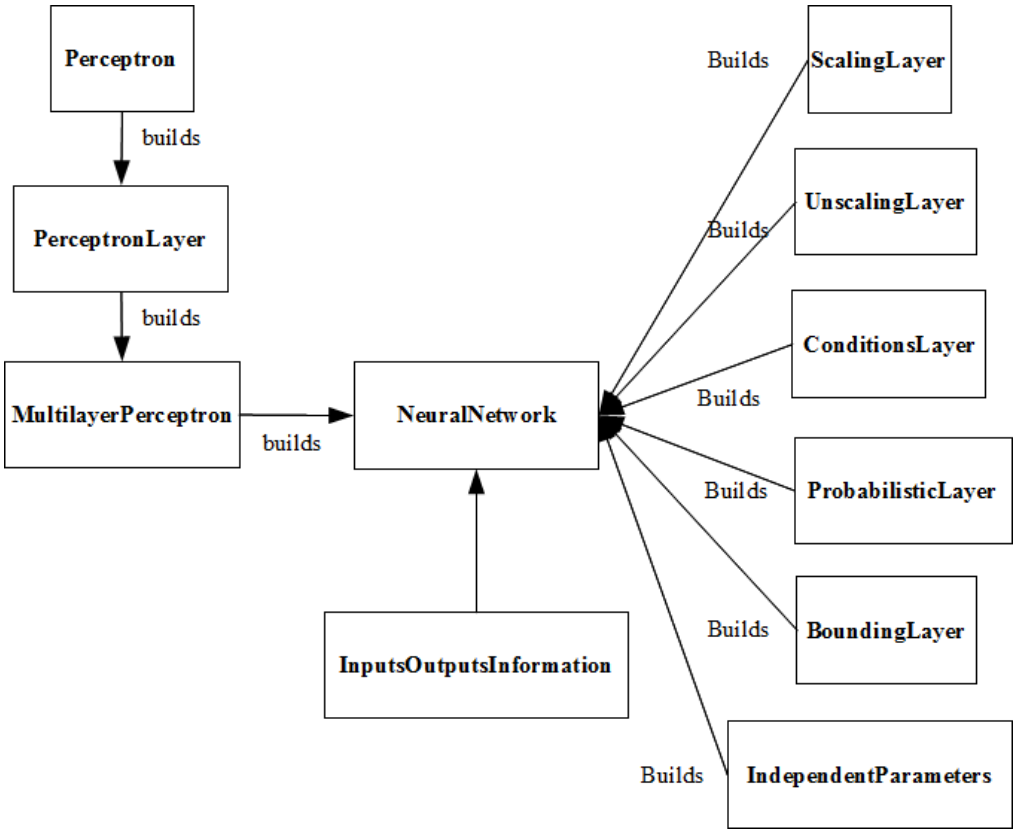

Figure 4.4: Association diagram for the NeuralNetwork class.

Unscaling layer The class UnscalingLayer is also concrete, and implements the minimum-maximum and mean-standard deviation unscaling methods.

Bounding layer The class BoundingLayer is concrete. It sets to their bound values those inputs which are below or above them.

Probabilistic layer The class ProbabilisticLayer is concrete, and implements the competitive and softmax methods.

Conditions layer The class ConditionsLayer has also been designed to be concrete. It implements methods to hold one or two condions. For more difficult situations, further classes must be derived.

Inputs-outputs information The class InputsOutputsInformation is concrete. It mainly stores a few strings with the names, units and descriptions of the neural network variables.

Independent parameters The class IndependentParameters is concrete. It contains other adjustable parameters than those belonging to the multilayer perceptron.

Attributes and operations

An attribute is a named value or relationship that exists for all or some instances of a class. An operation is a procedure associated with a class [36].

In UML class diagrams, classes are depicted as boxes with three sections: the top one indicates the name of the class, the one in the middle lists the attributes of the class, and the bottom one lists the operations [32].

Perceptron A perceptron neuron model has the following attributes:

- A bias.
- A set of synaptic weights.
- The activation function.

It performs the following main operations:

- Calculate the neuron output for a given input.
- Calculate the derivatives of the output with respect to the inputs.

Perceptron layer The perceptron layer has the following members:

- A set of perceptrons.

It performs the following methods:

- Calculate the layer output for a given input.
- Calculate the derivatives of the outputs with respect to the inputs.

Multilayer perceptron A multilayer perceptron has the following attributes:

- A set of layers of perceptrons.

It performs the following main operations:

- Calculate the output for a given input.
- Calculate the Jacobian for a given input.
- Calculate the Hessian form for a given input.

Scaling layer The scaling layer has the following members:

- The main statistics of the variables.
- The scaling method.

It implements the following main members:

- Calculate the scaled variables for unscaled variables.
- Calculate the derivatives of the scaling function.

Unscaling layer The unscaling layer is similar to the scaling layer, with the following members:

- The main statistics of the variables.
- The unscaling method.

It implements the following main members:

- Calculate the unscaled variables for scaled variables.
- Calculate the derivatives of the unscaling function.

Bounding layer The bounding layer contains the following attributes:

- The lower and upper bounds of the variables.

It performs the following main operations:

- Calculate bounded variables for unbounded ones.
- Calculate the derivatives of the bounding function.

Probabilistic layer The probabilist layer contains:

- The probabilistic method.

It computes the following functions:

- Calculate probabilistic variables for non-probabilistic ones.
- Calculate also the derivatives.

Conditions layer The conditions layer contains the following:

- The conditions values.
- The conditions method.

It performs the following:

- Calculate outputs holding some conditions.
- Calculate also the derivatives of that conditioned outputs.

Inputs-outputs information This class stores the following data:

- The names, units and descriptions of the input and output variables.

It performs the following:

- Write default names for the inputs and the outputs.

Independent parameters The class representing independent parameters contains the following main members:

- A set of parameters.
- Information and statistics on the parameters.
- Scaling/Unscaling and bounding methods.

The independent parameters class can perform the follwoing operations:

- Scale and unscale the parameters.
- Bounding the parameters.

4.3 NeuralNetwork classes

As it has been said, OpenNN implements quite a general neural network in the class NeuralNetwork. It contains a multilayer perceptron with an arbitrary number of layers of perceptrons. On the other hand, it includes aditional layers for inputs scaling, outputs unscaling, outputs bouding, outputs probabilizing or outputs holding some other conditions. This neural network can deal with a wide range of problems. Finally this class includes independent parameters, which can be useful for some problems.

The NeuralNetwork class is one of the most important in OpenNN, having many different members, constructors and methods.

Members

The NeuralNetwork class contains:

- A pointer to a multilayer perceptron.
- A pointer to a scaling layer.
- A pointer to an unscaling layer.
- A pointer to a bounding layer.
- A pointer to a probabilistic layer.
- A pointer to a conditions layer.
- A pointer to an inputs-outputs information object.
- A pointer to a set of independent parameters.

All that members are declared as private, and they can only be used with their corresponding get or set methods.

Constructors

There are several constructors for the NeuralNetwork class, with different arguments.

The default activation function for the hidden layers is the hyperbolic tangent, and for the output layer is the linear. No default information, statistics, scaling, boundary conditions or bounds are set.

The easiest way of creating a neural object is by means of the default constructor, which creates an empty neural network.

```
NeuralNetwork nn;
```

To construct neural network having a multilayer perceptron with, for example, 3 inputs and 2 output neurons, we use the one layer constructor

```
NeuralNetwork nn(3, 2);
```

All the parameters in the multilayer perceptron object that we have constructed so far are initialized with random values chosen from a normal distribution with mean 0 and standard deviation 1. By default, this one-layer perceptron will have linear activation function.

To construct a neural network containing a multilayer perceptron object with, for example, 1 input, a single hidden layer of 3 neurons and an output layer with 2 neurons, we use the two layers constructor

```
NeuralNetwork nn(1,6,2);
```

All the parameters here are also initialized at random. By default, the hidden layer will have hyperbolic tangent activation function and the output layer will have linear activation function.

In order to construct a neural network with a more complex multilayer perceptron, its architecture must be specified in a vector of unsigned integers. For instance, to construct a multilayer perceptron with 1 input, 3 hidden layers with 2, 4 and 3 neurons and an output layer with 1 neuron we can write

```
Vector<unsigned int> architecture(5);
architecture[0] = 1;
architecture[1] = 2;
architecture[2] = 4;
architecture[3] = 3;
architecture[4] = 1;

NeuralNetwork nn(architecture);
```

The network parameters here are also initialized at random.

The independent parameters constructor creates a neural network object without a multilayer perceptron but with a given number of independent parameters,

```
NeuralNetwork nn(3);
```

The above object can be used, for instance, as the basis for solving a function optimization problem not related to neural networks.

It is possible to construct a neural network by loading its members from a XML file. That is done in the following way,

```
NeuralNetwork nn('neural_network.xml');
```

Please follow strictly the format of the neural network file.

Finally, the copy constructor can be used to create an object by copying the members from another object,

```
NeuralNetwork nn1(2, 4, 3);
NeuralNetwork nn2(&nn1);
```

Methods

This class implements get and set methods for each member. The following sentences show the use of some of them,

```
NeuralNetwork nn(3, 2);

MultilayerPerceptronPointer* mlpp = nn.get_multilayer_perceptron_pointer();

unsigned int inputs_number = mlpp.count_inputs_number();
unsigned int outputs_number = mlpp.count_outputs_number();
```

The number of parameters of the neural network above can be accessed as follows

```
unsigned int parameters_number = nn.count_parameters_number();
```

The network parameters can be initialized with a given value by using the initialize method,

```
NeuralNetwork nn(4, 3, 2);

nn.initialize(0.0);
```

To calculate the output Vector of the network in response to an input Vector we use the method calculate_outputs. For instance, the sentence

```
Vector<double> inputs(1);
inputs[0] = 0.5;

Vector<double> outputs = nn.calculate_outputs(inputs);
```

returns the neural network output value for an input value.

To calculate the Jacobian Matrix of the network in response to an input Vector we use the method the method calculate_Jacobian. For instance, the sentence

```
Matrix<double> Jacobian = nn.calculate_Jacobian(inputs);
```

returns the partial derivatives of the outputs with respect to the inputs.

We can save a neural network object to a data file by using the method save. For instance,

```
NeuralNetwork nn;

nn.save('neural_network.xml');
```

saves the neural network object to the file neural_network.xml.

We can also load a neural network object from a data file by using the method load. Indeed, the sentence

```
nn.load('neural_network.xml');
```

loads the neural network object from the file neural_network.xml.

XML formats

Multilayer perceptron

The format of a XML multilayer perceptron element is the following,

```
<MultilayerPerceptron>
   <Architecture>vector</Architecture>
   <Parameters>vector</Parameters>
   <ActivationFunctions>vector</ActivationFunctions>
   <Display>bool</Display>
</MultilayerPerceptron>
```

Scaling layer

The format of a XML scaling layer element is the following,

```
<ScalingLayer>
    <Minimums>vector</Minimums>
    <Maximums>vector</Maximums>
    <Means>vector</Means>
    <StandardDeviations>vector</StandardDeviations>
    <ScalingMethod>string</ScalingMethod>
    <Display>bool</Display>
</ScalingLayer>
```

Unscaling layer

The format of a XML unscaling layer element is the following,

```
<UnscalingLayer>
    <Minimums>vector</Minimums>
    <Maximums>vector</Maximums>
    <Means>vector</Means>
    <StandardDeviations>vector</StandardDeviations>
    <UnscalingMethod>string</UnscalingMethod>
    <Display>bool</Display>
</UnscalingLayer>
```

Bounding layer

The format of a XML bounding layer element is the following,

```
<BoundingLayer>
    <LowerBounds>vector</LowerBounds>
    <UpperBounds>vector</UpperBounds>
    <Display>bool</Display>
</BoundingLayer>
```

Probabilistic layer

The format of a XML probabilistic layer element is the following,

```
<ProbabilisticLayer>
    <ProbabilisticNeuronsNumber>integer</ProbabilisticNeuronsNumber>
    <ProbabilisticMethod>string</ProbabilisticMethod>
    <Display>bool</Display>
</ProbabilisticLayer>
```

Conditions layer

The format of a XML conditions layer element is the following,

```
<ConditionsLayer>
    <InputsNumber>integer</InputsNumber>
    <ConditionsNeuronsNumber>integer</ConditionsNeuronsNumber>
    <ConditionsMethod>string</ConditionsMethod>
    <Display>bool</Display>
</ConditionsLayer>
```

Inputs-outputs information

```
<InputsOutputsInformation>
   <InputsName>
      <InputName Index="1">string</InputName>
      ...
      <InputName Index="N">string</InputName>
   </InputsName>
   <InputsUnits>
      <InputUnits Index="1">string</InputUnits>
      ...
      <InputUnits Index="N">string</InputUnits>
   </InputsUnits>
   <InputsDescription>
      <InputDescription Index="1">string</InputDescription>
      ...
      <InputDescription Index="N">string</InputDescription>
   </InputsDescription>
   <OutputsName>
      <OutputName Index="1">string</OutputName>
      ...
      <OutputName Index="N">string</OutputName>
   </OutputsName>
   <OutputsUnits>
      <OutputUnits Index="1">string</OutputUnits>
      ...
      <OutputUnits Index="N">string</OutputUnits>
   </OutputsUnits>
   <OutputsDescription>
      <OutputDescription Index="1">string</OutputDescription>
      ...
      <OutputDescription Index="M">string</OutputDescription>
   </OutputsDescription>
</InputsOutputsInformation>
```

Independent parameters

The format of a XML independent parameters element is the following,

```
<IndependentParameters>
   <Parameters>vector</Parameters>
   <Names>
      <Name Index="1">string</Name>
      ...
      <Name Index="NIP">string</Name>
   </Names>
   <Units>
      <Unit Index="1">string</Unit>
      ...
      <Unit Index="NIP">string</Unit>
   </Units>
   <Descriptions>
      <Description Index="1">string</Description>
      ...
      <Description Index="NIP">string</Description>
   </Units>
   <Minimums>vector</Minimums>
   <Maximums>vector</Maximums>
   <Means>vector</Means>
   <StandardDeviations>vector</StandardDeviations>
   <LowerBounds>vector</LowerBounds>
   <UpperBounds>vector</UpperBounds>
   <ScalingMethod>string</ScalingMethod>
```

```
    <ScalingFlag>boolean</ScalingFlag>
    <BoundingFlag>boolean</BoundingFlag>
    <DisplayRangeWarning>bool</DisplayRangeWarning>
    <Display>bool</Display>
</IndependentParameters>
```

Neural network

Finally, the format of a XML independent parameters element is the following,

```
<NeuralNetwork>
    <MultilayerPerceptron>
    multilayer_perceptron_element
    </MultilayerPerceptron>
    <ScalingLayer>
    scaling_layer_element
    </ScalingLayer>
    <UnscalingLayer>
    unscaling_layer_element
    </UnscalingLayer>
    <BoundingLayer>
    bounding_layer_element
    </BoundingLayer>
    <ProbabilisticLayer>
    probabilistic_layer_element
    </ProbabilisticLayer>
    <ConditionsLayer>
    conditions_layer_element
    </ConditionsLayer>
    <MultilayerPerceptronFlag>
    bool
    </MultilayerPerceptronFlag>
    <ScalingLayerFlag>
    bool
    </ScalingLayerFlag>
    <UnscalingLayerFlag>
    bool
    </UnscalingLayerFlag>
    <BoundingLayerFlag>
    bool
    </BoundingLayerFlag>
    <ProbabilisticLayerFlag>
    bool
    </ProbabilisticLayerFlag>
    <ConditionsLayerFlag>
    bool
    </ConditionsLayerFlag>
    <Display>
    bool
    </Display>
</NeuralNetwork>
```

Chapter 5

Performance functional

The performance functional defines the learning task for a neural network. In `OpenNN`, a performance functional consists of three different terms: objective, regularization and constraints.

5.1 Basic theory

Objective functional

The objective is the most important term in the performance functional expression. It defines the task that the neural network is required to accomplish.

For data modeling applications, such as function regression or pattern recognition, the sum squared error is the reference objective functional. It measures the diffence between the outputs from a neural network and the targets in a data set. Some related objective functionals here are the normalized squared error or the Minkowski error.

Applications in which the neural network learns from a mathematical model require other objective functionals. For instance, we can talk about minimum final time or desired trajectory optimal control problems. That two performance terms are called in `OpenNN` independent parameters error and solutions error, respectively.

Regularization functional

A problem is called well-posed if its solution meets existence, uniqueness and stability. A solution is said to be stable when small changes in the independent variable led to small changes in the dependent variable. Otherwise the problem is said to be ill-posed.

An approach for ill-posed problems is to control the effective complexity of the neural network [37]. This can be achieved by using a regularization term into the performance functional.

One of the simplest forms of regularization term consists on the norm of the neural parameters vector [5]. Adding that term to the objective functional will cause the neural network to have smaller weights and biases, and this will force its response to be smoother.

Regularization can be used in problems which learn from a data set. Function regression or pattern recognition problems with noisy data sets are common applications. It is also useful in optimal control problems which aim to save control action. More information of regularization theory for neural networks can be found in [17] and [8].

Constraints functional

A variational problem for a neural network can be specified by a set of constraints, which are equalities or inequalities that the solution must satisfy. Such constraints are expressed as functionals.

Here the aim is to find a solution which makes all the constraints to be satisfied and the objective functional to be an extremum.

Constraints are required in many optimal control or optimal shape design problems. For instance, we can talk about lenght, area or volume constraints. That type of performance term is called in `OpenNN` final solutions error.

Performance functional

The performance measure is a functional of the neural network which can take the following forms:

$$
\begin{aligned}
\text{performance functional} &= \text{Functional[neural network]},\\
\text{performance functional} &= \text{Functional[neural network, data set]},\\
\text{performance functional} &= \text{Functional[neural network, mathematical model]},\\
\text{performance functional} &= \text{Functional[neural network, mathematical model, data set]}.
\end{aligned}
$$

In order to perform a particular task a neural network must be associated a performance functional, which depends on the variational problem at hand. The learning problem is thus formulated in terms of the minimization of the performance functional.

The performance functional defines the task that the neural network is required to accomplish and provides a measure of the quality of the representation that the neural network is required to learn. In this way, the choice of a suitable performance functional depends on the particular application.

The learning problem can then be stated as to find a neural network for which the performance functional takes on a minimum or a maximum value. This is a variational problem.

In the context of neural network, the variational problems are can be treated as a function optimization problem. The variational approach looks at the performance as being a functional of the function represented by the neural network. The optimization approach looks at the performance as being a function of the parameters in the neural network.

Then, a performance function can be visualized as a hypersurface, with the neural network parameters as coordinates, see Figure 6.1.

The performance function evaluates the performance of the neural network by looking at its parameters. More complex calculations allow to obtain some partial derivatives of the performance with respect to the parameters. The

first partial derivatives are arranged in the gradient vector. The second partial derivatives are arranged in the Hessian matrix.

When the desired output of the neural network for a given input is known, the gradient and Hessian can usually be found analytically using back-propagation. In some other circumstances exact evaluation of that quantities is not possible and numerical differentiation must be used.

performance functional = objective term + regularization term + constraints term

5.2 Software model

Classes

In order to construct a software model for the performance functional, a few things need to be taken into account. First, a performance functional can be measured on a data set, on a mathematical model or on both of them. Second, a performance functional might be composed by three terms: an objective functional, a regularization functional and a constraints functional. Third, sometimes we will need numerical differentiation to calculate the derivatives of the performance with respect to the parameters in the neural network.

Data set The class which represents the concept of data set is called DataSet. This class is basically a data matrix with information on the variables (input or target) and the instances (training, generalization or testing).

Mathematical model The class representing the concept of mathematical model is called MathematicalModel. This is a very abstract class for calculating the solution of some mathematical model for a given input to that model.

Performance term The class which represents the concepts of objective, regularization and constraints terms is called PerformanceTerm. The objective functional is the most important term in the performance functional expression.

Numerical differentiation The class with utilities for numerical differentiation is called NumericalDifferentiation. While it is not needed for data modelling problems, it is in general a must when solving optimal control, optimal shape design or inverse problems.

Performance functional The class which represents the concept of performance functional is called PerformanceFunctional. A performance functional is defined as the sum of the objective, regularization and constraints functionals.

Associations

The associations among the concepts described above are the following:

Performance functional - Data set A performance functional might be measured on a data set.

Performance functional - Mathematical model A performance functional might be measured on a mathematical model.

Performance functional - Objective functional A performance functional might contain an objective term.

Performance functional - Regularization functional A performance functional might contain a regularization term.

Performance functional - Constraints functional A performance functional might contain a constraints term.

Figure 5.1 depicts an association diagram for the performance functional class.

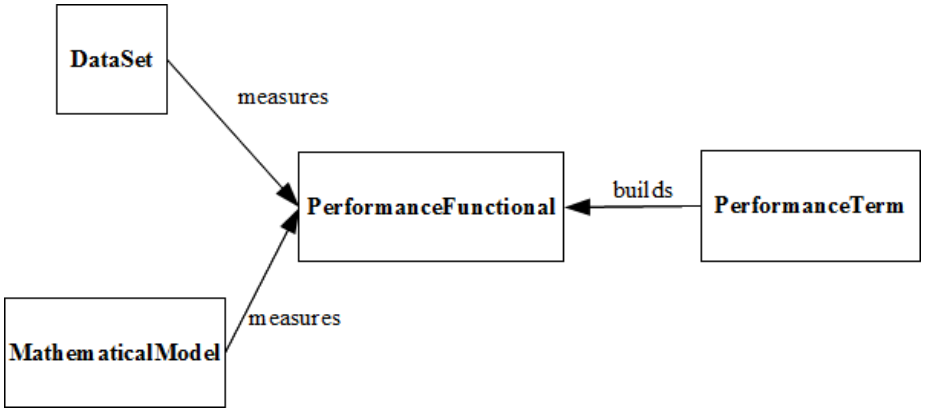

Figure 5.1: Association diagram for the PerformanceFunctional class.

Derived classes

The next task is then to establish which classes are abstract and to derive the necessary concrete classes to be added to the system. Let us then examine the classes we have so far:

Data set This is called DataSet, and it is a concrete class. It can be instantiated by loading the data matrix from a file and setting the variables and instances information.

Mathematical model The class MathematicalModel is abstract, since it needs a concrete representation. Derived classes here include OrdinaryDifferentialEquations and PlugIn. The mathematical model depends on the particular application, so further derivation might be needed. It is a current research line to get closer to a concrete nature of this class, by the use of a mathematical parser.

Performance term The class PerformanceTerm is abstract, because it does not represent a concrete performance term. Indeed, that depends on the problem at hand.

Some suitable error functionals for data function regression, pattern recognition and time series prediction problems are the sum squared error, the mean squared error, the root mean squared error, the normalized squared error or the Minkowski error. Therefore the SumSquaredError, MeanSquaredError, RootMeanSquaredError, NormalizedSquaredError and MinkowskiError concrete classes are derived from the PerformanceTerm abstract class. All of these error functionals are measured on a data set.

A specific objective functional for pattern recognition is the cross entropy error. This derived class is called CrossEntropyError.

Some common objective functionals for optimal control are also included.

A class InverseSumSquaredError for inverse problems involving a data set and a mathematical model is finally derived.

The most common regularization functional is the norm of the multilayer perceptron parameters. This method is implemented in the NeuralParametersNorm derived class.

Another useful regularization term consists on the integrals of the neural network outputs. This is included in the OutputsIntegrals class.

There are some common constraints functionals for optimal control or optimal shape design problems, such as the final solutions error. This derived class is called FinalSolutionsError. Related names here include SolutionsError or IndependentParametersError.

Performance functional The class PerformanceFunctional is concrete, since it is composed of a concrete objective functional, a concrete regularization functional and a concrete constraints functional. The performance functional is one of the main classes of `OpenNN`.

Figure 5.2 shows the UML class diagram for the with some of the derived classes included.

Attributes and operations

Data set

A data-set has the following attributes:

- A data matrix.

- A variables information object.

- An instances information object.

It performs the following operations:

- Load the data from a file.

- Scale/unscale the data.

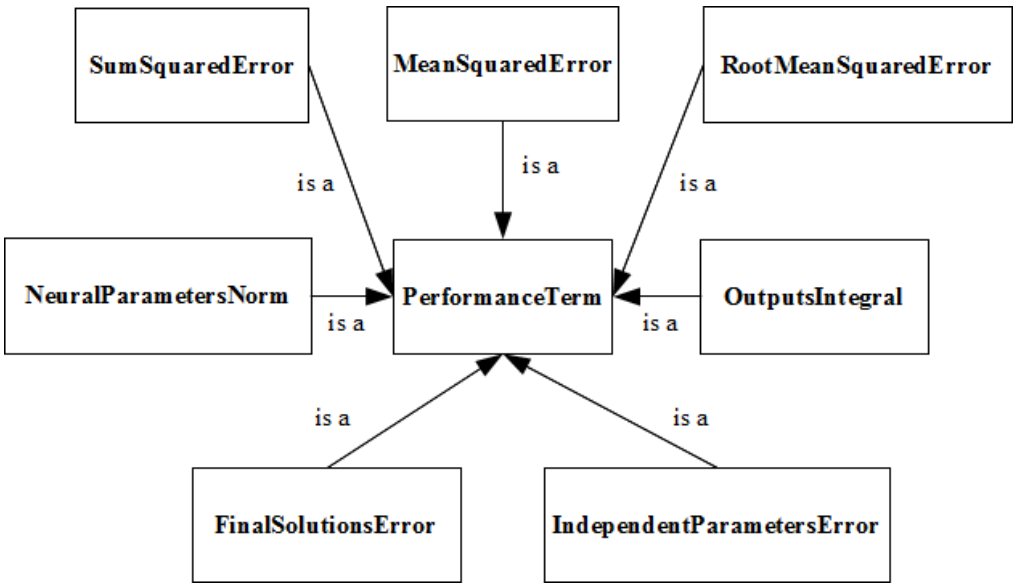

Figure 5.2: Derived classes of the performance term.

Mathematical model

A mathematical model has the following attributes:

- The number of independent variables.
- The number of dependent variables.

It performs the following operations:

- Calculate the solution of the mathematical model.

Performance term

A performance term has the following attributes:

- A relationship to a neural network. In C++ this is implemented as a pointer to a neural network object.
- An relationship to a data set.
- An relationship to a mathematical model.

It performs the following operations:

- Calculate the evaluation of the performance term.
- Calculate the derivatives of the performance term with respect to the neural network parameters.

Performance functional

A performance functional for a neural network has the following attributes:

- A relationship to a neural network. In C++ this is implemented as a pointer to a neural network object.
- An objective performance term.
- A regularization performance term.
- A constraints performance term.

It performs the following operations:

- Calculate the performance of a neural network.
- Calculate the derivatives of the performance with respect to the parameters.

5.3 PerformanceFunctional classes

`OpenNN` implements the PerformanceFunctional concrete class. This class manages different objective, regularization and constraints terms in order to construct a performance functional suitable for our problem.

Members

The PerformanceFunctional class contains:

- The type of objective term.
- The type of regularization term.
- The type of constraints term.
- A pointer to an objective performance term.
- A pointer to a regularization performance term.
- A pointer to a constraints performance term.

All that members are declared as private, and they can only be used with their corresponding get or set methods.

Constructors

As it has been said, the choice of the performance functional depends on the particular application. A default performance functional is not associated to a neural network, it is not measured on a data set or a mathematical model,

```
PerformanceFunctional pf;
```

The default objective functional in the performance functional above is a normalized squared error. This is very used in function regression, pattern recognition and time series prediction.

That performance functional does not contain any regularization or constraints terms by default. Also, it does not have numerical differentiation utilities.

The following sentence constructs a performance functional associated to a neural network and to be measured on a data set.

```
NeuralNetwork nn(1, 1);

DataSet ds(1, 1, 1);
ds.initialize_data(0.0);

PerformanceFunctional pf(&nn, &ds);
```

As before the default objective functional is the normalized squared error.

Methods

The calculate_performance method calculates the performance of some neural network. It is called as follows,

```
double performance = pf.calculate_performance();
```

Note that the evaluation of the performance functional is the sum of the objective, regularization and constraints terms.

The calculate_gradient method calculates the partial derivatives of the performance with respect to the neural network parameters.

```
PerformanceFunctional pf;

Vector<double> gradient = pf.calculate_gradient();
```

As before, the gradient of the performance functional is the sum of the objective, the regularization and the constraints gradients.

Note that, most of the times, an analytical solution for the gradient is achieved. An example is the normalized squared error. On the other hand, some applications might need numerical differentiation. An example is the outputs integrals performance term.

Similarly, the Hessian matrix can be computed using the calculate_Hessian method,

```
Matrix<double> Hessian = pf.calculate_Hessian();
```

The Hessian of the objective functional is also the sum of the objective, regularization and constraints matrices of second derivatives.

If the user wants another objective functional than the default, he can writte

```
pf.construct_objective_term('MEAN_SQUARED_ERROR');
```

The above sets the objective functional to be the mean squared error.

The performance functional is not regularized by default. To change that, the following can be used

```
pf.construct_regularization_term('NEURAL_PARAMETERS_NORM');
```

The above sets the default regularization method to be the multilayer perceptron parameters norm. Also, it sets the weight for that regularization term.

Finally, the performance functional does not include a constraints term by default. The use of constrains might be difficult, so the interested reader is referred to look at the examples included in OpenNN.

XML formats

Sum squared error

The XML format of a sum squared error is as follows.

```
<SumSquaredError>
   <Display>boolean</Display>
</SumSquaredError>
```

Mean squared error

The XML format of the mean squared error is very similar to that of the sum squared error.

```
<MeanSquaredError>
   <Display>boolean</Display>
</MeanSquaredError>
```

Root mean squared error

The XML format of the root mean squared error is very similar to that of the sum squared error.

```
<RootMeanSquaredError>
   <Display>boolean</Display>
</RootMeanSquaredError>
```

Normalized squared error

The XML format of the normalized squared error is very similar to that of the sum squared error.

```
<NormalizedSquaredError>
   <Display>boolean</Display>
</NormalizedSquaredError>
```

Minkowski squared error

The XML format of the root mean squared error is as follows.

```
<MinkowskiError>
   <MinkowskiParameter>real</MinkowskiParameter>
   <Display>boolean</Display>
</MinkowskiError>
```

Cross entropy error

The XML format of the cross entropy error is very similar to that of the sum squared error.

```
<CrossEntropyError>
    <Display>boolean</Display>
</CrossEntropyError>
```

Neural parameters norm

The XML format of the neural parameters norm performance term is as follows.

```
<NeuralParametersNorm>
    <NeuralParametersNormWeight>real</NeuralParametersNormWeight>
    <Display>boolean</Display>
</NeuralParametersNorm>
```

Outputs integrals

The XML format of the outputs integrals performance term is as follows.

```
<OutputsIntegrals>
    <NumericalIntegration>
    numerical_integration_element
    </NumericalIntegration>
    <OutputsIntegralsWeights>real_vector</OutputsIntegralsWeights>
    <Display>boolean</Display>
</OutputsIntegrals>
```

Final solutions error

The XML format of the final solutions error performance term is as follows.

```
<FinalSolutionsError>
    <NumericalDifferentiation>
    numerical_differentiation_element
    </NumericalDifferentiation>
    <TargetFinalSolutions>real_vector</TargetFinalSolutions>
    <FinalSolutionsErrorsWeights>real_vector</FinalSolutionsErrorsWeights>
    <Display>boolean</Display>
</FinalSolutionsError>
```

Solutions error

The XML format of the solutions error performance term is as follows.

```
<FinalSolutionsError>
    <NumericalDifferentiation>
    numerical_differentiation_element
    </NumericalDifferentiation>
    <SolutionsErrorMethod>string</SolutionsErrorMethod>
    <SolutionsErrorsWeights>real_vector</SolutionsErrorsWeights>
    <Display>boolean</Display>
</FinalSolutionsError>
```

Independent parameters error

The XML format of the independent parameters error performance term is as follows.

```
<IndependentParametersError>
   <NumericalDifferentiation>
   numerical_differentiation_element
   </NumericalDifferentiation>
   <TargetIndependentParameters>real_vector</TargetIndependentParameters>
   <IndependentParametersErrorsWeights>real_vector</IndependentParametersErrorsWeights>
   <Display>boolean</Display>
</IndependentParametersError>
```

Inverse sum squared error

The XML format of the independent parameters error performance term is as follows.

```
<InverseSumSquaredError>
   <NumericalDifferentiation>
   numerical_differentiation_element
   </NumericalDifferentiation>
   <Display>boolean</Display>
</InverseSumSquaredError>
```

Performance functional

The XML format of a complete performance functional object is as follows.

```
<PerformanceFunctional>
   <ObjectiveTermType>string</ObjectiveTermType>
   <RegularizationTermType>string</RegularizationTermType>
   <ConstraintsTermType>string</ConstraintsTermType>
   <ObjectiveTermFlag>boolean</ObjectiveTermFlag>
   <RegularizationTermFlag>boolean</RegularizationTermFlag>
   <ConstraintsTermFlag>boolean</ConstraintsTermFlag>
   <ObjectiveTerm>
   objective_term_element
   </ObjectiveTerm>
   <RegularizationTerm>
   regularization_term_element
   </RegularizationTerm>
   <ConstraintsTerm>
   constraints_term_element
   </ConstraintsTerm>
   <Display>boolean</Display>
</PerformanceFunctional>
```

Chapter 6

Training strategy

The procedure used to carry out the learning process in a neural network is called the training strategy. A training strategy might be composed of different training algorithms.

6.1 Basic theory

As we saw in the previous chapter the performance functional has a performance function associated. The performance function for a neural network can be visualized as a hypersurface, with the parameters as coordinates, see Figure 6.1.

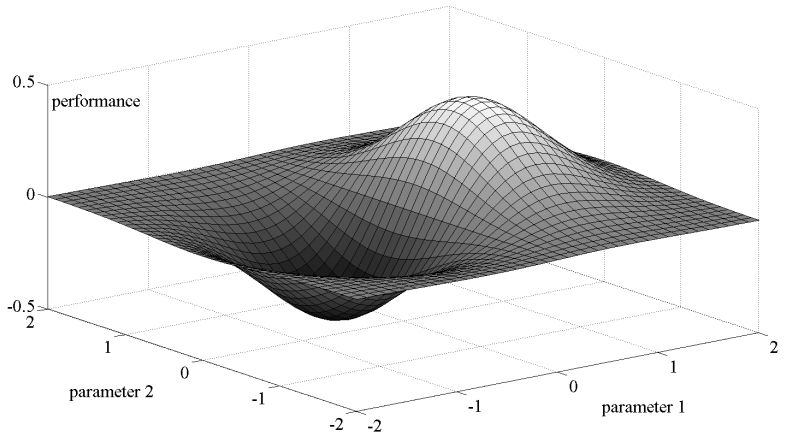

Figure 6.1: Geometrical representation of the performance function.

The minimum or maximum value of the performance functional is achieved for a vector of parameters at which the performance function takes on a minimum or maximum value. Therefore, the learning problem for neural networks,

formulated as a variational problem, can be reduced to a function optimization problem [24].

In this sense, a variational formulation for neural networks provides a direct method for solving variational problems. The universal approximation properties for the multilayer perceptron cause neural computation to be a very appropriate paradigm for the solution of these problems.

One-dimensional optimization

Although the performance function is multidimensional, one-dimensional optimization methods are of great importance. Indeed, one-dimensional optimization algorithms are very often used inside multidimensional optimization algorithms.

A function is said to have a relative or local minimum at some point if the function is always greater within some neighbourhood of that point. Similarly, a point is called a relative or local maximum if the function is always lesser within some neighbourhood of that point.

The function is said to have a global or absolute minimum at some point if the function is always greater within the whole domain. Similarly, a point will be a global maximum if the function is always greater within the whole domain. Finding a global optimum is, in general, a very difficult problem [39].

On the other hand, the tasks of maximization and minimization are trivially related to each other, since maximization of a function is equivalent to minimization of its negative, and vice versa.

In this regard, a one-dimensional optimization problem is one in which the argument which minimizes the performance function is to be found.

The necessary condition states that if the directional performance function has a relative optimum and if the derivative exists as a finite number. The condition for the optimum to be a minimum is that the second derivative is greater than zero, and vice versa.

The most elementary approach for one-dimensional optimization problems is to use a fixed step size or training rate. More sophisticated algorithms which are are widely used are the golden section method and the Brent's method. Both of the two later algortims begin by bracketing a minimum.

The golden section method brackets that minimum until the distance between the two outer points in the bracket is less than a defined tolerance [30].

The Brent's method performs a parabolic interpolation until the distance between the two outer points defining the parabola is less than a tolerance [30].

Multi-dimensional optimization

As it was shown in Chapter 5, the learning problem for neural networks is reduced to the searching for a parameter vector at which the performance function takes a maximum or a minimum value.

The concepts of relative or local and absolute or global optima for the multidimensional case apply in the same way as for the one-dimensional case. The tasks of maximization and minimization are also trivially related here.

The necessary condition states that if the neural network is at a minimum of the performance function, then the gradient is the zero vector.

The performance function is, in general, a non linear function of the parameters. As a consequence, it is not possible to find closed training algorithms for the minima. Instead, we consider a search through the parameter space consisting of a succession of steps, or epochs.

At each epoch, the performance will increase by adjusting the neural network parameters. The change of parameters between two epochs is called the parameters increment.

In this way, to train a neural network we start with some parameters vector (often chosen at random) and we generate a sequence of parameter vectors, so that the performance function is reduced at each iteration of the algorithm. The change of performance between two epochs is called the performance improvement.

The training algorithm stops when a specified condition is satisfied. Some stopping criteria commonly used are [9]:

1. The parameters increment norm is less than a minimum value.
2. The performance improvement in one epoch is less than a set value.
3. Performance has been minimized to a goal value.
4. The norm of the performance function gradient falls below a goal.
5. A maximum number of epochs is reached.
6. A maximum amount of computing time has been exceeded.

A stopping criterium of different nature is early stopping. This method is used in ill-possed problems in order to control the effective complexity of the neural network. Early stopping is a very common practice in neural networks and often produces good solutions to ill-possed problems.

Figure 6.2 is a state diagram of the training procedure, showing states and transitions in the training process of a neural network.

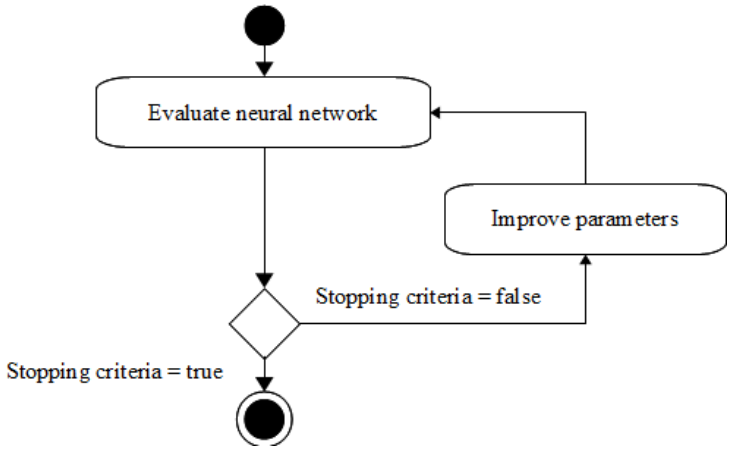

Figure 6.2: Training process.

The training process is determined by the way in which the adjustment of the parameters in the neural network takes place. There are many different training algorithms, which have a variety of different computation and storage requirements. Moreover, there is not a training algorithm best suited to all locations [39].

Training algorithms might require information from the performance function only, the gradient vector of the performance function or the Hessian matrix of the performance function [30]. These methods, in turn, can perform either global or local optimization.

Zero-order training algorithms make use of the performance function only. The most significant zero-order training algorithms are stochastic, which involve randomness in the optimization process. Examples of these are random search and evolutionary algorithms [18] [14] or particle swarm optimization [21], which are global optimization methods .

First-order training algorithms use the performance function and its gradient vector [3]. Examples of these are gradient descent methods, conjugate gradient methods, scaled conjugate gradient methods [28] or quasi-Newton methods. Gradient descent, conjugate gradient, scaled conjugate gradient and quasi-Newton methods are local optimization methods [25].

Second-order training algorithms make use of the performance function, its gradient vector and its Hessian matrix [3]. Examples for second-order methods are Newton's method and the Levenberg-Marquardt algorithm [19]. Both of them are local optimization methods [25].

Training strategy

Most of the times, application of a single training algorithm is enough to properly train a neural network. The quasi-Newton method is in general a good choice, since it provides good training times and deals successfully with most of the performance functions. The Levenberg-Marquardt algorithm could be also recommended for small and medium-sized data modelling problems.

However, some applications might need more training effort. In that cases we can combine different algorithms in order to do our best. In problems defined by mathematical models, with constraints, etc. a single training algorithm might fail.

Therefore, for difficult problems, we can try two use two or three different training algorithms. A general strategy consists on applying three different training algorithms:

1. Initialization training algorithm.

2. Main training algorithm.

3. Refinement training algorithm.

The initialization training algorithm is used to bring the neural network to a stable region of the performance function. Near the optimum, the performance function usually behaves better than far away. Zero order training algorithms, such as random search or the evolutionary algorithm might good for this initialization process. Indeed, they are very robust algorithms.

The main training algorithm does most of the job. The training strategy relies on them. First order training algorithms, such as the quasi-Newton method are a good choce here.

Finally, a refinement training algorithm can be used when a big accuracy is required. Second orden training algorithms, such as the Newton-method, require the most exact information of the performance function. Therefore they can perform better for refinement.

6.2 Software model

The `OpenNN` training strategy is composed by three algorithms: an initialization, a main and a refinement training algorithms. In this section we study the software model of the TrainingStrategy class.

Classes

In order to construct a software model for the training strategy, a few things need to be taken into account.

As we have seen, a training strategy is composed of three different training algorithms. On the other hand, some training algoritms use one-dimensional optimization for finding the optimal training rate. Therefore, the most important classes in the training strategy are:

Training algorithm The class TrainingAlgorithm represents a single training algorithm.

Training rate algorithm The class TrainingRateAlgorithm represents the one-dimensional optimization algorithm for the training rate.

Training strategy The class TrainingStrategy represents a complete training strategy, and it is composed of initialization, main and refinement training algorithms.

Associations

The associations among the concepts described above are the following:

Training algorithm - Training rate algorithm A training algorithm might require a training rate algorithm during the optimization process.

Training strategy - Training algorithm A training strategy might be composed of different training algorithms.

Figure 6.3 shows the UML class diagram for the training strategy with some of the derived classes included.

Derived classes

The next task is then to establish which classes are abstract and to derive the necessary concrete classes to be added. Let us then examine the classes we have so far:

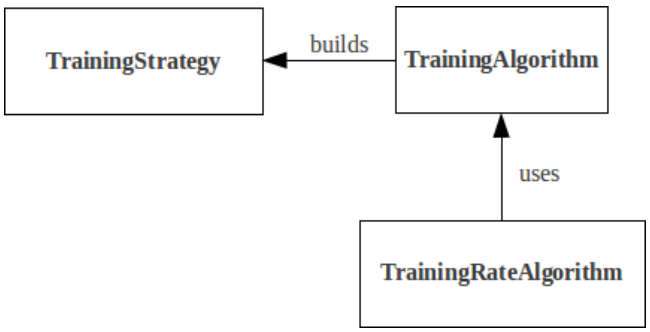

Figure 6.3: Association diagram for the training strategy.

Training algorithm The class TrainingAlgorithm is abstract, because it does not represent a training algorithm for a performance function of a neural network.

The concrete training algorithm classes included with `OpenNN` are RandomSearch, GradientDescent, NewtonMethod, ConjugateGradient, QuasiNewtonMethod and EvolutionaryAlgorithm.

Training rate algorithm This class is concrete, because it has several one-dimensional methods implemented.

Training strategy The class TrainingStrategy is concrete, since it is composed by concrete initialization, main and refinement training algorithms.

Figure 6.4 shows the UML class diagram for the training strategy with some of the derived classes included.

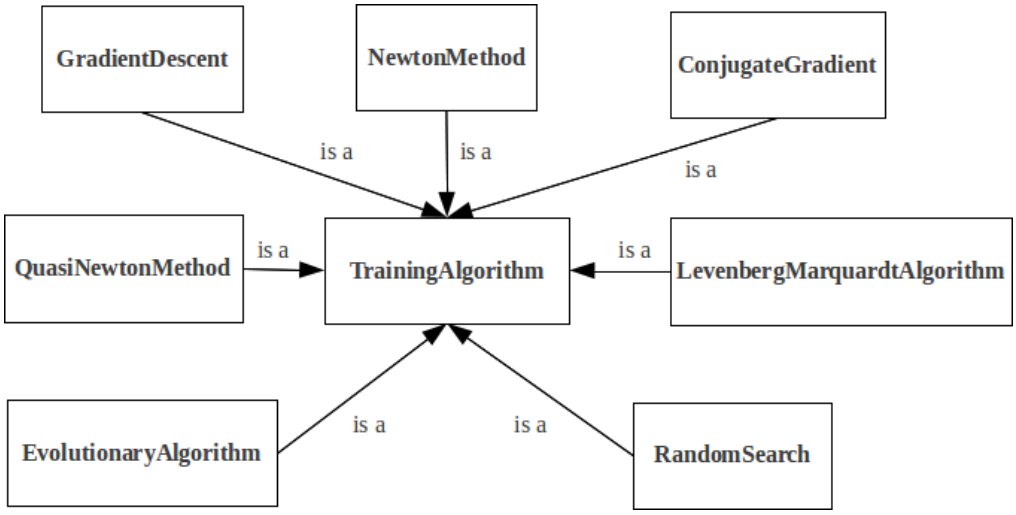

Figure 6.4: Derived classes related to the training strategy.

Attributes and operations

Training algorithm A training algorithm has the following attributes:

- A relationship to a performance functional. In C++ this is implemented as a pointer to a performance functional object.
- A set of training operators.
- A set of training parameters.
- A set of stopping criteria.

It performs the following operations:

- Train a neural network.

Training rate algorithm A training rate algorithm has the following attributes:

- A relationship to a performance functional.
- The training rate algorithm method to use.
- A set of parameters.

It performs the following operations:

- Calculate the training rate.

Training strategy A training strategy has the following attributes:

- A relationship to a performance functional.
- A pointer to an initialization training algorithm.
- A pointer to a main training algorithm.
- A pointer to a refinement training algorithm.
- A flag for using the initialization training algorithm.
- A flag for using the main training algorithm.
- A flag for using the refinement training algorithm.

It performs the following operations:

- Calculate the training rate.

6.3 TrainingStrategy classes

`OpenNN` includes the class TrainingStrategy to represent the concept of training strategy.

Members

The training strategy class contains:

- A pointer to a performance functional object.
- A pointer to an initialization training algorithm.
- A pointer to a main training algorithm.

- A pointer to a refinement training algorithm.

- The type of initialization training algorithm.

- The type of main training algorithm.

- The type of refinement training algorithm.

- A flag for using the initialization training algorithm.

- A flag for using the main training algorithm.

- A flag for using the refinement training algorithm.

All members are private, and must be accessed or modified by means of get and set methods, respectively.

Constructors

To construct a training strategy object associated to a performance functional object we do the following:

```
TrainingStrategy ts(&pf);
```

where pf is some performance functional object.

Methods

The default training strategy consists on a main training algorithm of the quasi-Newton method type. The next sentence sets a different training strategy.

```
RandomSearch* rsp = new RandomSearch(&pf);
rsp->set_epochs_number(10);
ts.set_initialization_training_algorithm(rsp);


GradientDescent* gdp = new GradientDescent(&pf);
gdp->set_epochs_number(25);
ts.set_main_training_algorithm(gdp);
```

The most important method of a training strategy is called perform_training. The use is as follows:

```
ts.perform_training();
```

where ts is some training strategy object.

We can save the above object to a XML file.

```
ts.save("training_strategy.xml");
```

XML formats

In this section we list the XML formats of the training strategy classes in `OpenNN`.

Training rate algorithm

Some training algorithms contain a training rate object inside. The format of this object is listed below.

```
<TrainingRateAlgorithm>
   <TrainingRateMethod>string</TrainingRateMethod>
   <BracketingFactor>real</BracketingFactor>
   <FirstTrainingRate>real</FirstTrainingRate>
   <Display>boolean</Display>
</TrainingRateAlgorithm>
```

Gradient descent

The file format of this class is listed below.

```
<GradientDescent>
   <TrainingRate>
   training_rate_element
   </TrainingRate>
   <WarningTrainingRate>real</WarningTrainingRate>
   <ErrorTrainingRate>real<ErrorTrainingRate>

   <MinimumParametersIncrementNorm>real</MinimumParametersIncrementNorm>
   <EvaluationGoal>real</EvaluationGoal>
   <MinimumEvaluationImprovement>real</MinimumEvaluationImprovement>
   <GradientNormGoal>real</GradientNormGoal>
   <MaximumEpochsNumber>integer</MaximumEpochsNumber>
   <MaximumTime>real</MaximumTime>

   <ReserveElapsedTimeHistory>boolean</ReserveElapsedTimeHistory>
   <ReserveParametersHistory>boolean</ReserveParametersHistory>
   <ReserveParametersNormHistory>boolean</ReserveParametersNormHistory>
   <ReservePerformanceHistory>boolean</ReserveEvaluationHistory>
   <ReserveValidationErrorHistory>boolean</ReserveValidationErrorHistory>
   <ReserveGradientHistory>boolean</ReserveGradientHistory>
   <ReserveGradientNormHistory>boolean</ReserveGradientNormHistory>
   <ReserveTrainingDirectionHistory>boolean</ReserveTrainingDirectionHistory>
   <ReserveTrainingDirectionNormHistory>boolean</ReserveTrainingDirectionNormHistory>
   <ReserveTrainingRateHistory>boolean</ReserveTrainingRateHistory>

   <WarningParametersNorm>real</WarningParametersNorm>
   <WarningGradientNorm>real</WarningGradientNorm>
   <Display>boolean</Display>
   <DisplayPeriod>integer</DisplayPeriod>
</GradientDescent>
```

Newton method

The file format of this class is listed below.

```
<NewtonMethod>
   <TrainingRateMethod>
   training_rate_method
   </TrainingRateMethod>

   <WarningParametersNorm>real</WarningParametersNorm>
   <WarningGradientNorm>real</WarningGradientNorm>
   <WarningTrainingRate>real</WarningTrainingRate>

   <ErrorParametersNorm>real</ErrorParametersNorm>
```

```
    <ErrorGradientNorm>real</ErrorGradientNorm>
    <ErrorTrainingRate>real</ErrorTrainingRate>

    <MinimumParametersIncrementNorm>real</MinimumParametersIncrementNorm>

    <MinimumEvaluationImprovement>real</MinimumEvaluationImprovement>
    <EvaluationGoal>real</EvaluationGoal>

    <GradientNormGoal>real</GradientNormGoal>

    <MaximumEpochsNumber>integer</MaximumEpochsNumber>

    <MaximumTime>real</MaximumTime>

    <ReserveParametersHistory>boolean</ReserveParametersHistory>
    <ReserveParametersNormHistory>boolean</ReserveParametersNormHistory>

    <ReserveEvaluationHistory>boolean</ReserveEvaluationHistory>
    <ReserveGeneralizationErrorHistory>boolean</ReserveGeneralizationErrorHistory>
    <ReserveGradientHistory>boolean</ReserveGradientHistory>
    <ReserveGradientNormHistory>boolean</ReserveGradientNormHistory>
    <ReserveInverseHessianHistory>boolean</ReserveInverseHessianHistory>

    <ReserveTrainingDirectionHistory>boolean</ReserveTrainingDirectionHistory>
    <ReserveTrainingDirectionNormHistory>boolean</ReserveTrainingDirectionNormHistory>
    <ReserveTrainingRateHistory>boolean</ReserveTrainingRateHistory>
    <ReserveElapsedTimeHistory>boolean</ReserveElapsedTimeHistory>

    <Display>boolean</Display>
    <DisplayPeriod>integer</DisplayPeriod>
</NewtonMethod>
```

Conjugate gradient

The file format of the conjugate gradien object is as follows:

```
<ConjugateGradient>
    <TrainingDirectionMethod>string</TrainingDirectionMethod>

    <TrainingRateAlgorithm>
    training_rate_algorithm_element
    </TrainingRateAlgorithm>

    <WarningParametersNorm>real</WarningParametersNorm>
    <WarningGradientNorm>real</WarningGradientNorm>
    <WarningTrainingRate>real</WarningTrainingRate>

    <ErrorParametersNorm>real</ErrorParametersNorm>
    <ErrorGradientNorm>real</ErrorGradientNorm>
    <ErrorTrainingRate>real</ErrorTrainingRate>

    <MinimumParametersIncrementNorm>real</MinimumParametersIncrementNorm>

    <MinimumEvaluationImprovement>real</MinimumEvaluationImprovement>
    <EvaluationGoal>real</EvaluationGoal>
    <GradientNormGoal>real</GradientNormGoal>

    <MaximumEpochsNumber>integer</MaximumEpochsNumber>
    <MaximumTime>real</MaximumTime>

    <ReserveParametersHistory>boolean</ReserveParametersHistory>
    <ReserveParametersNormHistory>boolean</ReserveParametersNormHistory>
```

```
    <ReserveEvaluationHistory>boolean</ReserveEvaluationHistory>
    <ReserveGeneralizationEvaluationHistory>boolean</ReserveGeneralizationEvaluationHistory>
    <ReserveGradientHistory>boolean</ReserveGradientHistory>
    <ReserveGradientNormHistory>boolean</ReserveGradientNormHistory>

    <ReserveTrainingDirectionHistory>boolean</ReserveTrainingDirectionHistory>
    <ReserveTrainingDirectionNormHistory>boolean</ReserveTrainingDirectionNormHistory>
    <ReserveTrainingRateHistory>boolean</ReserveTrainingRateHistory>
    <ReserveElapsedTimeHistory>boolean</ReserveElapsedTimeHistory>

    <DisplayPeriod>integer</DisplayPeriod>
    <Display>boolean</Display>
</ConjugateGradient>
```

Quasi-Newton method XML format

See below for the format of a quasi-Newton method XML-type file in `OpenNN`.

```
<QuasiNewtonMethod version="3.0">

    <TrainingRateAlgorithm>
    training_rate_algorithm_element
    </TrainingRateAlgorithm>

    <WarningParametersNorm>real</WarningParametersNorm>
    <WarningGradientNorm>real</WarningGradientNorm>
    <WarningTrainingRate>real</WarningTrainingRate>

    <ErrorParametersNorm>real</ErrorParametersNorm>
    <ErrorGradientNorm>real</ErrorGradientNorm>
    <ErrorTrainingRate>real</ErrorTrainingRate>

    <MinimumParametersIncrementNorm>real</MinimumParametersIncrementNorm>

    <MinimumEvaluationImprovement>real</MinimumEvaluationImprovement>
    <EvaluationGoal>real</EvaluationGoal>
    <GradientNormGoal>real</GradientNormGoal>

    <MaximumEpochsNumber>integer</MaximumEpochsNumber>
    <MaximumTime>real</MaximumTime>

    <ReserveParametersHistory>boolean</ReserveParametersHistory>
    <ReserveParametersNormHistory>boolean</ReserveParametersNormHistory>

    <ReserveEvaluationHistory>boolean</ReserveEvaluationHistory>
    <ReserveGeneralizationEvaluationHistory>boolean</ReserveGeneralizationEvaluationHistory>
    <ReserveGradientHistory>boolean</ReserveGradientHistory>
    <ReserveGradientNormHistory>boolean</ReserveGradientNormHistory>
    <ReserveInverseHessianHistory>boolean</ReserveInverseHessianHistory>

    <ReserveTrainingDirectionHistory>boolean</ReserveTrainingDirectionHistory>
    <ReserveTrainingDirectionNormHistory>boolean</ReserveTrainingDirectionNormHistory>
    <ReserveTrainingRateHistory>boolean</ReserveTrainingRateHistory>
    <ReserveElapsedTimeHistory>boolean</ReserveElapsedTimeHistory>

    <Display>boolean</Display>

    <DisplayPeriod>integer</DisplayPeriod>
</QuasiNewtonMethod>
```

Levenberg Marquardt algorithm

The file format of this class is listed below.

```
<LevenbergMarquardtAlgorithm>
    <LinearAlgebraicEquations>
    linear_algebraic_equations_element
    </LinearAlgebraicEquations>
    <DampingParameter>real</DampingParameter>
    <MinimumDampingParameter>real</MinimumDampingParameter>
    <MaximumDampingParameter>real</MaximumDampingParameter>
    <DampingParameterFactor>real</DampingParameterFactor>

    <WarningParametersNorm>real</WarningParametersNorm>
    <WarningGradientNorm>real</WarningGradientNorm>

    <ErrorParametersNorm>real</ErrorParametersNorm>
    <ErrorGradientNorm>real</ErrorGradientNorm>

    <MinimumParametersIncrementNorm>real</MinimumParametersIncrementNorm>
    <EvaluationGoal>real</EvaluationGoal>
    <MinimumEvaluationImprovement>real</MinimumEvaluationImprovement>
    <GradientNormGoal>real</GradientNormGoal>
    <MaximumEpochsNumber>integer</MaximumEpochsNumber>
    <MaximumTime>real</MaximumTime>

    <ReserveParametersHistory>boolean</ReserveParametersHistory>
    <ReserveParametersNormHistory>boolean</ReserveParametersNormHistory>

    <ReserveEvaluationHistory>boolean</ReserveEvaluationHistory>
    <ReserveGeneralizationEvaluationHistory>boolean</ReserveGeneralizationEvaluationHisto
    <ReserveGradientHistory>boolean</ReserveGradientHistory>
    <ReserveGradientNormHistory>boolean</ReserveGradientNormHistory>

    <ReserveTrainingDirectionHistory>boolean</ReserveTrainingDirectionHistory>
    <ReserveTrainingDirectionNormHistory>boolean</ReserveTrainingDirectionNormHistory>
    <ReserveTrainingRateHistory>boolean</ReserveTrainingRateHistory>
    <ReserveElapsedTimeHistory>boolean</ReserveElapsedTimeHistory>

    <WarningGradientNorm>real</WarningGradientNorm>
    <Display>boolean</Display>
    <DisplayPeriod>integer</DisplayPeriod>
</LevenbergMarquardtAlgorithm>
```

Random search

The file format of this class is listed below.

```
<RandomSearch>

    <FirstTrainingRate>double</FirstTrainingRate>
    <TrainingRateReductionFactor>double</TrainingRateReductionFactor>
    <TrainingRateReductionPeriod>unsigned int</TrainingRateReductionPeriod>
    <WarningParametersNorm>double</WarningParametersNorm>
    <WarningTrainingRate>double</WarningTrainingRate>
    <ErrorParametersNorm>double</ErrorParametersNorm>
    <ErrorTrainingRate>double</ErrorTrainingRate>

    <MinimumParametersIncrementNorm>double</MinimumParametersIncrementNorm>
    <MinimumPerformanceIncrease>double</MinimumPerformanceIncrease>
    <PerformanceGoal>double</PerformanceGoal>
    <MaximumGeneralizationEvaluationDecreases>unsigned int</MaximumGeneralizationEvaluat
```

```
    <MaximumEpochsNumber>unsigned int</MaximumEpochsNumber>
    <MaximumTime>double</MaximumTime>

    <ReserveParametersHistory>bool</ReserveParametersHistory>
    <ReserveParametersNormHistory>bool</ReserveParametersNormHistory>
    <ReserveEvaluationHistory>bool</ReserveEvaluationHistory>
    <ReserveGeneralizationEvaluationHistory>bool</ReserveGeneralizationEvaluationHistory>
    <ReserveTrainingDirectionHistory>bool</ReserveTrainingDirectionHistory>
    <ReserveTrainingDirectionNormHistory>bool</ReserveTrainingDirectionNormHistory>
    <ReserveTrainingRateHistory>bool</ReserveTrainingRateHistory>
    <ReserveElapsedTimeHistory>bool</ReserveElapsedTimeHistory>

    <DisplayPeriod>unsigned int</DisplayPeriod>
    <Display>bool</Display>
</RandomSearch>
```

Evolutionary algorithm

The next listing shows the format of an evolutionary algorithm data file in `OpenNN`. It is of XML-type.

```
<EvolutionaryAlgorithm>

    Training operators

    <FitnessAssignmentMethod>string</FitnessAssignmentMethod>
    <SelectionMethod>string</SelectionMethod>
    <RecombinationMethod>string</RecombinationMethod>
    <MutationMethod>string</MutationMethod>

    Training parameters

    <PopulationSize>integer</PopulationSize>
    <Elitism>boolean</Elitism>
    <SelectivePressure>real</SelectivePressure>
    <RecombinationSize>real</RecombinationSize>
    <MutationRate>real</MutationRate>
    <MutationRange>real</MutationRange>

    Stopping criteria

    <EvaluationGoal>real</EvaluationGoal>
    <MeanEvaluationGoal>real</MeanEvaluationGoal>
    <StandardDeviationEvaluationGoal>real</StandardDeviationEvaluationGoal>

    <MaximumGenerationsNumber>real</MaximumGenerationsNumber>
    <MaximumTime>real</MaximumTime>

    Training history

    <ReservePopulationHistory>boolean</ReservePopulationHistory>
    <ReserveMeanNormHistory>boolean</ReserveMeanNormHistory>
    <ReserveStandardDeviationNormHistory>boolean</ReserveStandardDeviationNormHistory>
    <ReserveBestNormHistory>boolean</ReserveBestNormHistory>

    <ReserveMeanEvaluationHistory>boolean</ReserveMeanEvaluationHistory>
    <ReserveStandardDeviationEvaluationHistory>boolean</ReserveStandardDeviationEvaluationHistory>
    <ReserveBestEvaluationHistory>boolean</ReserveBestEvaluationHistory>

    <Display>boolean</Display>
    <DisplayPeriod>integer</DisplayPeriod>
```

```
</EvolutionaryAlgorithm>
```

Training strategy

The XML format of a training strategy class is listed below. It might contain up to three training algorithms.

```
<TrainingStrategy>

    <InitializationTrainingAlgorithm>
    initialization_training_algorithm_element
    </InitializationTrainingAlgorithm>
    <MainTrainingAlgorithm>
    main_training_algorithm_element
    </MainTrainingAlgorithm>
    <RefinementTrainingAlgorithm>
    refinement_training_algorithm_element
    </RefinementTrainingAlgorithm>

    <InitializationTrainingAlgorithmFlag>boolean</InitializationTrainingAlgorithmFlag>
    <MainTrainingAlgorithmFlag>boolean</MainTrainingAlgorithmFlag>
    <RefinementTrainingAlgorithmFlag>boolean</RefinementTrainingAlgorithmFlag>

    <Display>boolean</Display>
</TrainingStrategy>
```

Chapter 7

Function regression

Many classes included with `OpenNN` are related to the problem of function regression, since these type of problems are traditional learning tasks for neural networks. The C++ code here include the data set classes, a number of performance terms, the model selection class and some testing methods for function regression.

7.1 Basic theory

Introduction

Here the neural network learns from knowledge represented by a data set consisting of input-target instances. The targets are a specification of what the response to the inputs should be, and are represented as continuous variables.

The basic goal in a function regression problem is to model the conditional distribution of the target variables, conditioned on the input variables [5]. This function is called the regression function.

The formulation of a function regression problem requires:

- A data set.
- A neural network.
- A performance functional.
- A training strategy.
- A model selection algorithm.
- A testing method.

A common feature of most data sets is that the data exhibits an underlying systematic aspect, represented by some function, but is corrupted with random noise.

The central goal is to produce a model which exhibits good generalization, or in other words, one which makes good predictions for new data. The best generalization to new data is obtained when the mapping represents the underlying systematic aspects of the data, rather capturing the specific details (i.e. the noise contribution) of the particular input-target set.

Data set

Table 7.1 shows the format of a data set for function regression. It consists q instances consisting of n input variables and m target variables. All input and targets are real values.

$input_{1,1}$	...	$input_{1,n}$	$target_{1,1}$	...	$target_{1,m}$
$input_{2,1}$	...	$input_{2,n}$	$target_{2,1}$	...	$target_{2,m}$
$\vdots$		$\vdots$	$\vdots$		$\vdots$
$input_{q,1}$	...	$input_{q,n}$	$target_{q,1}$	...	$target_{q,m}$

Table 7.1: Data set for function regression.

When solving function regression problems it is always convenient to split the data set into a training, a generalization and a testing subsets. The size of each subset is up to the designer. Some default values could be to use 60%, 20% and 20% of the instances for training, generalization and testing, respectively.

There are several data splitting methods. Two common approaches are to generate random indices or to specify the required indices for the training, generalization and testing instances.

A simple statistical analysis must be always performed in order to chech for data consistency. Basic statistics of a data set include the mean, standard deviation, minimum and maximum values of input and target variables for the whole data set and the training, generalization and testing subsets. An histogram of each input and target variables should also be plot in order to check the distribution of the available data.

Also, it is a must to scale the data with the data statistics. There are two main data scaling methods, the mean and standard deviation and the minimum and maximum.

The mean and standard deviation method scales the data for mean 0 and standard deviation 1. The minimum and maximum method scales the data for minimum -1 and maximum 1.

Neural network

A neural network is used to represent the regression function. The number of inputs must be equal to the number of inputs in the data set, n, and the number of outputs must be the number of targets, m. This neural network will contain a scaling layer, a multilayer perceptron and an unscaling layer. It might optionally contain a bounding layer. Figure 7.1 shows a general neural network for solving function regression problems.

In general, using a multilayer perceptron with a one hidden layer will be enough. A default value to start with for the size of that layer could be

$$\text{hidden neurons number} = \frac{\text{inputs number} + \text{outputs number}}{2}.$$

Please note that the complexity which is needed depends very much on the problem at hand, and the above equation is just a rule of thumb. However,

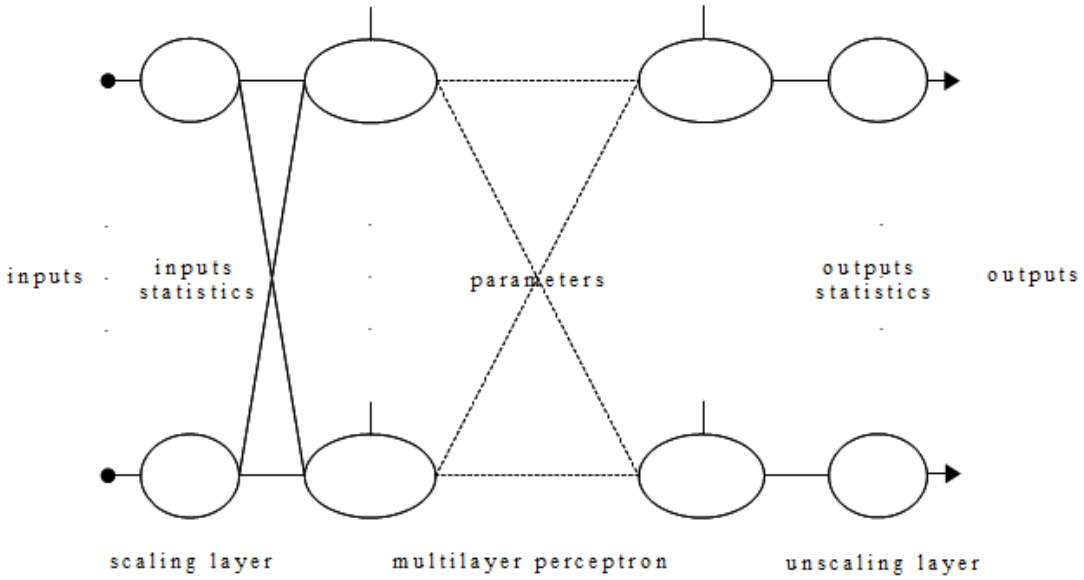

Figure 7.1: Neural network for function regression.

there are standard methods to find the correct complexity of a neural network for function regression problems. The most common is called model selection, which is described later in this section.

The activation functions for the hidden layers and the output layer are also design variables. However, hyperbolic tangent activation function for the hidden layers and linear activation function for the output layer are widely used when solving function regression problems.

Scaling of inputs and unscaling of outputs should not be used in the design phase, since the data set has been scaled already. When moving to a production phase, the inputs scaling and outputs unscaling methods should be coherent with the scaling method used for the data.

The neural network in Figure 7.1 spans a parameterized function space. That parameterized space of functions will be the basis to aproximate the regression function.

Performance functional

The regression function can be evaluated quantitatively by means of a performance functional of the form

$$\text{Performance functional} = \text{objective term} + \text{regularization term}.$$

For function regression problems, the objective term is measures the error between the outputs from the neural network and the targets in the data set.

In function regression problems, regularization is normally used when the number of instances in the data set is small or when the data is noisy. In other situations, regularization might not be necessary.

The solution approach to a function regression problem for is to obtain a neural network which minimizes the performance functional. Note that neural networks represent functions. In that way, the function regression problem is formulated as a variational problem.

Training strategy

The training strategy is entrusted to solve the reduced function optimization problem by minimizing the performance function.

In general, evaluation, gradient and Hessian of the error function can be computed analytically. Zero order training algorithms, such as the evolutionary algorithm, converge extremely slowly and they are not a good choice.

On the other hand, second order training algorithms, such as the Newton's method, need evaluation of the Hessian and are neither a good choice.

In practice, first order algorithms are recommended for solving function regression problems. The Levenberg-Marquardt is a good choice for small and medium sized problems. Due to storage requirements, that algorithm is not recommended for big sized problems, and a quasi-Newton method with BFGS training direction and Brent training rate is preferable.

In order to study the convergence of the optimization process, it is useful to plot the behaviour of some variables related to the multilayer perceptron, the error functional or the training algorithm as a function of the iteration step. Some common training history variables are:

- Parameters norm history.
- Evaluation history.
- Generalization evaluation history.
- Gradient norm history.
- Training direction norm history.
- Training rate history.
- Elapsed time history.

Form all the training history variables, may be the most important one is the evaluation history. Also, it is important to analyze the final values of some variables. The most important training results numbers are:

- Final parameters.
- Final parameters norm.
- Final error.
- Final generalization error.
- Final gradient.
- Final gradient norm.
- Number of iterations.
- Training time.

Model selection

Two frequent problems in function regression are called underfitting and overfitting. The best generalization is achieved by using a model whose complexity is the most appropriate to produce an adequate fit of the data [5]. In this way underfitting is defined as the effect of a generalization error increasing due to a too simple model, whereas overfitting is defined as the effect of a generalization error increasing due to a too complex model.

While underfitting can be prevented by simply increasing the complexity of the neural network, it is more difficult in advance to prevent overfitting.

A common method for preventing overfitting is to use a regularization term in the performance functional expression.

However, the best method for avoiding underfitting and overfitting is to use a neural network that is just large enough to provide an adequate fit. Such a neural network will not have enough power to overfit the data. Unfortunately, it is difficult to know beforehand how large a neural network should be for a specific application. A technique for that is called model selection.

In this technique the data set is divided into a training and a generalization subsets. The training subset is used for training the neural network by means of the training strategy. On the other hand, the error on the generalization subset is monitored during the training process. The generalization error normally decreases during the initial phase of training, as it does the training error.

However, when the neural network begins to overfit the data, the error on the generalization subset typically begins to rise. When the generalization error increases for a specified number of iterations, the training is stopped, and the parameters at the minimum of the generalization error are set to the neural network.

This process is repeated for several network architectures, and the final training and generalization errors are plotted. The optimal network architecture will be that providing minimal generalization error. Figure 7.2 illustrates this generalization analysis.

Testing analysis

The performance of a neural network can be measured to some extent by the performance evaluation on the testing set, but it is useful to investigate the response in more detail. One option is to perform a regression analysis between the network response and the corresponding targets for an independent testing subset.

This analysis leads to 3 parameters for each output variable. The first two parameters, a and b, correspond to the y-intercept and the slope of the best linear regression relating outputs and targets. The third parameter, R^2, is the correlation coefficient between the outputs and the targets.

If we had a perfect fit (outputs exactly equal to targets), the slope would be 1, and the y-intercept would be 0. If the correlation coefficient is equal to 1, then there is perfect correlation between the outputs from the neural network and the targets in the testing subset.

Figure 7.3 illustrates a linear regression analysis.

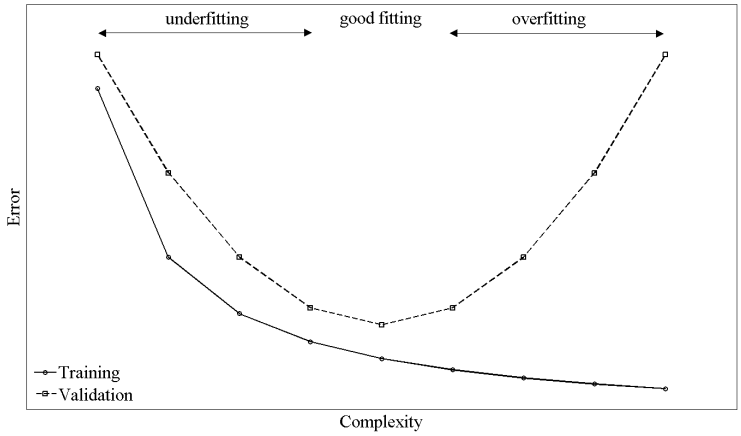

Figure 7.2: Model selection plot for function regression.

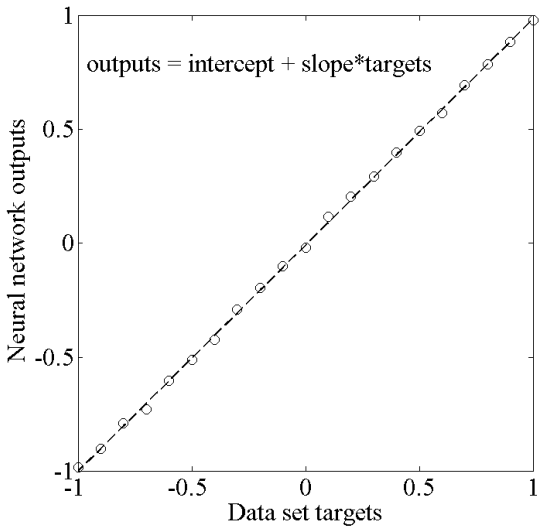

Figure 7.3: Linear regression analysis.

7.2 Examples

Simple function regression

In this example we have a data set with 1 input variable, x, 1 target variable, y, and 101 instances. The aim is to design a neural network that can predict y values for given x values. Figure 7.4 shows this data set.

Here the neural network composed of a multilayer perceptron. The performance functional is composed of an objective term, the normalized squared

error. Finally, the training strategy is composed of a main training algorithm, the quasi-Newton method.

Yacht resistance

Prediction of residuary resistance of sailing yachts at the initial design stage is of a great value for evaluating the ship's performance and for estimating the required propulsive power. Essential inputs include the basic hull dimensions and the boat velocity. Figure 7.5 illustrates this example. That picture has been taken from Wikipedia.

The Delft series are a semi-empirical model developed for that purpose from an extensive collection of full-scale experiments. They are expressed as a set of

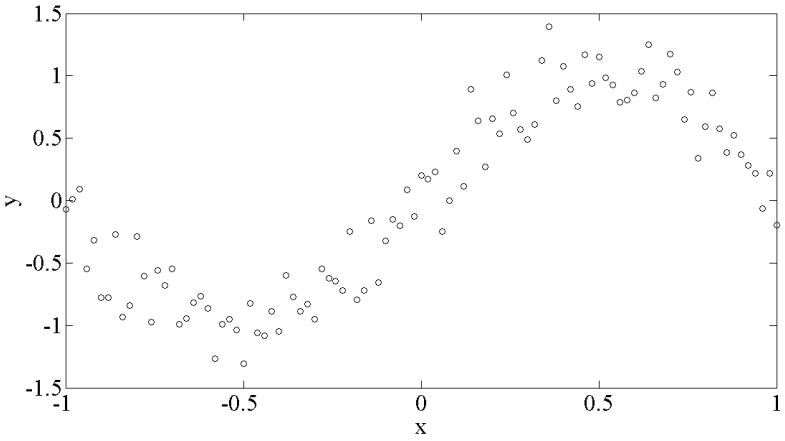

Figure 7.4: Simple function regression data set.

Figure 7.5: Sailing yatchs.

polynomials, and provide a prediction of the residuary resistance per unit weight of displacement, with hull geometry coefficients as variables and for discrete values of the Froude number [16]. The Delft series are widely used in the sailing yacht industry.

The Delft data set comprises 308 full-scale experiments, which were performed at the Delft Ship Hydromechanics Laboratory [16]. These experiments include 22 different hull forms, derived from a parent form closely related to the 'Standfast 43' designed by Frans Maas.

As it has been said, variations concern hull geometry coefficients and the Froude number:

1. Longitudinal position of the center of buoyancy, adimensional.
2. Prismatic coefficient, adimensional.
3. Length-displacement ratio, adimensional.
4. Beam-draught ratio, adimensional.
5. Length-beam ratio, adimensional.
6. Froude number, adimensional.

Also, the measured variable is the residuary resistance per unit weight of displacement:

1. Residuary resistance per unit weight of displacement, adimensional.

In this example, the neural network is composed by a multilayer perceptron with scaling and unscaling layers. The performance functional is composed of just an objective term, the normalized squared error. Finally, the training strategy is only composed of a main training algorithm, the quasi-Newton method.

Airfoil noise

The noise generated by an aircraft is an efficiency and environmental matter for the aerospace industry. An important component of the total airframe noise is the airfoil self-noise, which is due to the interaction between an airfoil blade and the turbulence produce in its own boundary layer and near wake. Figure 7.6 illustrates this example. That picture has been taken from Wikipedia.

The self-noise data set used in this example was processed by NASA in 1989 [6], and so it is referred here to as the NASA data set. It was obtained from a series of aerodynamic and acoustic tests of two and three-dimensional airfoil blade sections conducted in an anechoic wind tunnel.

The NASA data set comprises different size NACA 0012 airfoils at various wind tunnel speeds and angles of attack. The span of the airfoil and the observer position were the same in all of the experiments. The NASA data set contains 1503 instances.

In that way, this problem has the following inputs:

1. Frequency, in Hertzs.
2. Angle of attack, in degrees.

3. Chord length, in meters.

4. Free-stream velocity, in meters per second.

5. Suction side displacement thickness, in meters.

6. Scaled sound pressure level, in decibels.

The only output is:

1. Scaled sound pressure level, in decibels.

In this example, the neural network is composed by a multilayer perceptron with scaling and unscaling layers. The performance functional is composed of just an objective term, the normalized squared error. Finally, the training strategy is only composed of a main training algorithm, the quasi-Newton method.

Figure 7.6: Aircraft noise.

Chapter 8

Pattern recognition

Pattern recognition is also a traditional learning task for neural networks. `OpenNN` classes which are related to the solution of pattern recognition problems include the data set, several performance terms, the model selection and the testing analysis classes.

8.1 Basic theory

Introduction

Another traditional learning task for the neural networks is the pattern recognition (or classification) problem [5]. The task of pattern recognition can be stated as the process whereby a received pattern, characterized by a distinct set of features, is assigned to one of a prescribed number of classes. Here the neural network learns from knowledge represented by a data set consisting of input-target examples. The inputs include a set of features which characterize a pattern. The targets specify the class that each pattern belongs to.

Therefore, in order to solve a pattern recognition problem, the input space must be properly separated into regions, where each region is assigned to a class. A border between two regions is called a decision boundary. The goal in a pattern recognition problem is thus to obtain a neural network function as an approximation of the pattern recognition function.

The formulation of a pattern recognition problem requires:

- A data set.
- A neural network.
- A performance functional.
- A training strategy.
- A model selection algorithm.
- A testing method.

A common feature of most data sets is that the data exhibits an underlying systematic aspect, represented by some function, but is corrupted with random noise.

The central goal is to produce a model which exhibits good generalization, or in other words, one which makes good predictions for new data. The best generalization to new data is obtained when the mapping represents the underlying systematic aspects of the data, rather capturing the specific details (i.e. the noise contribution) of the particular data set.

Data set

In pattern recognition a pattern is represented by a set of attributes, viewed as a multi-dimensional feature vector. They are associated with one of a prescribed number of classes, which are in general of nominal nature.

Table 8.1 shows the format of a data set for pattern recognition. It consists of n input variables and m target variables, comprising q instances.

$input_{1,1}$	$\dots$	$input_{1,n}$	$\rfloor\updownarrow\dashv\int\int_1$
$input_{2,1}$	$\dots$	$input_{2,n}$	$\rfloor\updownarrow\dashv\int\int_2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$input_{q,1}$	$\dots$	$inputx_{q,n}$	$\rfloor\updownarrow\dashv\int\int_q$

Table 8.1: Data set for pattern recognition.

As we have said, the target variables are nominal variables, Therefore, for numerical computation, they need to be given numerical values.

For the case of two classes, the number of target variables will be just one. One class can be simply codified as 0 and the other as 1. For instance, in a medical diagnostic application, we can assign the value 0 to a sane person and 1 to an ill person.

For the case of multiple classes the target data can be codified with a 1-of-m scheme. For instance, consider a food industry which needs to classifying fishes into 3 species. That species can be given the targets $(1, 0, 0)$, $(0, 1, 0)$ and $(0, 0, 1)$, respectively.

Note that some attributes can also be of nominal nature. In this case the same coding scheme as that described above for the targets will be used for the inputs.

A simple statistical analysis must be always performed in order to check for data consistency. Basic statistics of a data set for pattern recognition include the mean, standard deviation, minimum and maximum values of the input variables and the frequency of the different classes.

Also, it is a must to scale the input data. Either the mean and standard deviation or the and the minimum and maximum methods can be used for this purpose. Note that the target data has already proper 0 and 1 values.

It is also convenient to split the data set into a training, a generalization and a testing subsets. The size of each subset is up to the designer, but ratios of 60%, 20% and 20% are quite common. The data can be divided at random or by specifying given indices.

The neural network represents the pattern recognition function. The number of inputs must be equal to the number of inputs in the data set, and the number of outputs must be the number of targets. The basis of this neural network is a multilayer perceptron. It might also include a scaling layer for the inputs, and a probabilistic layer for the outputs. On the other hand, the complexity of the neural network is up to the designer. This complexity will be given by the number and the sizes of the layers in the multilayer perceptron. Figure 8.1 shows a neural network to be used for pattern recognition.

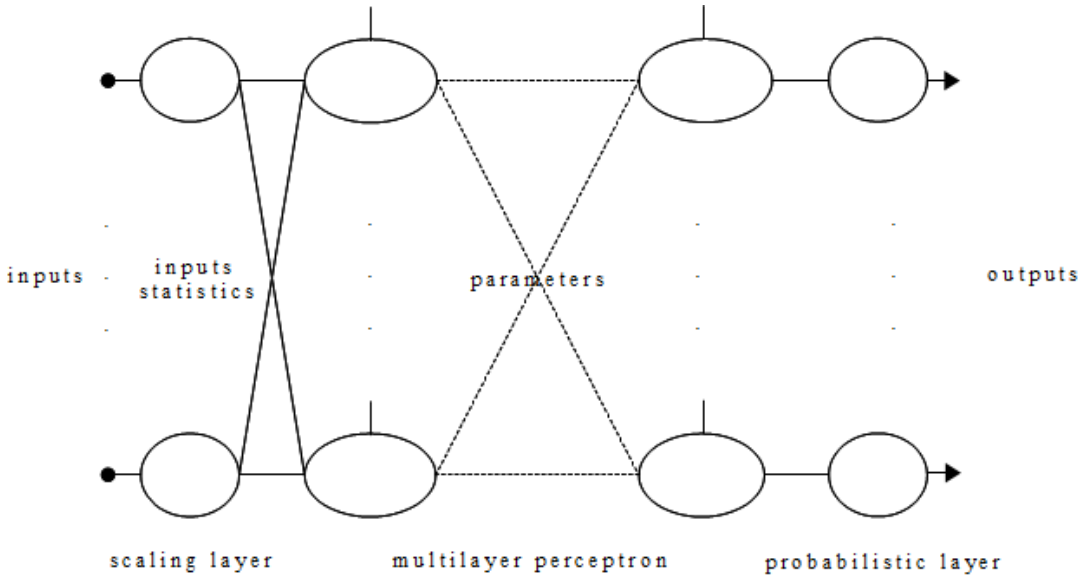

Figure 8.1: Neural network for pattern recognition.

In general, a multilayer perceptron with two layers is enough. A common activation function for the first layer is a sigmoid, such as the hyperbolic tangent or the logistic. Let us consider the activation function which should be used for the output layer.

If the number of classes is two, the number of outputs in the neural network will be one. Therefore, a logistic activation function will interpret the outputs as probabilities, since it lies in the range $(0, 1)$, and no probabilistic layer is needed here.

For multiple classes, we can use a multilayer perceptron with linear output layer. A probabilistic layer with softmax method is then added to form the neural network in Figure 8.1.

Performance functional

In pattern recognition problems, the performance functional evaluates quantitatively the performance of the pattern recognition function against the data set. It is of the form

$$\text{performance functional} = \text{objective term} + \text{regularization term}.$$

Common objective functionals for function regression, such as the sum squared error, the mean squared error, the root mean squared error, the normalized squared error and the Minkowski error are also commonly applied for pattern recognition. However, there are specific problems for this learning task, such as the cross-entropy error.

Training strategy

The training algorithm for pattern recognition problems applies in the same way as for function regression problems.

Model selection

The problems of underfitting and overfitting also might occur when solving a pattern recognition problem with a neural network. Underfitting is explained in terms of a too simple decision boundary which gives poor separation of the training data. On the other hand, overfitting is explained in terms of a too complex decision boundary which achieves good separation of the training data, but exhibits poor generalization.

A method for preventing underfitting and overfitting is to use a network that is just large enough to provide an adequate fit. An alternative approach to obtain good generalization is by using regularization theory.

As in function regression, early stopping can also be performed in pattern recognition to prevent overfitting. However, this technique usually produces underfitting and a more precise model selection analysis is preferible.

Testing analysis

The classification accuracy, error rate, sensitivity, specifity positive likelihood and negative likelihood are parameters for testing the performance of a pattern recognition problem with two classes.

The classification accuracy is the ratio of instances correctly classified,

$$\text{classification accuracy} = \frac{\text{true positives} + \text{true negatives}}{\text{true positives} + \text{true negatives} + \text{false positives} + \text{false negatives}}.$$

The error rate is the ratio of instances misclassified,

$$\text{error rate} = \frac{\text{false positives} + \text{false negatives}}{\text{true positives} + \text{true negatives} + \text{false positives} + \text{false negatives}}.$$

The sensitivity, or true positive rate, is the proportion of alcual positive which are predicted positive,

$$\text{sensitivity} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}.$$

The specifity, or true negative rate, is the proportion of actual negative which are predicted negative,

$$\text{specifity} = \frac{\text{true negatives}}{\text{true negatives} + \text{false positives}}.$$

The positive likelihood is the likelihood that a predicted positive is an actual positive

$$\text{positive likelihood} = \frac{sensitivity}{1 - specifity}.$$

The negative likelihood is the likelihood that a predicted negative is an actual negative

$$\text{negative likelihood} = \frac{specifity}{1 - sensitivity}.$$

Table 8.2 summarizes the binary classification performance variables

Classification accuracy
Error rate
Sensitivity
Specifity
True positive rate
True negative rate

Table 8.2: Binary classification performance variables.

In the confusion matrix the rows represent the target classes and the columns the output classes for a testing target data set. The diagonal cells in each table show the number of cases that were correctly classified, and the off-diagonal cells show the misclassified cases.

For the case of two classes the confusion matrix takes the form

$$\mathbf{C} = \begin{pmatrix} \text{true positives} & \text{false positives} \\ \text{false negatives} & \text{true negatives} \end{pmatrix}.$$

8.2 Examples

Simple pattern recognition

This is an academic example defined by a data set with 100 instances, 2 inputs, or attributes, and 1 target. The target variable represents two classes (0 and 1). The aim is to design a neural network that can predict the correct class for given attribute values. Figure 8.2 shows this data set.

Iris plant

This is perhaps the best known data set to be found in the pattern recognition literature. It contains 3 classes of 50 instances each, where each class refers to a type of iris plant. One class is linearly separable from the other two; the latter are not linearly separable from each other. Figure 8.3 illustrates this example. That picture is has been taken from Wikipedia.

The input variables are:

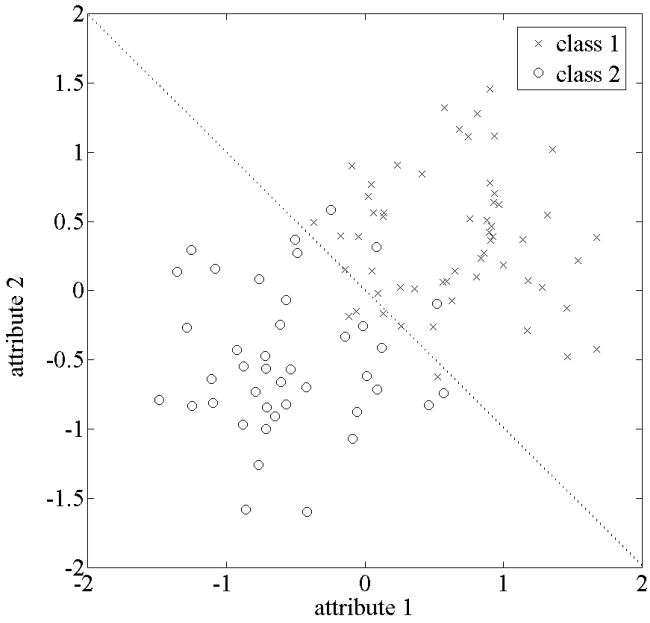

Figure 8.2: Data set for the simple pattern recognition example.

Figure 8.3: Iris versicolor.

1. Sepal length, in centimeters.
2. Sepal width, in centimeters.
3. Petal length, in centimeters.
4. Petal width, in centimeters.
5. Class -iris setosa, iris versicolour or iris virginica.

The predicted class is the class of iris plant:

1. Iris setosa (true or false).
2. Iris versicolour (true or false).
3. Iris virginica (true or false).

More information on this problem can be found in [15].

Pima indians diabetes

Pima Indians of Arizona have the population with the highest rate of diabetics in the world. It has been estimated that around 50% of adults suffer from this disease. The aim of this pattern recognition problem is to predict whether an individual of Pima Indian heritage has diabetes from personal characteristics and physical measurements. Figure 8.4 is a blood glucose testing device, showed here to illustrate this example. That picture has been taken from Wikipedia.

Figure 8.4: Blood glucose testing.

The data is taken from the UCI Machine Learning Repository [15]. The number of samples in the data set is 768. The number of input variables for each sample is 8. All input variables are numeric-valued, and represent personal characteristics and physical measurements of an individual. The number of target variables is 1, and represents the absence or presence of diabetes in an individual. Table 8.3 summarizes the data set information, while tables 8.4 and 8.5 depict the input and target variables information, respectively.

Number of instances:	768
Number of input variables:	8
Number of target variables:	1

Table 8.3: Data set information.

1.	Number of times pregnant.
2.	Plasma glucose concentration a 2 hours in an oral glucose tolerance test.
3.	Diastolic blood pressure ($mmHg$).
4.	Triceps skin fold thickness (mm).
5.	2-Hour serum insulin (muU/ml).
6.	Body mass index (weight in kg/(height in m)^2).
7.	Diabetes pedigree function.
8.	Age (years).

Table 8.4: Input variables information.

1.	Absence or presence of diabetes (0 or 1).

Table 8.5: Target variables information.

Chapter 9

Optimal control

Optimal control is also a learning tasks for neural networks. `OpenNN` classes which are related to the solution of optimal control problems include the mathematical model and several performance term classes.

9.1 Basic theory

Optimal control -which is playing an increasingly important role in the design of modern systems- has as its aim the optimization, in some defined sense, of physical processes. More specifically, the objective of optimal control is to determine the control signals that will cause a process to satisfy the physical constraints and at the same time minimize or maximize some performance criterion [22].

The formulation of an optimal control problem requires:

- A mathematical model.
- A neural network.
- A performance functional.
- A training strategys.

Mathematical model

The model of a process is a mathematical description that adequately predicts the response of the physical system to all anticipated inputs.

A mathematical model (or state equation) contains state variables and control variables. Mathematical models can be expressed as all algebraic, ordinary differential and partial differential equations. However, many optimal control problems in the literature are based on mathematical models described by a system of ordinary differential equations together with their respective initial conditions, representing a dynamical model of the system. Integration here is usually performed with the Runge-Kutta-Fehlberg method.

Neural network

A neural network is used to represent the control variables. The number of inputs is usually one, which represents the time, and the number of outputs is normally small, representing the control variables. Although the number of hidden layers and the sizes of each are design variables, that is not a critical issue in optimal control. Indeed, this class of problems are regarded as being well-possed, and a sufficient complexity for the function space selected is generally enough.

Figure 9.1 shows a neural network template for solving optimal control problems.

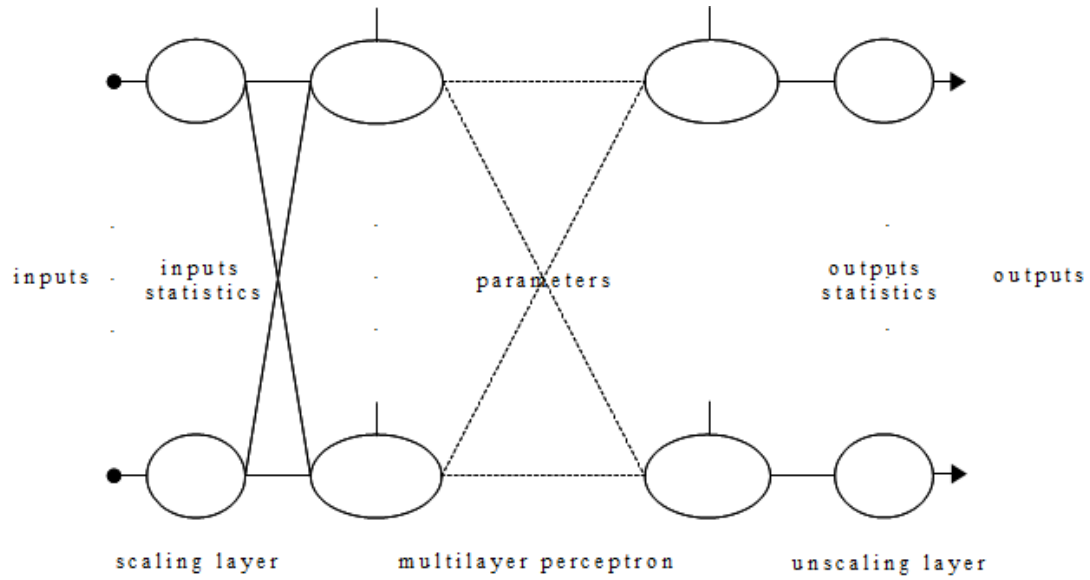

Figure 9.1: Neural network for optimal control.

An optimal control problem might be specified by a set of constraints on the control variables. Two important types of control constraints are boundary conditions and lower and upper bounds.

If some outputs are specified for given inputs, then the problem is said to include boundary conditions. On the other hand, if some control variables are restricted to fall in some interval, then the problem is said to have lower and upper bounds.

Also, some optimal control problems need a neural network with associated independent parameters. The most common are those with free final time.

Performance functional

The performance functional of an optimal control problem always includes an objective term. It might also include a regularization and a constraints terms,

Performance functional = objective term + regularization term + constraints term.

In certain cases the problem statement might clearly indicate which objective criterion is to be selected, whereas in other cases that selection is a subjective matter [22].

The regularization term makes the control variables to have smoother shapes.

State constraints are conditions that the physical system must satisfy. This type of constraints vary according to the problem at hand.

In this way, a control which satisfies all the control and state constraints is called an admissible control [22].

Similarly, a state which satisfies the state constraints is called an admissible state [22].

An optimal control is defined as one that minimizes or maximizes the performance criterion, and the corresponding state is called an optimal state. In this way, the problem of optimal control is formulated as a variational problem [22].

In general, the performance function, cannot be evaluated analytically. This makes that the gradient vector and the Hessian matrix can neither be computed analytically, and numerical differentiation must be used.

Training strategy

We have seen that the performance functional for optimal control problems might contain up to three terms: objective, regularization and constraints. On the other hand, in most of the cases, it cannot be computed analitycally. That makes that a single training algorithm might not fully converge if the solution is far away from the optimal one.

Therefore, when solving optimal control problems, it is recommended to use an initialization training algorithm before the main training process. The form of the training strategy is therefore as follows:

Training strategy: initialization training algorithm, main training algorithm.

The initialization training algorithm is usually a zero order algorithm, such as random search or the evolutionary algorithm; the main training algorithm might be a first order algorithm, such as conjugate gradient or the quasi-Newton method.

9.2 Examples

Car problem

Consider a car which is to be driven along the x-axis from some initial position and velocity to some desired position and velocity in a minimum time see Figure 9.2.

To simplify the problem, let us approximate the car by a unit point mass that can be accelerated by using the throttle or decelerated by using the brake. Selecting position and velocity as state variables the mathematical model of this system becomes a problem of two ordinary differential equations with their corresponding initial conditions.

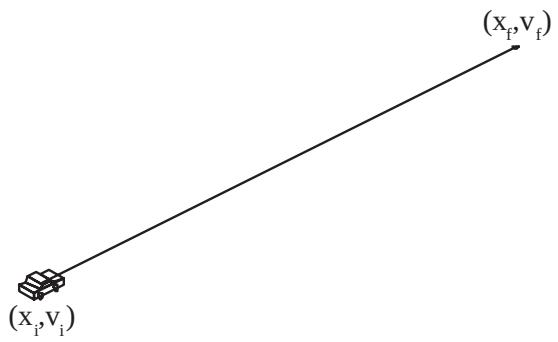

Figure 9.2: Car problem statement.

The acceleration is bounded by the capability of the engine, and the deceleration is limited by the braking system parameters.

As the objective is to make the car reach the final point as quickly as possible, the objective functional for this problem is given by the final time.

On the other hand, the car is to be driven to a desired position and a desired velocity. The erros in that target position and velocity are the constraints of the problem.

The statement and the solution itself of this car problem points out a number of significant issues. First, some variational problems might require a function space with independent parameters associated to it. Indeed, the final time is not part of the control, but it represents the interval when it is defined. Finally, this kind of applications demand spaces of functions with very good approximation properties, since they are likely to have very non- linear solutions. Here the optimal control even exhibits discontinuities.

Car problem neurocomputing

This problem is very similar to the one above. It can be formulated as an optimal control problem with one control and two state variables, and where the control is subject to two boundary conditions and lower and upper bounds. The performance functional has one constraint and requires the integration of a system of two ordinary differential equations.

Therefore, there main differences between these two car problems are in the number of control variables and in the characteristics of them. In the problem above the number of control variables is two (acceleration and deceleration), while that in this problem there is just one (acceleration, which can be negative to represent deceleration). On the other hand, in the problems above, the variables are not subject to any condition,while here the acceleration must be zero at both the initial and final times.

Fed batch fermenter

The fed batch fermenter problem formulated in this section is an optimal control problem with one control and four state variables, and defined by a performance

functional with one constraint and requiring the integration of a system of ordinary differential equations.

In many biochemical processes, the reactors are operated in fed batch mode, where the feed rate into the reactor is used for control. There is no outflow, so the feed rate must be chosen so that that batch volume does not exceed the physical volume of the reactor. As a specific example, an optimization study of the fed batch fermentation for ethanol production by Saccharomyces cerevisiae is presented. Figure 9.3, taken from Wikipedia, is a picture of a ethanol plant.

Figure 9.3: Chemical plant for ethanol production.

The fed batch fermentation process considered here is a process in which ethanol is produced by Saccharomyces cerevisiae and the production of ethanol is inhibited by itself.

A batch fermenter generally consists of a closed vessel provided with a means of stirring and with temperature control. It may be held at constant pressure or it can be entirely enclosed at a constant volume. In many biochemical processes, the reactors are operated in fed batch mode, where the feed rate into the reactor chosen so that that batch volume does not exceed the physical volume of the reactor [26].

The states of the plant are the concentration of cell mass, the concentration of substrate, the concentration of product and the broth volume in the fermenter. The control variable is the feeding rate, and it is is the only manipulated variable of this process [26].

The dynamic behavior of this fed batch fermentation process can be described by four differential-algebraic equations, together with their initial conditions.

The liquid volume of the reactor is limited by the vessel size. This constraint on the state of the system can be written as an error functional.

The desired objective is to obtain a maximum amount of yield at the end of the process. The actual yield in the reactor is given by the concentration of product multiplied by the broth volume in the reactor.

Since the equations describing the fermenter are nonlinear and the inputs

and states are constrained, the determination of the feed rate to maximize the yield can be quite difficult.

Aircraft landing

This is an optimal control problem of aeronautical engineering interest, with one control and four state variables, and where the objective functional is evaluated by integrating a system of four ordinary differential equations.

The landing of an aircraft consists of two main stages: the glide-path phase and the flare-out phase. Here we seek to determine the optimal control and the corresponding optimal trajectory of an aircraft during its final approach before landing. Figure 9.4 illustrates the landing process of an aircraft. That picture is taken from Wikipedia.

Figure 9.4: Landing of an aircraft.

The aircraft landing problem examined here is similar to that considered in [11].

In the flare-out phase the longitudinal dynamics of the aircraft are governed by the pitch angle, which in turn is controlled by the elevator deflection angle [1].

Figure 9.5 depicts the pitch and the elevator deflection angles of an aircraft.

Figure 9.5: Elevator deflection angle and pitch angle.

Thus the aim of the aircraft landing problem considered here is to determine an optimal elevator deflection angle as a function of time that satisfies a set of performance requirements.

As stated earlier, the elevator controls the longitudinal motion of the aircraft. It is assumed that any control signal is instantaneously represented by the elevator. The elevator deflection angle is also physically limited to a finite range.

The following variables will be used to describe the dynamics of the aircraft [11]: the pitch angle rate, the pitch angle, the altitude rate and, the altitude.

The velocity of the aircraft and it is assumed to be constant during the flare-out phase.

The mathematical model shows that the elevator deflection angle has a direct effect on the pitch angle rate, which in turn affects the pitch angle, the altitude rate and the altitude.

The performance requirements define the physical constraints and desired values of the control and the state variables. The most important requirements and constraints for the landing system considered in this problem are highlighted in the following section.

In our example problem the flare-out procedure ends at the final or touchdown time.

During a process it is often desirable to be able to define the desired value of a given state variable; this information can then be used to evaluate the performance of the system. The desired altitude of the aircraft is the most visual characteristic of the landing procedure. For this problem it is given by Figure 9.6.

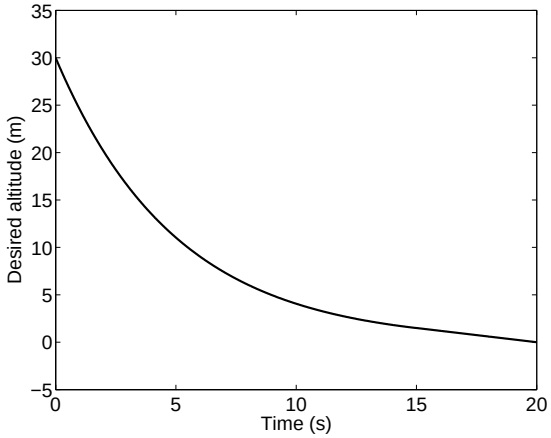

Figure 9.6: Desired landing altitude.

It is desirable to land without expending excessive amounts of control effort. Therefore, a regularization term can be added.

At the time of touchdown the pitch angle of the aircraft must lie in an appropriate range. This requirement is defined by a set of physical limitations. The lower limit serves to ensure the nose wheel of a tricycle landing gear does not touchdown prematurely. Similarly the upper limit is set to prevent the tail gear touching downing first. A desired pitch angle at touchdown could be specified as a constraints term.

Chapter 10

Optimal shape design

Optimal shape design is another learning tasks for neural networks. That problem type is formulated in a very similar way than optimal control. `OpenNN` classes which are related to the solution of optimal shape design problems include the mathematical model and several performance term classes.

10.1 Basic theory

Optimal shape design is a very interesting field both mathematically and for industrial applications. The goal here is to computerize the design process and therefore shorten the time it takes to design or improve some existing design. In an optimal shape design process one wishes to optimize a criteria involving the solution of some mathematical model with respect to its domain of definition [27]. The detailed study of this subject is at the interface of variational calculus and numerical analysis.

In order to properly define an optimal shape design problem the following concepts are needed:

1. Mathematical model.
2. Neural network.
3. Performance functional.
4. Training strategy.

Mathematical model

The mathematical model or state equation is a well-formed formula which involves the physical form of the device to be optimized. It contains shape variables and state variables.

A mathematical model might be described by algebraic equations, ordinary differential equations or partial differential equations.

Neural network

A neural network is used to represent the shape variables. Optimal shape design problems are usually defined by constraints on the shape function. Two important types of shape constraints are boundary conditions and lower and upper bounds.

Performance functional

The performance functional of an optimal shape design problem always includes an objective term. It usually includes a constraints term,

$$\text{Performance functional} = \text{objective term} + \text{constraints term}.$$

An optimal shape design problem might also be specified by a set of constraints on the state variables of the device.

State constraints are conditions that the solution to the problem must satisfy. This type of constraints vary according to the problem at hand.

In this way, a design which satisfies all the shape and state constraints is called an admissible shape.

Similarly, a state which satisfies the constraints is called an admissible state.

The performance criterion expresses how well a given design does the activity for which it has been built.

Optimal shape design problems solved in practice are, as a rule, multi-criterion problems. This property is typical when optimizing the device as a whole, considering, for example, weight, operational reliability, costs, etc. It would be desirable to create a device that has extreme values for each of these properties. However, by virtue of contradictory of separate criteria, it is impossible to create devices for which each of them equals its extreme value.

Trainining strategy

The performance functional for optimal shape design problems might contain up to three terms: objective, regularization and constraints. On the other hand, in most of the cases, it cannot be computed analitycally. That makes that a single training algorithm might not fully converge if the solution is far away from the optimal one.

Therefore, when solving optimal shape design problems, it is recommended to use an initialization training algorithm before the main training process. The form of the training strategy is therefore as follows:

Training strategy: initialization training algorithm, main training algorithm.

The initialization training algorithm is usually a zero order algorithm, such as random search or the evolutionary algorithm; the main training algorithm might be a first order algorithm, such as conjugate gradient or the quasi-Newton method.

10.2 Examples

Minimum drag problem

The minimum drag problem formulated here an optimal shape design problem with one input and one output variables, besides two boundary conditions. It is defined by an unconstrained objective functional requiring the integration of a function.

Consider the design of a body of revolution with given length and diameter providing minimum drag at zero angle of attack and for neglected friction effects. Figure 10.1 is a picture of a space shuttle, which could be that body of revolution. That picture is taken from Wikipedia.

Figure 10.1: Space shuttle.

For a slender body, the pressure coefficient can be approximated by the Newtonian flow relation. The Newtonian flor provides us with a simple approximation for the drag.

Chapter 11

Inverse problems

Iverse problems are the most complex applications for neural networks, since learning is performed here from both mathematical models and data sets. `OpenNN` classes which are related to the solution of inverse problems include the mathematical model, the data set, several performance term classes and the testing analysis classes.

11.1 Basic theory

Inverse problems can be described as being opposed to direct problems. In a direct problem the cause is given, and the effect is determined. In an inverse problem the effect is given, and the cause is estimated [23]. There are two main types of inverse problems: input estimation, in which the system properties and output are known and the input is to be estimated; and properties estimation, in which the the system input and output are known and the properties are to be estimated [23]. Inverse problems are found in many areas of science and engineering.

An inverse problem is specified by the following concepts:

- Mathematical model.
- Data set.
- Neural network.
- Performance functional.
- Training strategy.

Mathematical model

The mathematical model can be defined as a representation of the essential aspects of some system which presents knowledge of that system in usable form.

Data set

Inverse problems are those where a set of measured results is analyzed in order to get as much information as possible on a mathematical model which is proposed to represent a real system.

Therefore, a data set on the state variables is needed in order to estimate the unknown variables of that system.

In general, that data set is invariably affected by noise and uncertainty, which will translate into uncertainties in the system inputs or properties.

Neural network

The neural network represents here the inputs to the system or the properties of that system. They might include boundary conditions or bounds.

Performance functional

For inverse problems, the presence of restrictions is typical. State constraints are those conditions that the system needs to hold. This type of restrictions depend on the particular problem.

In this way, an unknown which satisfies all the input and state constraints is called an admissible unknown.

Also, a state which satisfies the state constraints is called an admissible state.

The inverse problem provides a link between the outputs from the model and the observed data. When formulating and solving inverse problems the concept of error functional is used to specify the proximity of the state variable to the observed data.

Some common performance functionals for invers problems are the inverse sum squared error or the inverse Minkowski error. Regularization theory can also be applied here [7].

The solution of inverse problems is then reduced to finding the extremum of a functional.

On the other hand, inverse problems might be ill-posed [37]. A problem is said to be well possed if the following conditions are true: (a) the solution to the problem exists; (b) the solution is unique; and (c) the solution is stable. This implies that for the above-considered problems, these conditions can be violated. Therefore, their solution requires application of special methods. In this regard, the use of regularization theory is widely used [12].

In some elementary cases, it is possible to establish analytic connections between the sought inputs or properties and the observed data. But for the majority of cases the search of extrema for the error functional must be carried out numerically, and the so-called direct methods can be applied.

Training strategy

The training strategy is entrusted to solve the reduced function optimization problems. When possible, a quasi-Newton problem should be used. If the gradient of the performance function cannot be computed accurately, an evolutionary algorithm could be used instead.

11.2 Examples

Precipitate dissolution modeling

This is an property estimation problem with one input variable, one output variable and one independent parameter. The mathematical model here is expressed as an algebraic equation.

The objective is to model the dissolution rate of hardening precipitates in aluminium alloys. The effective activation energy is also in unison determined as that providing the best results. Aluminium alloys 2014-T6 and 7449-T79 are considered. Figure 11.1 shows a Vickers hardness tester (picture from Wikipedia).

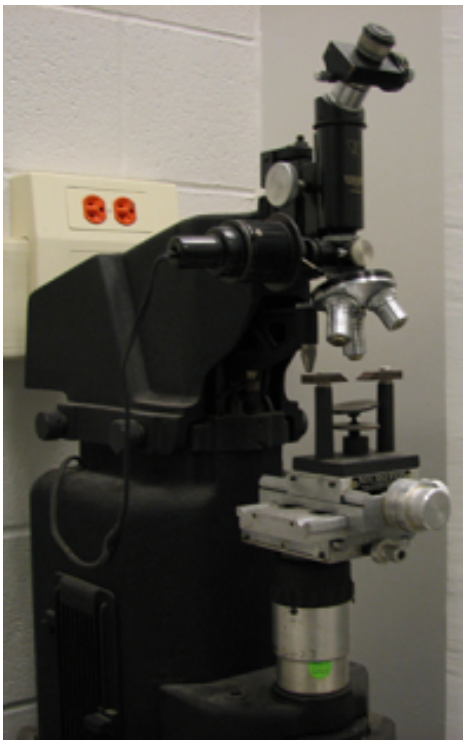

Figure 11.1: Vickers hardness tester.

Assuming that the nucleation of precipitates is negligible compared to the dissolution of precipitates, the following linear relationship between the Vickers hardness and the volumetric fraction of precipitates can be established [29].

The Vickers hardness equation is extremely useful since the hardness is much easier to measure than the relative volume fraction of precipitates.

The dissolution modeling process is to estimate an activation energy providing minimum dispersion for the experimental data while a function providing minimum error. Mathematically, this can be formulated as a variational problem including independent parameters.

Experimental tests have been performed in order to get the isothermal time evolution of Vickers hardness at different temperatures and for various alu-

minium alloys. In particular, two materials have been used for the isothermal heat treatments, 2014-T6 and 7449-T79 aluminium alloys.

The Vickers hardness data for aluminium alloy 2014-T6 is taken from [35], while that for aluminium alloy 7449-T79 is obtained from an independent test performed within the DEEPWELD Specific Targeted Research Project (STREP) co-funded by the 6th Framework Programme of the European Community (AST4-CT-2005-516134).

Figures 11.2 and 11.3 depict these Vickers hardness test for aluminium alloys 2014-T6 and AA7449-T6, respectively. Note that, in both figures the Vickers hardness decreases with the time, due to dissolution of hardness precipitates.

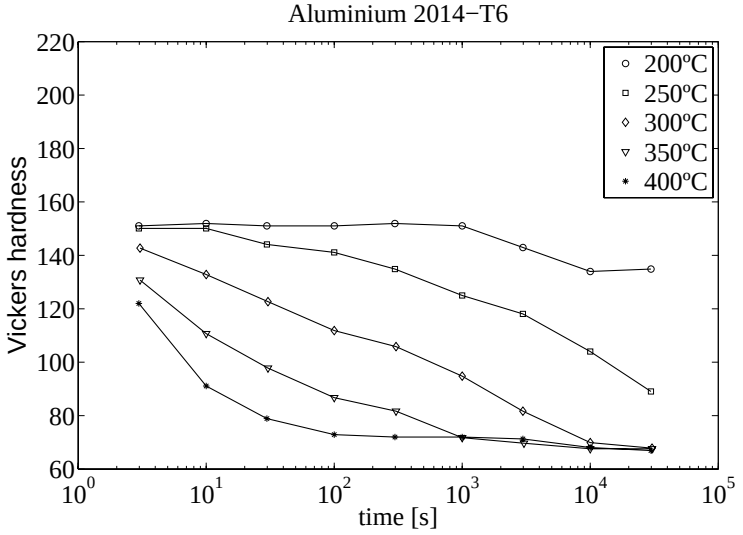

Figure 11.2: Vickers hardness test for aluminium alloy 2014-T6.

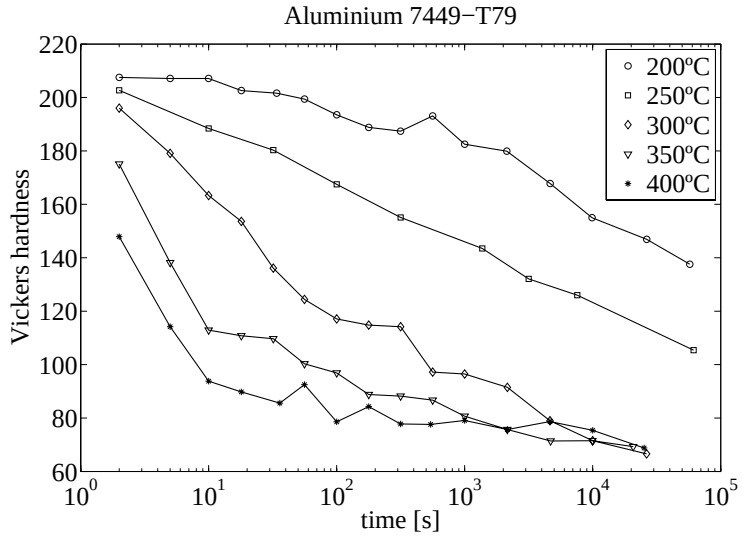

Figure 11.3: Vickers hardness test for aluminium alloy 7449-T79.

Chapter 12

Function optimization

`Open` provides a workaround for function optimization problem. It also includes some benchmark problems in function optimization.

12.1 Basic theory

The variational problem is formulated in terms of finding a function which is an extremal argument of some performance functional. On the other hand, the function optimization problem is formulated in terms of finding a vector which is an extremal argument of some performance function.

While neural networks naturally leads to the solution of variational problems, `OpenNN` provides a workaround for function optimization problems by means of the independent parameters.

Function optimization refers to the study of problems in which the aim is to minimize or maximize a real function. In this way, the performance function defines the optimization problem itself.

The formulation of a function optimization problem requires:

- A neural network.
- A performance functional.
- A training strategy.

Neural network

The independent parameters of a neural network spans a vector space to represent the possible solutions of a function optimization problem.

Performance functional

The function to be optimized is called the performance function. The domain of the objective function for a function optimization problem is the set of independent parameteres, and the image of that function is the set of real numbers. The number of variables in the objective function is the number of independent parameters.

A function optimization problem can be specified by a set of constraints, which are equalities or inequalities that the solution must satisfy. Such constraints are expressed as functions.

Thus, the constrained function optimization problem can be formulated so as to find a vector such that the constrainst functions are zero and for which the performance function takes on a minimum value.

In other words, the constrained function optimization problem consists of finding an argument which makes all the constraints to be satisfied and the objective function to be an extremum. The integer l is known as the number of constraints in the function optimization problem.

Training strategy

The training strategy is the solving strategy for the optimization problem. If possible, the quasi-Newton method should be applied here. If that fails, the evolutionary algorithm can be used.

12.2 Examples

This section describes a number of test functions for optimization. That functions are taken from the literature on both local and global optimization.

The Rosenbrock's function

The Rosenbrock's function, also known as banana function, is an unconstrained and unimodal function. The optimum is inside a long, narrow, parabolic shaped flat valley. Convergence to that optimum is difficult and hence this problem has been repeatedly used in assess the performance of optimization algorithms. The Rosenbrock's function optimization problem in d variables can be stated as:

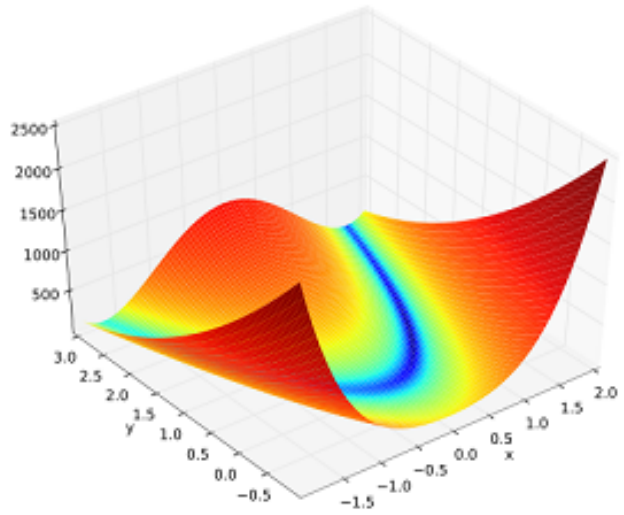

Figure 12.1: The Rosenbrock's function in 2 variables.

The minimal argument of the Rosenbrock's function is found at $(1, \ldots, 1)$. The minimum value of that function is $= 0$. Figure 12.1 is a plot of the Rosenbrock's function in 2 variables.

The Rastrigin's function

The Rastrigin's function is based on the De Jong's function with the addition of cosine modulation to produce many local minima. As a result, this function is highly multimodal. However, the location of the minima are regularly distributed. The Rastrigin's function optimization problem in d variables can be stated as:

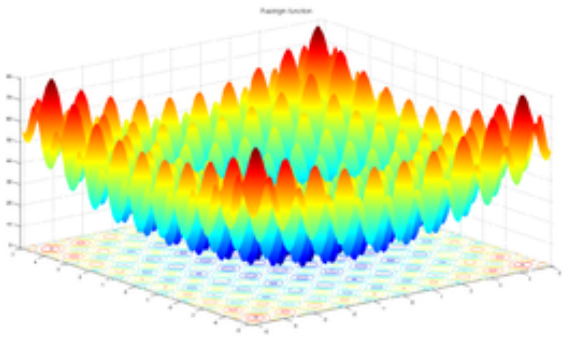

Figure 12.2: The Rastrigin's function in 2 variables.

The global minimum of the Rastrigin's Function is at $(0, \ldots, 0)$. At this minimal argument the value of the function is 0. Figure 12.2 is a plot of the Rastrigin's function in 2 variables.

The gradient vector for the Rastrigin's function is given by and the Hessian matrix by

Appendix A

Unit testing

The unit testing development pattern

Unit testing is the process of creating integrated tests into a source code, and running those tests every time it is to be built. In that way, the build process checks not only for syntax errors, but for semantic errors as well.

In that regard, unit testing is generally considered a development pattern, in which the tests would be written even before the actual code. If tests are written first, they:

- Describe what the code is supposed to do in concrete, verifiable terms.
- Provide examples of code use rather than just academic descriptions.
- Provide a way to verify when the code is finished (when all the tests run correctly).

Related code

There exist several available frameworks for incorporating test cases in C++ code, such as CppUnit or Cpp test. However, for portability reasons, `OpenNN` comes with a simple unit testing utility class for handing automated tests. Also, every classes and methods have test classes and methods associated.

The UnitTesting class in OpenNN

`OpenNN` includes the UnitTesting abstract class to provide some simple mechanisms to build test cases and test suites.

Constructor

Unit testing is to be performed on classes and methods. Therefore the UnitTesting class is abstract and it can't be instantiated. Concrete test classes must be derived here.

Members

The UnitTesting class has the following members:

- The counted number of tests.
- The counted number of passed tests.
- The counted number of failed tests.
- The output message.

That members can be accessed or modified using get and set methods, respectively.

Methods

Derived classes must implement the pure virtual run_test_case method, which includes all testing methods. The use of this method is as follows:

```
TestMockClass tmc;
tmc.run_test_case();
```

The assert_true and assert_false methods are used to prove if some condition is satisfied or not, respectively. If the result is correct, the counter of passed tests is increased by one; otherwise the counter of failed tests is increased by one,

```
unsigned int a = 0;
unsigned int b = 0;

TestMockClass tmc;
tmc.assert_true(a == b, "Increase tests passed count");
tmc.assert_false(a == b, "Increase tests failed count");
```

Finally, the print_results method prints the testing outcome,

```
TestMockClass tmc;
tmc.run_test_case();
tmc.print_results();
```

The unit testing classes

Every single class in `OpenNN` has a test class associated, and every single method of that class has also a test method associated.

On the other hand, a test suite of all the classes distributed within `OpenNN` can be found in the folder AllTests.

Bibliography

[1] H. Ashley. *Engineering Analysis of Flight Vechicles*. Dover Publishing, 1992.

[2] E. Balsa-Canto. *Algoritmos Eficientes para la Optimizacion Dinamica de Procesos Distribuidos*. PhD thesis, Universidad de Vigo, 2001.

[3] R. Battiti. First and second order methods for learning: Between steepest descent and newton's method. *Neural Computation*, 4(2):141–166, 1992.

[4] L. Belanche. *Heterogeneous Neural Networks*. PhD thesis, Technical University of Catalonia, 2000.

[5] C. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[6] T.F. Brooks, D.S. Pope, and A.M. Marcolini. Airfoil self-noise and prediction. Technical report, NASA RP-1218, July 1989.

[7] D. Bucur and G. Buttazzo. *Variational Methods in Shape Optimization Problems*. Birkhauser, 2005.

[8] Z. Chen and S. Haykin. On different facets of regularization theory. *Neural Computation*, 14(12):2791–2846, 2002.

[9] H. Demuth, M. Beale, and M. Hagan. *Neural Network Toolbox User's Gide*. The MathWorks, Inc., 2009.

[10] B. Eckel. *Thinking in C++. Second Edition*. Prentice Hall, 2000.

[11] F.J. Ellert and C.W. Merriam. Synthesis of feedback controls using optimization theory - an example. *IEEE Transactions on Automatic Control*, 8(2):89–103, 1963.

[12] H.W. Engl, M. Hanke, and A. Neubauer. *Regularization of Inverse Problems*. Springer, 2000.

[13] S. Eyi, J.O. Hager, and K.D. Lee. Airfoil design optimization using the navier-stokes equations. *Journal of Optimization Theory and Applications*, 83(3):447–461, 1994.

[14] D.B. Fogel. An introduction to simulated evolutionary optimization. *IEEE Transactions on Neural Networks*, 5(1):3–14, 1994.

[15] A. Frank and A. Asuncion. UCI machine learning repository, 2010.

[16] J. Gerritsma, R. Onnink, and A. Versluis. Geometry, resistance and stability of the delft systematic yacht hull series. In *International Shipbuilding Progress*, volume 28, pages 276–297, 1981.

[17] F. Girosi, M. Jones, and T. Poggio. Regularization theory and neural network architectures. *Neural Computation*, 7(2):219–269, 1995.

[18] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley, 1988.

[19] M.T. Hagan and M. Menhaj. Training feedforward networks with the marquardt algorithm. *IEEE Transactions on Neural Networks*, 5(6):989–993, 1994.

[20] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 1994.

[21] J. Kennedy and R.C. Eberhart. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.

[22] D.E. Kirk. *Optimal Control Theory. An Introduction*. Prentice Hall, 1970.

[23] A. Kirsch. *An Introduction to the Mathematical Theory of Inverse Problems*. Springer, 1996.

[24] R. Lopez and E. Oñate. A variational formulation for the multilayer perceptron. In *Proceedings of the 16th International Conference on Artificial Neural Networks ICANN 2006*, 2006.

[25] D.G. Luenberger. *Linear and Nonlinear Programming*. Addison Wesley, 1984.

[26] R. Luus. Application of dynamic programming to differential-algebraic process systems. *Computers and Chemical Engineering*, 17(4):373–377, 1993.

[27] B. Mohammadi and O. Pironneau. Shape optimization in fluid mechanics. *Annual Review of Fluid Mechanics*, 36:255–279, 2004.

[28] M.F. Moller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*, 6:525–533, 1993.

[29] O.R. Myrh and Ø. Grong. Process modelling applied to 6082-t6 aluminium weldments - ii. applications of model. *Acta Metallurgica et Materialia*, 39(11):2703–2708, 1991.

[30] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C++: The Art of Scientific Computing*. Cambridge University Press, 2002.

[31] A.G. Ramm. *Inverse Problems. Mathematical and Analytical Techniques with Applications to Engineering*. Springer, 2005.

[32] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.

[33] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (Third Edition)*. Prentice Hall, 2009.

[34] P.C. Sabatier. Past and future of inverse problems. *Journal of Mathematical Physics*, 41:4082–4124, 2000.

[35] H.R. Shercliff, M.J. Russel, A. Taylor, and T.L. Dickerson. Microstructural modelling in friction stir welding of 2000 series aluminium alloys. *Mecanique & Industries*, 6:25–35, 2005.

[36] B. Stroustrup. *The C++ Programing Language*. Addison Wesley, 2000.

[37] A.N. Tikhonov and V.Y. Arsenin. *Solution of ill-posed problems*. Wiley, 1977.

[38] J. Šíma and P. Orponen. General-purpose computation with neural networks: A survey of complexity theoretic results. *Neural Computation*, 15:2727–2778, 2003.

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