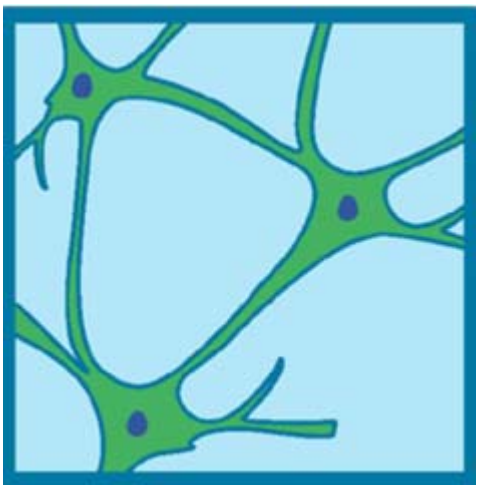
Flood

An Open Source Neural Networks C++ Library

R. López



www.cimne.com/flood

Monograph CIMNE Nº-120, September 2010

Flood

An Open Source Neural Networks C++ Library

R. López

Monograph CIMNE Nº-120, September 2010

International Center for Numerical Methods in Engineering
Gran Capitán s/n, 08034 Barcelona, Spain

INTERNATIONAL CENTER FOR NUMERICAL METHODS IN ENGINEERING
Edificio C1, Campus Norte UPC
Gran Capitán s/n
08034 Barcelona, Spain
www.cimne.upc.es

First edition: September 2010

FLOOD. AN OPEN SOURCE NEURAL NETWORKS C++ LIBRARY
Monograph CIMNE M120

ISBN: 978-84-96736-96-2

Depósito legal: B-31072-2010

Preface

The multilayer perceptron is an important model of neural network, and much of the literature in the field is referred to that model. The multilayer perceptron has found a wide range of applications, which include function regression, pattern recognition, time series prediction, optimal control, optimal shape design or inverse problems. All these problems can be formulated as variational problems. That neural network can learn either from databases or from mathematical models.

`Flood` is a comprehensive class library which implements the multilayer perceptron in the C++ programming language. It has been developed following the functional analysis and calculus of variations theories. In this regard, this software tool can be used for the whole range of applications mentioned above. `Flood` 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.

The user's guide is organized as follows. Chapter 1 describes the most basic data structures of `Flood`. In Chapter 2, a brief introduction to the principal concepts of neural networks and the multilayer perceptron is given. Chapters 3, 4, 5 and 6 state the learning problem for the multilayer perceptron and provide a collection of related algorithms. In Chapters 7, 8, 9, 10, 11 and 12 the most important learning tasks for that neural network are formulated and several practical applications are also solved. Appendixes A and B present some activities related to the software engineering process of `Flood`. Finally, Appendix C introduces some numerical integration algorithms.

List of symbols

The symbols used for the most common quantities in this user's guide are listed below:

Lower case letters

$a(\cdot)$	activation function
b	bias
$c(\cdot)$	combination function
d	parameters number
$e(\cdot)$	error function
$f(\cdot)$	objective function
h	hidden layers number
l	constraints number
m	outputs number
n	inputs number
p	population size
q	instances number
r	recombination size
s	hidden layer size
t	target
w	synaptic weight
x	input
$y(\cdot)$	output function

Lower case bold letters

$\mathbf{c}(\cdot)$	layer activation function
$\mathbf{b}$	layer biases
$\mathbf{c}(\cdot)$	layer combination function
$\mathbf{d}$	training direction
$\mathbf{f}$	population evaluation
$\mathbf{l}$	lower bound
$\mathbf{s}$	selection vector
$\mathbf{t}$	target vector
$\mathbf{x}$	input vector
$\mathbf{y}(\cdot)$	several variables output function
$\mathbf{u}$	upper bound
$\mathbf{w}$	perceptron synaptic weights

Upper case letters

$C[\cdot]$	constraint functional
$E[\cdot]$	error functional
$F[\cdot]$	objective functional
I	independent parameters
L	layer
N	neural parameters
R	Minkowski parameter
V	function space

Upper case bold letters

$\mathbf{C}$	Confusion matrix
$\mathbf{G}$	inverse Hessian approximation
$\mathbf{H}$	Hessian matrix
$\mathbf{J}$	Jacobian matrix
$\mathbf{P}$	population matrix
$\mathbf{W}$	layer synaptic weights

Lower case greek letters

γ	conjugate gradient parameter
δ	delta quantity
ζ	parameter
η	training rate
μ	mean
ν	regularization weight
ρ	penalty term weight
σ	standard deviation
$\varphi(\cdot)$	boundary condition function

Lower case greek bold letters

$\boldsymbol{\zeta}$	parameter vector
$\boldsymbol{\mu}$	mean vector
$\boldsymbol{\sigma}$	standard deviation vector
$\boldsymbol{\phi}$	fitness vector
$\boldsymbol{\varphi}(\cdot)$	several variables boundary condition function

Upper case greek letters

$\Omega[\cdot]$	regularization functional

Other symbols

$\bar{\cdot}$	scaling
$\widehat{\cdot}$	boundary conditions
$\widetilde{\cdot}$	lower-upper bounding

Contents

Chapter 1

Preliminaries

The `Flood` namespace allows to group all the entities in the library under a name. On the other hand, the `Flood` library includes its own Vector and Matrix container classes. That classes contain high level constructors, operators and methods which allow abstraction from some hard details of C++.

1.1 The Flood namespace

Each set of definitions in the `Flood` library is 'wrapped' in the namespace Flood. 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 [16]. For the Flood namespace the following sentence can be written:

```
using namespace Flood;
```

1.2 The Vector class

The Vector class is a template, which means that it can be applied to different types [16]. That is, we can create a Vector or **int** numbers, MyClass objects, etc.

Members

The only members of the Vector class are:

- The size of the vector.

- 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 of XML type.

```
<Flood version='3.0' class='Vector'>
<Size>
size
</Size>
<Display>
display
</Display>
<Data>
element_0 element_1 ... element_N
</Data>
```

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 get_size returns the size of a Vector.

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

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

```
Vector<bool> v(3);
v.set_size(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 serialization = 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.

1.3 The Matrix class

As it happens with the Vector class, the Matrix class is also a template [16]. 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 of XML type.

```
<Flood version='3.0' class='Matrix'>
<RowsNumber>
rows_number
</RowsNumber>
<ColumnsNumber>
columns_number
</ColumnsNumber>
<Display>
display
</Display>
<Data>
element_00 ... element_0M
...
element_N0 ... element_NM
</Data>
```

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

Introduction

There are many different types of neural networks, from which the multilayer perceptron is an important one. Most of the literature in the field is referred to that neural network. Here we formulate the learning problem and describe some learning tasks which a multilayer perceptron can solve.

2.1 Learning problem

The multilayer perceptron is characterized by a neuron model, a network architecture and associated objective functionals and training algorithms. The learning problem is then formulated as to find a multilayer perceptron which optimizes an objective functional by means of a training algorithm.

Perceptron

A neuron model is a mathematical model of the behavior of a single neuron in a biological nervous system. The characteristic neuron model in the multilayer perceptron 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.

Although a perceptron can solve some very simple learning tasks, the power of neural networks comes when many of that neuron models are connected in a network architecture.

Multilayer perceptron

In the same way a biological nervous system is composed of interconnected biological neurons, an artificial neural network is built up by organizing ar-

tificial neurons in a network architecture. In this way, the architecture of a network refers to the number of neurons, their arrangement and connectivity. The characteristic network architecture in the multilayer perceptron is the so called feed-forward architecture.

The multilayer perceptron can then be defined as a network architecture of perceptron neurons. This neural network represents a parameterized function of several variables with very good approximation properties.

Objective functional

The objective 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 network is required to learn. The choice of a suitable objective functional depends on the particular application.

Function regression and pattern recognition problems share the same objective functionals. They are based on the sum squared error. On the other hand a concrete objective functional must be derived when solving optimal control, optimal shape design or inverse problems.

Training algorithm

The procedure used to carry out the learning process is called training algorithm, or learning algorithm. The training algorithm is applied to the network to in order to obtain a desired performance. The type of training is determined by the way in which the adjustment of the parameters in the neural network takes place.

One of the most suitable training algorithms for the multilayer perceptron is 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 in the multilayer perceptron 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. The multilayer perceptron provides 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 multilayer perceptron which will approximate the solution to the

problem. In the second step the variational problem is formulated by selecting an appropriate objective functional. The third step is to solve the reduced function optimization problem with a training algorithm capable of finding an optimal set of parameters.

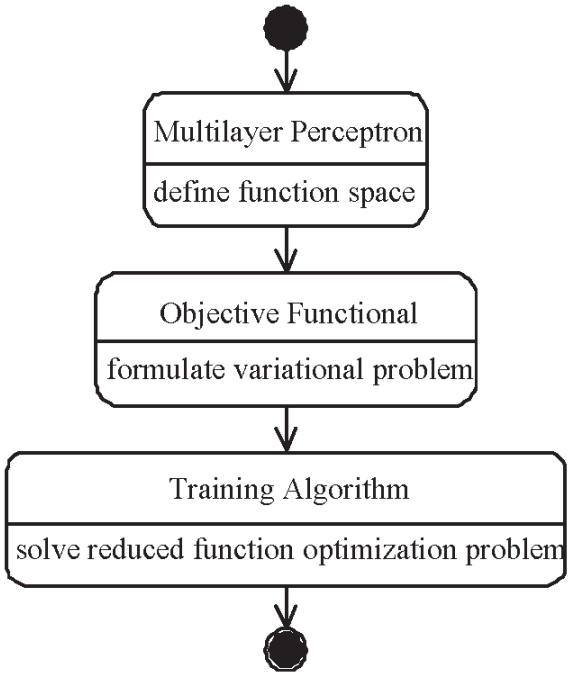


Figure 2.1: Learning problem for the multilayer perceptron.

2.2 Learning tasks

Learning tasks for the multilayer perceptron can be classified according to the way in which they can be applied for a particular purpose. The learning problem is stated in terms of variational calculus. In this way, some classes of learning tasks of practical interest are function regression, pattern recognition, optimal control, optimal shape design or invers problems. Here the neural network can learn from dabases and from mathematical models.

Function regression

Function regression is the most popular learning task for the multilayer perceptron. It is also called modelling. The function regression problem can be

regarded as the problem of approximating a function from a database consisting of input-target instances [24]. The targets are a specification of what the response to the inputs should be [8]. While input variables might be quantitative or qualitative, in function regression target variables are quantitative.

Objective functionals 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, some specialities 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 [15].

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 [24]. 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 [8].

Classification problems can be, in fact, formulated as being modelling problems. As a consequence, objective 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 [8].

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 [29] [4].

The knowledge in optimal control problems is not represented in the form of a database, it is given by a mathematical model. A different objective functional must be derived for every different system to be controlled. These obejctive 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.

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 [10].

As in the previous case, the neural network here learns from a mathematical model, and an objective functional must be derived for each shape to be designed. Evaluation of the objective 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 [18] [38]. 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 [30] [48] [46]. 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. Form a theoretical point of view, the multilayer perceptron here needs both mathematical models and experimental data. The objective is usually formulated as to find properties or inputs which make a mathematical model to comply with the experimental data. 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

The knoledge for a multilayer perceptron can be represented in the form of databases or mathematical models. The neural network learns from databases 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 databases in inverse problems. Please note that other possible variational applications can be added to these learning tasks.

Figure 2.2 shows the learning tasks for the multilayer perceptron described in this section. As we can see, that neural network is capable of dealing with a great range of applications. Any of that learning tasks is formulated as being a variational problem. All af 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 be as well very useful.

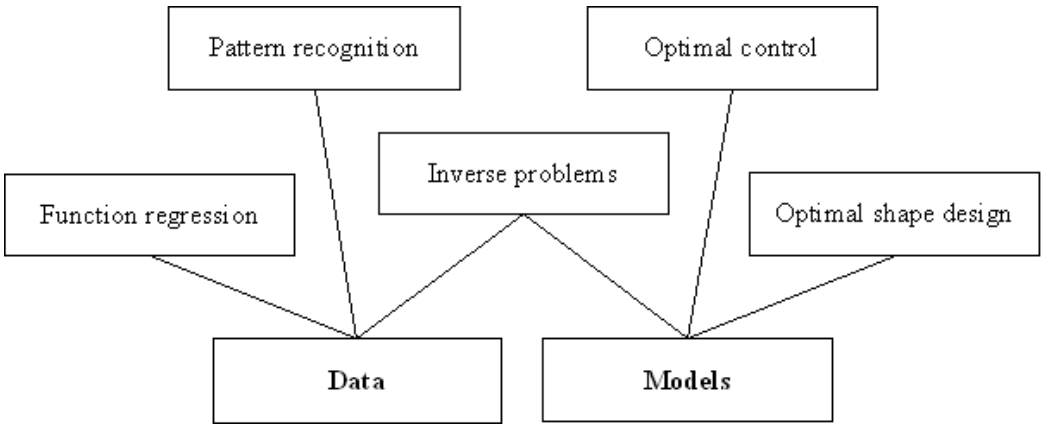


Figure 2.2: Learning tasks for the multilayer perceptron.

Chapter 3

The 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.

3.1 Neuron model

Following current practice [53], 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 3.1 is a graphical representation of a perceptron [24].

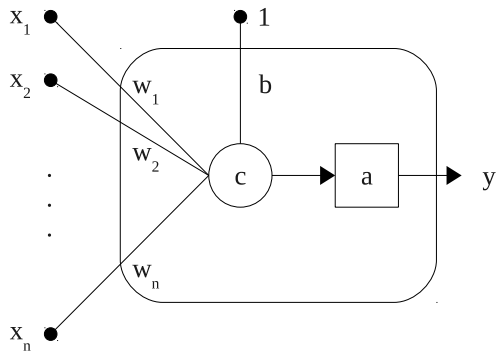


Figure 3.1: Perceptron neuron model.

Here we identify three basic elements, which transform a vector of inputs $\mathbf{x} = (x_1, \ldots, x_n)$ into a single output y [6]: (i) A set of parameters consisting

of a bias b and a vector of synaptic weights $\mathbf{w} = (w_1, \ldots, w_n)$; (ii) a combination function $c(\cdot)$; and (iii) an activation function or transfer function $a(\cdot)$.

Example 1 *The perceptron neuron model in Figure 3.2 has three inputs. It transforms the inputs (x_1, x_2, x_3) into an output y. The combination function $c(\cdot)$ merges that inputs with the bias b and the synaptic weights (w_1, w_2, w_3). The activation function $a(\cdot)$ takes that net input to produce the output from the neuron.*

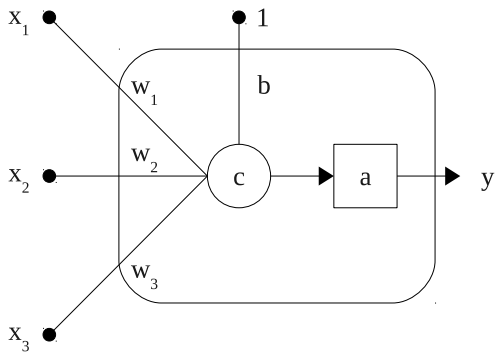


Figure 3.2: Perceptron example.

3.2 Perceptron parameters

The parameters of the perceptron are a set $(b, \mathbf{w}) \in \mathbb{R} \times \mathbb{R}^n$, where $b \in \mathbb{R}$ is called the bias and $\mathbf{w} \in \mathbb{R}^n$ is called the synaptic weight vector [24]. Note then that the number of neuron parameters of this neuron model is $1 + n$, where n is the number of inputs.

Example 2 *Consider the perceptron of Example 1. That neuron model has a bias and three synaptic weights, since the number of inputs is three. The number of parameters here is therefore four.*

If the bias and the synaptic weights are set to $b = -0.5$ and $\mathbf{w} = (1.0, -0.75, 0.25)$, respectively, then the set of parameters is

$$(\mathbf{w}, b) = (-0.5, 1.0, -0.75, 0.25) \in \mathbb{R}^4$$

3.3 Combination function

The combination function $c : X \to C$, with $X \subseteq \mathbb{R}^n$ and $C \subseteq \mathbb{R}$ takes the input vector $\mathbf{x}$ and the neuron parameters $(b, \mathbf{w})$ to produce a combination value, or net input. In the perceptron, the combination function computes the bias plus the dot product of the input and the synaptic weight vectors,

$$c(\mathbf{x}; b, \mathbf{w}) = b + \mathbf{w} \cdot \mathbf{x} \tag{3.1}$$

Note that the bias increases or reduces the net input to the activation function, depending on whether it is positive or negative, respectively. The bias is sometimes represented as a synaptic weight connected to an input fixed to +1.

Example 3 *Consider the perceptron of Example 1. If the inputs are set to* $\mathbf{x} = (-0.8, 0.2, -0.4)$ *and the neuron parameters are set to* $b = -0.5$ *and* $\mathbf{w} = (1.0, -0.75, 0.25)$*, then the combination value of this perceptron is*

$$\begin{aligned} c(-0.8, 0.2, -0.4; b = -0.5, 1.0, -0.75, 0.25) &= -0.5 + 1.0 \cdot -0.8 - 0.75 \cdot 0.25 \cdot -0.4 \\ &= -1.55 \end{aligned}$$

3.4 Activation function

The activation function or transfer function $a : C \to Y$, with $C \subseteq \mathbb{R}$ and $Y \subseteq \mathbb{R}$, will define the output from the neuron in terms of its combination. In practice we can consider many useful activation functions [15]. Some of the most used are the threshold function, the symmetric threshold function, the logistic function, the hyperbolic tangent or the linear function [24].

Threshold function

The threshold activation function $a : C \to [0, 1]$, with $C \subseteq \mathbb{R}$, limits the output of the neuron to 0 if the combination is negative, or to 1 if the combination is zero or positive,

$$a(c) = \begin{cases} 0 & c < 0, \\ 1 & c \geq 0. \end{cases} \tag{3.2}$$

This activation function is represented in Figure 3.3.

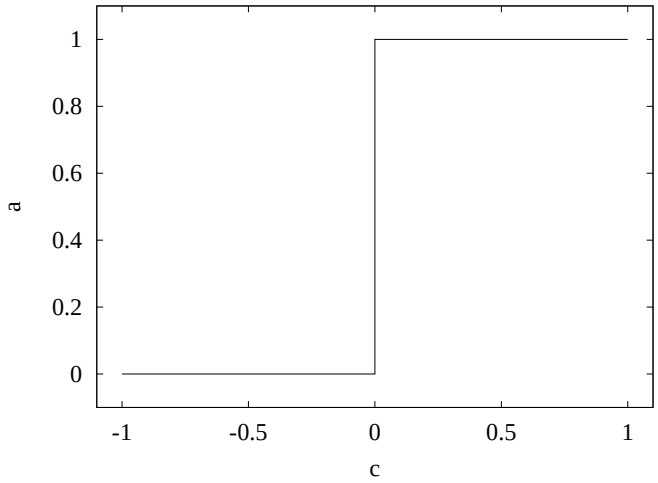


Figure 3.3: Threshold activation function.

Example 4 *Consider a perceptron with threshold activation function. If the combination value to that perceptron is* $c = -1.55$*, the activation value from that perceptron will be*

$$a(-1.55) = 0.$$

Symmetric threshold

The symmetric threshold activation function $a : C \to [-1, 1]$, with $C \subseteq \mathbb{R}$, is very similar to the threshold function, except that its image is $[-1, 1]$ instead of $[0, 1]$,

$$a(c) = \begin{cases} -1 & c < 0, \\ 1 & c \geq 0. \end{cases} \tag{3.3}$$

The shape of this function is represented in Figure 3.4.

Example 5 *If a perceptron has a symmetric threshold activation function, the activation value for a combination value* $c = -1.55$ *is*

$$a(-1.55) = -1.$$

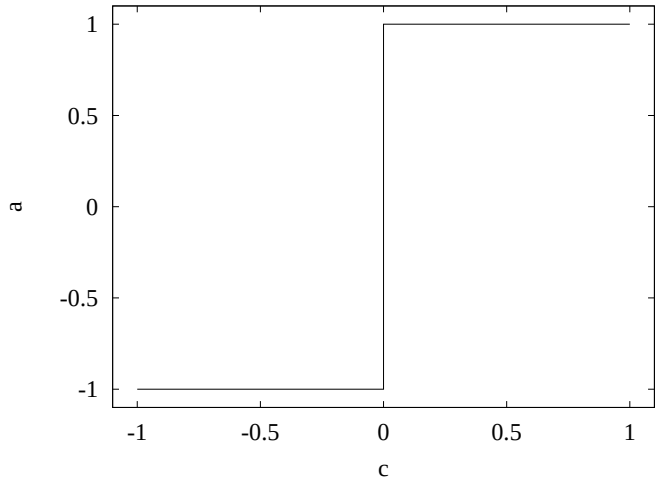


Figure 3.4: A symmetric threshold activation function.

Logistic function

The logistic function $a : C \to (0, 1)$, with $C \subseteq \mathbb{R}$, has a sigmoid shape. This activation function is a monotonous crescent function which exhibits a good balance between a linear and a non-linear behavior. It is defined by

$$a(c) = \frac{1}{1 + \exp(-c)}. \tag{3.4}$$

The logistic function is widely used when constructing neural networks. This function is represented in Figure 3.5.

Example 6 *A perceptron with logistic activation and combination* $c = -1.55$ *will have an activation*

$$\begin{aligned} a(-1.55) &= \frac{1}{1 + \exp(1.55)} \\ &= 0.175. \end{aligned}$$

Hyperbolic tangent

The hyperbolic tangent $a : C \to (0, 1)$ with $C \subseteq \mathbb{R}$, is also a sigmoid function very used in the neural networks field. It is very similar to the logistic function. The main difference is that the image of the hyperbolic tangent

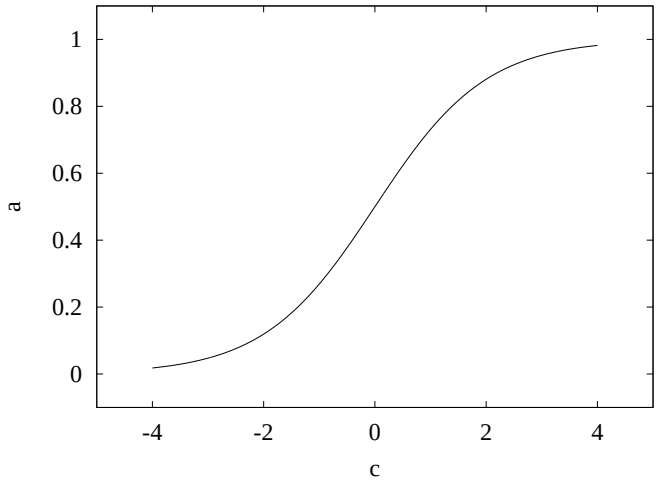


Figure 3.5: Logistic function.

is $(-1, 1)$, while the image of the logistic function is $(0, 1)$. The hyperbolic tangent is defined by

$$a(c) = \tanh(c). \tag{3.5}$$

The hyperbolic tangent function is represented in Figure 3.6.

Example 7 *The value of the activation for a combination $c = -1.55$ in the case of a perceptron with hyperbolic tangent activation function is*

$$\begin{aligned} a(-1.55) &= \tanh(-1.55) \\ &= -0.914. \end{aligned}$$

Linear function

For the linear activation function, $a : C \to A$, with $C \subseteq \mathbb{R}$ and $A \subseteq \mathbb{R}$, we have

$$a(c) = c. \tag{3.6}$$

Thus, the output of a neuron model with linear activation function is equal to its combination.

The linear activation function is described in Figure 3.7.

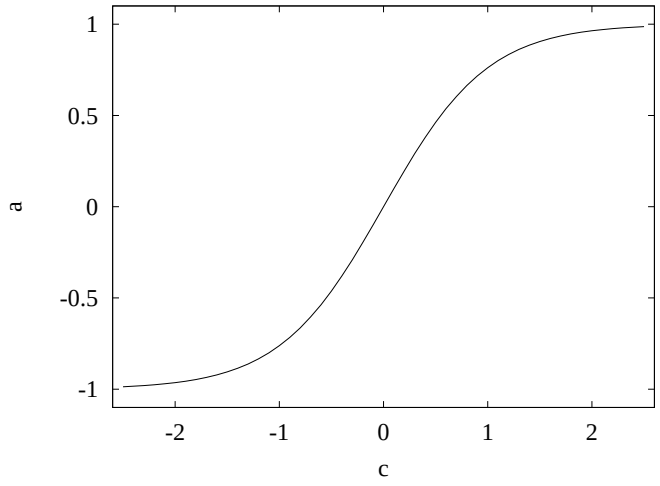


Figure 3.6: Hyperbolic tangent function.

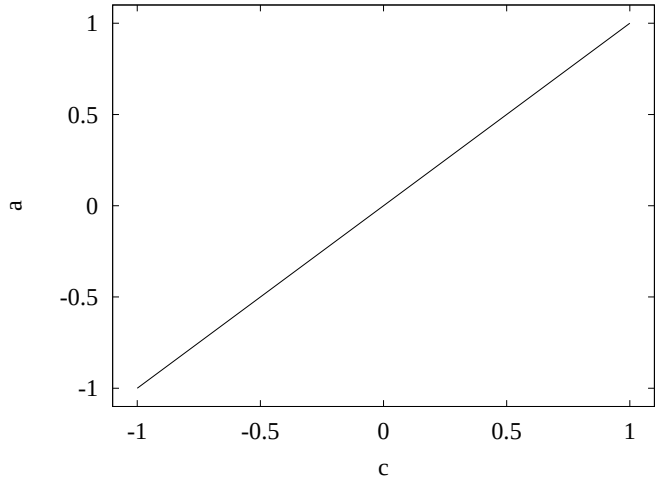


Figure 3.7: Linear function.

Example 8 *Consider a linear perceptron. The activation for a combination* $c = -1.55$ *is*

$$a(-1.55) = -1.55.$$

3.5 Perceptron function

In this section we write down an explicit expression of the output from a perceptron as a function of the inputs to it, $y : X \to Y$, with $X \subseteq \mathbb{R}^n$ and $Y \subseteq \mathbb{R}$. Here the combination is first computed as the bias plus the dot product of the the synaptic weights and the inputs, to give

$$c(\mathbf{x}; b, \mathbf{w}) = b + \mathbf{w} \cdot \mathbf{x}. \tag{3.7}$$

The output from the neuron is obtained transforming the combination in Equation (3.7) with an activation function a to give

$$y(\mathbf{x}; b, \mathbf{w}) = a\left(b + \mathbf{w} \cdot \mathbf{x}\right). \tag{3.8}$$

Thus, the output function is represented in terms of composition of the activation and the combination functions,

$$y = a \circ c. \tag{3.9}$$

Figure 3.8 is an activity diagram of how the information is propatated in the perceptron.

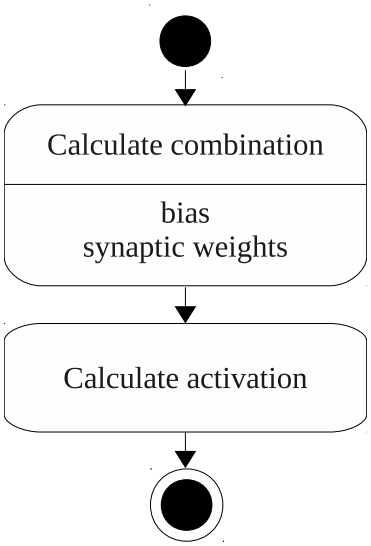


Figure 3.8: Propagation in the perceptron.

Mathematically, a perceptron spans a function space V from an input $X \subset \mathbb{R}^n$ to an output $Y \subset \mathbb{R}$. Elements of V are therefore of the form $y : X \to Y$. That functions are parameterized by the bias and the vector of synaptic weights of the neuron. In this way the dimension of V is $d = 1 + n$.

Distinct activation functions cause distinct families of functions which a perceptron can define. Similarly, distinct sets of neuron parameters cause distinct elements in the function space which a specific perceptron defines.

Example 9 *Let P be a logistic perceptron with number of inputs $n = 2$. Let set the bias to $b = -0.5$ and the synaptic weights to $\mathbf{w} = (1.0, 0.25)$. The function represented by that perceptron $y : X \to (0, 1)$, with $X \subseteq \mathbb{R}^n$, is given by*

$$y(x_1, x_2; -0.5, 1.0, 0.25) = \frac{1}{1 + \exp\left(-(-0.5 + 1.0x_1 + 0.25x_2)\right)}.$$

The output for an input $\mathbf{x} = (-0.2, 0.5)$ is

$$\begin{aligned} y(-0.2, 0.5; -0.5, 1.0, 0.25) &= \frac{1}{1 + \exp\left(-(-0.5 + 1.0 \cdot -0.2 + 0.25 \cdot 0.5)\right)} \\ &= 0.36. \end{aligned}$$

3.6 Activation derivative

There might be some cases when we need to compute the activation derivative $a' : C \to Y'$, where $C \subseteq \mathbb{R}$ and $Y' \subseteq \mathbb{R}$ of the neuron,

$$a'(c) \equiv \frac{da}{dc}. \tag{3.10}$$

Threshold function derivative

The threshold activation function is not differentiable at the point $c = 0$.

Symmetric threshold derivative

The symmetric threshold activation function is neither differentiable at the point $c = 0$.

Logistic function derivative

The logistic function derivative, $a' : C \to (0, 0.25]$, with $C \subseteq \mathbb{R}$, is given by

$$a'(c) = \frac{\exp(-c)}{(1 + \exp(-c))^2}. \tag{3.11}$$

This derivative function is represented in Figure 3.9.

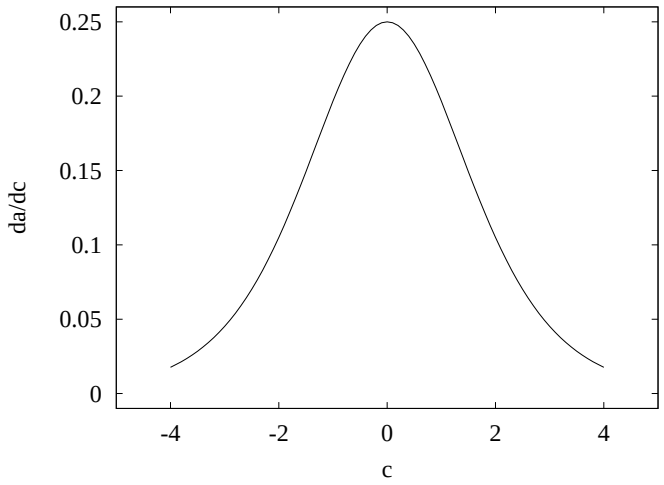


Figure 3.9: Logistic derivative.

Example 10 *Let P be a logistic perceptron. The activation derivative for a combination $c = -1.55$ is*

$$\begin{aligned} a'(-1.55) &= \frac{\exp(-1.55)}{(1 + \exp(-1.55))^2} \\ &= 0.14. \end{aligned}$$

Hyperbolic tangent derivative

The derivative of the hyperbolic tangent activation function $a' : C \to (0, 1]$, with $C \subseteq \mathbb{R}$, is given by

$$a'(c) = 1 - \tanh^2(c), \tag{3.12}$$

The hyperbolic tangent function derivative is represented in Figure 3.10.

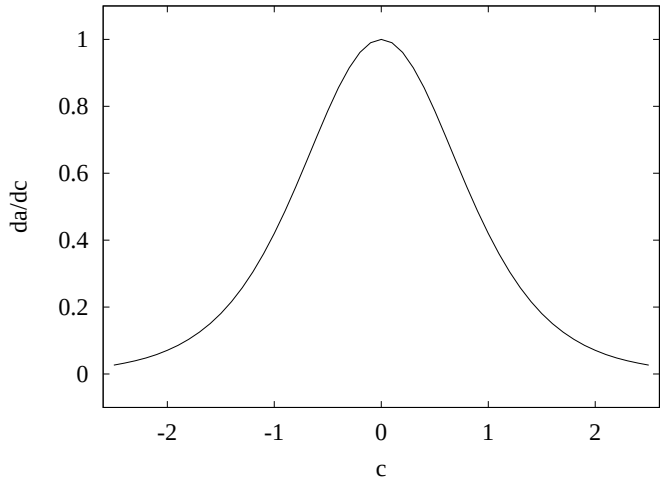


Figure 3.10: Hyperbolic tangent derivative.

Example 11 *The activation derivative of an hyperbolic tangent perceptron for a combination* $c = -1.55$ *is*

$$\begin{aligned} a'(-1.55) &= 1 - \tanh^2(-1.55) \\ &= 0.16. \end{aligned}$$

Linear function derivative

For the linear function, the activation derivative, $a' : C \to 1$, with $C \subseteq \mathbb{R}$, is given by

$$a'(c) = 1, \tag{3.13}$$

The linear activation function derivative is described in Figure 3.11.

Example 12 *Let* P *be a perceptron with linear activation. If the combination value is* $c = -1.55$*, the activation derivative is*

$$a'(-1.55) = 1.$$

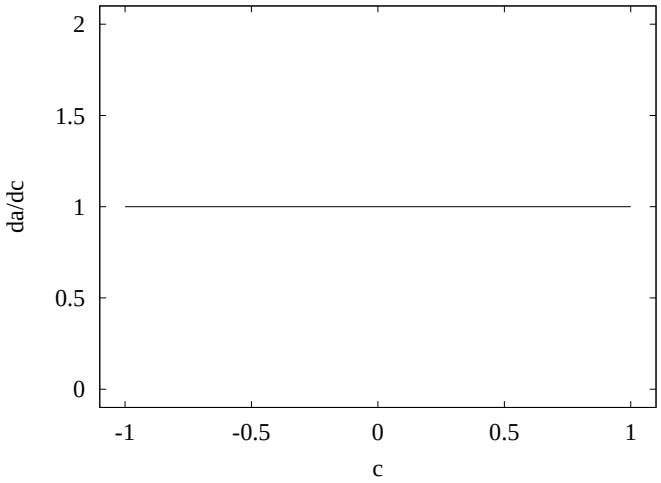


Figure 3.11: Linear derivative.

3.7 Activation second derivative

There also might be some ocasions when we need to compute the second derivative of the activation function, $a' : C \to Y''$, where $C \subseteq \mathbb{R}$ and $Y'' \subseteq \mathbb{R}$,

$$a''(c) \equiv \frac{d^2 a}{dc^2}. \tag{3.14}$$

Threshold function second derivative

The threshold activation function is not differentiable at the point $c = 0$.

Symmetric threshold second derivative

The symmetric threshold activation function is neither differentiable at the point $c = 0$.

Logistic function second derivative

The second derivative of the logistic function, $a'' : C \to C'$, with $C, C' \subseteq \mathbb{R}$, is given by

$$a''(c) = -\frac{\exp(-c)}{(1+\exp(-c))^2} + 2\frac{(\exp(-c))^2}{(1+exp(-c))^3} \tag{3.15}$$

This function is represented in Figure 3.12.

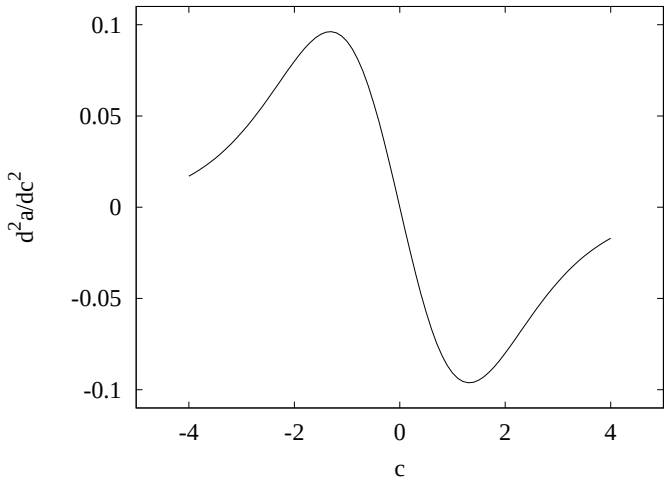


Figure 3.12: Logistic second derivative.

Example 13 *Consider a logistic perceptron P. The activation second derivative for a combination $c = -1.55$ is*

$$\begin{aligned} a''(-1.55) &= -\exp(-1.55)\frac{\exp(-1.55)-1}{(\exp(-1.55+1))^3} \\ &= 0.87. \end{aligned}$$

Hyperbolic tangent second derivative

The second derivative of this activation function, $a'' : C \to C''$, with $C, C'' \subseteq \mathbb{R}$, is given by

$$a''(c) = -2\tanh(c)(1-\tanh^2(c)). \tag{3.16}$$

The hyperbolic tangent function second derivative is represented in Figure 3.13.

Example 14 *Consider a perceptron P with hyperbolic tangent activation function. The activation second derivative produced for a combination $c = -1.55$ is*

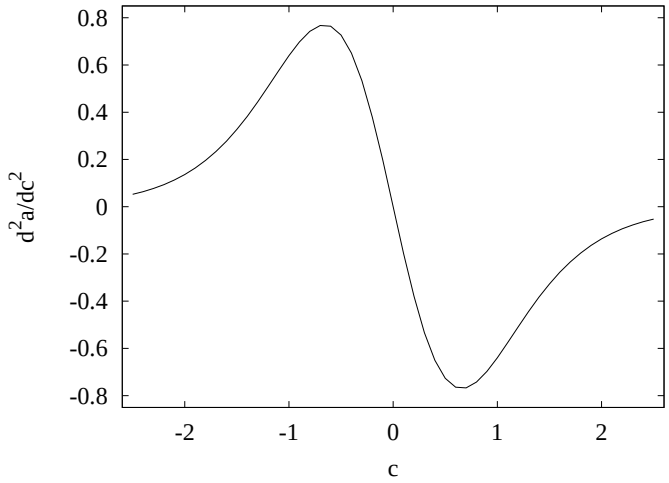


Figure 3.13: Hyperbolic tangent function second derivative.

$$\begin{aligned} a''(-1.55) &= -2\tanh(-1.55)(1-\tanh^2(-1.55)) \\ &= 0.30. \end{aligned}$$

Linear function second derivative

The second derivative of the linear function, $a'' : C \to 0$, with $C \subseteq \mathbb{R}$, is given by

$$a''(c) = 0. \tag{3.17}$$

The linear activation function second derivative is described in Figure 3.14.

Example 15 *If P is a logistic perceptron which produces a combination $c = -1.55$, then the activation second derivative is*

$$a''(-1.55) = 0$$

3.8 The Perceptron class

`Flood` includes the class Perceptron to represent the concept of perceptron neuron model.

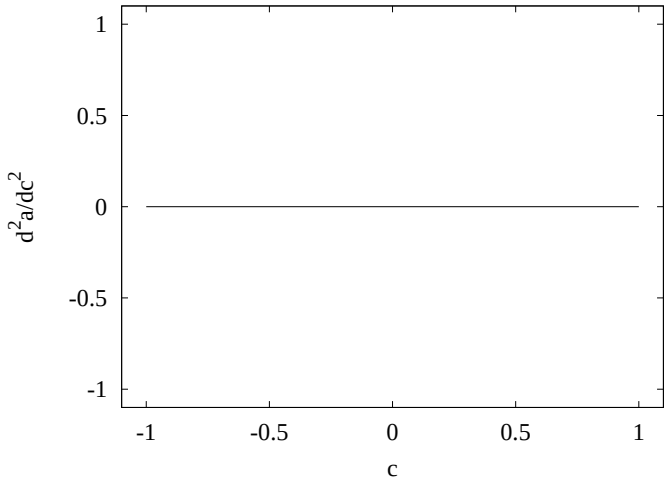


Figure 3.14: Linear second derivative.

Constructors

The Perceptron class implements a number of constructors with different arguments. The default activation function is the hyperbolic tangent.

The default constructor creates a perceptron object with zero inputs. The default activation function is hyperbolic tangent. It is called using

```
Perceptron p;
```

To construct a Perceptron object with a given number of inputs we use the following sentence

```
Perceptron p(3);
```

That objects has bias and synaptic weights values chosen at random from a normal distribution with mean 0 and standard deviation 1, and activation function set by default to hyperbolic tangent.

The following constructor builds a Perceptron object with a given number of inputs and with neuron parameters initialized to a given value,

```
Perceptron p(1, 0.0);
```

To create a neuron by loading its members from a file we use the file constructor in the following way

```
Perceptron p('Perceptron.dat');
```

The copy constructor builds a Perceptron object by copying all the members from another object of that class,

```
Perceptron p(2);
Perceptron q(p);
```

Members

That class contain the following members:
-The activation function.
-The number of inputs.
-The neuron's bias.
-The neuron's synaptic weights.

All that members are private, and get and set methods are implemented for each.

Methods

As it has been said, the Perceptron implements get and set methdos for every single member of that class.

We use the get_inputs_number method to obtain the number of inputs of a neuron,

```
Perceptron p(5);
int inputs_number = p.get_inputs_number();
```

Similarly, the get_bias and get_synaptic_weights methods return the bias and the synaptic weights of a Perceptron object, respectively

```
Perceptron p(1);
double bias = p.get_bias();
Vector<double> synaptic_weights = p.get_synaptic_weights();
```

The get_activation_function method returns the activation function of the neuron,

```
Perceptron p;
Perceptron::ActivationFunction activation_function
= p.get_activation_function();
```

We can set the number of inputs, bias, synaptic weights and activation function of a Perceptron object by writing

```
Perceptron p(2, 0.0);
p.set_inputs_number(5);
p.set_bias(1.0);
Vector<double> synaptic_weights(5, 3.1415927);
p.set_synaptic_weights(synaptic_weights);
p.set_activation_function(Perceptron::Linear);
```

There are several initialization methods for the neuron parameters. The folloing sentence initializes the bias and the synaptic weight values from a normal distribution with mean 1 and standard deviation 2,

```
Perceptron p(10);
p.initialize_normal(1.0, 2.0);
```

The calculate_combination computes the combination of the inputs and the neuron parameters,

```
Perceptron p(2, 0.0);
Vector<double> input(2, 0.0);
double combination = p.calculate_combination(input);
```

The method calculate_activation computes the activation of the neuron for a given combination value. For instance,

```
Perceptron p;
double activation = p.calculate_activation(1.0);
```

The calculate_output method implements the composition of the calculate_combination and the calculate_activation methods,

```
Perceptron p(4, 0.0);
Vector<double> input(2, 0.0);
double output = p.calculate_output(input);
```

In order to calculate the activation first and second derivatives of a neuron for a given combination we use the calculate_activation_derivative and the calculate_activation_second_derivative methods, respectively.

```
Perceptron p;
double activation_derivative
= p.calculate_activation_derivative(-0.5);
double activation_second_derivative
= p.calculate_activation_second_derivative(-0.5);
```

File format

A perceptron object can be serialized or deserialized to or from a data file which contains the member values. The file format of an object of the Perceptron class is of XML type.

```
<Flood version='3.0' class='Perceptron'>
<InputsNumber>
inputs_number
</InputsNumber>
<ActivationFunction>
activation_function
</ActivationFunction>
<Bias>
bias
```

```
</Bias>
<SynapticWeights>
synaptic_weight_1 ... synaptic_weight_n
</SynapticWeights>
<Display>
display
</Display>
```

Chapter 4

The multilayer perceptron

Perceptron neurons can be combined 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.

4.1 Network architecture

Neurons can be combined to form a neural network. The architecture of a neural network refers to the number of neurons, their arrangement and connectivity. Any network 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 [6].

Most neural networks, even biological neural networks, exhibit a layered structure. In this work layers are the basis to determine the architecture of a neural network [53]. 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.

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.

Feed-forward architectures contain no cycles, i.e., the architecture of a feed-forward neural network can then be represented as an acyclic graph.

Hence, neurons in a feed-forward neural network are grouped into a sequence of $h+1$ layers of neurons $L^{(1)}, \ldots, L^{(h)}, L^{(h+1)}$, so that neurons in any layer are connected only to neurons in the next layer.

The input layer consists of n external inputs and is not a layer of neurons; the hidden layers $L^{(1)}, \ldots, L^{(h)}$ contain $s^{(1)}, \ldots, s^{(h)}$ hidden neurons, respectively; and the output layer $L^{(h+1)}$ is composed of m output neurons.

Figure 4.1 shows the network architecture of a multilayer perceptron, with n inputs, h hidden layers with $s^{(i)}$ neurons, for $i = 1, \ldots, h$, and m neurons in the output layer. In this User's Guide, superscripts are used to identify layers.

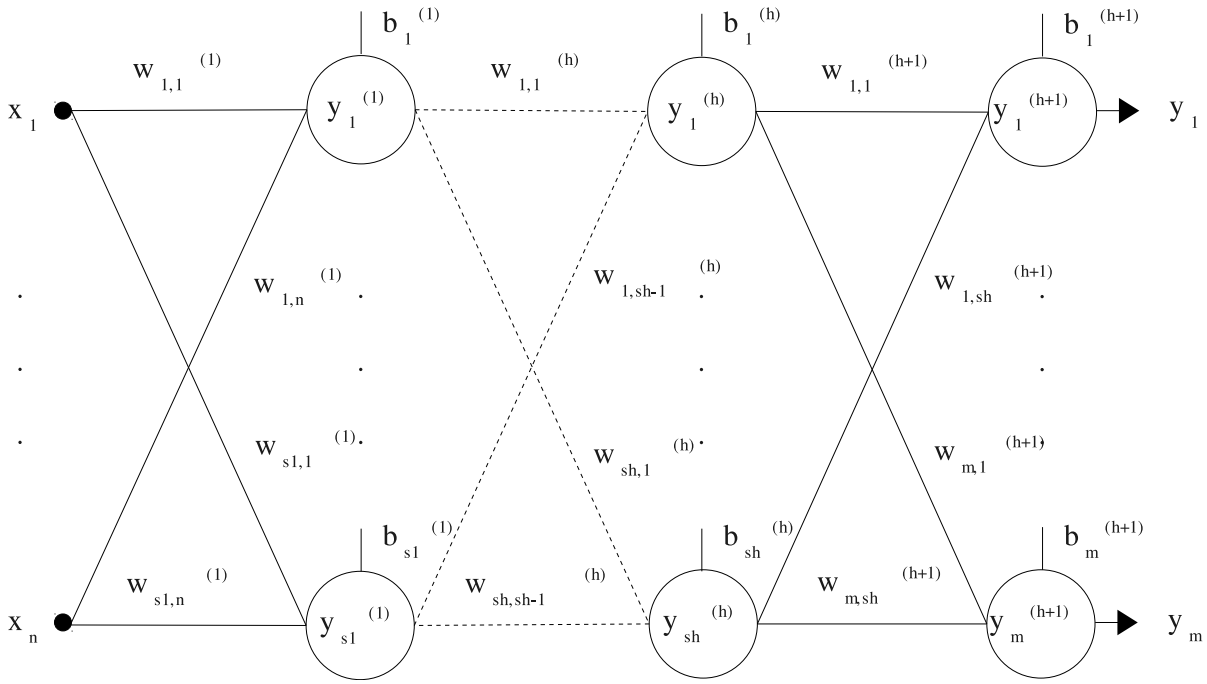


Figure 4.1: Multilayer perceptron.

Some basic information related to the input and output variables of a multilayer perceptron 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.

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 [53].

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 [8].

Example 16 *The multilayer perceptron of Figure 4.2 has number of inputs* $n = 5$*, number of hidden layers* $h = 2$*, size of first hidden layer* $s^{(1)} = 4$*, size of second hidden layer* $s^{(2)} = 6$ *and number of outputs* $m = 3$*.*

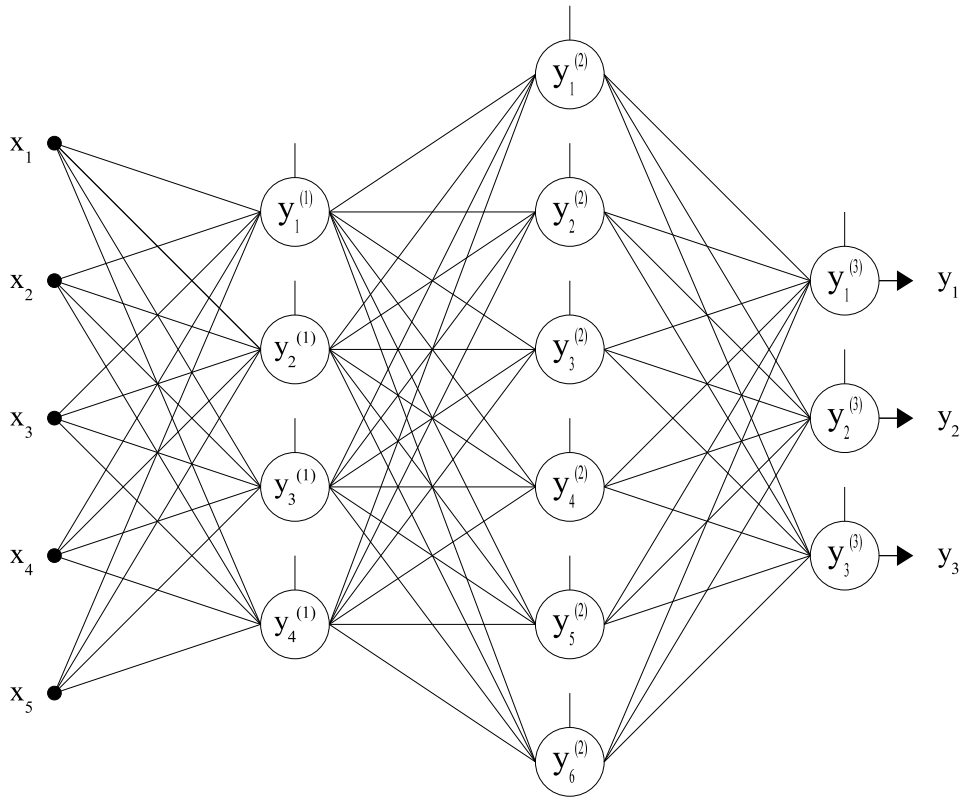


Figure 4.2: Multilayer perceptron example.

4.2 Multilayer perceptron parameters

Two types of parameters can be found in a multilayer percepton, neural parameters and independent parameters. The first group defines the output from the neural network for a given input. The second group provides some separate sort of information.

Neural parameters

The neural parameters of a multilayer perceptron involve the parameters of each perceptron in the network architecture.

A bias vector $\mathbf{b}$ is defined for each layer of neurons. The size of this vector is the layer size,

$$\mathbf{b}^{(L)} = \left(b_1^{(L)}, \ldots, b_{s^{(L)}}^{(L)} \right), \tag{4.1}$$

for $i = 1, \ldots, h+1$ and with $s^{(0)} = n$ and $s^{(h+1)} = m$.

Similarly, a weight matrix $\mathbf{W}$ can be considered for every layer in the neural network. The rows number is the layer size and the columns number is the layer inputs number,

$$\mathbf{W}^{(L)} = \begin{pmatrix} w_{1,1}^{(L)} & \cdots & w_{1,s^{(L-1)}}^{(L)} \\ \vdots & \ddots & \vdots \\ w_{s^{(L)},1}^{(L)} & \cdots & w_{s^{(L)},s^{(L-1)}}^{(L)} \end{pmatrix} \tag{4.2}$$

with $s^{(0)} = n$ and $s^{(h+1)} = m$.

The number of parameters in a given layer is the number of biases plus the number of synaptic weights,

$$d^{(L)} = s^{(L)}(1 + s^{(L-1)}). \tag{4.3}$$

The number of neural parameters in a multilayer perceptron is the sum of the number of parameters in each layer,

$$d^{(N)} = \sum_{i=1}^{h+1} d^{(i)}, \tag{4.4}$$

where the superscript (N) stands for neural parameters.

The neural parameters can be grouped together in a $d^{(N)}$-dimensional vector $\boldsymbol{\zeta}^{(N)} \in \mathbb{R}^{dN}$,

$$\boldsymbol{\zeta}^{(N)} = (\zeta_1^{(N)}, \ldots, \zeta_d^{(N)}). \tag{4.5}$$

We can express the norm of the neural parameters vector as

$$\|\boldsymbol{\zeta}^{(N)}\| = \sqrt{\sum_{i=1}^{dN} \zeta_i^{(N)2}}. \tag{4.6}$$

Example 17 *Consider the multilayer perceptron in Example 16, which as* 5 *inputs,* 4 *neurons in the first hidden layer,* 6 *neurons in the second hidden layer and* 3 *output neurons.*

The bias vector of the first hidden layer is of the form

$$\mathbf{b}^{(1)} = \left(b_1^{(1)}, \ldots, b_4^{(1)}\right),$$

and the synaptic weight matrix,

$$\mathbf{W}^{(1)} = \begin{pmatrix} w_{1,1}^{(1)} & \cdots & w_{1,5}^{(1)} \\ \vdots & \ddots & \vdots \\ w_{4,1}^{(1)} & \cdots & w_{4,5}^{(1)} \end{pmatrix}.$$

The number of parameters in the firs hiden layer is then

$$\begin{aligned} d^{(1)} &= 4(1+5) \\ &= 24. \end{aligned}$$

Similarly, the bias vector of the second hidden layer is

$$\mathbf{b}^{(2)} = \left(b_1^{(2)}, \ldots, b_6^{(2)}\right),$$

and the synaptic weight matrix of that layer

$$\mathbf{W}^{(2)} = \begin{pmatrix} w_{11}^{(2)} & \cdots & w_{14}^{(2)} \\ \vdots & \ddots & \vdots \\ w_{61}^{(2)} & \cdots & w_{64}^{(2)} \end{pmatrix}$$

In this way, the number of parameters in the second hidden layer is

$$\begin{aligned} d^{(2)} &= 6(1+4) \\ &= 30. \end{aligned}$$

The bias vector of the output layer is

$$\mathbf{b}^{(3)} = \left(b_1^{(3)}, \ldots, b_3^{(3)} \right),$$

and the synaptic weight matrix

$$\mathbf{W}^{(3)} = \begin{pmatrix} w_{11}^{(3)} & \cdots & w_{16}^{(3)} \\ \vdots & \ddots & \vdots \\ w_{31}^{(3)} & \cdots & w_{36}^{(3)} \end{pmatrix}.$$

Therefore, the number of parameters in the output layer is

$$\begin{aligned} d^{(3)} &= 3(1+6) \\ &= 21. \end{aligned}$$

The number of neural parameters is the sum biases and synaptic weights in the hidden and the output layers,

$$\begin{aligned} d^{(n)} &= 24 + 30 + 21 \\ &= 75. \end{aligned}$$

All the biases and synaptic weights can be grouped together in the neural parameters vector

$$\boldsymbol{\zeta}^{(N)} = (\zeta_1^{(N)}, \ldots, \zeta_{75}^{(N)}).$$

The norm of that vector can be computed as

$$\|\boldsymbol{\zeta}^{(N)}\| = \sqrt{\sum_{i=0}^{75} \zeta_i^{(N)2}}.$$

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 can be grouped together in a $d^{(I)}$-dimensional vector $\boldsymbol{\zeta}^{(I)} \in \mathbb{R}^{d^{(I)}}$

$$\boldsymbol{\zeta}^{(I)} = (\zeta_1^{(I)}, \ldots, \zeta_{d^{(I)}}^{(I)}). \tag{4.7}$$

To calculate the norm of the independent parameters vector we use the following expression

$$\|\boldsymbol{\zeta}^{(I)}\| = \sqrt{\sum_{i=1}^{d^{(I)}} \zeta_i^{(I)2}}. \tag{4.8}$$

Example 18 *Let MLP be a multilayer perceptron with* 0 *inputs,* 0 *hidden layers and* 0 *outputs. If we associate* 10 *independent parameters to that neural network, the independent parameters vector is*

$$\boldsymbol{\zeta}^{(I)} = (\zeta_1^{(I)}, \ldots, \zeta_{10}^{(I)}),$$

and the norm of that vector is

$$\|\boldsymbol{\zeta}^{(I)}\| = \sqrt{\sum_{i=1}^{10} \zeta_i^{(I)2}}.$$

Parameters

The total set of parameters $\boldsymbol{\zeta} \in \mathbb{R}^{d^{(N)}} \times \mathbb{R}^{d^{(I)}}$ in a multilayer perceptron is composed by the biases and synaptic weights and the independent parameters,

$$\boldsymbol{\zeta} = (\boldsymbol{\zeta}^{(N)}, \boldsymbol{\zeta}^{(I)}). \tag{4.9}$$

The number of parameters is then the number of neural parameters plus the number of independent parameters,

$$d = d^{(N)} + d^{(I)}. \tag{4.10}$$

The norm of the parameters vector is defined in the usual way,

$$\|\boldsymbol{\zeta}\| = \sqrt{\sum_{i=1}^{d} \zeta_i^2}. \tag{4.11}$$

Example 19 *Consider a multilayer perceptron with* 5 *inputs,* 2 *hidden layers with* 4 *and* 6 *neurons,* 3 *output neurons and* 10 *independent parameters.*

The number of neural parameters is 75*, and the number of independent parameters is* 10*. The parameters vector here is of the form*

$$\boldsymbol{\zeta} = (\zeta_1^{(N)}, \ldots, \zeta_{75}^{(N)}, \zeta_1^{(I)}, \ldots, \zeta_{10}^{(N)}).$$

The number of parameters of that neural network is

$$\begin{aligned} d &= 75 + 10 \\ &= 85. \end{aligned}$$

Finally, the parameters norm is given by the quantity

$$\|\boldsymbol{\zeta}^{(I)}\| = \sqrt{\sum_{i=1}^{85} \zeta_i^2}.$$

4.3 Layer combination function

The layer combination function $\mathbf{c} : X^{(L)} \to C^{(L)}$, with $X^{(L)} \subseteq \mathbb{R}^{s^{(L-1)}}$ and $C^{(L)} \subseteq \mathbb{R}^{s^{(L)}}$, takes an input vector to that layer to produce a combination vector, or net input vector. The layer combination function computes the combination of each perceptron,

$$c_i^{(L)} = b_i^{(L)} + \mathbf{w}_i^{(L)} \mathbf{x}^{(L)}, \tag{4.12}$$

for $i = 1, \ldots, s^{(L)}$. Here $b_i^{(L)}$ and $\mathbf{w}_i^{(L)}$ are the bias and synaptic weights of neuron i in layer L, respectively.

Example 20 *Consider a layer of perceptrons* L *with size* $s^{(L)} = 2$ *and number of inputs* $s^{(0)} = 3$*. Let the bias vector be* $\mathbf{b}^{(L)} = (0, 0)$ *and the synaptic*

weight matrix be $\mathbf{W}^{(L)} = (0.1\,0.2\,0.3;\ -0.1\ -0.2\quad -0.3)$. *If the input to that layer is* $\mathbf{x} = (1, 2, 3)$, *the combination will be*

$$\begin{aligned} c_1 &= 0 + 0.1 \cdot 1 + 0.2 \cdot 2 + 0.3 \cdot 3 \\ &= 1.4 \\ c_2 &= 0 - 0.1 \cdot 1 - 0.2 \cdot 2 - 0.3 \cdot 3 \\ &= -1.4. \end{aligned}$$

4.4 Layer activation function

The activation function of a layer in a multilayer perceptron, $\mathbf{a}^{(L)} : C^{(L)} \to A^{(L)}$, takes that layer's combination, $C^{(L)} \subseteq \mathbb{R}^{s^{(L)}}$, to produce a layer's activation , $A^{(L)} \subseteq \mathbb{R}^{s^{(L)}}$. The activation values are those computed by each single perceptron,

$$a_i^{(L)}(\mathbf{c}^{(L)}) = a_i(c_i^{(L)}), \tag{4.13}$$

for $i = 1, \ldots, s^{(L)}$.

Layer threshold function

This layer activation function $\mathbf{a}^{(L)} : C^{(L)} \to [0,1]^{s^{(L)}}$, with $C^{(L)} \subseteq \mathbb{R}^{s^{(L)}}$, computes the threshold function for each perceptron in that layer,

$$a_i^{(L)}(\mathbf{c}^{(L)}) = \begin{cases} 0 & c_i^{(L)} < 0, \\ 1 & c_i^{(L)} \geq 0, \end{cases} \tag{4.14}$$

for $i = 1, \ldots, s^{(L)}$.

Threshold activation functions are usually found in the output layer of a multilayer perceptron, and not in the hidden layers.

Example 21 *Consider a layer* L *of size* $s^{(L)} = 3$ *and with threshold activation function. If the combination of that layer is* $\mathbf{c}^{(L)} = (-1, 0.5, 1.5)$, *the activation will be*

$$\mathbf{a}^{(L)}(-1, 0.5, 1.5) = (0, 1, 1).$$

Layer symmetric threshold

The symmetric threshold layer activation function $\mathbf{a}^{(L)} : C^{(L)} \to [-1, 1]^{s^{(L)}}$, with $C^{(L)} \subseteq \mathbb{R}^{s^{(L)}}$ is given by

$$a_i^{(L)}(\mathbf{c}^{(L)}) = \begin{cases} -1 & c_i^{(L)} < 0, \\ 1 & c_i^{(L)} \geq 0, \end{cases} \tag{4.15}$$

for $i = 1, \ldots, s^{(L)}$.

As it happens with the threshold function, the symmetric threshold is usually found in the output layer, rather than in the hidden layers, of a multilayer perceptron.

Example 22 *Let L be a layer with 3 neurons and symmetric threshold activation function. The activation of that layer for a combination $\mathbf{c}^{(L)} = (-1, 0.5, 1.5)$ will be*

$$\mathbf{a}^{(L)}(-1, 0.5, 1.5) = (-1, 1, 1).$$

Layer logistic function

A layer's logistic function, $\mathbf{a}^{(L)} : C^{(L)} \to (0, 1)^{s^{(L)}}$, with $C^{(L)} \subseteq \mathbb{R}^{s^{(L)}}$, computes a logistic activation function for each layer's perceptron,

$$a_i^{(L)}(\mathbf{c}^{(L)}) = \frac{1}{1 + \exp\left(-c_i^{(L)}\right)}. \tag{4.16}$$

for $i = 1, \ldots, s^{(L)}$.

Logistic activation functions are usually found in the hidden layers of a multilayer perceptron. Nevertheless they can be also found in the output layer.

Example 23 *Consider a layer of perceptrons L with size $s^{(L)} = 3$ and logistic activation function. The combination of that layer is $\mathbf{c}^{(L)} = (-1, 0.5, 1.5)$. Then, the activation of that layer will be*

$$\mathbf{a}^{(L)}(-1, 0.5, 1.5) = (0.269, 0.622, 0.818).$$

Layer hyperbolic tangent

The hyperbolic tangent layer activation function, $\mathbf{a}^{(L)} : C^{(L)} \to (-1,1)^{s^{(L)}}$, with $C^{(L)} \subseteq \mathbb{R}^{s^{(L)}}$, is very symilar to the logistic, but the images of that two functions are different. For this case it is given by

$$a_i^{(L)}(\mathbf{c}^{(L)}) = \tanh\left(c_i^{(L)}\right). \tag{4.17}$$

Hyperbolic tangent activation functions are usually found in the hidden layers of a multilayer perceptron. although sometimes they are also found in the hidden layers.

Example 24 *Let L be am hyperbolic tangent layer of size $s^{(L)} = 3$. If the combination of that layer is $\mathbf{c}^{(L)} = (-1, 0.5, 1.5)$, then activation will be*

$$\mathbf{a}^{(L)}(-1, 0.5, 1.5) = (-0.762, 0.462, 0.905).$$

Layer linear function

Finally, a layer's linear activation function, $\mathbf{a}^{(L)} : C^{(L)} \to A^{(L)}$, with $C^{(L)} \subseteq \mathbb{R}^{s^{(L)}}$ and $A^{(L)} \subseteq \mathbb{R}^{s^{(L)}}$, is simply given by

$$a_i^{(L)}(\mathbf{c}^{(L)}) = c_i^{(L)}, \tag{4.18}$$

for $i = 1, \ldots, s^{(L)}$.

Linear activation functions are usually found in the output layer of neural networks.

Example 25 *Consider a layer of perceptrons called L with size $s^{(L)} = 3$ and linear activation function. The activation for a combination $\mathbf{c}^{(L)} = (-1, 0.5, 1.5)$ is*

$$\mathbf{a}^{(L)}(-1, 0.5, 1.5) = (-1, 0.5, 1.5).$$

4.5 Layer output function

The output function of a layer $\mathbf{y} : X^{(L)} \to Y^{(L)}$, with $X^{(L)} \subseteq \mathbb{R}^{s^{(L-1)}}$ and $Y^{(L)} \subseteq \mathbb{R}^{s^{(L)}}$ is given by the output function of each individual neuron,

$$y_i^{(L)} = a_i^{(L)}(b_i^{(L)} + \mathbf{w}_i^{(L)} \cdot \mathbf{x}^{(L)}), \tag{4.19}$$

for $i = 1, \ldots, s^{(L)}$.

Example 26 *Let L be a layer of 2 hyperbolic tangent perceptrons with 3 inputs. Let the bias vector be $\mathbf{b}^{(L)} = (0, 0)$, and the synaptic weight matrix be $\mathbf{W}^{(L)} = (0.1\, 0.2\, 0.3;\ -0.1\ -0.2\ -0.3)$. It the input to that layer is $\mathbf{x}^{(L)} = (-2, -1, 0, 1, 2)$, the output from it will be*

$$\begin{aligned} y_1^{(L)} &= \tanh(0 + 0.1 \cdot 1 + 0.2 \cdot 2 + 0.3 \cdot 3) \\ &= 0.885, \\ y_2^{(L)} &= \tanh(0 - 0.1 \cdot 1 - 0.2 \cdot 2 - 0.3 \cdot 3) \\ &= -0.885. \end{aligned}$$

4.6 Multilayer perceptron function

In Section 3.5 we considered the space of functions that a perceptron neuron model can define. As it happens with a single perceptron, a multilayer perceptron neural network may be viewed as a parameterized function space V from an input $X \subset \mathbb{R}^n$ to an output $Y \subset \mathbb{R}^m$. Elements of V are of the form $\mathbf{y} : X \to Y$. They are parameterized by the neural parameters, which can be grouped together in a d-dimensional vector $\boldsymbol{\zeta} = (\zeta_1, \ldots, \zeta_d)$. The dimension of the function space V is therefore d.

We can write down the analytical expression for the elements of the function space which the multilayer perceptron shown in Figure 4.1 can define [8].

For the first hidden layer $L^{(1)}$, the combination function is obtained by adding to the biases the dot product of the synaptic weights and the inputs, to give

$$\mathbf{c}^{(1)} = \mathbf{b}^{(1)} + \mathbf{W}^{(1)} \cdot \mathbf{x}. \tag{4.20}$$

The output of that layer is obtained transforming the combination with an activation function $\mathbf{a}^{(1)}$,

$$\mathbf{y}^{(1)} = \mathbf{a}^{(1)}\left(\mathbf{c}^{(1)}\right). \tag{4.21}$$

Similarly, for the last hidden layer, $L^{(h)}$, the combination function is given by

$$\mathbf{c}^{(h)} = \mathbf{b}^{(h)} + \mathbf{W}^{(h)} \cdot \mathbf{y}^{(h-1)}. \tag{4.22}$$

The output of that layer is found by using an activation $\mathbf{a}^{(h)}$,

$$\mathbf{y}^{(h)} = \mathbf{a}^{(h)} \left(\mathbf{c}^{(h)} \right), \tag{4.23}$$

The outputs of the neural network are obtained by transforming the outputs of the last hidden layer by the neurons in the output layer $L^{(h+1)}$. Thus, the combination of the output layer is of the form

$$\mathbf{c}^{(h+1)} = \mathbf{b}^{(h+1)} + \mathbf{W}^{(h+1)} \cdot \mathbf{y}^{(h)} \tag{4.24}$$

The output of the output layer is obtained transforming the combination of that layer with an activation $\mathbf{a}^{(h+1)}$ to give

$$\mathbf{y}^{(h+1)} = \mathbf{a}^{(h+1)} \left(\mathbf{c}^{(h+1)} \right), \tag{4.25}$$

Combining Equations (4.20), (4.21), (4.22), (4.23), (4.24) and (4.25), we obtain an explicit expression for the function represented by the neural network diagram in Figure 4 of the form

$$\mathbf{y} = \mathbf{a}^{(h+1)} \left(\mathbf{b}^{(h+1)} + \mathbf{W}^{(h+1)} \cdot \mathbf{a}^{(h)} \left(\mathbf{b}^{(h)} + \mathbf{W}^{(h)} \mathbf{a}^{(h-1)} \left(\cdots \mathbf{a}^{(1)} \left(\mathbf{b}^{(1)} + \mathbf{W}^{(1)} \cdot \mathbf{x} \right) \right) \right) \right) \tag{4.26}$$

In that way, the multilayer perceptron functions are represented in terms of the composition of the layer output functions,

$$\mathbf{y} = \mathbf{y}^{(h+1)} \circ \mathbf{y}^{(h)} \circ \cdots \circ \mathbf{y}^{(1)}. \tag{4.27}$$

Figure 4.3 is an activity diagram of how the information is propagated in the multilayer perceptron. As there is no recurrence, that is forward-propagated.

In this way, the multilayer perceptron can be considered as a function of many variables composed by superposition and addition of functions of one variable. Distinct activation functions cause distinct families of functions which a multilayer perceptron can define. Similarly, distinct sets of neural parameters cause distinct elements in the function space which a specific multilayer perceptron defines.

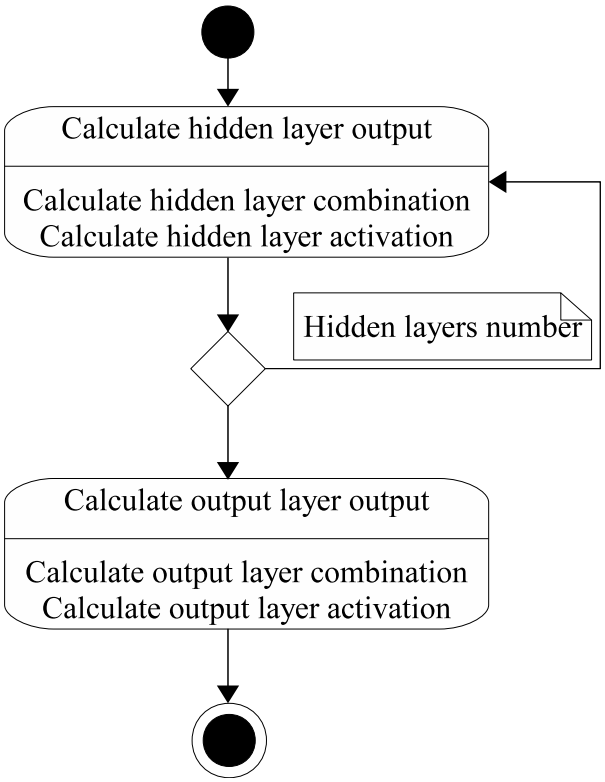


Figure 4.3: Forward-propagation in the multilayer perceptron.

Example 27 *Let MLP be a multilayer perceptron with network architecture $(1, 1, 1)$ and activation functions $(tanh, linear)$. The bias and synaptic weights of the hidden and the output layers are $\mathbf{b}^{(1)} = (1)$, $\mathbf{W}^{(1)} = (2)$, $\mathbf{b}^{(2)} = (3)$ and $\mathbf{W}^{(2)} = (4)$, respectively. Let apply an input $\mathbf{x} = -0.5$ to the neural network. The output of the hidden layer is*

$$\begin{aligned} y^{(1)} &= \tanh\left(1 + 2 \cdot -0.5\right) \\ &= 0, \end{aligned}$$

and the output of the neural network is

$$\begin{aligned} y &= 3 + 4 \cdot 0 \\ &= 3. \end{aligned}$$

4.7 Universal approximation

A multilayer perceptron with as few as one hidden layer of sigmoid neurons and an output layer of linear neurons provides a general framework for approximating any function from one finite dimensional space to another up to any desired degree of accuracy, provided sufficiently many hidden neurons are available. In this sense, multilayer perceptron networks are a class of universal approximators [27].

A detailed statement of the universal approximation theorem for the multilayer perceptron is out of the scope of this work, so that it is not included here. The interested reader can find this theorem, as well as its demonstration, in [27].

The universal approximation capability of the multilayer perceptron implies that any lack of success in an application must arise from a wrong number of hidden neurons, the lack of the objective functional or inadequate training.

4.8 Scaling and unscaling

Scaling might be useful or necessary under certain circumstances, e.g. when variables spam different ranges. In a multilayer perceptron scaling might be applied to the input variables, the output variables or the independent parameters.

Inputs and outputs scaling

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.

There are several scaling or processing methods. Two of the most used are the mean and standard deviation and the minimum and maximum methods.

The input scaling function $\overline{\mathbf{x}} : X \to \overline{X}$, with $X, \overline{X} \subseteq \mathbb{R}^n$ processes the inputs to the neural network so that they are of order zero. The most important input scaling methods are the meand and standard deviation and the minimum and maximum methods.

With the mean and standard deviation method the inputs are scaled so that they will have mean $\boldsymbol{\mu} = 0$ and standard deviation $\boldsymbol{\sigma} = 1$,

$$\overline{\mathbf{x}} = \frac{\mathbf{x} - \boldsymbol{\mu}^{(0)}}{\boldsymbol{\sigma}^{(0)}}, \tag{4.28}$$

where $\mathbf{x}$ are the inputs, $\overline{\mathbf{x}}$ are the scaled inputs, and $\boldsymbol{\mu}^{(0)}$ and $\boldsymbol{\sigma}^{(0)}$ are an estimation of the mean and the standard deviation of the input variables, respectively.

Note that inputs whose standard deviation is zero cannot be scaled.

Example 28 *Let MLP be a multilayer perceptron with $n = 3$ inputs. The mean and the standard deviation of the input variables are $\boldsymbol{\mu}^{(0)} = (-1, 0, 1)$ and $\boldsymbol{\sigma}^{(0)} = (2, 1, 4)$, respectively. The scaled input for an input $\mathbf{x} = (0, 0, 0)$ is*

$$\begin{aligned} \overline{\mathbf{x}} &= \frac{(0,0,0) - (-1,0,1)}{(2,1,4)} \\ &= (0.5, 0, 0.25). \end{aligned}$$

The minimum and maximum scaling function processes the inputs so that their minimum and maximum values are $\mathbf{min} = -1$ and $\mathbf{max} = 1$,

$$\overline{\mathbf{x}} = 2\frac{\mathbf{x} - \mathbf{min}^{(0)}}{\mathbf{max}^{(0)} - \mathbf{min}^{(0)}} - 1, \tag{4.29}$$

where $\mathbf{x}$ is the input, $\overline{\mathbf{x}}$ is the scaled input, and $\mathbf{min}^{(0)}$ and $\mathbf{max}^{(0)}$ are an estimation of the minimum and the maximum values of the input variables, respectively.

Note that variables whose minimum and maximum values are equal cannot be scaled.

Example 29 *Consider a multilayer perceptron MLP with number of inputs $n = 3$. Let the minimum and maximum of the input variables be $\mathbf{min}^{(0)} = (-1, -2, -3)$ and $\mathbf{max}^{(0)} = (2, 4, 6)$. The scaled input for an input $\mathbf{x} = (0, 0, 0)$ is*

$$\begin{aligned} \overline{\mathbf{x}} &= 2\frac{(0,0,0) - (-1,-2,-3)}{(2,4,6) - (-1,-2,-3)} - 1 \\ &= (-0.333, -0.333, -0.333). \end{aligned}$$

The output unscaling function $\overline{\mathbf{y}} : Y \rightarrow \overline{Y}$, with $Y, \overline{Y} \subseteq \mathbb{R}^m$ processes the scaled outputs from the neural network to produce the unscaled outputs, which are in the original units. The most important output unscaling methods are the mean and standard deviation and the minimum and maximum.

The mean and standard deviation method unscales the outputs in the following manner,

$$\overline{\mathbf{y}} = \boldsymbol{\mu}^{(h+1)} + \boldsymbol{\sigma}^{(h+1)}\mathbf{y}, \tag{4.30}$$

where $\mathbf{y}$ is the output, $\overline{\mathbf{y}}$ is the unscaled output, and $\boldsymbol{\mu}^{(h+1)}$ and $\boldsymbol{\sigma}^{(h+1)}$ are an estimation of the mean and the standard deviation of the output variables, respectively.

Example 30 *Let MLP be a multilayer perceptron with $m = 2$ outputs . The mean and standard deviation of the output variables are $\boldsymbol{\mu}^{(h+1)} = (-1, 2)$ and $\boldsymbol{\sigma}^{(h+1)} = (-2, 3)$, respectively. The output from the neural network for a scaled output $\mathbf{y} = (-1, 1)$ is*

$$\begin{aligned} \overline{\mathbf{y}} &= (-1, 2) + (-2, 3)(-1, 1) \\ &= (1, 5). \end{aligned}$$

The minimum and maximum unscaling function takes the scaled output from the neural network to produce the output

$$\overline{\mathbf{y}} = 0.5\,(\mathbf{y} + 1)\left(\mathbf{max}^{(h+1)} - \mathbf{min}^{(h+1)}\right) + \mathbf{min}^{(h+1)}, \tag{4.31}$$

where $\mathbf{y}$ are the outputs, $\overline{\mathbf{y}}$ are the unscaled outputs, and $\mathbf{min}^{(h+1)}$ and $\mathbf{max}^{(h+1)}$ are an estimation of the minimum and the maximum values of the output variables, respectively.

Example 31 *Consider a multilayer perceptron MLP with outputs number $m = 1$, output variables minimum $\mathbf{min}^{(h+1)} = (-1000)$ and output variables maximum $\mathbf{max}^{(h+1)} = (1000)$. If the scaled output from the neural network is $\mathbf{y} = (0.1)$, the unscaled output will be*

$$\begin{aligned} \overline{\mathbf{y}} &= 0.5\,((0.1) + 1)\,((1000) - (-1000)) + (-1000) \\ &= (100). \end{aligned}$$

Independent parameters scaling

As it happens with the input and output variables, the independent parameters are usually scaled and unscaled

The independent parameters scaling function $\overline{\boldsymbol{\zeta}}^{(I)} : Z \to \overline{Z}$, with $Z, \overline{Z} \subseteq \mathbb{R}^{d^{(I)}}$ processes the independent parameters so that they are of order zero.

The mean and standard deviation scaling function for the independent parameters is given by

$$\overline{\boldsymbol{\zeta}}^{(I)} = \frac{\boldsymbol{\zeta}^{(I)} - \boldsymbol{\mu}^{(I)}}{\boldsymbol{\sigma}^{(I)}}. \tag{4.32}$$

where $\boldsymbol{\zeta}^{(I)}$ are the unscaled independent parameters, $\overline{\boldsymbol{\zeta}}^{(I)}$ are the scaled independent parameters, $\boldsymbol{\mu}^{(I)}$ are the mean values of the independent parameters and $\boldsymbol{\sigma}^{(I)}$ are the standard deviation values of the independent parameters.

The minimum and maximum method is of the form

$$\overline{\boldsymbol{\zeta}}^{(I)} = 2\frac{\boldsymbol{\zeta}^{(I)} - \mathbf{min}^{(I)}}{\mathbf{max}^{(I)} - \mathbf{min}^{(I)}} - 1. \tag{4.33}$$

where $\boldsymbol{\zeta}^{(I)}$ are the unscaled independent parameters, $\overline{\boldsymbol{\zeta}}^{(I)}$ are the scaled independent parameters, $\mathbf{min}^{(I)}$ are the minimum values of the independent parameters and $\mathbf{max}^{(I)}$ are the maximum values of the independent parameters.

The independent parameters unscaling function $\overline{\boldsymbol{\zeta}}^{(I)} : Z \to \overline{Z}$, with $Z, \overline{Z} \subseteq \mathbb{R}^{d^{(I)}}$ postprocesses the scaled independent parameters to obtain values in the original ranges.

The mean and standard deviation method is

$$\overline{\boldsymbol{\zeta}}^{(I)} = \boldsymbol{\mu}^{(I)} + \boldsymbol{\sigma}^{(I)}\boldsymbol{\zeta}^{(I)}, \tag{4.34}$$

where $\boldsymbol{\zeta}^{(I)}$ and $\overline{\boldsymbol{\zeta}}^{(I)}$ are the unscaled and scaled independent parameteres and $\boldsymbol{\mu}^{(I)}$ and $\boldsymbol{\sigma}^{(I)}$ are the independent parameters mean and standard deviation.

Finally, the minimum and maximum unscaling function for the independent parameters is

$$\overline{\boldsymbol{\zeta}}^{(I)} = 0.5(\boldsymbol{\zeta}^{(I)} + 1\left(\mathbf{max}^{(I)} - \mathbf{min}^{(I)}\right) + \mathbf{min}^{(I)}, \tag{4.35}$$

where $\mathbf{min}^{(I)}$ and $\mathbf{max}^{(I)}$ are the independent parameters minimum and maximum, respectively.

4.9 Boundary conditions

If some outputs are specified for given inputs, then the problem is said to include boundary conditions. A boundary condition between some input $\mathbf{x} = \mathbf{a}$ and some output $\mathbf{y} = \mathbf{y_a}$ is written $\mathbf{y}(\mathbf{a}) = \mathbf{y_a}$.

In order to deal with boundary conditions the output from the neural network can be post-processed with the boundary conditions function $\hat{\mathbf{y}} : Y \to \hat{Y}$, with $Y, \hat{Y} \subseteq \mathbb{R}^m$, defined by

$$\hat{\mathbf{y}} = \boldsymbol{\varphi}_0(\mathbf{x}) + \boldsymbol{\varphi}_1(\mathbf{x})\mathbf{y}, \tag{4.36}$$

where $\mathbf{y}$ is the raw output and $\hat{\mathbf{y}}$ is the output satisfying the boundary conditions. The function $\boldsymbol{\varphi}_0(\mathbf{x})$ is called a particular solution term and the function $\boldsymbol{\varphi}_1(\mathbf{x})$ is called an homogeneous solution term.

The particular solution term, $\boldsymbol{\varphi}_0 : X \to \Phi_0$, with $X \in \mathbb{R}^n$ and $\Phi_0 \in \mathbb{R}^m$ must hold $\boldsymbol{\varphi}_0(\mathbf{a}) = \mathbf{y_a}$ if there is a condition $\mathbf{y}(\mathbf{a}) = \mathbf{y_a}$. The homogeneous solution term $\boldsymbol{\varphi}_1 : X \to \Phi_1$, with $X \in \mathbb{R}^n$ and $\Phi_1 \in \mathbb{R}^m$ must hold $\boldsymbol{\varphi}_1(\mathbf{a}) = \mathbf{0}$ if there is a condition $\mathbf{y}(\mathbf{a}) = \mathbf{y_a}$.

It is easy to see that the approach above makes all the elements of the function space to satisfy the boundary conditions. Please note that the expressions of the particular and homogeneous solution terms depend on the problem at hand. The particular and homogeneous solution terms might be difficult to derive if the number of input and output variables is high and the number of boundary conditions is also high.

Example 32 *Consider a multilayer perceptron MLP with one input, $n = 1$, one output, $m = 1$, and one boundary condition, $y(a) = y_a$. A possible set of particular and homogeneous solution terms could be*

$$\begin{aligned} \varphi_0(x) &= a, \\ \varphi_1(x) &= x - a, \end{aligned}$$

which indeed satisfy $\varphi_0(a) = y_a$ and $\varphi_1(a) = 0$. The output from that neural network is then

$$\hat{y} = a + (x - a)y,$$

which holds $y(a) = y_a$.

Example 33 *Let MLP be a multilayer perceptron with inputs number $n = 1$, outputs number $m = 1$ and boundary conditions $y(a) = y_a$ and $y(b) = y_b$. The particular and homogeneous terms could be*

$$\begin{aligned} \varphi_0(x) &= y_a + \frac{y_b - y_a}{b - a} x, \\ \varphi_1(x) &= (x - a)(x - b). \end{aligned}$$

which satisfy $\varphi_0(a) = y_a$, $\varphi_0(b) = y_b$, $\varphi_1(a) = 0$ and $\varphi_1(b) = 0$. In that way the output is given by

$$\hat{y} = y_a + \frac{y_b - y_a}{b - a} x + (x - a)(x - b)y.$$

4.10 Lower and upper bounds

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

Output variables bounds

If some output variables are restricted to fall in some interval, then the problem is said to have lower and upper bounds in the output variables. An easy way to treat lower and upper bounds is to post-process the outputs from the neural network with the bounding function $\widetilde{\mathbf{y}} : Y \to \widetilde{Y}$, with $Y, \widetilde{Y} \subseteq \mathbb{R}^m$, given by

$$\widetilde{\mathbf{y}} = \begin{cases} \mathbf{l}^{(h+1)}, & \mathbf{y} < \mathbf{l}^{(h+1)}, \\ \mathbf{y}, & \mathbf{l}^{(h+1)} \leq \mathbf{y} \leq \mathbf{u}^{(h+1)}, \\ \mathbf{u}^{(h+1)}, & \mathbf{y} > \mathbf{u}^{(h+1)}, \end{cases} \tag{4.37}$$

where $\mathbf{y}$ is the unbounded output and $\widetilde{\mathbf{y}}$ is the bounded output. The vectors $\mathbf{l}^{(h+1)}$ and $\mathbf{u}^{(h+1)}$ represent the lower and upper bounds of the output variables, respectively.

Example 34 *Let MLP be a multilayer perceptron with $m = 3$ outputs. Let also $\mathbf{l}^{(h+1)} = (0, 0, 0)$ and $\mathbf{u}^{(h+1)} = (\infty, 1, \infty)$ be the lower and upper bounds of the outpus. The bounded output for an unbounded output $\mathbf{y} = (-1, 0, 1)$ is*

$$\widetilde{\mathbf{y}} = (0, 0, 1).$$

Independent parameters bounds

Similarly, if some independent parameters are bounded they can be post-processed with the bounding function $\widetilde{\boldsymbol{\zeta}}^{(I)} : Z \to \widetilde{Z}$, with $Z, \widetilde{Z} \subseteq \mathbb{R}^{d^{(I)}}$,

$$\widetilde{\boldsymbol{\zeta}}^{(I)} = \begin{cases} \mathbf{l}^{(I)}, & \boldsymbol{\zeta}^{(I)} < \mathbf{l}^{(I)}, \\ \boldsymbol{\zeta}^{(I)}, & \mathbf{l}^{(I)} \leq \boldsymbol{\zeta}^{(I)} \leq \mathbf{u}^{(I)}, \\ \mathbf{u}^{(I)}, & \boldsymbol{\zeta}^{(I)} > \mathbf{u}^{(I)}, \end{cases} \tag{4.38}$$

where $\boldsymbol{\zeta}^{(I)}$ are the unbounded independent parameters and $\widetilde{\boldsymbol{\zeta}}^{(I)}$ are the bounded independent parameters. $\mathbf{l}^{(I)}$ and $\mathbf{u}^{(I)}$ represent the lower and upper bounds, respectively.

Example 35 *Let MLP be a multilayer perceptron with $d^{(I)} = 2$ independent parameters. Let also $\mathbf{l}^{(I)} = (-1, -1)$ and $\mathbf{u}^{(h+1)} = (1, 1)$ be the lower and upper bounds of the independent parameters. The bounded output for an unbounded output $\zeta^{(I)} = (-2, 2)$ is*

$$\widetilde{\boldsymbol{\zeta}}^{(I)} = (-1, 1).$$

4.11 Multilayer perceptron activity diagram

Following the contents of this chapteer, an activity diagram for the input-output process in a multilayer perceptron can be drawn as in Figure 4.4. Here the input to the neural network is first scaled using the scaling function. The scaled input is forward propagated through the layers of perceptrons to obtain the scaled output. That is then unscaled by means of the unscaling function. If some boundary conditions must be satisfied, that unscaled output is postprocessed with the particular and homogeneous solution terms. That unbounded output is finally bounded to obtain the proper output.

4.12 Layer activation derivative

Up to now the function represented by a multilayer perceptron has been considered. In this section and some succeeding ones the derivatives of that function shall be studied.

The activation derivative of a layer of perceptrons, $\mathbf{a}'^{(L)} : C^{(L)} \to A'^{(L)}$, with $C^{(L)}, A'^{(L)} \subseteq \mathbb{R}^{s^{(L)}}$ defines the derivative of each neuron's activation with respect to the combination,

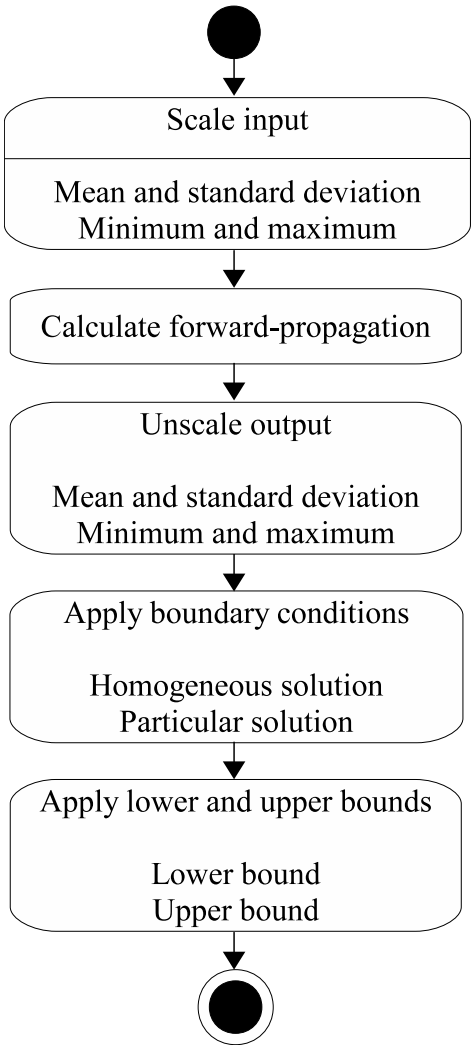


Figure 4.4: Input-output process in the multilayer perceptron.

$$a_i'^{(L)} = \frac{da_i^{(L)}}{dc_i^{(L)}}, \tag{4.39}$$

for $i = 1, \ldots, s^{(L)}$.

Layer threshold function derivative

The threshold function is not derivable at the point $c = 0$.

Layer symmetric threshold derivative

The symmetric threshold activation function is neither derivable at $c = 0$.

Example 36

Layer logistic function derivative

This layer activation function derivative, $\mathbf{a}' : C \to (0, 0.25]^{s^{(L)}}$, with $C \subseteq \mathbb{R}^{s^{(L)}}$, is given by

$$a_i'^{(L)}(\mathbf{c}^{(L)}) = \frac{\exp\left(-c_i^{(L)}\right)}{(1 + \exp\left(-c_i^{(L)}\right))^2}, \tag{4.40}$$

for $i = 1, \ldots, s^{(L)}$.

Example 37 *Let L be a layer of logistic perceptrons and let $s^{(L)} = 2$ be the size of that layer. The activation derivative for a combination $\mathbf{c}^{(L)} = (-5, 5)$ is*

$$\mathbf{a}^{(L)} = (0.006, 0.006).$$

Layer hyperbolic tangent derivative

The hyperbolic function activation derivative of a layer of perceptrons, $\mathbf{a}' : C \to (0, 1]^{s^{(L)}}$, with $C \subseteq \mathbb{R}^{s^{(L)}}$, is given by

$$a_i'^{(L)}(\mathbf{c}^{(L)}) = 1 - \tanh^2\left(c_i^{(L)}\right), \tag{4.41}$$

for $i = 1, \ldots, s^{(L)}$.

Example 38 *A layer L with size $s^{(L)} = 2$ and hyperbolic tangent activation function with combiantion $\mathbf{c}^{(L)} = (-5, 5)$ has the following activation*

$$\mathbf{a}'^{(L)} = (1.81 \cdot 10^{-4}, 1.81 \cdot 10^{-4}).$$

Layer linear function derivative

The layer linear function derivative $\mathbf{a}' : C \to Y'$, with $C^{(L)}, A'^{(L)} \subseteq \mathbb{R}^{s^{(L)}}$, is given by

$$a_i'^{(L)}(\mathbf{c}^{(L)}) = c_i^{(L)}, \tag{4.42}$$

for $i = 1, \ldots, s^{(L)}$.

Example 39 *Consider a layer L with size $s^{(L)} = 2$ and linear activation. If the combination of that layer is $\mathbf{c}^{(L)} = (-5, 5)$, then the activation derivative is*

$$\mathbf{a}'^{(L)} = (1, 1).$$

4.13 Layer Jacobian matrix

Consider the derivatives of the outputs of a layer with respect to its inputs. That derivatives can be grouped together in the layer Jacobian matrix, which is given by

$$\mathbf{J}^{(L)} = \begin{pmatrix} \dfrac{\partial y_1^{(L)}}{\partial x_1^{(L)}} & \cdots & \dfrac{\partial y_1^{(L)}}{\partial x_{s^{(L-1)}}^{(L)}} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial y_{s^{(L)}}^{(L)}}{\partial x_1^{(L)}} & \cdots & \dfrac{\partial y_{s^{(L)}}^{(L)}}{\partial x_{s^{(L-1)}}^{(L)}} \end{pmatrix}, \tag{4.43}$$

where each such derivative is evaluated with all other inputs held fixed.

The element (i, j) of that matrix can be evaluated as

$$J_{ij}^{(L)} = a_i'^{(L)} w_{ij}^{(L)}. \tag{4.44}$$

4.14 Multilayer perceptron Jacobian matrix

The Jacobian matrix for the multilayer perceptron groups together the derivatives of the neural network outputs with respect to the neural network inputs,

$$\mathbf{Jy}(\mathbf{x}) = \begin{pmatrix} \dfrac{\partial y_1}{\partial x_1} & \cdots & \dfrac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial y_m}{\partial x_1} & \cdots & \dfrac{\partial y_m}{\partial x_n} \end{pmatrix}, \tag{4.45}$$

where each such derivative is evaluated with all other inputs held fixed.

Please note that the use of scaling and unscaling, boundary conditions and lower and upper bounds will affect the derivative values of the Jacobian matrix.

The Jacobian matrix can be evaluated either by using a forward-propagation procedure, or by means of numerical differentiation.

Forward-propagation for the Jacobian matrix

The chain rule for the derivative provides us with a direct expression of the Jacobian matrix for the multilayer perceptron as

$$\mathbf{J} = \mathbf{J}^{(h+1)} \cdot \mathbf{J}^{(h)} \cdot \ldots \cdot \mathbf{J}^{(1)}, \tag{4.46}$$

where $\mathbf{J}^{(L)}$ is the Jacobian matrix of layer L.

Numerical differentiation for the Jacobian matrix

The Jacobian matrix for the multilayer perceptron can also be evaluated using numerical differentiation [8]. This can be done by perturbing each input in turn, and approximating the derivatives by using forward differences,

$$\frac{\partial y_j}{\partial x_i} = \frac{y_j(x_i + \epsilon) - y_j(x_i)}{\epsilon} + \mathcal{O}(\epsilon), \tag{4.47}$$

for $i = 1, \ldots, n$, $j = 1, \ldots, m$ and for some small numerical value of ϵ.

The accuracy of the forward differences method can be improved significantly by using central differences of the form

$$\frac{\partial y_j}{\partial x_i} = \frac{y_j(x_i + \epsilon) - y_j(x_i - \epsilon)}{2\epsilon} + \mathcal{O}(\epsilon^2), \tag{4.48}$$

also for $i = 1, \ldots, n$, $j = 1, \ldots, m$ and for some small numerical value of ϵ.

4.15 Layer activation second derivative

We can also consider the activation second derivative of a layer of perceptrons, $\mathbf{a}''^{(L)} : C^{(L)} \to A''^{(L)}$, with $C^{(L)}, A''^{(L)} \subseteq \mathbb{R}^{s^{(L)}}$. That function is defined by

$$a_i''^{(L)}(\mathbf{c}^L) = \frac{d^2 a_i^{(L)}}{dc_i^{(L)2}} \tag{4.49}$$

for $i = 1, \ldots, s^{(L)}$.

Layer threshold function second derivative

The threshold function is not derivable at the point $c = 0$.

Layer symmetric threshold second derivative

The symmetric threshold activation function is neither derivable at the point $c = 0$.

Layer logistic function second derivative

The second derivative of this activation function $\mathbf{a}''^{(L)} : C^{(L)} \to C''^{s^{(L)}}$, with $C^{(L)} \subseteq \mathbb{R}^{s^{(L)}}$, is given by

$$a_i''^{(L)}(\mathbf{c}^{(L)}) = -\exp(c_i^{(L)}) \frac{\exp(c_i^{(L)}) - 1}{\left(\exp(c_i^{(L)} + 1)\right)^3}, \tag{4.50}$$

for $i = 1, \ldots, s^{(L)}$.

Example 40 *Consider a layer of logistic perceptrons L with size $s^{(L)} = 2$. The activation second derivative for a combination $\mathbf{c}^{(L)} = (-1, 1)$ is*

$$\mathbf{a}^{(L)} = (0.232, 0.011).$$

Layer hyperbolic tangent second derivative

The second derivative of the hyperbolic tangent is similar to that for the logistic function $\mathbf{a}''^{(L)} : C^{(L)} \to C''^{s^{(L)}}$, with $C^{(L)} \subseteq \mathbb{R}^{s^{(L)}}$, is

$$a_i''^{(L)}(\mathbf{c}^{(L)}) = -2 \tanh(c_i^{(L)})(1 - \tanh^2(c_i^{(L)})), \tag{4.51}$$

for $i = 1, \ldots, s^{(L)}$.

Example 41 *Let L be a layer of $s^{(L)} = 2$ perceptrons with hyperbolic tangent activation function. If the combination of that is $\mathbf{c}^{(L)} = (-1, 1)$ the activation second derivative will be*

$$\mathbf{a}^{(L)} = (0.639, -0.639).$$

Layer linear function second derivative

Finally, a linear layer has second derivative $\mathbf{a}''^{(L)} : C^{(L)} \to 0^{s^{(L)}}$, with $C^{(L)} \subseteq \mathbb{R}^{s^{(L)}}$,

$$a_i''^{(L)}(\mathbf{c}^{(L)}) = 0. \tag{4.52}$$

for $i = 1, \ldots, s^{(L)}$.

Example 42 *The activation second derivative of a linear layer of perceptrons L of size $s^{(L)=3}$ for the combination $\mathbf{c}^{(L)} = (-1, 0, 1)$ is*

$$\mathbf{a}''^{(L)}(-1, 0, 1) = (0, 0, 0).$$

4.16 The MultilayerPerceptron class

`Flood` implements a multilayer perceptron with an arbitrary number of hidden layers of perceptrons and an output layer of perceptrons in the class MultilayerPerceptron. This neural network can approximate any function [27].

The MultilayerPerceptron class is the bigest one in `Flood`, having many different members, constructors and methods.

Members

This class contains:
- The number of inputs.
- The size of the hidden layers.
- The number of outputs.
- A vector of vectors of hidden perceptrons.
- An vector of output perceptrons.
- The activation functions of the hidden layers.
- The activation function of the output layer.
- The name of the input and output variables.

- The units of the input and output variables.
- The description of the input and output variables.
- The mean and the standard deviation of the input and output variables.
- The minimum and maximum values of the input and output variables.
- The lower and upper bounds of the output variables.
- The input variables scaling and output variables unscaling methods.
- The independent parameters vector.
- The name of the independent parameters.
- The units of the independent parameters.
- The description of the independent parameters.
- The mean and the standard deviation of the independent parameters.
- The minimum and maximum values of the independent parameters.
- The lower and upper bounds of the independent parameters.
- The independent parameters scaling and unscaling method.

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 MultilayerPerceptron 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 multilayer perceptron object is by means of the default constructor, which creates a multilayer perceptron without network architecture and without independent parameters.

```
MultilayerPerceptron mlp;
```

To construct a multilayer perceptron object with, for example, 1 input, a single hidden layer of 3 neurons and an output layer with 1 neuron, we use the one hidden layer constructor

```
MultilayerPerceptron mlp(1,6,1);
```

All the neural 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.

In order to construct a neural network with more hidden layers the number of hidden neurons for each layer must be specified in a vector of 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<int> hidden_layers_size(3);
hidden_layers_size[0] = 2;
hidden_layers_size[1] = 4;
hidden_layers_size[2] = 3;
MultilayerPerceptron mlp(1, hidden_layers_size, 1);
```

The neural parameters here are also initialized at random.

The independent parameters constructor creates a multilayer perceptron object with no network architecture and a given number of independent parameters,

```
MultilayerPerceptron mlp(3);
```

It is possible to construct a multilayer perceptron by loading its members from a data file. That is done in the following way,

```
MultilayerPerceptron mlp('MultilayerPerceptron.dat');
```

Please follow strictly the format of the multilayer perceptron file. If that is not correct, `Flood` will launch an error and terminate the program.

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

```
MultilayerPerceptron mlp1(2,4,3);
MultilayerPerceptron mlp2(&mlp1);
```

Methods

This class implements get and set methods for each member.

The get_inputs_number, get_hidden_layers_size and get_outputs_number methods return the number of inputs, the size of the hidden layers and the number of outputs, respectively.

```
MultilayerPerceptron mlp(2, 4, 3);
int inputs_number = mlp.get_inputs_number();
Vector<int> hidden_layers_size = mlp.get_hidden_layers_size();
int outputs_number = mlp.get_outputs_number();
```

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

```
int neural_parameters_number = mlp.get_neural_parameters_number();
```

The activation functions can be changed by doing

```
Vector<MultilayerPerceptron::LayerActivationFunction>
hidden_layers_activation_function(3, MultilayerPerceptron::Logistic);
mlp.set_hidden_layers_activation_function
(hidden_layers_activation_function);
mlp.set_output_layer_activation_function
(MultilayerPerceptron::HyperbolicTangent);
```

To set the mean, the standard deviation, the minimum and the maximum values of the input and output variables we can use the set_statistics method. For instance, the sentences

```
MultilayerPerceptron mlp(1,3,1);

Vector<double> output_variables_mean(1, 0.0);
Vector<double> output_variables_standard_deviation(1, 1.0);

Vector<double> input_variables_minimum(1, -1.0);
Vector<double> input_variables_maximum(1, 1.0);

Vector<double> output_variables_minimum(1, -1.0);
Vector<double> output_variables_maximum(1, 1.0);

Vector< Vector<double> > statistics(6);

statistics[0] = input_variables_mean;
statistics[1] = input_variables_standard_deviation;
statistics[2] = output_variables_mean;
statistics[3] = output_variables_standard_deviation;
statistics[4] = input_variables_minimum;
statistics[5] = input_variables_maximum;

mlp.set_statistics(statistics);
```

set the mean and standard deviation of both input and output variables to 0 and 1, and the minimum and maximum of the input and output variables to -1 and 1.

By default, a multilayer perceptron has not assigned any scaling and unscaling method. In order to use the mean and standard deviation inputs scaling method we can write

```
MultilayerPerceptron mlp(2,3,4);
mlp.set_inputs_scaling_method
(MultilayerPerceptron::MeanStandardDeviation);
```

In the same way, if we want to use the minimum and maximum outputs unscaling method we can use

```
MultilayerPerceptron mlp(4,3,2);
mlp.set_outputs_unscaling_method
(MultilayerPerceptron::MinimumMaximum);
```

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

```
MultilayerPerceptron mlp(4,3,2);
mlp.initialize(0.0);
```

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

```
Vector<double> input(1);
input[0] = 0.5;
Vector<double> output = mlp.calculate_output(input);
```

returns the network's output value $y = y(x)$ for an input value $x = 0.5$.

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 = mlp.calculate_Jacobian(input);
```

returns the network's output derivative value $\partial y(x)/\partial x$ for the same input value as before.

A set of independent parameters can be associated to the multilayer perceptron using the set_independent_parameters_number. For example,

```
MultilayerPerceptron mlp;
mlp.set_independent_parameters_number(2);
```

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

```
MultilayerPerceptron mlp;
mlp.save('MultilayerPerceptron.dat');
```

saves the multilayer perceptron object to the file MultilayerPerceptron.dat.

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

```
MultilayerPerceptron mlp;
mlp.load('MultilayerPerceptron.dat');
```

loads the multilayer perceptron object from the file MultilayerPerceptron.dat.

File format

A multilayer perceptron object can be serialized or deserialized to or from a data file which contains the member values. The file format of an object of the MultilayerPerceptron class is of XML type.

```
<Flood version='3.0' class='MultilayerPerceptron'>
<InputsNumber>
inputs_number
</InputsNumber>
<HiddenLayersNumber>
hidden_layers_number
</HiddenLayersNumber>
<HiddenLayersSize>
hidden_layer_size_1 ... hidden_layer_size_h
```

```
</HiddenLayersSize>
<OutputsNumber>
outputs_number
</OutputsNumber>
<IndependentParametersNumber>
independent_parameters_number
</IndependentParametersNumber>
<HiddenLayersActivationFunction>
hidden_layer_activation_function_1 ... hidden_layer_activation_function_h
</HiddenLayersActivationFunction>
<OutputLayerActivationFunction>
output_layer_activation_function
</OutputLayerActivationFunction>
<NeuralParameters>
neural_parameters_1 ... neural_parameters_d^N
</NeuralParameters>
<InputVariablesName> input_variable_name_1 ... input_variable_name_n
</InputVariablesName>
<InputVariablesUnits>
input_variable_units_1 ... input_variable_units_n
</InputVariablesUnits>
<InputVariablesDescription>
input_variable_description_1 ... input_variable_description_n
</InputVariablesDescription>
<OutputVariablesName>
output_variable_name_1 ... output_variable_name_m
</OutputVariablesName>
<OutputVariablesUnits>
output_variable_units_1 ... output_variable_units_m
</OutputVariablesUnits>
<OutputVariablesDescription>
output_variable_description_1 ... output_variable_description_m
</OutputVariablesDescription>
<InputVariablesMean>
input_variable_mean_1 ... input_variable_mean_n
</InputVariablesMean>
<InputVariablesStandardDeviation>
input_variable_standard_deviation_1 ... input_variable_standard_deviation_n
</InputVariablesStandardDeviation>
<InputVariablesMinimum>
input_variable_minimum_1 ... input_variable_minimum_n
</InputVariablesMinimum>
<InputVariablesMaximum>
input_variable_maximum_1 ... input_variable_maximum_n
</InputVariablesMaximum>
<OutputVariablesMean>
output_variable_mean_1 ... output_variable_mean_m
</OutputVariablesMean>
<OutputVariablesStandardDeviation>
output_variable_standard_deviation_1 ... output_variable_standard_deviation_m
</OutputVariablesStandardDeviation>
<OutputVariablesMinimum>
output_variable_minimum_1 ... output_variable_minimum_m
</OutputVariablesMinimum>
<OutputVariablesMaximum>
output_variable_maximum_1 ... output_variable_maximum_m
</OutputVariablesMaximum>
<OutputVariablesLowerBound>
output_variable_lower_bound_1 ... output_variable_lower_bound_m
</OutputVariablesLowerBound>
<OutputVariablesUpperBound>
output_variable_upper_bound_1 ... output_variable_upper_bound_m
```

```
</OutputVariablesUpperBound>
<InputsScalingMethod>
inputs_scaling_method
</InputsScalingMethod>
<OutputsUnscalingMethod>
outputs_unscaling_method
</OutputsUnscalingMethod>
<IndependentParameters>
independent_parameter_1 ... independent_parameter_d^I
</IndependentParameters>
<IndependentParametersName>
independent_parameter_name_1 ... independent_parameter_name_d^I
</IndependentParametersName>
<IndependentParametersUnits>
independent_parameter_units_1 ... independent_parameter_units_d^I
</IndependentParametersUnits>
<IndependentParametersDescription>
independent_parameter_description_1 ... independent_parameter_description_d^I
</IndependentParametersDescription>
<IndependentParametersMean>
independent_parameter_mean_1 ... independent_parameter_mean_d^I
</IndependentParametersMean>
<IndependentParametersStandardDeviation>
independent_parameter_standard_deviation_1 ... independent_parameter_standard_deviation_d^I
</IndependentParametersStandardDeviation>
<IndependentParametersMinimum>
independent_parameter_minimum_1 ... independent_parameter_minimum_d^I
</IndependentParametersMinimum>
<IndependentParametersMaximum>
independent_parameter_maximum_1 ... independent_parameter_maximum_d^I
</IndependentParametersMaximum>
<IndependentParametersLowerBound>
independent_parameter_lower_bound_1 ... independent_parameter_lower_bound_d^I
</IndependentParametersLowerBound>
<IndependentParametersUpperBound>
independent_parameter_upper_bound_1 ... independent_parameter_upper_bound_d^I
</IndependentParametersUpperBound>
<IndependentParametersScalingMethod>
independent_parameters_scalingmethod
</IndependentParametersScalingMehtod>
```

Chapter 5

The objective functional

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

5.1 Unconstrained variational problems

An objective functional for the multilayer perceptron $F : V \to \mathbb{R}$, being V the space of functions spanned by the neural network is of the form

$$F = F[\mathbf{y}(\mathbf{x})]. \tag{5.1}$$

The objective functional defines the task the 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 objective functional depends on the particular application.

The learning problem for the multilayer perceptron can then be stated as the searching in the neural network function space for an element $\mathbf{y}^*(\mathbf{x})$ at which the objective functional $F[\mathbf{y}(\mathbf{x})]$ takes a maximum or a minimum value.

The tasks of maximization and minimization are trivially related to each other, since maximization of F is equivalent to minimization of $-F$, and vice versa. On the other hand, a minimum can be either a global minimum, the smallest value of the functional over its entire domain, or a local minimum, the smallest value of the functional within some local neighborhood.

The simplest variational problems for the multilayer perceptron are those in which no constraints are posed on the solution $\mathbf{y}^*(\mathbf{x})$. In this way, the

general unconstrained variational problem for the multilayer perceptron can be formulated as follows:

Problem 1 (Unconstrained variational problem) *Let V be the space of all functions $\mathbf{y}(\mathbf{x})$ spanned by a multilayer perceptron, and let d be the dimension of V. Find a function $\mathbf{y}^*(\mathbf{x}) \in V$ for which the functional*

$$F[\mathbf{y}(\mathbf{x})],$$

defined on V, takes on a minimum value.

In other words, the unconstrained variational problem for the multilayer perceptron is stated in terms of the minimization of the objective functional associated to the neural network [34].

Example 43 (Geodesic problem) *Given two points $A = (x_a, y_a)$ and $B = (x_b, y_b)$ in a plane, find the shortest path between A and B. Figure 43 depicts graphically the formulation for this case study.*

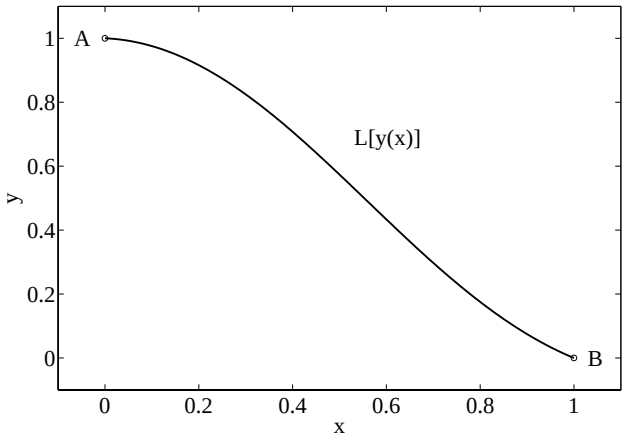


Figure 5.1: Geodesic problem statement.

The arc length between point A and point B of a curve $y(x)$ is given by the functional

$$L[y(x)] = \int_{x_a}^{x_b} \sqrt{1 + [y'(x)]^2} dx.$$

The analytical solution to the geodesic problem in the plane is obviously a straight line. For the particular case when $A = (1, 0)$ and $B = (0, 1)$, the Euler-Lagrange equation provides the following function as the minimal value for the arc length functional.

$$y^*(x) = 1 - x, \tag{5.2}$$

which gives $L[y^*(x)] = 1.414214$.

Example 44 *Given a collection of data points $(x_1, y_1), \ldots, (x_n, y_n)$, find a function which fits that data. Figure 44 shows the statement of this problem.*

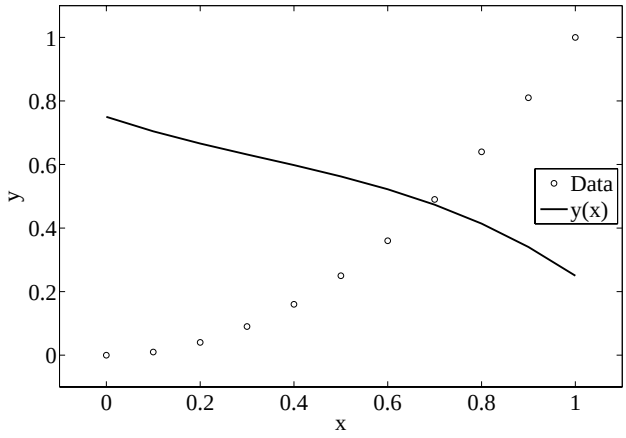


Figure 5.2: Data modelling statement.

The sum squared error E of a curve $y(x)$ with respect to the points $(x_1, y_1), \ldots, (x_n, y_n)$ is a functional. The value $E[y(x)]$ is given by

$$E[y(x)] = \sum_{i=1}^{n} \left(y(x_i) - y_i\right)^2.$$

5.2 Constrained variational problems

A variational problem for the multilayer perceptron can be specified by a set of constraints, which are equalities or inequalities that the solution $\mathbf{y}^*(\mathbf{x})$ must satisfy. Such constraints are expressed as functionals. Thus, the general constrained variational problem for the multilayer perceptron can be formulated as follows:

Problem 2 (Constrained variational problem) *Let V be the space of all functions $\mathbf{y}(\mathbf{x})$ spanned by a multilayer perceptron, and let d be the dimension of V. Find a function $\mathbf{y}^*(\mathbf{x}) \in V$ such that*

$$C_i[\mathbf{y}^*(\mathbf{x})] = 0,$$

for $i = 1, \ldots, l$, and for which the functional

$$F[\mathbf{y}(\mathbf{x})],$$

defined on V, takes on a minimum value.

In other words, the constrained variational problem for the multilayer perceptron consists of finding a vector of parameters which makes all the constraints to be satisfied and the objective functional to be an extremum.

A common approach when solving a constrained variational problem is to reduce it into an unconstrained problem. This can be done by adding a penalty term to the objective functional for each of the constrains in the original problem. Adding a penalty term gives a large positive or negative value to the objective functional when infeasibility due to a constrain is encountered.

For the minimization case, the general constrained variational problem for the multilayer perceptron can be reformulated as follows:

Problem 3 (Reduced unconstrained variational problem) *Let V be the space consisting of all functions $\mathbf{y}(\mathbf{x})$ that a given multilayer perceptron can define, and let d be the dimension of V. Find a function $\mathbf{y}^*(\mathbf{x}) \in V$ for which the functional*

$$F[\mathbf{y}(\mathbf{x})] + \sum_{i=1}^{l} \rho_i \left\| C_i[\mathbf{y}(\mathbf{x})] \right\|^2,$$

defined on V and with $\rho_i > 0$, for $i = 1, \ldots, l$, takes on a minimum value.

The parameters ρ_i, for $i = 1, \ldots, l$, are called the penalty term weights, being l the number of constraints. Note that, while the squared norm of the constrained is the metric most used, any other suitable metric can be used.

For large values of ρ, it is clear that the solution $\mathbf{y}^*(\mathbf{x})$ of Problem 3 will be in in a region where $C[\mathbf{y}(\mathbf{x})]$ is small. Thus, for increasing values of

ρ, it is expected that the the solution $\mathbf{y}^*(\mathbf{x})$ of Problem 3 will approach the constraints and, subject to being close, will minimize the objective functional $F[\mathbf{y}(\mathbf{x})]$. Ideally then, as $\rho \to \infty$, the solution of Problem 3 will converge to the solution of Problem 2 [35].

Example 45 (Catenary problem) *To find the curve assumed by a loose string of length l hung freely from two fixed points $A = (x_a, f_a)$ and $B = (x_b, f_b)$. Figure 5.3 graphically declares the catenary problem.*

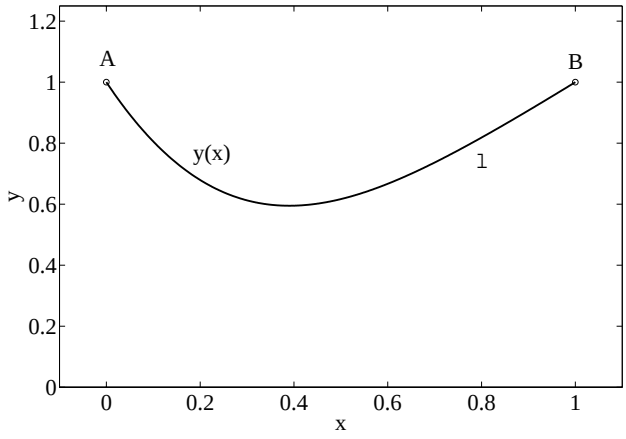


Figure 5.3: Catenary problem statement.

The length of a chain $y(x)$ is given by

$$L[y(x)] = \int_{x_a}^{x_b} \sqrt{1 + [y'(x)]^2} dx.$$

This chain is constrained to have length l, which can be written

$$\begin{aligned} E_L[y(x)] &= \int_{x_a}^{x_b} \sqrt{1 + [y'(x)]^2} dx - l \\ &= 0. \end{aligned} \tag{5.3}$$

On the other hand, the shape to be found is that which minimizes the potential energy. For a chain $y(x)$ with uniformly distributed mass this is given by

$$V[y(x)] = \int_{x_a}^{x_b} y(x) \sqrt{1 + [y'(x)]^2} dx.$$

The analytical solution to the catenary problem is an hyperbolic cosine. For the particular case when $l = 1.5$, $A = (0, 1)$ *and* $B = (1, 1)$, *it is written*

$$y^*(x) = 0.1891 + 0.3082 \cosh\left(\frac{x - 0.5}{0.3082}\right). \tag{5.4}$$

The potential energy of this catenary is $V[y^*(x)] = 1.0460$.

5.3 Reduced function optimization problem

As we saw in Section 5.1, the objective functional, $F : V \to \mathbb{R}$, with V the multilayer perceptron function space, is of the form

$$F = F[\mathbf{y}(\mathbf{x})]. \tag{5.5}$$

That objective functional has an objective function associated, $f : \mathbb{R}^d \to \mathbb{R}$, with d the number of parameters in the neural network [34], which is of the form

$$f = f(\boldsymbol{\zeta}). \tag{5.6}$$

The objective function for the multilayer perceptron, represented as $f(\boldsymbol{\zeta})$, can be visualized as a hypersurface, with $\zeta_1, \ldots, \zeta_d$ as coordinates, see Figure 5.4.

The minimum or maximum value of the objective functional is achieved for a vector of parameters at which the objective function takes on a minimum or maximum value. Therefore, the learning problem in the multilayer perceptron, formulated as a variational problem, can be reduced to a function optimization problem [34].

Problem 4 (Reduced function optimization problem) *Let* $\mathbb{R}^d$ *be the space of all vectors* $\boldsymbol{\zeta}$ *spanned by the parameters of a multilayer perceptron. Find a vector* $\boldsymbol{\zeta}^* \in \mathbb{R}^d$ *for which the function*

$$f(\boldsymbol{\zeta}),$$

defined on $\mathbb{R}^d$, *takes on a minimum value.*

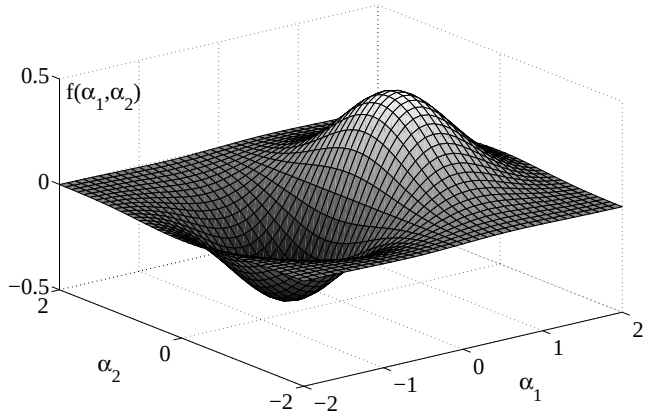


Figure 5.4: Geometrical representation of the objective function.

In this sense, a variational formulation for the multilayer perceptron 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.

Example 46 (De Jong's function) *Find a vector* $\boldsymbol{\zeta}^* \in \mathbb{R}^{12}$ *for which the function* $f : \mathbb{R}^{12} \to \mathbb{R}$ *defined by*

$$f(\boldsymbol{\zeta}) = \sum_{i=1}^{12} \zeta_i^2,$$

takes on a minimum value.

The De Jong's function has minimal argument $\boldsymbol{\zeta}^* = (0, \ldots, 0)$*, which gives a minimum value* $f(\boldsymbol{\zeta}^*) = 0$*.*

5.4 Objective function gradient

We have seen that the objective functional for the multilayer perceptron, $F[\mathbf{y}(\mathbf{x})]$, has an objective function associated, $f(\boldsymbol{\zeta})$, which is defined as a function of the parameters of the neural network ; the learning problem in the multilayer perceptron is solved by finding the values of the parameters which make the objective function to be an extremum.

For a multilayer perceptron, the gradient of the objective function, denoted $\nabla f(\boldsymbol{\zeta})$, is the vector of partial derivatives

$$\nabla f(\boldsymbol{\zeta}) = \left(\frac{\partial f}{\partial \zeta_1}, \ldots, \frac{\partial f}{\partial \zeta_d} \right). \tag{5.7}$$

The use of gradient information is of central importance in using training algorithms which are sufficiently fast to be of practical use for large-scale applications. Figure 5.5 represents the objective function gradient for the hypothetical case of a multilayer perceptron with two parameters.

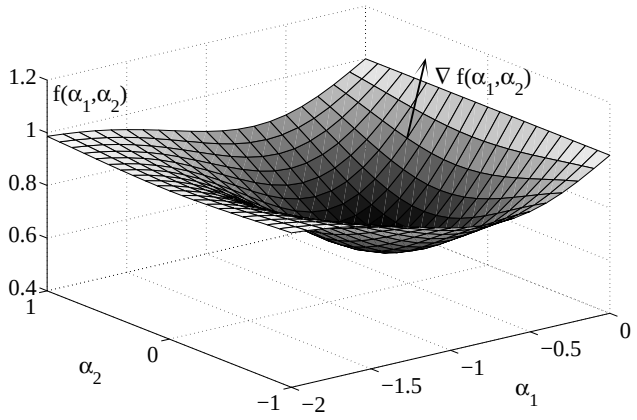


Figure 5.5: Illustration of the objective function gradient.

Example 47 *Consider the objective function* $f : \mathbb{R}^3 \to \mathbb{R}$ *defined by*

$$f(\boldsymbol{\zeta}) = \sum_{i=1}^{3} \zeta_i^2.$$

The partial derivatives $\partial f / \partial \zeta$ *are given by*

$$\frac{\partial f}{\partial \zeta_i} = 2\zeta_i,$$

for $i = 1, \ldots, 3$. *The gradient at* $\boldsymbol{\zeta} = (-1, 0, 1)$ *is*

$$\nabla f(-1, 0, 1) = (-2, 0, 2).$$

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

The back-propagation algorithm for the objective function gradient

The back-propagation algorithm is a principal result in the neural networks field. Here we obtain the objective function gradient for the multilayer perceptron using that method.

We want to find a procedure to evaluate the partial derivatives $\partial f/\partial \zeta_i$, for $i = 1, \ldots, d$.

The first step is to obtain the combination, activation and activation derivative of all hidden and output neurons by a consecutive applying of Equations (4.12), (4.13) and (4.39). This process is called forward propagation derivative, since it can be considered as a feed flow of information through the neural network.

Consider now the evaluation of the derivative of the objective function f with respect to some parameter $\zeta_{ij}^{(L)}$, where L is the layer index, i is the perceptron index and j is the input index.

The objective function f depends on the parameter $\zeta_{ij}^{(L)}$ only though the combination $c_i^{(L)}$. We can then apply the chain rule for partial derivatives to give, for layer L

$$\frac{\partial f}{\partial \zeta_{ij}^{(L)}} = \frac{\partial f}{\partial c_i^{(L)}} \frac{\partial c_i^{(L)}}{\partial \zeta_{ij}^{(L)}}, \tag{5.8}$$

for $i = 1, \ldots, s^{(L)}$ and $j = 1, \ldots, s^{(L-1)}$.

We now introduce the notation

$$\delta \equiv \frac{\partial f}{\partial c}, \tag{5.9}$$

where the quantity δ is called delta and it is considered for each neuron in the neural network. In this way

$$\frac{\partial f}{\partial \zeta_{ij}^{(L)}} = \delta_i^{(L)} \frac{\partial c_i^{(L)}}{\partial \zeta_{ij}^{(L)}}, \tag{5.10}$$

for $i = 1, \ldots, s^{(L)}$ and $j = 1, \ldots, s^{(L-1)}$.

The evaluation of delta for the output neurons is quite simple. By definition,

$$\begin{aligned}\delta_i^{(h+1)} &\equiv \frac{\partial f}{\partial c_i^{(h+1)}} \\ &= \frac{\partial f}{\partial a_i^{(h+1)}} \frac{\partial a_i{}^{(h+1)}}{\partial c_i^{(h+1)}} \\ &= \frac{\partial f}{\partial a_i^{(h+1)}} a_i'^{(h+1)}, \end{aligned} \tag{5.11}$$

for $i = 1, \ldots, m$.

To evaluate delta for the neurons in the last hidden layer we make use of the chain rule for partial derivatives,

$$\begin{aligned}\delta_i^{(h)} &\equiv \frac{\partial f}{\partial c_i^{(h)}} \\ &= \sum_{j=1}^{m} \frac{\partial f}{\partial c_j^{(h+1)}} \frac{\partial c_j^{(h+1)}}{\partial c_i^{(h)}} \\ &= \delta^{(h+1)}, \end{aligned} \tag{5.12}$$

for $i = 1, \ldots, s^{(h)}$. Substituting the definition of delta for the neurons in the output layer, and making use of (4.12), (4.13), and (4.39), we obtain

$$\delta_i^{(h)} = a_i'^{(h)} \sum_{j=1}^{m} \zeta_{ji}^{(h+1)} \delta_j^{(h+1)}, \tag{5.13}$$

for $i = 1, \ldots, s^{(h)}$.

Similarly, to evaluate delta for the first layer

$$\delta_i^{(1)} = a_i'^{(1)} \sum_{j=1}^{s^{(2)}} \zeta_{ji}^{(2)} \delta_j^{(2)}, \tag{5.14}$$

for $i = 1, \ldots, s^{(1)}$.

The derivatives $\partial c / \partial \zeta$ for the first hidden layer are given by

$$\frac{\partial c_i^{(1)}}{\partial \zeta_{ij}^{(1)}} = x_i, \tag{5.15}$$

for $i = 1, \ldots, n$ and $j = 1, \ldots, s^{(1)}$. Similarly, for the last hidden layer, the derivatives $\partial c/\partial \zeta$ are

$$\frac{\partial c_j^{(h)}}{\partial \zeta_{ji}^{(h)}} = y_j^{(h-1)}, \tag{5.16}$$

for $i = 1, \ldots, s^{(h-1)}$ and $j = 1, \ldots, s^{(h)}$. Finally we can write the derivatives $\partial c/\partial \zeta$, for the output layer,

$$\frac{\partial c_k^{(2)}}{\partial \zeta_{kj}^{(2)}} = y_j^{(1)}, \tag{5.17}$$

for $j = 1, \ldots, s^{(h)}$ and $k = 1, \ldots, s^{(h)}$. We then obtain, for the hidden layer

$$\frac{\partial f}{\partial \zeta_{ji}^{(1)}} = \delta_j^{(1)} x_i, \tag{5.18}$$

for $i = 1, \ldots, n$ and $j = 1, \ldots, h_1$. Likewise, we obtain, for the output layer

$$\frac{\partial f}{\partial \zeta_{kj}^{(2)}} = \delta_k^{(2)} y_j^{(1)}, \tag{5.19}$$

for $j = 1, \ldots, h_1$ and $k = 1, \ldots, m$.

We now obtain

$$\delta_j^{(1)} = a'^{(1)}(c_j^{(1)}) \sum_{k=1}^{m} \zeta_{kj}^{(2)} \delta_k^{(2)}, \tag{5.20}$$

for $j = 1, \ldots, h_1$. We can summarize the back-propagation procedure to evaluate the derivatives of the objective function with respect to the parameters in just four steps:

Calculate forward propagation derivative Calculate output errors Calculate hidden errors Calculate hidden layers error gradient Calculate output layer error gradient

1. Apply an input $\mathbf{x}$ to the neural network and forward-propagate it to find the activation of all hidden and output neurons.

2. Evaluate the errors $\delta_k^{(2)}$ for all output neurons.

3. Back-propagate the errors $\delta_k^{(2)}$ by using to obtain $\delta_j^{(1)}$ for each hidden neuron in the neural network.

4. Evaluate the required derivatives of the objective function with respect to the parameters in the hidden and output layers, respectively.

Numerical differentiation for the objective function gradient

There are many applications when it is not possible to obtain the objective function gradient using the back-propagation algorithm, and it needs to be computed numerically. This can be done by perturbing each parameter in turn, and approximating the derivatives by using the finite differences method

$$\frac{\partial f}{\partial \zeta_i} = \frac{f(\zeta_i + \epsilon) - f(\zeta_i)}{\epsilon} + \mathcal{O}(\epsilon), \tag{5.21}$$

for $i = 1, \ldots, d$ and for some small numerical value of ϵ.

The accuracy of the finite differences method can be improved significantly by using central differences of the form

$$\frac{\partial f}{\partial \zeta_i} = \frac{f(\zeta_i + \epsilon) - f(\zeta_i - \epsilon)}{2\epsilon} + \mathcal{O}(\epsilon^2), \tag{5.22}$$

also for $i = 1, \ldots, d$ and for some small numerical value of ϵ.

In a software implementation, when possible, derivatives of the objective function f with respect to the parameters in the neural network $\boldsymbol{\zeta}$ should be evaluated using back-propagation, since this provides the greatest accuracy and numerical efficiency.

5.5 Objective function Hessian

There are some training algorithms which also make use of the Hessian matrix of the objective function to search for an optimal set of parameters. The Hessian matrix of the objective function $f : \mathbb{R}^d \to \mathbb{R}$ is written

$$\mathbf{H}f(\boldsymbol{\zeta}) = \begin{pmatrix} \dfrac{\partial^2 f}{\partial \zeta_1^2} & \cdots & \dfrac{\partial^2 f}{\partial \zeta_1 \partial \zeta_d} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial \zeta_d \partial \zeta_1} & \cdots & \dfrac{\partial^2 f}{\partial \zeta_d^2} \end{pmatrix}. \tag{5.23}$$

Example 48 *Consider the objective function* $f : \mathbb{R}^3 \to \mathbb{R}$ *defined by*

$$f(\boldsymbol{\zeta}) = \sum_{i=1}^{3} \zeta_i^2.$$

The second derivatives $\partial^2 f / \partial \zeta^2$ *are given by*

$$\frac{\partial^2 f}{\partial \zeta_{ij}^2} = \begin{cases} 2, & i = j, \\ 0, & i \neq j, \end{cases}$$

for $i, j = 1, \ldots, 3$. *The Hessian at* $\boldsymbol{\zeta} = (-1, 0, 1)$ *is*

$$\mathbf{H}f(-1, 0, 1) = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix}$$

The most general scheme to calculate the Hessian matrix is to apply numerical differentiation. However, there are some objective functions which have an analytical solution for the Hessian matrix, which can be calculated using a back-propagation algorithm [8].

Numerical differentiation for the objective function Hessian

As it happens for the gradient vector, there are many applications when analytical evaluation of the Hessian is not possible, and it must be computed numerically. This can be done by perturbing each argument element in turn, and approximating the derivatives by using numerical differentiation.

The forward differences method for the Hessian matrix gives the following expression

$$\begin{aligned} \frac{\partial^2 f}{\partial \zeta_i \partial \zeta_j} & \\ &= \frac{f(\zeta_i + \epsilon, \zeta_j + \epsilon)}{4\epsilon^2} - \frac{f(\zeta_i + \epsilon, \zeta_j - \epsilon)}{4\epsilon^2} \\ &- \frac{f(\zeta_i - \epsilon, \zeta_j + \epsilon)}{4\epsilon^2} + \frac{f(\zeta_i - \epsilon, \zeta_j - \epsilon)}{4\epsilon^2} + \mathcal{O}(\epsilon^2), \end{aligned} \tag{5.24}$$

The accuracy of forward differences can be improved by using central differences,

$$\begin{aligned}\frac{\partial^2 f}{\partial\zeta_i\partial\zeta_j} & \\ &= \frac{f(\zeta_i+\epsilon,\zeta_j+\epsilon)}{4\epsilon^2} - \frac{f(\zeta_i+\epsilon,\zeta_j-\epsilon)}{4\epsilon^2} \\ &- \frac{f(\zeta_i-\epsilon,\zeta_j+\epsilon)}{4\epsilon^2} + \frac{f(\zeta_i-\epsilon,\zeta_j-\epsilon)}{4\epsilon^2} + \mathcal{O}(\epsilon^2), \end{aligned} \tag{5.25}$$

for $i, j = 1, \ldots, d$, and where d is the number of parameters in the neural network.

5.6 Regularization theory

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 $\mathbf{x}$ led to small changes in the dependent variable $\mathbf{y}$. 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 [51]. This can be achieved by choosing an objective functional $\overline{F} : V \to \mathbb{R}$ which adds a regularization term $\Omega : V \to \mathbb{R}$ to the original functional $F : V \to \mathbb{R}$, being V the multilayer perceptron function space [15]. The regularized objective functional then becomes

$$\overline{F}[\mathbf{y}(\mathbf{x})] = F[\mathbf{y}(\mathbf{x})] + \nu\Omega[\mathbf{y}(\mathbf{x})], \tag{5.26}$$

where the parameter ν is called the regularization term weight. The value of the functional Ω depends on the function $\mathbf{y}(\mathbf{x})$, and if $\Omega[\mathbf{y}(\mathbf{x})]$ is chosen appropriately, it can be used to control ill-posseness [8].

One of the simplest forms of regularization term is called parameter decay and consists on the sum of the squares of the parameters in the neural network divided by the number of parameters [8].

$$\Omega[\mathbf{y}(\mathbf{x})] = \frac{1}{d}\sum_{i=1}^{d}\zeta_i^2, \tag{5.27}$$

where d is the number of parameters. Adding this term to the objective function will cause the neural network to have smaller weights and biases, and this will force its response to be smoother.

The problem with regularization is that it is difficult to determine the optimum value for the term weight ν. If we make this parameter too small, we will not reduce ill-possenes. If the regularization term weight is too large, the regularization term will dominate the objective functional and the solution will not be correct. In this way, it is desirable to determine the optimal regularization parameters in an automated fashion.

5.7 ObjectiveFunctional classes

`Flood` includes the ObjectiveFunctional abstract class to represent the concept of objective functional. This class does not represent any concrete objective functional, and derived classes must be implemented in order to define a variational problem.

Members

That class contains:
- A relationship to a multilayer perceptron object.
- The objective weight.
- The regularization method.
- The regularization weight.
- An evaluation counter.
- A gradient counter.
- A Hessian counter.
- The numerical differentiation method.
- The numerical epsilon method.
- The numerical epsilon value.
- A display flag.

As usual, that members are private. They are accesed or modified by using their corresponding get and set methods.

File format

The default file format of an objective functional class is listed below.

```
<Flood version='3.0' class='ObjectiveFunctional'>
<RegularizationMethod>
regularization_method
</RegularizationMethod>
<ObjectiveWeight>
objective_weight
</ObjectiveWeight>
<RegularizationWeight>
regularization_weight
```

```
</RegularizationWeight>
<CalculateEvaluationCount>
calculate_evaluation_count
</CalculateEvaluationCount>
<CalculateGradientCount>
calculate_gradient_count
</CalculateGradientCount>
<CalculateHessianCount>
calculate_Hessian_count
</CalculateHessianCount>
<NumericalDifferentiationMethod>
numerical_differentiation_method
</NumericalDifferentiationMethod>
<NumericalEpsilonMethod>
numerical_epsilon_method
</NumericalEpsilonMethod>
<NumericalEpsilon>
numerical_epsilon
</NumericalEpsilon>
<Display>
display
</Display>
```

Constructors

As it has been said, the choice of the objective functional depends on the particular application. Therefore instantiation of the ObjectiveFunctional class is not possible, and concrete classes must be derived.

Methods

Any derived class must implement the pure virtual calculate_objective method. This returns the objective term objective term of a multilayer perceptron for some objective functional.

```
MockObjectiveFunctional mof;
double objective = mof.calculate_objective();
```

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

Derived classes might also implement the calculate_objective_gradient method. By default, calculation of the gradient vector is performed with numerical differentiation. Implementation of that method will override them and allow to compute the derivatives analytically. The use of that method is as follows

```
MockObjectiveFunctional mof;
Vector<double> objective_gradient = mof.calculate_objective_gradient();
```

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

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

```
MockObjectiveFunctional mof;
Matrix<double> objective_Hessian = mof.calculate_objective_Hessian();
```

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

The objective functional is not regularized by default. To change that, the set_regularization_method method is used

```
MockObjectiveFunctional mof;
mof.set_regularization_method(ObjectiveFunctional::NeuralParametersNorm);
```

The default method for numerical differentiation is central differences. If you want to use forward differences instead, you can write

```
MockObjectiveFunctional mof;
mof.set_numerical_differentiation_method
(ObjectiveFunctional::ForwardDifferences);
```

Derived classes

For data modeling problems, such as function regression or pattern recognition, `Flood` includes the classes SumSquaredError, MeanSquaredError, RootMeanSquaredError, NormalizedSquaredError and MinkowskiError. Read Chapters 7 and 8 to learn about modeling of data and the use of that classes.

On the other hand, other types of variational problems require programming another derived class. As a way of illustration, `Flood` includes the examples GeodesicProblem, BrachistochroneProblem, CatenaryProblem and IsoperimetricProblem, which are classical problems in the calculus of variations.

Examples for optimal control problems included are CarProblem, CarProblemNeurocomputing, FedBatchFermenterProblem and AircraftLandingProblem, see Chapter 9.

Regarding inverse problems, `Flood` includes the example PrecipitateDissolutionModeling. Read Chapter 11 to see how this type of problems are formulated and how to solve them by means of a neural network.

As an example of optimal shape design, the MinimumDragProblem example is included. All these is explained in Chapter 10.

Finally, `Flood` can be used as a software tool for function optimization problems. Some examples included are DeJongFunction, RosenbrockFunction, RastriginFunction, PlaneCylinder and WeldedBeam. Please read Chapter 12 if you are interested on that.

Chapter 6

The training algorithm

The procedure used to carry out the learning process in a neural network is called the training algorithm. There are many different training algorithms for the multilayer perceptron. Some of the most used are the quasi-Newton method or the evolutionary algorithm.

6.1 One-dimensional optimization

Although the objective function for the multilayer perceptron is multidimensional, one-dimensional optimization methods are of great importance here. Indeed, one-dimensional optimization algorithms are very often used inside multidimensional optimization algorithms.

Consider an objective function of one parameter $f : \mathbb{R} \to \mathbb{R}$ given by

$$f = f(\eta), \tag{6.1}$$

continuous, derivable, and with continuous derivatives.

The function f is said to have a relative or local minimum at $\eta^* \in \mathbb{R}$ if $f(\eta^*) \leq f(\eta^* + h)$ for all sufficiently small positive and negative values of h. Similarly, a point $\eta^* \in \mathbb{R}$ is called a relative or local maximum if $f(\eta^*) \geq f(\eta^* + h)$ for all values of h sufficiently close to zero.

The function f is said to have a global or absolute minimum at η^* if $f(\eta^*) \leq f(\eta)$ for all $\eta \in \mathbb{R}$, and not just for all η close to η^*. Similarly, a point η^* will be a global maximum of f if $f(\eta^*) \geq f(\eta)$ for all $\eta \in \mathbb{R}$ in the domain. Finding a global optimum is, in general, a very difficult problem [56]. On the other hand, the tasks of maximization and minimization are trivially related to each other, since maximization of $f(\eta)$ is equivalent to minimization of $-f(\eta)$, and vice versa.

In this regard, a one-dimensional optimization problem is one in which the argument η^* which minimizes the objective function f is to be found.

The necessary condition states that if the function $f(\eta)$ has a relative optimum at η^* and if the derivative $f'(\eta)$ exists as a finite number at η^*, then

$$f'(\eta^*) = 0. \tag{6.2}$$

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.

Bracketing a minimum

Line minimization algorithms begin by locating an interval in which the minimum of the objective function along occurs. A minimum is known to be bracketed when there is a triplet of points $a < b < c$ such that $f(a) > f(b) < f(c)$. In this case we know that f has a minimum in the interval (a, c).

Golden section

The golden section method brackets that minimum until the distance between the two outer points in the bracket is less than a defined tolerance [45].

Brent's method

The Brent's method performs a parabolic interpolation until the distance between the two outer points defining the parabola is less than a tolerance [45].

6.2 Multidimensional optimization

As we saw in Chapter 5, the learning problem in the multilayer perceptron is reduced to the searching in a d-dimensional space for a parameter vector $\boldsymbol{\zeta}^*$ for which the objective function f takes a maximum or a minimum value.

Consider an objective function $f : \mathbb{R}^d \to \mathbb{R}$, defined by

$$f = f(\boldsymbol{\zeta}). \tag{6.3}$$

continuous, derivable, and with continuous derivatives.

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 function $f(\boldsymbol{\zeta})$ has a relative optimum at $\boldsymbol{\zeta}^* \in \mathbb{R}^d$ and if the gradient $\nabla f(\boldsymbol{\zeta})$ exists as a finite vector at $\boldsymbol{\zeta}^*$, then

$$\nabla f(\boldsymbol{\zeta}^*) = \mathbf{0}. \tag{6.4}$$

The objective 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 of the form

$$\boldsymbol{\zeta}_{i+1} = \boldsymbol{\zeta}_i + \mathbf{d}_i \eta_i, \tag{6.5}$$

where i labels the iteration step, or epoch. The vector $\mathbf{d}_i \eta_i$ is called the parameters increment. $\mathbf{d}_i$ is the training direction and η_i is the training rate. Different training algorithms involve different choices for the training direction and the training rate.

In this way, to train a multilayer perceptron we start with an initial parameter vector $\boldsymbol{\zeta}_0$ (often chosen at random) and we generate a sequence of parameter vectors $\boldsymbol{\zeta}_1$, $\boldsymbol{\zeta}_2, \ldots$, so that the objective function f is reduced at each iteration of the algorithm, that is

$$f(\boldsymbol{\zeta}_{i+1}) < f(\boldsymbol{\zeta}_i), \tag{6.6}$$

where the quantity $f(\boldsymbol{\zeta}_{i+1}) - f(\boldsymbol{\zeta}_i)$ is called the evaluation improvement.

The training algorithm stops when a specified condition is satisfied. Some stopping criteria commonly used are [15]:

1. The parameters increment norm is less than a minimum value.
2. Evaluation improvement in one epoch is less than a set value.
3. Evaluation has been minimized to a goal value.
4. The norm of the objective 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 multilayer perceptron. Early stopping is a very common practice in neural networks and often produces good solutions to ill-possed problems.

Figure 6.1 is a state diagram of the training procedure, showing states and transitions in the training process of a multilayer perceptron.

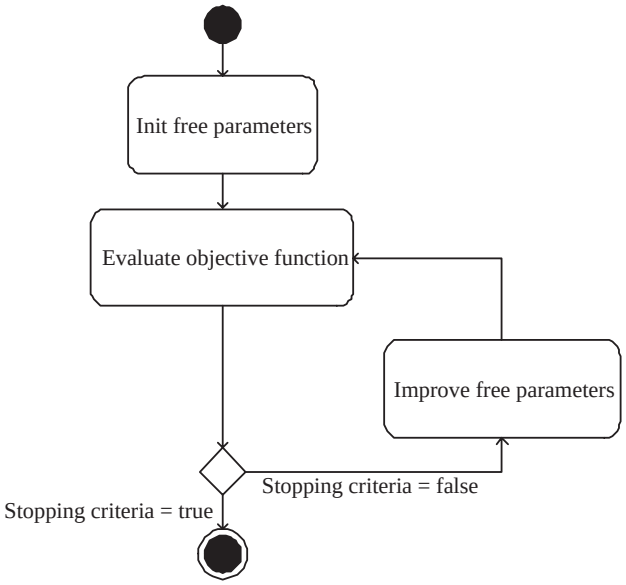


Figure 6.1: Training process in the multilayer perceptron.

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 [56].

Training algorithms might require information from the objective function only, the gradient vector of the objective function or the Hessian matrix of the objective function [45]. These methods, in turn, can perform either global or local optimization.

Zero-order training algorithms make use of the objective 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 [22] [20] or particle swarm optimization [28], which are global optimization methods .

First-order training algorithms use the objective function and its gradient vector [5]. Examples of these are gradient descent methods, conjugate gradient methods, scaled conjugate gradient methods [39] or quasi-Newton methods. Gradient descent, conjugate gradient, scaled conjugate gradient and quasi-Newton methods are local optimization methods [35].

Second-order training algorithms make use of the objective function, its gradient vector and its Hessian matrix [5]. Examples for second-order methods are Newton's method and the Levenberg-Marquardt algorithm [23]. Both of them are local optimization methods [35].

6.3 Gradient descent

Gradient descent, sometimes also known as steepest descent, is a local method which requires information from the gradient vector, and hence it is a first order training algorithm. It acts in a deterministic manner.

The method begins at a point $\boldsymbol{\zeta}_0$ and, until a stopping criterium is satisfied, moves from $\boldsymbol{\zeta}_i$ to $\boldsymbol{\zeta}_{i+1}$ along the line extending from $\boldsymbol{\zeta}_i$ in the training direction $\mathbf{d} = -\nabla f(\boldsymbol{\zeta}_i)$, the local downhill gradient. The gradient vector of the objective function for the multilayer perceptron is described in Section 5.4.

Therefore, starting from a parameter vector $\boldsymbol{\zeta}_0$, the gradient descent method takes the form of iterating

$$\boldsymbol{\zeta}_{i+1} = \boldsymbol{\zeta}_i - \nabla f(\boldsymbol{\zeta}_i) \cdot \eta_i, \tag{6.7}$$

for $i = 0, 1, \ldots$, and where the parameter η is the training rate. This value can either set to a fixed value or found by line minimization along the train direction at each epoch. Provided that the train rate is well chosen, the value of f will decrease at each successive step, eventually reaching to vector of parameters $\boldsymbol{\zeta}^*$ at which some stopping criterium is satisfied.

The choose of a suitable value for a fixed train rate presents a serious difficulty. If η is too large, the algorithm may overshoot leading to an increase in f and possibly to divergent oscillations, resulting in a complete breakdown in the algorithm. Conversely, if η is chosen to be very small the search can proceed extremely slowly, leading to long computation times. Furthermore, a good value for η will typically change during the course of training [8].

For that reason, an optimal value for the train rate obtained by line minimization at each successive epoch is generally preferable. Here a search is made along the train direction to determine the optimal train rate, which

minimizes the objective function along that line. Section 6.1 describes different one-dimensional minimization algorithms.

Figure 6.2 is a state diagram for the training process of a neural network with gradient descent. Improvement of the parameters is performed by obtaining first the gradient descent train direction and then a suitable training rate.

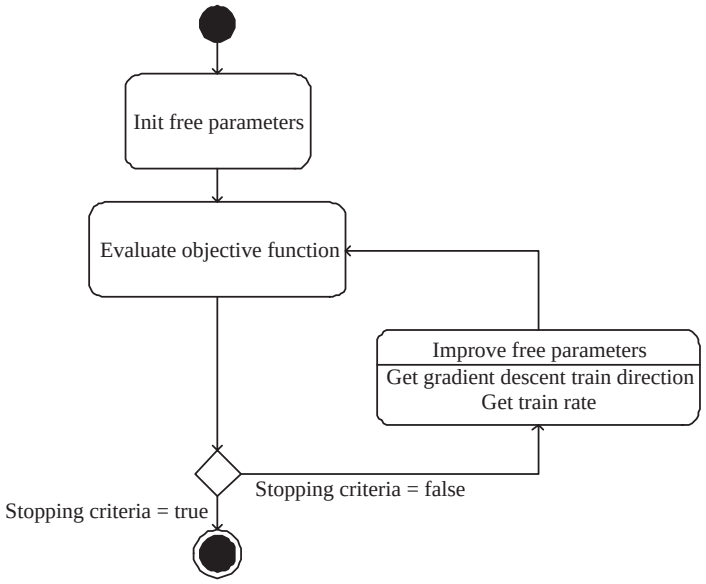


Figure 6.2: Training process with the gradient descent training algorithm.

The gradient descent training algorithm has the severe drawback of requiring many iterations for functions which have long, narrow valley structures. Indeed, the local downhill gradient is the direction in which the objective function decreases most rapidly, but this does not necessarily produce the fastest convergence. See [35] for a detailed discussion of this optimization method.

6.4 Newton's method

The Newton's method is a class of local algorithm which makes use of the Hessian matrix of the objective function. In this way it is a second order method. On the other hand, the Newton's method behaves in a deterministic fashion.

Consider the quadratic approximation of f at $\boldsymbol{\zeta}_0 \in \mathbb{R}^d$ using the Taylor's series expansion

$$f(\boldsymbol{\zeta}) = f(\boldsymbol{\zeta}_0) + \nabla f(\boldsymbol{\zeta} - \boldsymbol{\zeta}_0) + \frac{1}{2}(\boldsymbol{\zeta} - \boldsymbol{\zeta}_0) \cdot \mathbf{H} f(\boldsymbol{\zeta}_0) \cdot (\boldsymbol{\zeta} - \boldsymbol{\zeta}_0), \tag{6.8}$$

where $\mathbf{H}f(\boldsymbol{\zeta}_0)$ is the Hessian matrix of f evaluated at the point $\boldsymbol{\zeta}_0$. The Hessian matrix of the objective function for the multilayer perceptron is described in Section 5.5.

By setting $\nabla f(\boldsymbol{\zeta})$ in Equation (6.8) equal to $\mathbf{0}$ for the minimum of $f(\boldsymbol{\zeta})$, we obtain

$$\begin{aligned} \nabla f(\boldsymbol{\zeta}) &= \nabla f(\boldsymbol{\zeta}_0) + \mathbf{H} f(\boldsymbol{\zeta}_0) \cdot (\boldsymbol{\zeta} - \boldsymbol{\zeta}_0) \\ &= \mathbf{0}. \end{aligned} \tag{6.9}$$

If $\mathbf{H}f(\boldsymbol{\zeta}_0)$ is not singular, Equation (6.9) leads to an expression for the location of the minimum of the objective function,

$$\boldsymbol{\zeta}^* = \boldsymbol{\zeta}_0 - \mathbf{H}^{-1} f(\boldsymbol{\zeta}_0) \cdot \nabla f(\boldsymbol{\zeta}_0), \tag{6.10}$$

where $\mathbf{H}^{-1}f(\boldsymbol{\zeta}_0)$ is the inverse of the Hessian matrix of f evaluated at the point $\boldsymbol{\zeta}_0$.

Equation (6.10) would be exact for a quadratic objective function. However, since higher order terms have been neglected, this is to be used iteratively to find the optimal solution $\boldsymbol{\zeta}^*$. Therefore, starting from a parameter vector $\boldsymbol{\zeta}_0$, the iterative formula for the Newton's method can be written

$$\boldsymbol{\zeta}_{i+1} = \boldsymbol{\zeta}_i - \mathbf{H}^{-1} f(\boldsymbol{\zeta}_i) \cdot \nabla f(\boldsymbol{\zeta}_i), \tag{6.11}$$

for $i = 0, 1, \ldots$ and until some stopping criterium is satisfied.

The vector $\mathbf{H}^{-1}f(\boldsymbol{\zeta}) \cdot \nabla f(\boldsymbol{\zeta})$ is known as the Newton's increment. But note that this increment for the parameters may move towards a maximum or a saddle point rather than a minimum. This occurs if the Hessian is not positive definite, so that there exist directions of negative curvature. Thus, the objective function evaluation is not guaranteed to be reduced at each iteration. Moreover, the Newton's increment may be sufficiently large that it takes us outside the range of validity of the quadratic approximation. In this case the algorithm could become unstable.

In order to prevent such troubles, the Newton's method in Equation (6.4) is usually modified as

$$\boldsymbol{\zeta}_{i+1} = \boldsymbol{\zeta}_i - \mathbf{H}^{-1} f(\boldsymbol{\zeta}_i) \cdot \nabla f(\boldsymbol{\zeta}_i) \cdot \eta_i, \tag{6.12}$$

where the training rate η can either set to a fixed value or found by line minimization. See Section 6.1 for a description of several one-dimensional minimization algorithms.

In Equation (6.12), the vector $\mathbf{d} = \mathbf{H}^{-1} f(\boldsymbol{\zeta}) \cdot \nabla f(\boldsymbol{\zeta})$ is now called the Newton's train direction. The sequence of points $\boldsymbol{\zeta}_0, \boldsymbol{\zeta}_1, \ldots$ can be shown here to converge to the actual solution $\boldsymbol{\zeta}^*$ from any initial point $\boldsymbol{\zeta}_0$ sufficiently close the to solution, and provided that $\mathbf{H}$ is nonsingular.

The state diagram for the training process with the Newton's method is depicted in Figure 6.3. Here improvement of the parameters is performed by obtaining first the Newton's method train direction and then a suitable training rate.

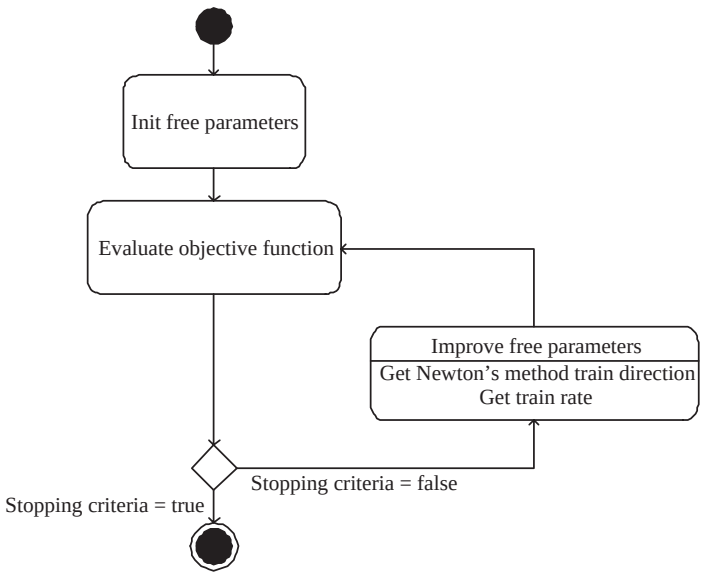


Figure 6.3: Training process with the Newton's method.

There are still several difficulties with such an approach, however. First, an exact evaluation of the Hessian matrix is computationally demanding. This evaluation would be prohibitively expensive if done at each stage of an iterative algorithm. Second, the Hessian must be inverted, and so is also computationally demanding. In [35] a complete description of the Newton's method can be found.

6.5 Conjugate gradient

Conjugate gradient is a local algorithm for an objective function whose gradient can be computed, belonging for that reason to the class of first order methods. According to its behavior, it can be described as a deterministic method.

The conjugate gradient method can be regarded as being somewhat intermediate between the method of gradient descent and Newton's method [35]. It is motivated by the desire to accelerate the typically slow convergence associated with gradient descent while avoiding the information requirements associated with the evaluation, storage, and inversion of the Hessian matrix as required by the Newton's method. In the conjugate gradient algorithm search is performed along conjugate directions, which produces generally faster convergence than steepest descent directions [15]. These train directions are conjugated with respect to the Hessian matrix. A set of vectors $\mathbf{d}_k$ are said to be conjugated with respect to the matrix $\mathbf{H}$ if only if

$$\mathbf{d}_i^{\mathrm{T}} \mathbf{H} \mathbf{d}_j = 0, \tag{6.13}$$

for all $i \neq j$ and provided that $\mathbf{H}$ is not singular. An elemental property of a set of conjugate directions is that these vectors are linearly independent. Therefore, if the number of parameters is d, the maximum size of a set of conjugate directions is also d.

Let denote $\mathbf{d}$ the train direction vector. Then, starting with an initial parameter vector $\boldsymbol{\zeta}_0$ and an initial train direction vector $\mathbf{d}_0 = -\nabla f(\boldsymbol{\zeta}_0)$, the conjugate gradient method constructs a sequence of train directions from the recurrence

$$\mathbf{d}_{i+1} = \nabla f(\boldsymbol{\zeta}_{i+1}) + \mathbf{d}_i \cdot \gamma_i, \tag{6.14}$$

for $i = 0, 1, \ldots$ and where γ is called the conjugate parameter.

The various versions of conjugate gradient are distinguished by the manner in which the conjugate parameter is constructed.

For the Fletcher-Reeves update the procedure is [19]

$$\gamma_{FRi} = \frac{\nabla f(\boldsymbol{\zeta}_{i+1}) \cdot \nabla f(\boldsymbol{\zeta}_{i+1})}{\nabla f(\boldsymbol{\zeta}_i) \cdot \nabla f(\boldsymbol{\zeta}_i)}, \tag{6.15}$$

where γ_{FR} is called the Fletcher-Reeves parameter.

For the Polak-Ribiere update the procedure is

$$\gamma_{PRi} = \frac{\big(\nabla f(\boldsymbol{\zeta}_{i+1}) - \nabla f(\boldsymbol{\zeta}_i)\big) \cdot \nabla f(\boldsymbol{\zeta}_{i+1})}{\nabla f(\boldsymbol{\zeta}_i) \cdot \nabla f(\boldsymbol{\zeta}_i)}, \tag{6.16}$$

where γ_{PR} is called the Polak-Ribiere parameter.

It can be shown that both the Fletcher-Reeves and the Polak-Ribiere train directions indeed satisfy Equation (6.17).

The parameters are then improved according to the formula

$$\boldsymbol{\zeta}_{i+1} = \boldsymbol{\zeta}_i + \mathbf{d}_i \eta_i, \tag{6.17}$$

also for $i = 0, 1, \ldots$, and where η is the train rate, which is usually found by line minimization.

For all conjugate gradient algorithms, the train direction is periodically reset to the negative of the gradient. The standard reset point occurs every d epochs, the number of parameters in the multilayer perceptron [44].

There is some evidence that the Polak-Ribiere formula accomplishes the transition to further iterations more efficiently: When it runs out of steam, it tends to reset the train direction $\mathbf{d}$ to be down the local gradient $-\nabla f(\boldsymbol{\zeta})$, which is equivalent to beginning the conjugate-gradient procedure again [45].

Figure 6.4 is a state diagram for the training process with the conjugate gradient. Here improvement of the parameters is done by first computing the conjugate gradient train direction and then a suitable train rate in that direction.

Conjugate gradient methods have proved to more effective than gradient descent or the Newton's method in dealing with general objective functions. A detailed discussion of the conjugate gradient method can be found in [35].

6.6 Quasi-Newton method

The quasi-Newton method can be classified as a local, first order and deterministic training algorithm for the multilayer perceptron.

In Section 6.4 it was argued that a direct application of the Newton's method, as given by Equation (6.12), would be computationally prohibitive since it would require too many operations to evaluate the Hessian matrix and compute its inverse. Alternative approaches, known as quasi-Newton or variable metric methods, are based on that, but instead of calculating the Hessian directly, and then evaluating its inverse, they build up an approximation to the inverse Hessian over a number of steps.

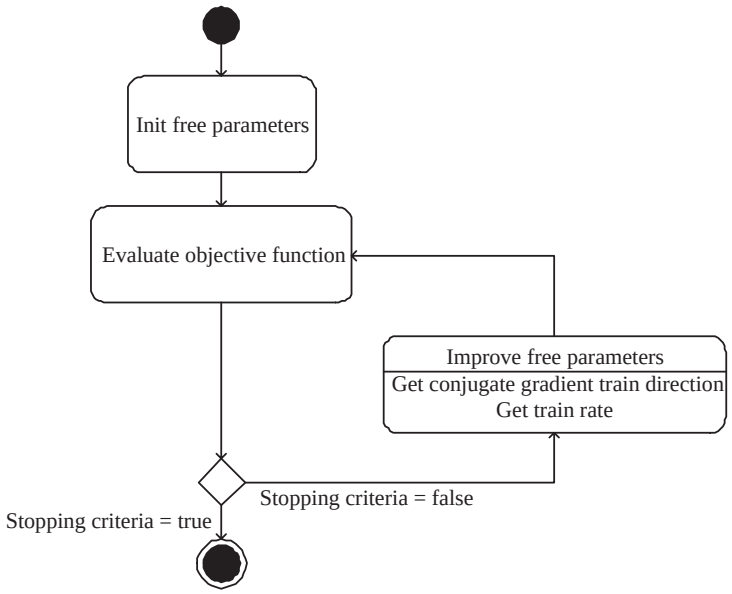


Figure 6.4: Training process with the conjugate gradient training algorithm.

The Hessian matrix is composed of the second partial derivatives of the objective function. The basic idea behind the quasi-Newton or variable metric methods is to approximate $\mathbf{H}^{-1}f(\boldsymbol{\zeta})$ by another matrix $\mathbf{G}f(\boldsymbol{\zeta})$, using only the first partial derivatives of the objective function f. If $\mathbf{H}^{-1}$ is approximated by $\mathbf{G}$, the Newton formula (6.12) can be expressed as

$$\boldsymbol{\zeta}_{i+1} = \boldsymbol{\zeta}_i - \mathbf{G}f(\zeta_i) \cdot \nabla f(\boldsymbol{\zeta}_i) \cdot \eta_i, \tag{6.18}$$

where the training rate η can either set to a fixed value or found by line minimization.

Implementation of Equation (6.18) involves generating a sequence of matrices $\mathbf{G}f(\boldsymbol{\zeta})$ which represent increasingly accurate approximation to the inverse Hessian $\mathbf{H}^{-1}f(\boldsymbol{\zeta})$, sing only information on the first derivatives of the objective function. The problems arising from Hessian matrices which are not positive definite are solved by starting from a positive definite matrix (such as the unit matrix) and ensuring that the update procedure is such that the approximation to the inverse Hessian is guaranteed to remain positive definite. The approximation of the inverse Hessian must be constructed so as to satisfy this condition also.

The two most commonly used update formulae are the Davidon-Fletcher-Powel (DFP) algorithm and the Broyden-Fletcher-Goldfarb-Shanno (BFGS)

algorithm.

The DFP algorithm is given by

$$\begin{aligned}\mathbf{G}f(\boldsymbol{\zeta}_{i+1}) &= \mathbf{G}f(\boldsymbol{\zeta}_i) \\ &+ \frac{(\boldsymbol{\zeta}_{i+1}-\boldsymbol{\zeta}_i)\otimes(\boldsymbol{\zeta}_{i+1}-\boldsymbol{\zeta}_i)}{(\boldsymbol{\zeta}_{i+1}-\boldsymbol{\zeta}_i)\cdot(\mathbf{g}_{i+1}-\mathbf{g}_i)} \\ &+ \frac{[\mathbf{G}f(\boldsymbol{\zeta}_i)\cdot(\mathbf{g}_{i+1}-\mathbf{g}_i)]\otimes[\mathbf{G}f(\boldsymbol{\zeta}_i)\cdot(\mathbf{g}_{i+1}-\mathbf{g}_i)]}{(\mathbf{g}_{i+1}-\mathbf{g}_i)\cdot\mathbf{G}f(\boldsymbol{\zeta}_i)\cdot(\mathbf{g}_{i+1}-\mathbf{g}_i)},\end{aligned} \tag{6.19}$$

$$\begin{aligned}\mathbf{G}_{i+1} &= \mathbf{G}_i \\ &+ \frac{(\boldsymbol{\zeta}_{i+1}-\boldsymbol{\zeta}_i)\otimes(\boldsymbol{\zeta}_{i+1}-\boldsymbol{\zeta}_i)}{(\boldsymbol{\zeta}_{i+1}-\boldsymbol{\zeta}_i)\cdot(\mathbf{g}_{i+1}-\mathbf{g}_i)} \\ &+ \frac{[\mathbf{G}_i\cdot(\mathbf{g}_{i+1}-\mathbf{g}_i)]\otimes[\mathbf{G}_i\cdot(\mathbf{g}_{i+1}-\mathbf{g}_i)]}{(\mathbf{g}_{i+1}-\mathbf{g}_i)\cdot\mathbf{G}_i\cdot(\mathbf{g}_{i+1}-\mathbf{g}_i)},\end{aligned} \tag{6.20}$$

where $\otimes$ denotes the outer or direct product of two vectors, which is a matrix: The ij component of $\mathbf{u}\otimes\mathbf{v}$ is $u_i v_j$.

The BFGS algorithm is exactly the same, but with one additional term

$$\begin{aligned}\mathbf{G}_{i+1} &= \mathbf{G}_i \\ &+ \frac{(\boldsymbol{\zeta}_{i+1}-\boldsymbol{\zeta}_i)\otimes(\boldsymbol{\zeta}_{i+1}-\boldsymbol{\zeta}_i)}{(\boldsymbol{\zeta}_{i+1}-\boldsymbol{\zeta}_i)\cdot(\mathbf{g}_{i+1}-\mathbf{g}_i)} \\ &+ \frac{[\mathbf{G}_i\cdot(\mathbf{g}_{i+1}-\mathbf{g}_i)]\otimes[\mathbf{G}_i\cdot(\mathbf{g}_{i+1}-\mathbf{g}_i)]}{(\mathbf{g}_{i+1}-\mathbf{g}_i)\cdot\mathbf{G}_i\cdot(\mathbf{g}_{i+1}-\mathbf{g}_i)} \\ &+ [(\mathbf{g}_{i+1}-\mathbf{g}_i)\cdot\mathbf{G}_i\cdot(\mathbf{g}_{i+1}-\mathbf{g}_i)]\,\mathbf{u}\otimes\mathbf{u},\end{aligned} \tag{6.21}$$

where the vector $\mathbf{u}$ is given by

$$\begin{aligned}\mathbf{u} &= \frac{(\boldsymbol{\zeta}_{i+1}-\boldsymbol{\zeta}_i)}{(\boldsymbol{\zeta}_{i+1}-\boldsymbol{\zeta}_i)\cdot(\mathbf{g}_{i+1}-\mathbf{g}_i)} \\ &- \frac{\mathbf{G}_i\cdot(\mathbf{g}_{(i+1)}-\mathbf{g}_i)}{(\mathbf{g}_{i+1}-\mathbf{g}_i)\cdot\mathbf{G}_i\cdot(\mathbf{g}_{i+1}-\mathbf{g}_i)}.\end{aligned} \tag{6.22}$$

It has become generally recognized that, empirically, the BFGS scheme is superior than the DFP scheme [45].

A state diagram of the training process of a multilayer perceptron is depicted in Figure 6.5. Improvement of the parameters is performed by first

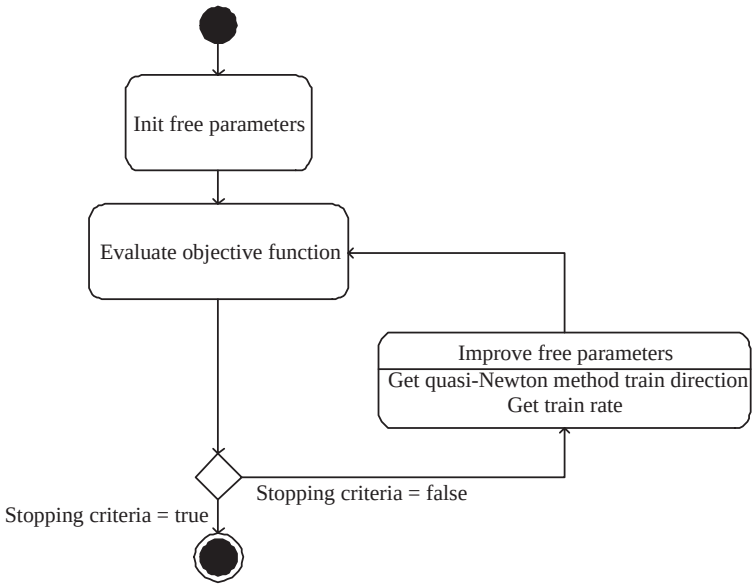


Figure 6.5: Training process with the quasi-Newton method.

obtaining the quasi-Newton train direction and then finding a satisfactory train rate.

The quasi-Newton method is the algorithm of choice in most of the applications included in this User's Guide. This is discussed in detail in [35].

6.7 Random search

Random search is the simplest possible training algorithm for the multilayer perceptron. It is a stochastic method which requires information from the objective function only, and therefore a zero order optimization method.

Random search is based on generating a sequence of improved approximations to the minimum, each derived from the preceding approximation. Thus if ζ_i is the parameters vector in the ith epoch, the new parameters vector is found from the relation

$$\boldsymbol{\zeta}_{i+1} = \boldsymbol{\zeta}_i + \eta_i \mathbf{d}_i, \tag{6.23}$$

for $i = 0, \ldots$ and where η is a prescribed training rate and d is a unit training direction vector. The training rate is reduced after all succesful improvements of the objective function by a given reduction factor.

Unfortunately, convergence is extremely slow in most cases, so this training algorithm is only in practice used to obtain a good initial guess for other more efficient methods.

6.8 Evolutionary algorithm

A global training algorithm for the multilayer perceptron is the evolutionary algorithm, or genetic algorithm. This is a stochastic method based on the mechanics of natural genetics and biological evolution. The evolutionary algorithm requires information from the objective function only, and therefore is a zero order method.

The evolutionary algorithm can be used for problems that are difficult to solve with traditional techniques, including problems that are not well defined or are difficult to model mathematically. It can also be used when computation of the objective function is discontinuous, highly nonlinear, stochastic, or has unreliable or undefined derivatives.

This Section describes a quite general evolutionary algorithm with fitness assignment, selection, recombination and mutation. Different variants on that training operators are also explained in detail.

Evolutionary algorithms operate on a population of individuals applying the principle of survival of the fittest to produce better and better approximations to a solution. At each generation, a new population is created by the process of selecting individuals according to their level of fitness in the problem domain, and recombining them together using operators borrowed from natural genetics. The offspring might also undergo mutation. This process leads to the evolution of populations of individuals that are better suited to their environment than the individuals that they were created from, just as in natural adaptation [43]. A state diagram for the training process with the evolutionary algorithm is depicted in Figure 6.8.

Next the training operators for the evolutionary algorithm together with their corresponding training parameters are described in detail.

Initial population

The evolutionary algorithm starts with an initial population of individuals, represented by vectors of parameters and often chosen at random

$$\mathbf{Z}_0 = \begin{pmatrix} \zeta_{11,0} & \cdots & \zeta_{1d,0} \\ \vdots & \ddots & \vdots \\ \zeta_{p1,0} & \cdots & \zeta_{pd,0} \end{pmatrix},$$

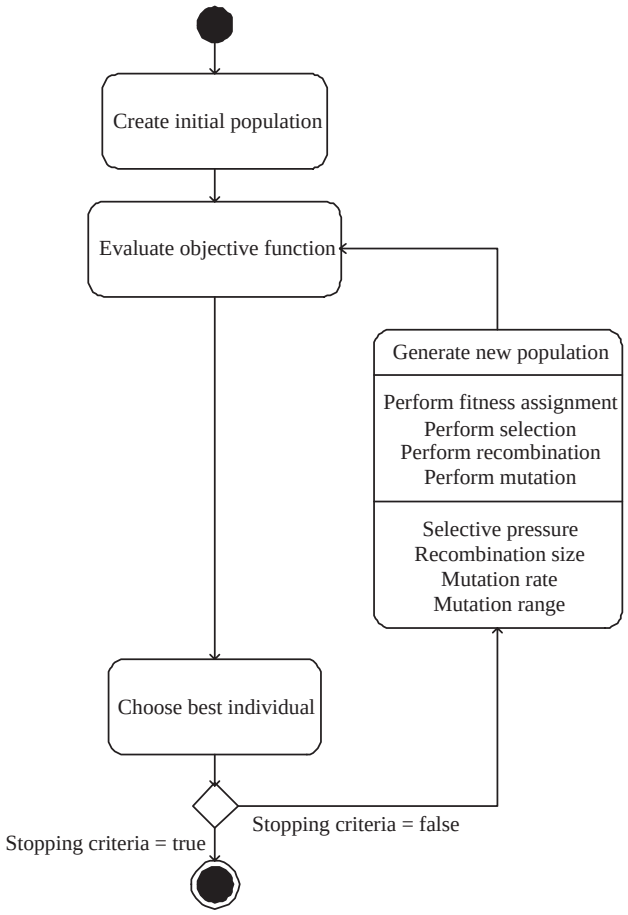


Figure 6.6: Training process with the evolutionary algorithm.

where $\mathbf{Z}$ is called the population matrix. The number of individuals in the population p is called the population size. Due to implementation issues, the population size must be an even number equal or greater than four.

Evaluation

The objective function is evaluated for all the individuals

$$\mathbf{f}_0 = \left(f(\boldsymbol{\zeta}_{1,0}), \ldots, f(\boldsymbol{\zeta}_{p,0}) \right), \tag{6.24}$$

being $\mathbf{f}$ the so called the evaluation vector. The individual with best evaluation $\boldsymbol{\zeta}_0^*$ is then chosen and stored.

Fitness assignment

If no stopping criterium is met the generation of a new population $\mathbf{Z}_1$ starts by performing fitness assignment to the old population $\mathbf{Z}_0$,

$$\boldsymbol{\phi}_0 = (\phi_{1,0}, \ldots, \phi_{p,0}), \tag{6.25}$$

where $\boldsymbol{\phi}$ is called the fitness vector.

There are many methods of computing the fitness of the population. Proportional fitness assignment gives each individual a fitness value dependent on its actual objective function evaluation. In rank-based fitness assignment the evaluation vector is sorted. The fitness assigned to each individual depends only on its position in the individuals rank and not on the actual objective function value. Rank-based fitness assignment behaves in a more robust manner than proportional fitness assignment and, thus, is the method of choice [2] [55].

Linear ranking assigns a fitness to each individual which is linearly proportional to its rank [43]. This operator is controlled by a single parameter called selective pressure, ϖ. Linear ranking allows values for the selective pressure in the interval $[1, 2]$.

Consider p the number of individuals in the population, r_i the rank of some individual i in the population, where the least fit individual has $r = 1$ and the fittest individual has $r = p$, and p the selective pressure. The fitness vector for that individual is calculated as

$$\phi_i(r_i) = 2 - \varpi + 2(\varpi - 1)\frac{r_i - 1}{p - 1}, \tag{6.26}$$

for $i = 1, \ldots, d$.

Selection

After fitness assignment has been performed, some individuals in the population are selected for recombination, according to their level of fitness [3] [2]. Selection determines which individuals are chosen for recombination and how many offspring each selected individual produces,

$$\boldsymbol{\varsigma}_0 = (\varsigma_{1,0}, \ldots, \varsigma_{p,0}), \tag{6.27}$$

where $\boldsymbol{\varsigma}$ is called the selection vector. The elements of this vector are boolean values, that is, an individual can be either selected for recombination (1) or

not (0). In this implementation of the evolutionary algorithm, the number of individuals to be selected is half of the population size, i.e. $d/2$.

The simplest selection operator is roulette-wheel, also called stochastic sampling with replacement [3]. The individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness. A random number is generated and the individual whose segment spans the random number is selected. The process is repeated until the desired number of individuals is obtained (called mating population). This technique is analogous to a roulette wheel with each slice proportional in size to the fitness, see Figure 6.7.

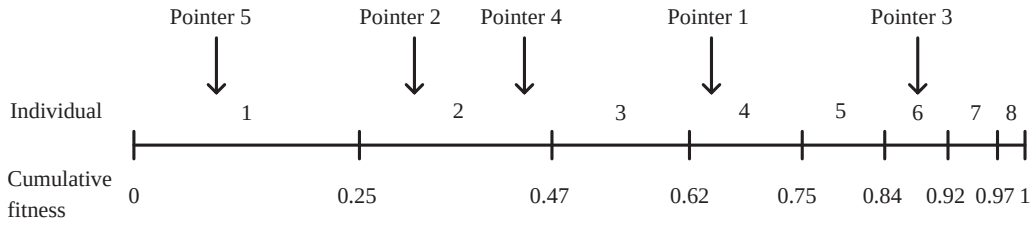


Figure 6.7: Illustration the roulette wheel selection method.

A better selection operator might be stochastic universal sampling [43]. The individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness exactly as in roulette-wheel selection. Here equally spaced pointers are placed over the line as many as there are individuals to be selected. If the number of individuals to be selected is $0.5d$, then the distance between the pointers are $1/0.5d$ and the position of the first pointer is given by a randomly generated number in the range $[0, 1/0.5d]$. Figure 6.8 illustrates this method.

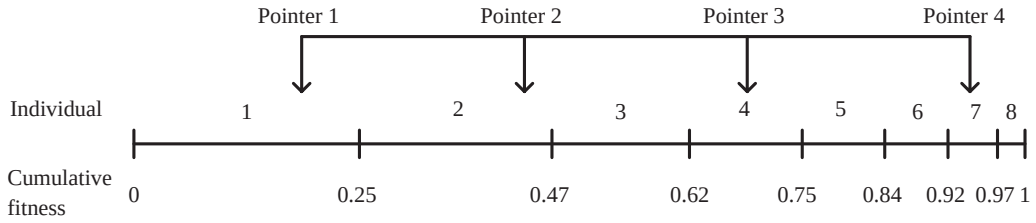


Figure 6.8: Illustration of the stochastic universal sampling selection method.

If elitism is to be used in the selection operator, then the most fitted individual will always be selected to perform recombination.

Recombination

Recombination produces a population matrix by combining the parameters of the selected individuals,

$$\mathbf{P} = \begin{pmatrix} \zeta_{11} & \dots & \zeta_{1d} \\ \dots & \dots & \dots \\ \zeta_{p1} & \dots & \zeta_{pd} \end{pmatrix}$$

There are also many operators to perform recombination. Two of the most used are line recombination and intermediate recombination. Both line and intermediate recombination are controlled by a single parameter called recombination size, denoted d and with allowed values equal or greater than 0. In both operators, the recombination size defines the size of the area for possible offspring. A value of $d = 0$ defines the area for offspring the same size as the area spanned by the parents. Because most variables of the offspring are not generated on the border of the possible area, the area for the variables shrinks over the generations. This effect can be prevented by using a larger recombination size. A value of $d = 0.25$ ensures (statistically), that the variable area of the offspring is the same as the variable area spanned by the variables of the parents.

In line recombination the parameters of the offspring are chosen in the hyperline joining the parameters of the parents [43]. Offspring are therefore produced according to

$$\zeta_i^{(offspring)} = a\zeta_i^{(parent1)} + (1-a)\zeta_i^{(parent2)}, \tag{6.28}$$

for $i = 1, \dots, d$ and with a chosen at random in the interval $[-d, 1+d]$.

Figure 6.9 illustrates the line recombination training operator.

Similarly, in intermediate recombination the parameters of the offspring are chosen somewhere in and around the hypercube defined by the parameters of the parents [43]. Here offspring is produced according to the rule

$$\zeta_i^{(offspring)} = a_i\zeta_i^{(parent1)} + (1-a_i)\zeta_i^{(parent2)}, \tag{6.29}$$

for $i = 1, \dots, d$ and with a_i chosen at random, for each i, in the interval $[-d, 1+d]$.

Figure 6.10 is an illustration of intermediate recombination.

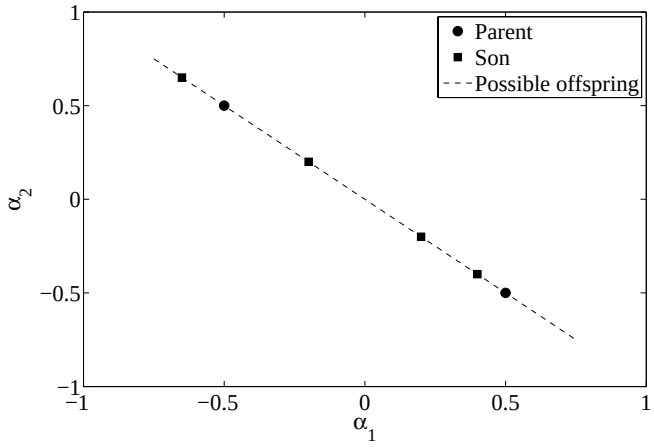


Figure 6.9: Illustration of line recombination.

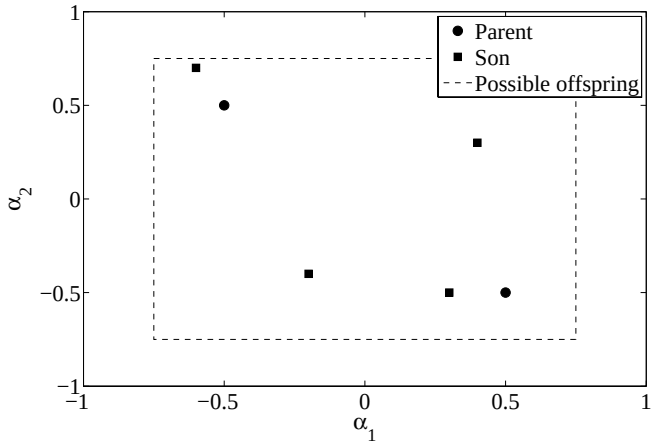


Figure 6.10: Illustration of intermediate recombination.

Mutation

Finally, some offspring undergo mutation in order to obtain the new generation,

$$\mathbf{P}^{(1)} = \begin{pmatrix} \zeta_{11}^{(1)} & \cdots & \zeta_{1d}^{(1)} \\ \cdots & \cdots & \cdots \\ \zeta_{p1}^{(1)} & \cdots & \zeta_{pd}^{(1)} \end{pmatrix}$$

The probability of mutating a parameter is called the mutation rate and

denoted p [43]. The mutation rate allows values in the interval $[0, 1]$. On the other hand, mutation is achieved by adding or subtracting a random quantity to the parameter. In this way, each parameter ζ_i subject to mutation is mutated to become ζ_i',

$$\zeta_i' = \zeta_i + \Delta\zeta_i. \tag{6.30}$$

The most common kinds of mutation procedures are uniform mutation and normal mutation. Both the uniform and normal mutation operators are controlled by a single parameter called mutation range, r, which allows values equal or greater than 0.

In uniform mutation, $\Delta\zeta_i$ is a number chosen at random in the interval $[0, r]$. Figure 6.11 illustrates the effect of uniform mutation for the case of two parameters.

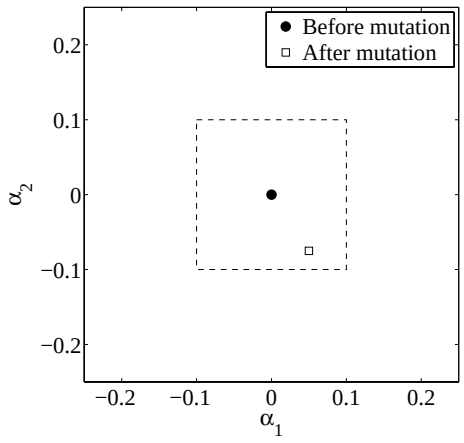


Figure 6.11: Illustration of uniform mutation.

In normal mutation, $\Delta\zeta_i$ is a value obtained from a normal distribution with mean 0 and standard deviation r. Figure 6.12 illustrates the effect of normal mutation for the case of two parameters.

Example 49

Stopping criteria

The whole fitness assignment, selection recombination and mutation process is repeated until a stopping criterium is satisfied. Some common stopping criteria for the evolutionary algorithm are:

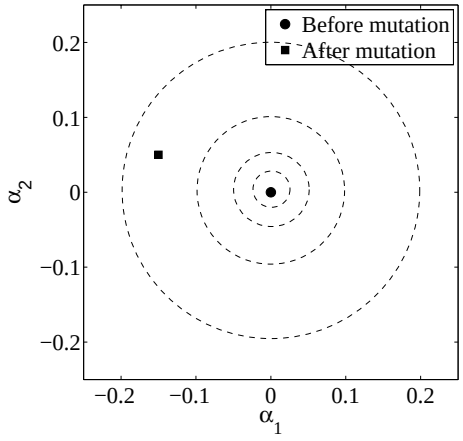


Figure 6.12: Illustration of normal mutation.

1. Evaluation of the best individual ever reaches a goal value.
2. The mean evaluation of the population reaches a goal value.
3. The standard deviation of the evaluation vector reaches a goal value.
4. A maximum amount of computing time is exceeded.
5. A maximum number of generations is reached.

6.9 The TrainingAlgorithm classes

`Flood` includes the abstract class TrainingAlgorithm to represent the concept of training algorithm. As this class is abstract, it cannot be instantiated, and concrete training algorithms must be derived.

Constructors

The training algorithm class is abstract and it cannot be instantiated. The concrete classes RandomSearch, GradientDescent, ConjugateGradient, NewtonMethod, QuasiNewtonMethod and EvolutionaryAlgorithm represent the concepts of the different training algorithms described in this chapter.

To construct a concrete training algorithm object associated to a concrete objective functional object we do the following

```
MockTrainingAlgorithm mock_training_algorithm(&mock_objective_functional);
```

where mock_objective_functional is some concrete objective functional object.

Members

The training algorithm class contains:
- A relationship to an objective functional object.
- A set of training operators.
- A set of training parameters.
- A set of stopping criteria.

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

Methods

Derived classes must implement the pure virtual train method. This starts the training process and stops it when some stopping criterium is satisfied. The use is as follows:

```
mock_training_algorithm.train();
```

where mock_training_algorithm is some concrete training algorithm object.

File format

The default file format of a training algorithm object is of XML-type, and it is listed below.

```
<Flood version='3.0' class='TrainingAlgorithm'>
<TrainingRateMethod>
training_rate_method
</TrainingRateMethod>
<BracketingFactor>
bracketing_factor
</BracketingFactor>
<FirstTrainingRate>
first_training_rate
</FirstTrainingRate>
<TrainingRateTolerance>
training_rate_tolerance
</TrainingRateTolerance>
<WarningParametersNorm>
warning_parameters_norm
</WarningParametersNorm>
<WarningGradientNorm>
warning_gradient_norm
</WarningGradientNorm>
<WarningTrainingRate>
warning_training_rate
</WarningTrainingRate>
<ErrorParametersNorm>
```

```
error_parameters_norm
</ErrorParametersNorm>
<ErrorGradientNorm>
error_gradient_norm
</ErrorGradientNorm>
<ErrorTrainingRate>
error_training_rate
</ErrorTrainingRate>
<MinimumParametersIncrementNorm>
minimum_parameters_increment_norm
</MinimumParametersIncrementNorm>
<MinimumEvaluationImprovement>
minimum_evaluation_improvement
</MinimumEvaluationImprovement>
<EvaluationGoal>
evaluation_goal
</EvaluationGoal>
<EarlyStopping>
early_stopping
</EarlyStopping>
<GradientNormGoal>
gradient_norm_goal
</GradientNormGoal>
<MaximumEpochsNumber>
maximum_epochs_number
</MaximumEpochsNumber>
<MaximumTime>
maximum_time
</MaximumTime>
<ReserveParametersHistory>
reserve_parameters_history
</ReserveParametersHistory>
<ReserveParametersNormHistory>
reserve_parameters_norm_history
</ReserveParametersNormHistory>
<ReserveEvaluationHistory>
reserve_evaluation_history
</ReserveEvaluationHistory>
<ReserveValidationErrorHistory>
reserve_validation_error_history
</ReserveValidationErrorHistory>
<ReserveGradientHistory>
reserve_gradient_history
</ReserveGradientHistory>
<ReserveGradientNormHistory>
reserve_gradient_norm_history
</ReserveGradientNormHistory>
<ReserveInverseHessianHistory>
reserve_inverse_Hessian_history
</ReserveInverseHessianHistory>
<ReserveTrainingDirectionHistory>
reserve_training_direction_history
</ReserveTrainingDirectionHistory>
<ReserveTrainingDirectionNormHistory>
reserve_training_direction_norm_history
</ReserveTrainingDirectionNormHistory>
<ReserveTrainingRateHistory>
reserve_training_rate_history
</ReserveTrainingRateHistory>
<ReserveElapsedTimeHistory>
reserve_elapsed_time_history
</ReserveElapsedTimeHistory>
```

```
<Display>
display
</Display>
<DisplayPeriod>
display_period
</DisplayPeriod>
```

Derived classes

To construct a RandomSearch object associated to some concrete objective functional object, we use the following sentence

```
RandomSearch random_search(&mock_objective_functional);
```

where &mock_objective_functional is a reference to a concrete ObjectiveFunctional object which has been previously constructed.

Some stopping criteria for the random search object are listed below

```
random_search.set_evaluation_goal(0.001);
quasi_newton_method.set_maximum_time(1000.0);
```

The method train trains a multilayer perceptron according to the random search method.

```
random_search.train();
```

We can save or load a random search object to or from a XML-type file, by using the methods save(**char***) and load(**char***), respectively.

```
random_search.save("RandomSearch.dat");
```

The file format of a random search training algorithm XML-type file is listed below.

```
<Flood version="3.0" class='RandomSearch'>
<FirstTrainingRate>
first_training_rate
</FirstTrainingRate>
<TrainingRateReductionPeriod>
training_rate_reduction_period
</TrainingRateReductionPeriod>
<TrainingRateReductionFactor>
training_rate_reduction_factor
</TrainingRateReductionFactor>
<WarningParametersNorm>
warning_parameters_norm
</WarningParametersNorm>
<ErrorParametersNorm>
error_parameters_norm
</ErrorParametersNorm>
<EvaluationGoal>
evaluation_goal
</EvaluationGoal>
<MaximumEpochsNumber>
maximum_epochs_number
</MaximumEpochsNumber>
```

```
<MaximumTime>
maximum_time
</MaximumTime>
<ReservePotentialParametersHistory>
reserve_potential_parameters_history
</ReservePotentialParametersHistory>
<ReserveParametersHistory>
reserve_parameters_history
</ReserveParametersHistory>
<ReservePotentialParametersNormHistory>
reserve_potential_parameters_norm_history
</ReservePotentialParametersNormHistory>
<ReserveParametersNormHistory>
reserve_parameters_norm_history
</ReserveParametersNormHistory>
<ReservePotentialEvaluationHistory>
reserve_potential_evaluation_history
</ReservePotentialEvaluationHistory>
<ReserveEvaluationHistory>
reserve_evaluation_history
</ReserveEvaluationHistory>
<ReserveElapsedTimeHistory>
reserve_elapsed_time_history
</ReserveElapsedTimeHistory>
<DisplayPeriod>
display_period
</DisplayPeriod>
<Display>
display
</Display>
```

GradientDescent

To construct a gradient descent object associated to a objective functional object called mock_objective_functional we can do

```
GradientDescent gradient_descent(&mock_objective_functional);
```

Next, some training operators, training parameters, stopping criteria and training history members are set.

```
gradient_descent.set_training_rate_method(GradientDescent::BrentMethod);
gradient_descent.set_training_rate_tolerance(1.0e-6);
gradient_descent.set_minimum_evaluation_improvement_(1.0e-9);
gradient_descent.set_reserve_evaluation_history(true);
```

To train a the multilayer perceptron associated to that objective functional we write

```
gradient_descent.train();
```

The methods save and load save or load a gradient descent object to or from a XML-type file, respectively. The training history can also be saved using the save_training_history_method.

```
gradient_descent.save("GradientDescent.dat");
gradient_descent.save_training_history("TrainingHistory.dat");
```

The file format of this class is listed below.

```
<Flood version="3.0" class='GradientDescent'>
<TrainingRateMethod>
training_rate_method
</TrainingRateMethod>
<BracketingFactor>
bracketing_factor
</BracketingFactor>
<FirstTrainingRate>
first_training_rate
</FirstTrainingRate>
<FirstTrainingRate>
training_rate_tolerance
</FirstTrainingRate>
<WarningTrainingRate>
warning_training_rate
</WarningTrainingRate>
<ErrorTrainingRate>
error_training_rate
<ErrorTrainingRate>
<MinimumParametersIncrementNorm>
minimum_parameters_increment_norm
</MinimumParametersIncrementNorm>
<EvaluationGoal>
evaluation_goal
</EvaluationGoal>
<EarlyStopping>
early_stopping
</EarlyStopping>
<MinimumEvaluationImprovement>
minimum_evaluation_improvement
</MinimumEvaluationImprovement>
<GradientNormGoal>
gradient_norm_goal
</GradientNormGoal>
<MaximumEpochsNumber>
maximum_epochs_number
</MaximumEpochsNumber>
<MaximumTime>
maximum_time
</MaximumTime>
<ReserveElapsedTimeHistory>
reserve_elapsed_time_history
</ReserveElapsedTimeHistory>
<ReserveParametersHistory>
reserve_parameters_history
</ReserveParametersHistory>
<ReserveParametersNormHistory>
reserve_parameters_norm_history
</ReserveParametersNormHistory>
<ReserveEvaluationHistory>
reserve_evaluation_history
</ReserveEvaluationHistory>
<ReserveValidationErrorHistory>
reserve_validation_error_history
</ReserveValidationErrorHistory>
<ReserveGradientHistory>
reserve_gradient_history
</ReserveGradientHistory>
<ReserveGradientNormHistory>
reserve_gradient_norm_history
```

```
</ReserveGradientNormHistory>
<ReserveTrainingDirectionHistory>
reserve_training_direction_history
</ReserveTrainingDirectionHistory>
<ReserveTrainingDirectionNormHistory>
reserve_training_direction_norm_history
</ReserveTrainingDirectionNormHistory>
<ReserveTrainingRateHistory>
reserve_training_rate_history
</ReserveTrainingRateHistory>
<WarningParametersNorm>
warning_parameters_norm
</WarningParametersNorm>
<WarningGradientNorm>
warning_gradient_norm
</WarningGradientNorm>
<Display>
display
</Display>
<DisplayPeriod>
display_period
</DisplayPeriod>
```

Newton method

The constructor of the Newton method is used as follows

```
NewtonMethod Newton_method(&mock_objective_functional);
```

To train the multilayer perceptron associated with that objective functional we use

```
Newton_method.train();
```

The file format for saving or loading objects of this class is listed below.

```
<Flood version="3.0" class='NewtonMethod'>
<TrainingRateMethod>
training_rate_method
</TrainingRateMethod>
<BracketingFactor>
bracketing_factor
</BracketingFactor>
<FirstTrainingRate>
first_training_rate
</FirstTrainingRate>
<TrainingRateTolerance>
training_rate_tolerance
</TrainingRateTolerance>
<WarningParametersNorm>
warning_parameters_norm
</WarningParametersNorm>
<WarningGradientNorm>
warning_gradient_norm
</WarningGradientNorm>
<WarningTrainingRate>
warning_training_rate
</WarningTrainingRate>
<ErrorParametersNorm>
error_parameters_norm
```

```
</ErrorParametersNorm>
<ErrorGradientNorm>
error_gradient_norm
</ErrorGradientNorm>
<ErrorTrainingRate>
error_training_rate
</ErrorTrainingRate>
<MinimumParametersIncrementNorm>
minimum_parameters_increment_norm
</MinimumParametersIncrementNorm>
<MinimumEvaluationImprovement>
minimum_evaluation_improvement
</MinimumEvaluationImprovement>
<EvaluationGoal>
evaluation_goal
</EvaluationGoal>
<EarlyStopping>
early_stopping
</EarlyStopping>
<GradientNormGoal>
gradient_norm_goal
</GradientNormGoal>
<MaximumEpochsNumber>
maximum_epochs_number
</MaximumEpochsNumber>
<MaximumTime>
maximum_time
</MaximumTime>
<ReserveParametersHistory>
reserve_parameters_history
</ReserveParametersHistory>
<ReserveParametersNormHistory>
reserve_parameters_norm_history
</ReserveParametersNormHistory>
<ReserveEvaluationHistory>
reserve_evaluation_history
</ReserveEvaluationHistory>
<ReserveValidationErrorHistory>
reserve_validation_error_history
</ReserveValidationErrorHistory>
<ReserveGradientHistory>
reserve_gradient_history
</ReserveGradientHistory>
<ReserveGradientNormHistory>
reserve_gradient_norm_history
</ReserveGradientNormHistory>
<ReserveInverseHessianHistory>
reserve_inverse_Hessian_history
</ReserveInverseHessianHistory>
<ReserveTrainingDirectionHistory>
reserve_training_direction_history
</ReserveTrainingDirectionHistory>
<ReserveTrainingDirectionNormHistory>
reserve_training_direction_norm_history
</ReserveTrainingDirectionNormHistory>
<ReserveTrainingRateHistory>
reserve_training_rate_history
</ReserveTrainingRateHistory>
<ReserveElapsedTimeHistory>
reserve_elapsed_time_history
</ReserveElapsedTimeHistory>
<Display>
```

```
display
</Display>
<DisplayPeriod>
display_period
</DisplayPeriod>
```

Conjugate gradient

The conjugate gradient class is very similar to the gradient descent class. The next listing shows how to use that.

```
ConjugateGradient conjugate_gradient(&mock_objective_functional);

conjugate_gradient.set_training_direction_method(ConjugateGradient::PolakRibiere);
conjugate_gradient.set_training_rate_method(ConjugateGradient::GoldenSection);

conjugate_gradient.set_minimum_evaluation_improvement(1.0e-9);
conjugate_gradient.set_maximum_epochs_number(500);

conjugate_gradient.train();

conjugate_gradient.save("ConjugateGradient.dat");
conjugate_gradient.save_training_history("ConjugateGradientTrainingHistory.dat");
```

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

```
<Flood version="3.0" class='ConjugateGradient'>
<TrainingDirectionMethod>
training_direction_method
</TrainingDirectionMethod>
<TrainingRateMethod>
training_rate_method
</TrainingRateMethod>
<BracketingFactor>
bracketing_factor
</BracketingFactor>
<FirstTrainingRate>
first_training_rate
</FirstTrainingRate>
<FirstTrainingRate>
training_rate_tolerance
</FirstTrainingRate>
<WarningParametersNorm>
warning_parameters_norm
</WarningParametersNorm>
<WarningGradientNorm>
warning_gradient_norm
</WarningGradientNorm>
<WarningTrainingRate>
warning_training_rate
</WarningTrainingRate>
<ErrorParametersNorm>
error_parameters_norm
</ErrorParametersNorm>
<ErrorGradientNorm>
error_gradient_norm
</ErrorGradientNorm>
```

```
<ErrorTrainingRate>
error_training_rate
</ErrorTrainingRate>
<MinimumParametersIncrementNorm>
minimum_parameters_increment_norm
</MinimumParametersIncrementNorm>
<MinimumEvaluationImprovement>
minimum_evaluation_improvement
</MinimumEvaluationImprovement>
<EvaluationGoal>
evaluation_goal
</EvaluationGoal>
<EarlyStopping>
early_stopping
</EarlyStopping>
<GradientNormGoal>
gradient_norm_goal
</GradientNormGoal>
<MaximumEpochsNumber>
maximum_epochs_number
</MaximumEpochsNumber>
<MaximumTime>
maximum_time
</MaximumTime>
<ReserveParametersHistory>
reserve_parameters_history
</ReserveParametersHistory>
<ReserveParametersNormHistory>
reserve_parameters_norm_history
</ReserveParametersNormHistory>
<ReserveEvaluationHistory>
reserve_evaluation_history
</ReserveEvaluationHistory>
<ReserveValidationErrorHistory>
reserve_validation_error_history
</ReserveValidationErrorHistory>
<ReserveGradientHistory>
reserve_gradient_history
</ReserveGradientHistory>
<ReserveGradientNormHistory>
reserve_gradient_norm_history
</ReserveGradientNormHistory>
<ReserveTrainingDirectionHistory>
reserve_training_direction_history
</ReserveTrainingDirectionHistory>
<ReserveTrainingDirectionNormHistory>
reserve_training_direction_norm_history
</ReserveTrainingDirectionNormHistory>
<ReserveTrainingRateHistory>
reserve_training_rate_history
</ReserveTrainingRateHistory>
<ReserveElapsedTimeHistory>
reserve_elapsed_time_history
</ReserveElapsedTimeHistory>
<Display>
display
</Display>
<DisplayPeriod>
display_period
</DisplayPeriod>
```

Quasi-Newton method

To construct a QuasiNewtonMethod object associated to a concrete objective functional object we use the following sentence

```
QuasiNewtonMethod quasi_newton_method(&mock_objective_functional);
```

where &mock_objective_functional is a reference to a concrete objective functional object which has been previously constructed.

The use of training operators for this method is illustrated below.

```
quasi_newton_method.set_inverse_Hessian_approximation_method(QuasiNewtonMethod::BFGS);
quasi_newton_method.set_training_rate_method(QuasiNewtonMethod::BrentMethod);
```

Next we set some training parameters to the quasi-Newton method.

```
conjugate_gradient.set_minimum_evaluation_improvement(1.0e-9);
conjugate_gradient.set_maximum_epochs_number(500);
```

The following sentences set some stopping criteria for the quasi-Newton method.

```
quasi_newton_method.set_evaluation_goal(0.001);
quasi_newton_method.set_gradient_norm_goal(0.001);
quasi_newton_method.set_maximum_time(1000.0);
quasi_newton_method.set_maximum_epochs_number(1000);
```

The history of some training variables are reserved next.

```
quasi_newton_method.set_reserve_evaluation_history(true);
quasi_newton_method.set_reserve_gradient_norm_history(true);
```

The method train trains a multilayer perceptron according to the quasi-Newton method.

```
quasi_newton_method.train();
```

We can save or load a quasi-Newton method object to or from a data file, by using the save(**char***) and load(**char***) methods, respectively. Similarly, the training history is saved with the save_training_history (**char***) method.

```
quasi_newton_method.save("QuasiNewtonMethod.dat");
quasi_newton_method.save_training_history("TrainingHistory.dat");
```

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

```
<Flood version="3.0" class='QuasiNewtonMethod'>
<TrainingRateMethod>
training_rate_method
</TrainingRateMethod>
<BracketingFactor>
bracketing_factor
</BracketingFactor>
<FirstTrainingRate>
```

```
first_training_rate
</FirstTrainingRate>
<TrainingRateTolerance>
training_rate_tolerance
</TrainingRateTolerance>
<WarningParametersNorm>
warning_parameters_norm
</WarningParametersNorm>
<WarningGradientNorm>
warning_gradient_norm
</WarningGradientNorm>
<WarningTrainingRate>
warning_training_rate
</WarningTrainingRate>
<ErrorParametersNorm>
error_parameters_norm
</ErrorParametersNorm>
<ErrorGradientNorm>
error_gradient_norm
</ErrorGradientNorm>
<ErrorTrainingRate>
error_training_rate
</ErrorTrainingRate>
<MinimumParametersIncrementNorm>
minimum_parameters_increment_norm
</MinimumParametersIncrementNorm>
<MinimumEvaluationImprovement>
minimum_evaluation_improvement
</MinimumEvaluationImprovement>
<EvaluationGoal>
evaluation_goal
</EvaluationGoal>
<EarlyStopping>
early_stopping
</EarlyStopping>
<GradientNormGoal>
gradient_norm_goal
</GradientNormGoal>
<MaximumEpochsNumber>
maximum_epochs_number
</MaximumEpochsNumber>
<MaximumTime>
maximum_time
</MaximumTime>
<ReserveParametersHistory>
reserve_parameters_history
</ReserveParametersHistory>
<ReserveParametersNormHistory>
reserve_parameters_norm_history
</ReserveParametersNormHistory>
<ReserveEvaluationHistory>
reserve_evaluation_history
</ReserveEvaluationHistory>
<ReserveValidationErrorHistory>
reserve_validation_error_history
</ReserveValidationErrorHistory>
<ReserveGradientHistory>
reserve_gradient_history
</ReserveGradientHistory>
<ReserveGradientNormHistory>
reserve_gradient_norm_history
</ReserveGradientNormHistory>
```

```
<ReserveInverseHessianHistory>
reserve_inverse_Hessian_history
</ReserveInverseHessianHistory>
<ReserveTrainingDirectionHistory>
reserve_training_direction_history
</ReserveTrainingDirectionHistory>
<ReserveTrainingDirectionNormHistory>
reserve_training_direction_norm_history
</ReserveTrainingDirectionNormHistory>
<ReserveTrainingRateHistory>
reserve_training_rate_history
</ReserveTrainingRateHistory>
<ReserveElapsedTimeHistory>
reserve_elapsed_time_history
</ReserveElapsedTimeHistory>
<Display>
display
</Display>
<DisplayPeriod>
display_period
</DisplayPeriod>
```

EvolutionaryAlgorithm

To construct a EvolutionaryAlgorithm object associated to a concrete objective functional object we can use the following sentence

```
EvolutionaryAlgorithm evolutionary_algorithm(&mock_objective_functional);
```

where &mock_objective_functional is a reference to that objective functional object.

In order to set a new number of individuals in the population we use the method set_population_size(**int**).

```
evolutionary_algorithm.set_population_size(100);
```

The following sentences set some stopping criteria for the evolutionary algorithm

```
evolutionary_algorithm.set_evaluation_goal(0.001);
evolutionary_algorithm.set_maximum_time(1000.0);
evolutionary_algorithm.set_maximum_generations_number(1000);
```

The method train trains a multilayer perceptron according to the evolutionary algorithm method.

```
evolutionary_algorithm.train();
```

We can also save or load a evolutionary algorithm object to or from a data file, by using the methods save(**char***) and load(**char***), respectively.

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

```
<Flood version="3.0" class='EvolutionaryAlgorithm'>
Training operators
```

```
<FitnessAssignmentMethod>
fitness_assignment_method
</FitnessAssignmentMethod>
<SelectionMethod>
selection_method
</SelectionMethod>
<RecombinationMethod>
recombination_method
</RecombinationMethod>
<MutationMethod>
mutation_method
</MutationMethod>
<PopulationSize>
Training parameters
<PopulationSize>
population_size
</PopulationSize>
<Elitism>
perform_elitism
</Elitism>
<SelectivePressure>
selective_pressure
</SelectivePressure>
<RecombinationSize>
recombination_size
</RecombinationSize>
<MutationRate>
mutation_rate
</MutationRate>
<MutationRange>
mutation_range
</MutationRange>
Stopping criteria
<EvaluationGoal>
evaluation_goal
</EvaluationGoal>
<MeanEvaluationGoal>
mean_evaluation_goal
</MeanEvaluationGoal>
<StandardDeviationEvaluationGoal>
standard_deviation_evaluation_goal
</StandardDeviationEvaluationGoal>
<MaximumGenerationsNumber>
maximum_generations_number
</MaximumGenerationsNumber>
<MaximumTime>
maximum_time
</MaximumTime>
<EarlyStopping>
early_stopping
</EarlyStopping>
Training history
<ReservePopulationHistory>
reserve_population_history
</ReservePopulationHistory>
<ReserveMeanNormHistory>
reserve_mean_norm_history
</ReserveMeanNormHistory>
<ReserveStandardDeviationNormHistory>
reserve_standard_deviation_norm_history
</ReserveStandardDeviationNormHistory>
<ReserveBestNormHistory>
```

```
reserve_best_norm_history
</ReserveBestNormHistory>
<ReserveMeanEvaluationHistory>
reserve_mean_evaluation_history
</ReserveMeanEvaluationHistory>
<ReserveStandardDeviationEvaluationHistory>
reserve_standard_deviation_evaluation_history
</ReserveStandardDeviationEvaluationHistory>
<ReserveBestEvaluationHistory>
reserve_best_evaluation_history
</ReserveBestEvaluationHistory>
Display
<WarningParametersNorm>
warning_parameters_norm
</WarningParametersNorm>
<Display>
display
</Display>
<DisplayPeriod>
display_period
</DisplayPeriod>
```

Chapter 7

Function regression

The function regression problem can be regarded as the problem of ap- proximating a function from data. 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. The function regression problem is formulated from a variational point of view.

7.1 Problem formulation

Introduction

The function regression problem [24] can be regarded as the problem of approximating a function from data.

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.

The basic goal in a function regression problem is to model the conditional distribution of the output variables, conditioned on the input variables [8]. This function is called the regression function.

The formulation of a function regression problem requires:
- An input-target data set.
- A multilayer perceptron.
- An error functional.
- A training algorithm.

- A testing method.

Input-target data set

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.

Table 7.1 shows the format of an input-target data set for function regression. It consists of n input variables, $x_1, \ldots, x_n$, and m target variables, $t_1, \ldots, t_m$, comprising Q instances.

$x_1^{(1)}$	$\ldots$	$x_n^{(1)}$	$t_1^{(1)}$	$\ldots$	$t_m^{(1)}$
$x_1^{(2)}$	$\ldots$	$x_n^{(2)}$	$t_1^{(2)}$	$\ldots$	$t_m^{(2)}$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$x_1^{(Q)}$	$\ldots$	$x_n^{(Q)}$	$t_1^{(Q)}$	$\ldots$	$t_m^{(Q)}$

Table 7.1: Input-target data set.

When solving function regression problems it is always convenient to split the input-target data set into a training, a validation 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, validation 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, validation and testing instances.

A simple statistical analysis must be always performed in order to chech for data consistency. Table 7.2 depicts the basic statistics of an input-target data set. It includes the mean, standard deviation, minimum and maximum values of input and target variables for the whole data set and the training, validation 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 training and validation sets with the training 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 training data for mean 0 and standard deviation 1, and the validation data for similar mean and standard deviation values. The expression of this scaling method is

Variable	μ	σ	min	max
x_1	All Training Validation Testing	All Training Validation Testing	All Training Validation Testing	All Training Validation Testing
...	...	...	...	...
x_n	All Training Validation Testing	All Training Validation Testing	All Training Validation Testing	All Training Validation Testing
t_1	All Training Validation Testing	All Training Validation Testing	All Training Validation Testing	All Training Validation Testing
...	...	...	...	...
t_m	All Training Validation Testing	All Training Validation Testing	All Training Validation Testing	All Training Validation Testing

Table 7.2: Input-target data set statistics.

$$\overline{\mathbf{x}} = \frac{\mathbf{x} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}, \tag{7.1}$$

$$\overline{\mathbf{t}} = \frac{\mathbf{t} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}, \tag{7.2}$$

where the overline means scaled data and the mean and standard deviation values refer to the training subset.

The minimum and maximum method scales the training data for minimum -1 and maximum 1, and the validation data for similar minimum and maximum values. The expression here is

$$\overline{\mathbf{x}} = 2\frac{\mathbf{x} - \mathbf{min}}{\mathbf{max} - \mathbf{min}} - 1, \tag{7.3}$$

$$\overline{\mathbf{t}} = 2\frac{\mathbf{t} - \mathbf{min}}{\mathbf{max} - \mathbf{min}} - 1, \tag{7.4}$$

where, as before, the overline means scaled data and the minimum and maximum values refer to the training subset.

Multilayer perceptron

A multilayer perceptron is used to represent the regression function. The number of inputs in this multilayer perceptron must be equal to the number of inputs in the data set, and the number of outputs must be the number of targets. On the other hand, the number of hidden layers and the size of each layer are up to the designer. In general, one hidden layer will be enough. A default value to start with for the size of that layer could be

$$s^{(1)} = round\left(\sqrt{n^2 + m^2}\right). \tag{7.5}$$

Please note that the complexity which is needed depends very much on the problem at hand, and Equation (7.5) is just a rule of thumb.

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 input-target 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.

This multilayer perceptron spans a function space V of dimension d. The elements of that space, $\mathbf{y} : X \to Y$, are of the form

$$\mathbf{y} = \mathbf{y}(\mathbf{x}). \tag{7.6}$$

That parameterized space of functions will be the basis to aproximate the regression function.

Error functional

The regression function can be evaluated quantitatively by means of the error functional $E : V \to \mathbb{R}$, which is of the form

$$E = E[\mathbf{y}(\mathbf{x})].$$

The goal in a function regression problem for the multilayer perceptron is to obtain a function $\mathbf{y}^*(\mathbf{x})$ which minimizes the error functional $E[\mathbf{y}(\mathbf{x})]$. More specifically, this problem can be formulated as follows:

Problem 5 (Function regression problem) *Let V be the function space spanned by a multilayer perceptron. Find a function $\mathbf{y}^*(\mathbf{x}) \in V$ for which the error functional $E : V \to \mathbb{R}$ given by*

$$E = E[\mathbf{y}(\mathbf{x})]$$

takes on a minimum value. The function $\mathbf{y}^*(\mathbf{x})$ is called the regression function.

In that way, the function regression problem is formulated as a variational problem.

The sum squared error

One of the most common error functionals used in function regression is the sum squared error. This objective functional is measured on a training data set. The sum of the squares of the errors is used instead of the errors absolute values because this allows the objective function to be treated as a continuous differentiable function. It is written as a sum, over all the samples in the training data set, of a squared error defined for each sample separately.

The expression for the sum squared error functional, $E : V \to \mathbb{R}$, is given by

$$E[\mathbf{y}(\mathbf{x})] = \sum_{q=1}^{Q} \left(\mathbf{y}(\mathbf{x}^{(q)}) - \mathbf{t}^{(q)}\right)^2. \tag{7.7}$$

Both the gradient, $\nabla e(\boldsymbol{\zeta})$, and the Hessian, $\mathbf{H}e(\boldsymbol{\zeta})$ of the sum squared error function can be computed analytically by means of the back-propagation algorithm.

The mean squared error

The mean squared error error functional has the same properties than the sum squared error and the advantage that its value does not grow with the size of the training data set [8].

The expression for the mean squared error, $E : V \to \mathbb{R}$, is

$$E[\mathbf{y}(\mathbf{x})] = \frac{1}{Q} \sum_{q=1}^{Q} \left(\mathbf{y}(\mathbf{x}^{(q)}) - \mathbf{t}^{(q)}\right)^2. \tag{7.8}$$

As before, the gradient vector and the Hessian matrix of the mean squared error function can be computed analytically by means of back-propagation.

The root mean squared error

The expression for the root mean squared error is given by

$$E[\mathbf{y}(\mathbf{x})] = \sqrt{\frac{1}{Q}\sum_{q=1}^{Q}\left(\mathbf{y}(\mathbf{x}^{(q)}) - \mathbf{t}^{(q)}\right)^2}. \tag{7.9}$$

The gradient $\nabla e(\boldsymbol{\zeta})$ and the Hessian $\mathbf{H}e(\boldsymbol{\zeta})$ of the root mean squared error function can be computed analytically by means of the back-propagation algorithm.

The normalized squared error

Another useful objective functional for function regression is the normalized squared error, which takes the form

$$E[\mathbf{y}(\mathbf{x})] = \frac{\sum_{q=1}^{Q}\left(\mathbf{y}(\mathbf{x}^{(q)}) - \mathbf{t}^{(q)}\right)^2}{\sum_{q=1}^{Q}\left(\mathbf{t}^{(q)} - \bar{\mathbf{t}}\right)^2}. \tag{7.10}$$

As before, the gradient $\nabla e(\boldsymbol{\zeta})$ and the Hessian $\mathbf{H}e(\boldsymbol{\zeta})$ of the normalized squared error function can be computed analytically with the back-propagation algorithm.

The normalized squared error has the advantage that its value does not grow with the size of the input-target data set. If it has a value of unity then the neural network is predicting the data 'in the mean', while a value of zero means perfect prediction of the data [8]. As a consequence, the normalized squared error takes the same value for preprocessed data without pre and postprocessing method in the multilayer perceptron and non-preprocessed data with pre and postprocessing method in the multilayer perceptron.

The Minkowski error

One of the potential difficulties of the sum squared error objective functional is that it can receive a too large contribution from points which have large errors [8]. If there are long tails on the distribution then the solution can be dominated by a very small number of points which have particularly large error. In such occasions, in order to achieve good generalization, it is preferable to chose a more suitable objective functional.

We can derive more general error functions than the sum squared error for the case of a supervised learning problem. Omitting irrelevant constants, the Minkowski R-error is defined as

$$E[\mathbf{y}(\mathbf{x})] = \sum_{q=1}^{Q} \left(\mathbf{y}(\mathbf{x}^{(q)}) - \mathbf{t}^{(q)}\right)^R. \tag{7.11}$$

This reduces to the usual sum squared error when $R = 2$ [8].

The gradient and the Hessian of the root mean squared error function can also be computed with back-propagation.

Training algorithm

The training algorithm is entrusted to solve the reduced function optimization problem by minimizing the error 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. A quasi-Newton method with BFGS training direction and Brent training rate usually works well here.

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 listed in Table 7.3.

Symbol	Description
$[CPU]$	Elapsed time history
$[\|\boldsymbol{\zeta}\|]$	Parameters norm history
$[e]$	Error history
$[v]$	Validation error history
$[\|\nabla e\|]$	Gradient norm history
$[\|\mathbf{d}\|]$	Training direction norm history
$[\eta]$	Training rate history

Table 7.3: Training history variables.

Form all the training history variables, may be the most important one is the error history. Also, it is important to analyze the final values of some variables. Table 7.4 summarizes the most important training result numbers.

Symbol	Description
N	Number of iterations
CPU	Training time
$\boldsymbol{\zeta}^*$	Final parameters
$\|\boldsymbol{\zeta}^*\|$	Final parameters norm
$e(\zeta^*)$	Final error
$v(\zeta^*)$	Final validation error
$\nabla e(\zeta^*)$	Final gradient
$\|\nabla e(\zeta^*)\|$	Final gradient norm

Table 7.4: Training result variables.

Linear regression analysis

The performance of a neural network can be measured to some extent by the mean squared error 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,

$$y = a + bx, \tag{7.12}$$

$$a = \frac{\sum_{i=1}^{n} y_i \sum_{i=1}^{n} x_i^2 - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} x_i y_i}{n \sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}, \tag{7.13}$$

$$b = \frac{n \sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{n \sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}. \tag{7.14}$$

The third parameter, R^2, is the correlation coefficient between the outputs and the targets,

$$R^2 = \sum_{i=1}^{n} \left(y_i - (a + bx_i)\right)^2. \tag{7.15}$$

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.

Underfitting and overfitting

We can illustrate the function regression task by generating a synthetic training data set in a way which is intended to capture some of the basic properties of real data used in regression problems [8]. Specifically, we generate a training data set from the function

$$h(x) = 0.5 + 0.4sin(2\pi x), \tag{7.16}$$

by sampling it at equal intervals of x and then adding random noise with a Gaussian distribution having standard deviation $\sigma = 0.05$. Figure 7.1 shows this training data set, together with the function $h(x)$.

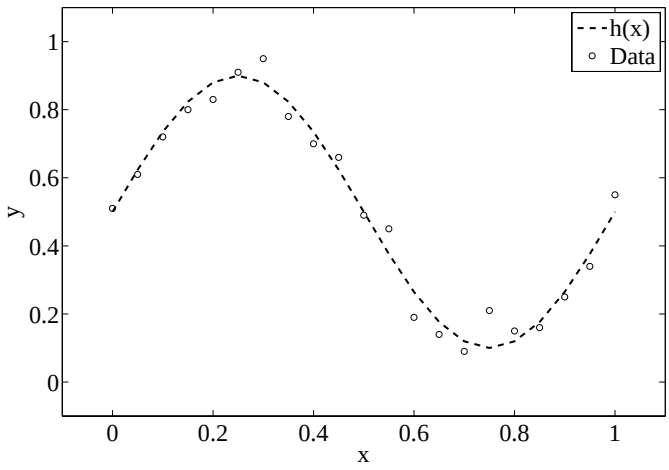


Figure 7.1: Illustration of an input-target data set for function regression.

The goal in this function regression problem for the multilayer perceptron is to obtain a function $y^*(x)$ which approximates the regression function $h(x)$.

Two frequent problems which can appear when solving a function regression problem with the sum squared error 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 [15]. 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.

Figure 7.2 shows an underfitting case for the function regression problem formulated in this section. In this case we have used a neural network which is too simple to produce an adequate fit.

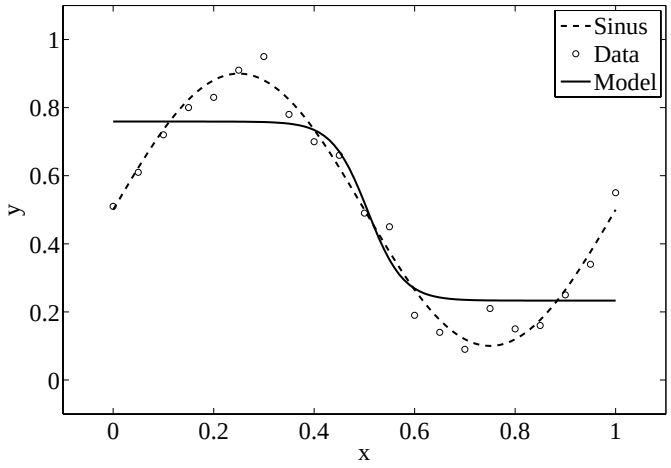


Figure 7.2: An underfitting case, showing a poor approximation obtained by a too simple model.

Figure 7.3 shows an overfitting case for the function regression problem formulated in this section. Here the error on the training data set is very small, but when new data is presented to the neural network the error is large. The neural network has memorized the training examples, but it has not learned to generalize to new situations. The model is too complex to produce an adequate fit.

Figure 7.4 shows a case when the neural network provides a good fitting of the data. The size of the model here is correct and the neural network has approximated the regression function well.

While underfitting can be prevented by simply increasing the complexity of the neural network, it is more difficult in advance to prevent overfitting. In the next two sections we introduce regularization theory and early stopping, which are methods to prevent overfitting.

Regularization theory

A problem is called well-possed if its solution meets existence, uniqueness and stability. A solution is said to be stable when small changes in the independent variable $\mathbf{x}$ led to small changes in the dependent variable $\mathbf{y}(\mathbf{x})$. Otherwise the problem is said to be ill-possed. In this way, the function regression problem for a neural network with the sum squared error is ill-posed [13]. Indeed, the solution exists for any network architecture, but for neural networks of big complexity it might be non-unique and unstable.

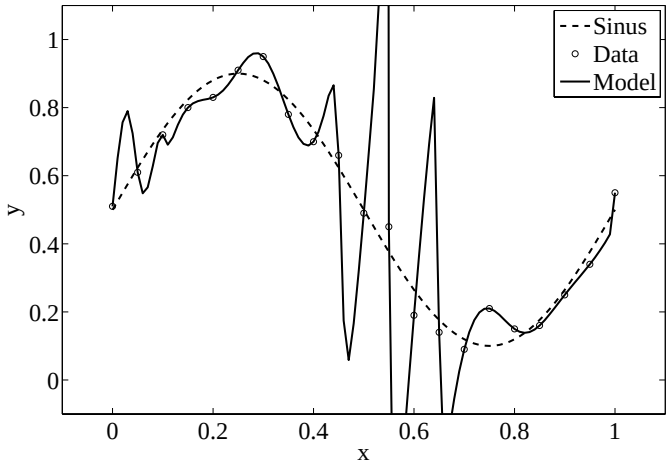


Figure 7.3: An overfitting case, showing a poor approximation obtained by a too large model.

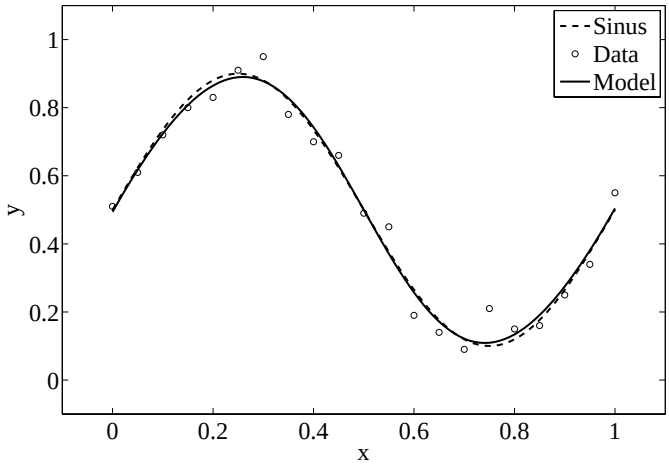


Figure 7.4: A good fitting case, showing a correct approximation obtained by a well-sized model.

In a function regression problem with the sum squared error or the Minkowski error objective functionals, the best generalization is achieved by a model whose complexity is neither too small nor too large [52]. Thus, a 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 [15].

An alternative approach to obtain good generalization in a neural network is to control its effective complexity [51]. This can be achieved by choosing an objective functional which adds a regularization term Ω to the error functional E [15]. The objective functional then becomes

$$F[\mathbf{y}(\mathbf{x}); \zeta] = E[\mathbf{y}(\mathbf{x}); \zeta] + \nu\Omega[\mathbf{y}(\mathbf{x}; \boldsymbol{\zeta})], \tag{7.17}$$

where the parameter ν is called the regularization term weight. The value of $\Omega[\mathbf{y}(\mathbf{x}; \boldsymbol{\zeta})]$ typically depends on the mapping function $\mathbf{y}(\mathbf{x})$, and if the functional form $\Omega[\mathbf{y}(\mathbf{x}; \boldsymbol{\zeta})]$ is chosen appropriately, it can be used to control overfitting [8].

One of the simplest forms of regularization term is called parameter decay and consists on the sum of the squares of the free parameters in the neural network divided by the number of free parameters [8].

$$\Omega[\mathbf{y}(\mathbf{x}; \boldsymbol{\zeta})] = \frac{1}{s} \sum_{i=1}^{s} \zeta_i^2, \tag{7.18}$$

where s is the number of free parameters. Adding this term to the objective function will cause the neural network to have smaller weights and biases, and this will force its response to be smoother and less likely to overfit.

The problem with regularization is that it is difficult to determine the optimum value for the regularization ratio ν. If we make this parameter too small, we may get overfitting. If the regularization ratio is too large, the neural network will not adequately fit the training data [15].

It is desirable to determine the optimal regularization parameters in an automated fashion. One approach is the Bayesian framework of David MacKay [37]

Early stopping

Another method for improving generalization is called early stopping. In this technique the input-target data set is divided into a training and a validation subsets. The training subset is used for training the neural network by means of the training algorithm. On the other hand, the error on the validation subset is monitored during the training process. The validation 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 validation subset typically begins to rise. When the validation error increases for a specified number of iterations, the training is stopped, and the parameters at the minimum of the validation error are set to the neural network.

Principal component analysis

In some situations, the number of input variables is large, but that variables are highly correlated (redundant). It is useful in this situation to reduce the dimension of the input space. An effective procedure for performing this operation is principal component analysis [15].

Principal component analysis is not reliably reversible. Therefore it is only recommended for input processing, since outputs require reversible processing functions [15].

Flood does not implement principal component analysis, although it will include that preprocessing method in future versions.

7.2 A simple example

In this section a simple function regression problem with just one input and one output variables is solved by means of a multilayer perceptron [15]. The data and the source code for this problem can be found within Flood.

Problem statement

In this example we have an input target data set with 101 instances, and 1 input (x) and 1 target (y) variables. The objective is to design a multilayer perceptron that can predict y values for given x values. Figure 7.5 shows this data set.

The input-target data set is divided into a training and a testing subsets. No validation analysis will be performed here. The training data represents 75% of the original data, while the testing data represents the remaining 25%. This assigns 76 and 25 instances for training and testing, respectively.

Selection of function space

The first step in solving the problem formulated in this section is to choose a network architecture to represent the regression function. Here a multilayer perceptron with a sigmoid hidden layer and a linear output layer is used. The

multilayer perceptron must have one input, since there is one input variable; and one output neuron, since there is one target variable.

The size of the hidden layer is set to 2. This neural network can be denoted as $1 : 2 : 1$. It defines a family V of parameterized functions $y(x)$ of dimensions $s = 4$, which is the number of neural parameters in the multilayer perceptron. Figure 7.6 is a graphical representation of that network architecture.

The neural parameters are initialized at random with a normal distribution of mean 0 and standard deviation 1.

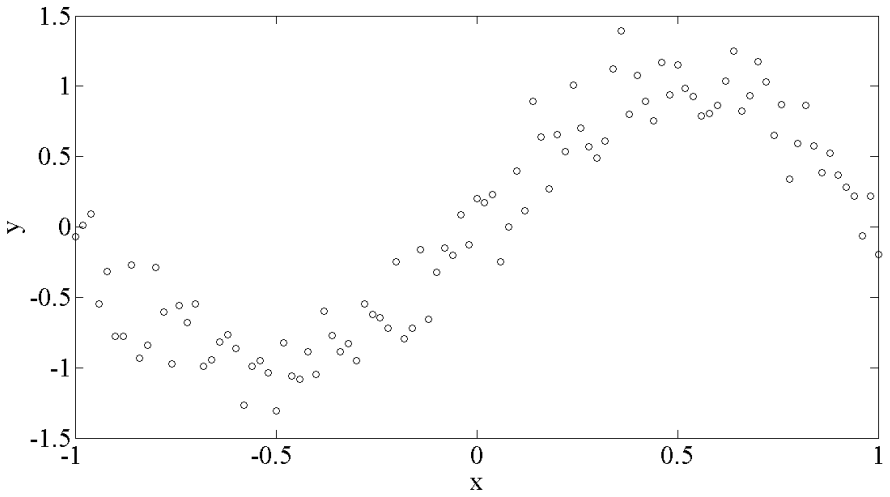


Figure 7.5: Input-target data set for the simple function regression example.

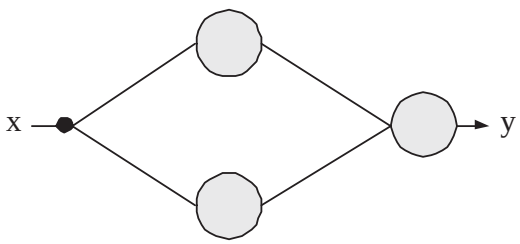


Figure 7.6: Network architecture for the simple function regression example.

Formulation of variational problem

The second step is to assign the multilayer perceptron an objective functional. This is to be the normalized squared error defined by Equation (7.10).

The variational statement of the function regression problems being considered here is then to find a function $y^*(x) \in V$ for which the functional

$$E[y(x)] = \frac{\sum_{q=1}^{Q} \left(y(x^{(q)}) - t^{(q)}\right)^2}{\sum_{q=1}^{Q} \left(t^{(q)} - \bar{t}\right)^2}. \tag{7.19}$$

defined on V, takes on a minimum value. Note that Q is here the number of training instances.

Evaluation of the objective functional in Equation (7.19) just require explicit expressions for the function represented by the different multilayer perceptrons. This is given in Section 4.1.

On the other hand, evaluation of the objective function gradient vector $\nabla e(\zeta)$, is obtained by the back-propagation algorithm derived in Section 5. This technique gives the greatest accuracy and numerical efficiency.

Solution of reduced function optimization problem

The third step in solving this problem is to assign the objective function $e(\zeta)$ a training algorithm. We use the quasi-Newton method described in Section 6 for training.

In this example, we set the training algorithm to stop after 1000 epochs of the training algorithm.

The presence of noise in the training data set makes the objective function to have local minima. This means that, when solving function regression problems, we should always repeat the learning process from several different starting positions.

During the training process the objective function decreases until the stopping criterium is satisfied. Table 7.5 shows the training results for the problem considered here.

Here $\|\zeta^*\|$ is the final parameter vector norm, $e(\zeta^*)$ the final normalized squared error and $\|\nabla e(\zeta^*)\|$ the final gradient norm, N the number of epochs and CPU the training time in a PC.

Testing analysis

The last step is to test the generalization performance of the trained neural network. Here we compare the values provided by this technique to the actually observed values.

$\|\zeta^*\|$	=	75.693
$e(\zeta^*)$	=	0.066
$\|\nabla e(\zeta^*)\|$	=	$8.049 \cdot 10^{-4}$
N	=	1000
CPU	=	$15s$

Table 7.5: Training results for the simple function regression example.

Figure 7.7 shows the linear regression analysis for this example. The intercept and slope obtained by this testing method are $a = -0.013$ and $b = 0.827$, respectively. In this case, the intercept value is quite close to 0 and the slope value is also close to 1. The plot also shows good correlation between the output and the target values. Note that the data for this example is quite noisy.

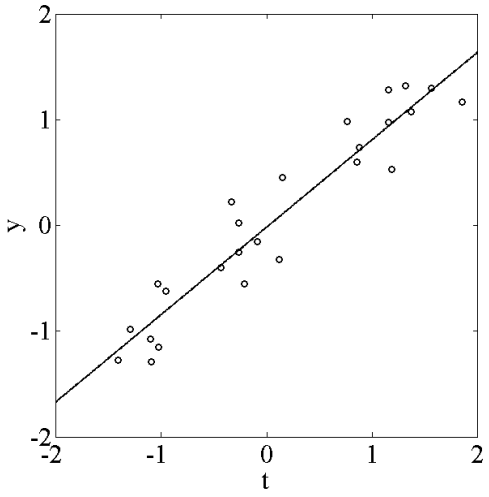


Figure 7.7: Linear regression analysis for the simple function regression example.

The multilayer perceptron is now ready to predict outputs for inputs that it has never seen.

The mathematical expression of the trained and tested multilayer perceptron is listed below.

```
<InputsScaling>
scaled_x=(x+2.63815e-017)/0.586003;
</InputsScaling>
```

```
<ForwardPropagation>
y11=tanh(-0.0383786+0.55872*scaled_x);
y12=tanh(0.0472245-0.438857*scaled_x);
scaled_y=(-0.835477+49.2648*y11+57.4557*y12);
</ForwardPropagation>
<OutputsUnscaling>
y=-0.000128472+0.750166*scaled_y;
</OutputsUnscaling>
```

7.3 A practical application: Residuary resistance of sailing yachts

In this Section an empirical model for the residuary resistance of sailing yachts as a function of hull geometry coefficients and the Froude number is constructed by means of a neural network [42]. Both the data and the source code for this problem can be found within Flood.

Introduction

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.

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 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 [21]. The Delft series are widely used in the sailing yacht industry.

In this example we follow a neural networks approach to residuary resistance of sailing yachts prediction. Here a multilayer perceptron is trained with the Delf data set to provide an estimation of the residuary resistance per unit weight of displacement as a function of hull geometry coefficients and the Froude number.

Experimental data

The Delft data set comprises 308 full-scale experiments, which were performed at the Delft Ship Hydromechanics Laboratory [21]. These experiments include 22 different hull forms, derived from a parent form closely related to the 'Standfast 43' designed by Frans Maas. Variations concern

longitudinal position of the center of buoyancy (LCB), prismatic coefficient (C_p), length-displacement ratio ($L_{WL}/\nabla_c^{1/3}$) beam-draught ratio (B_{WL}/T_C), and length-beam ratio (L_{WL}/B_{WL}). For every hull form 14 different values for the Froude number (F_N) ranging from 0.125 to 0.450 are considered. As it has been said, the measured variable is the residuary resistance per unit weight of displacement ($1000 \cdot R_R/\triangle_c$).

The data set is scaled with the mean and standard deviation method.

Selection of function space

A feed-forward neural network with a sigmoid hidden layer and a linear output layer of perceptrons is used to span the function space for this problem. It must have 6 inputs (LCB, C_p, $L_{WL}/\nabla_c^{1/3}$, B_{WL}/T_C, L_{WL}/B_{WL} and F_N), and 1 output neuron ($1000 \cdot R_r/\triangle_c$).

While the numbers of inputs and output neurons are constrained by the problem, the number of neurons in the hidden layer is a design variable. In this way, and in order to draw the best network architecture, different sizes for the hidden layer are tested, and that providing the best generalization properties is adopted. In particular, the performance of three neural networks with 6, 9 and 12 hidden neurons is compared.

For that purpose, the data is divided into training, validation and testing subsets, containing 50%, 25% and 25% of the samples, respectively. More specifically, 154 samples are used here for training, 77 for validation and 77 for testing.

Table 7.6 shows the training and validation errors for the three multilayer perceptrons considered here. E_T and E_V represent the normalized squared errors made by the trained neural networks on the training and validation data sets, respectively.

Number of hidden neurons	6	9	12
E_T	0.000394	0.000223	0.000113
E_V	0.002592	0.001349	0.001571

Table 7.6: Training and validation errors in the yacht resistance problem.

As we can see, the training error decreases with the complexity of the neural network, but the validation error shows a minimum value for the multilayer perceptron with 9 hidden neurons. A possible explanation is that the lowest model complexity produces under-fitting, and the highest model complexity produces over-fitting.

In this way, the optimal number of neurons in the hidden layer turns out to be 9. This neural network can be denoted as a $6 : 9 : 1$ multilayer perceptron, and it is depicted in Figure 7.8.

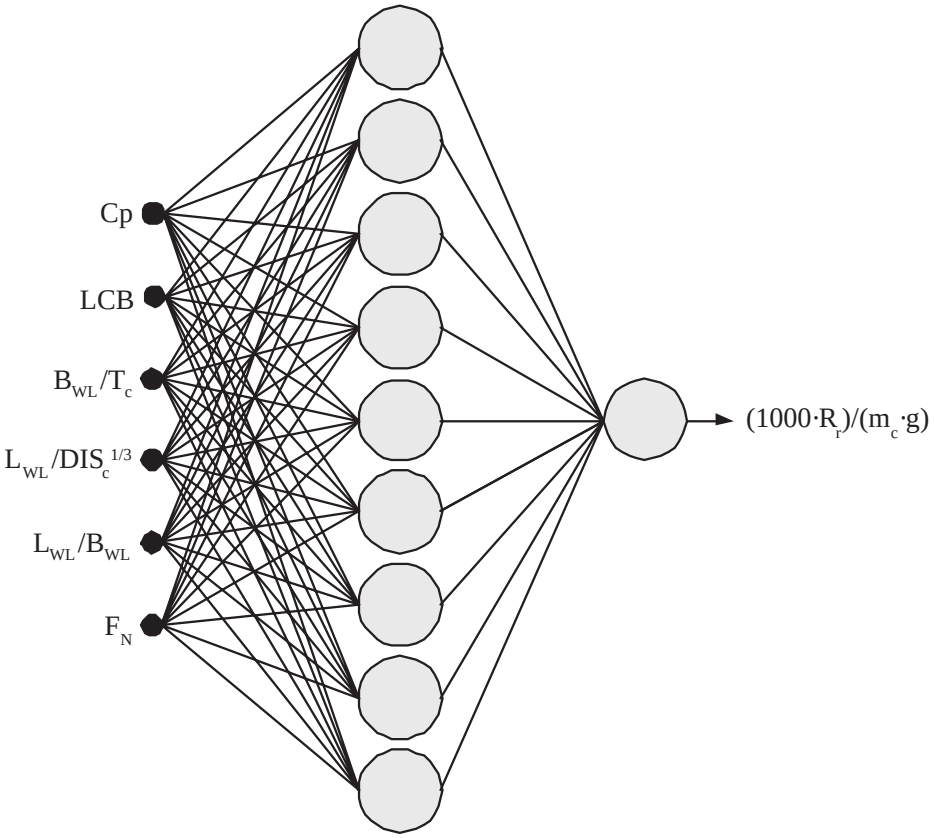


Figure 7.8: Network architecture for the yacht resistance problem.

The family of functions spanned by the neural network in Figure 7.8 can be denoted V and it is of dimension $s = 73$, the number of parameters.

Let denote

$$\mathbf{x} = (LCB, C_p, L_{WL}/\nabla_c^{1/3}, B_{WL}/T_C, L_{WL}/B_{WL}, F_N) \tag{7.20}$$

and

$$y = 1000 \cdot R_r/\triangle_c. \tag{7.21}$$

The elements of V are thus written as functions $y : \mathbb{R}^6 \to \mathbb{R}$ defined by

$$y(\mathbf{x}) = \sum_{j=0}^{9} \zeta_j^{(2)} \tanh\left(\sum_{i=0}^{6} \zeta_{ji}^{(1)} x_i\right). \tag{7.22}$$

Finally, all the biases and synaptic weights in the neural network are initialized at random.

Formulation of variational problem

The objective functional chosen for this problem is the normalized squared error between the outputs from the neural network and the target values in the Delft data set.

The variational statement of the function regression problem considered here is then to find a function $y^*(\mathbf{x}) \in V$ for which the functional

$$E[y(\mathbf{x})] = \frac{\sum_{q=1}^{Q} \|y(\mathbf{x}^{(q)}) - t^{(q)}\|^2}{\sum_{q=1}^{Q} \|t^{(q)} - \bar{t}\|^2}, \tag{7.23}$$

defined on V, takes on a minimum value.

Evaluation of the objective function gradient vector is performed with the back-propagation algorithm for the normalized squared error.

Solution of reduced function optimization problem

The selected training algorithm for solving the reduced function optimization problem is a quasi-Newton method with BFGS train direction and Brent optimal train rate. Training is set to stop when the improvement between two successive epochs is less than 10^{-12}.

The evaluation and gradient norm histories are shown in Figures 7.9 and 7.10, respectively. Note that these plots have a logarithmic scale for the Y-axis.

Table 7.7 shows the training results for this application. Here N is the number of epochs, CPU the training time in a laptop AMD 3000, $\|\zeta^*\|$ the final parameter vector norm, $e(\zeta^*)$ the final normalized squared error and $\|\nabla e(\zeta^*)\|$ the final gradient norm.

Once the neural network is trained, the inputs must pre-processed with the means and the standard deviations of the input data. Similarly, the outputs are be post-processed with the mean and the standard deviation of the target data.

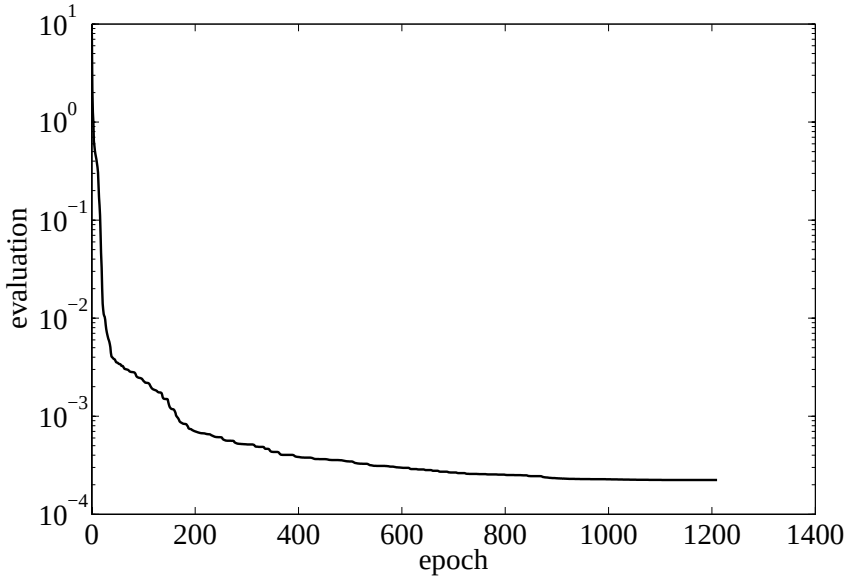


Figure 7.9: Evaluation history for the yacht resistance problem.

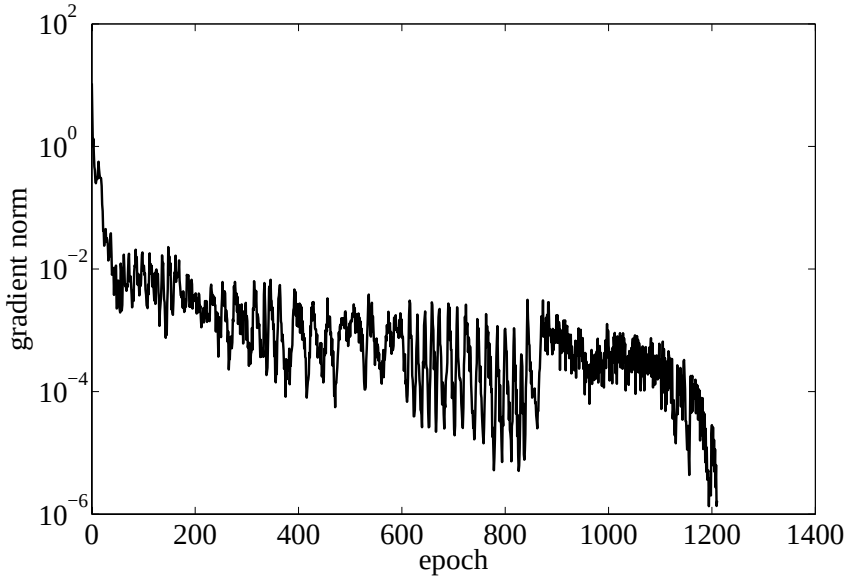


Figure 7.10: Gradient norm history for the yacht resistance problem.

The explicit expression for the residuary resistance model obtained by the neural network is

N	=	1210
CPU	=	$348s$
$\|\zeta^*\|$	=	720
$e(\zeta^*)$	=	0.000223
$\|\nabla e(\zeta^*)\|$	=	$1.648 \cdot 10^{-6}$

Table 7.7: Training results for the yacht resistance problem.

$$
\begin{aligned}
x_1 &= \frac{x_1 + 2.38182}{1.51322}, \\
x_2 &= \frac{x_2 - 0.564136}{0.02329}, \\
x_3 &= \frac{x_3 - 4.78864}{0.253057}, \\
x_4 &= \frac{x_4 - 3.93682}{0.548193}, \\
x_5 &= \frac{x_5 - 3.20682}{0.247998}, \\
x_6 &= \frac{x_6 - 0.2875}{0.100942}, \\
y &= 155.425 \\
&+ 63.2639 \tanh(-3.2222 + 0.0613793x_1 + 0.112065x_2 + 0.292097x_3 \\
&- 0.172921x_4 - 0.277616x_5 + 0.569819x_6) \\
&+ 91.0489 \tanh(-147.226 - 75.3342x_1 + 24.7384x_2 + 15.5625x_3 \\
&- 82.6019x_4 - 88.8575x_5 + 1.03963x_6) \\
&+ 0.00875896 \tanh(-77.0309 - 156.769x_1 - 244.11x_2 + 62.4042x_3 \\
&+ 70.2066x_4 + 12.1324x_5 - 76.0004x_6) \\
&+ 1.59825 \tanh(-2.94236 - 0.0526764x_1 - 0.21039x_2 - 0.266784x_3 \\
&+ 0.131973x_4 + 0.317116x_5 + 1.9489x_6) \\
&- 0.0124328 \tanh(-207.601 - 210.038x_1 + 99.7606x_2 + 106.485x_3 \\
&+ 252.003x_4 - 100.549x_5 - 51.3547x_6) \\
&+ 0.026265 \tanh(-17.9071 - 11.821x_1 + 5.72526x_2 - 52.2228x_3 \\
&+ 12.1563x_4 + 56.2703x_5 + 56.7649x_6) \\
&+ 0.00923066 \tanh(69.9392 - 133.216x_1 + 70.5299x_2 - 21.4377x_3 \\
&+ 47.7976x_4 + 15.1226x_5 + 100.747x_6) \\
&- 0.215311 \tanh(4.54348 - 1.11096x_1 + 0.862708x_2 + 1.61517x_3 \\
&- 1.11889x_4 - 0.43838x_5 - 2.36164x_6) \\
&+ 0.010475 \tanh(23.4595 - 223.96x_1 - 43.2384x_2 + 13.8833x_3 \\
&+ 75.4947x_4 - 7.87399x_5 - 200.844x_6), \\
y^*(\mathbf{x}; \boldsymbol{\zeta}^*) &= 10.4954 + 15.1605y^*(\mathbf{x}).
\end{aligned}
$$

Testing of results

A possible testing technique for the neural network model is to perform a linear regression analysis between the predicted and their corresponding experimental residuary resistance values, using and independent testing set. This analysis leads to a line $y = a + bx$ with a correlation coefficient R^2. In this way, a perfect prediction would give $a = 0$, $b = 1$ and $R^2 = 1$.

Table 7.8 shows the three parameters given by this validation analysis.

a	=	0.110
b	=	0.975
R^2	=	0.958

Table 7.8: Linear regression analysis parameters for the yacht resistance problem.

Figure 7.11 illustrates a graphical output provided by this testing analysis. The predicted residuary resistances are plotted versus the experimental ones as open circles. The solid line indicates the best linear fit. The dashed line with $R^2 = 1$ would indicate perfect fit.

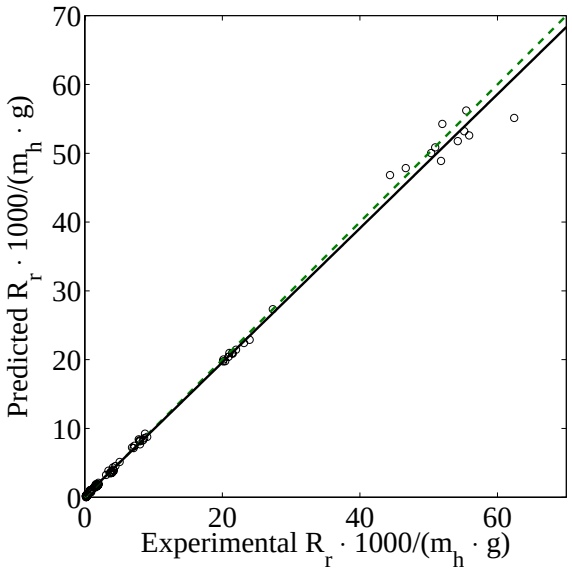


Figure 7.11: Linear regression analysis plot for the yacht resistance problem.

From Table 7.8 and Figure 7.11 we can see that the neural network is

predicting very well the entire range of residuary resistance data. Indeed, the a, b and R^2 values are very close to 0, 1 and 1, respectively.

The multilayer perceptron is now ready to estimate the residuary resistance of sailing yachts with satisfactory quality over the same range of data.

7.4 Related code

Many classes included with `Flood` are related to the problem of function regression, since these type of problems are traditional learning tasks for the multilayer perpeptron. The C++ code here include the InputTargetDataSet class, a number of error functional classes, and the FunctionRegressionUtilities class.

The InputTargetDataSet class in Flood

This class represents the concept of input-target data set. It presents the available knowledge to the neural network in function regression.
Constructors

We can construct an input-target data set object directly by loading its members from a data file. This will set up the number of instances, the number of input and target variables, and the input and target data. Optionally, also the names, units and description of input and target variables.

```
InputTargetDataSet itds("InputTargetDataSet.dat");
```

where the file InputTargetDataSet.dat must have the proper XML-type format.

An alternative way of constructing an input-target data set object without creating such a XML-type format is by first passing the numbers of instances and input and target variables and then loading the data matrix from a file.

```
InputTargetDataSet
itds(instances_number, input_variables_number, target_variables_number);

itds.load_data("Data.dat");
```

where the file Data.dat is a raw data file containing the data matrix.
Members

The InputTargetDataSet class represent the concept of input-target data set.
It contains:
- The data matrix.
- The indices of training instances.
- The indices of validation instances.
- The indices of testing instances.
- The indices of input variables.

- The indices of target variables.
- The names of the variables.
- The units of the variables.
- The description of the variables.

All that members can be accessed or modified by means of get and set methods.

Methods

The InputTargetDataSet class also contains methods to perform simple statistics on the data, which will be useful when solving data modeling problems.

All mean, standard deviation, minimum and maximum of input and target variables can be calculated by just calling the calculate_statistics method.

```
Vector< Vector<double> > variables_statistics
= itds.calculate_variables_statistics();
```

The method preprocess_variables_mean_standard_deviation scales all input and target variables so that the mean of all input and target variables is 0 and their standard deviation is 1. It also returns the basic statistics of the input and target variables.

```
Vector< Vector<double> > variables_statistics
= itds.preprocess_variables_mean_standard_deviation();
```

An alternative preprocessing method for an input-target data set is the preprocess_variables_minimum_aximum method, which scales all input and target data so that the minimum value of all input and target variables is -1 and their maximum value is 1.

```
Vector< Vector<double> > variables_statistics
= itds.preprocess_variables_minimum_maximum();
```

Please note that preprocessing modifies the data matrix. This needs to be taken into account when subsequent operations are going to be performed with that data.

When solving function regression problems it is always convenient to split the input-target data set into a training, a validation and a testing subsets. The method split_random(**double**, **double**, **double**) splits the data matrix into a training, a validation and a testing subsets. The data is separated at random.

```
itds.split(0.8, 0.2, 0.2);
```

File format

An input-target data set object can be serialized or deserialized to or from a data file which contains the member values. The file format of an object of the InputTargetDataSet class is of XML type, and it is listed below.

```
<Flood version="3.0" class='InputTargetDataSet'>
```

```
<InstancesNumber>
instances_number
</InstancesNumber>
<VariablesNumber>
variables_number
</VariablesNumber>
<TrainingInstancesNumber>
training_instances_number
</TrainingInstancesNumber>
<TrainingInstancesIndices>
training_instance_index_1 ... training_instance_q1
</TrainingInstancesIndices>
<ValidationInstancesNumber>
validation_instances_number
</ValidationInstancesNumber>
<ValidationInstancesIndices>
validation_instance_index_1 ... validation_instance_q2
</ValidationInstancesIndices>
<TestingInstancesNumber>
testing_instances_number
</TestingInstancesNumber>
<TestingInstancesIndices>
testing_instance_index_1 ... testing_instance_q2
</TestingInstancesIndices>
<InputVariablesNumber>
input_variables_number
</InputVariablesNumber>
<InputVariablesIndices>
input_variable_index_1 ... input_variable_index_n
</InputVariablesIndices>
<TargetVariablesNumber>
target_variables_number
</TargetVariablesNumber>
<TargetVariablesIndices>
target_variable_index_1 ... target_variable_index_m
</TargetVariablesIndices>
<VariablesName>
<Name>
first_variable_name
</Name>
...
<Name>
last_variable_name
</Name>
</VariablesName>
<VariablesUnits>
<Units>
first_variable_units
</Units>
...
<Units>
last_variable_units
</Units>
</VariablesUnits>
<VariablesDescription>
<Description>
first_variable_description
</Description>
...
<Description>
last_variable_description
</Description>
```

```
</VariablesDescription>
<Display>
1
</Display>
<Data>
input_instance_1 target_instance_1
...
input_instance_q target_instance_q
</Data>
```

The error functional classes in Flood

Regarding objective functionals for function regression, `Flood` includes the classes SumSquaredError, MeanSquaredError, RootMeanSquaredError, NormalizedSquaredError, and MinkowskiError to represent that error functionals.

That classes contain:

1. A relationship to a multilayer perceptron object.

2. A relationship to an input-target data set object.

They implement the calculate_objective and calculate_objective_gradient methods.

Sum squared error

The next listing shows illustrates the sum squared error class use.

```
InputTargetDataSet itds(1,1,1);
itds.initialize_data(0.0);

MultilayerPerceptron mlp(1,1,1);

SumSquaredError sse(&mlp, &itds);

double objective = sse.calculate_objective();

Vector<double> objective_gradient = sse.calculate_objective_gradient();
```

Mean squared error

The use of this class is very similar to that of the sum squared error. The file format is also the default objective functional one.

Root mean squared error

The use of this class is very similar to that of the sum squared error. The file format is also the default objective functional one.

Normalized squared error

The use of this class is very similar to that of the sum squared error. The file format is also the default objective functional one.

Minkowski error

The use of this class is very similar to that of the sum squared error. The file format of the Minkowski error class is of XML-type and it is listed below.

```
<Flood version="3.0" class='MinkowskiError'>
<MinkowskiParameter>
minkowski_parameter
</MinkowskiParameter>
<RegularizationMethod>
regularization_method
</RegularizationMethod>
<ObjectiveWeight>
objective_weight
</ObjectiveWeight>
<RegularizationWeight>
regularization_weight
</RegularizationWeight>
<CalculateEvaluationCount>
calculate_evaluation_count
</CalculateEvaluationCount>
<CalculateGradientCount>
calculate_gradient_count
</CalculateGradientCount>
<CalculateHessianCount>
calculate_Hessian_count
</CalculateHessianCount>
<Display>
display
</Display>
```

The FunctionRegressionUtilities class in Flood

The FunctionRegressionUtilities contains a variety of algorithms for this kind of applications. Basically, it can be used to generate artificial data sets for proving concepts or to perform linear regression analysis for the purpose of testing a trained neural network.

Constructors

Objects of this class can be constructed empty or associated to a multilayer perceptron object, an input target data set object or both objects.

```
FunctionRegressionUtilities fru(&mlp, &itds);
```

where &mlp is a reference to a MultilayerPerceptron object and
&itds is a reference to an InputTargetDataSet object.

Members

The main members of the function regression utilities class are:

- A multilayer perceptron object.
- An input-target data set object.

That members can get or set with the corresponding methods.

Methods

The most important method of this class calculates the linear regression analysis parameters between the outputs of multilayer perceptron and the testing instances of the target variables in an input-target data set,

```
Vector< Vector<double> > linear_regression_parameters = fru.calculate_linear_regression
```

Chapter 8

Pattern recognition

8.1 Problem formulation

Another traditional learning task for the multilayer perceptron is the pattern recognition (or classification) problem [24]. 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.

The basic goal in a pattern recognition problem is to model the posterior probabilities of class membership, conditioned on the input variables [8]. This function is called the pattern recognition function.

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 formulation of a pattern recognition problem requires:- An input-target data set.- A multilayer perceptron.- An error functional.- A training algorithm.- A testing method.

Input-target data set

Here the neural network learns from knowledge represented by a training 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,

$$\{\mathbf{x}^{(1)}, \mathbf{t}^{(1)}\}, \{\mathbf{x}^{(2)}, \mathbf{t}^{(2)}\}, ..., \{\mathbf{x}^{(Q)}, \mathbf{t}^{(Q)}\}.$$

Here the target data usually has a 1-of-c coding scheme, so that $t_i^j = \delta_{ij}$.

Multilayer perceptron

A multilayer perceptron is used to represent the pattern recognition. The number of inputs in this multilayer perceptron must be equal to the number of inputs in the data set, and the number of outputs must be the number of targets. On the other hand, the number of hidden layers and the size of each layer are up to the designer. In general, one hidden layer will be enough.

Error functional

The error functional $E : V \to \mathbb{R}$ evaluates quantitatively the performance of the regression function against the input-target data set. It is of the form

$$E = E[\mathbf{y}(\mathbf{x})].$$

Common error 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. One is the cross-entropy error, which is not considered here.

The pattern recognition problem for the multilayer perceptron translates as follows:

Problem 6 (Pattern recognition problem) *Let V be the space consisting of all functions $\mathbf{y}(\mathbf{x})$ that a given multilayer perceptron can define, with dimension d. Find a pattern recognition function $\mathbf{y}^*(\mathbf{x}) \in Y$ for which the error functional $E : V \to \mathbb{R}$ defined as*

$$E = E[\mathbf{y}(\mathbf{x})],$$

takes on a minimum or a maximum value.

Training algorithm

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

Underfitting and overfitting

The problems of underfitting and overfitting also might occur when solving a pattern recognition problem with a multilayer perceptron. 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.

Regularization theory

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, described in Section 5.6.

Early stopping

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 validation analysis is preferible.

The cofusion matrix

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

$$\mathbf{C} = \begin{pmatrix} TP & FP \\ FN & TN \end{pmatrix}$$

where TP are the true positives, FP are the false positives, FN are the false negatives and TN are the true negatives.

Example 50 *Consider a problem of pattern recognition between two classes, $\mathcal{C}_1$ and $\mathcal{C}_2$. The number of testing instances belonging to each of the two classes is 50. The number of instances correctly classified for $\mathcal{C}_1$ is 45, and for $\mathcal{C}_2$ is 40. On the other hand, the number $\mathcal{C}_1$ true instances misclassified as belongning to $\mathcal{C}_2$ is 5, and the number of $\mathcal{C}_2$ instances misclassified as belongning to $\mathcal{C}_1$ is 10. The confusion matrix here is*

$$\mathbf{C} = \begin{pmatrix} 45 & 5 \\ 10 & 40 \end{pmatrix}$$

Binary classification

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,

$$classification\,accuracy = \frac{TP + TN}{TP + TN + FP + FN}. \tag{8.1}$$

The error rate is the ratio of instances misclassified,

$$error \qquad rate = \frac{FP + FN}{TP + TN + FP + FN} \tag{8.2}$$

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

$$sensitivity = \frac{TP}{TP + FP} \tag{8.3}$$

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

$$sensitivity = \frac{TN}{TN + FP} \tag{8.4}$$

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

$$positive\,likelihood = \frac{sensitivity}{1 - specifity} \tag{8.5}$$

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

$$negative\,likelihood = \frac{specifity}{1 - sensitivity} \tag{8.6}$$

Table 8.1 summarizes the binary classification performance variables

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

Table 8.1: Binary classification performance variables.

8.2 A simple example

In this section a simple pattern recognition problem with two inputs and one output is solved by means of a multilayer perceptron [15]. The data and the source code for this problem can be found within Flood.

Problem statement

In this example we have an input target data set with 100 instances, 2 inputs (x_1, x_2) and 1 target (y). The target variable represents two classes (0 and 1). The objective is to design a multilayer perceptron that can predict the correct class for given (x_1, x_2) values. Figure 8.1 shows this data set.

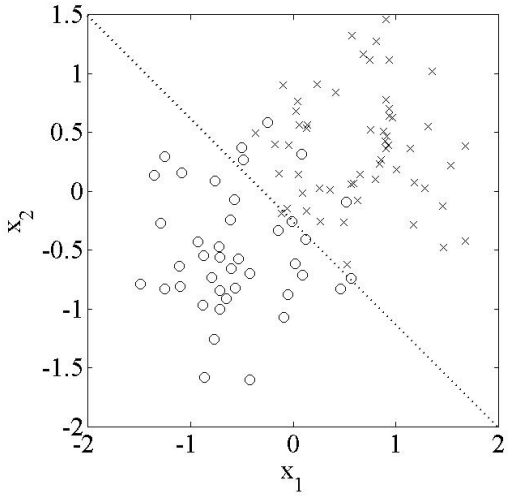


Figure 8.1: Input-target data set for the simple pattern recognition example.

The input-target data set is divided into a training and a testing subsets. No validation analysis will be performed here. The training data represents

75% of the original data, while the testing data represents the remaining 25%. This assigns 76 and 25 instances for training and testing, respectively.

Selection of function space

The first step in solving the problem formulated in this section is to choose a network architecture to represent the pattern recognition function. Here a multilayer perceptron with a sigmoid hidden layer and a linear output layer is used. The multilayer perceptron must have two inputs, since there are two input variables; and one output neuron, since there is one target variable.

The size of the hidden layer is set to 2. This neural network can be denoted as $2:2:1$. It defines a family V of parameterized functions $y(x)$ of dimensions $s = 9$, which is the number of neural parameters in the multilayer perceptron.

The neural parameters are initialized at random with a normal distribution of mean 0 and standard deviation 1.

Formulation of variational problem

The second step is to assign the multilayer perceptron an objective functional. This is to be the normalized squared error defined by Equation (7.10).

The variational statement of the pattern recognition problem being considered here is then to find a function $y^*(x) \in V$ for which the functional

$$E[y(x)] = \frac{\sum_{q=1}^{Q} \left(y(x^{(q)}) - t^{(q)}\right)^2}{\sum_{q=1}^{Q} \left(t^{(q)} - \bar{t}\right)^2}. \tag{8.7}$$

defined on V, takes on a minimum value. Note that Q is here the number of training instances.

Evaluation of the objective functional in Equation (8.7) just require explicit expressions for the function represented by the different multilayer perceptrons. This is given in Section 4.1.

On the other hand, evaluation of the objective function gradient vector $\nabla e(\zeta)$, is obtained by the back-propagation algorithm derived in Section 5. This technique gives the greatest accuracy and numerical efficiency.

Solution of reduced function optimization problem

The third step in solving this problem is to assign the objective function $e(\zeta)$ a training algorithm. We use the quasi-Newton method described in Section 6 for training.

In this example, we set the training algorithm to stop after 1000 epochs of the training algorithm.

The presence of noise in the training data set makes the objective function to have local minima. This means that, when solving pattern recognition problems, we should always repeat the learning process from several different starting positions.

During the training process the objective function decreases until the stopping criterium is satisfied. Table 8.2 shows the training results for the problem considered here.

Here $\|\zeta^*\|$ is the final parameter vector norm, $e(\zeta^*)$ the final normalized squared error and $\|\nabla e(\zeta^*)\|$ the final gradient norm, N the number of epochs and CPU the training time in a PC.

$\|\zeta^*\|$	=	753.586
$e(\zeta^*)$	=	0.098
$\|\nabla e(\zeta^*)\|$	=	$4.940 \cdot 10^{-8}$
N	=	43
CPU	=	$1s$

Table 8.2: Training results for the simple pattern recognition example.

Testing analysis

The last step is to test the generalization performance of the trained neural network. Here we compare the values provided by this technique to the actually observed values.

The binary classification performance parameters for the trained neural network on the testing instances is listed below.

```
<ClassificationAccuracy>
0.88
</ClassificationAccuracy>
<ErrorRate>
0.12
</ErrorRate>
<Sensitivity>
0.9
</Sensitivity>
<Specifity>
0.866667
```

```
</Specifity>
<PositiveLikelihood>
6.75
</PositiveLikelihood>
<NegativeLikelihood>
8.66667
</NegativeLikelihood>
```

The multilayer perceptron is now ready to predict outputs for inputs that it has never seen.

The mathematical expression of the trained and tested multilayer perceptron is listed below.

```
<InputsScaling>
scaled_x1=(x1-0.101633)/0.843221;
scaled_x2=(x2--0.0126211)/0.660989;
</InputsScaling>
<ForwardPropagation>
y11=tanh(113.654+514.321*scaled_x1+383.688*scaled_x2);
y12=tanh(368.419+32.9912*scaled_x1+67.7521*scaled_x2);
y=(-29.9135-0.477273*y11+30.4362*y12);
</ForwardPropagation>
```

8.3 A practical application: Pima indians diabetes

In this section a pattern recognition application in medicine is solved by means of a multilayer perceptron.

Introduction

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.

Experimental data

The data is taken from the UCI Machine Learning Repository [41]. 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 input-target 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: Input-target 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 $kg/(\text{height in } m)^2$).
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.

In order to test the results, we divide the input target data set into a training and a testing subsets. 75% of the instances will be assigned for training and 25% for testing. Training and testing indices are chosen at random. Table 8.6 summarizes the training and testing data sets information.

Number of samples for training:	576
Number of samples for testing:	192

Table 8.6: Training and testing data sets information.

It is always convenient to perform a linear rescaling of the training data. Here we normalize the mean and the standard deviation of the input and target data so that all variables have zero mean and unity standard deviation.

After the network is trained any future inputs that are applied to the network must be pre-processed to produce the correct input signals. The output signals from the trained network must also be post-processed to produce the proper outputs.

Selection of function space

The fist step is to choose a network architecture to represent the pattern recognition function. Here a multilayer perceptron with a sigmoid hidden layer and a linear output layer is used. This class of network is very useful in pattern recognition problems, since it is a class of universal approximator [27]. The neural network must have 8 inputs, since there are eight input variables, and 1 output, since there is one target variable. As an initial guess, we use 6 neurons in the hidden layer. This neural network can be denoted as a $8:6:1$ multilayer perceptron. It defines a family V of parameterized functions $y(\mathbf{x})$ of dimension $s = 61$, which is the number of free parameters. Elements V are of the form

$y : \mathbb{R}^8 \to \mathbb{R}$

Figure 8.2 is a graphical representation of this neural network.

Formulation of variational problem

The second step is to assign the multilayer perceptron an objective functional. For pattern recognition problems, the sum of squares error can approximate the posterior probabilities of class membership, again conditioned on the input variables. The mean squared error has the same properties than the sum of squares error, and the advantage that its value does not grow with the size of the input-target data set. Therefore we choose the mean squared error as the objective functional for this problem.

The variational statement of this pattern recognition problem is then to find a function $\mathbf{y}^*(\mathbf{x})$ for which the functional

$$E[y(\mathbf{x})] = \frac{1}{576} \sum_{q=1}^{576} \left(y(\mathbf{x}^{(q)}) - t^{(q)} \right)^2 \tag{8.8}$$

takes on a minimum value.

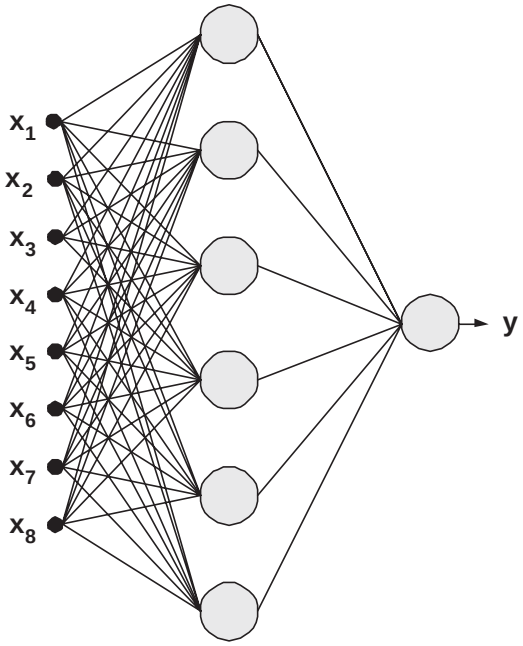


Figure 8.2: Network architecture for the pima indians diabetes problem.

Solution of reduced function optimization problem

The third step is to choose a training algorithm for solving the reduced function optimization problem. We will use the quasi-Newton method for training.

It is very easy for gradient algorithms to get stuck in local minima when learning multilayer perceptron weights. This means that we should always repeat the learning process from several different starting positions.

During the training process the objective function decreases until a stopping criterium is satisfied.

Once the network has been trained we set the mean and the standard deviation of the input and target data to be the mean and the standard deviation of the input and output variables in the multilayer perceptron.

Testing of results

The last step is to validate the generalization performance of the trained neural network. To validate a forecasting technique we need to compare the values provided by this technique to the actually observed values.

The confusion matrix is depicted next.

```
<Confusion>
36 27
25 104
</Confusion>
```

The listing below depicts the testing parameters for this problem.

```
<ClassificationAccuracy>
0.729167
</ClassificationAccuracy>
<ErrorRate>
0.270833
</ErrorRate>
<Sensitivity>
0.590164
</Sensitivity>
<Specifity>
0.793893
</Specifity>
<PositiveLikelihood>
2.86339
</PositiveLikelihood>
<NegativeLikelihood>
1.9371
</NegativeLikelihood>
```

Trying other network architectures with more neurons in the hidden layer does not improve the quality.

Production mode

Once the generalization performance of the multilayer perceptron has been tested, the neural network can be saved for future use in the so called production mode.

The mathematical expression of the function represented by the multilayer perceptron is listed next.

```
<InputsScaling>
scaled_x1=(x1-3.84505)/3.36958;
scaled_x2=(x2-120.895)/31.9726;
scaled_x3=(x3-69.1055)/19.3558;
scaled_x4=(x4-20.5365)/15.9522;
```

```
scaled_x5=(x5-79.7995)/115.244;
scaled_x6=(x6-31.9926)/7.88416;
scaled_x7=(x7-0.471876)/0.331329;
scaled_x8=(x8-33.2409)/11.7602;
</InputsScaling>

<ForwardPropagation>
y11=tanh(4.57656-92.5785*scaled_x1+30.6881*scaled_x2
+2.28317*scaled_x3+45.7042*scaled_x4+63.8847*scaled_x5
-58.3376*scaled_x6-73.4676*scaled_x7+22.0004*scaled_x8);

y12=tanh(-54.7888+61.6894*scaled_x1+11.5055*scaled_x2
+5.68579*scaled_x3+188.846*scaled_x4+62.1316*scaled_x5
+110.376*scaled_x6+16.5942*scaled_x7+73.5779*scaled_x8);

y13=tanh(-87.6615-84.421*scaled_x1+61.0545*scaled_x2
-29.0454*scaled_x3-45.4726*scaled_x4+24.2132*scaled_x5
+52.7024*scaled_x6-51.4027*scaled_x7+67.1938*scaled_x8);

y14=tanh(-39.043-4.48696*scaled_x1+131.624*scaled_x2
-112.286*scaled_x3-85.2925*scaled_x4-74.8874*scaled_x5
+34.8647*scaled_x6+19.3088*scaled_x7-32.0588*scaled_x8);

y15=tanh(63.0214-31.984*scaled_x1-29.5837*scaled_x2
+15.5865*scaled_x3+38.6653*scaled_x4-19.5926*scaled_x5
+62.5646*scaled_x6-114.814*scaled_x7+72.9811*scaled_x8);

y16=tanh(-58.0502-58.2068*scaled_x1-139.021*scaled_x2
+2.79391*scaled_x3+8.80812*scaled_x4-49.925*scaled_x5
-20.7181*scaled_x6+4.42621*scaled_x7-7.45716*scaled_x8);

scaled_y1=logistic(-0.242571-1.27699*y11+128.86*y12
+1.35776*y13+128.057*y14-0.891829*y15-1.31128*y16);
</ForwardPropagation>
```

8.4 Related code

Flood classes which are related to the solution of pattern recognition problems include the InputTargetDataSet class, several error functional classes, and the PatternRecognitionUtilities class.

The InputTargetDataSet class in Flood

The InputTargetDataSet class represent the concept of input-target data set.

That class is described in Section 7.4. The only difference here is that the values of the target variables will be, in general, 0s and 1s.

The error functional classes in Flood

Regarding objective functionals for modeling of data, `Flood` includes the classes SumSquaredError, MeanSquaredError, RootMeanSquaredError, NormalizedSquaredError and MinkowskiError to represent that error functionals. All of them are also described in Section 7.4.

The PatternRecognitionUtilities class in Flood

This class contains a variety of algorithms for this kind of applications. Basically, it can be used to generate artificial data sets for proving concepts or to perform testing analysis on a trained neural network.

Constructors

Objects of this class can be constructed empty or associated to a multilayer perceptron object, an input target data set object or both objects.

```
PatternRecognitionUtilities pru(&mlp, &itds);
```

where &mlp is a reference to a MultilayerPerceptron object and &itds is a reference to an InputTargetDataSet object.

Members

The main members of the pattern recognition utilities class are:

- A multilayer perceptron object.
- An input-target data set object.

That members can get or set with the corresponding methods.

Methods

The most important method of this class calculates the confusion matrix of a multilayer perceptron on an testing data set,

```
Vector
The \lstinline"FunctionRegressionUtilities" contains a variety of algorithms fo

\noindent\textit{Constructors}

Objects of this class can be constructed empty or associated to a multilayer pe

\begin{lstlisting}
FunctionRegressionUtilities fru(&mlp, &itds);
```

where &mlp is a reference to a MultilayerPerceptron object and
&itds is a reference to an InputTargetDataSet object.
Members

The main members of the function regression utilities class are:
- A multilayer perceptron object.
- An input-target data set object.

That members can get or set with the corresponding methods.
Methods

The most important method of this class calculates the linear regression analysis parameters between the outputs of multilayer perceptron and the testing instances of the target variables in an input-target data set,

```
Matrix<double> confusion = pru.calculate_confusion();
```

The calculate_binary_classification_test performs a binary classification test (classification accuracy, error rate, sensitivity, specifity, positive likelihood and negative likelihood).

```
Vector<double> binary_classification_test
= pru.calculate_binary_classification_test();
```

Chapter 9

Optimal control

9.1 Problem formulation

Optimal control -which is playing an increasingly important role in the design of modern systems- has as its objective 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 [29].

The formulation of an optimal control problem requires:
- A mathematical model.
- A multilayer perceptron.
- An objective functional.
- A training algorithm.

Mathematical model

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

Let denote $\mathbf{u}(\mathbf{x})$ the state variables of the system and $\mathbf{y}(\mathbf{x})$ the state variables to the system. Then the mathematical model (or state equation) can be written as

$$\mathcal{L}(\mathbf{x}, \mathbf{u}(\mathbf{x}); \mathbf{x}(\mathbf{x})) = \mathbf{f}, \tag{9.1}$$

where $\mathcal{L}$ is some algebraic or differential operator and $\mathbf{f}$ is some forcing term.

The number of state variables is represented by N and the number of control variables by m.

Mathematical models can be expressed as all algebraic, ordinary differential and partial differential equations.

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. In this particular case the mathematical model is of the form

$$\begin{aligned} \frac{du_1}{dt} &= f_1(t, u_1, \ldots, u_N, y_1, \ldots, y_m), && (9.2) \\ \frac{du_N)}{dt} &= f_1(t, u_1, \ldots, u_N, y_1, \ldots, y_m), && (9.3) \\ u_1(0) &= u_{1i} && (9.4) \\ u_N(0) &= u_{Ni} && (9.5) \end{aligned}$$

Integration here is usually performed with the Runge-Kutta-Fehlberg method.

Multilayer perceptron

A multilayer perceptron is used to represent the control variables.

For optimal control problems, 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 not regarded as being ill-possed, and a sufficient complexity for the function space selected is gen- erally enough.

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

This multilayer perceptron spans a function space V of dimension d. The elements of that space, $\mathbf{y} : X \to Y$, are of the form

$$\mathbf{y} = \mathbf{y}(\mathbf{x}). \tag{9.6}$$

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.

Objective functional

In order to evaluate the performance of a system quantitatively, a criterion must be chosen. The performance criterion is a scalar functional of the control variables of the form

$F : Y \to \mathbb{R}$

$$F = F[\mathbf{y}(\mathbf{x})].$$

In certain cases the problem statement might clearly indicate which performance criterion is to be selected, whereas in other cases that selection is a subjective matter [29].

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 [29].

Similarly, a state which satisfies the state constraints is called an admissible state [29].

Definition 1 (Admissible control) *A control* $\mathbf{y}(\mathbf{x})$ *which satisfies all the constraints is called an admissible control. The set of admissible controls is denoted* Y*, and the notation* $\mathbf{y}(\mathbf{x}) \in Y$ *means that* $\mathbf{y}(\mathbf{x})$ *is admissible.*

Definition 2 (Admissible state) *A state* $\mathbf{u}(\mathbf{x})$ *which satisfies the state variable constraints is called an admissible state. The set of admissible states will be denoted by* $\mathbf{U}$*, and* $\mathbf{u}(\mathbf{x}) \in \mathbf{U}$ *means that the state* $\mathbf{u}(\mathbf{x})$ *is admissible.*

An optimal control is defined as one that minimizes or maximizes the performance criterion, and the corresponding state is called an optimal state. More specifically, the optimal control problem can be formulated as

Problem 7 (Optimal control problem) *Let* Y *and* U *be the function spaces of admissible controls and states, respectively. Find an admissible control* $\mathbf{y}^*(\mathbf{x}) \in Y$ *which causes the system*

$$\mathcal{L}(\mathbf{y}(\mathbf{x}), \mathbf{u}(\mathbf{x}), \mathbf{x}) = \mathbf{f}$$

to be in an admissible state $\mathbf{u}^*(\mathbf{x}) \in U$ *and for which the objective functional*

$$F = F[\mathbf{y}(\mathbf{x})]$$

takes on a minimum or maximum value. The function $\mathbf{y}^*(\mathbf{x})$ *is called an optimal control and the function* $\mathbf{u}^*(\mathbf{x})$ *an optimal state.*

In this way, the problem of optimal control is formulated as a variational problem [29].

In general, the objective function $f(\boldsymbol{\zeta})$, cannot be evaluated analytically. Its gradient vector $\nabla f(\boldsymbol{\zeta})$, and its Hessian matrix $f(\boldsymbol{\zeta})$ cannot be computed analytically by means of the back-propagation algorithm, and numerical differentiation is used.

Training algorithm

In general, optimal control problems lead to a variational problem that cannot be solved analytically to obtain the optimal control signal. In order to achieve this goal, two types of numerical methods are found in the literature, namely, direct and indirect [7]. From them, direct methods are the most widely used.

As it has been explained in this report, a variational formulation for neural networks provides a direct method for the solution of variational problems. Therefore optimal control problems can be approached with this numerical technique.

9.2 A simple example

The car problem for the multilayer perceptron is an optimal control problem with two controls and two state variables. It is defined by an objective functional with two constraints and requiring the integration of a system of ordinary differential equations.

Problem statement

Consider a car which is to be driven along the x-axis from some position x_i at velocity v_i to some desired position x_f at desired velocity v_f in a minimum time t_f, see Figure 9.1.

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 Cauchy problem of two ordinary differential equations with their corresponding initial conditions,

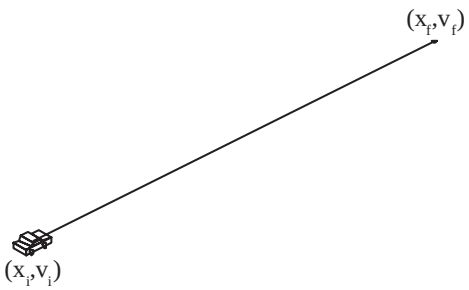


Figure 9.1: The car problem statement.

$$\begin{aligned} \dot{x}(t) &= v(t), && (9.7)\\ \dot{v}(t) &= a(t) + d(t), && (9.8)\\ x(0) &= x_i, && (9.9)\\ v(0) &= v_i, && (9.10) \end{aligned}$$

for $t \in [0, t_f]$ and where the controls $a(t)$ and $d(t)$ are the throttle acceleration and the braking deceleration, respectively.

The acceleration is bounded by the capability of the engine, and the deceleration is limited by the braking system parameters. If the maximum acceleration is $\sup(a) > 0$, and the maximum deceleration is $\sup(d) > 0$, such bounds on the control variables can be written

$$\begin{aligned} 0 \leq \; & a(t) \; \leq \sup(a), && (9.11)\\ -\sup(d) \leq \; & d(t) \; \leq 0. && (9.12) \end{aligned}$$

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

$$F[(a, d)(t)] = t_f. \tag{9.13}$$

On the other hand, the car is to be driven to a desired position x_f and a desired velocity v_f, therefore $x(t_f) = x_f$ and $v(t_f) = v_f$. Such constraints on the state variables can be expressed as error functionals,

$$\begin{aligned} E_x[(a, d)(t)] &= x(t_f) - x_f \\ &= 0, && (9.14) \end{aligned}$$

$$\begin{aligned} E_v[(a,d)(t)] &= v(t_f) - v_f \\ &= 0. \end{aligned} \tag{9.15}$$

where E_x and E_v are called the final position and velocity errors, respectively.

If we set the initial position, initial velocity, final position, final velocity, maximum acceleration and maximum deceleration to be $x_i = 0$, $v_i = 0$, $x_f = 1$, $v_f = 0$, $\sup(a) = 1$ and $\sup(d) = 1$, respectively. This problem has an analytical solution for the optimal control given by [29]

$$a^*(t) = \begin{cases} 1, & 0 \leq t < 1, \\ 0, & 1 \leq t \leq 2, \end{cases} \tag{9.16}$$

$$d^*(t) = \begin{cases} 0, & 0 \leq t < 1, \\ -1, & 1 \leq t \leq 2, \end{cases} \tag{9.17}$$

which provides a minimum final time $t_f^* = 2$.

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.

Selection of function space

Here a multilayer perceptron with a sigmoid hidden layer and a linear output layer is chosen to represent the control $(a, b)(t)$. The neural network must have one input, t, and two output neurons, a and d. Although the size of the hidden layer is a design variable, that number is not a critical issue in optimal control. Indeed, this class of problems are not regarded as being ill-possed, and a sufficient complexity for the function space selected is generally enough. In this problem we use three hidden neurons. Figure 9.2 is a graphical representation of this network architecture.

Also this neural network needs an associated independent parameter representing the final time t_f.

Such a multilayer perceptron spans a family V of parameterized functions $(a, b)(t)$ of dimension $s = 14 + 1$, being 14 the number of biases and synaptic weights and 1 the number of independent parameters. Elements V are of the form

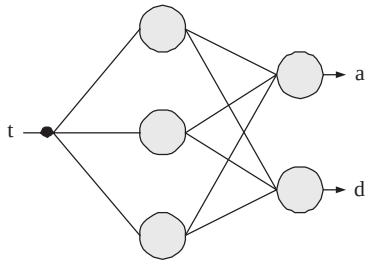


Figure 9.2: Network architecture for the car problem.

$(a, d) : \mathbb{R} \to \mathbb{R}^2$
where

$$a(t) = b_1^{(2)} + \sum_{j=1}^{3} w_{1j}^{(2)} \tanh\left(b_j^{(1)} + w_{j1}^{(1)} t\right), \tag{9.18}$$

$$d(t) = b_2^{(2)} + \sum_{j=1}^{3} w_{2j}^{(2)} \tanh\left(b_j^{(1)} + w_{j1}^{(1)} t\right). \tag{9.19}$$

Equation (9.18) represents just one function, in many of the parameters are shared.

The control variable is constrained to lie in the interval $[0, 1]$. To deal with such constraints we bound the network outputs in the form

$$a(t) = \begin{cases} 0, & a(t) < 0. \\ a(t), & 0 \le a(t) \le 1. \\ 1, & a(t) > 1. \end{cases} \tag{9.20}$$

$$d(t) = \begin{cases} -1, & d(t) < -1. \\ d(t), & -1 \le d(t) \le 0. \\ 0, & d(t) > 0. \end{cases} \tag{9.21}$$

All the parameters in the neural network are initialized at random.

Formulation of variational problem

From Equations (9.14), (9.15) and (9.13), the car problem formulated in this Section can be stated so as to find a control $(a, d)^*(t) \in V$ and t_f^* such that

$$E_x[(a, d)^*(t)] = 0, \tag{9.22}$$

$$E_v[(a, d)^*(t)] = 0, \tag{9.23}$$

and for which the functional

$$T[(a, d)(t)],$$

defined on V, takes on a minimum value.

This constrained problem can be converted to an unconstrained one by the use of penalty terms. The statement of this unconstrained problem is now to find a control $(a, d)^*(t)$ for which the objective functional

$$F[(a, d)(t)] = \rho_X E_x^2 + \rho_V E_v^2 + \rho_T T, \tag{9.24}$$

defined on V, takes on a minimum value.

The values $\rho_X = 10^{-3}$, $\rho_V = 10^{-3}$ and $\rho_T = 1$ are called the final time, error position an error velocity term weights, respectively.

Please note that evaluation of the objective functional (9.24) requires a numerical method for integration of ordinary differential equations. Here we choose the Runge-Kutta-Fehlberg method with tolerance 10^{-12} [49].

The objective function gradient vector, $\nabla f(\boldsymbol{\zeta})$, must be evaluated with numerical differentiation. In particular, we use the symmetrical central differences method [8] with an epsilon value of 10^{-6}.

Solution of reduced function optimization problem

Here we use a quasi-Newton method with BFGS train direction and Brent optimal train rate methods [45]. The tolerance in the Brent's method is set to 10^{-6}. While other direct methods might suffer from local optima with that algorithm in this problem, the neural networks method proposed here has demonstrated fully convergence to the global optimum.

In this example, training is stopped when the Brent's method gives zero rate for some gradient descent direction. The evaluation of the initial guess was 0.827; after 112 epochs of training this value falls to $1.999 \cdot 10^{-3}$.

Table 9.1 shows the training results for this problem. Here N denotes the number of epochs, M the number of evaluations, F the final objective functional value, ∇f the final objective function gradient norm, E_x the final position error, E_v the final velocity error and t_f^* the optimum time. As we can see, the final errors in the position and the velocity of the car are very small, and the final time provided by the neural network matches the analytical final time provided by the optimal function in Equation (9.16). More specifically, the errors made in the constraints are around $5 \cdot 10^{-3}$ and the error made in the final time is around 0.2%.

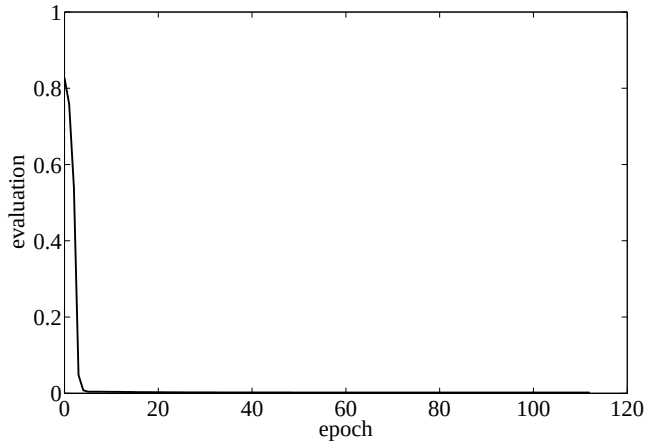


Figure 9.3: Evaluation history for the car problem.

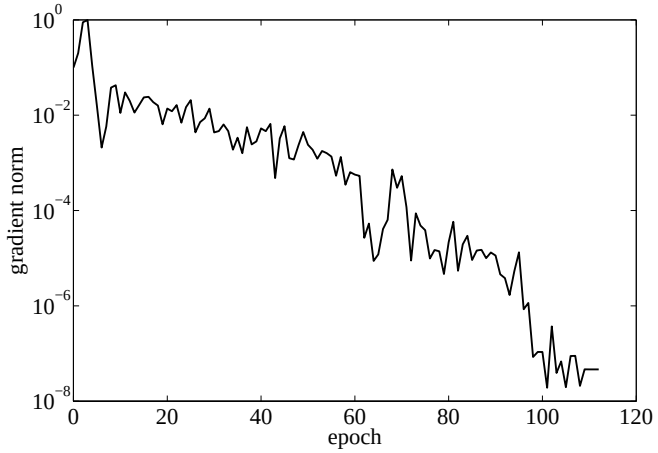


Figure 9.4: Gradient norm history for the car problem.

The analytical form of the optimal control addressed by the neural network is as follows

N	=	113
M	=	7565
CPU	=	$18s$
$\|\boldsymbol{\zeta}^*\|$	=	6.84336
$f(\boldsymbol{\zeta}^*, t_f^*)$	=	0.00199951
$e_x(\boldsymbol{\zeta}^*, t_f^*)$	=	$5.00358 \cdot 10^{-4}$
$e_v(\boldsymbol{\zeta}^*, t_f^*)$	=	$4.99823 \cdot 10^{-4}$
$\|\nabla f(\boldsymbol{\zeta}^*, t_f^*)\|/s$	=	$4.63842 \cdot 10^{-8}$
t_f^*	=	1.99901

Table 9.1: Training results for the car problem.

$$
\begin{aligned}
a^*(t) &= -1.31175 \\
&+ 6.85555 \tanh\left(-1.1448 + 1.48771t\right) \\
&- 0.387495 \tanh\left(2.52653 - 1.5223t\right) \\
&+ 16.1508 \tanh\left(12.2927 - 12.3053t\right), \qquad (9.25) \\
d^*(t) &= 1.82681 \\
&- 4.91867 \tanh\left(-1.1448 + 1.48771t\right) \\
&- 0.839186 \tanh\left(2.52653 - 1.5223t\right) \\
&+ 6.76623 \tanh\left(12.2927 - 12.3053t\right), \qquad (9.26)
\end{aligned}
$$

subject to the lower and upper bounds

$$
a^*(t) = \begin{cases} 0, & a^*(t) < 0. \\ a^*(t), & 0 \leq a^*(t) \leq 1. \\ 1, & a^*(t) > 1. \end{cases} \qquad (9.27)
$$

$$
d^*(t) = \begin{cases} -1, & d^*(t) < -1. \\ d^*(t), & -1 \leq d^*(t) \leq 0. \\ 0, & d^*(t) > 0. \end{cases} \qquad (9.28)
$$

and for $t \in [0, 1.99901]$.

The optimal control (acceleration and deceleration) and the corresponding optimal trajectories (position and velocity) obtained by the neural network are shown in Figures 9.5, 9.6, 9.7 and 9.8, respectively.

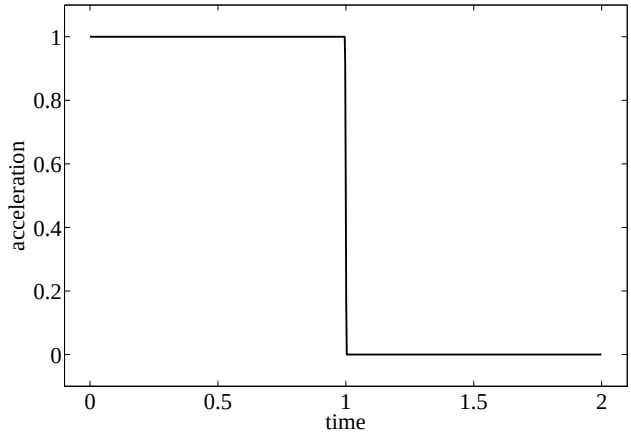


Figure 9.5: Neural network solution for the optimal acceleration in the car problem.

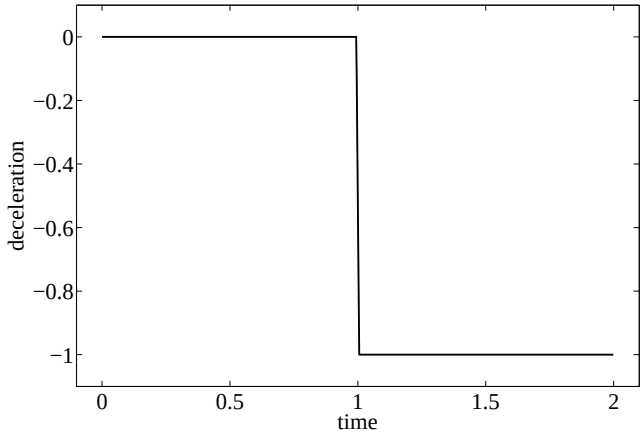


Figure 9.6: Neural network solution for the optimal deceleration in the car problem.

9.3 A practical application: Fed batch fermenter

The fed batch fermenter problem for the multilayer perceptron is an optimal control problem with one control and four state variables, and defined by an objective functional with one constraint and requiring the integration

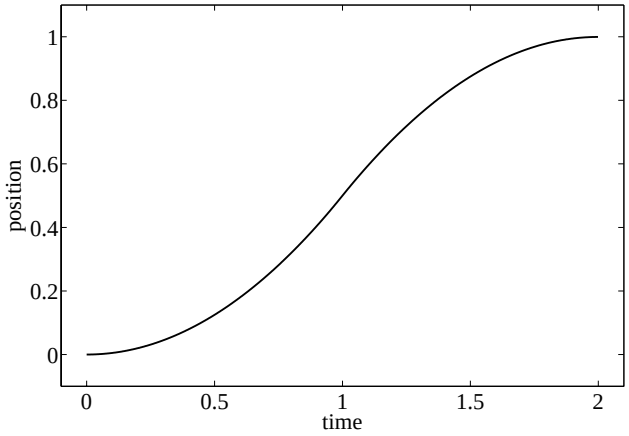


Figure 9.7: Corresponding optimal trajectory for the position in the car problem.

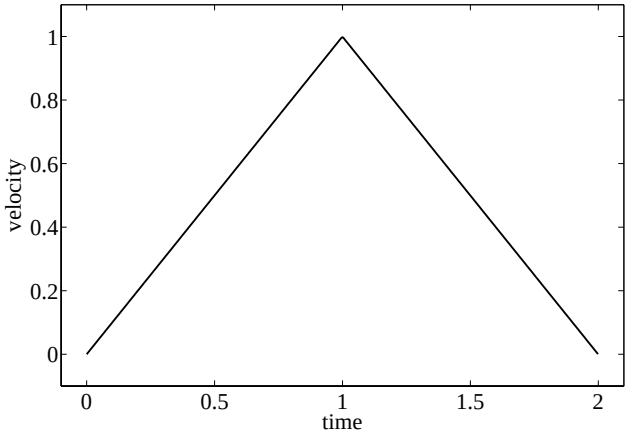


Figure 9.8: Corresponding optimal trajectory for the position in the car problem.

of a system of ordinary differential equations. The implementation of this problem is included in Flood [31].

Introduction

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.

In this Section, we seek to determine the optimal control law and the corresponding optimal trajectory of a fed batch fermenter problem using a neural network. We also compare the results by this numerical method against those provided by other authors.

Problem statement

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 [1]. Figure 9.3 shows the basics of the reactor.

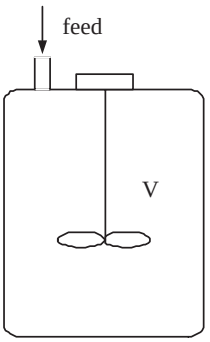


Figure 9.9: The fed batch fermenter.

State equations

The states of the plant are the concentration of cell mass $x(t)$, the concentration of substrate $s(t)$, the concentration of product $p(t)$ and the broth volume in the fermenter $v(t)$. The amount $u(t)$ is the feeding rate, which

is the only manipulated variable of this process [12]. The dynamic behavior of this fed batch fermentation process can be described by the following differential-algebraic equations

$$\frac{dx(t)}{dt} = x(t)\mu(t) - u(t)\frac{x(t)}{v(t)}, \tag{9.29}$$

$$\frac{ds(t)}{dt} = -x(t)\frac{\mu(t)}{Y} + u(t)\frac{s_0 - s(t)}{v(t)}, \tag{9.30}$$

$$\frac{dp(t)}{dt} = -x(t)\eta(t) - u(t)\frac{p(t)}{v(t)}, \tag{9.31}$$

$$\frac{dv(t)}{dt} = u(t), \tag{9.32}$$

$$\mu(t) = \frac{\mu_0}{1 + \frac{p(t)}{K_p}}\frac{s(t)}{K_s + s(t)}, \tag{9.33}$$

$$\eta(t) = \frac{\nu_0}{1 + \frac{p(t)}{K'_p}}\frac{s(t)}{K'_s + s(t)}, \tag{9.34}$$

together with their initial conditions

$$x(t_i) = x_i, \tag{9.35}$$

$$s(t_i) = s_i, \tag{9.36}$$

$$p(t_i) = p_i, \tag{9.37}$$

$$v(t_i) = v_i. \tag{9.38}$$

Here μ is the specific growth rate, η the specific productivity, Y the yield coefficient and s_0 the substrate concentration of the feed. The kinetic constants for Saccharomyces cerevisiae growing of glucose are $\mu_0 = 0.408\,h^{-1}$, $K_p = 16.0\,g\,l^{-1}$, $K_s = 0.22\,g\,l^{-1}$, $\eta_0 = 1.0\,h^{-1}$, $K'_p = 71.5\,g\,l^{-1}$ and $K'_s = 0.44\,g\,l^{-1}$ [26].

Input constraints

Here the feed rate to the reactor constrained to lie in the interval

$$u(t) \in [\inf(u), \sup(u)] \tag{9.39}$$

for $t \in [t_i, t_f]$.

State constraints

The liquid volume of the reactor is limited by the vessel size, $v(t) \leq V$. This constraint on the state of the system can be written as an error functional,

$$\begin{aligned} E_V[u(t)] &\equiv V - v(t_f) \\ &= 0. \end{aligned} \tag{9.40}$$

Performance requirements

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. More specifically, the aim is to choose a feed rate which maximizes the functional

$$Y[u(t)] = p(t_f)v(t_f), \tag{9.41}$$

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.

Nominal values

The nominal values for all the parameters here are the same as those used by R. Luus in [36], so as to compare the neural network results to those reported by that author.

The yield coefficient, Y, is assumed to be a constant of 0.1. The initial state is specified as $x_0 = 1g\,l^{-1}$, $s_0 = 150g\,l^{-1}$, $p_0 = 0$ and $v_0 = 10\,l$, and the final time of the process, t_f, is set at $54\,h$. Besides, The feed rate to the reactor constrained by $0 \leq u(t) \leq 12$.

Selection of function space

The control variable $u(t)$ is to be represented by a multilayer perceptron with a hidden layer of sigmoid neurons and an output layer of linear neurons. Necessarily, the number of inputs is one, t, and the number of output neurons is also one, u. The size of the hidden layer is a design variable in the problem, which is set here to two. Figure 9.10 is a graphical representation of the resulting network architecture.

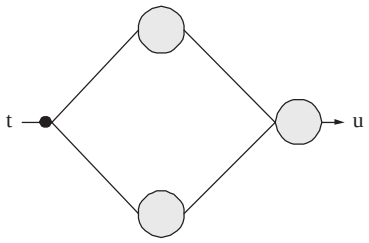


Figure 9.10: Network architecture for the fed batch fermenter problem.

This neural network spans a family V of parameterized functions $u(t)$ of dimension $s = 7$, which is the number of biases and synaptic weights. Elements of V are of the form

$u : \mathbb{R} \to \mathbb{R}$

where

$$u(t) = b_1^{(2)} + \sum_{j=1}^{2} w_{1j}^{(2)} \tanh\left(b_j^{(1)} + w_{j1}^{(1)} t\right). \tag{9.42}$$

The minimum and maximum input and output values are by far outside the range $[-1, 1]$. It is therefore necessary to pre and post-process input and output in order to achieve better approximation properties. The minimum and maximum values used here for pre and post-processing are listed in Table 9.2.

$\min(t)$	$=$	0
$\max(t)$	$=$	54
$\min(u(t))$	$=$	0
$\max(u(t))$	$=$	12

Table 9.2: Minimum and maximum for pre and post-processing in the fermenter problem.

In this way, the input is pre-processed to produce an input signal with minimum -1 and maximum 1,

$$t = 2\frac{t}{54} - 1. \tag{9.43}$$

Similarly, the output signal from the neural network is then post-processed to produce an output ranging from -1 and 1,

$$u(t) = 0.4\,(u(t)12 + 1)\,. \tag{9.44}$$

It is easy to see that this form of pre and post-processing produces input and output signals in the range $[-1, 1]$.

On the other hand, the feed rate to the fermenter $u(t)$ is constrained to lie in the interval $[0, 12]$. To deal with such constraints we bound the network outputs in the form

$$u(t) = \begin{cases} 0, & u(t) < 0. \\ u(t), & 0 \leq u(t) \leq 12. \\ 12, & u(t) > 12. \end{cases} \tag{9.45}$$

Formulation of variational problem

Following Equations (9.40) and (9.41), the fed batch fermenter for the multilayer perceptron can be stated as to find an optimal control $u^*(t) \in V$ so that

$$E_V[u^*(t)] = 0, \tag{9.46}$$

and for which the functional

$$Y[u(t)],$$

defined on V, takes on a minimum value.

This constrained problem can be formulated as an unconstrained one by adding a penalty term for the constraint to the original objective functional. More specifically, the unconstrained fed batch fermenter problem is stated as to find an optimal control $u^*(t) \in V$ for which the functional

$$F[u(t)] = \rho_E (E_V[u(t)])^2 - \rho_Y Y[u(t)], \tag{9.47}$$

defined on V, takes on a minimum value.

The volume error an yield term weights ρ_E and ρ_Y, are set here to 10^{-3} and 10^{-10}, respectively.

Please note that evaluation of the objective functional (9.47) requires a numerical method for integration of ordinary differential equations. Here we choose the Runge-Kutta-Fehlberg method [49]. For this problem we set the tolerance to 10^{-12}.

On the other hand, there are no target outputs for the neural network here, so a back-propagation algorithm for the objective function gradient vector, $\nabla f(\boldsymbol{\zeta})$, can not be derived for this problem. Instead, we use the central differences method for numerical differentiation with $\epsilon = 10^{-6}$.

Solution of reduced function optimization problem

Here we use the quasi-Newton method for training. This method is applicable here, since the objective function is differentiable.

In this example, the training algorithm stops when it can not keep minimizing the objective function. When this situation occurs, the Brent's method gives zero train rate for a gradient descent train direction.

The training algorithm required 492 epochs to go. The evaluation of the initial guess is -0.0246886. After training this value falls to -0.0417673. On the other side, the gradient norm deceases from 0.0394488 to $2.7123 \cdot 10^{-6}$. Figures 9.11 and 9.12 explain this process in a graphical fashion.

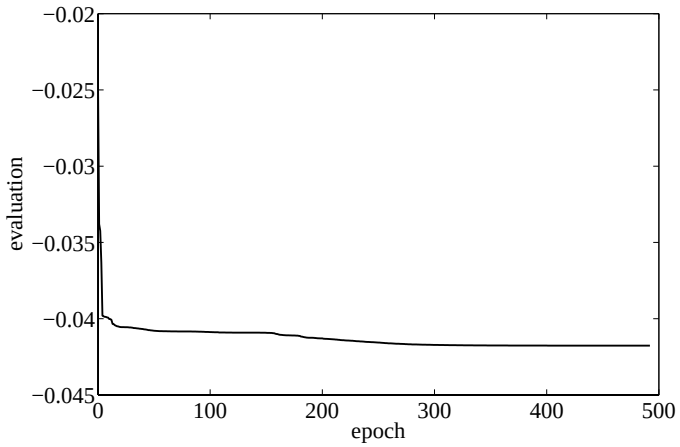


Figure 9.11: Evaluation history in the fed batch fermenter problem.

Table 9.3 shows the training results for this problem. Here N is the number of epochs, M the number of objective function evaluations, CPU the computing time in a laptop AMD 3000, $\|\boldsymbol{\zeta}^*\|$ the final parameters norm, $f(\boldsymbol{\zeta}^*)$ the final objective value, $e_v(\boldsymbol{\zeta}^*)$ the final error in the volume constraint and $y(\boldsymbol{\zeta}^*)$ the final yield. The final volume error is around 0.1% of the total volume, giving a very slight violation of the constraint. On the other side, the yield obtained by the neural network is about 0.2% higher than the $20406\,l$ reported by Luus [36].

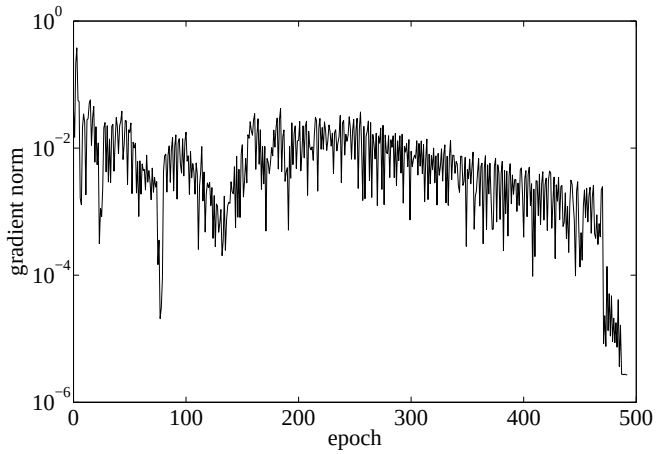


Figure 9.12: Gradient norm history in the fed batch fermenter problem.

N	=	493
M	=	24776
CPU	=	1019
$\|\boldsymbol{\zeta}^*\|$	=	23.0654
$f(\boldsymbol{\zeta}^*)$	=	$-4.17673 \cdot 10^{-2}$
$e_v(\boldsymbol{\zeta}^*)$	=	0.205
$y(\boldsymbol{\zeta}^*)$	=	20447.8
$\|\nabla f(\boldsymbol{\zeta}^*)\|$	=	$2.7123 \cdot 10^{-6}$

Table 9.3: Training results for the fed batch fermenter problem.

The optimal feed rate obtained by the neural network can be written in a explicit form as

$$\begin{aligned} t &= \frac{2}{54}t - 1, \\ u^*(t) &= -27.3844 \\ &+ 37.5292 \tanh\left(27.1799 - 26.3943t\right) \\ &+ 11.2443 \tanh\left(-1.83915 + 0.719688t\right), \\ u^*(t) &= 0.5(u^*(t) + 1)12, \end{aligned} \tag{9.48}$$

for $t \in [0, 54]$.

The optimal control obtained by the neural network is plotted in Figure 9.13. On the other hand, the optimal trajectories for the cell mass concentra-

tion, substrate concentration, product concentration and broth volume are depicted in Figures 9.14, 9.15, 9.16 and 9.17, respectively. All these process histories for the state variables are quite smooth functions. This is a desired property, since the process is desired to be stable. Finally, the optimal specific growth rate and productivity are plotted in Figures 9.18 and 9.19.

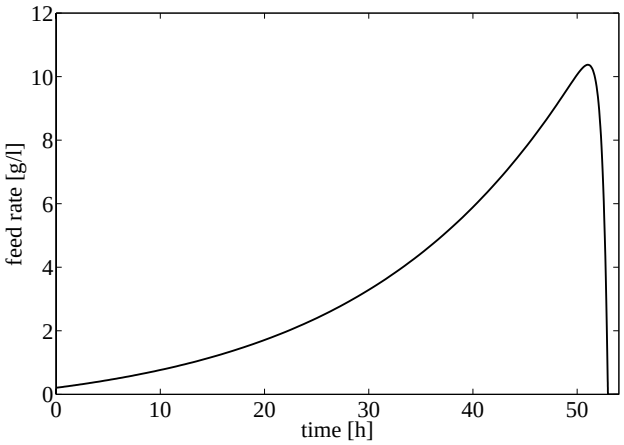


Figure 9.13: Optimal control for the fed batch fermenter.

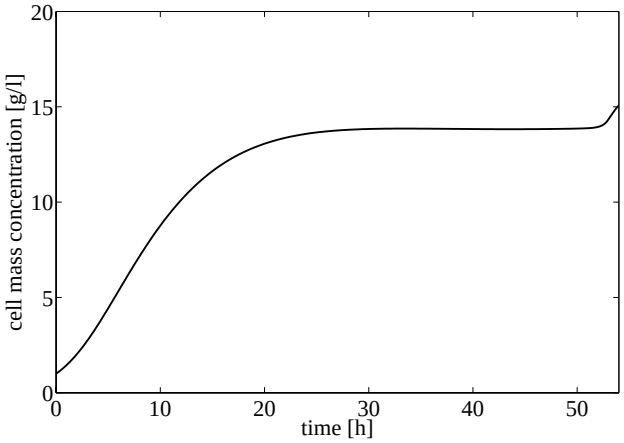


Figure 9.14: Optimal trajectory for the concentration of cell mass in fed batch fermenter.

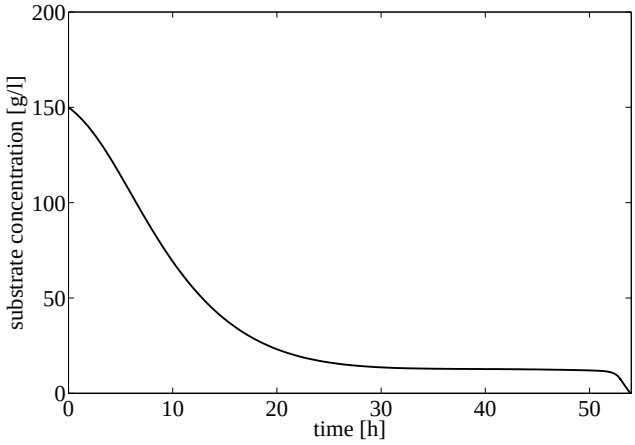


Figure 9.15: Optimal trajectory for the concentration of substrate in the fed batch fermenter.

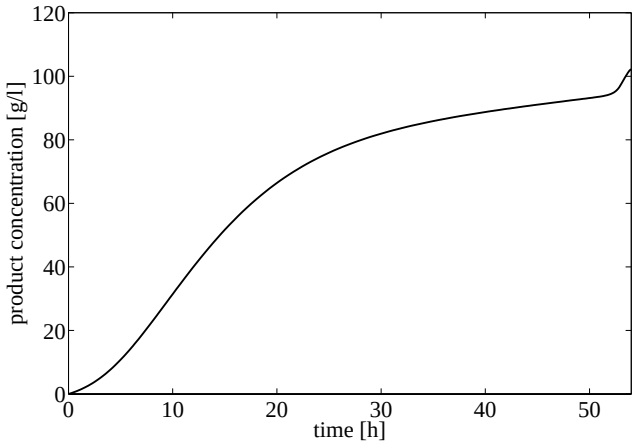


Figure 9.16: Optimal trajectory for the concentration of product in the fed batch fermenter.

9.4 Related code

Flood includes several examples of optimal control problems. These include the car problem, the car problem neurocomputing, the fed batch fermenter problem and the aircraft landing problem.

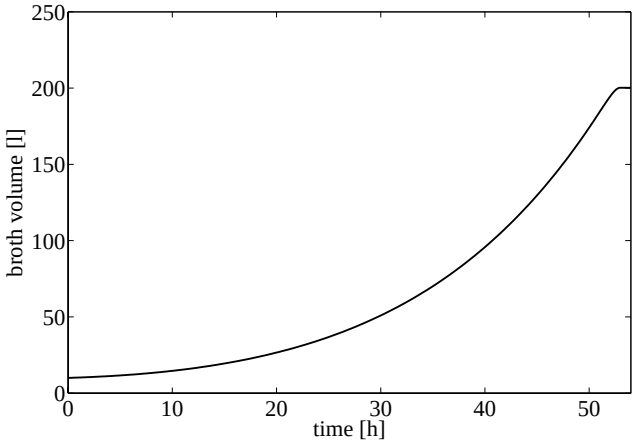


Figure 9.17: Optimal trajectory for the broth volume in the fed batch fermenter problem.

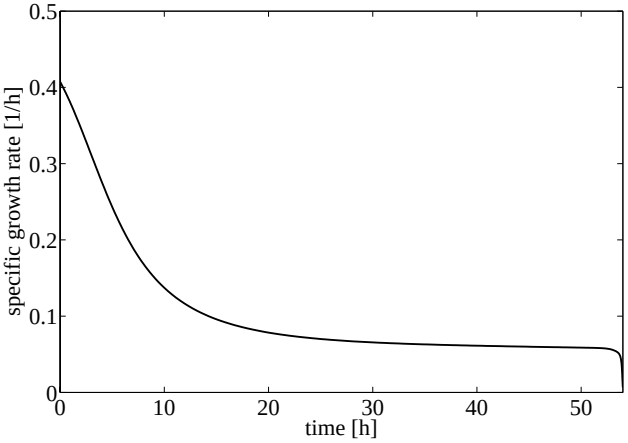


Figure 9.18: Optimal specific growth rate in the fed batch fermenter problem.

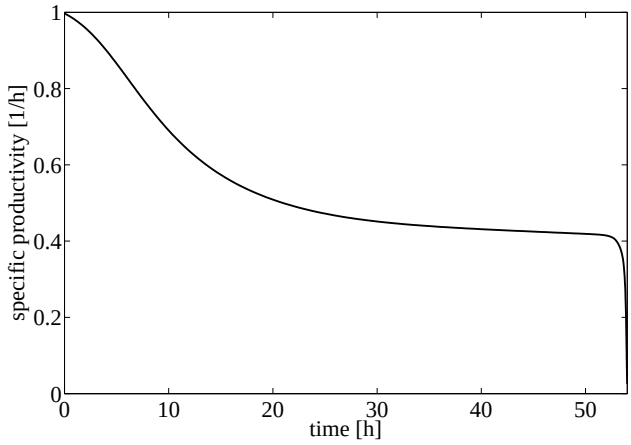


Figure 9.19: Optimal specific productivity in the fed batch fermenter problem.

Chapter 10

Optimal shape design

10.1 Problem formulation

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 [38]. 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:
- A mathematical model of the system. - A multilayer perceptron. - An objective functional. - A training algorithm.

Mathematical model

The mathematical model is a well-formed formula which involves the physical form of the device to be optimized. Let define $\mathbf{y}(\mathbf{x})$ the shape variables and $\mathbf{u}(\mathbf{x})$ the state variables. The mathematical model or state equation can then be written as

$$\mathcal{L}(\mathbf{y}(\mathbf{x}), \mathbf{u}(\mathbf{x}), \mathbf{x}) = \mathbf{f}, \tag{10.1}$$

where $\mathcal{L}$ is some algebraic or differential operator and $\mathbf{f}$ some forcing term.

Multilayer perceptron

A multilayer perceptron is used to represent the shape variables. Optimal shape design problems are usually defined by constraints on the shape func-

tion. Two important types of shape constraints are boundary conditions and lower and upper bounds.

Objective functional

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

Definition 3 (Admissible shape) *A shape* $\mathbf{y}(\mathbf{x})$ *which satisfies all the constraints is called an admissible shape. The set of admissible shapes is denoted* Y*, and the notation* $\mathbf{y}(\mathbf{x}) \in Y$ *means that* $\mathbf{y}(\mathbf{x})$ *is admissible.*

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

Definition 4 (Admissible state) *A state* $\mathbf{u}(\mathbf{x})$ *which satisfies the state variable constraints is called an admissible state. The set of admissible states will be denoted by* $\mathbf{U}$*, and* $\mathbf{u}(\mathbf{x}) \in \mathbf{U}$ *means that the state* $\mathbf{u}(\mathbf{x})$ *is admissible.*

Objective functional

The performance criterion expresses how well a given design does the activity for which it has been built. In optimal shape design the performance criterion is a functional of the form $F : V \to \mathbb{R}$,

$$F = F[\mathbf{y}(\mathbf{x})]$$

Optimal shape design problems solved in practice are, as a rule, multicriterion 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.

To sum up, the optimal shape design problem can be formulated as

Problem 8 (Optimal shape design problem) *Let Y and U be the function spaces of admissible shapes and states, respectively. Find an admissible shape $\mathbf{y}^*(\mathbf{x}) \in Y$ which causes the system*

$$\mathcal{L}(\mathbf{y}(\mathbf{x}), \mathbf{u}(\mathbf{x}), \mathbf{x}) = \mathbf{f}$$

to be in an admissible state $\mathbf{u}^*(\mathbf{x}) \in U$ and for which the objective functional

$$F[\mathbf{y}(\mathbf{x})]$$

takes on a minimum or maximum value. The function $\mathbf{y}^*(\mathbf{x})$ is called an optimal shape and the function $\mathbf{u}^*(\mathbf{x})$ an optimal state.

Training algorithm

In general, there are no automatic solutions to optimal shape design problems. Therefore, the use of direct methods usually becomes necessary.

A variational formulation for neural networks provides a direct method for the solution of variational problems. Therefore optimal shape design problems can be approached with this numerical technique.

10.2 A simple example

The minimum drag problem for the multilayer perceptron is 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. This problem is included with the Flood library [31], and it has been published in [33] and [32].

Problem statement

Consider the design of a body of revolution with given length l and diameter d providing minimum drag at zero angle of attack and for neglected friction effects, see Figure 10.1.

The drag of such a body $y(x)$ can be expressed as

$$D[y(x)] = 2\pi q \int_0^l y(x)[C_p y'(x)]dx, \tag{10.2}$$

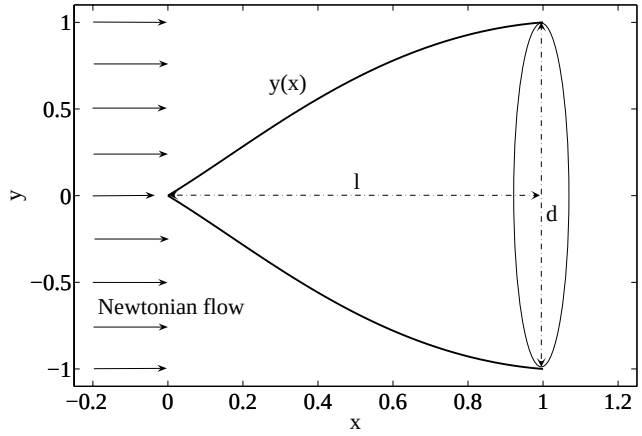


Figure 10.1: The minimum drag problem statement.

where q is the free-stream dynamic pressure and C_p the pressure coefficient [9]. For a slender body $y'(x) \ll 1$, the pressure coefficient can be approximated by the Newtonian flow relation

$$C_p = 2\left[y'(x)\right]^2, \tag{10.3}$$

which is valid provided that the inequality $y'(x) \geq 0$ is satisfied.

From Equations (10.2) and (10.3) we obtain the following approximation for the drag,

$$D[y(x)] = 4\pi q \int_0^l y(x)[y'(x)]^3 dx. \tag{10.4}$$

It is convenient to introduce the following dimensionless variables associated with the axial coordinate and the radial coordinate

$$\xi = \frac{x}{l}, \tag{10.5}$$

$$\eta = \frac{2y}{d}. \tag{10.6}$$

In that way, both ξ and η vary from 0 to 1.

Also, a dimensionless coefficient associated with the drag can be defined as

$$C_D[\eta(\xi)] \;=\; \tau^2 \int_0^1 \eta(\xi)[\eta'(\xi)]^3 d\xi, \tag{10.7}$$

where $\tau = d/l$ is the slenderness of the body.

The analytical solution to the minimum drag problem formulated in this section is given by

$$\eta^*(\xi) = \xi^{3/4}, \tag{10.8}$$

which provides a minimum value for the drag coefficient $C_D/\tau^2 = 0.4220$.

Selection of function space

The body of revolution $\eta(\xi)$, for $\xi \in [0, 1]$, will be represented by a multilayer perceptron with a sigmoid hidden layer and a linear output layer. This axisymmetric structure is to be written in cartesian coordinates, so the neural network must have one input and one output neuron. On the other hand, an appropriate number of hidden neurons is believed to be three for this particular application. This network architecture is depicted in Figure 10.2.

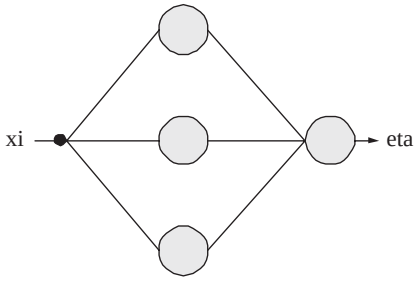


Figure 10.2: Network architecture for the minimum drag problem.

Such a multilayer perceptron spans a family V of parameterized functions $\eta(\xi)$ of dimension $s = 10$, which is the number of parameters in the network. The elements of this function space are of the form

$\eta : \mathbb{R} \to \mathbb{R}$

where

$$\eta(\xi) = b_1^{(2)} + \sum_{j=1}^{3} w_{1j}^{(2)} \cdot \tanh\left(b_j^{(1)} + w_{j1}^{(1)}\xi\right). \tag{10.9}$$

The outputs from the neural network in Figure 10.2 must hold the boundary conditions $\eta(0) = 0$ and $\eta(1) = 1$. A suitable set of particular and homogeneous solution terms here is

$$\varphi_0(\xi) = \xi, \tag{10.10}$$
$$\varphi_1(\xi) = \xi(\xi - 1), \tag{10.11}$$

respectively. This gives

$$\eta(\xi) = \xi + \xi(\xi - 1)\eta(\xi). \tag{10.12}$$

Also, the functions $\eta(\xi)$ are constrained to lie in the interval $[0, 1]$. To deal with such constraints the neural network outputs are bounded in the form

$$\eta(\xi) = \begin{cases} 0, & \eta(\xi) < 0. \\ \eta(\xi), & 0 \leq \eta(\xi) \leq 1. \\ 1, & \eta(\xi) > 1. \end{cases} \tag{10.13}$$

The elements of the function space constructed so far indeed satisfy the boundary conditions and the input constraints. Also, they are thought to have a correct complexity for this case study.

Experience has shown that this method does not require a good initial guess for the solution, so the parameters in the neural network are initialized at random. This is a potential advantage over other direct methods, in which a good initial guess for the solution might be needed. Figure 10.3 depicts the starting random shape for this problem.

Formulation of variational problem

From Equation (10.7), the variational statement of this problem is to find a function $\eta^*(\xi) \in V$ for which the functional

$$C_D[\eta(\xi)]/\tau^2 = \int_0^1 \eta(\xi)[\eta'(\xi)]^3 d\xi, \tag{10.14}$$

defined on V, takes on a minimum value.

In order to evaluate the objective functional in Equation (10.14) the integration of a function is needed. Here we apply the Runge-Kutta-Fehlberg method [49] with tolerance 10^{-6}.

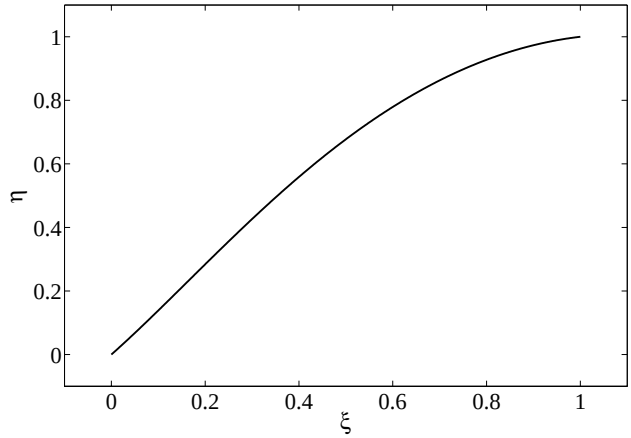


Figure 10.3: Initial guess for the minimum drag problem.

Solution of reduced function optimization problem

Here we use a quasi-Newton method with BFGS train direction and Brent optimal train rate methods for training [8]. The tolerance in the Brent's method is set to 10^{-6}.

The objective function gradient vector $\nabla f(\boldsymbol{\zeta})$ is calculated by means of numerical differentiation. In particular, the symmetrical central differences method is used with $\epsilon = 10^{-6}$ [8].

The evaluation and the gradient norm of the initial guess are 0.56535 and 0.324097, respectively. Trained is performed until the algorithm can not perform any better, that is, when the Brent's method gives zero train rate for a gradient descent train direction. This occurs after 759 epochs, at which the objective function evaluation is 0.422. At this point the gradient norm takes a value of $2.809 \cdot 10^{-4}$. Figures 10.4 and 10.5 illustrate this training process.

Table 10.1 shows the training results for this problem. Here N is the number of training epochs, M the number of objective function evaluations, CPU the CPU time for a laptop AMD 3000, $\|\boldsymbol{\zeta}^*\|$ the final parameters norm, $f(\boldsymbol{\zeta}^*)$ the final value for the objective function and $\|\nabla f(\boldsymbol{\zeta}^*)\|$ the final gradient norm.

Comparing the drag coefficient provided by that neural network (0.4223) to that by the analytical result (0.4220), these two values are almost the same. In particular, the percentage error made by the numerical method is less than 0.1%.

The optimal shape design by the multilayer perceptron is depicted in

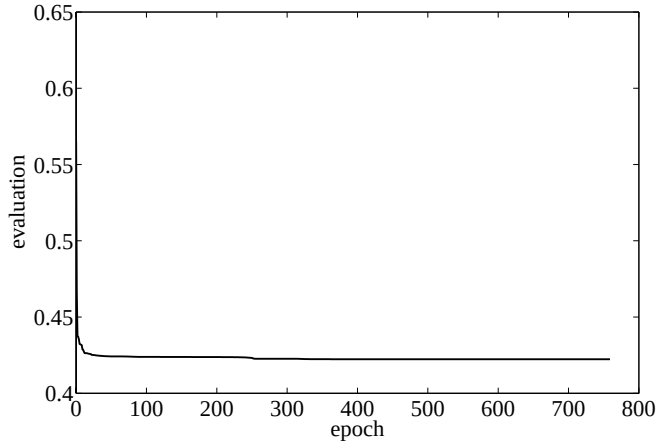


Figure 10.4: Evaluation history for the minimum drag problem.

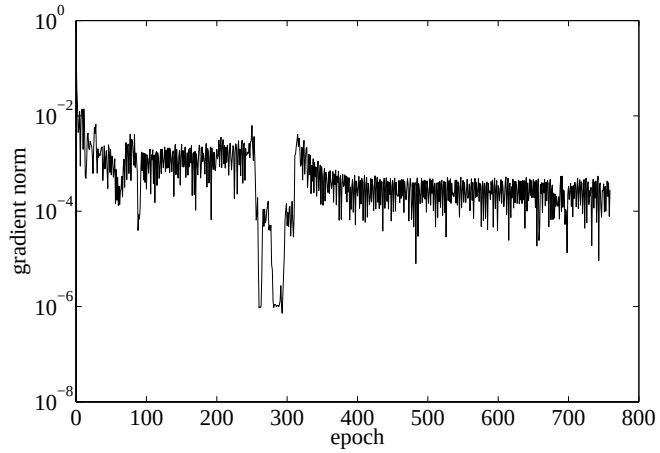


Figure 10.5: Gradient norm history for the minimum drag problem.

Figure 10.6.

Finally, an explicit expression of the shape of such an axisymmetric body is given by

$$\begin{aligned}\eta^*(\xi) &= \xi + \xi(\xi-1)[-164.639 \\ &- 275.014\tanh(-2.97601 - 27.2435\xi) \\ &- 79.9614\tanh(-2.62125 - 3.7741\xi) \\ &+ 201.922\tanh(-1.78294 + 0.0113036\xi)]. \qquad (10.15)\end{aligned}$$

N	=	759
M	=	38426
CPU	=	354
$\|\boldsymbol{\zeta}^*\|$	=	122.752
$f(\boldsymbol{\zeta}^*)$	=	0.422325
$\|\nabla f(\boldsymbol{\zeta}^*)\|$	=	$2.80939 \cdot 10^{-4}$

Table 10.1: Training results for the minimum drag problem.

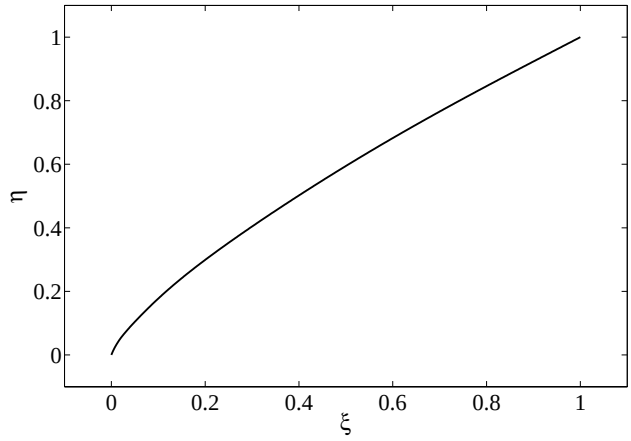


Figure 10.6: Neural network results to the minimum drag problem.

10.3 Related code

Flood includes one optimal shape design example, which is the minimum drag problem.

Chapter 11

Inverse problems

11.1 Problem formulation

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 [30]. 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 [30]. Inverse problems are found in many areas of science and engineering.

An inverse problem is specified by:
- A mathematical model.
- An experimental data set.
- A multilayer perceptron.
- An error functional.
- A training algorithm.

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.

Let us represent $\mathbf{y}(\mathbf{x})$ the vector of unknowns (inputs or properties) and $\mathbf{u}(\mathbf{x})$ the vector of state variables. The mathematical model, or state equation, relating unknown and state variables takes the form

$$\mathcal{L}(\mathbf{y}(\mathbf{x}), \mathbf{u}(\mathbf{x}), \mathbf{x}) = \mathbf{f}, \tag{11.1}$$

where $\mathcal{L}$ is some algebraic or differential operator and $\mathbf{f}$ is some forcing term.

Observed data

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 set of experimental values on the state variables is needed in order to estimate the unknown variables of that system. This observed data is here denoted $\widehat{\mathbf{u}}(\mathbf{x})$.

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

Multilayer perceptron

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

Error functional

For inverse problems, the presence of restrictions is typical. Two possible classes of constraints are unknowns and state constraints.

The former are defined by the allowable values on the inputs or the properties of the system, depending on whether the we are talking about an input or a property estimation problem. Two typical types of constraints here are boundary conditions and lower and upper bounds.

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 [29].

Definition 5 (Admissible unknown) *An unknown* $\mathbf{y}(\mathbf{x})$ *which satisfies all the constraints is called an admissible unknown. The set of admissible unknowns can be denoted* Y*, and the notation* $\mathbf{y}(\mathbf{x}) \in Y$ *means that the unknown* $\mathbf{y}(\mathbf{x})$ *is admissible.*

Also, a state which satisfies the state constraints is called an admissible state [29].

Definition 6 (Admissible state) *A state* $\mathbf{u}(\mathbf{x})$ *which satisfies the state constraints is called an admissible state. The set of admissible states will be denoted by* $\mathbf{U}$*, and* $\mathbf{u}(\mathbf{x}) \in \mathbf{U}$ *means that* $\mathbf{u}(\mathbf{x})$ *is admissible.*

Error functional

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 $\mathbf{u}(\mathbf{x})$ to the observed data $\widehat{\mathbf{u}}(\mathbf{x})$:

The error functional $E : Y \to \mathbb{R}$ is of the form

$$E[\mathbf{y}(\mathbf{x})] = \|\mathbf{u}(\mathbf{x}) - \widehat{\mathbf{u}}(\mathbf{x})\|, \tag{11.2}$$

where any of the generally used norms may be applied to characterize the proximity of $\mathbf{u}(\mathbf{x})$ and $\widehat{\mathbf{u}}(\mathbf{x})$. Some of them are the sum squared error or the Minkowski error. Regularization theory can also be applied here [10].

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

Problem 9 (Inverse problem) *Let* Y *and* U *be the function spaces of all admissible unknowns and states, respectively. Find an admissible unknown* $\mathbf{y}^*(\mathbf{x}) \in Y$ *which causes the system*

$$\mathcal{L}(\mathbf{y}(\mathbf{x}), \mathbf{u}(\mathbf{x}), \mathbf{x}) = \mathbf{f}$$

to follow an admissible state $\mathbf{u}^*(\mathbf{x}) \in U$*, and for which the error functional*

$$E = E[\mathbf{y}(\mathbf{x})],$$

defined on Y*, takes on a minimum value.*

On the other hand, inverse problems might be ill-posed [51]. 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 [17].

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 algorithm

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

11.2 A simple example

Problem statement

For this problem, consider a 2-dimensional inhomogeneous medium with domain Ω and boundary Γ, in which the thermal conductivity is to be estimated. The mathematical model of heat transfer here is

$$\begin{aligned} \nabla\left(\rho(x,y)\nabla T(x,y;t)\right) &= \frac{\partial T(x,y;t)}{\partial t} \quad in \quad \Omega, && (11.3)\\ T(x,y;0) &= T_0 \quad in \quad \Omega, && (11.4)\\ T(x,y;t) &= T_\Gamma \quad on \quad \Gamma, && (11.5) \end{aligned}$$

for $t \in [0, t_f]$, and where $\rho(x, y)$ is the thermal conductivity, T_0 is the initial temperature and T_Γ is the boundary temperature.

On the other hand, experimental data is obtained from measurements of the temperature for different time steps and at different points on the domain

$$\begin{array}{cccccc} t_1 & \hat{T}_{11} & \hat{T}_{12} & \dots & \hat{T}_{1Q} \\ t_2 & \hat{T}_{21} & \hat{T}_{22} & \dots & \hat{T}_{2Q} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ t_P & \hat{T}_{P1} & \hat{T}_{P2} & \dots & \hat{T}_{PQ} \end{array}$$

where P and Q are the number of time steps and points considered, respectively.

Figure 11.1 is a graphical statement of this situation.

The goal here is to estimate the thermal conductivity $\rho(x, y)$, so that the heat equation model matches the experimental data. In this way, the objective functional for this problem can be the mean squared error between the computed temperature for a given thermal conductivity and the measured temperature,

$$E[\rho(x,y)] = \frac{1}{PQ}\sum_{i=1}^{P}\left(\sum_{j=1}^{Q}\left(T(x_j,y_j;t_i) - \hat{T}_{ij}\right)^2\right). \qquad (11.6)$$

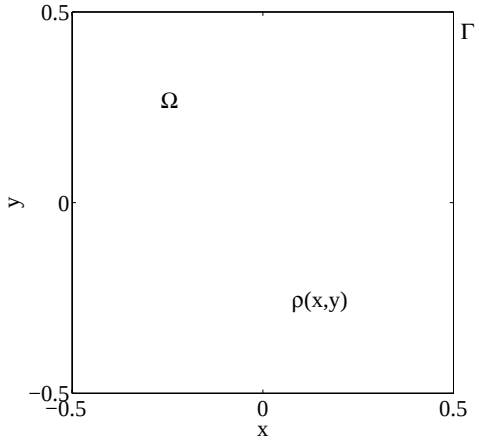


Figure 11.1: The thermal conductivity estimation problem statement.

For this example we take the problem domain to be the square $\Omega = \{(x, y) : |x| \leq 0.5, |y| \leq 0.5\}$ with boundary $\Gamma = \{(x, y) : |x| = 0.5, |y| = 0.5\}$. That domain is discretized with a triangular mesh composed of 888 elements and 485 nodes, and the time is also discretized in 11 time steps. On the other hand, artificial temperature data is generated with the following expression for the thermal conductivity, which will be considered to be the analytical solution to this problem,

$$\rho^*(x, y) = x^2 + y^2. \tag{11.7}$$

Selection of function space

The thermal conductivity $\rho(x, y)$ in Ω is represented by a multilayer perceptron with a sigmoid hidden layer and a linear output layer. The neural network must have two inputs and one output neuron. We guess a good number of neurons in the hidden layer to be three. This neural network is denoted as a $2 : 3 : 1$ multilayer perceptron.

It spans a family V of parameterized functions $\rho(x, y)$ of dimension $s = 13$, which is the number of parameters in the neural network. Figure 11.3 is a graphical representation of this network architecture.

Such a multilayer perceptron spans a family V of parameterized functions $\rho(x, y)$ of dimension $s = 19$, which is the number of parameters in the network. Elements V are of the form

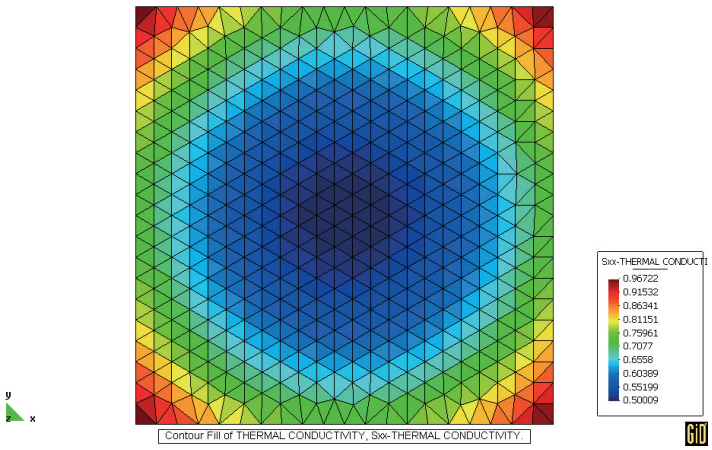


Figure 11.2: Analytical thermal conductivity in Ω.

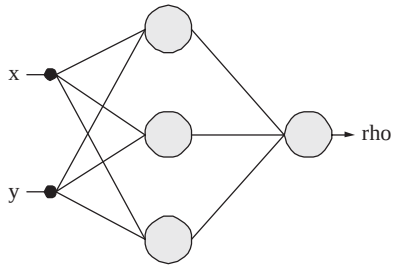


Figure 11.3: Network architecture for the thermal conductivity estimation problem.

$$\rho : \mathbb{R}^2 \to \mathbb{R}$$

where

$$\rho(x, y) = b_1^{(2)} + \sum_{j=1}^{3} w_{1j}^{(2)} \cdot \tanh\left(b_j^{(1)} + w_{j1}^{(1)} x + w_{j2}^{(1)} y\right). \tag{11.8}$$

On the other hand, the biases and synaptic weight are initialized at random, which means that a random initial guess is used for the thermal conductivity.

Formulation of variational problem

Following Equation 11.6 the variational statement for this properties estimation inverse problem is to find a function $\rho^*(x, y)$ for which the mean squared error functional

$$E[\rho(x,y)] = \frac{1}{PQ}\sum_{i=1}^{P}\left(\sum_{j=1}^{Q}\left(T(x_j,y_j;t_i)-\hat{T}_{ij}\right)^2\right), \qquad (11.9)$$

defined on V takes on a minimum value

Evaluation of the objective functional (11.9) requires a numerical method for integration of partial differential equations. Here we choose the Finite Element Method [57]. For this problem we use a triangular mesh with 888 elements and 485 nodes.

Last, evaluation of the gradient, $\nabla f(\zeta)$, is carried out by means of numerical differentiation. In particular, the central differences method is used with an ϵ value of 10^{-6} [8] .

Solution of reduced function optimization problem

A suitable training algorithm for this problem is a quasi-Newton method with BFGS train direction and Brent optimal train rate methods for training [45]. Indeed, this training algorithm has demonstrated fully convergence to the global optimum in this problem.

In this example, we set the training algorithm to stop after when the training rate is zero for a gradient descent train direction. During the training process, which lasts for 168 epochs, the error decreases from 1.117 to $2.868 \cdot 10^{-5}$, and the gradient norm from 0.466 to $1.106 \cdot 10^{-6}$. Figures 11.4 and 11.5 show, with a logarithmic scale for the y-axis, the evaluation and gradient norm histories of the neural network training process.

Table 11.1 shows the training results for this problem. Here N denotes the number of training epochs, M the number of objective functional evaluations, CPU the CPU time in seconds for a laptop AMD 3000, $\|\zeta\|$ the final parameters norm, $f(\zeta^*)$ the final objective function evaluation, and $\|\nabla f(\zeta^*)\|$ its gradient norm.

The solution here is good, since the mean squared error between the output from the model and the experimental data is of order 10^{-6} and the estimated thermal conductivity matches very well the actual thermal conductivity given by Equation (11.7). Figure 11.6 shows the thermal conductivity estimated by the neural network for this problem.

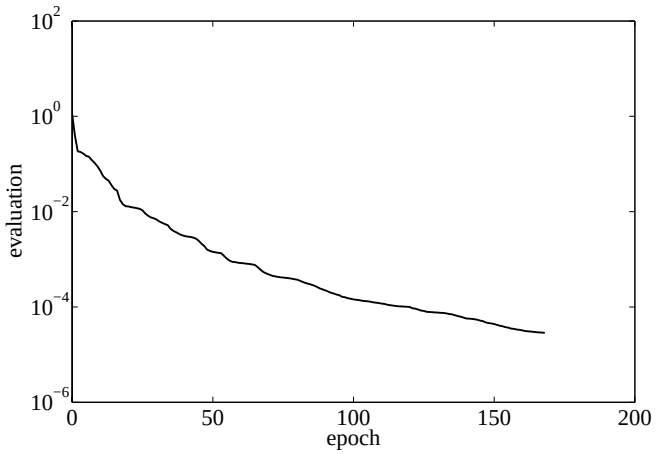


Figure 11.4: Evaluation history for the thermal conductivity estimation problem.

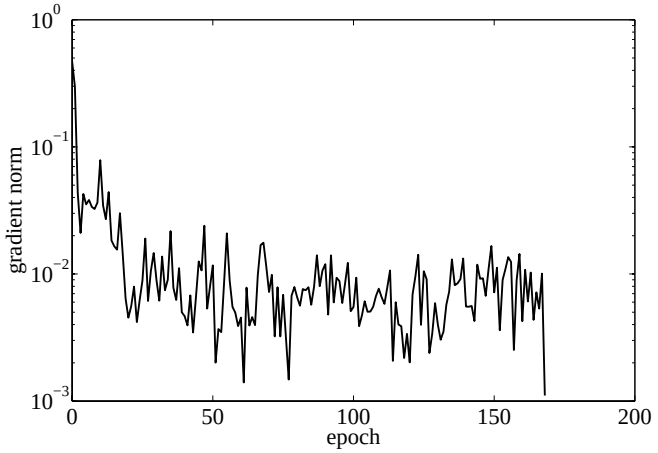


Figure 11.5: Gradient norm history for the thermal conductivity estimation problem.

Finally, the analytical expression of the function represented by the trained multilayer perceptron is as follows,

N	$=$	168
M	$=$	10476
CPU	$=$	16253
$\|\zeta^*\|$	$=$	21.606
$f(\zeta^*)$	$=$	$2.868 \cdot 10^{-5}$
$\|\nabla f(\zeta^*)\|$	$=$	$1.106 \cdot 10^{-3}$

Table 11.1: Training results for the thermal conductivity estimation problem.

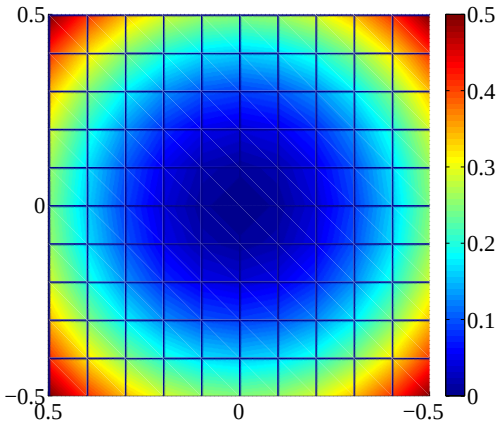


Figure 11.6: Neural network results to the thermal conductivity estimation problem.

$$\begin{aligned}\rho^*(x,y) &= 51.6455 \\ &+ 45.7732 \tanh\left(-0.551456 - 0.143683x - 0.0824186y\right) \\ &- 30.2139 \tanh\left(0.646978 - 0.238919x + 0.0769619y\right) \\ &- 19.8059 \tanh\left(0.661936 - 0.00304418x - 0.333647y\right). \quad (11.7)\end{aligned}$$

11.3 Related code

Flood does not include any inverse problem example, since they usually require external codes in order to calculate the system states.

Chapter 12

Function optimization

12.1 Problem formulation

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

While the multilayer perceptron naturally leads to the solution of variational problems, Flood 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 objective function defines the optimization problem itself.

The function regression problem [24] can be regarded as the problem of approximating a function from data. The formulation of a function regression problem requires:
- A multilayer perceptron.
- An objective functional.
- A training algorithm.

Multilayer perceptron

This multilayer perceptron spans a vector space V of dimension d. The elements of that space, $\mathbf{y} : X \to Y$, are of the form

$$\boldsymbol{\zeta} = (\zeta_1, \ldots, \zeta_d). \tag{12.1}$$

That parameterized space of functions will be the basis to aproximate the regression function.

Objective functional

The simplest function optimization problems are those in which no constraints are posed on the solution. The general unconstrained function optimization problem can be formulated as follows:

Problem 10 (Unconstrained function optimization problem) *Let $V \subseteq \mathbb{R}^d$ be a real vector space. Find a vector $\boldsymbol{\zeta}^* \in V$ for which the function $f : V \to \mathbb{R}$ defined by*

$$f = f(\boldsymbol{\zeta})$$

takes on a minimum or a maximum value.

The function $f(\boldsymbol{\zeta})$ is called the objective function. The domain of the objective function for a function optimization problem is a subset V of $\mathbb{R}^d$, and the image of that function is the set $\mathbb{R}$. The integer d is known as the number of variables in the objective function.

The vector at which the objective function takes on a minimum or maximum value is called the minimal or the maximal argument of that function, respectively. The tasks of minimization and maximization are trivially related to each other, since maximization of $f(\boldsymbol{\zeta})$ is equivalent to minimization of $-f(\boldsymbol{\zeta})$, and vice versa. Therefore, without loss of generality, we will assume function minimization problems.

On the other hand, a minimum can be either a global minimum, the smallest value of the function over its entire range, or a local minimum, the smallest value of the function within some local neighborhood. Functions with a single minimum are said to be unimodal, while functions with many minima are said to be multimodal.

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 as follows:

Problem 11 (Constrained function optimization problem) *Let $V \subseteq \mathbb{R}^d$ be a real vector space. Find a vector $\boldsymbol{\zeta}^* \in V$ such that the functions $c_i : V \to \mathbb{R}$, for $i = 1, \ldots, l$ and defined by*

$$c_i = c_i(\boldsymbol{\zeta})$$

hold $c_i(\boldsymbol{\zeta}^*) = 0$, for $i = 1, \ldots, l$, and for which the function $f : V \to \mathbb{R}$ defined by

$$f = f(\boldsymbol{\zeta})$$

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.

A common approach when solving a constrained function optimization problem is to reduce it into an unconstrained problem. This can be done by adding a penalty term to the objective function for each of the constrains in the original problem. Adding a penalty term gives a large positive or negative value to the objective function when infeasibility due to a constrain is encountered.

For the minimization case, the general constrained function optimization problem can be reformulated as follows:

Problem 12 (Reduced constrained function optimization problem) *Let $v \subseteq \mathbb{R}^d$ be a real vector space, and let $\rho_i > 0$, for $i = 1, \ldots, l$, be real numbers. Find a vector $\boldsymbol{\zeta}^* \in V$ for which the function $f : V \to \mathbb{R}$ defined by*

$$\bar{f}(\boldsymbol{\zeta}) = f(\boldsymbol{\zeta}) + \sum_{i=1}^{l} \rho_i \left(c_i(\boldsymbol{\zeta})\right)^2,$$

takes on a minimum value.

The parameters ρ_i, for $i = 1, \ldots, l$, are called the penalty term ratios, being l the number of constraints.

For large values of the ratios ρ_i, it is clear that the solution $\boldsymbol{\zeta}^*$ of Problem 12 will be in in a region where $c_i(\boldsymbol{\zeta})$ are small.

Thus, for increasing values of ρ_i it is expected that the the solution $\boldsymbol{\zeta}^*$ of Problem 12 will approach the constraints and, subject to being close, will minimize the objective function $f(\boldsymbol{\zeta})$. Ideally then, as $\rho_i \to \infty$ the solution of Problem 12 will converge to the solution of Problem 11 [35].

Note that, while the squared norm of the constrained is the metric most used, any other suitable metric can be used.

Many optimization algorithms use the gradient vector of the objective function to search for the minimal argument. The gradient vector of the objective function is written:

$$\nabla f = \left(\frac{\partial f}{\partial \zeta_1}, \ldots, \frac{\partial f}{\partial \zeta_d} \right). \tag{12.-2}$$

While for some objective functions the gradient vector can be evaluated analytically, there are many applications when that is not possible, and the objective function gradient vector needs to be computed numerically. This can be done by perturbing each argument element in turn, and approximating the derivatives by using the forward or the central differences methods.

The objective function Hessian matrix

There are some optimization algorithms which also make use of the Hessian matrix of the objective function to search for the minimal argument. The Hessian matrix of the objective function is written:

$$Hf = \begin{pmatrix} \dfrac{\partial^2 f}{\partial \zeta_1^2} & \cdots & \dfrac{\partial^2 f}{\partial \zeta_1 \partial \zeta_d} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial \zeta_d \partial \zeta_1} & \cdots & \dfrac{\partial^2 f}{\partial \zeta_d^2} \end{pmatrix} \tag{12.-1}$$

As it happens for the gradient vector, there are many applications when analytical evaluation of the Hessian is not possible, and it must be computed numerically. This can be done by perturbing each argument element in turn, and approximating the derivatives by using the forward or the central differences method.

Training algorithm

The training algorithm is the solving algorithm 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 A simple example

The problem in this example is to find the minimum point on the plane $\zeta_1 + \zeta_2 = 1$ which also lies in the cylinder $\zeta_1^2 + \zeta_2^2 = 1$.

Figure 12.1 is a graphical representation of this function optimization problem.

This constrained function optimization problem can be stated as:

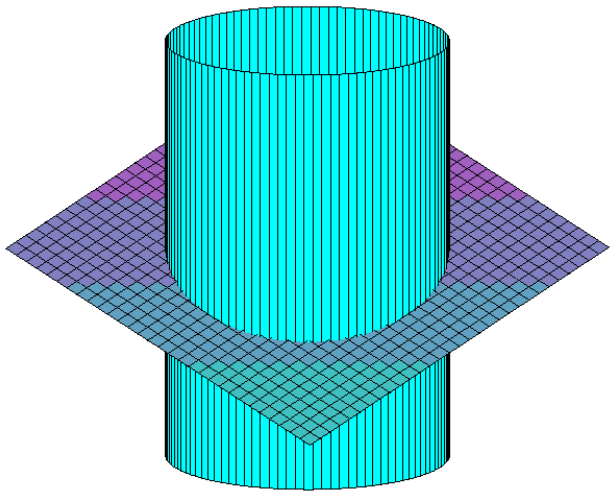


Figure 12.1: The plane-cylinder function optimization problem.

Problem 13 (Plane-cylinder function optimization problem) *Let* $V = [-1, 1]^2$ *be a real vector space. Find a vector* $\boldsymbol{\zeta}^* \in X$ *such that the function* $c : V \to \mathbb{R}$ *defined by*

$$c(\boldsymbol{\zeta}) = \zeta_1^2 + \zeta_2^2 - 1, \tag{12.0}$$

holds $c(\boldsymbol{\zeta}^*) \leq 0$ *and for which the function* $f : X \to \mathbb{R}$ *defined by*

$$f(\boldsymbol{\zeta}) = \zeta_1 + \zeta_2 - 1, \tag{12.1}$$

takes on a minimum value.

This constrained problem can be reduced to an unconstrained problem by the use of a penalty function:

Problem 14 (Reduced plane-cylinder function optimization problem) *Let* $V \subseteq \mathbb{R}^2$ *be a real vector space, and let* $\rho \in \mathbb{R}^+$ *be a positive real number. Find a vector* $\boldsymbol{\zeta}^* \in V$ *for which the function* $\bar{f} : V \to \mathbb{R}$ *defined by*

$$\bar{f}(\boldsymbol{\zeta}) = \zeta_1 + \zeta_2 - 1 + \rho \left(\zeta_1^2 + \zeta_2^2 - 1\right)^2,$$

takes on a minimum value.

12.3 Related code

`Flood` includes the examples `DeJongFunction`, `RosenbrockFunction`, `RastriginFunction`, and `PlaneCylinder` to represent the concepts of the De Jong's, Rosenbrock's, Rastrigin's and Plane-Cylinder objective functions, respectively.

Appendix A

Software model

In this Appendix we present the software model of Flood. The whole process is carried out in the Unified Modeling Language (UML), which provides a formal framework for the modeling of software systems. The final implementation is to be written in the C++ Programming Language.

A.1 The 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 [47].

UML class diagrams are the mainstay of object-oriented analysis and design. They show the classes of the system, their interrelationships and the attributes and operations of the classes.

In order to construct a model for the multilayer perceptron, 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 for the multilayer perceptron, we iteratively model:

1. Classes.
2. Associations.
3. Derived classes.
4. Attributes and operations.

A.2 Classes

In colloquial terms a concept is an idea or a thing. In object-oriented modeling concepts are represented by means of classes [50]. 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 [47].

Through all this work, we have seen that neural networks are characterized by a neuron model, a network architecture, an objective functional and a training algorithm. The characterization in classes of these four concepts for the multilayer perceptron is as follows:

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

Network architecture The class representing the concept of network architecture in the multilayer perceptron is called `MultilayerPerceptron`.

Objective functional The class which represents the concept of objective functional in a multilayer perceptron is called `ObjectiveFunctional`.

Training algorithm The class representing the concept of training algorithm in a multilayer perceptron is called `TrainingAlgorithm`.

Figure A.1 depicts a starting UML class diagram for the conceptual model of the multilayer perceptron.

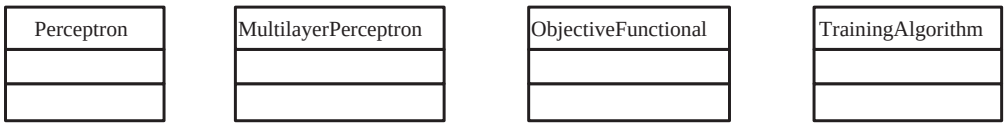


Figure A.1: A conceptual diagram for the multilayer perceptron.

A.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 [50]. 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 [47].

The appropriate associations in the system are next identified to be included to the UML class diagram of the system:

Neuron model - Multilayer perceptron A multilayer perceptron *is built by* perceptrons.

Network architecture - Objective functional A multilayer perceptron *has assigned* an objective functional.

Objective functional - Training algorithm An objective functional *is improved by* a training algorithm.

Figure A.2 shows the above UML class diagram with these associations aggregated.

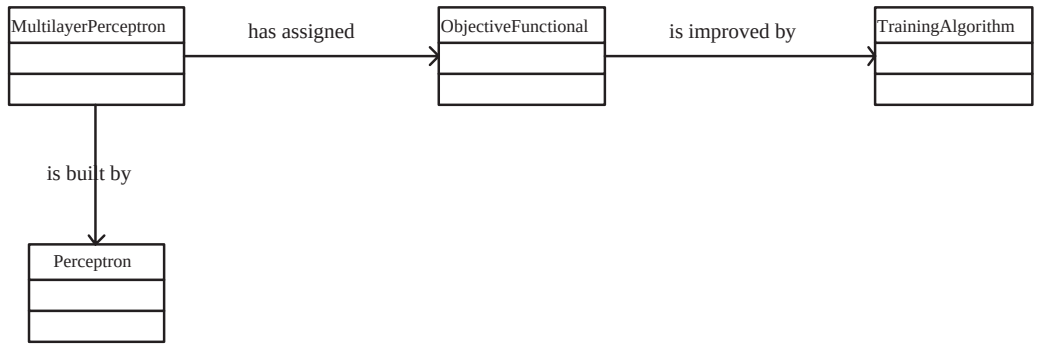


Figure A.2: Aggregation of associations to the conceptual diagram.

A.4 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 [50]. Associations between a base class an a derived class are of the kind *is a* [47].

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:

Neuron model The class `Perceptron` is concrete, because it represents an actual neuron model. Therefore a perceptron object can be instantiated.

Network architecture The class `MultilayerPerceptron` is a concrete class and is itself suitable for instantiation.

Objective functional The class `ObjectiveFunctional` is abstract, because it does not represent a concrete objective functional for the multilayer perceptron. The objective functional for the multilayer perceptron depends on the problem at hand.

Some suitable error functionals for data modeling 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 `ObjectiveFunctional` abstract class. All of these error functionals are measured on an input-target data set, so we add to the model a class which represents that concept. This is called `InputTargetDataSet`, and it is a concrete class.

In order to solve other types of variational applications, such as optimal control, inverse problems or optimal shape design, a new concrete class must be in general derived from the `ObjectiveFunctional` abstract class. However, in order to facilitate that task `Flood` includes some examples which can be used as templates to start with. For instance, the `BrachistochroneProblem` or `IsoperimetricProblem` classes are derived to solve two classical problems in the calculus of variations. Other concrete classes for some specific optimal control, inverse problems or optimal shape design are also derived.

On the other hand, evaluation of the objective functional in some applications requires integrating functions, ordinary differential equations or a partial differential equations. In this way, we add to the model the utility classes called `IntegrationOfFunctions` and `OrdinaryDifferentialEquations` for the two firsts. For integration of partial differential equations the use of the Kratos software is suggested [14].

Training algorithm The class `TrainingAlgorithm` is abstract, because it does not represent a training algorithm for an objective function of a multilayer perceptron.

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

Figure A.3 shows the UML class diagram for the multilayer perceptron with some of the derived classes included.

A.5 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 [50]. 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 [47].

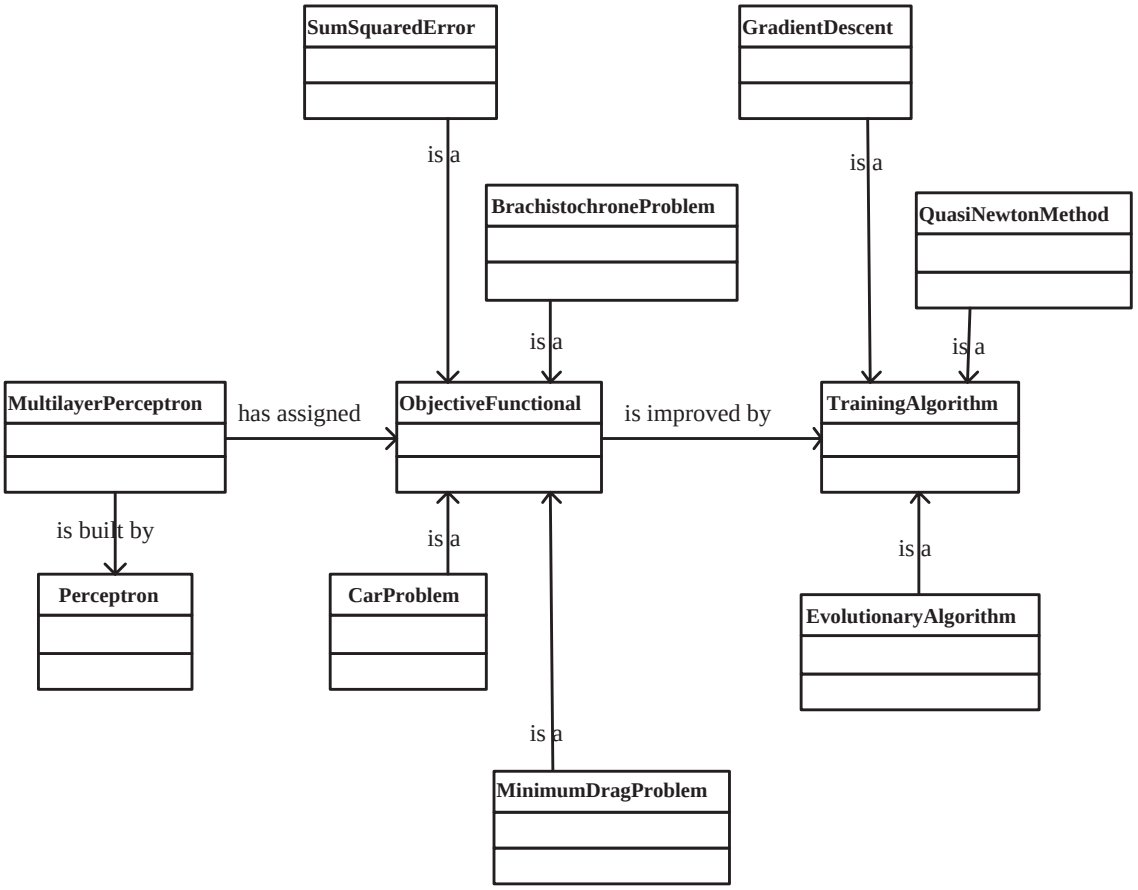


Figure A.3: Aggregation of derived classes to the association diagram.

Perceptron

A perceptron neuron model has the following attributes:
- A number of neuron inputs.
- The activation function.
- A bias.
- A set of synaptic weights.

It performs the following main operations:
- Calculate the output for a given input.

Multilayer perceptron

A multilayer perceptron has the following attributes:
- The sizes of the network architecture - One or several hidden layers of perceptrons.
- An output layer of perceptrons.

It performs the following main operations:- Calculate the output for a given input.

Objective functional

An objective functional for a multilayer perceptron has the following attributes:
- A relationship to a multilayer perceptron. In C++ this is implemented as a pointer to a multilayer perceptron object.

It performs the following operations:
- Calculate the evaluation of a multilayer perceptron.
- Calculate the objective function gradient vector of a multilayer perceptron.

Training algorithm

A training algorithm for a multilayer perceptron has the following attributes:
- A relationship to an objective functional for a multilayer perceptron. In C++ this is implemented as a pointer to an objective functional object.
- A set of training operators.
- A set of training parameters.
- A set of stopping criteria.

It performs the following operations:
- Train a multilayer perceptron.

Appendix B

Unit testing

B.1 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).

B.2 Related code

There exist several available frameworks for incorporating test cases in C++ code, such as CppUnit or Cpp test. However, for portability reasons, Flood 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 Flood

`Flood` 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,

```
int a = 0;
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 Flood 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 `Flood` can be found in the folder AllTests.

Appendix C

Numerical integration

Evaluation of the objective functional of a neural network often requires to integrate functions, ordinary differential equations and partial differential equations. In general, approximate approaches need to be applied here.

Some standard methods for numerical integration are described in this Appendix. They are meant to be utilities which can be embedded into a specific problem when necessary.

C.1 Integration of functions

Introduction

Numerical integration is the approximate computation of an integral using numerical techniques. More specifically, the problem is to compute the definite integral of a given real function $f(x)$ over a closed interval $[a, b]$,

$$I[y(x)] = \int_a^b f(x)dx. \tag{C.1}$$

There are a wide range of methods available for numerical integration [45]. The numerical computation of an integral is sometimes called quadrature.

Closed Newton-Cotes formulas

The Newton-Cotes formulas are an extremely useful and straightforward family of numerical integration techniques. The integration formulas of Newton and Cotes are obtained dividing the interval $[a, b]$ into n equal parts such that $f_n = f(x_n)$ and $h = (b-a)/n$. Then the integrand $f(x)$ is replaced by a suitable interpolating polynomial $P(x)$, so that

$$\int_a^b f(x) \sim \int_a^b P(x). \tag{C.2}$$

To find the fitting polynomials, the Newton-Cotes formulas use Lagrange interpolating polynomials [49]. Next we examine some rules of this kind.

The trapezoidal rule

The 2-point closed Newton-Cotes formula is called the trapezoidal rule, because it approximates the area under a curve by a trapezoid with horizontal base and sloped top (connecting the endpoints a and b). In particular, let call the lower and upper integration limits x_0 and x_1 respectively, the integration interval h, and denote $f_n = f(x_n)$. Then the trapezoidal rule states that [54]

$$\begin{aligned} \int_{x_1}^{x_2} f(x)dx &= h\left[\frac{1}{2}f_1 + \frac{1}{2}f_2\right] \\ &+ \mathcal{O}(h^3 f''), \end{aligned} \tag{C.2}$$

were the error term $O(\cdot)$ means that the true answer differs from the estimate by an amount that is the product of some numerical coefficient times h^3 times the value of the functions second derivative somewhere in the interval of integration.

Simpson's rule

The 3-point closed Newton-Cotes formula is called Simpson's rule. It approximates the integral of a function using a quadratic polynomial. In particular, let the function f be tabulated at points x_0, x_1, and x_2, equally spaced by distance h, and denote $f_n = f(x_n)$. Then Simpson's rule states that [54]

$$\begin{aligned} \int_{x_1}^{x_3} f(x)dx &= h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{1}{3}f_3\right] \\ &+ \mathcal{O}(h^5 f^{(4)}). \end{aligned} \tag{C.2}$$

Here $f^{(4)}$ means the fourth derivative of the function f evaluated at an unknown place in the interval. Note also that the formula gives the integral over an interval of size $2h$, so the coefficients add up to 2.

Extended Newton-Cotes formulas

The Newton-Cotes formulas are usually not applied to the entire interval of integration $[a, b]$, but are instead used in each one of a collection of subintervals into which the interval $[a, b]$ has been divided. The full integral is the approximated by

the sum of the approximations to the subintegrals. The locally used integration rule is said to have been extended, giving rise to a composite rule [49]. We proceed to examine some composite rules of this kind.

Extended trapezoidal rule

For n tabulated points, using the trapezoidal rule $n-1$ times and adding the results gives [54]

$$\begin{aligned}\int_{x_1}^{x_n} f(x)dx &= h\left[\frac{1}{2}f_1 + f_2 + f_{n-1} + \frac{1}{2}f_n\right] \\ &+ \mathcal{O}\left(\frac{(b-a)^3 f''}{N^2}\right). \end{aligned} \tag{C.2}$$

Note that the error estimate is here written in terms of the interval $b-a$ and the number of points N instead of in terms of h.

Extended Simpson's rule

For and odd number n of tabulated points, the extended Simpson's rule is [54]

$$\begin{aligned}\int_{x_1}^{x_n} f(x)dx &= h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{2}{3}f_3 + \frac{4}{3}f_4 + \ldots + \frac{2}{3}f_{n-2} + \frac{4}{3}f_{n-1} + \frac{1}{3}f_n\right] \\ &+ \mathcal{O}\left(\frac{1}{N^4}\right). \end{aligned} \tag{C.2}$$

Ordinary differential equation approach

The evaluation of the integral (C.1) is equivalent to solving for the value $I \equiv y(b)$ the ordinary differential equation

$$\frac{dy}{dx} = f(x), \tag{C.3}$$

$$y(a) = 0. \tag{C.4}$$

Section C.2 of this report deals with the numerical integration of differential equations. In that section, much emphasis is given to the concept of 'variable' or 'adaptive' choices of stepsize.

C.2 Ordinary differential equations

Introduction

An ordinary differential equation (ODE) is an equality involving a function and its derivatives. An ODE of order n is an equation of the form

$$F(x, y(x), y'(x), \ldots, y^{(n)}(x)) = 0, \tag{C.5}$$

where y is a function of x, $y' = dy/dx$ is the first derivative of y with respect to x, and $y^{(n)} = dy^n/dx^n$ is the n-th derivative of y with respect to x.

The generic problem of a n-th order ODE can be reduced to the study of a set of n coupled first-order differential equations for the functions y_i, $i = 1, \ldots, n$, having the general form

$$\frac{dy_i(x)}{dx} = f_i(x, y_1, \ldots, y_N), \tag{C.6}$$

for $i = 1, \ldots, n$ and where the functions f_i on the right-hand side are known.

While there are many general techniques for analytically solving different classes of ODEs, the only practical solution technique for complicated equations is to use numerical methods. The most popular of these are the Runge-Kutta and the Runge-Kutta-Fehlberg methods.

A problem involving ODEs is not completely specified by its equations. In initial value problems all the y_i are given at some starting value x's, and it is desired to find the y_i's at some final point x_f, or at some discrete list of points (for example, at tabulated intervals).

The Euler method

The formula for the Euler method is

$$y_{n+1} = y_n + hf(x_n, y_n) + \mathcal{O}(h^2), \tag{C.7}$$

which advances a solution from x_n to $x_{n+1} = x_n + h$. The formula is unsymmetrical: It advances the solution through an interval h, but uses derivative information only at the beginning of that interval.

There are several reasons that Eulers method is not recommended for practical use, among them, (i) the method is not very accurate when compared to other methods run at the equivalent stepsize, and (ii) neither is it very stable [45].

The Runge-Kutta method

Consider the use of a step like (C.7) to take a 'trial' step to the midpoint of the interval. The use the value of both x and y at that midpoint to compute the 'real' step across the whole interval. This can be written

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1) \\ y_{n+1} &= y_n + k_2 + \mathcal{O}(h^3) \end{aligned} \tag{C.6}$$

As indicated in the error term, this symmetrization cancels out the first-order error term, making the method second order. A method is conventionally called nth order if its error term is $\mathcal{O}(h^{n+1})$- In fact Equation (C.6) is called the second-order Runge-Kutta or midpoint method.

There are various specific formulas that derive from this basic idea. The most often used is the classical fourth-order Runge-Kutta formula,

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \\ k_3 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \\ k_4 &= hf(x_n + h, y_n + k_3) \\ y_{n+1} &= y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + \mathcal{O}(h^5) \end{aligned} \tag{C.3}$$

The fourth order Runge-Kutta method requires four evaluations of the right-hand side per step h.

The Runge-Kutta-Fehlberg method

Implementation of adaptive stepsize control requires that the stepping algorithm signal information about its performance, most important, an estimate of its truncation error.

It is this difference that we shall endeavor to keep to a desired degree of accuracy, neither too large nor too small. We do this by adjusting h.

An step size adjustment algorithm is based on the embedded Runge-Kuta formulas, originally invented by Fehlberg. The general form of a fifth-order Runge-Kutta formula is

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \\ &\cdots \\ k_3 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \\ k_4 &= hf(x_n + h, y_n + k_3) \\ y_{n+1} &= y_n + c_1k_1 + c_2k_2 + c_3k_3 + c_4k_4 + c_5k_5 + c_6k_6 + \mathcal{O}(h^6) \end{aligned} \tag{C.-1}$$

The embedded fourth order formula is

$$y^*_{n+1} = y_n + c^*_1k_1 + c^*_2k_2 + c^*_3k_3 + c^*_4k_4 + c^*_5k_5 + c^*_6k_6 + \mathcal{O}(h^5) \tag{C.0}$$

And so the error estimate is

$$\begin{aligned} \Delta &\equiv y_{n+1} - y^*_{n+1} \\ &= \sum_{i=1}^{6} (c_i - c^*_i) k_i. \end{aligned} \tag{C.0}$$

The particular values of the various constants that we favor are those found by Cash and Karp [11], and given in Table C.1. These give a more efficient method than Fehlberg's original values [45].

i	a_i	b_{i1}	b_{i2}	b_{i3}	b_{i4}	b_{i5}	c_i	c^*_i
1							$\frac{37}{378}$	$\frac{2825}{27648}$
2	$\frac{1}{5}$	$\frac{1}{5}$					0	0
3	$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$				$\frac{250}{621}$	$\frac{18575}{48384}$
4	$\frac{3}{5}$	$\frac{3}{10}$	$-\frac{9}{10}$	$\frac{6}{5}$			$\frac{125}{594}$	$\frac{13525}{55296}$
5	1	$-\frac{11}{54}$	$\frac{5}{2}$	$-\frac{70}{27}$	$\frac{35}{27}$		0	$\frac{277}{14336}$
6	$\frac{7}{8}$	$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$	$\frac{512}{1771}$	$\frac{1}{4}$

Table C.1: Cash and Karp parameters for the Runge-Kutta-Fehlberg method.

Now that we know, approximately, what the error is, we need to consider how to keep it within desired bounds. What is the relation between Δ and h? According to Equations (C.-1) and (C.0), Δ scales as h^5. If we take a step h_1 and produce an error Δ_1, therefore, the step h_0 that would have given some other value Δ_0 is readily estimated as

$$h_0 = h_1 \left| \frac{\Delta_0}{\Delta_1} \right|^{0.2} \tag{C.1}$$

Henceforth we will let Δ_0 denote the desired accuracy. Then, Equation (C.1) is used in two ways: If Δ_1 is larger than Δ_0 in magnitude, the equation tells how much to decrease the step when we retry the present (failed) step. If Δ_1 is smaller than Δ_0, on the other hand, then the equation tells how much we can safely increase the stepsize for the next step.

This notation hides the fact that Δ_0 is actually a vector of desired accuracies, one for each equation in the set of ODEs. In general, the accuracy requirement will be that all equations are within their respective allowed errors.

$$h_0 = \begin{cases} S h_1 \left| \frac{\Delta_0}{\Delta_1} \right|^{0.20} & \Delta_0 \geq \Delta_1 \\ S h_1 \left| \frac{\Delta_0}{\Delta_1} \right|^{0.25} & \Delta_0 < \Delta_1 \end{cases} \tag{C.1}$$

C.3 Partial differential equations

Introduction

Partial differential equations arise in all fields of science and engineering, since most real physical processes are governed by them. A partial differential equation is an equation stating a relationship between a function of two or more independent variables and the partial derivatives of this function with respect to the independent variables. In most problems, the independent variables are either space (x, y, z) or space and time (x, y, z, t). The dependent variable depends on the physical problem being modeled.

Partial differential equations are usually classified into the three categories, hyperbolic, parabolic, and elliptic, on the basis of their characteristics [45].

The prototypical example of a hyperbolic equation is the wave equation

$$\frac{\partial^2 u(\mathbf{x}, t)}{\partial t^2} = v^2 \nabla^2 u(\mathbf{x}, t), \tag{C.2}$$

where v is the velocity of wave propagation.

The prototypical parabolic equation is the diffusion equation

$$\frac{\partial u(\mathbf{x}, t)}{\partial t} = \nabla(D \nabla u(\mathbf{x}, t)), \tag{C.3}$$

where D is the diffusion coefficient.

The prototypical elliptic equation is the Poisson equation

$$\nabla^2 u(\mathbf{x}) = \rho(\mathbf{x}), \tag{C.4}$$

where the source term ρ is given. If the source term is equal to zero, the equation is Laplace's equation.

From a computational point of view, the classification into these three canonical types is not as important as some other essential distinctions [45]. Equations (C.2) and (C.3) both define initial value (or Cauchy) problems. By contrast, Equation (C.4) defines a boundary value problem.

In an initial value problem information of u is given at some initial time t_0 for all $\mathbf{x}$. The PDE then describes how $u(\mathbf{x}, t)$ propagates itself forward in time [45].

On the other hand, boundary value problems direct to find a single static function u which satisfies the equation within some region of interest $\mathbf{x}$, and which has some desired behavior on the boundary of that region [45].

In a very few special cases, the solution of a PDE can be expressed in closed form. In the majority of problems in engineering and science, the solution must be obtained by numerical methods.

The finite differences method

In the finite differences approach, all the derivatives in a differential equation are replaced by algebraic finite difference approximations, which changes the differential equation into an algebraic equation that can be solved by simple arithmetic [25].

The error between the approximate solution and the true solution here is determined by the error that is made by going from a differential operator to a difference operator. This error is called the discretization error or truncation error [40]. The term truncation error reflects the fact that a difference operator can be viewed as a finite part of the infinite Taylor series of the differential operator.

The finite element method

Another approach for solving differential equations is based on approximating the exact solution by an approximate solution, which is a linear combination of specific trial functions, which are typically polynomials. These trial functions are linearly independent functions that satisfy the boundary conditions. The unknown coefficients in the trial functions are then determined by solving a system of linear algebraic equations [25].

Because finite element methods can be adapted to problems of great complexity and unusual geometries, they are an extremely powerful tool in the solution of important problems in science and engineering. It is out of the scope of this work to provide a detailed explanation of the finite element method. The interested reader is referred to [57].

Bibliography

[1] R. Aris. *Elementary Chemical Reactor Analysis*. Butterworths, 1989.

[2] T. Bäck and F. Hoffmeister. Extended selection mechanisms in genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms, San Mateo, California, USA*, pages 92–99, 1991.

[3] J.E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms, Hillsdale, New Jersey, USA*, pages 14–21, 1987.

[4] E. Balsa-Canto. *Algoritmos Eficientes para la Optimizacion Dinamica de Procesos Distribuidos*. PhD thesis, Universidad de Vigo, 2001.

[5] R. Battiti. First and second order methods for learning: Between steepest descent and newton's method. *Neural Computation*, 4(2):141–166, 1992.

[6] L. Belanche. *Heterogeneous Neural Networks*. PhD thesis, Technical University of Catalonia, 2000.

[7] J.T. Betts. A survey of numerical methods for trajectory optimization. *AIAA Journal of Guidance, Control and Dynamics*, 21(2):193–207, 1998.

[8] C. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[9] D.G. Brown S.L., Hull. Axisymmetric bodies of minimum drag in hypersonic flow. *Journal of Optimization Theory and Applications*, 3(1):52–71, 1969.

[10] D. Bucur and G. Buttazzo. *Variational Methods in Shape Optimization Problems*. Birkhauser, 2005.

[11] J.R. Cash and A.H. Karp. A variable order runge-kutta method for initial value problems with rapidly varying right hand sides. *ACM Transactions on Mathematical Software*, 16(3):201–222, 1990.

[12] C.T. Chen and C. Hwang. Optimal control computation for differential-algebraic process systems with general constraints. *Chemical Engineering Communications*, 97:9–26, 1990.

[13] Z. Chen and S. Haykin. On different facets of regularization theory. *Neural Computation*, 14(12):2791–2846, 2002.

[14] P. Davand. Kratos: An object-oriented environment for development of multi-physics analysis software. www.cimne.upc.edu/kratos, 2010.

[15] H. Demuth, M. Beale, and M. Hagan. *Neural Network Toolbox User's Gide.* The MathWorks, Inc., 2009.

[16] B. Eckel. *Thinking in C++. Second Edition.* Prentice Hall, 2000.

[17] H.W. Engl, M. Hanke, and A. Neubauer. *Regularization of Inverse Problems.* Springer, 2000.

[18] 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.

[19] R. Fletcher and C.M. Reeves. Function minimization by conjugate gradients. *Computer Journal*, 7:149–154, 1964.

[20] D.B. Fogel. An introduction to simulated evolutionary optimization. *IEEE Transactions on Neural Networks*, 5(1):3–14, 1994.

[21] 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.

[22] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison Wesley, 1988.

[23] M.T. Hagan and M. Menhaj. Training feedforward networks with the marquardt algorithm. *IEEE Transactions on Neural Networks*, 5(6):989–993, 1994.

[24] S. Haykin. *Neural Networks: A Comprehensive Foundation.* Prentice Hall, 1994.

[25] J.D. Hoffman. *Numerical Methods for Engineers and Scientists.* CRC Press, second edition, 2001.

[26] J. Hong. Optimal substrate feeding policy for a fed batch fermentation with substrate and product inhibition kinetics. *Biotechnology and Bioengineering*, 28:1421–1431, 1986.

[27] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[28] J. Kennedy and R.C. Eberhart. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.

[29] D.E. Kirk. *Optimal Control Theory. An Introduction.* Prentice Hall, 1970.

[30] A. Kirsch. *An Introduction to the Mathematical Theory of Inverse Problems.* Springer, 1996.

[31] R. Lopez. Flood: An open source neural networks c++ library. www.cimne.com/flood, 2010.

[32] R. Lopez, E. Balsa-Canto, and E. Oñate. Neural networks for variational problems in engineering. *International Journal for Numerical Methods in Engineering*, In press, 2008.

[33] R. Lopez, X. Diego, R. Flores, M. Chiumenti, and E. Oñate. Artificial neural networks for the solution of optimal shape design problems. In *Proceedings of the 8th World Congress on Computational Mechanics WCCM8*, 2008.

[34] 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.

[35] D.G. Luenberger. *Linear and Nonlinear Programming.* Addison Wesley, 1984.

[36] R. Luus. Application of dynamic programming to differential-algebraic process systems. *Computers and Chemical Engineering*, 17(4):373–377, 1993.

[37] D.J.C. MacKay. Bayesian interpolation. *Neural Computation*, 4(3):415–447, 1992.

[38] B. Mohammadi and O. Pironneau. Shape optimization in fluid mechanics. *Annual Review of Fluid Mechanics*, 36:255–279, 2004.

[39] M.F. Moller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*, 6:525–533, 1993.

[40] K.W. Morton and D.F. Mayers. *Numerical Solution of Partial Differential Equations, an Introduction.* Cambridge University Press, 2005.

[41] Hettich S. Blake C.L. Newman, D.J. and C.J. Merz. Uci repository of machine learning databases. http://www.ics.uci.edu/-mlearn/MLRepository.html, 1998.

[42] I Ortigosa, R. Lopez, and J. Garcia. A neural networks approach to residuary resistance of sailing yachts prediction. In *Proceedings of the International Conference on Marine Engineering MARINE 2007*, 2007.

[43] H. Pohlheim. Geatbx - genetic and evolutionary algorithm toolbox for use with matlab. http://www.geatbx.com, 2007.

[44] M.J.D. Powell. Restart procedures for the conjugate gradient method. *Mathematical Programming*, 12:241–254, 1977.

[45] 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.

[46] A.G. Ramm. *Inverse Problems. Mathematical and Analytical Techniques with Applications to Engineering.* Springer, 2005.

[47] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison Wesley, 1999.

[48] P.C. Sabatier. Past and future of inverse problems. *Journal of Mathematical Physics*, 41:4082–4124, 2000.

[49] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis.* Springer-Verlag, 1980.

[50] B. Stroustrup. *The C++ Programing Language.* Addison Wesley, 2000.

[51] A.N. Tikhonov and V.Y. Arsenin. *Solution of ill-posed problems.* Wiley, 1977.

[52] V. Vapnik. *Statistical Learning Theory.* John Wiley and Sons, 1998.

[53] J. Šíma and P. Orponen. General-purpose computation with neural networks: A survey of complexity theoretic results. *Neural Computation*, 15:2727–2778, 2003.

[54] E. W. Weisstein. Mathworld - a wolfram web resource. http://mathworld.wolfram.com, 2010.

[55] D. Whitley. The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In *Proceedings of the Third International Conference on Genetic Algorithms, San Mateo, California, USA*, pages 116–121, 1989.

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

[57] O. Zienkiewicz and R. Taylor. *The Finite Element Method, Volumes 1 and 2.* Mc Graw Hill, 1988.

Index