Flood 2

An Open Source Neural Networks C++ Library.

www.cimne.com/flood

User's Guide.

Roberto Lopez
International Center for Numerical Methods in Engineering (CIMNE)
Technical University of Catalonia (UPC)
Barcelona, Spain
E-mail: rlopez@cimne.upc.edu

December 2008

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, inverse problems or optimal shape design. All these problems can be formulated as variational problems.

`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.

Contents

Chapter 1

Preliminaries

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 [17]. For the Flood namespace the following sentence can be written:

```
using namespace Flood;
```

1.2 The Vector and Matrix classes in `Flood`

The `Flood` library includes its own Vector and Matrix container classes. The Vector and Matrix classes are templates, which means that they can be applied to different types [17]. That is, we can create a Vector or **int** numbers, a Matrix of MyClass objects, etc.

For example, in order to construct an empty Vector of **int** numbers we use

```
Vector<int> vector1;
```

The following sentence constructs a Vector of 3 **int** numbers and sets the 0, 1 and 2 elements of the Vector to 1, 2 and 3, respectively. Note that indexing goes from 0 to $n-1$, where n is the Vector size.

```
Vector<int> vector2(3);
vector2[0] = 1;
vector2[1] = 2;
vector2[2] = 3;
```

If we want to construct Vector of 5 **int** numbers and initialize all the elements to 0, we can use

```
Vector<int> vector3(5, 0);
```

The following sentence copies vector3 into vector1.

```
vector1 = vector3;
```

The method getSize(**void**) returns the size of a Vector.

```
int sizeOfVector1 = vector1.getSize();
```

In order to construct an empty Matrix of MyClass objects with 2 files and 3 columns we use

```
Matrix<MyClass> matrix(2,3);
```

The methods getNumberOfRows(**void**) and getNumberOfColumns(**void**) return the numbers of rows and columns in a Matrix, respectively.

```
int numberOfRows = matrix.getNumberOfRows();
int numberOfColumns = matrix.getNumberOfColumns();
```

In order to put all these ideas together, file VectorAndMatrixApplication.cpp lists the source code of a sample application (main function) which makes use of the Vector and Matrix container classes. To execute this program on Linux run the VectorAndMatrixMakefile makefile. To execute it on Windows just use that application file in the compiler's project.

Chapter 2

Introduction

There are many different types of neural networks. The multilayer perceptron is an important one, and most of the literature in the field is referred to this neural network. In this Chapter, the learning problem in the multilayer perceptron is formulated from a variational point of view. In this way, learning tasks lie in terms of finding a function which causes some functional to assume an extreme value. As we shall see, the multilayer perceptron provides a general framework for solving variational problems.

The multilayer perceptron is characterized by a neuron model, a network architecture and associated objective functionals and training algorithms. This four concepts are briefly described next.

1. *Neuron model*. 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 a set of numerical input signals. This information is then integrated with a set of free parameters to produce a message in the form of a single numerical output signal.

2. *Network architecture*. In the same way a biological nervous system is composed of interconnected biological neurons, an artificial neural network is built up by organizing artificial 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.

3. *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.

4. *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 free parameters in the neural network takes place.

Figure 2.1 depicts an activity diagram for the learning problem in the multilayer perceptron. The solving approach here consists of three steps. The first step is to choose a suitable parameterized function space in which the solution to the problem is to be approximated. The elements of this family of functions are those spanned by a multilayer perceptron. In the second step the variational problem is formulated by selecting an appropriate objective functional, defined on the function space chosen before. The third step is to solve the reduced function optimization problem. This is performed with a training algorithm capable of finding an optimal set of parameters.

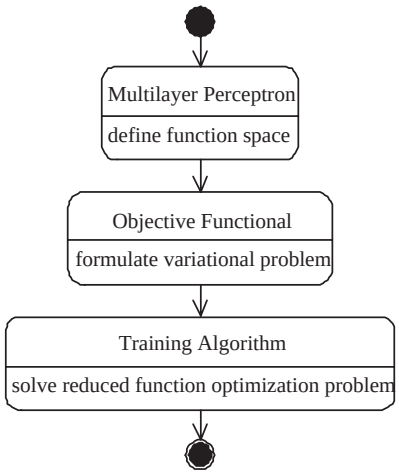

Figure 2.1: Activity diagram for the learning problem in the multilayer perceptron.

Chapter 3

The perceptron neuron model

As we have said, a neuron model is the basic information processing unit in a neural network. The perceptron is the characteristic neuron model in the multilayer perceptron. Following current practice [64], 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.

3.1 The elements of the perceptron

The block diagram in Figure 3.1 is a graphical representation of the perceptron. Here we identify three basic elements, which transform the input signals $(x_1, \ldots, x_n)$ in a single output signal y [6]:

- A set of neuron parameters $\underline{\alpha}$, which consists of a bias b and a vector of synaptic weights $(w_1, \ldots, w_n)$.
- A combination function h, which combines the input signals and the neuron parameters to produce a single net input signal u.
- An activation function or transfer function g, which takes as argument the net input signal and produces the output signal.

Next we describe in more detail the three basic elements of this neuron model [27]:

Neuron parameters

The neuron parameters allows a neuron model to be trained to perform a task. In the perceptron, the set of neuron parameters is:

$$\underline{\alpha} = (b, \mathbf{w}) \in \mathbb{R} \times \mathbb{R}^n \tag{3.1}$$

where b is called the bias and $\mathbf{w} = (w_1, \ldots, w_n)$ is called the synaptic weight vector. Note then that the number of neuron parameters of this neuron model is $1 + n$, where n is the number of inputs in the neuron.

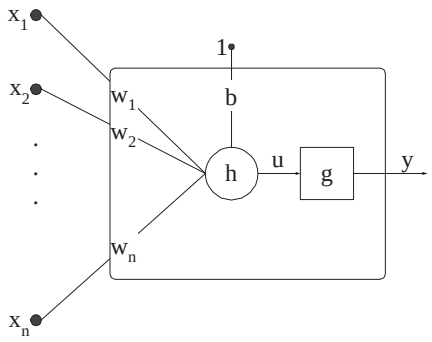

Figure 3.1: Perceptron neuron model.

Combination function

In the perceptron, the combination function h computes the inner product of the input vector $\mathbf{x} = (x_1, \ldots, x_n)$ and the synaptic weight vector $\mathbf{w} = (w_1, \ldots, w_n)$ to produce a net input signal u. This model also includes a bias externally applied, denoted by b, which increases or reduces the net input signal to the activation function, depending on whether it is positive or negative, respectively. The bias is often represented as a synaptic weight connected to an input fixed to $+1$,

$$h(\mathbf{x}; b, \mathbf{w}) \quad = \quad b + \sum_{i=1}^{n} w_i x_i. \tag{3.2}$$

Activation function

The activation function g defines the output signal y from the neuron in terms of its net input signal u. In practice we can consider many useful activation functions [16]. Some of the most used activation functions are the threshold function, the logistic function, the hyperbolic tangent or the linear function [27].

Threshold function

For this activation function, represented in Figure 3.2 we have

$$g(u) = \begin{cases} -1 & u < 0 \\ 1 & u \geq 0 \end{cases} \tag{3.3}$$

That is, the threshold activation function limits the output of the neuron to -1 if the net input u is less than 0; or 1 if u is equal to or greater than 0.

There might be some cases when we need to compute the activation derivative of the neuron,

$$g'(u) \equiv \frac{dg(u)}{du}. \tag{3.4}$$

or its second derivative,

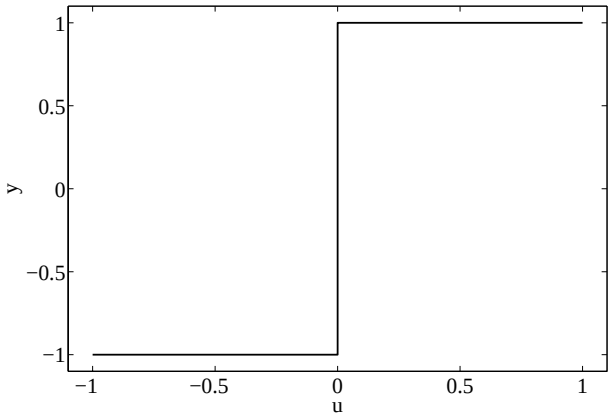

Figure 3.2: A threshold activation function.

$$g''(u) \equiv \frac{d^2 g(u)}{du^2}. \tag{3.5}$$

The problem with the threshold activation function is that it is not differentiable at the point $u = 0$.

Logistic function

The logistic function has a sigmoid shape. It is widely used when constructing neural networks. The logistic function is a monotonous crescent function which exhibits a good balance between a linear and a non-linear behavior. It is defined by

$$g(u) = \frac{1}{1 + \exp(-u)}. \tag{3.6}$$

The logistic function is represented in Figure 3.3.

For this sigmoid function, the activation derivative is given by

$$g'(u) = \frac{\exp(u)}{(1 + \exp(u))^2} \tag{3.7}$$

and the second derivative of the activation is given by

$$g''(u) = -\exp(u)\frac{\exp(u) - 1}{(\exp(u + 1))^3} \tag{3.8}$$

Hyperbolic tangent

The hyperbolic tangent is also a sigmoid function widely used when constructing neural networks. It is very similar to the logistic function. The main difference is that the hyperbolic tangent ranges in the interval $(-1, 1)$, while the logistic function ranges in the interval $(0, 1)$. The hyperbolic tangent is defined by

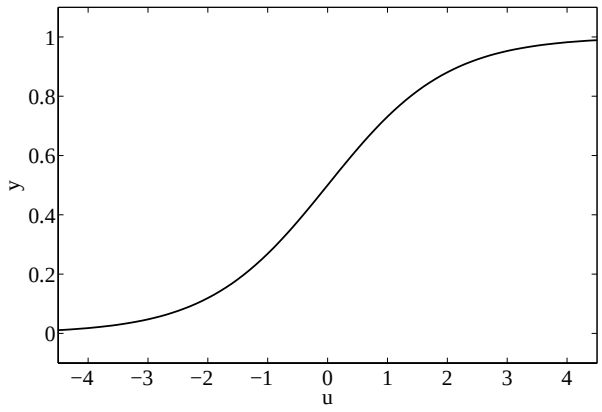

Figure 3.3: A logistic activation function.

$$g(u) = \tanh(u). \tag{3.9}$$

The hyperbolic tangent function is represented in Figure 3.4.

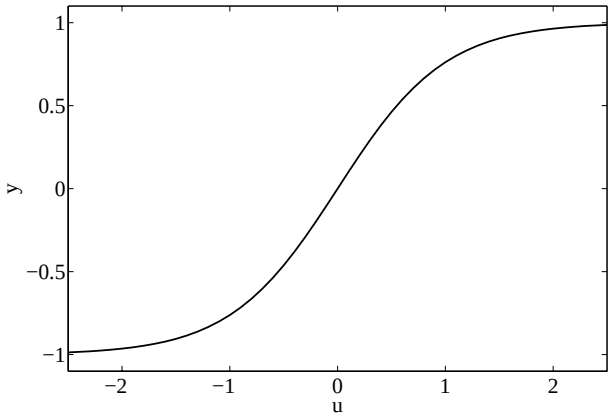

Figure 3.4: A hyperbolic tangent activation function.

For this sigmoid function, the activation derivative is given by

$$g'(u) = 1 - \tanh^2(u), \tag{3.10}$$

and the second derivative of the activation is given by

$$g''(u) = -2\tanh(u)(1 - \tanh^2(u)). \tag{3.11}$$

Linear function

For the linear activation function, described in Figure 3.5 we have

$$g(u) = u. \tag{3.12}$$

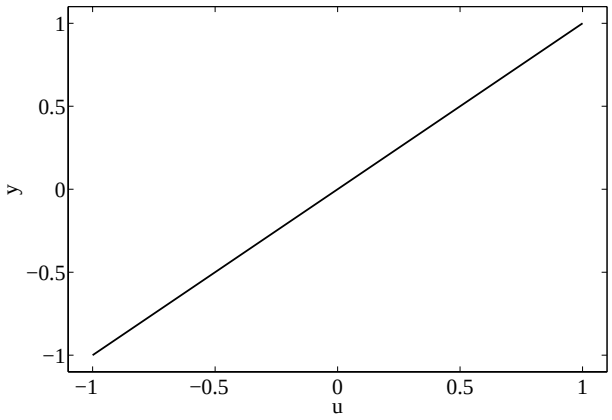

Figure 3.5: A linear activation function.

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

For the linear function, the activation derivative is given by

$$g'(u) = 1, \tag{3.13}$$

and the second derivative of the activation is given by

$$g''(u) = 0. \tag{3.14}$$

3.2 The perceptron function space

Mathematically, a perceptron neuron model may be viewed as a parameterized function space V from an input $X \subset \mathbb{R}^n$ to an output $Y \subset \mathbb{R}$. Elements of V are parameterized by the bias and the vector of synaptic weights of the neuron. In this way the dimension of V is $n + 1$.

We can write down an expression for the elements of the function space which a perceptron can define [8]. The net input to the neuron is obtained by first forming a linear combination of the input signals and the synaptic weights, and then adding the bias, to give

$$u \quad = \quad b + \sum_{i=1}^{n} w_i x_i. \tag{3.15}$$

The output of the neuron is obtained transforming the linear combination in Equation (3.15) with an activation function g to give

$$y(\mathbf{x}; b, \mathbf{w}) = g\left(b + \sum_{i=1}^{n} w_i x_i\right). \tag{3.16}$$

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. The concepts of linear and sigmoid perceptron are implemented within the classes LinearPerceptron and SigmoidPerceptron of the C++ library Flood [37].

Although a single perceptron can solve some simple learning tasks, the power of neural computation comes from connecting many neurons in a network architecture [64].

3.3 The Perceptron class in Flood

`Flood` includes the class Perceptron to represent the concept of perceptron neuron model. That class contain the following members:

- The activation function.
- The number of input signals.
- The neuron's bias.
- The neuron's synaptic weights.

For example, to construct a sigmoid perceptron object with 3 inputs we use the following sentence

```
Perceptron perceptron(3);
```

The perceptron objects we have constructed so far have bias and synaptic weights values chosen at random from a normal distribution with mean 0 and standard deviation 1.

The default activation function is the hyperbolic tangent. In order to change that the setActivationFunction method is used:

```
perceptron.setActivationFunction(Perceptron::Linear);
```

In order to calculate the output signal from a neuron for a Vector of input signals we use the calculateOutputSignal(Vector<**double**>) method. For the perceptron object constructed above

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

inputSignal[0] = 0.0;

inputSignal[1] = 1.0;

inputSignal[2] = 2.0;

double outputSignal = perceptron.calculateOutputSignal(inputSignal);
```

In order to calculate the activation first and second derivatives of a neuron for a Vector of input signals we use the calculateOutputSignalDerivative(Vector<**double**>) and the calculateOutputSignalSecondDerivative(Vector<**double**>) methods, respectively.

```
double outputSignalDerivative =
perceptron.calculateOutputSignalDerivative(inputSignal);

double outputSignalSecondDerivative =
perceptron.calculateOutputSignalSecondDerivative(inputSignal);
```

File PerceptronApplication.cpp is an example application of the Perceptron class in `Flood`.

Chapter 4

The multilayer perceptron 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 free 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 [64]. 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.

4.1 Feed-forward architectures

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 $c+1$ layers $L^{(1)}, \ldots, L^{(c+1)}$, so that neurons in any layer are connected only to neurons in the next layer. The input layer $L^{(0)}$ consists of n external inputs and is not counted as a layer of neurons; the hidden layers $L^{(1)}, \ldots, L^{(c)}$ contain $h_1, \ldots, h_c$ hidden neurons, respectively; and the output layer $L^{(c+1)}$ is composed of m output neurons. 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 [64].

Figure 4.1 shows a feed-forward architecture, with n inputs, c hidden layers with h_i neurons, for $i = 1, \ldots, c$, and m neurons in the output layer.

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].

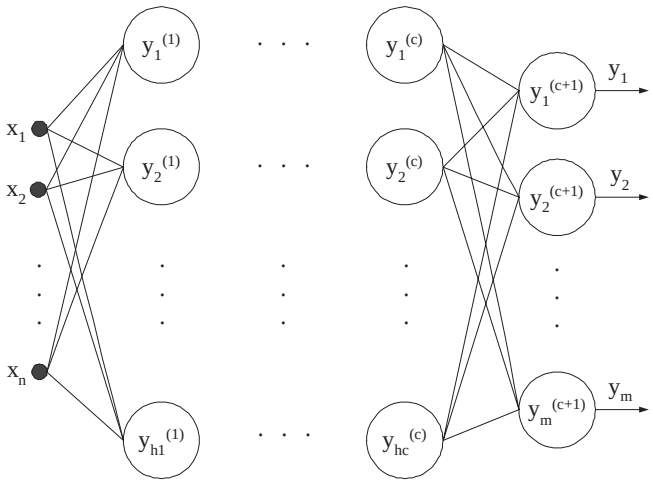

Figure 4.1: The feed-forward network architecture.

4.2 The multilayer perceptron function space

In Section 3 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 parameterized by the biasses and synaptic weights in the neural network, which can be grouped together in a s-dimensional vector $\underline{\alpha} = (\alpha_1, \ldots, \alpha_s)$. The dimension of the function space V is therefore s.

Figure 4.2 shows a multilayer perceptron, with n inputs, one hidden layer with h_1 neurons and m neurons in the output layer. Biases in the hidden layer are represented as synaptic weights from an extra input with a fixed value of $x_0 = 1$. Similarly, biases in the output layer are represented as synaptic weights from an extra hidden neuron, with a fixed activation also fixed to $y_0^{(1)} = 1$.

We can write down the analytical expression for the elements of the function space which the multilayer perceptron shown in Figure 4.2 can define [8]. Indeed, the net input to the hidden neuron j is obtained first by forming a linear combination of the n input signals, and adding the bias, to give

$$\begin{aligned} u_j^{(1)} &= b_j^{(1)} + \sum_{i=1}^{n} w_{ji}^{(1)} x_i \\ &= \sum_{i=0}^{n} \alpha_{ji}^{(1)} x_i, \end{aligned} \tag{4.1}$$

for $j = 1, \ldots, h_1$. $\alpha_{j0}^{(1)}$ denotes the bias of neuron j in layer (1), for which $x_0 = 1$; $\alpha_{ji}^{(1)}$, for $i = 1, \ldots, n$, denotes the synaptic weight of neuron j in layer (1) which comes from input i.

The output of hidden neuron j is obtained transforming the linear combination in Equation (4.1) with an activation function $g^{(1)}$ to give

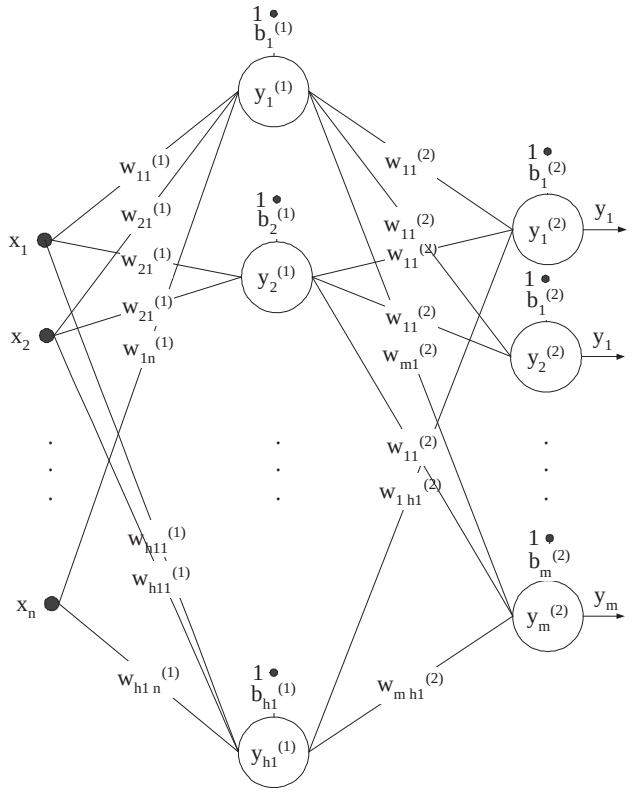

Figure 4.2: A two-layer perceptron.

$$y_j^{(1)} = g^{(1)}\left(u_j^{(1)}\right), \tag{4.2}$$

for $j = 1, \ldots, h_1$. The neural network outputs are obtained transforming the output signals of the neurons in the hidden layer by the neurons in the output layer. Thus, the net input for each output neuron k is obtained forming a linear combination of the output signals from the hidden layer neurons of the form

$$\begin{aligned} u_k^{(2)} &= b_k^{(2)} + \sum_{j=1}^{h_1} w_{kj}^{(2)} y_j^{(1)} \\ &= \sum_{j=0}^{h_1} \alpha_{kj}^{(2)} y_j^{(1)}, \end{aligned} \tag{4.3}$$

for $k = 1, \ldots, m$. The value $\alpha_{k0}^{(2)}$ denotes the bias of neuron k in layer (2), for which $y_0^{(1)} = 1$; similarly, the value $\alpha_{kj}^{(2)}$, $j = 1, \ldots, h_1$, denotes the synaptic weight of neuron k in layer (2) which comes from input j.

The activation of the output neuron k is obtained transforming the linear combination in Equation (4.3) with an activation function $g^{(2)}$ to give

$$y_k = g^{(2)}\left(u_k^{(2)}\right), \tag{4.4}$$

for $k = 1, \ldots, m$. Note that the activation function in the output layer does not need to be the same than for that in the hidden layer.

If we combine (4.1), (4.2), (4.3) and (4.4) we obtain an explicit expression for the function represented by the neural network diagram in Figure 4.2 of the form

$$y_k = g^{(2)}\left(\sum_{j=0}^{h_1} \alpha_{kj}^{(2)} g^{(1)}\left(\sum_{i=0}^{n} \alpha_{ji}^{(1)} x_i\right)\right), \tag{4.5}$$

for $k = 1, \ldots, m$. 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. The neural network shown in Figure 4.2 corresponds to a transformation of the input variables by two successive different networks with a single layer. This class of networks can be extended by considering new successive transformation of the same kind, which corresponds to networks with more hidden layers.

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.

A multilayer perceptron with an arbitrary number of hidden layers is implemented within the C++ class MultilayerPerceptron of Flood [37].

4.3 The Jacobian matrix

There are some cases when, in order to evaluate the objective functional of a multilayer perceptron, we need to compute the derivatives of the network outputs with respect to the inputs. That derivatives can be grouped together in the Jacobian matrix, whose elements are given by

$$J_{ki} \equiv \frac{\partial y_k}{\partial x_i}, \tag{4.6}$$

for $i = 1, \ldots, n$, $k = 1, \ldots, m$ and where each such derivative is evaluated with all other inputs held fixed.

In Sections 8.2 and 8.4 we make use of the Jacobian matrix to evaluate two different objective functionals for the multilayer perceptron.

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

The back-propagation algorithm for the calculus of the Jacobian matrix

Here we evaluate the Jacobian matrix for the multilayer perceptron using the back-propagation algorithm. Results are derived for the one hidden layer perceptron shown in Figure 4.2, but they are easily generalized to networks with several hidden layers of neurons.

We start by writing the element J_{ki} in the form

$$
\begin{aligned}
J_{ki} &= \frac{\partial y_k}{\partial x_i} \\
&= \sum_{j=1}^{h_1} \frac{\partial y_k}{\partial u_j^{(1)}} \frac{\partial u_j^{(1)}}{\partial x_i},
\end{aligned} \tag{4.7}
$$

for $i = 1, \ldots, n$ and $k = 1, \ldots, m$. In order to determine the derivative $\partial u_j^{(1)} / \partial x_i$ let us consider the net input signal for each neuron in the hidden layer

$$
u_j^{(1)} = \sum_{i=0}^{n} \alpha_{ji}^{(1)} x_i, \tag{4.8}
$$

for $j = 1, \ldots, h_1$. Thus, the mentioned derivative yields

$$
\frac{\partial u_j^{(1)}}{\partial x_i} = \alpha_{ji}^{(1)}, \tag{4.9}
$$

for $i = 1, \ldots, n$ and $j = 1, \ldots, h_1$. Equation (4.7) becomes

$$
J_{ki} = \sum_{j=1}^{h_1} \alpha_{ij}^{(1)} \frac{\partial y_k}{\partial u_j^{(1)}}, \tag{4.10}
$$

for $i = 1, \ldots, n$ and $k = 1, \ldots, m$. We now write down a recursive back-propagation formula to determine the derivatives $\partial y_k / \partial u_j^{(1)}$.

$$
\frac{\partial y_k}{\partial u_j^{(1)}} = \sum_{l=1}^{m} \frac{\partial y_k}{\partial u_l^{(2)}} \frac{\partial u_l^{(2)}}{\partial u_j^{(1)}}, \tag{4.11}
$$

for $j = 1, \ldots, h_1$ and $k = 1, \ldots, m$. The derivative $\partial u_l^{(2)} / \partial u_j^{(1)}$ can be calculated by first considering the net input signal of each neuron in the output layer

$$
u_l^{(2)} = \sum_{j=0}^{h_1} \alpha_{lj}^{(2)} y_j^{(1)}, \tag{4.12}
$$

for $l = 1, \ldots, m$. The activation of each neuron in the hidden layer is given by

$$
y_j^{(1)} = g^{(1)}(u_j^{(1)}), \tag{4.13}
$$

for $j = 1, \ldots, h_1$. So we can write an expression for the net input signal of each neuron in the output layer in the form

$$
u_l^{(2)} = \sum_{j=0}^{h_1} \alpha_{lj}^{(2)} g^{(1)}(u_j^{(1)}), \tag{4.14}
$$

for $l = 1, \ldots, m$. Thus, the derivative $\partial u_l^{(2)} / \partial u_j^{(1)}$ yields

$$\frac{\partial u_l^{(2)}}{\partial u_j^{(1)}} = w_{lj}^{(2)} g'^{(1)}(u_j^{(1)}), \tag{4.15}$$

for $j = 1, \ldots, h_1$ and $l = 1, \ldots, m$. Equation (4.8) becomes

$$\frac{\partial y_k}{\partial u_j^{(1)}} = g'(u_j^{(1)}) \sum_{l=1}^{m} \alpha_{lj}^{(2)} \frac{\partial y_k^{(2)}}{\partial u_l^{(2)}}, \tag{4.16}$$

for $j = 1, \ldots, h_1$ and $k = 1, \ldots, m$. For the output neurons we have

$$y_k = g^{(2)}(u_k^{(2)}), \tag{4.17}$$

for $k = 1, \ldots, m$, and from which

$$\frac{\partial y_k}{\partial u_{k'}^{(2)}} = g'^{(2)}(u_k^{(2)}) \delta_{kk'}. \tag{4.18}$$

$\delta_{kk'}$ is the Kronecker delta symbol, and equals 1 if $k = k'$ and 0 otherwise. We can therefore summarize the procedure for evaluating the Jacobian matrix as follows [8]:

1. Apply the input vector corresponding to the point in input space at which the Jacobian matrix is to be found, and forward propagate in the usual way to obtain the activations of all of the hidden and output neurons in the network.

2. For each row k of the Jacobian matrix, corresponding to the output neuron k, back-propagate to the hidden units in the network using the recursive relation (4.16), and starting with (4.18).

3. Use (4.10) to do the back-propagation to the inputs. The second and third steps are the repeated for each value of k, corresponding to each row of the Jacobian matrix.

A C++ software implementation of this algorithm can be found inside the class MultilayerPerceptron of Flood [37].

Numerical differentiation for the calculus of the Jacobian matrix

The Jacobian matrix for the multilayer perceptron J_{ki} 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,

$$\frac{\partial y_k}{\partial x_i} = \frac{y_k(x_i + \epsilon) - y_k(x_i)}{\epsilon} + \mathcal{O}(\epsilon), \tag{4.19}$$

for $i = 1, \ldots, n$, $k = 1, \ldots, m$ 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

$$\frac{\partial y_k}{\partial x_i} = \frac{y_k(x_i + \epsilon) - y_k(x_i - \epsilon)}{2\epsilon} + \mathcal{O}(\epsilon^2), \tag{4.20}$$

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

In a software implementation the Jacobian matrix for the multilayer perceptron J_{ki} should be evaluated using the back-propagation algorithm, since this gives the greatest accuracy and numerical efficiency [8]. The implementation of such algorithm should be checked by using numerical differentiation. Both forward and central differences for the Jacobian are also implemented inside the class MultilayerPerceptron of Flood [37].

4.4 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 [29].

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 [29].

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.

4.5 Pre and post-processing

In practice it is always convenient to pre-process the inputs in order to produce input signals to the neural network of order zero. In this way, if all the neural parameters are of order zero, the output signals will be also of order zero.

There are several pre and post-processing methods. Two of the most used are the mean and standard deviation and the minimum and maximum pre and post-processing method. Note that these two forms of pre and post-processing will affect the back-propagation algorithm for the Jacobian matrix.

The mean and standard deviation pre and post-processing method

With this method the inputs are pre-processed so that the input signals have mean 0 and standard deviation 1,

$$\hat{x}_i = \frac{x_i - \rho(x_i)}{\sigma(x_i)}, \tag{4.21}$$

for $i = 1, \ldots, n$, and where x_i are the inputs, $\hat{x}_i$ are the input signals to the neural network, and $\rho(x_i)$ and $\sigma(x_i)$ are an estimation of the mean and the standard deviation of the input variables, respectively.

The output signals from the neural network are then post-processed to produce the outputs

$$y_i = \rho(y_i) + \sigma(y_i)\hat{y}_i, \tag{4.22}$$

for $i = 1, \ldots, m$, and where y_i are the outputs, $\hat{y}_i$ are the output signals from the network, and $\rho(y_i)$ and $\sigma(y_i)$ are an estimation of the mean and the standard deviation of the output variables, respectively.

The minimum and maximum pre and post-processing method

Here the inputs are pre-processed to produce input signals whose minimum and maximum values are -1 and 1,

$$\hat{x}_i = 2\frac{x_i - \min(x_i)}{\max(x_i) - \min(x_i)} - 1, \tag{4.23}$$

for $i = 1, \ldots, n$, and where x_i are the inputs, $\hat{x}_i$ are the input signals to the network, and $\min(x_i)$ and $\max(x_i)$ are an estimation of the minimum and the maximum values of the input variables, respectively.

The output signals from the network are then post-processed to produce the outputs

$$y_i = 0.5\,(\hat{y}_i + 1)\,(\max(y_i) - \min(y_i)) + \min(y_i), \tag{4.24}$$

for $i = 1, \ldots, m$, and where y_i are the outputs, $\hat{y}_i$ are the output signals from the network, and $\min(y_i)$ and $\max(y_i)$ are an estimation of the minimum and the maximum values of the outputs from the neural network, respectively.

4.6 Multilayer perceptron extensions

This Section describes some possible extensions for the multilayer perceptron. They include independent parameters, boundary conditions and lower and upper bounds. In some situations these extensions can improve the performance by the numerical method. In some other cases they allow to deal with applications which would be untractable otherwise. In summary, this augmented class of neural network might be able to span a more suited function space for some variational problems.

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 q-dimensional vector $\underline{\beta} = (\beta_1, \ldots, \beta_q)$.

Consequently, a multilayer perceptron with associated independent parameters spans a space of functions V from an input $X \subseteq \mathbb{R}^n$ to an output $Y \subseteq \mathbb{R}^m$, where n and m are the number of inputs and outputs, respectively. The elements of this

family of functions are parameterized by both, the biases and synaptic weights vector, $\underline{\alpha} = (\alpha_1, \ldots, \alpha_p)$, and the independent parameters vector, $\underline{\beta} = (\beta_1, \ldots, \beta_q)$. The dimension of V is thus $p + q$, and the functions here are of the form

$$\begin{aligned} \mathbf{y} : X &\rightarrow Y \\ \mathbf{x} &\mapsto \mathbf{y}(\mathbf{x}; \underline{\alpha}, \underline{\beta}). \end{aligned}$$

Note that the total set of free parameters is composed by the biases and synaptic weights and the 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.

Examples of problems in which independent parameters are used can be found in Sections 10.2 and 9.3 .

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 $x = a$ and some output $y = y_a$ is written $y(a) = y_a$. In order to deal with boundary conditions the output signals from the neural network can be post-processed as follows

$$\mathbf{y}(\mathbf{x}; \underline{\alpha}, \underline{\beta}) = \varphi_0(\mathbf{x}) + \varphi_1(\mathbf{x})\mathbf{y}(\mathbf{x}; \underline{\alpha}, \underline{\beta}), \tag{4.25}$$

where the function $\varphi_0(\mathbf{x})$ is called a particular solution term and the function $\varphi_1(\mathbf{x})$ is called an homogeneous solution term. The first must hold $\varphi_0(a) = y_a$ if there is a condition $y(a) = y_a$. The second must hold $\varphi_1(a) = 0$ if there is a condition $y(a) = y_a$. It is easy to see that this approach makes all the elements of the function space to satisfy the boundary conditions. It is important to remark that the expressions of the particular and homogeneous solution terms depend on the problem at hand.

For the simplest case of one input and one output variables and one boundary condition $y(a) = y_a$, a possible set of particular and homogeneous solution terms could be

$$\varphi_0(x) = a, \tag{4.26}$$

$$\varphi_1(x) = x - a. \tag{4.27}$$

In the situation of one input and one output variables and two conditions $y(a) = y_a$ and $y(b) = y_b$ we could have

$$\varphi_0(x) = y_a + \frac{y_b - y_a}{b - a} x, \tag{4.28}$$

$$\varphi_1(x) = (x - a)(x - b). \tag{4.29}$$

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.

Also note that the Jacobian matrix for the multilayer perceptron will be affected if the problem includes boundary conditions. The approach is to make use of the product rule for differentiation in Equation (4.25)

Some examples of the use of boundary conditions for the multilayer perceptron can be found in Sections 8.1, 8.2, 8.3 and 8.4.

Lower and upper bounds

If some output variables are restricted to fall in some interval, then the problem is said to have lower and upper bounds. An easy way to treat lower and upper bounds is to post-process the outputs from Equation (4.25) in the next way,

$$\mathbf{y}(\mathbf{x};\underline{\alpha},\underline{\beta}) = \begin{cases} \inf(\mathbf{y}), & \mathbf{y}(\mathbf{x};\underline{\alpha},\underline{\beta}) < \inf(\mathbf{y}), \\ \mathbf{y}(\mathbf{x};\underline{\alpha},\underline{\beta}), & \inf(\mathbf{y}) \leq \mathbf{y}(\mathbf{x};\underline{\alpha},\underline{\beta}) \leq \sup(\mathbf{y}), \\ \sup(\mathbf{y}), & \mathbf{y}(\mathbf{x};\underline{\alpha},\underline{\beta}) > \sup(\mathbf{y}), \end{cases} \tag{4.30}$$

where $\inf(\mathbf{y})$ and $\sup(\mathbf{y})$ represent the infimum and supremum of the output vector $\mathbf{y}$, respectively.

Similarly, if some independent parameters are bounded they can be post-processed in the following manner,

$$\underline{\beta} = \begin{cases} \inf(\underline{\beta}), & \underline{\beta} < \inf(\underline{\beta}), \\ \underline{\beta}, & \inf(\underline{\beta}) \leq \underline{\beta} \leq \sup(\underline{\beta}), \\ \sup(\underline{\beta}), & \underline{\beta} > \sup(\underline{\beta}), \end{cases} \tag{4.31}$$

where $\inf(\underline{\beta})$ and $\sup(\underline{\beta})$ represent the infimum and supremum of the independent parameter vector $\underline{\beta}$, respectively.

Examples of problems with lower and upper bounds are found in Sections 9.3, or 11.2.

4.7 The input-output activity diagram

Following the contents of this section, an activity diagram for the input-output process in an extended class of multilayer perceptron with a single hidden layer of neurons can be drawn as in Figure 4.3.

4.8 The MultilayerPerceptron class in Flood

`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 [29].

This class contains

- The number of inputs.
- The numbers of hidden neurons.
- 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.

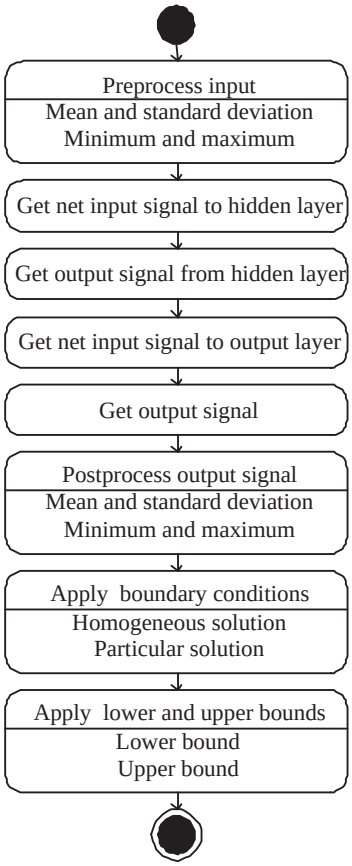

Figure 4.3: Activity diagram for the input-output process in the multilayer perceptron.

- 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 number of independent parameters.
- A vector of independent parameters.
- 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 pre and post-processing method.

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 following sentence:

```
MultilayerPerceptron multilayerPerceptron(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. On the other hand, the activation functions of the hidden layer and the output layer are set by default to hyperbolic tangent and linear, respectively.

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 input we can write

```
Vector<int> numbersOfHiddenNeurons(3);

numbersOfHiddenNeurons[0] = 2;

numbersOfHiddenNeurons[1] = 4;

numbersOfHiddenNeurons[2] = 3;

MultilayerPerceptron
multilayerPerceptron(1,numbersOfHiddenNeurons,1);
```

The neural parameters here are also initialized at random. Also, the activation functions are set to hyperbolic tangent for the hidden neurons and to linear for the output neurons. However, the activation functions can be changed by doing

```
Vector<Perceptron::ActivationFunction>
hiddenLayersActivationFunction(3, Perceptron::Logistic);

multilayerPerceptron
.setHiddenLayersActivationFunction(hiddenLayersActivationFunction);

multilayerPerceptron
.setOutputLayerActivationFunction(Perceptron::HyperbolicTangent);
```

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

```
int numberOfNeuralParameters =
multilayerPerceptron.getNumberOfNeuralParameters();
```

To set the mean and the standard deviation of the input and output variables we can use the methods setMeanAndStandardDeviationOfInputVariables(Matrix<**double**>) and setMeanAndStandardDeviationOfOutputVariables(Matrix<**double**>), respectively. For instance, the sentences

```
Matrix<double> meanAndStandardDeviationOfInputVariables(2,1);
meanAndStandardDeviationOfInputVariables[0][0] = 0.0;
meanAndStandardDeviationOfInputVariables[1][0] = 1.0;

multilayerPerceptron.setMeanAndStandardDeviationOfInputVariables
(meanAndStandardDeviationOfInputVariables);

Matrix<double> meanAndStandardDeviationOfOutputVariables(2,1);
meanAndStandardDeviationOfOutputVariables[0][0] = 0.0;
meanAndStandardDeviationOfOutputVariables[1][0] = 1.0;

multilayerPerceptron.setMeanAndStandardDeviationOfOutputVariables
(meanAndStandardDeviationOfOutputVariables);
```

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

To set the minimum and the maximum values of the input and output variables we can use the methods setMinimumAndMaximumOfInputVariables(Matrix<**double**>) and setMinimumAndMaximumOfOutputVariables(Matrix<**double**>), respectively. Indeed, the sentences

```
Matrix<double> minimumAndMaximumOfInputVariables(2,1,0.0);
minimumAndMaximumOfInputVariables[0][0] = -1.0;
minimumAndMaximumOfInputVariables[1][0] = 1.0;

multilayerPerceptron.setMinimumAndMaximumOfInputVariables
(minimumAndMaximumOfInputVariables);

Matrix<double> minimumAndMaximumOfOutputVariables(2,1,0.0);
minimumAndMaximumOfOutputVariables[0][0] = -1.0;
minimumAndMaximumOfOutputVariables[1][0] = 1.0;

multilayerPerceptron.setMinimumAndMaximumOfOutputVariables
(minimumAndMaximumOfOutputVariables);
```

set the minimum and maximum values of both input and output variables to -1 and 1, respectively.

By default, a multilayer perceptron has not assigned any pre and post-processing method. In order to use the mean and standard deviation pre and post-processing method we can write

```
multilayerPerceptron.setPreAndPostProcessingMethod
(MultilayerPerceptron::MeanAndStandardDeviation);
```

In the same way, if we want to use the minimum and maximum pre and post-processing method we can use

```
multilayerPerceptron.setPreAndPostProcessingMethod
(MultilayerPerceptron::MinimumAndMaximum);
```

To calculate the output Vector of the network in response to an input Vector we use the method calculateOutput(Vector<**double**>). For instance, the sentence

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

Vector<double> output = multilayerPerceptron.calculateOutput(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 calculateJacobian(Vector<**double**>). For instance, the sentence

```
Matrix<double> jacobian =
multilayerPerceptron.calculateJacobian(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 setNumberOfIndependentParameters(**unsigned int**). For example,

```
multilayerPerceptron.setNumberOfIndependentParameters(2);
```

Note that the number of free parameters of this neural network is now the number of neural parameters plus the number of independent parameters,

```
int numberOfFreeParameters =
multilayerPerceptron.getNumberOfFreeParameters();
```

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

```
multilayerPerceptron.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(**char***). Indeed, the sentence

```
multilayerPerceptron.load("MultilayerPerceptron.dat");
```

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

The file MultilayerPerceptronTemplate.dat in `Flood` shows the format of a multilayer perceptron data file.

Finally, the source code of a sample application (main function) using the MultilayerPerceptron class can be found at the MultilayerPerceptronApplication.cpp file of `Flood`.

Chapter 5

The objective functional

In order to perform a particular task a multilayer perceptron must be associated an objective functional. The objective functional defines the task the neural network is required to accomplish and provides a measure of the quality of the representation that the network is required to learn. The choice of a suitable objective functional for a multilayer perceptron 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 The variational problem

Traditionally, the learning problem for the multilayer perceptron has been formulated in terms of the minimization of an error function of the free parameters in the neural network, in order to fit the neural network to an input-target data set [8]. In that way, the only learning tasks allowed for the multilayer perceptron are data modeling type problems.

In a variational formulation for neural networks, the concept of error function, $e(\underline{\alpha})$, is changed by the concept or objective functional, $F[\mathbf{y}(\mathbf{x};\underline{\alpha})]$ [42]. An objective functional for the multilayer perceptron is of the form

$$\begin{array}{rccl} F: & V & \rightarrow & \mathbb{R} \\ & \mathbf{y}(\mathbf{x};\underline{\alpha}) & \mapsto & F[\mathbf{y}(\mathbf{x};\underline{\alpha})]. \end{array}$$

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 $\mathbf{y}^*(\mathbf{x};\underline{\alpha}^*)$ at which the objective functional $F[\mathbf{y}(\mathbf{x};\underline{\alpha})]$ 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.

As we will see, changing the concept of error function by the concept of objective functional allows us to extend the number of learning tasks for the multilayer perceptron to any variational problem. Some examples are optimal control problems [38], inverse problems [14] or optimal shape design problems [40].

The simplest variational problems are those in which no constraints are imposed on the solution. An unconstrained variational problem of historical interest is the brachistochrone problem [22]. However, there are many applications of the calculus of variations in which constraints are imposed. Such constraints are expressed as functionals. A classical constrained problem in the calculus of variations is the isoperimetric problem [22].

Unconstrained problems

The simplest variational problems for the multilayer perceptron are those in which no constraints are posed on the solution $\mathbf{y}^*(\mathbf{x}; \underline{\alpha}^*)$. 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 $\mathbf{y}(\mathbf{x}; \underline{\alpha})$ spanned by a multilayer perceptron, and let s be the dimension of V. Find a function $\mathbf{y}^*(\mathbf{x}; \underline{\alpha}^*) \in V$ for which the functional*

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

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 [42].

In Chapter 7 we use a multilayer perceptron to solve function regression and pattern recognition problems. That are unconstrained variational problems. The geodesic problem and the brachistochrone problem, which are solved in Chapter 8 are also examples of unconstrained variational problems.

Constrained problems

A variational problem for the multilayer perceptron can be specified by a set of constraints, which are equalities or inequalities that the solution $\mathbf{y}^*(\mathbf{x}; \underline{\alpha}^*)$ 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 $\mathbf{y}(\mathbf{x}; \underline{\alpha})$ spanned by a multilayer perceptron, and let s be the dimension of V. Find a function $\mathbf{y}^*(\mathbf{x}; \underline{\alpha}^*) \in V$ such that*

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

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

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

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 free 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 $\mathbf{y}(\mathbf{x};\underline{\alpha})$ that a given multilayer perceptron can define, and let s be the dimension of V. Find a function $\mathbf{y}^*(\mathbf{x};\underline{\alpha}^*) \in V$ for which the functional*

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

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_i, it is clear that the solution $\mathbf{y}^*(\mathbf{x};\underline{\alpha}^*)$ of Problem 3 will be in in a region where $C_i[\mathbf{y}(\mathbf{x};\underline{\alpha})]$, $i = 1, \ldots, l$, are small. Thus, for increasing values of ρ_i, it is expected that the the solution $\mathbf{y}^*(\mathbf{x};\underline{\alpha}^*)$ of Problem 3 will approach the constraints and, subject to being close, will minimize the objective functional $F[\mathbf{y}(\mathbf{x};\underline{\alpha})]$. Ideally then, as $\rho_i \to \infty$, $i = 1, \ldots, l$, the solution of Problem 3 will converge to the solution of Problem 2 [45].

In Chapter 8 we use a multilayer perceptron to solve the catenary problem and the isoperimetric problem. Other constrained problems are solved in Chapters 9 and 11.

5.2 The reduced function optimization problem

The objective functional, $F[\mathbf{y}(\mathbf{x};\underline{\alpha})]$, has an objective function associated, $f(\underline{\alpha})$, which is defined as a function of the free parameters in the neural network [42],

$$\begin{aligned} f : \mathbb{R}^s &\to \mathbb{R} \\ \underline{\alpha} &\mapsto f(\underline{\alpha}). \end{aligned}$$

The objective function for the multilayer perceptron is represented as $f(\underline{\alpha})$, and can be visualized as a hypersurface, with $\alpha_1, \ldots, \alpha_s$ as coordinates, see Figure 5.1.

The minimum or maximum value of the objective functional is achieved for a vector of free 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 [42].

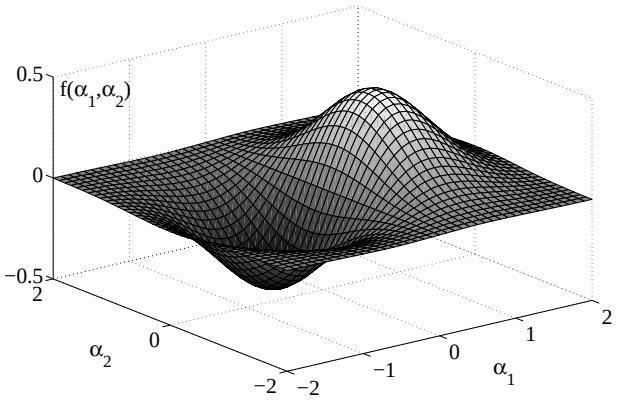

Figure 5.1: Geometrical representation of the objective function.

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

$$f(\underline{\alpha}),$$

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

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.

5.3 The objective function gradient

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

The use of gradient information is of central importance in finding training algorithms which are sufficiently fast to be of practical use for large-scale applications. For a multilayer perceptron, the gradient vector ∇ of the objective function $f(\alpha_1, \ldots, \alpha_s)$ is written:

$$\nabla f(\alpha_1, \ldots, \alpha_s) = \left(\frac{\partial f}{\partial \alpha_1}, \ldots, \frac{\partial f}{\partial \alpha_s} \right). \tag{5.1}$$

Figure 5.2 represents the objective function gradient vector for the hypothetical case of a multilayer perceptron with two free parameters.

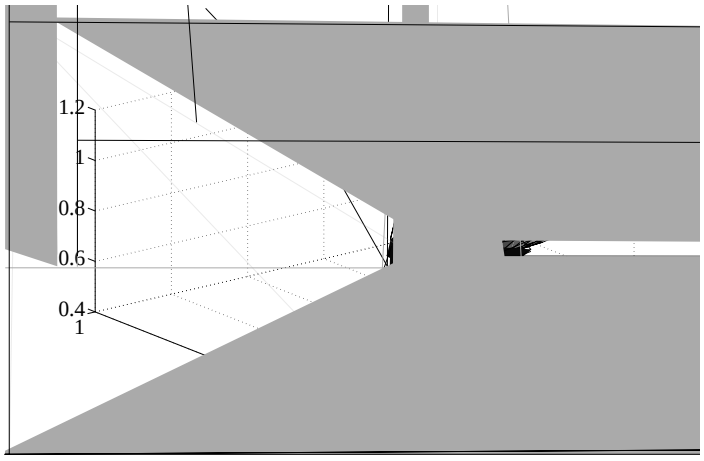

Figure 5.2: Illustration of the objective function gradient vector.

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 calculus of the objective function gradient

Here we obtain the objective function gradient $\nabla f(\underline{\alpha})$ for the multilayer perceptron using the back-propagation algorithm. Results are derived for the one hidden layer perceptron shown in Figure 4.2, but they can be easily generalized to networks with several hidden layers of neurons.

In a one hidden layer perceptron, the net input signal for each neuron in the hidden layer is of the form

$$u_j^{(1)} = \sum_{i=0}^{n} \alpha_{ji}^{(1)} x_i, \tag{5.2}$$

for $j = 1, \ldots, h_1$. In the same way, the net input signal for each neuron in the output layer is of the form

$$u_k^{(2)} = \sum_{j=0}^{h_1} \alpha_{kj}^{(2)} y_j^{(1)}, \tag{5.3}$$

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

The net input signal of neuron j in the hidden layer is transformed by an activation function $g^{(1)}$ to give the activation $y_j^{(1)}$

$$y_j^{(1)} = g^{(1)}(u_j^{(1)}), \tag{5.4}$$

for $j = 1, \ldots, h_l$. Similarly, for the output layer,

$$y_k = g^{(2)}(u_k^{(2)}), \tag{5.5}$$

for $k = 1, \ldots, m$. As we said before, the learning algorithm will aim to find suitable values for the free parameters in the neural network by optimizing the objective function $f(\underline{\alpha})$. On the other hand, we consider objective functions which ca be expressed as a differentiable function of the output signals $\mathbf{y}$ from the neural network. Therefore, the objective function of a multilayer perceptron can be expressed as a function of the output signals, which depend in turn on the free parameters

$$f = f(\alpha_1, \ldots, \alpha_s; y_1, \ldots, y_m). \tag{5.6}$$

We want to find a procedure to evaluate the derivatives of the objective function f with respect to the free parameters in the network $\alpha_1, \ldots, \alpha_s$.

First we obtain the activation of all hidden and output neurons by a consecutive applying of (5.2), (5.4), (5.3) and (5.5). This process is often called feed-propagation, 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 free parameter α_{ji}. f depends on the free parameter α_{ji} only though the net input signal u_j to neuron j. We can then apply the chain rule for partial derivatives to give, for the hidden layer

$$\frac{\partial f}{\partial \alpha_{ji}^{(1)}} = \frac{\partial f}{\partial u_j^{(1)}} \frac{\partial u_j^{(1)}}{\partial \alpha_{ji}^{(1)}}, \tag{5.7}$$

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

$$\frac{\partial f}{\partial \alpha_{kj}^{(2)}} = \frac{\partial f}{\partial u_k^{(2)}} \frac{\partial u_k^{(2)}}{\partial \alpha_{kj}^{(2)}}, \tag{5.8}$$

for $j = 1, \ldots, h_1$ and $k = 1, \ldots, m$. We now introduce the notation

$$\delta \equiv \frac{\partial f}{\partial u}. \tag{5.9}$$

The quantity δ is called back-propagation error, and it is considered for each neuron in the neural network. Therefore, the definition of the back-propagation error for a neuron in the hidden layer is

$$\delta_j^{(1)} \equiv \frac{\partial f}{\partial u_j^{(1)}}, \tag{5.10}$$

for $j = 1, \ldots, h_1$. The definition of the back-propagation error for a neuron in the output layer is

$$\delta_k^{(2)} \equiv \frac{\partial f}{\partial u_k^{(2)}}, \tag{5.11}$$

for $k = 1, \ldots, m$. On the other hand, using Equation (5.2) we can write, for the hidden layer

$$\frac{\partial u_j^{(1)}}{\partial \alpha_{ji}^{(1)}} = x_i, \tag{5.12}$$

for $i = 1, \ldots, n$ and $j = 1, \ldots, h_1$. Similarly, using Equation (5.3) we can write, for the output layer,

$$\frac{\partial u_k^{(2)}}{\partial \alpha_{kj}^{(2)}} = y_j^{(1)}, \tag{5.13}$$

for $j = 1, \ldots, h_1$ and $k = 1, \ldots, m$. Substituting (5.10) and (5.12) in (5.7) we obtain, for the hidden layer

$$\frac{\partial f}{\partial \alpha_{ji}^{(1)}} = \delta_j^{(1)} x_i, \tag{5.14}$$

for $i = 1, \ldots, n$ and $j = 1, \ldots, h_1$. Likewise, substituting (5.11) and (5.13) in (5.8) we obtain, for the output layer

$$\frac{\partial f}{\partial \alpha_{kj}^{(2)}} = \delta_k^{(2)} y_j^{(1)}, \tag{5.15}$$

for $j = 1, \ldots, h_1$ and $k = 1, \ldots, m$. Thus, to evaluate the required derivatives, we only need to evaluate the value of δ for each neuron in the neural network and then to apply (5.10) and (5.11).

The evaluation of $\delta_k^{(2)}$ for the output neurons is quite simple. From the definition in Equation (5.10) we have

$$\begin{aligned} \delta_k^{(2)} &\equiv \frac{\partial f}{\partial u_k^{(2)}} \\ &= \frac{\partial f}{\partial y_k^{(2)}} \frac{\partial {y_k}^{(2)}}{\partial u_k^{(2)}} \\ &= g'^{(2)}(u_k^{(2)}) \frac{\partial f}{\partial y_k^{(2)}}, \end{aligned} \tag{5.16}$$

for $k = 1, \ldots, m$. To evaluate $\delta_j^{(1)}$ for the hidden neurons we make use of the chain rule for partial derivatives,

$$\begin{aligned} \delta_j^{(1)} &\equiv \frac{\partial f}{\partial u_j^{(1)}} \\ &= \sum_{k=1}^{m} \frac{\partial f}{\partial u_k^{(2)}} \frac{\partial u_k^{(2)}}{\partial u_j^{(1)}}, \end{aligned} \tag{5.17}$$

for $j = 1, \ldots, h_1$. Substituting now the definition of δ for the neurons in the output layer given in (5.11) into (5.17), and making use of (5.2) and (5.4), we obtain

$$\delta_j^{(1)} = g'^{(1)}(u_j^{(1)}) \sum_{k=1}^{m} \alpha_{kj}^{(2)} \delta_k^{(2)}, \tag{5.18}$$

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 free parameters in just four steps:

1. Apply an input vector $\mathbf{x}$ to the neural network and feed-propagate it using (5.2), (5.4), (5.3) and (5.5) to find the activation of all hidden and output neurons.

2. Evaluate the errors $\delta_k^{(2)}$ for all output neurons using (5.16).

3. Back-propagate the errors $\delta_k^{(2)}$ by using (5.18) to obtain $\delta_j^{(1)}$ for each hidden neuron in the neural network.

4. Use (5.10) and (5.11) to evaluate the required derivatives of the objective function with respect to the free parameters in the hidden and output layers, respectively.

A C++ implementation of this algorithm for some particular objective functions can be found in the classes SumSquaredError, MeanSquaredError, RootMeanSquaredError, NormalizedSquaredError, MinkowskiError and RegularizedMinkowskiError of Flood [37].

In Sections 7.5 and 7.6 we make use of the back-propagation algorithm to obtain the objective function gradient of the sum squared error, which is a common objective functional for function regression and pattern recognition problems.

Numerical differentiation for the calculus of the objective function gradient

There are many applications when it is not possible to obtain the objective function gradient $\nabla f(\underline{\alpha})$ using the back-propagation algorithm, and it needs to be computed numerically. This can be done by perturbing each free parameter in turn, and approximating the derivatives by using the finite differences method

$$\frac{\partial f}{\partial \alpha_i} = \frac{f(\alpha_i + \epsilon) - f(\alpha_i)}{\epsilon} + \mathcal{O}(\epsilon), \tag{5.19}$$

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

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

$$\frac{\partial f}{\partial \alpha_i} = \frac{f(\alpha_i + \epsilon) - f(\alpha_i - \epsilon)}{2\epsilon} + \mathcal{O}(\epsilon^2), \tag{5.20}$$

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

We have seen that the derivatives of the objective function with respect to the free parameters in the hidden layer can be expressed efficiently through the relation

$$\frac{\partial f}{\partial \alpha_{ji}^{(1)}} = \frac{\partial f}{\partial u_j^{(1)}} x_i, \tag{5.21}$$

for $i = 1, \ldots, n$ and $j = 1, \ldots, h_1$. Similarly, the derivatives of the objective function with respect to the free parameters in the output layer are given by

$$\frac{\partial f}{\partial \alpha_{kj}^{(2)}} = \frac{\partial f}{\partial u_k^{(2)}} y_j^{(1)}, \tag{5.22}$$

for $j = 1, \ldots, h_1$ and $k = 1, \ldots, m$. Instead of using the technique of central differences to evaluate the derivatives $\partial f / \partial \alpha$ directly, we can use it to estimate $\partial f / \partial u$ since

$$\frac{\partial f}{\partial u_i} = \frac{f(u_i + \epsilon) - f(u_i - \epsilon)}{2\epsilon} + \mathcal{O}(\epsilon^2), \tag{5.23}$$

for $i = 1, \ldots, r$, where $r = h_1 + m$ is the total number of neurons in the network. We can then make use of Equations (5.21) and (5.22) to evaluate the required derivatives.

In Sections 8.2 and 8.4 we make use of numerical differentiation to obtain the objective function gradient of the objective functionals for the brachistochrone and the isoperimetric problems, respectively.

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

The source code for the calculus of the gradient vector by means of central differences is within the class ObjectiveFunctional of Flood [37].

5.4 The 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 free parameters. The Hessian matrix of the objective function is written

$$\mathbf{H} f(\underline{\alpha}) = \begin{pmatrix} \dfrac{\partial^2 f}{\partial \alpha_1^2} & \cdots & \dfrac{\partial^2 f}{\partial \alpha_1 \partial \alpha_n} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial \alpha_n \partial \alpha_1} & \cdots & \dfrac{\partial^2 f}{\partial \alpha_n^2} \end{pmatrix} \tag{5.24}$$

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 calculus of 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 the central differences method

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

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

Equation (5.25) is implemented in the class ObjectiveFunctional of Flood [37].

5.5 The ObjectiveFunctional classes in Flood

`Flood` includes the abstract class ObjectiveFunctional to represent the concept of objective functional. That class contains:

1. A relationship to a multilayer perceptron object.
2. The numerical epsilon method.
3. The numerical differentiation method.

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.

Any derived class must implement the pure virtual calculateEvaluation(**void**) method, which returns the evaluation of a multilayer perceptron for some objective functional.

Derived classes might also implement the calculateGradient(**void**) and calculateHessian(**void**) methods. By default, calculation of the gradient vector and the Hessian matrix is performed with numerical differentiation. Implementation of that methods will override them and allow to compute the derivatives analytically.

For data modeling problems, such as function regression, pattern recognition or time series prediction, `Flood` includes the classes SumSquaredError, MeanSquaredError, RootMeanSquaredError, NormalizedSquaredError, MinkowskiError and RegularizedMinkowskiError. Read Chapter 7 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 classes GeodesicProblem, BrachistochroneProblem, CatenaryProblem and IsoperimetricProblem, which are classical problems in the calculus of variations. Chapter 8 is devoted to that types of problems and explains how to solve them.

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

Regarding inverse problems, `Flood` includes the PrecipitateDissolutionModeling class as an example. Read Chapter 10 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 class is included. All these is explained in Chapter 11.

Finally, `Flood` can be used as a software tool for function optimization problems. Some examples included in `Flood` are DeJongFunction, RosenbrockFunction, RastriginFunction and PlaneCylinder. 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.

As we said in Chapter 5, the learning problem in the multilayer perceptron, formulated as a variational problem, can be reduced to a function optimization problem. The training algorithm is entrusted to solve the reduced function optimization problem, by adjusting the parameters in the neural network so as to optimize the objective function.

6.1 The function optimization problem

The learning problem in the multilayer perceptron has been reduced to the searching in an s-dimensional space for a parameter vector $\underline{\alpha}^*$ for which the objective function f takes a maximum or a minimum value.

The tasks of maximization and minimization are trivially related to each other, since maximization of $f(\underline{\alpha})$ is equivalent to minimization of $-f(\underline{\alpha})$, and vice versa.

On the other side, 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. Finding a global minimum is, in general, a very difficult problem [67].

Consider an objective function

$$\begin{aligned} f \quad &: \quad \mathbb{R}^s \to \mathbb{R} \\ &\quad\;\; \underline{\alpha} \quad \mapsto f(\underline{\alpha}) \end{aligned}$$

continuous, derivable, and with continuous derivatives.

The global minimum condition for the parameter vector $\underline{\alpha}^*$ is stated as

$$f(\underline{\alpha}^*) \leq f(\underline{\alpha}), \tag{6.1}$$

for all $\alpha_1, \ldots, \alpha_s$ in the parameter space.

On the other hand, the necessary local minimum condition for the parameter vector $\underline{\alpha}^*$ is stated as

$$\nabla f(\underline{\alpha}^*) = 0. \tag{6.2}$$

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

$$\alpha^{(i+1)} = \alpha^{(i)} + \Delta\alpha^{(i)}, \tag{6.3}$$

where i labels the iteration step, or epoch. The quantity $\Delta\alpha^{(i)}$ is called parameter vector increment. Different training algorithms involve different choices for this vector.

In this way, to train a multilayer perceptron we start with an initial estimation of the parameter vector $\underline{\alpha}^{(0)}$ (often chosen at random) and we generate a sequence of parameter vectors $\underline{\alpha}^{(1)}$, $\underline{\alpha}^{(2)}, \ldots$, so that the objective function f is reduced at each iteration of the algorithm, that is

$$f(\alpha^{(i+1)}) < f(\alpha^{(i)}). \tag{6.4}$$

The training algorithm stops when a specified condition is satisfied. Some stopping criteria commonly used are [16]:

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

Figure 6.1 is a state diagram of this iterative procedure, showing states and transitions in the training process of a 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 [67].

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 [56]. 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[25] [21] or particle swarm optimization [31], which are global optimization methods .

First-order training algorithms use the objective function and its gradient vector [4]. Examples of these are gradient descent methods, conjugate gradient methods,

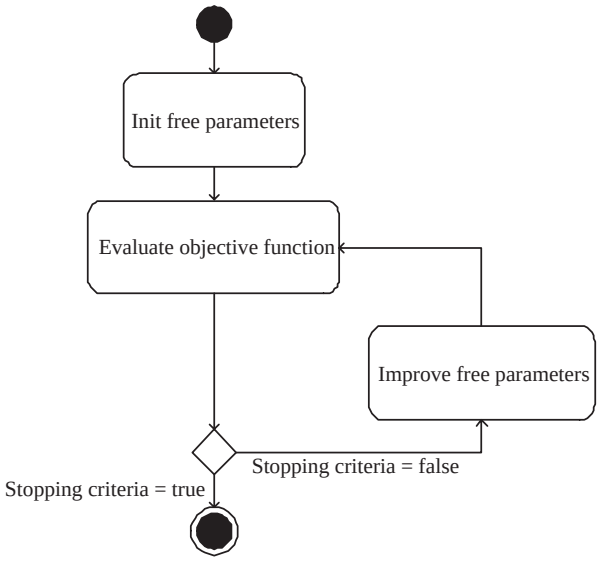

Figure 6.1: Training process in the multilayer perceptron.

scaled conjugate gradient methods [48] or quasi-Newton methods. Gradient descent, conjugate gradient, scaled conjugate gradient and quasi-Newton methods are local optimization methods [45].

Second-order training algorithms make use of the objective function, its gradient vector and its Hessian matrix [4]. Examples for second-order methods are Newton's method and the Levenberg-Marquardt algorithm [26]. Both of them are local optimization methods [45].

6.2 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.

The random search method simply consists of sampling a stream parameter vectors

$$\underline{\alpha}^{(1)}, \underline{\alpha}^{(2)}, \ldots$$

distributed at random, and while evaluating their objective function

$$f^{(1)}(\underline{\alpha}^{(1)}), f^{(2)}(\underline{\alpha}^{(2)}), \ldots$$

until a stopping criterium is satisfied.

Random search can be performed either using a uniform distribution, i.e., with constant probability, or using a normal distribution, i.e., with probability defined by a mean and an standard deviation.

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. A C++ implementation of random search can be found in the class RandomSearch of Flood [37].

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 $\underline{\alpha}^{(0)}$ and, until a stopping criterium is satisfied, moves from $\underline{\alpha}^{(i)}$ to $\underline{\alpha}^{(i+1)}$ along the line extending from $\underline{\alpha}^{(i)}$ in the direction of $-\nabla f(\underline{\alpha}^{(i)})$, the local downhill gradient. The gradient vector of the objective function for the multilayer perceptron is described in Section 5.3.

Let denote $\mathbf{g} \equiv \nabla f(\underline{\alpha})$ the gradient vector. Therefore, starting from a parameter vector $\underline{\alpha}^{(0)}$, the gradient descent method takes the form of iterating

$$\underline{\alpha}^{(i+1)} = \underline{\alpha}^{(i)} - \eta^{(i)} \mathbf{g}^{(i)}, \tag{6.5}$$

for $i = 0, 1, \ldots$, and where the parameter $\eta^{(i)}$ is called 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 $\underline{\alpha}^*$ 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 6.7 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.

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 [45] for a detailed discussion of this optimization method. The class GradientDescent within Flood [37] contains a C++ source of the gradient descent training algorithm.

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

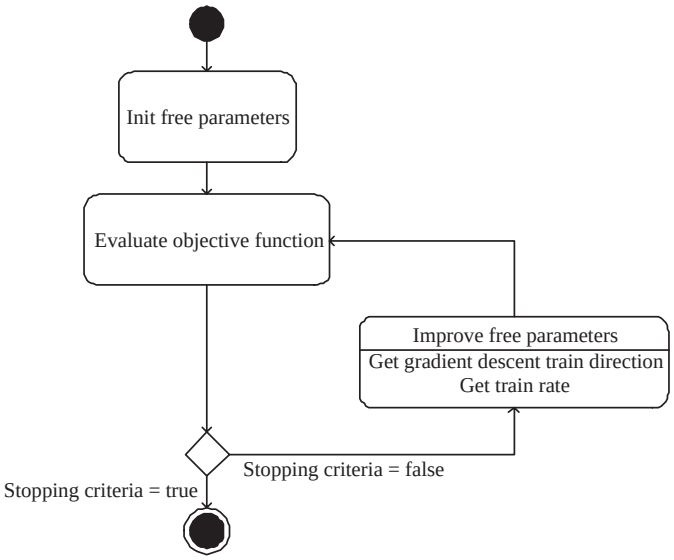

Figure 6.2: Training process with the gradient descent training algorithm.

other hand, the Newton's method behaves in a deterministic fashion.

Consider the quadratic approximation of f at $\underline{\alpha} = \underline{\alpha}^{(i)}$ using the Taylor's series expansion

$$f(\underline{\alpha}) = f(\underline{\alpha}^{(i)}) + \nabla f(\underline{\alpha} - \underline{\alpha}^{(i)}) + \frac{1}{2}(\underline{\alpha} - \underline{\alpha}^{(i)})^{\mathrm{T}} \cdot \mathbf{H}f(\underline{\alpha}^{(i)}) \cdot (\underline{\alpha} - \underline{\alpha}^{(i)}), \tag{6.6}$$

where $\mathbf{H}f(\underline{\alpha}^{(i)})$ is the Hessian matrix of f evaluated at the point $\underline{\alpha}^{(i)}$. The Hessian matrix of the objective function for the multilayer perceptron is described in Section 5.4.

By setting $\nabla f(\underline{\alpha})$ in Equation (6.6) equal to $\mathbf{0}$ for the minimum of $f(\underline{\alpha})$, we obtain

$$\begin{aligned} \nabla f(\underline{\alpha}) &= \nabla f(\underline{\alpha}^{(i)}) + \mathbf{H}f(\underline{\alpha}^{(i)}) \cdot (\underline{\alpha} - \underline{\alpha}^{(i)}) \\ &= \mathbf{0}. \end{aligned} \tag{6.7}$$

If $\mathbf{H}f(\underline{\alpha}^{(i)})$ is not singular, Equation (6.7) leads to an expression for the location of the minimum of the objective function,

$$\underline{\alpha}^{*} = \underline{\alpha}^{(i)} - \mathbf{H}^{-1}f(\underline{\alpha}^{(i)}) \cdot \nabla f(\underline{\alpha}^{(i)}), \tag{6.8}$$

where $\mathbf{H}^{-1}f(\underline{\alpha}^{(i)})$ is the inverse of the Hessian matrix of f evaluated at the point $\underline{\alpha}^{(i)}$.

Equation (6.8) 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 $\underline{\alpha}^{*}$. Let denote $\mathbf{g} \equiv \nabla f(\underline{\alpha})$ and $\mathbf{H} \equiv \mathbf{H}f(\underline{\alpha})$. Therefore, starting from a parameter vector $\underline{\alpha}^{(0)}$, the iterative formula for the Newton's method can be written

$$\underline{\alpha}^{(i+1)} = \underline{\alpha}^{(i)} - \mathbf{H}^{-1(i)}\mathbf{g}^{(i)}, \tag{6.9}$$

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

The vector $\mathbf{H}^{-1}\mathbf{g}$ 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

$$\underline{\alpha}^{(i+1)} = \underline{\alpha}^{(i)} - \eta^{(i)}\mathbf{H}^{-1(i)}\mathbf{g}^{(i)}, \tag{6.10}$$

where the training rate $\eta^{(i)}$ can either set to a fixed value or found by line minimization. See Section 6.7 for a description of several one-dimensional minimization algorithms.

In Equation (6.10), the vector $\mathbf{H}^{-1}\mathbf{g}$ is now called the Newton's train direction. The sequence of points $\underline{\alpha}^{(0)}, \underline{\alpha}^{(1)}, \ldots$ can be shown here to converge to the actual solution $\underline{\alpha}^*$ from any initial point $\underline{\alpha}^{(0)}$ 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 6.2. Here improvement of the parameters is performed by obtaining first the Newton's method train direction and then a suitable training rate.

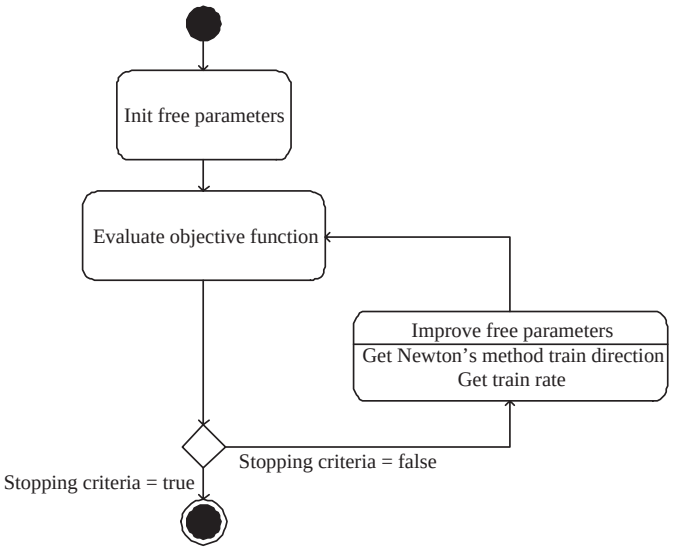

Figure 6.3: Training process with the Newton's method.

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 [45] a complete description of the Newton's method can be found. Also, a C++ implementation of this training algorithm is included within the class NewtonMethod of Flood [37]

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 [45]. 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 [16]. 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

$$\mathbf{d}_i^{\mathrm{T}} \mathbf{H} \mathbf{d}_j = 0, \tag{6.11}$$

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 s, the maximum size of a set of conjugate directions is also s.

Let denote $\mathbf{g} \equiv \nabla f(\underline{\alpha})$ and $\mathbf{h}$ the train direction vector. Then, starting with an initial parameter vector $\underline{\alpha}^{(0)}$ and an initial train direction vector $\mathbf{h}^{(0)} = -\mathbf{g}^{(0)}$, the conjugate gradient method constructs a sequence of train directions from the recurrence

$$\mathbf{h}^{(i+1)} = \mathbf{g}^{(i+1)} + \gamma^{(i)} \mathbf{h}^{(i)}, \tag{6.12}$$

for $i = 0, 1, \ldots$.

The parameters can then be improved according to the formula

$$\underline{\alpha}^{(i+1)} = \underline{\alpha}^{(i)} + \eta^{(i)} \mathbf{h}^{(i)}, \tag{6.13}$$

also for $i = 0, 1, \ldots$, and where $\eta^{(i)}$ is the train rate, which is usually found by line minimization.

The various versions of conjugate gradient are distinguished by the manner in which the parameter $\gamma^{(i)}$ is constructed. For the Fletcher-Reeves update the procedure is [20]

$$\gamma_{FR}^{(i)} = \frac{\mathbf{g}^{(i+1)} \mathbf{g}^{(i+1)}}{\mathbf{g}^{(i)} \mathbf{g}^{(i)}}, \tag{6.14}$$

where $\gamma_{FR}^{(i)}$ is called the Fletcher-Reeves parameter.

For the Polak-Ribiere update the procedure is

$$\gamma_{PR}^{(i)} = \frac{(\mathbf{g}^{(i+1)} - \mathbf{g}^{(i)}) \mathbf{g}^{(i+1)}}{\mathbf{g}^{(i)} \mathbf{g}^{(i)}}, \tag{6.15}$$

where $\gamma_{PR}^{(i)}$ is called the Polak-Ribiere parameter.

It can be shown that both the Fletcher-Reeves and the Polak-Ribiere train directions indeed satisfy Equation (6.13).

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 **h** to be down the local gradient, which is equivalent to beginning the conjugate-gradient procedure again [56].

For all conjugate gradient algorithms, the train direction is periodically reset to the negative of the gradient. The standard reset point occurs every s epochs, being s is the number of parameters in the multilayer perceptron [55].

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.

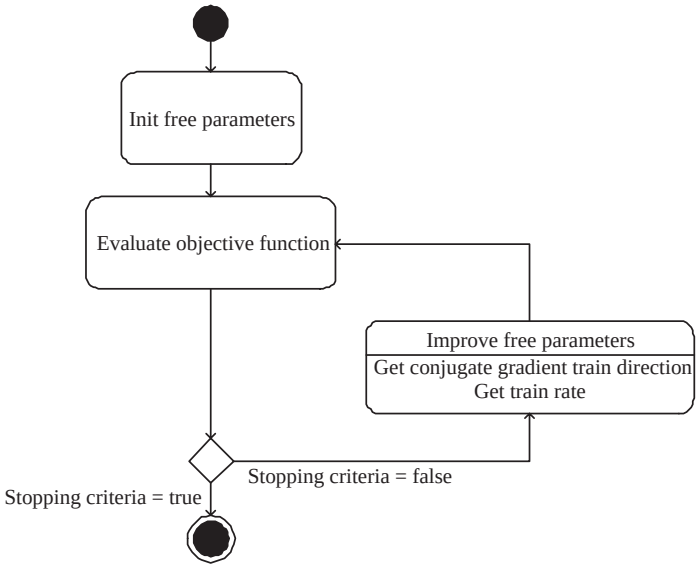

Figure 6.4: Training process with the conjugate gradient training algorithm.

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 [45]. The library Flood also implements this method in the C++ class ConjugateGradient [37].

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 (6.10), 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 Equation (6.10), 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.

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 $\mathbf{H}^{-1}$ by another matrix $\mathbf{G}$, using only the first partial derivatives of f. If $\mathbf{H}^{-1}$ is approximated by $\mathbf{G}$, the Newton formula (6.10) can be expressed as

$$\underline{\alpha}^{(i+1)} = \underline{\alpha}^{(i)} - \eta^{(i)}\mathbf{G}^{(i)}\mathbf{g}^{(i)}, \tag{6.16}$$

where $\eta^{(i)}$ can either set to a fixed value or found by line minimization.

Implementation of Equation (6.16) involves generating a sequence of matrices $\mathbf{G}$ which represent increasingly accurate approximation to the inverse Hessian $\mathbf{H}^{-1}$, 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 $\mathbf{G}$ 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 (sometimes referred to as simply Fletcher-Powel (FP) algorithm) and the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm.

The DFP algorithm is given by

$$\begin{aligned}
\mathbf{G}^{(i+1)} &= \mathbf{G}^{(i)} \\
&+ \frac{(\underline{\alpha}^{(i+1)} - \underline{\alpha}^{(i)}) \otimes (\underline{\alpha}^{(i+1)} - \underline{\alpha}^{(i)})}{(\underline{\alpha}^{(i+1)} - \underline{\alpha}^{(i)}) \cdot (\mathbf{g}^{(i+1)} - \mathbf{g}^{(i)})} \\
&+ \frac{\left[\mathbf{G}^{(i)} \cdot (\mathbf{g}^{(i+1)} - \mathbf{g}^{(i)})\right] \otimes \left[\mathbf{G}^{(i)} \cdot (\mathbf{g}^{(i+1)} - \mathbf{g}^{(i)})\right]}{(\mathbf{g}^{(i+1)} - \mathbf{g}^{(i)}) \cdot \mathbf{G}^{(i)} \cdot (\mathbf{g}^{(i+1)} - \mathbf{g}^{(i)})},
\end{aligned} \tag{6.17}$$

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

$$\begin{aligned}
\mathbf{G}^{(i+1)} &= \mathbf{G}^{(i)} \\
&+ \frac{(\underline{\alpha}^{(i+1)} - \underline{\alpha}^{(i)}) \otimes (\underline{\alpha}^{(i+1)} - \underline{\alpha}^{(i)})}{(\underline{\alpha}^{(i+1)} - \underline{\alpha}^{(i)}) \cdot (\mathbf{g}^{(i+1)} - \mathbf{g}^{(i)})} \\
&+ \frac{\left[\mathbf{G}^{(i)} \cdot (\mathbf{g}^{(i+1)} - \mathbf{g}^{(i)})\right] \otimes \left[\mathbf{G}^{(i)} \cdot (\mathbf{g}^{(i+1)} - \mathbf{g}^{(i)})\right]}{(\mathbf{g}^{(i+1)} - \mathbf{g}^{(i)}) \cdot \mathbf{G}^{(i)} \cdot (\mathbf{g}^{(i+1)} - \mathbf{g}^{(i)})} \\
&+ \left[(\mathbf{g}^{(i+1)} - \mathbf{g}^{(i)}) \cdot \mathbf{G}^{(i)} \cdot (\mathbf{g}^{(i+1)} - \mathbf{g}^{(i)})\right] \mathbf{u} \otimes \mathbf{u},
\end{aligned} \tag{6.18}$$

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

$$\begin{aligned}
\mathbf{u} &= \frac{(\underline{\alpha}^{(i+1)} - \underline{\alpha}^{(i)})}{(\underline{\alpha}^{(i+1)} - \underline{\alpha}^{(i)}) \cdot (\mathbf{g}^{(i+1)} - \mathbf{g}^{(i)})} \\
&- \frac{\mathbf{G}^{(i)} \cdot (\mathbf{g}^{(i+1)} - \mathbf{g}^{(i)})}{(\mathbf{g}^{(i+1)} - \mathbf{g}^{(i)}) \cdot \mathbf{G}^{(i)} \cdot (\mathbf{g}^{(i+1)} - \mathbf{g}^{(i)})}.
\end{aligned} \tag{6.19}$$

It has become generally recognized that, empirically, the BFGS scheme is superior than the DFP scheme [56].

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

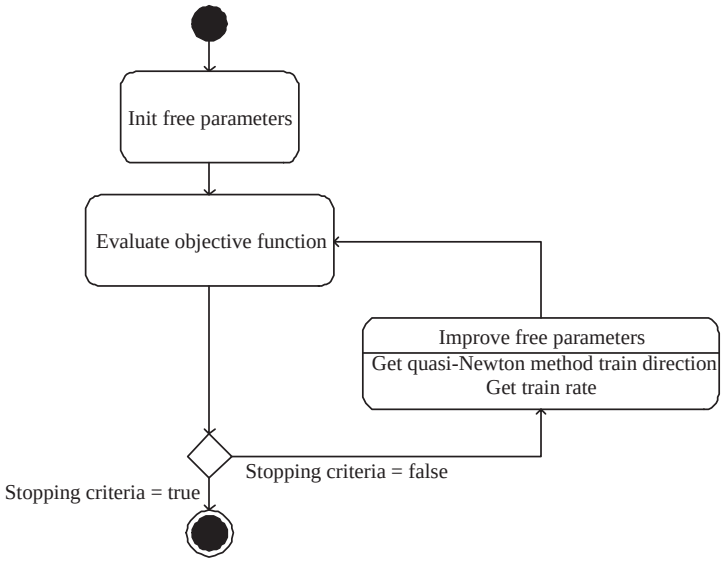

Figure 6.5: Training process with the quasi-Newton method.

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 [45]. A software implementation of the quasi-Newton method can be found in the QuasiNewtonMethod class of Flood.

6.7 One-dimensional minimization algorithms

The basic schema of most the training algorithms for the multilayer perceptron is to produce a sequence of improved approximations to the optimal parameter vector according to the following basis:

1. Start with an initial trial vector $\underline{\alpha}^{(0)}$.
2. Find a suitable train direction $\mathbf{d}^{(0)}$ in the general direction of the optimum.
3. Find an appropriate train rate $\eta^{(0)}$ for movement along that train direction.
4. Obtain the new approximation $\underline{\alpha}^{(1)}$.
5. Test whether any stopping criteria is satisfied. Otherwise, repeat step 2 onward.

The train rate in step 3 can either set to a fixed value or found by line minimization along the train direction at each epoch. In general, an optimal value for the train rate obtained by line minimization at each successive epoch is preferable.

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).

The golden section method brackets that minimum until the distance between the two outer points in the bracket is less than a defined tolerance [56].

The Brent's method performs a parabolic interpolation until the distance between the two outer points defining the parabola is less than a tolerance [56].

Flood implements the golden section and the Brent's method inside the C++ classes GradientDescent, NewtonMethod, ConjugateGradient and QuasiNewtonMethod of Flood [37].

6.8 Evolutionary algorithm

A global training algorithm for the multilayer perceptron is the evolutionary algorithm, a class of which is the 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. All is implemented in the class EvolutionaryAlgorithm of the C++ library Flood [37].

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 [54]. 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

$$\mathbf{P}^{(0)} = \begin{pmatrix} \alpha_{11}^{(0)} & \cdots & \alpha_{1s}^{(0)} \\ \cdots & \cdots & \cdots \\ \alpha_{N1}^{(0)} & \cdots & \alpha_{Ns}^{(0)} \end{pmatrix},$$

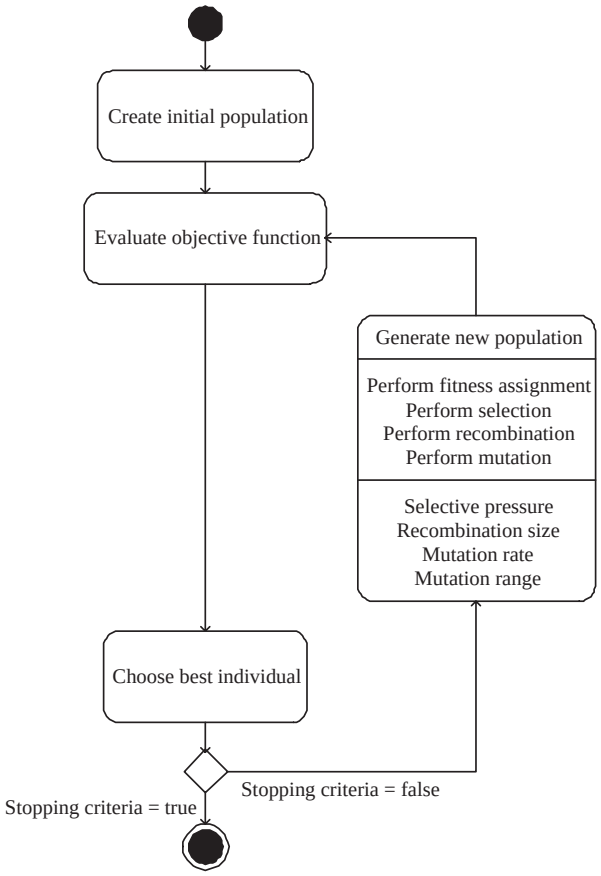

Figure 6.6: Training process with the evolutionary algorithm.

where $\mathbf{P}$ is called the population matrix. The number of individuals in the population N is called the population size.

The objective function is evaluated for all the individuals

$$\mathbf{f}^{(0)} = \{f_1^{(0)}(\underline{\alpha}_1^{(0)}), \ldots, f_N^{(0)}(\underline{\alpha}_N^{(0)})\}, \tag{6.20}$$

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

Fitness assignment

If no stopping criterium is met the generation of a new population $\mathbf{P}^{(1)}$ starts by performing fitness assignment to the old population $\mathbf{P}^{(0)}$.

$$\Phi^{(0)} = \{\Phi_1^{(0)}, \ldots, \Phi_N^{(0)}\}, \tag{6.21}$$

where Φ 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] [66].

Linear ranking assigns a fitness to each individual which is linearly proportional to its rank [54]. This operator is controlled by a single parameter called selective pressure. Linear ranking allows values for the selective pressure in the interval $[1, 2]$.

Consider N 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 $r = N$, and p the selective pressure. The fitness value for that individual is calculated as:

$$\Phi_i(r_i) = 2 - p + 2(p-1)\frac{r_i - 1}{N - 1}. \tag{6.22}$$

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,

$$\mathbf{S}^{(0)} = \{S_1^{(0)}, \ldots, S_N^{(0)}\}, \tag{6.23}$$

where $\mathbf{S}$ is called the selection vector. The elements of this vector are boolean values, that is, an individual can be either selected for recombination ($true$) or not ($false$).

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.

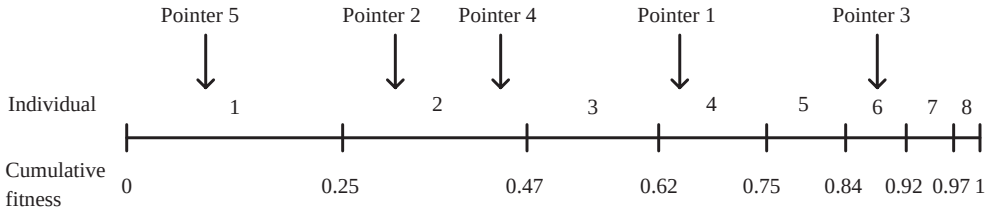

Figure 6.7: Illustration the roulette wheel selection method.

A better selection operator might be stochastic universal sampling [54]. 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 M, then the distance between the pointers are $1/M$ and the position of the first pointer is given by a randomly generated number in the range $[0, 1/M]$. Figure 6.8 illustrates this method.

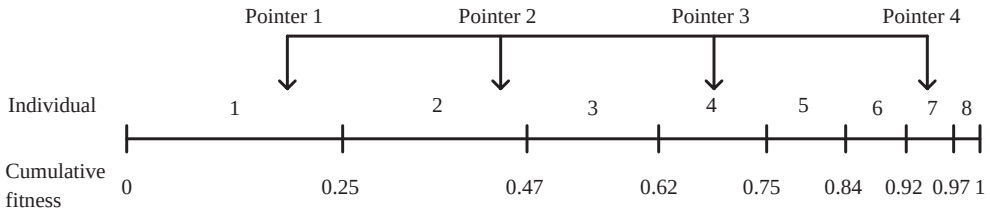

Figure 6.8: Illustration of the stochastic universal sampling selection method.

Recombination

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

$$\mathbf{P} = \begin{pmatrix} \alpha_{11} & \dots & \alpha_{1s} \\ \dots & \dots & \dots \\ \alpha_{N1} & \dots & \alpha_{Ns} \end{pmatrix}$$

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 [54]. Offspring are therefore produced according to

$$\alpha_i^{(offsprint)} = a\alpha_i^{(parent1)} + (1-a)\alpha_i^{(parent2)}, \tag{6.24}$$

for $i = 1, \dots, s$ 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 [54]. Here offspring is produced according to the rule

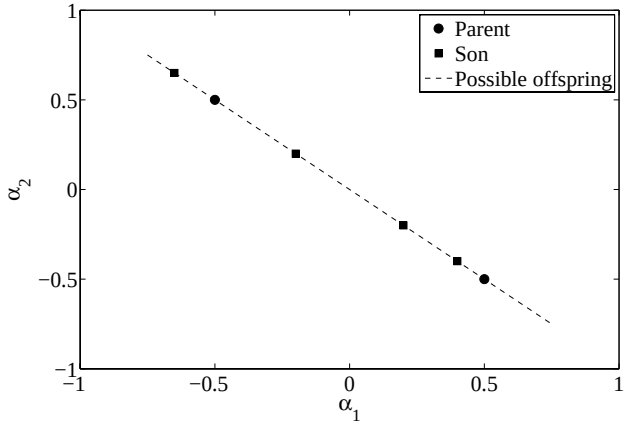

Figure 6.9: Illustration of line recombination.

$$\alpha_i^{(offspring)} = a_i \alpha_i^{(parent1)} + (1 - a_i)\alpha_i^{(parent2)}, \tag{6.25}$$

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

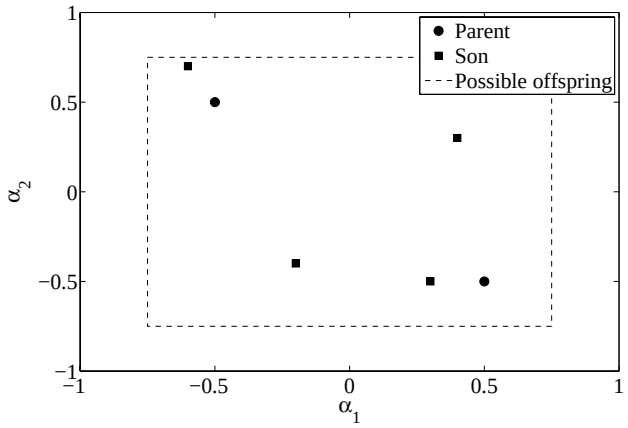

Figure 6.10: Illustration of intermediate recombination.

Mutation

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

$$\mathbf{P}^{(1)} = \begin{pmatrix} \alpha_{11}^{(1)} & \ldots & \alpha_{1s}^{(1)} \\ \ldots & \ldots & \ldots \\ \alpha_{N1}^{(1)} & \ldots & \alpha_{Ns}^{(1)} \end{pmatrix}$$

The probability of mutating a parameter is called the mutation rate and denoted p [54]. 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 α_i subject to mutation is mutated to become α_i',

$$\alpha_i' = \alpha_i + \Delta\alpha_i. \tag{6.26}$$

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, $\Delta\alpha_i$ 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.

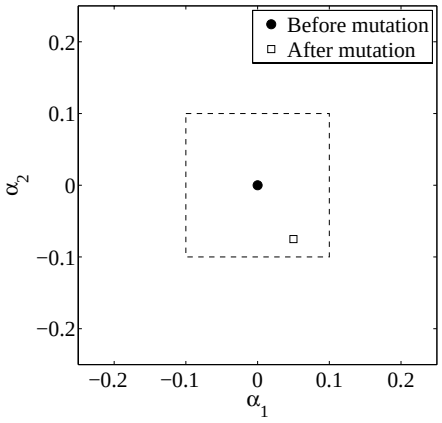

Figure 6.11: Illustration of uniform mutation.

In normal mutation, $\Delta\alpha_i$ 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.

The whole fitness assignment, selection recombination and mutation process is repeated until a stopping criterium is satisfied.

6.9 The TrainingAlgorithm classes in Flood

`Flood` includes the abstract class TrainingAlgorithm to represent the concept of training algorithm. That class contains:

1. A relationship to an objective functional object.
2. A set of training operators.
3. A set of training parameters.
4. A set of stopping criteria.

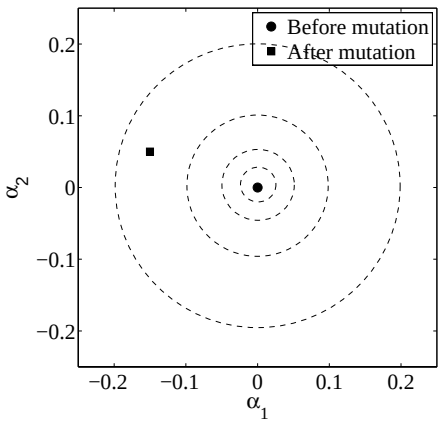

Figure 6.12: Illustration of normal mutation.

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 QuasiNewtonMethod object associated, for example, to a mean squared error object, we use the following sentence

```
QuasiNewtonMethod quasiNewtonMethod(&meanSquaredError);
```

where &meanSquaredError is a reference to a MeanSquaredError object. Note that this object must have been constructed previously.

The following sentences set some stopping criteria for the quasi-Newton method

```
quasiNewtonMethod.setEvaluationGoal(0.001);
quasiNewtonMethod.setGradientNormGoal(0.001);
quasiNewtonMethod.setMaximumTime(1000.0);
quasiNewtonMethod.setMaximumNumberOfEpochs(1000);
quasiNewtonMethod.setMinimumImprovement(1.0e-6);
```

The method train(**void**) trains a multilayer perceptron according to the quasi-Newton method.

```
quasiNewtonMethod.train();
```

We can save or load a quasi-Newton method object to or from a data file, by using the methods save(**char***) and load(**char***), respectively. See the QuasiNewtonMethodTemplate.dat file for the format of a quasi-Newton method data file in `Flood`.

To construct a EvolutionaryAlgorithm object associated to a mean squared error object we can use the following sentence

```
EvolutionaryAlgorithm evolutionaryAlgorithm(&meanSquaredError);
```

where &meanSquaredError is a reference to a MeanSquaredError object.

In order to set a new number of individuals in the population we use the method setPopulationSize(**int**).

```
evolutionaryAlgorithm.setPopulationSize(100);
```

The following sentences set some stopping criteria for the evolutionary algorithm

```
evolutionaryAlgorithm.setEvaluationGoal(0.001);
evolutionaryAlgorithm.setMaximumTime(1000.0);
evolutionaryAlgorithm.setMaximumNumberOfGenerations(1000);
```

The method train(**void**) trains a multilayer perceptron according to the evolutionary algorithm method.

```
evolutionaryAlgorithm.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 file EvolutionaryAlgorithmTemplate.dat shows the format of an evolutionary algorithm data file in `Flood`.

The source code of sample applications (main functions) with the QuasiNewtonMethod and the EvolutionaryAlgorithm classes can be found in the QuasiNewtonMethodApplication.cpp and EvolutionaryAlgorithmApplication.cpp files, respectively. Using the other training algorithm classes available in `Flood` should not involve any difficulty from that applications and the rest of this chapter.

Chapter 7

Modeling of data

Any learning task for the multilayer perceptron can be formulated as a variational problem. This statement indeed includes the traditional applications for that neural network.

In this Chapter we express the data modeling theory from a variational point of view. Actual case studies in medicine and the naval and aeronautical industries are also solved within this theory.

7.1 Problem formulation

Three main types of data modeling problems are function regression, pattern recognition and time series prediction. Some theory and typical objective functionals related to the two former are next explained.

7.1.1 Function regression

A traditional learning task for the multilayer perceptron is the function regression problem [27], which can be regarded as the problem of approximating a function from data. Here the neural network learns from knowledge represented by an input-target data set consisting of input-target examples. The targets are a specification of what the response to the inputs should be,

$$\{\mathbf{p}^{(1)}, \mathbf{t}^{(1)}\}, \{\mathbf{p}^{(2)}, \mathbf{t}^{(2)}\}, ..., \{\mathbf{p}^{(Q)}, \mathbf{t}^{(Q)}\}.$$

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.

A common feature of most input-target data sets is that the data exhibits an underlying systematic aspect, represented by some function $\mathbf{h}(\mathbf{x})$, 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. More specificially, the goal

in a function regression problem for the multilayer perceptron is to obtain a function $\mathbf{y}^*(\mathbf{x})$ which approximates the regression function $\mathbf{h}(\mathbf{x})$.

One of the most common objective functionals in function regression problems is the sum squared error, described in Section 7.1.3. Also, a very common objective functional for this type of problems is the Minkowski error, which is described in Section 7.1.5.

Two frequent problems which can appear when solving a function regression problem 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 [16]. In this way underfitting is defined as the effect of a generalization error increasing due to a too simple model, whereas overfitting is defined as the effect of a generalization error increasing due to a too complex model.

While underfitting can be prevented by simply increasing the complexity of the neural network, it is more difficult in advance to prevent overfitting. In Section 7.1.6 we introduce the regularization theory, which is a method to prevent overfitting.

7.1.2 Pattern recognition

Another traditional learning task for the multilayer perceptron is the pattern recognition (or classification) problem [27]. The task of pattern recognition can be stated as the process whereby a received pattern, characterized by a distinct set of features, is assigned to one of a prescribed number of classes. Here the neural network learns from knowledge represented by an input-target 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,

$$\{\mathbf{p}^{(1)}, \mathbf{t}^{(1)}\}, \{\mathbf{p}^{(2)}, \mathbf{t}^{(2)}\}, ..., \{\mathbf{p}^{(Q)}, \mathbf{t}^{(Q)}\}.$$

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, and it is denoted $\mathbf{h}(\mathbf{x})$.

Therefore, in order to solve a pattern recognition problem, the input space must be properly separated into regions, where each region is assigned to a class. A border between two regions is called a decision boundary. The goal in a pattern recognition problem for the multilayer perceptron is thus to obtain a function $\mathbf{y}^*(\mathbf{x})$ as an approximation of the pattern recognition function $\mathbf{h}(\mathbf{x})$.

Two of the most used objective functionals in pattern recognition problems is the sum squared error and the Minkowski error, which are described in Sections 7.1.3 and 7.1.5, respectively.

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

A method for preventing underfitting and overfitting is to use a network that is just large enough to provide an adequate fit [5]. An alternative approach to obtain good generalization is by using regularization theory, which is described in Section 7.1.6.

7.1.3 The sum squared error

As it has been said, one of the most common objective functionals for the multilayer perceptron used in function regression and pattern recognition is the sum squared error (SSE). This objective functional is measured on an input target 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 input-target data set, of a squared error defined for each sample separately. The expression for the sum squared error is given by

$$\begin{aligned} E[\mathbf{y}(\mathbf{x};\underline{\alpha})] &= \sum_{q=1}^{Q}(E^{(q)}[\mathbf{y}(\mathbf{x};\underline{\alpha})])^2 \\ &= \sum_{q=1}^{Q}\sum_{k=1}^{m}\left(y_k(\mathbf{x}^{(q)};\underline{\alpha}) - t_k^{(q)}\right)^2, \end{aligned} \tag{7.1}$$

where Q is the number of samples in the data set, n is the number of input variables and m is the number of target variables.

There are several variant objective functionals of the sum squared error (SSE). Two of the most used are the mean squared error (MSE) and the root mean squared error (RMSE). Both objective functionals have the same properties than the sum squared error and the advantage that their value do not grow with the size of the input-target data set [8].

The expression for the mean squared error is given, in terms of the sum squared error, by

$$MSE = \frac{1}{Q}SSE, \tag{7.2}$$

and the expression for the root mean squared error is given by

$$RMSE = \sqrt{\frac{1}{Q}SSE}, \tag{7.3}$$

where, in both cases, Q is the number of samples in the data set.

`Flood` includes the classes SumSquaredError, MeanSquaredError and RootMeanSquaredError, which represent the concepts of the sum squared error, mean squared error and root mean squared error functionals, respectively.

7.1.4 The normalized squared error

Another useful objective functional for data modeling problems is the normalized squared error, which takes the form

$$E[\mathbf{y}(\mathbf{x};\underline{\alpha})] = \frac{\sum_{q=1}^{Q}\|\mathbf{y}(\mathbf{x}^{(q)};\underline{\alpha}) - \mathbf{t}^{(q)}\|^2}{\sum_{q=1}^{Q}\|\mathbf{t}^{(q)} - \bar{\mathbf{t}}\|^2}. \tag{7.4}$$

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.

`Flood` also includes the class NormalizedSquaredError to represent the concept of the normalized squared error.

7.1.5 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

$$\begin{aligned} E[\mathbf{y}(\mathbf{x};\underline{\alpha})] &= \sum_{q=1}^{Q} (E^{(q)}[\mathbf{y}(\mathbf{x};\underline{\alpha})])^R \\ &= \sum_{q=1}^{Q} \sum_{k=1}^{m} |y_k(\mathbf{x}^{(q)};\underline{\alpha}) - t_k^{(q)}|^R. \end{aligned} \tag{7.5}$$

This reduces to the usual sum squared error when $R = 2$ [8].

The Minkowski error is also included in the `Flood` library, and it is implemented in the class MinkowskiError.

7.1.6 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.

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 [63]. 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 [24].

An alternative approach to obtain good generalization in a neural network is to control its effective complexity [62]. This can be achieved by choosing an objective functional which adds a regularization term Ω to the error functional E [16]. The objective functional then becomes

$$F[\mathbf{y}(\mathbf{x};\underline{\alpha})] = E[\mathbf{y}(\mathbf{x};\underline{\alpha})] + \nu\Omega[\mathbf{y}(\mathbf{x};\underline{\alpha})], \tag{7.6}$$

where the parameter ν is called the regularization term weight. The value of $\Omega[\mathbf{y}(\mathbf{x};\underline{\alpha})]$ typically depends on the mapping function $\mathbf{y}(\mathbf{x})$, and if the functional form $\Omega[\mathbf{y}(\mathbf{x};\underline{\alpha})]$ 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 parameters in the neural network divided by the number of parameters [8].

$$\Omega[\mathbf{y}(\mathbf{x};\underline{\alpha})] = \frac{1}{s}\sum_{i=1}^{s}\alpha_i^2, \tag{7.7}$$

where s 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 and less likely to overfit.

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 may get overfitting. If the regularization term weight is too large, the neural network will not adequately fit the data [16]. In this way, it is desirable to determine the optimal regularization parameters in an automated fashion. One approach is the Bayesian framework of David MacKay [46].

The Flood`Flood` library includes the class RegularizedMinkowskiSquaredError, which is a C++ implementation of the Minkowski error with parameter decay.

7.1.7 Linear regression analysis

The performance of a neural network can be measured to some extent by the sum squared error on a validation data 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 set.

This analysis leads to 3 parameters. The first two, m and b, correspond to the slope and the y-intercept of the best linear regression relating targets to neural network outputs. If we had a perfect fit (outputs exactly equal to targets), the slope would be 1, and the y-intercept would be 0. The third parameter is the correlation coefficient (R-value) between the outputs and targets. If this number is equal to 1, then there is perfect correlation between targets and outputs.

The LinearRegressionAnalysis class is included in the `Flood` library in order to perform this validation analysis.

7.1.8 Correct predictions analysis

The target values in a pattern recognition problems are discrete. Therefore a linear regression analysis is not a suitable validation technique for this kind of applications.

A more convenient method might be to provide the ratio of correct predictions made by the neural network on an independent testing set.

`Flood` includes the CorrectPredictionsAnalysis class to perform this validation technique for pattern recognition problems.

7.2 The logical operations problem

Learning logical operations is a traditional benchmark application for neural networks. Here a single multilayer perceptron is used to learn a set of logical operations. This will also put together some of the concepts described earlier.

Problem statement

In this section we will train a neural network to learn the logical operations AND, OR, NAND, NOR, XOR and XNOR. Table 7.1 shows the input-target data set for this problem.

a	b	AND	OR	NAND	NOR	XOR	XNOR
1	1	1	1	0	0	0	1
1	0	0	1	1	0	1	0
0	1	0	1	1	0	1	0
0	0	0	0	1	1	0	1

Table 7.1: Logical operations input-target data set.

The number of samples in the data set is 4. The number of input variables for each sample is 2, and the number of target variables is 6. Despite all input and target variables are boolean values, they are represented as real values.

It is always convenient to perform a linear rescaling of the training data. We can normalize the mean and the input and target data so that the minimum and maximum of all input and target data is -1 and 1, respectively.

After the neural 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 first step is to choose a network architecture to represent the logical operations function. Here a multilayer perceptron with a sigmoid hidden layer and a linear output layer is used. This is a class of universal approximator [29]. The neural network must have 2 inputs, since there are two input variables (a and b), and 6 outputs, since there are six target variables (AND, OR, $NAND$, NOR, XOR and $XNOR$). As an initial guess, we use 6 neurons in the hidden layer. This neural network can be denoted as a $2 : 6 : 6$ multilayer perceptron. It defines a family V of parameterized functions $\mathbf{y}(\mathbf{x}; \underline{\alpha})$, with $\mathbf{x} = (a, b)$ and $\mathbf{y} = (AND, OR, NAND, NOR, XOR, XNOR)$. The dimension of V is $s = 61$, which is the number of neural parameters in the network. Elements V are of the form

$$\begin{aligned} \mathbf{y}: \quad \mathbb{R}^2 &\rightarrow \mathbb{R}^6 \\ \mathbf{x} &\mapsto \mathbf{y}(\mathbf{x}; \underline{\alpha}), \end{aligned}$$

Figure 7.1 is a graphical representation of this neural network.

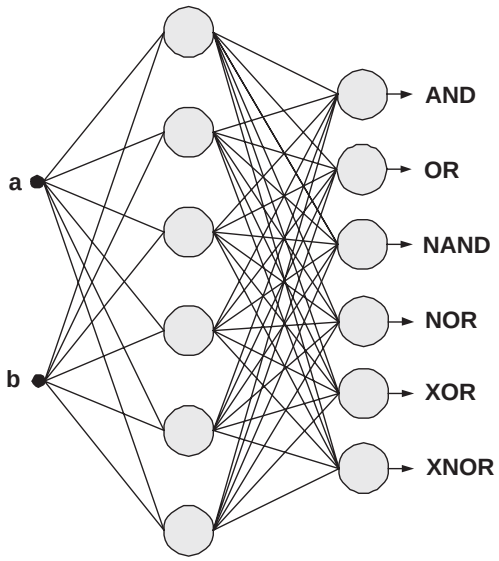

Figure 7.1: Network architecture for the logical operations problem.

The neural parameters in the neural network (biases and synaptic weights) are initialized at random with values chosen from a normal distribution with mean 0 and standard deviation 1.

Formulation of variational problem

The second step is to assign an objective functional to the multilayer perceptron, in order to formulate the variational problem. We can choose here the mean squared error.

The variational statement of this logical operations problem is then to find a function $\mathbf{y}^*(\mathbf{x}; \underline{\alpha}^*)$ for which the functional

$$E[\mathbf{y}(\mathbf{x}; \underline{\alpha})] = \frac{1}{4} \sum_{q=1}^{4} \left(\mathbf{y}(\mathbf{x}^{(q)}; \underline{\alpha}) - \mathbf{t}^{(q)} \right)^2, \tag{7.8}$$

defined on V, takes on a minimum value.

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. In this

example, we set the training algorithm to stop when one of the following stopping criteria is satisfied:
- The mean squared error evaluation $e(\underline{\alpha})$ is less than 0.01.
- The norm of the objective function gradient $\|\nabla e(\underline{\alpha})\|$ falls below 0.01.
- A maximum computing time of $100s$ has been exceed.
- The training algorithm has performed a maximum number of 100 epochs.

It is very easy for gradient algorithms to get stuck in local minima when learning multilayer perceptron neural parameters. 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. Figure 7.2 shows the objective and the gradient norm versus the training epoch.

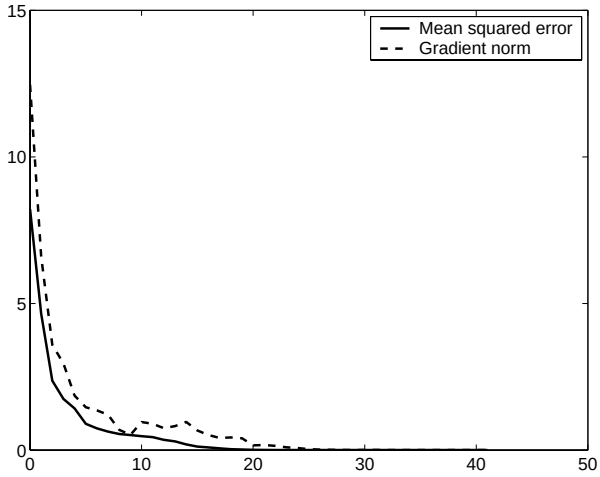

Figure 7.2: Training process for the logical operations problem.

Once the neural 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, respectively.

Validation of the results

The last step is to validate the performance of the trained neural network. Here the multilayer perceptron has correctly learned all the AND, OR, $NAND$, NOR, XOR and $XNOR$ logical operations. As soon as the multilayer perceptron results have been validated, we can save the neural network to a data file for future use.

7.3 The sinus problem

In this section a simple function regression problem with just one input and one output variables is solved. This application illustrates the problems of underfitting and overfitting, very common in the application of neural networks.

Problem statement

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

$$h(x) = 0.5 + 0.4sin(2\pi x), \tag{7.9}$$

by sampling it at equal intervals of x and then adding random noise with a Gaussian distribution having standard deviation $\sigma = 0.05$.

A common feature of most training data sets is that the data exhibits an underlying systematic aspect, represented in this case by the function $h(x)$, 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 training data set. We will therefore be interested in obtaining a function $y(x)$ close to the regression function $h(x)$. Figure 7.3 shows this training data set, together with the function $h(x)$.

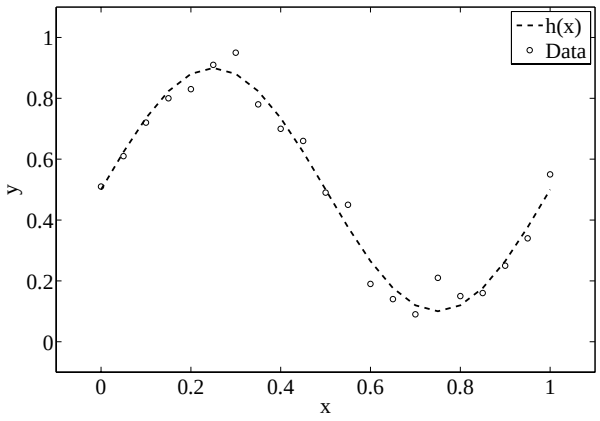

Figure 7.3: The sinus problem statement.

The goal in this function regression problem for the multilayer perceptron is to obtain a function $y^*(x;\underline{\alpha}^*)$ which approximates the regression function $h(x)$. The next sections describe some common objective functionals for the multilayer perceptron which might be able to approximate the regression function.

Table 7.2 shows the numbers of samples, input variables and target variables, Table 7.3 shows the actual input-target data set, and Table 7.4 shows the basic statistics of the data.

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

Number of samples = 21
Number of input variables = 1
Number of target variables = 1

Table 7.2: Properties of the input-target data set for the sinus problem.

x	y
0.00	0.51
0.05	0.61
0.10	0.72
0.15	0.80
0.20	0.83
0.25	0.91
0.30	0.95
0.35	0.78
0.40	0.70
0.45	0.66
0.50	0.49
0.55	0.45
0.60	0.19
0.65	0.14
0.70	0.09
0.75	0.21
0.80	0.15
0.85	0.16
0.90	0.25
0.95	0.34
1.00	0.55

Table 7.3: Input-target data set for the sinus problem.

$\min(x) = 0$
$\max(x) = 1$
$\rho(x) = 0.5$
$\sigma(x) = 0.3102$
$\min(y) = 0.09$
$\max(y) = 0.95$
$\rho(y) = 0.4995$
$\sigma(y) = 0.2814$

Table 7.4: Basic statistics for the sinus input-target data set.

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.

In order to compare the performance of different network architectures, 1, 2 and 10 neurons in the hidden layer will be used. These neural networks can denoted as $1:1:1$, $1:2:1$ and $1:10:1$ multilayer perceptrons. They define three different families V_1, V_2 and V_{10} of parameterized functions $y_1(x;\underline{\alpha}_1)$, $y_2(x;\underline{\alpha}_2)$ and $y_{10}(x;\underline{\alpha}_{10})$ of dimensions $s_1 = 4$, $s_2 = 7$ and $s_{10} = 31$, which are the number of neural parameters in each multilayer perceptron. Figures 7.4, 7.5and 7.6 are graphical representations of the $1:1:1$, $1:2:1$ and $1:10:1$ network architectures, respectively. As it will be shown, the first one is undersized, the second one has a correct size and the third one is oversized.

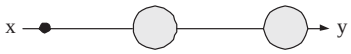

Figure 7.4: An undersized network architecture for the sinus problem.

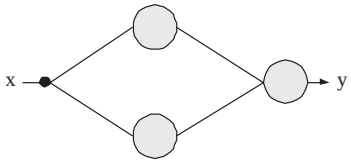

Figure 7.5: A medium sized network architecture for the sinus problem.

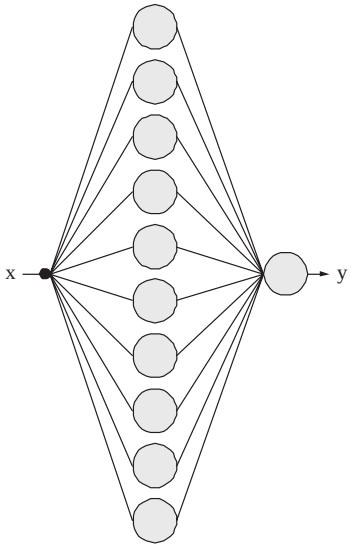

Figure 7.6: An oversized network architecture for the sinus problem.

All that 3 neural networks 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 mean squared error defined by Equation (7.2).

The variational statement of the function regression problems being considered here is then to find vectors of neural parameters $\underline{\alpha}_i^* \in \mathbb{R}^{s_i}$ that define functions $y_i^*(x; \underline{\alpha}_i^*) \in V_i$ for which the functionals defined on V_i

$$E_i[y_i(x; \underline{\alpha}_i)] = \frac{1}{Q} \sum_{q=1}^{Q} \left(y_i(x^{(q)}; \underline{\alpha}_i) - t^{(q)} \right)^2, \quad i = 1, 2, 10, \tag{7.10}$$

take on a minimum value.

Evaluation of the objective functionals in Equation (7.10) just require explicit expressions for the function represented by the different multilayer perceptrons. This is given in Section 4.

On the other hand, evaluation of the objective function gradient vectors $\nabla e_i(\underline{\alpha}_i), i = 1, 2, 10$, 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 functions $e_i(\underline{\alpha}_i), i = 1, 2, 10$ 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 when it cannot follow anymore with the optimization procedure. At that point, the line search method gives zero train rate for a gradient descent train direction.

Table 7.5 shows the training results for the three neural networks considered here. That table depicts a training error decreasing with the network complexity, that is, the more hidden neurons in the multilayer perceptron, the less training error.

h	1	2	10
Training error	0.197	0.025	0.001

Table 7.5: Training results for the sinus problem.

Figure 7.7 shows an under-fitting case for the function regression problem formulated in this section. In this case we have used multilayer perceptron with just one neuron in the hidden layer. The model is too simple to produce an adequate fit. Figure 7.8 shows a correct model for the regression function. Here a correct neural network size with two hidden neurons has been used. Figure refOverfittingSinusProblem shows an over-fitting case. 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. In this case we have used the multilayer perceptron with 10 neurons in the hidden layer. The model is too complex to produce an adequate fit.

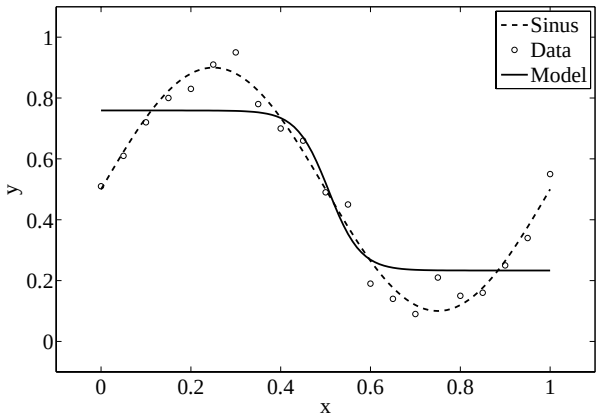

Figure 7.7: Undersized neural network results to the sinus problem.

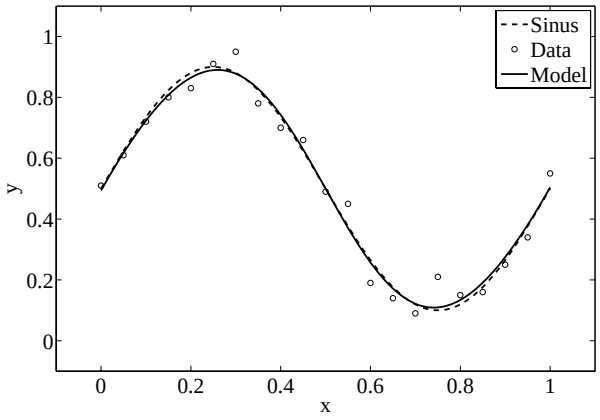

Figure 7.8: Medium-sized neural network results to the sinus problem.

7.4 The Pima Indians diabetes problem

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 [52]. 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 7.6 summarizes the input-target data set information, while tables 7.7 and 7.8 depict the input and target variables information, respectively.

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

Table 7.6: Input-target data set information.

In order to validate the results, we divide the input target data set into training and validation subsets. The first three fourths of the data will be assigned for training and the last fourth for validation. Table 7.9 summarizes the training and validation data sets information.

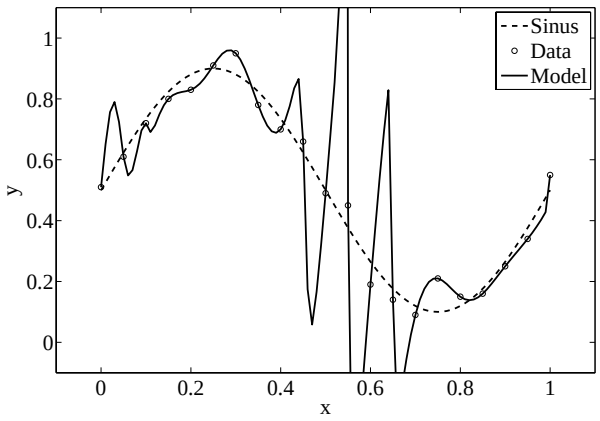

Figure 7.9: Oversized neural network results to the sinus problem.

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 7.7: Input variables information.

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

Table 7.8: Target variables information.

Number of samples for training:	576
Number of samples for validation:	192

Table 7.9: Training and validation 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 [29]. 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 $y(\mathbf{x};\underline{\alpha})$ of dimension $s = 61$, which is the number of free parameters. Elements V are of the form

$$\begin{aligned} y: \quad \mathbb{R}^8 &\rightarrow \mathbb{R} \\ \mathbf{x} &\mapsto y(\mathbf{x};\underline{\alpha}). \end{aligned}$$

Figure 7.10 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

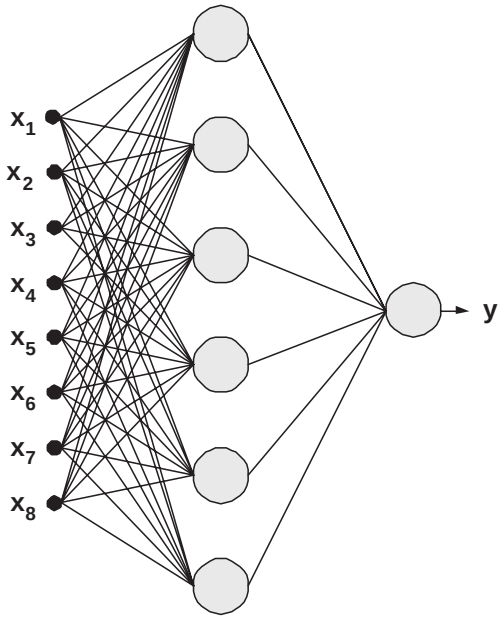

Figure 7.10: Network architecture for the pima indians diabetes problem.

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 $\mathbf{y}^*(\mathbf{x}; \underline{\alpha}^*)$ for which the functional

$$E[y(\mathbf{x}; \underline{\alpha})] = \frac{1}{576} \overset{576}{(}$$

Validation 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 quality of a prediction in a classification problem is defined as the ratio between right and wrong answers. The quality provided by this method is around 75%. Trying other network architectures with more neurons in the hidden layer does not improve the quality.

Once the generalization performance of the multilayer perceptron has been validated, the neural network can be saved for future use.

The whole source code of this sample application is listed in the PimaIndiansDiabetesApplication.cpp file.

7.5 The residuary resistance of sailing yachts problem

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. Both the data and the source code for this problem can be found within the Flood library [37]. The results from this Section were published in [53].

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 [23]. The Delft series are widely used in the sailing yacht industry.

In this work we present a neural networks approach to residuary resistance of sailing yachts prediction. Here a multilayer perceptron has been 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 [23]. 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 ($1000 \cdot R_R/\triangle_c$).

Table 7.10 lists some basic statistics of these experiments. They include mean (ρ), standard deviation (σ), minimum value (min) and maximum value (max).

	LCB	C_p	$L_{WL}/\nabla_c^{1=3}$	B_{WL}/T_C	L_{WL}/B_{WL}	F_N	$(1000 \cdot R_R/\triangle_c)$
ρ	-2.38636	0.562955	4.78955	3.92455	3.20818	0.2875	10.4956
σ	1.51312	0.0220637	0.25147	0.54449	0.245434	0.100942	15.2083
min	-5	0.53	4.34	2.81	2.76	0.125	0.01
max	0	0.6	5.14	5.35	3.64	0.45	62.42

Table 7.10: Basic data set statistics in the yacht resistance problem.

The Delft series

The Delft series are a method for predicting the residuary resistance per unit weight of displacement of the canoe body in a sailing yacht. This model was developed from the Delf data set [23].

The Delft series are expressed as a set of 14 polynomials of the form

$$\left(\frac{1000 \cdot R_R}{\triangle_c}\right)_i = A_{i0} + A_{i1}C_p + A_{i2}C_p^2 + A_{i3}LCB + A_{i4}LCB^2 + A_{i5}\frac{B_{WL}}{T_C} + A_{i6}\frac{L_{WL}}{\nabla_c^{1=3}} \quad (7.12)$$

for $i = 1, \ldots, 14$, and where the subindex i represents discrete values for the Froude number. The values of the coefficients A_{ij} can be found in [23].

Although the Delft series seem to track the experimental residuary resistances quite well, a standard data modeling procedure was not performed here when they were created. Indeed, the whole Delft data set was used for adjusting the polynomials (7.12), and no validation with an independent test set was done here. Also, a residuary resistance model allowing continuous values for the Froude number would be very desirable.

Neural networks approach

Here a multilayer perceptron is trained to learn the relationship between the input and the target variables in the Delft data set, and to make good predictions for new data.

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 7.11 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 7.11: 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 7.11.

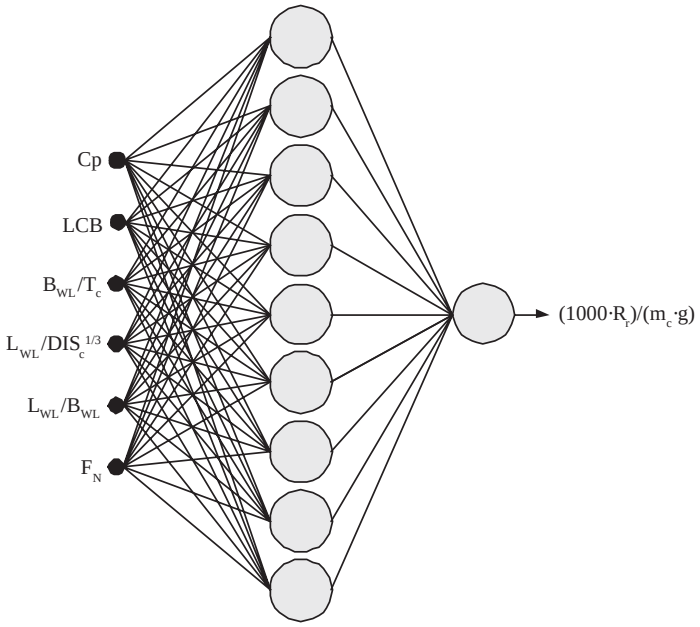

Figure 7.11: Network architecture for the yacht resistance problem.

The family of functions spanned by the neural network in Figure 7.11 can be denoted V and it is of dimension $s = 73$, the number of parameters.

Let denote

$$\mathbf{x} = (LCB, C_p, L_{WL}/\nabla_c^{1=3}, B_{WL}/T_C, L_{WL}/B_{WL}, F_N) \tag{7.13}$$

and

$$y = 1000 \cdot R_r / \triangle_c. \tag{7.14}$$

The elements of V are thus written

$$\begin{array}{rccc} y: & \mathbb{R}^6 & \rightarrow & \mathbb{R} \\ & \mathbf{x} & \mapsto & y(\mathbf{x}; \underline{\alpha}), \end{array}$$

where

$$y(\mathbf{x}; \underline{\alpha}) = \sum_{j=0}^{9} \alpha_j^{(2)} \tanh \left(\sum_{i=0}^{6} \alpha_{ji}^{(1)} x_i \right). \tag{7.15}$$

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 training data set is preprocessed with the means and the standard deviations of Table 7.10.

The variational statement of the function regression problem considered here is then to find a function $y^*(\mathbf{x}; \underline{\alpha}^*) \in V$ for which the functional

$$E[y(\mathbf{x}; \underline{\alpha})] = \frac{\sum_{q=1}^{Q} \|y(\mathbf{x}^{(q)}; \underline{\alpha}) - t^{(q)}\|^2}{\sum_{q=1}^{Q} \|t^{(q)} - \bar{t}\|^2}, \tag{7.16}$$

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 7.12 and 7.13, respectively. Note that these plots have a logarithmic scale for the Y-axis.

Table 7.12 shows the training results for this application. Here N is the number of epochs, CPU the training time in a laptop AMD 3000, $\|\underline{\alpha}^*\|$ the final parameter vector norm, $e(\underline{\alpha}^*)$ the final normalized squared error and $\|\nabla e(\underline{\alpha}^*)\|$ 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.

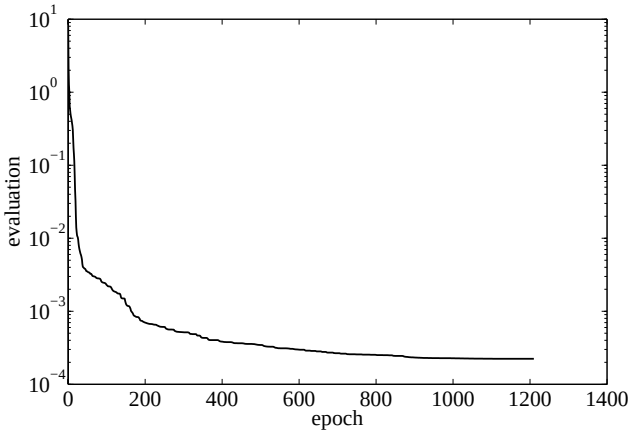

Figure 7.12: Evaluation history for the yacht resistance problem.

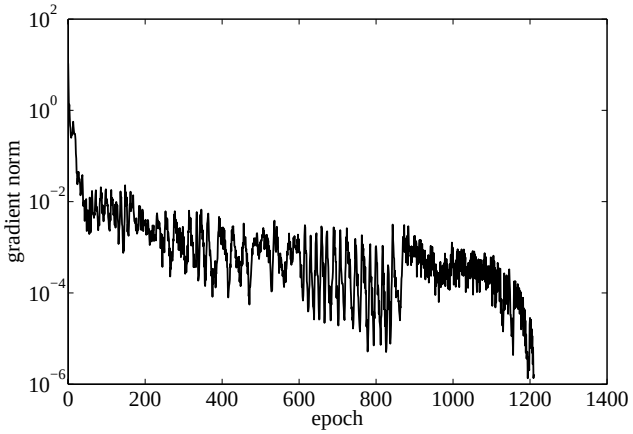

Figure 7.13: Gradient norm history for the yacht resistance problem.

N	=	1210
CPU	=	$348s$
$\|\underline{\alpha}^*\|$	=	720
$e(\underline{\alpha}^*)$	=	0.000223
$\|\nabla e(\underline{\alpha}^*)\|$	=	$1.648 \cdot 10^{-6}$

Table 7.12: Training results for the yacht resistance problem.

The explicit expression for the residuary resistance model obtained by the neural network is

$$x_1 = \frac{x_1 + 2.38182}{1.51322},$$

$$
\begin{aligned}
x_2 &= \frac{x_2 - 0.564136}{0.02329}, \\
x_3 &= \frac{x_3 - 4.78864}{0.253057}, \\
x_4 &= \frac{x_4 - 3.93682}{0.548193}, \\
x_5 &= \frac{x_5 - 3.20682}{0.247998}, \\
x_6 &= \frac{x_6 - 0.2875}{0.100942}, \\
y &= 155.425 \\
&+ 63.2639 \tanh(-3.2222 + 0.0613793x_1 + 0.112065x_2 + 0.292097x_3 \\
&- 0.172921x_4 - 0.277616x_5 + 0.569819x_6) \\
&+ 91.0489 \tanh(-147.226 - 75.3342x_1 + 24.7384x_2 + 15.5625x_3 \\
&- 82.6019x_4 - 88.8575x_5 + 1.03963x_6) \\
&+ 0.00875896 \tanh(-77.0309 - 156.769x_1 - 244.11x_2 + 62.4042x_3 \\
&+ 70.2066x_4 + 12.1324x_5 - 76.0004x_6) \\
&+ 1.59825 \tanh(-2.94236 - 0.0526764x_1 - 0.21039x_2 - 0.266784x_3 \\
&+ 0.131973x_4 + 0.317116x_5 + 1.9489x_6) \\
&- 0.0124328 \tanh(-207.601 - 210.038x_1 + 99.7606x_2 + 106.485x_3 \\
&+ 252.003x_4 - 100.549x_5 - 51.3547x_6) \\
&+ 0.026265 \tanh(-17.9071 - 11.821x_1 + 5.72526x_2 - 52.2228x_3 \\
&+ 12.1563x_4 + 56.2703x_5 + 56.7649x_6) \\
&+ 0.00923066 \tanh(69.9392 - 133.216x_1 + 70.5299x_2 - 21.4377x_3 \\
&+ 47.7976x_4 + 15.1226x_5 + 100.747x_6) \\
&- 0.215311 \tanh(4.54348 - 1.11096x_1 + 0.862708x_2 + 1.61517x_3 \\
&- 1.11889x_4 - 0.43838x_5 - 2.36164x_6) \\
&+ 0.010475 \tanh(23.4595 - 223.96x_1 - 43.2384x_2 + 13.8833x_3 \\
&+ 75.4947x_4 - 7.87399x_5 - 200.844x_6), \\
y^*(\mathbf{x}; \underline{\alpha}^*) &= 10.4954 + 15.1605y^*(\mathbf{x}; \underline{\alpha}^*).
\end{aligned}
$$

Validation of results

A possible validation 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 7.13 shows the three parameters given by this validation analysis.

a	=	0.110
b	=	0.975
R^2	=	0.958

Table 7.13: Linear regression analysis parameters for the yacht resistance problem.

Figure 7.14 illustrates a graphical output provided by this validation 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.

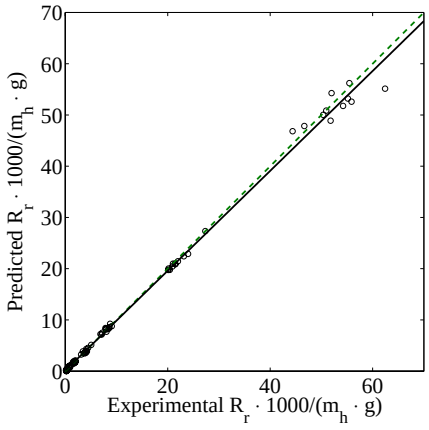

Figure 7.14: Linear regression analysis plot for the yacht resistance problem.

From Table 7.13 and Figure 7.14 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.

Conclusions

A neural network model has been developed for estimating the residuary resistance of sailing yachts, and it was found that the quality of the prediction on a testing data set is very satisfactory over the entire range of data. Moreover, the approach presented here allows continuous values for the Froude number, while the Delf series do not.

7.6 The airfoil self-noise problem

A neural network model for airfoil self-noise prediction has been created using a noise spectrum database collected from a family of NACA 0012 airfoils. This application is included within the Flood library [37] and it has been published as [34].

Introduction

The noise generated by an aircraft is an efficiency and environmental matter for the aerospace industry. NASA have issued a mandate to reduce the external noise generated by the whole airframe of an aircraft by 10 decibels (dB) in the near term future [36].

An important component of the total airframe noise is the airfoil self-noise, which is due to the interaction between an airfoil blade and the turbulence produce in its own boundary layer and near wake.

There have been many attempts to predict the noise of an airfoil. Howe [30] reviewed the various methodologies and grouped the different approaches into three groups: (i) theories based upon the Lighthill acoustic analogy [35], (ii) theories based on the solution of special problems approximated by the linearized hydrodynamics equations [1], and (iii) semi-empirical models [9].

The model of Brooks, Pope and Marcolini [9], referred here as the BPM model, is a semi-empirical approach. It is formulated from an extensive set of acoustic wind tunnel tests performed by NASA, upon different chord length NACA 0012 airfoil sections. The BPM model is used to predict the self-generated noise of an airfoil blade encountering smooth flow as a function of the angle of attack, the free-stream velocity and the geometric parameters of the airfoil. Formulated in 1989, the BPM model is still utilized nowadays.

However, it can be shown that the BPM model does not predict accurately the SPL variable. For example, it substantially underestimates the airfoil self-noise at low values. In this regard, a new, more accurate, model to predict the SPL generated by an airfoil is necessary.

Technologies based on neural networks are currently being developed which may assist in addressing a wide range of problems in aeronautics. They have already been successfully applied to a range of aeronautical problems in fault diagnostics, modeling and simulation and control systems [19].

In this work a neural networks approach to airfoil self-noise prediction is presented. After reviewing the prediction quality of all the configurations tested, it is shown that the neural network model provides good results for an independent testing data set across the entire range of sound pressure level data.

Experimental data

The self-noise data set used in this work was processed by NASA in 1989 [9], and so it is referred here to as the NASA data set. It was obtained from a series of aerodynamic and acoustic tests of two and three-dimensional airfoil blade sections conducted in an anechoic wind tunnel. A complete description of these experiments is reported in [9].

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

The aim of the acoustic measurements was to determine spectra for self-noise from airfoils encountering smooth airflow. These spectra are affected by the presence of extraneous contributions, and some processing of the data was needed. The results were presented as 1/3-octave spectra.

In that way, the NASA data set contains 1503 entries and consists of the following variables:

1. Frequency, $f\,[Hz]$.
2. Angle of attack, $\alpha\,[^{\circ}]$.
3. Chord length, $c\,[m]$.
4. Free-stream velocity, $U\,[ms^{-1}]$.
5. Suction side displacement thickness, $s\,[m]$.
6. Scaled sound pressure level, $SPL_{1=3}\,[dB]$.

Here the suction side displacement thickness was determined using an expression derived from boundary layer experimental data from [9].

Table 7.14 depicts some basic statistics of the NASA data set. Here ρ is the mean, σ the standard deviation, min the minimum and max the maximum.

Variable	$f\,[Hz]$	$\alpha\,[^\circ]$	$c\,[m]$	$U\,[ms^{-1}]$	$s\,[m]$	$SPL_{1=3}\,[dB]$
ρ	2886.38	6.7823	0.136548	50.8607	0.0111399	124.836
σ	3152.57	5.91813	0.0935407	15.5728	0.0131502	6.89866
min	200	0	0.0254	31.7	0.000400682	103.38
max	20000	22.2	0.3048	71.3	0.0584113	140.987

Table 7.14: Basic input-target data set statistics in the airfoil noise problem.

The Brooks-Pope-Marcolini (BPM) model

The BPM model is a semi-empirical approach based on previous theoretical studies and calibrated using the NASA data set [9].

The airfoil self-noise is the total noise produced when an airfoil encounters smooth non-turbulent inflow. In this way Brooks et al. identified and modeled five self-noise mechanisms due to specific boundary-layer phenomena [9]: (i) boundary-layer turbulence passing the training edge, (ii) separated-boundary-layer and stalled-airfoil flow, (iii) vortex shedding due to laminar-boundary-layer instabilities, (iv) vortex shedding from blunt trailing edges, and (v) turbulent vortex flow existing near the tips of lifting blades. This five noise generation mechanisms are illustrated in Figure 7.15.

The total sound pressure level is determined by summing up the contributions from each of the different phenomena enumerated above. The particular noise mechanisms which contribute to the total sound pressure level are dependent on the flow conditions upon the airfoil. In the Brooks model there are two distinct boundary layer conditions: tripped and untrippred (or natural). Depending on which is present the total self-noise expression has a different form.

For the tripped boundary layer condition the laminar boundary layer noise mechanism cannot occur, as tripping the boundary layer causes it to become turbulent. Therefore its contribution to the total sound pressure level is ignored, and only the trailing edge noise is evaluated. In the BPM model the trailing edge noise consists of noise generated by separation (SPL), and pressure and suction side effects (SPL_P and SPL_S). The total self-noise for the tripped boundary layer condition is as follows,

$$SPL_{Total} = 10\log\left(10^{SPL_{P}=10} + 10^{SPL_{S}=10} + 10^{SPL_{\alpha}=10}\right). \tag{7.17}$$

Conversely, if the boundary layer is untripped, then the laminar boundary layer vortex shedding noise mechanism must also be considered. This contribution is termed SPL_{LBL-VS}. The total sound pressure level for the untripped boundary layer condition is then given by the following equation,

$$SPL_{Total} = 10\log\left(10^{SPL_{P}=10} + 10^{SPL_{S}=10} + 10^{SPL_{\alpha}=10} + 10^{SPL_{LBL-VS}=10}\right). \tag{7.18}$$

In this case study we will only deal with the tripped boundary layer condition.

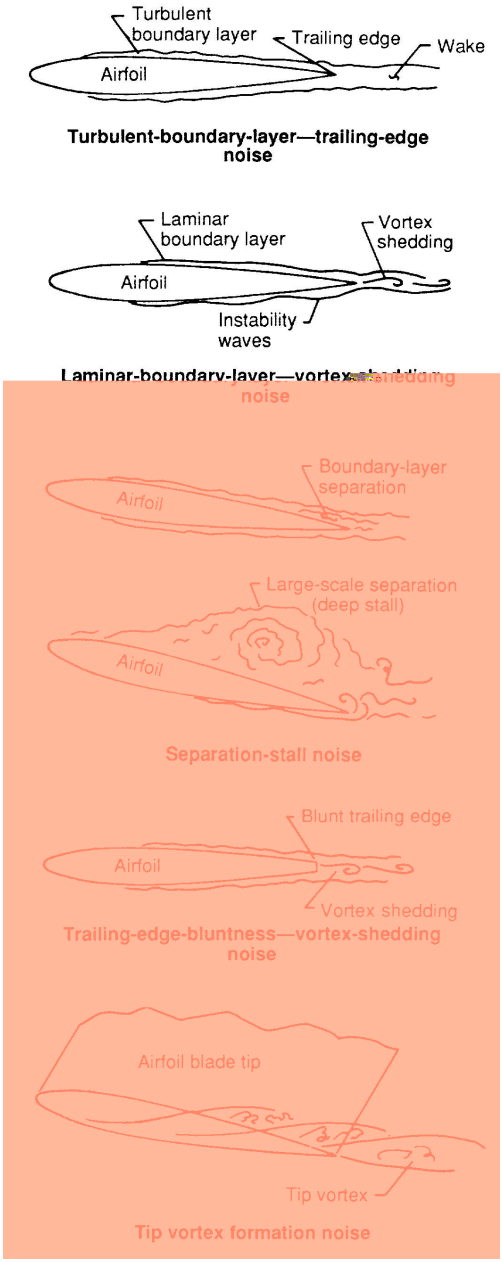

Figure 7.15: Airfoil self-noise mechanisms.

Neural networks approach

In this Section, a noise prediction model for a turbulent boundary layer noise mechanism has been created using a multilayer perceptron trained to learn the underlying aspects of the NASA data set.

Selection of function space

Here a multilayer perceptron with a sigmoid hidden layer and a linear output layer is used to predict airfoil self-noise ($SPL_{1=3}$) as a function of frequency (f), angle of attack (α), chord length (c), suction side displacement thickness (s) and free-stream speed (U). Therefore, this neural network must have 5 inputs, f, α, c, s and U, and 1 output neuron, $SPL_{1=3}$.

The number of neurons in the hidden layer will define the degree of complexity of the airfoil self-noise model, and it needs to be chosen carefully. Indeed, two frequent problems which can appear when solving a data modeling problem are called under-fitting and over-fitting. The best generalization is achieved by using a model whose complexity is the most appropriate to produce an adequate fit of the data [16]. In this way under-fitting is defined as the effect of a generalization error increasing due to a too simple model, whereas over-fitting is defined as the effect of a generalization error increasing due to a too complex model.

In this way, and in order to draw a correct network architecture for this problem, the NASA data set was first split into three, training, validation and testing, subsets. They represent 50%, 25% and 25% of the full data set, respectively. Therefore the training data set contains 752 samples, the validation data set contains 376 samples and the testing data set contains 375 samples.

Three different neural networks with 6, 9 and 12 neurons in the hidden layer are then trained with the normalized squared error objective functional and the quasi-Newton method training algorithm. The generalization performance of each single neural network was then assessed by measuring the error between the outputs from the trained neural networks and the target values in the validation data set. This error can indicate if the current model is under or over-fitting the data. Table 7.15 shows the training and the validation errors for the three different network architectures considered. 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. Indeed, the lowest model complexity seems to produce under-fitting, while the highest model complexity seems to produce over-fitting.

Number of hidden neurons	6	9	12
Training error	0.122405	0.082553	0.060767
Validation error	0.131768	0.116831	0.129884

Table 7.15: Training and validation errors in the airfoil noise problem.

From these results follows that the optimal network architecture for this problem is that including 9 hidden neurons. This neural network can be denoted as a $5:9:1$ multilayer perceptron, and it is depicted in Figure 7.16.

This multilayer perceptron spans a function space V of dimension $s = 64$. If we denote $\mathbf{x} = (f, \alpha, c, U, s)$ and $y = SPL_{1=3}$ the functions belonging to that family are of the form

$$\begin{array}{rccc} y: & \mathbb{R}^5 & \to & \mathbb{R} \\ & \mathbf{x} & \mapsto & y(\mathbf{x}; \underline{\alpha}), \end{array}$$

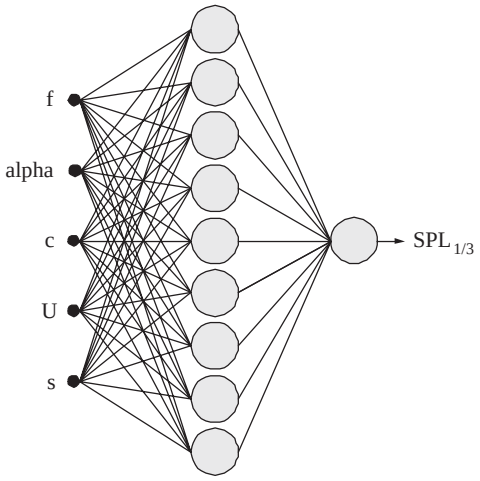

Figure 7.16: Optimal network architecture for the airfoil self-noise problem.

where

$$y(\mathbf{x};\underline{\alpha}) = \sum_{j=0}^{9} \alpha_{kj}^{(2)} \tanh\left(\sum_{i=0}^{5} \alpha_{ji}^{(1)} x_i\right). \tag{7.19}$$

Here the parameter vector $\underline{\alpha}$ is randomly initialized.

Formulation of variational problem

First, the training data set is preprocessed with the means and standard deviations of the NASA data set in Table 7.14.

A statistical analysis of the NASA data set did not show a significant presence of outliers. On the other hand, the optimal network architecture for the validation data has been selected, which means that the multilayer perceptron will not over-fit the data. In this way the simple normalized squared error is used instead the more complex Minkowski error or Bayesian regularization.

Therefore, the variational statement of this problem is to find a function $y^*(\mathbf{x};\underline{\alpha}^*) \in V$ for which the functional

$$E[y(\mathbf{x};\underline{\alpha})] \quad = \quad \frac{\sum_{q=1}^{Q} \|y(\mathbf{x}^{(q)};\underline{\alpha}) - t^{(q)}\|^2}{\sum_{q=1}^{Q} \|t^{(q)} - \bar{t}\|^2}, \tag{7.20}$$

defined on V, takes on a minimum value.

The objective function gradient $\nabla e(\underline{\alpha})$ is computed with a back-propagation algorithm for the normalized squared error.

Solution of reduced function optimization problem

The training algorithm used to optimize the associated objective function is a quasi-Newton method with BFGS train direction and Brent train rate.

The training process is set to stop when improvement between two successive epochs is less than 10^{-12}. Figures 7.17 and 7.17 show the history of this process for the evaluation and the gradient norm, respectively. Note that these plots have a logarithmic scale for the Y-axis. The evaluation history shows a significant reduction of this variable during the first few epochs and a posterior stabilization. On the other hand, during the training process the gradient norm decreases in several orders of magnitude. These two signs demonstrate that the training algorithm has certainly reached a minimum.

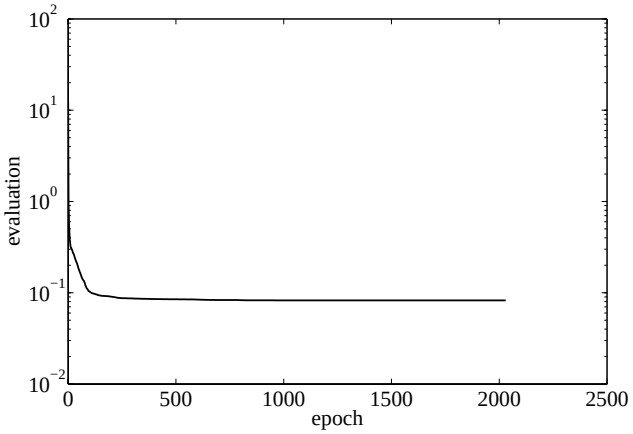

Figure 7.17: Evaluation history for the airfoil self-noise problem problem.

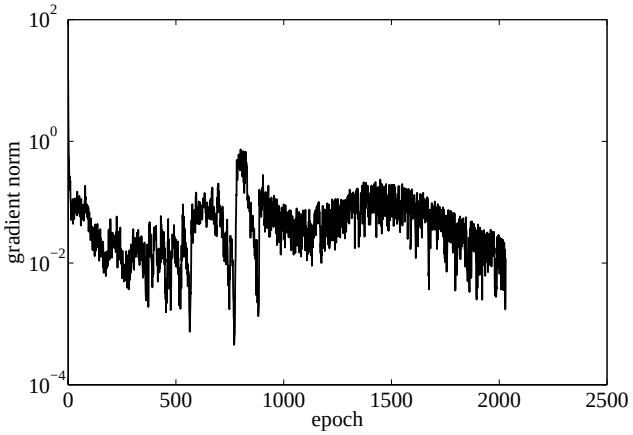

Figure 7.18: Gradient norm history for the airfoil self-noise problem.

Some final results from training are listed in Table 7.16. Here N is the number of epochs, CPU the computing time in a laptop AMD 3000, $\|\underline{\alpha}^*\|$ the final parameters norm, $e(\underline{\alpha}^*)$ the final normalized squared error and $\|\nabla e(\underline{\alpha}^*)\|$ its gradient norm.

The explicit expression of the neural network model for airfoil self-noise prediction

N	=	2031
CPU	=	$2218s$
$\|\underline{\alpha}^*\|$	=	11257.4
$e(\underline{\alpha}^*)$	=	0.082553
$\|\nabla e(\underline{\alpha}^*)\|$	=	0.011912

Table 7.16: Training results for the airfoil self-noise problem.

is

$$
\begin{aligned}
x_1 &= \frac{x_1 - 2886.38}{3152.57}, \\
x_2 &= \frac{x_2 - 6.7823}{5.91813}, \\
x_3 &= \frac{x_3 - 0.136548}{0.0935407}, \\
x_4 &= \frac{x_4 - 50.8607}{15.5728}, \\
x_5 &= \frac{x_5 - 0.0111399}{0.0131502}, \\
y^*(\mathbf{x}, \underline{\alpha}^*) &= -3.26798 \\
&- 1.97066 \tanh(2.28999 + 0.0677432x_1 + 0.198962x_2 - 0.224465x_3 + 0.0961816x_4 + 2.7212x_5) \\
&+ 4.69623 \tanh(3.42765 + 2.34839x_1 + 0.635714x_2 + 0.0415776x_3 - 0.0832574x_4 + 2.30865x_5) \\
&+ 0.199113 \tanh(-4010.32 - 3442.27x_1 - 2549.77x_2 + 252.083x_3 - 281.831x_4 + 2783.93x_5) \\
&+ 1921.5 \tanh(0.444241 + 0.148645x_1 - 0.0397378x_2 - 0.371608x_3 + 0.102558x_4 + 0.0102109x_5) \\
&- 0.433932 \tanh(0.618045 + 1.44893x_1 + 3.03601x_2 + 2.75553x_3 - 0.669881x_4 - 1.16721x_5) \\
&- 0.313583 \tanh(3824 + 3180.03x_1 + 2505.72x_2 + 6753.55x_3 - 303.828x_4 - 561.2x_5) \\
&+ 1.71381 \tanh(6.81775 + 8.18818x_1 - 0.222292x_2 + 0.430508x_3 - 0.152801x_4 + 0.801288x_5) \\
&- 3.91 \tanh(2.20453 + 2.68659x_1 + 0.96715x_2 + 0.0871504x_3 - 0.102282x_4 + 0.00203128x_5) \\
&- 1917.76 \tanh(0.443727 + 0.149241x_1 - 0.0404185x_2 - 0.372191x_3 + 0.102622x_4 + 0.0107115x_5 \\
y^*(\mathbf{x}, \underline{\alpha}^*) &= 124.836 + 6.89866y^*(\mathbf{x}, \underline{\alpha}^*).
\end{aligned}
$$

Validation of results

The accuracy of the neural network model over the experimental configurations in the testing data set is measured by performing a linear regression analysis between the outputs from the neural network and the corresponding targets.

The values of the linear regression parameters here are listed in Table 7.17. Here a is the x-intercept, b is the slope and R^2 is the correlation coefficient.

a	=	12.5411
b	=	0.900159
R^2	=	0.894

Table 7.17: Linear regression analysis parameters for the airfoil self noise problem.

Figure 7.19 is a plot of this validation analysis. The broken line represents the optimal model, in which the prediction would perfectly match the experimental data. From this figure we can see that the neural network model provides a good fit across all of the equipment configurations.

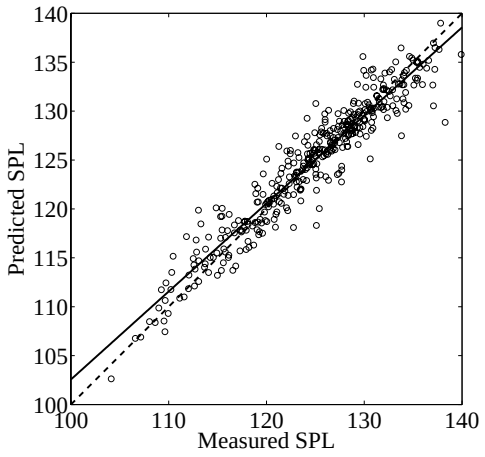

Figure 7.19: Linear regression analysis in the airfoil noise problem.

Conclusions

An overall prediction method based on neural networks has been developed. The approach is based on data from a series of aerodynamic and acoustic tests of isolated airfoil sections.

The results of the neural network model were checked against an independent testing data set, and it was found that the quality of the prediction was satisfactory over the entire range of sound pressure level data.

7.7 Related classes in Flood

The InputTargetDataSet class in Flood

The InputTargetDataSet class represent the concept of input-target data set. It contains:

- The number of samples in the data set.
- The number of input variables.
- The number of target variables.
- The name of the input variables.
- The name of the target variables.
- The units of the input variables.
- The units of the target variables.

- The description of the input variables.

- The description of the target variables.

- The input data.

- The target data.

In order to set up an input-target data set in `Flood`, we first construct an empty input-target data set object. This object will have 0 samples and 0 input and target variables.

```
InputTargetDataSet inputTargetDataSet;
```

Then we load from a file the input-target data set object members, i.e., the number of samples, the number of input and target variables, the names, units and description of input and target variables, and the input and target data.

```
inputTargetDataSet.load("InputTargetDataSet.dat");
```

The InputTargetDataSetTemplate.dat file shows the format of an input-target data set file in `Flood`.

The InputTargetDataSet class also contains methods to perform simple statistics on the input and target data, which will be useful when solving data modeling problems.

The method calculateMeanAndStandardDeviationOfInputData(**void**) returns a matrix where the first row contains the mean of the input variables and the second row contains the standard deviation of the input variables. Similarly, the method calculateMeanAndStandardDeviationOfTargetData(**void**) returns a matrix containing the mean and the standard deviation of the target variables.

```
Matrix<double> meanAndStandardDeviationOfInputData =
inputTargetDataSet.calculateMeanAndStandardDeviationOfInputData();

Matrix<double> meanAndStandardDeviationOfTargetData =
inputTargetDataSet.calculateMeanAndStandardDeviationOfTargetData();
```

The method calculateMinimumAndMaximumOfInputData(**void**) returns a matrix where the first row contains the minimum value of the input variables and the second row contains the maximum value of the input variables. Similarly, the method calculateMinimumAndMaximumOfTargetData(**void**) returns a matrix containing the minimum and the maximum of the target variables.

```
Matrix<double> minimumAndMaximumOfInputData =
inputTargetDataSet.calculateMinimumAndMaximumOfInputData();

Matrix<double> minimumAndMaximumOfTargetData =
inputTargetDataSet.calculateMinimumAndMaximumOfTargetData();
```

All mean, standard deviation, minimum and maximum of input and target data can be calculated by just calling the calculateAllStatistics (**void**) method.

The method preprocessMeanAndStandardDeviation(**void**) scales all input and target data so that the mean of all input and target variables is 0 and their standard deviation is 1.

```
inputTargetDataSet.preprocessMeanAndStandardDeviation();
```

An alternative preprocessing method for an input-target data set is the preprocessMinimumAndMaximum(**void**) 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.

```
inputTargetDataSet.preprocessMinimumAndMaximum();
```

Please note that preprocessing modifies the input and the target data. This needs to be taken into account when subsequent operations are going to be performed with that data.

When solving data modeling applications it is always convenient to split the input-target data set into a training, a validation and a testing subsets. The method split(**double**, **double**) creates two input-target data set objects with the same input and target variables but split input and target data. The data is separated into two ratios specified by the user.

```
Vector<InputTargetDataSet> trainingAndValidation =
inputTargetDataSet.split(50.0,50.0);
```

The source code of a sample application which makes use of the InputTargetDataSet class is listed in the InputTargetDataSetApplication.cpp file.

The error functional classes in Flood

Regarding objective functionals for modeling of data, `Flood` includes the classes SumSquaredError, MeanSquaredError, RootMeanSquaredError, NormalizedSquaredError, MinkowskiError and RegularizedMinkowskiError 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.

To construct a sum squared error object, for instance, we can use the following sentence:

```
SumSquaredError sumSquaredError(&multilayerPerceptron,
&inputTargetDataSet);
```

where &multilayerPerceptron is a reference to a MultilayerPerceptron object and &inputTargetDataSet is a reference to an InputTargetDataSet object.

The calculateEvaluation(**void**) method here returns the sum squared error between the outputs in the multilayer perceptron and the targets in the input-target data set,

```
double evaluation = sumSquaredError.calculateEvaluation();
```

The method calculateGradient(**void**) returns the gradient of the objective function. In all the classes mentioned above it uses the error back-propagation method.

```
Vector<double> gradient = sumSquaredError.calculateGradient();
```

The source code of a sample application (main function) with the MeanSquaredError class is listed in the file MeanSquaredErrorApplication.cpp. Using the other error functional classes mentioned in this chapter should not involve any difficulty from this application.

The LinearRegressionAnalysis class in Flood

Linear regression analysis is a validation technique for data modeling problems which consists on performing a linear regression between the outputs from a multilayer perceptron and the corresponding targets from a testing data set.

In order to construct a LinearRegressionAnalysis object

to compare the outputs from a multilayer perceptron and the targets in a testing data set, the following sentence can be used

```
LinearRegressionAnalysis
linearRegressionAnalysis(&multilayerPerceptron,
&inputTargetDataSet);
```

where &multilayerPerceptron is a reference to a MultilayerPerceptron object and &inputTargetDataSet is a reference to an InputTargetDataSet object.

The calculateRegressionParameters(**void**) performs a linear regression analysis between the neural network outputs and the corresponding data set targets and returns all the provided parameters in a single matrix.

```
Matrix<double> regressionParameters =
linearRegressionAnalysis.calculateRegressionParameters();
```

It is possible to save to a data file all the results provided by this validation technique. This includes the regression parameters but also the target values for the testing data and their corresponding output values from the multilayer perceptron,

```
linearRegressionAnalysis.save("LinearRegressionAnalysis.dat");
```

Execute the LinearRegressionAnalysisApplication.cpp main function to see a simple example of the use of linear regression analysis.

The CorrectPredictionsAnalysis class in Flood

Correct prediction analysis is a validation technique for pattern recognition problems. It simply provides the ratio of correct predictions made by a multilayer perceptron on an independent input target data set.

To construct a CorrectPredictionsAnalysis object to compare the patterns recognized by a multilayer perceptron to the actual ones in a testing data set, the following sentence can be used

```
CorrectPredicitonsAnalysis
correctPredictionsAnalysis(&multilayerPerceptron,
&inputTargetDataSet);
```

where &multilayerPerceptron is a reference to a MultilayerPerceptron object and &inputTargetDataSet is a reference to an InputTargetDataSet object.

The method saveResults(**char***) saves the results of this validation technique for pattern recognition problems to a data file,

```
correctPredictionsAnalysis.saveResults("CorrectPredictionsAnalaysis.dat");
```

The CorrectPredictionsAnalysisApplication.cpp file contains an example of how to use correct predictions analysis.

Chapter 8

Classical problems in the calculus of variations

A variational formulation for the multilayer perceptron provides a direct method for variational problems. Within this formulation, the solution approach consists of three steps, selection of function space, formulation of variational problem and solution of reduced function optimization problem.

For the purpose of validating this numerical method, neural computing is used in this Chapter to solve some classical problems in the calculus of variations. The neural network answers are compared to the analytical ones, with encouraging results. It is very important to understand that no input-target data is used in any of these examples. Instead, the neural networks learn here from mathematical models.

8.1 The geodesic problem

The geodesic problem for the multilayer perceptron is a variational problem with one input and one output variables, two boundary conditions and where evaluation of the objective functional is obtained by integrating a function. All the source code used to solve this problem is included in the C++ library Flood [37].

Problem statement

The geodesic problem on the plane can be stated as:

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 8.1 depicts graphically the formulation for this case study.

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

$$\begin{aligned} L[y(x)] &= \int_A^B ds \\ &= \int_{x_a}^{x_b} \sqrt{dx^2 + dy^2} \end{aligned}$$

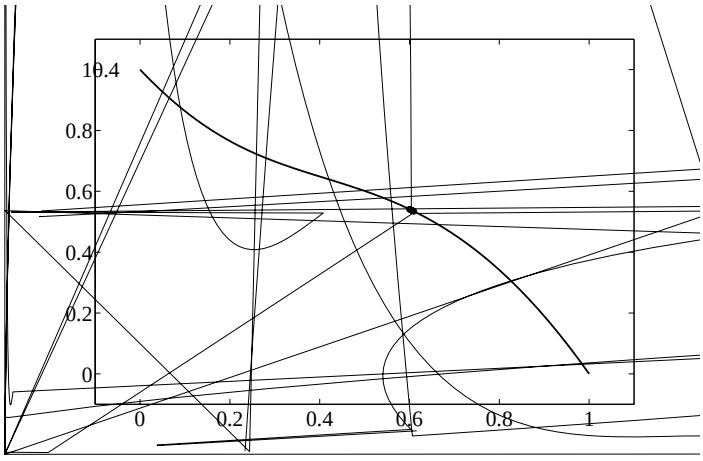

Figure 8.1: The geodesic problem statement.

$$= \int_{x_a}^{x_b} \sqrt{1 + [y'(x)]^2} dx. \tag{8.1}$$

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 functional in Equation (8.1)

$$y^*(x) = 1 - x, \tag{8.2}$$

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

Selection of function space

The neural network chosen to span a function space for the geodesic problem is a multilayer perceptron with one input, x, a sigmoid hidden layer of size three and one linear output neuron, y. Such a computational device is drawn in Figure 8.2.

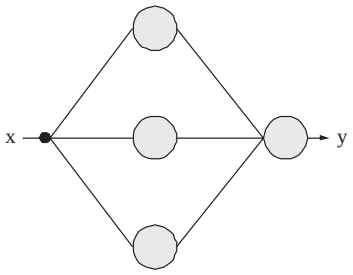

Figure 8.2: Network architecture for the geodesic problem.

The function space spanned by this neural network is denoted V and it is of dimension $s = 10$, the number of parameters. Elements of V are written

$$\begin{array}{rccc} y: & \mathbb{R} & \to & \mathbb{R} \\ & x & \mapsto & y(x;\underline{\alpha}), \end{array}$$

with

$$y(x;\underline{\alpha}) = b_1^{(2)} + \sum_{j=1}^{3} w_{1j}^{(2)} \tanh\left(b_j^{(1)} + w_{j1}^{(1)} x\right). \tag{8.3}$$

The geodesic problem stated in this Section is required to hold $y(0) = 1$ and $y(1) = 0$, the boundary conditions. Therefore this multilayer perceptron must be extended with a particular and an independent solution terms. A suitable set here is

$$\varphi_0(x) = (1 - x), \tag{8.4}$$
$$\varphi_1(x) = x(x - 1), \tag{8.5}$$

respectively. They give

$$y(x;\underline{\alpha}) = (1 - x) + x(x - 1)y(x;\underline{\alpha}). \tag{8.6}$$

Note that all the functions in Equation (8.6) indeed satisfy $y(0) = 1$ and $y(1) = 0$.

On the other hand, all the biases and synaptic weights in the neural network are initialized here at random. The resulting initial guess is depicted in Figure 8.3.

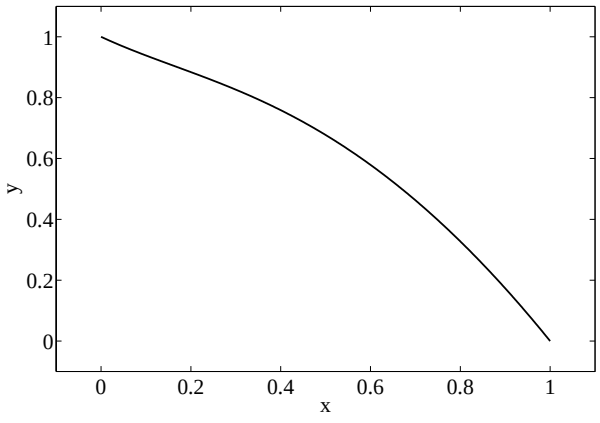

Figure 8.3: Initial guess for the geodesic problem.

Formulation of variational problem

From Equation (8.1), the variational statement of the geodesic problem for the multilayer perceptron is to find a function $y^*(x;\underline{\alpha}^*) \in V$ for which the functional

$$L[y(x;\underline{\alpha})] = \int_{x_a}^{x_b} \sqrt{1 + [y'(x;\underline{\alpha})]^2} dx, \tag{8.7}$$

defined on V, takes on a minimum value.

The derivative of the output with respect to the input, $y'(x;\underline{\alpha})$, is computed with the back-propagation algorithm for the Jacobian matrix.

On the other hand, calculating the arc-length of some function $y(x;\underline{\alpha}) \in V$ requires to integrate a function. Here we use an ordinary differential equation approach for that purpose. In this way, the value of the integral in Equation (8.7) is equivalent to obtain the value $y(b)$ which is the solution to

$$\frac{dy}{dx} = \sqrt{1 + [y'(x)]^2}, \tag{8.8}$$

$$y(a) = y_a. \tag{8.9}$$

This Cauchy problem can be solved with an adaptive stepsize algorithm. Here we use the Runge-Kutta-Fehlberg method with a tolerance of 10^{-15}.

Finally, a back-propagation algorithm for the objective function gradient, $\nabla f(\underline{\alpha})$, is not possible to be derived here, since the target outputs from the neural network are not known. Instead, the central differences method for numerical differentiation is to be used, with $\epsilon = 10^{-6}$.

Solution of reduced function optimization problem

A quasi-Newton method is the training algorithm used to solve the reduced function optimization problem in this application. The particular train direction chosen is that by the BFGS method and the train rate is computed with the Brent's method.

The training algorithm is set to stop when it is not able to reduce the evaluation value of the objective function anymore. The arc-length of the randomly initialized neural network is 1.44235. The quasi-Newton method requires 73 epochs to converge. The final value of the arc-length is 1.41421. On the other side the gradient norm decreases from 0.0588223 to $7.61254 \cdot 10^{-7}$. Figure 8.4 shows the evaluation value of the neural network as a function of the training epoch, while Figure 8.5 depicts the gradient norm training history with a logarithmic scale for the y-axis. There is a significant decrease in both the evaluation value and the gradient norm.

Some numbers resulted from this training process are listed in Table 8.1. These include the number of epochs

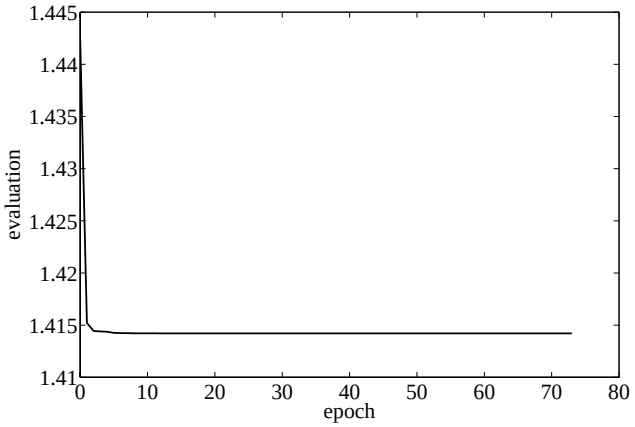

Figure 8.4: Evaluation history for the geodesic problem.

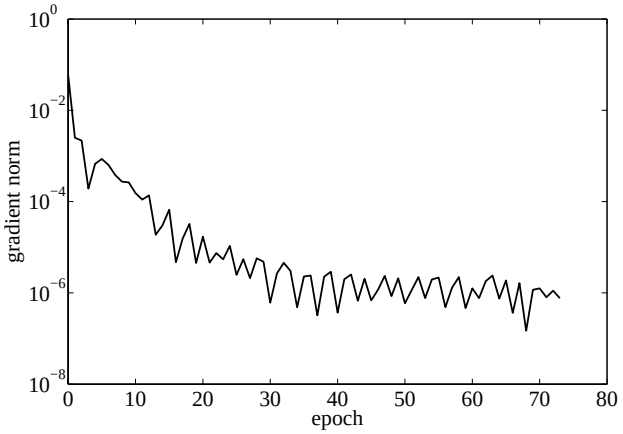

Figure 8.5: Gradient norm history for the geodesic problem.

N	=	73
M	=	4260
CPU	=	$62s$
$\|\underline{\alpha}^*\|$	=	1.46693
$l(\underline{\alpha}^*)$	=	1.41421
$\|\nabla l(\underline{\alpha}^*)\|$	=	$7.61254 \cdot 10^{-7}$

Table 8.1: Training results for the geodesic problem.

for $x \in [0, 1]$.

To conclude, the final shape of the neural network is shown in Figure 8.6. As we can see, this is very much a straight line joining the two points A and B, which is the

analytical solution for the geodesic problem.

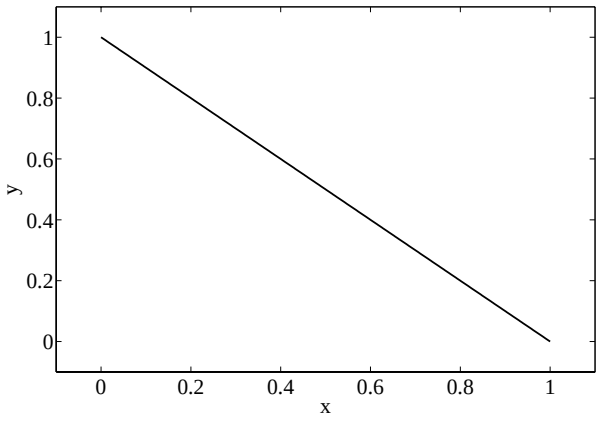

Figure 8.6: Neural network results for the geodesic problem.

8.2 The brachistochrone problem

The brachistochrone problem for the multilayer perceptron can be stated as a variational problem with one input and one output variables, two boundary conditions and an objective functional defined by an improper integral. A C++ software implementation of this problem can also be found within Flood [37]. Similar results to those included here are published in [42].

Problem statement

The original statement of the brachistochrone problem is:

Given two points $A = (x_a, y_a)$ and $B = (x_b, y_b)$ in a vertical plane, what is the curve traced out by a particle acted on only by gravity, which starts at A and reaches B in the shortest time?

In this example we employ a cartesian representation, i.e., the brachistochrone is chosen to be of the form $y = y(x)$. Nevertheless, other representations would be possible, such as a parametric representation, in which the brachistochrone would take the form $(x, y) = (x(t), y(t))$, see Figure 8.2.

The time for a particle to travel from point A to point B within a curve $y(x)$ is given by the functional

$$T[y(x)] = \int_A^B \frac{ds}{v}, \tag{8.11}$$

where s is the arc length and v is the speed. The arc length element here can be written

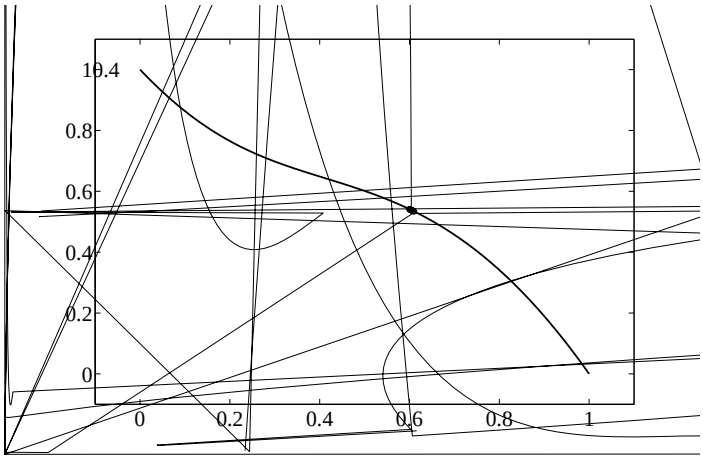

Figure 8.7: The brachistochrone problem statement.

$$\begin{aligned} ds &= \sqrt{dx^2 + dy^2} \\ &= \sqrt{1 + [y'(x)]^2}dx, \end{aligned} \tag{8.12}$$

On the other hand, and in order to obtain an expression for the speed, consider the law of conservation of energy,

$$mgy_{\mathsf{a}} = mgy(x) + \frac{1}{2}mv^2, \tag{8.13}$$

where $g = 9.81$ is the gravitational acceleration. This gives

$$v = \sqrt{2g(y_{\mathsf{a}} - y(x))}. \tag{8.14}$$

Plugging (8.12) and (8.14) into (8.11) produces the final expression for the descent time

$$T[y(x)] = \frac{1}{\sqrt{2g}} \int_{x_a}^{x_b} \sqrt{\frac{1 + [y'(x)]^2}{y_{\mathsf{a}} - y(x)}}dx. \tag{8.15}$$

The analytical solution to the brachistochrone problem is a segment of cycloid [65]. Taking $A = (0, 1)$ and $B = (1, 0)$, the optimal function is given in parametric equations by

$$x^*(\theta) = 0.583(\theta - \sin\theta), \tag{8.16}$$

$$y^*(\theta) = 1 - 0.583(1 - \cos\theta), \tag{8.17}$$

for $\theta \in [0, 2.412]$. Equations (8.16) and (8.17) yield an optimum descent time $T[(x, y)^*(\theta)] = 0.577$.

Selection of function space

Here a multilayer perceptron with a sigmoid hidden layer and a linear output layer is used to represent the descent curve $y(x)$. As it is to be described in cartesian coordinates, the neural network must have one input, x, and one output neuron, y. The size of the hidden layer is a design variable. However, this is not a very critical issue here, since this problem is well-posed, and we use three neurons in the hidden layer. Figure 8.8 is a graphical representation of this network architecture.

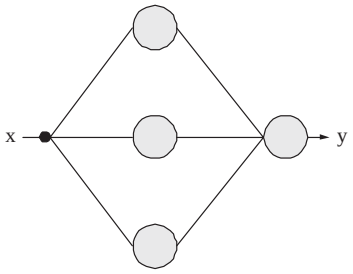

Figure 8.8: Network architecture for the brachistochrone problem.

Such a multilayer perceptron spans a family V of parameterized functions $y(x;\underline{\alpha})$ of dimension $s = 10$, which is the number of parameters in the neural network. Elements of V are of the form

$$\begin{array}{rccc} y: & \mathbb{R} & \rightarrow & \mathbb{R} \\ & x & \mapsto & y(x;\underline{\alpha}), \end{array}$$

where

$$y(x;\underline{\alpha}) = b_1^{(2)} + \sum_{j=1}^{3} w_{1j}^{(2)} \tanh\left(b_j^{(1)} + w_{j1}^{(1)} x\right). \tag{8.18}$$

This multilayer perceptron must be extended so as to satisfy the boundary conditions $y(0) = 1$ and $y(1) = 0$. In this regard, a set of possible particular and homogeneous solution terms is

$$\varphi_0(x) = (1 - x), \tag{8.19}$$
$$\varphi_1(x) = x(x - 1). \tag{8.20}$$

Equations (8.19) and (8.20) give

$$y(x;\underline{\alpha}) = (1 - x) + x(x - 1)y(x;\underline{\alpha}), \tag{8.21}$$

Also, the descent curve $y(x)$ must lie in the interval $(-\infty, 1)$, so the network outputs are bounded in the form

$$y(x;\underline{\alpha}) = \begin{cases} y(x;\underline{\alpha}), & y(x;\underline{\alpha}) < 1, \\ 1-\epsilon, & y(x;\underline{\alpha}) \geq 1, \end{cases} \tag{8.22}$$

being ϵ a small number. As we will see, this is a very delicate choice in this problem. Here we have use $\epsilon = 10^{-5}$.

On the other hand, the neural network biases and synaptic weights are initialized at random. Experience has shown that this direct method does not require a good initial guess for the solution in this problem, which is a positive result. Figure 8.9 shows the initial neural network response for this set of parameters chosen at random.

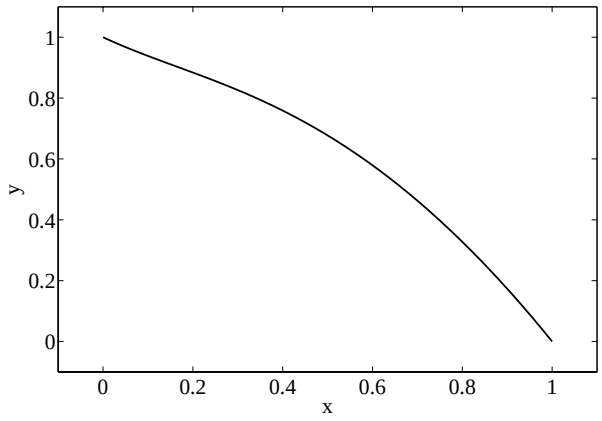

Figure 8.9: Initial guess for the brachistochrone problem.

Formulation of variational problem

Making use of Equation (8.15), the variational statement of the brachistochrone problem for the multilayer perceptron is to find a function $y^*(x;\underline{\alpha}^*) \in V$ for which the functional

$$F[y(x;\underline{\alpha})] = \frac{1}{\sqrt{2g}} \int_0^1 \sqrt{\frac{1+[y'(x;\underline{\alpha})]^2}{1-y(x;\underline{\alpha})}}\, dx, \tag{8.23}$$

defined on V, takes on a minimum value.

The derivative $y'(x;\underline{\alpha})$ can be evaluated analytically with the back-propagation algorithm for the Jacobian matrix, see Section 4.

On the other hand, and to evaluate the objective functional in Equation (8.23), the integration of a function is needed. Moreover, this objective functional has an integrable singularity at the lower limit. This singularity makes this problem very difficult to be solved numerically. One possibility here is to make

$$\int_{x_a}^{x_b} f(x)dx \approx \int_{x_a+}^{x_b} f(x)dx, \tag{8.24}$$

being ϵ a small number. Here we use $\epsilon = 10^{-5}$. The integration is done with an ordinary differential equation approach and the Runge-Kutta method with 1001 integration points, see Sections B.1 and B.2.

Last, evaluation of the objective function gradient, $\nabla f(\underline{\alpha})$, 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 quasi-Newton method with BFGS train direction and Brent optimal train rate methods is the training algorithm chosen to solve the reduced function optimization problem. The tolerance in the Brent's method is set to 10^{-6}.

The objective function in this problem has shown to contain a single, the unique, global minimum. In contrast, other representations for the descent curve, such as piece-wise linear or polynomial, could lead to objective functions with many local minima. Indeed, convergence properties are related to the function space used.

The evaluation of the initial guess is 0.734351. The training algorithm is set to stop when it can not continue with the training process. This occurs after 237 epochs, when the Brent's method gives zero optimal train rate. Of course, a good initial guess would decrease the number of epochs. After training the evaluation falls to 0.595314. The gradient norm decreases from 0.197046 to $2.61258 \cdot 10^{-8}$. Figures 8.10 and 8.11 depict the evaluation value of the objective function and its gradient norm, versus the training epoch, respectively. Note that a logarithmic scale is used for the y-axis in the second plot. The evaluation history shows a strong reduction of the objective function during the first few epochs to become nearly constant until the stopping criterium is satisfied. On the other hand, the gradient norm decreases in several orders of magnitude during the training process to reach a very small value. These two situations show that the quasi-Newton method has certainly converged.

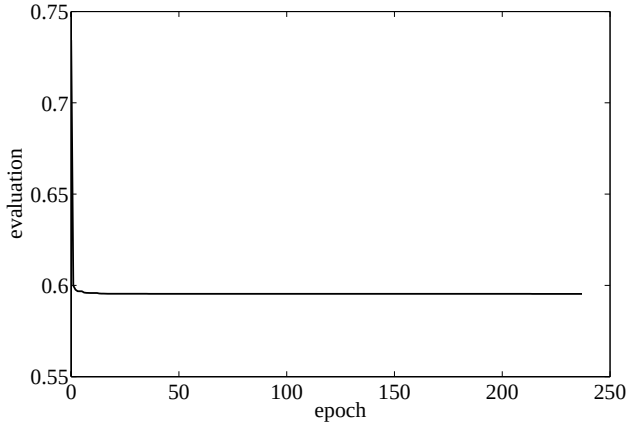

Figure 8.10: Evaluation history for the brachistochrone problem.

Table 8.2 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 in seconds in a laptop AMD 3000, $\|\underline{\alpha}^*\|$ the final parameter vector norm, $t(\underline{\alpha}^*)$ the

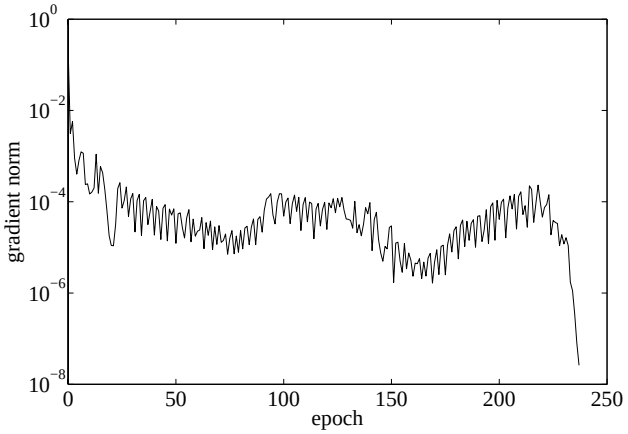

Figure 8.11: Gradient norm history for the brachistochrone problem.

final evaluation value and $\nabla f(\underline{\alpha}^*)$ the final gradient norm. The descent time provided by the neural network yields a percentage error of around -3.0764% when compared to the analytical solution, being the singularity in the integral (8.23) the main cause of error. Anyway, results here are good, since the descent time provided by the network is quite similar to the descent time for the brachistochrone.

N	$=$	237
M	$=$	13526
CPU	$=$	$1166s$
$\|\underline{\alpha}^*\|$	$=$	47.6955
$t(\underline{\alpha}^*)$	$=$	0.595314
$\|\nabla t(\underline{\alpha}^*)\|$	$=$	$2.61258 \cdot 10^{-8}$

Table 8.2: Training results for the brachistochrone problem.

The explicit form of the function represented by the trained neural network is

$$\begin{aligned} y^*(x;\underline{\alpha}^*) &= (1-x) + x(x-1)[38.7381 \\ &- 59.6973 \tanh(2.19639 + 1.57101x) \\ &+ 21.9216 \tanh(48.4807 + 120.955x) \\ &+ 0.392256 \tanh(0.990452 - 14.656x)]. \end{aligned} \tag{8.25}$$

for $x \in [0,1]$. Figure 8.12 illustrates the neural network solution. This shape also agrees to that of the cycloid, which is the analytical solution to the brachistochrone problem.

8.3 The catenary problem

The catenary problem for the multilayer perceptron is a variational problem with one input and one output variables, two boundary conditions, and an objective functional

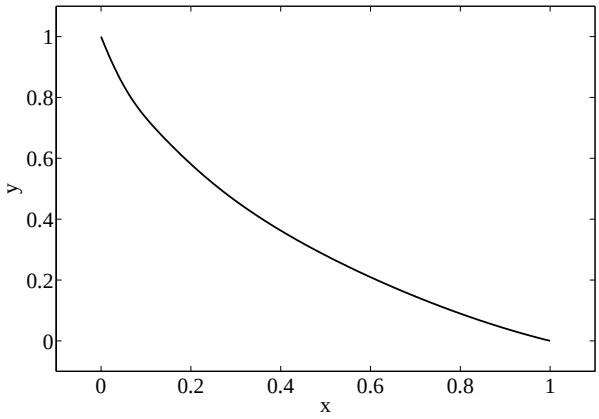

Figure 8.12: Neural network results for the brachistochrone problem.

with one constraint and requiring the integration of two functions. This problem is implemented in the C++ class library Flood [37].

Problem statement

The catenary problem is posed as follows:

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 8.13 graphically declares the catenary problem.

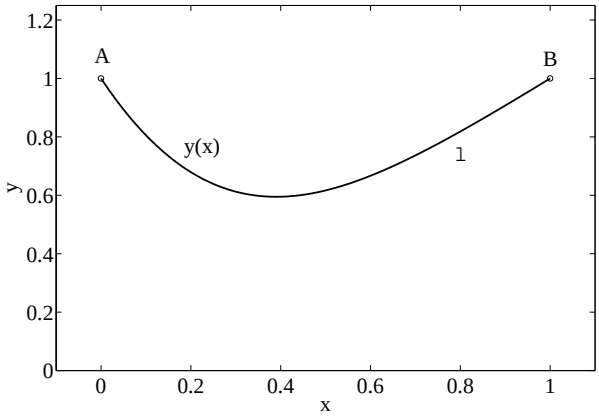

Figure 8.13: The catenary problem statement.

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

$$L[y(x)] = \int_{x_a}^{x_b} \sqrt{1 + [y'(x)]^2} dx. \tag{8.26}$$

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

$$\begin{aligned} E_L[y(x)] &= \int_{x_a}^{x_b} \sqrt{1 + [y'(x)]^2} dx - l \\ &= 0. \end{aligned} \tag{8.27}$$

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

$$V[y(x)] = \int_{x_a}^{x_b} y(x) \sqrt{1 + [y'(x)]^2} dx. \tag{8.28}$$

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

$$y^*(x) = 0.1891 + 0.3082 \cosh\left(\frac{x - 0.5}{0.3082}\right). \tag{8.29}$$

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

Selection of function space

A multilayer perceptron with one input, three sigmoid neurons in the hidden layer and a linear output neuron is used to span a function space for the catenary problem. Here the input represents the independent variable x, while the output represents the dependent variable y, see Figure 8.14.

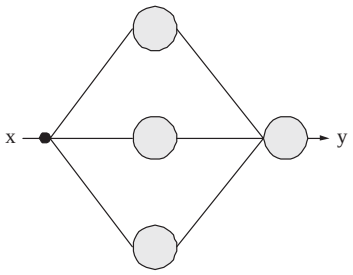

Figure 8.14: Network architecture for the catenary problem.

The family of functions created by this neural network is denoted V and it is of dimension $s = 10$. The functions belonging to V are of the type

$$\begin{aligned} y: \quad \mathbb{R} &\to \mathbb{R} \\ x &\mapsto y(x; \underline{\alpha}), \end{aligned}$$

where

$$y(x;\underline{\alpha}) = b_1^{(2)} + \sum_{j=1}^{3} w_{1j}^{(2)} \tanh\left(b_j^{(1)} + w_{j1}^{(1)} x\right). \tag{8.30}$$

However, this is not a suitable starting point for the catenary problem, since the elements of V do not hold the boundary conditions $y(0;\underline{\alpha}) = 1$ and $y(1;\underline{\alpha}) = 1$. For that reason, the multilayer perceptron of Equation (8.30) must be extended with a particular and an homogeneous solution terms such as

$$\begin{aligned} \varphi_0(x) &= 1, & (8.31)\\ \varphi_1(x) &= x(x-1), & (8.32) \end{aligned}$$

to give

$$y(x;\underline{\alpha}) = 1 + x(x-1)y(x;\underline{\alpha}). \tag{8.33}$$

Now all the functions of the family defined by Equation (8.33) satisfy $y(0;\underline{\alpha}) = 1$ and $y(1;\underline{\alpha}) = 1$.

The neural network must be initialized by assigning some values to the parameters. Here we adopt a random initial guess, see Figure 8.15.

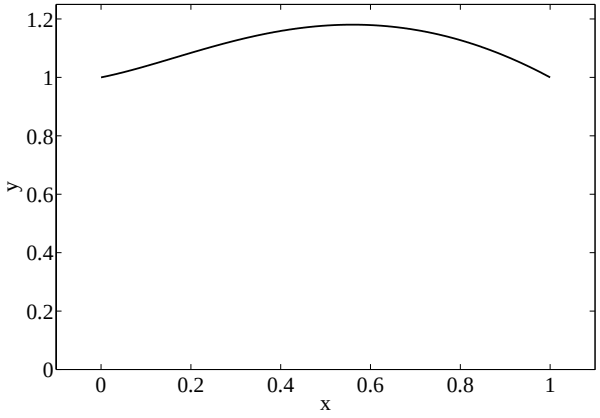

Figure 8.15: Initial guess for the catenary problem.

Formulation of variational problem

From Equations (8.27) and (8.28), the constrained problem can be stated as to find a function $y^*(x;\underline{\alpha}^*) \in V$ such that

$$\begin{aligned} E_L[y^*(x;\underline{\alpha}^*)] &= \int_{x_a}^{x_b} \sqrt{1 + [y'(x;\underline{\alpha}^*)]^2}\, dx - l \\ &= 0 \end{aligned} \tag{8.34}$$

and for which the functional

$$V[y(x;\underline{\alpha})] = \int_{x_a}^{x_b} y(x;\underline{\alpha})\sqrt{1+[y'(x;\underline{\alpha})]^2}dx, \tag{8.35}$$

defined on V, takes on a minimum value.

The use of a penalty term for the constraint in the length can approximate this problem to an unconstrained one, which is now formulated as to find a function $y^*(x;\underline{\alpha}^*) \in V$ for which the functional

$$F[y(x;\underline{\alpha})] = \rho_L \left(E_L\left[y(x;\underline{\alpha}^*)\right]\right)^2 + \rho_V V[y(x;\underline{\alpha}^*)], \tag{8.36}$$

defined on V, takes on a minimum value.

The numbers ρ_L and ρ_V can be named length error term weight and potential energy term weight, respectively. These are design variables, and their values should be chosen so that the constraint term dominates over the objective term. Here we have set $\rho_L = 100$ and $\rho_V = 1$.

The value of $y'(x;\underline{\alpha})$ in Equations (8.34) and (8.35) is obtained by means of the back-propagation algorithm for the Jacobian matrix. This is derived in Section 4.

On the other hand, evaluation of (8.36) requires the integration of two functions. Here we use an ordinary differential equation approach to approximate the integrals with the Runge-Kutta-Fehlberg method, see Sections B.1 and B.2. The tolerance is set to 10^{-15}.

As it is not possible to derive a back-propagation algorithm for the gradient $\nabla f(\underline{\alpha})$ in this problem, that item is obtained numerically with central differences and using $\epsilon = 10^{-6}$

Solution of reduced function optimization problem

A quasi-Newton method with BFGS inverse Hessian approximation and Brent's line minimization methods is used here for training. No stopping criterion is chosen. Instead training will finish when the algorithm can't keep on doing. At that point the Brent's method gives zero train rate. Figures 8.16 and 8.17 depict the training histories for the evaluation and the gradient norm. They both supply good signs about the this process, since the evaluation and the gradient norm decrease significantly.

Some results by the quasi-Newton method are listed in Table 8.3. This contains number of epochs N, number of evaluations M, CPU time in a laptop AMD 3000 CPU, final parameters norm $\|\underline{\alpha}^*\|$, final evaluation $f(\underline{\alpha}^*)$, final length error $e_l(\underline{\alpha}^*)$, final potential energy $v(\underline{\alpha}^*)$ and final gradient norm $\|\nabla f(\underline{\alpha}^*)\|$.

The multilayer perceptron for the catenary problem can be written explicitly as

$$\begin{aligned} y^*(x;\underline{\alpha}^*) &= 1 + x(x-1)[0.417922 \\ &- 5.13772\tanh\left(0.390226 + 0.507345x\right) \\ &+ 2.5535\tanh\left(-1.24642 + 1.08916x\right) \\ &+ 6.05167\tanh\left(4.20948 + 1.65389x\right)], \end{aligned} \tag{8.37}$$

for $x \in [0,1]$.

To conclude, a plot of Equation (8.37) is shown in Figure 8.18. Here we can see a good match between the neural network results and the analytical catenary.

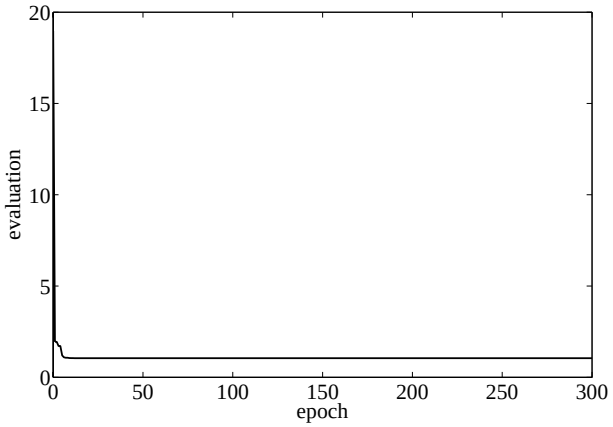

Figure 8.16: Evaluation history for the catenary problem.

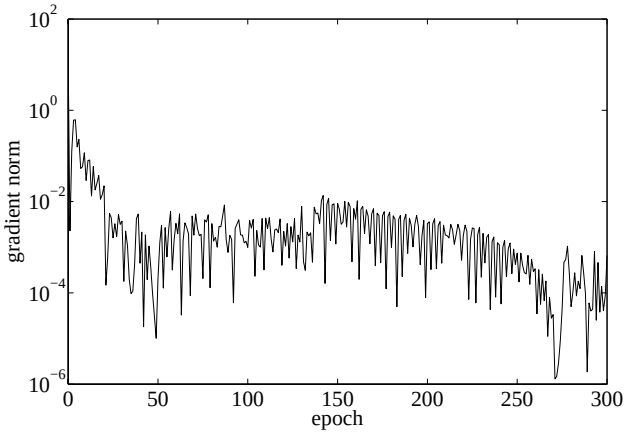

Figure 8.17: Gradient norm history for the catenary problem.

N	=	300
M	=	17328
CPU	=	$871s$
$\|\underline{\alpha}^*\|$	=	3.05481
$f(\underline{\alpha}^*)$	=	1.04588
$e_l(\underline{\alpha}^*)$	=	$9.53529 \cdot 10^{-4}$
$v(\underline{\alpha}^*)$	=	1.04579
$\|\nabla f(\underline{\alpha}^*)\|$	=	$6.60019 \cdot 10^{-4}$

Table 8.3: Training results for the catenary problem.

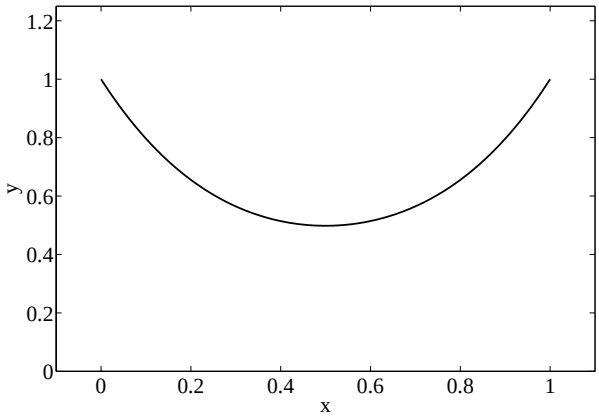

Figure 8.18: Neural network results for the catenary problem.

8.4 The isoperimetric problem

The isoperimetric problem for the multilayer perceptron is a variational problem with one input and two output variables, four boundary conditions and one constraint in the objective functional, which is defined by the integration of two functions. All the code in the C++ programming language needed to solve this problem can be found in Flood [37]. Another variation of the isoperimetric problem for the multilayer perceptron has been published in [42].

Problem statement

The isoperimetric problem is another classical problem in the calculus of variations [65]. The statement of this problem is:

Of all simple closed curves in the plane of a given length l, which encloses the maximum area?.

Figure 8.19 is a graphical statement of the isoperimetric problem. Note that we can not use here a function $y(x)$ to specify the closed curve, since closed curves will necessarily make the function multi-valued. Instead, we use the parametric equations $(x, y)(t)$, for $t \in [0, 1]$, and such that $x(0) = 0$, $x(1) = 0$, $y(0) = 0$ and $y(1) = 0$.

For a plane curve specified in parametric equations as $(x, y)(t)$, the arc length is given by [65]

$$L[(x, y)(t)] = \int_0^1 \sqrt{[x'(t)]^2 + [y'(t)]^2} dt. \tag{8.38}$$

From Equation (8.38), the constraint on the length can be expressed as an error functional,

$$E_L[(x, y)(t)] = \int_0^1 \sqrt{[x'(t)]^2 + [y'(t)]^2} dt - l$$

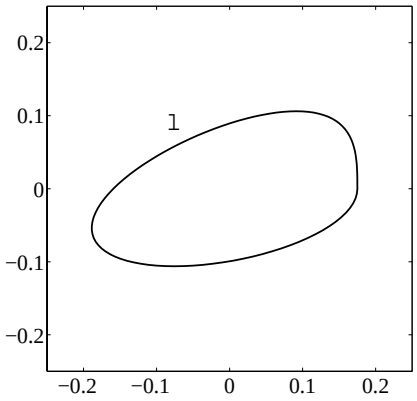

Figure 8.19: The isoperimetric problem statement.

$$= \quad 0. \tag{8.39}$$

On the other hand, Green's theorem gives the signed area as [65]

$$A[(x,y)(t)] = \frac{1}{2}\int_0^1 \left(x(t)y'(t) - x'(t)y(t)\right) dt. \tag{8.40}$$

The analytical solution to the isoperimetric problem is a circle [65]. Taking, for instance, the perimeter goal to be $l = 1$, the optimal function is given by

$$x(t) = -\frac{1}{2\pi} + \frac{1}{2\pi}\cos(2\pi t), \tag{8.41}$$

$$y(t) = \frac{1}{2\pi}\sin(2\pi t), \tag{8.42}$$

for $t \in [0,1]$. That circle yields a maximum area $A[(x,y)^*(t)] = 1/(4\pi)$.

Selection of function space

A neural network consisting of a hidden layer of sigmoid perceptrons and an output layer of linear perceptrons is used here to represent the closed curve. As this shape is written in parametric equations as $(x,y) = (x,y)(t)$, the network must have one input, t, and two output neurons, x and y. The number of neurons in the hidden layer is up to the designer. Anyhow, this is not a very important choice in this problem, since it can be regarded well-conditioned. Therefore, a neural network with sufficient complexity will be enough. Here we use three neurons in the hidden layer. Figure 8.20 depicts this network architecture.

The neural network in Figure 8.20 spans a function space V of dimension $s = 14$. The elements of this family of functions are of the form

$$\begin{aligned} (x,y): \quad \mathbb{R} &\to \mathbb{R}^2 \\ t &\mapsto (x,y)(t;\underline{\alpha}), \end{aligned}$$

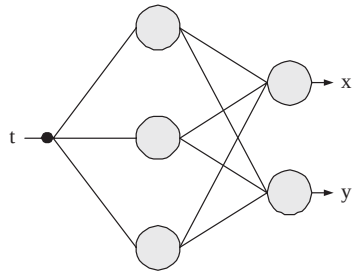

Figure 8.20: Network architecture for the isoperimetric problem.

where

$$x(t;\underline{\alpha}) = b_1^{(2)} + \sum_{j=1}^{3} w_{1j}^{(2)} \tanh\left(b_j^{(1)} + w_{j1}^{(1)} t\right), \tag{8.43}$$

$$y(t;\underline{\alpha}) = b_2^{(2)} + \sum_{j=1}^{3} w_{2j}^{(2)} \tanh\left(b_j^{(1)} + w_{j1}^{(1)} t\right). \tag{8.44}$$

The function space for this problem is required to satisfy the boundary conditions $x(0) = 0$, $x(1) = 0$, $y(0) = 0$ and $y(1) = 0$. A set of homogeneous and particular solution terms here could be

$$\begin{aligned} \varphi_{x0}(t) &= 0, & (8.45)\\ \varphi_{x1}(t) &= t(t-1), & (8.46)\\ \varphi_{y0}(t) &= 0, & (8.47)\\ \varphi_{y1}(t) &= t(t-1). & (8.48) \end{aligned}$$

Elements of V thus become

$$\begin{aligned} x(t;\underline{\alpha}) &= t(t-1)x(t;\underline{\alpha}), & (8.49)\\ y(t;\underline{\alpha}) &= t(t-1)y(t;\underline{\alpha}). & (8.50) \end{aligned}$$

Please note that although (8.49) is written as two different equations they represent just one function, where many of the parameters are shared by both output variables. This is a potential advantage over other direct methods which might use as many different functions as output variables. Indeed, if the outputs are related in some way, shared parameters lead in a reduction in the dimension of the function space.

Finally, the neural network is initialized with parameters chosen at random. Figure (8.21) shows the initial guess for the isoperimetric problem considered here. As the initial solution is a random shape, it does not necessarily satisfy the constraint on the length.

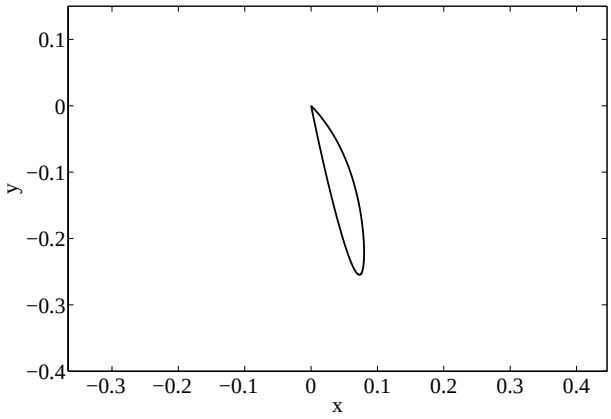

Figure 8.21: Initial guess for the isoperimetric problem.

Formulation of variational problem

Considering Equations (8.39) and (8.40) the variational statement of the isoperimetric problem for the multilayer perceptron is to find a function $(x, y)^*(t; \underline{\alpha}^*) \in V$, for $t \in [0, 1]$, such that

$$\begin{aligned} E_L\left[(x, y)^*(t; \underline{\alpha}^*)\right] &= \int_0^1 \sqrt{[x'(t)]^2 + [y'(t)]^2} dt - l \\ &= 0, \end{aligned} \tag{8.51}$$

and for which the functional

$$A[(x, y)(t; \underline{\alpha})] = \frac{1}{2} \int_0^1 \left(x(t; \underline{\alpha}) y'(t; \underline{\alpha}) - x'(t; \underline{\alpha}) y(t; \underline{\alpha})\right) dt, \tag{8.52}$$

defined on V, takes on a maximum value.

This constrained problem can be reformulated as an unconstrained problem by adding a penalty term for the constraint E_L to the original objective functional A. The reduced problem is now stated as to find a function $(x, y)^*(t; \underline{\alpha}^*) \in V$, for $t \in [0, 1]$, and for which the objective functional

$$F[(x, y)(t; \underline{\alpha})] = \rho_L \left(E_L\left[(x, y)(t; \underline{\alpha})\right]\right)^2 - \rho_A A[(x, y)(t; \underline{\alpha})], \tag{8.53}$$

defined on V, takes on a minimum value.

Here ρ_L is the length error term weight and ρ_A is the area term weight. Both parameters are design variables in the problem, and should be chosen so that the initial value of the objective functional is of order unity and so that the values of the penalty term is much smaller than the value of the objective term. For this problem we have chosen $\rho_L = 100$ and $\rho_A = 1$.

The quantities $x'(t;\underline{\alpha})$ and $y'(t;\underline{\alpha})$ are computed with the back-propagation algorithm for the derivatives of the output variables with respect to the input variables, i.e., the Jacobian matrix.

On the other hand, evaluation of (8.53) requires the integration of two functions, which is performed here numerically. An ordinary differential equation approach using an adaptive stepsize algorithm gives the greatest accuracy. In particular we use the Runge-Kutta-Fehlberg method with a tolerance of 10^{-15}.

Finally, the gradient vector of the objective function, $\nabla f(\underline{\alpha}^*)$, is computed numerically applying central differences with $\epsilon = 10^{-6}$.

Solution of reduced function optimization problem

A quasi-Newton method with BFGS train direction and Brent optimal train rate is adopted here for solving the reduced function optimization problem. This particular objective function has demonstrated not to contain local minima. Certainly, the function space spanned by the neural network has good convergence properties. On the contrary, other representations for the closed curve, such as piece-wise linear or polynomial, could lead to objective functions with many local minima.

In this example, no more reduction in the objective function will cause to finish the training process. It might happen that the Brent's method gives zero train rate for some BFGS train direction. In such occasions we can restart the training process. On the other hand, constrained problems need in general more training effort that unconstrained problems. Also, a good initial guess can decrease very much that computing activity. Figure 8.22 shows a evaluation history which decreases very rapidly to become almost constant, while Figure 8.23 depicts a gradient norm history decreasing in several orders of magnitude and with a very small final value. Therefore, we can say that the quasi-Newton method has found a minimum.

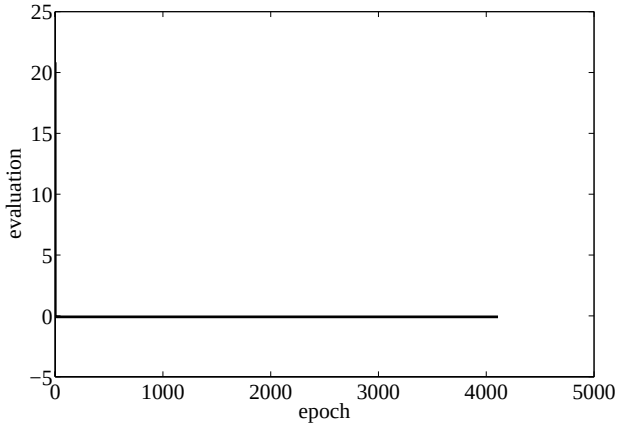

Figure 8.22: Evaluation history for the isoperimetric problem.

Table 8.4 shows the training results for this problem. Here N is the number of training epochs, M the number of evaluations, CPU the training time for a laptop AMD 3000, $\|\underline{\alpha}^*\|$ the final parameters norm $f(\underline{\alpha}^*)$ the final objective function value, $e_l(\underline{\alpha}^*)$ the final length error, $a(\underline{\alpha}^*)$ the final closed curve area and $\|\nabla f(\underline{\alpha}^*)\|$ the final

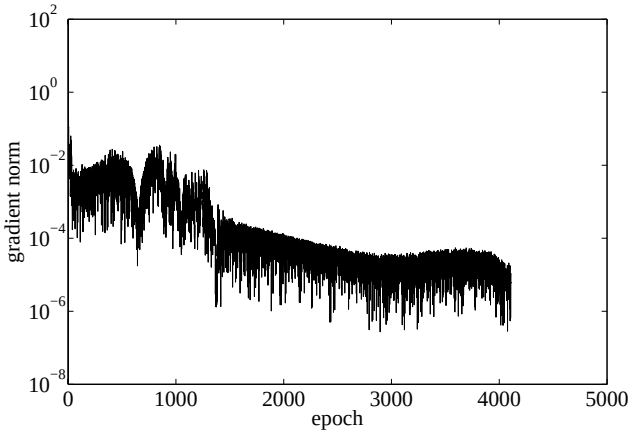

Figure 8.23: Gradient norm history for the isoperimetric problem.

gradient norm. The perimeter error is less than 10^{-3} and the error in the closed curve by the neural network is 0.1256%. Results here are also good, since the perimeter error is very small and the neural network area is very similar to that of the circle.

N	$=$	4109
M	$=$	138507
CPU	$=$	$24044s$
$\|\underline{\alpha}^*\|$	$=$	11.0358
$f(\underline{\alpha}^*)$	$=$	-0.0796397
$e_l(\underline{\alpha}^*)$	$=$	$7.96674 \cdot 10^{-4}$
$a(\underline{\alpha}^*)$	$=$	0.0797032
$\|\nabla f(\underline{\alpha}^*)\|$	$=$	$1.10828 \cdot 10^{-5}$

Table 8.4: Training results for the isoperimetric problem.

The explicit expression of the function addressed by this neural network is written

$$\begin{aligned}
x^*(t;\underline{\alpha}^*) &= t(t-1)[10.7483 \\
&+ 4.68016\tanh(-4.05158 + 2.43348t) \\
&- 12.4708\tanh(-0.196899 + 0.680774t) \\
&+ 14.4052\tanh(-0.789403 + 0.839184t)], \qquad (8.54) \\
y^*(t;\underline{\alpha}^*) &= t(t-1)[-23.7525 \\
&- 24.3861\tanh(-4.05158 + 2.43348t) \\
&+ 4.62781\tanh(-0.196899 + 0.680774t) \\
&+ 0.374342\tanh(-0.789403 + 0.839184t)], \qquad (8.55)
\end{aligned}$$

for $t \in [0,1]$ and where all the parameter values have been rounded to the third decimal point.

To end, Figure 8.24 illustrates the neural network solution to the isoperimetric problem. We can see that the shape provided by the numerical method is that of the circle.

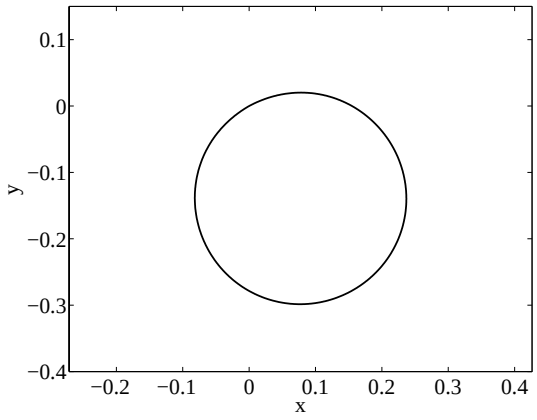

Figure 8.24: Neural network results for the isoperimetric problem.

8.5 Related classes in Flood

The GeodesicProblem class in Flood

The GeodesicProblem class in `Flood` implements the objective functional for the geodesic problem. To construct an object of this class the following sentence is used,

```
GeodesicProblem geodesicProblem(&multilayerPerceptron);
```

where &multilayerPerceptron is a reference to a MultilayerPerceptron object. This neural network must have one input and one output variable, otherwise `Flood` will throw an error message.

On the other hand, the geodesic problem is defined by two boundary conditions, $y(a) = y_a$ and $y(b) = y_b$. In order to make the multilayer perceptron to hold them it must be extended with a particular and independent solution terms. To do that two things must be taken into account:

First, the methods
calculateParticularSolution(Vector<**double**>),
calculateHomogeneousSolution(Vector<**double**>),
calculateParticularSolutionDerivative(Vector<**double**>) and
calculateHomogeneousSolutionDerivative(Vector<**double**>)
need to be implemented within the GeodesicProblem class. Second, any output from the neural network must be calculated using the method
calculateOutput(Vector<**double**>input,BoundaryConditionsType&boundaryConditions),
which is implemented in the MultilayerPerceptron class. Inside the GeodesicProblem class this is done with the following syntax,

```
output = multilayerPerceptron->calculateOutput(input, *this);
```

In order to implement the GeodesicProblem class, the calculateEvaluation(**void**) method must be written. This method returns the arc length of some function represented by the multilayer perceptron as in Equation 8.7.

The calculateArcLengthIntegrand(**double**, **double**) method returns the integrand of that expression.

The calculateArcLength(**void**) method makes use of the OrdinaryDifferentialEquations class to integrate numerically the arc length integrand. To do that, an ordinary differential equations object is included as a member in the GeodesicProblem class. When an equation needs to be integrated the following method is used

```
int calculateRungeKuttaFehlbergIntegral(T, Vector<double>&, Vector<double>&,
double (T::*f)(double, double),double, double, double, double, int).
```

To sum up, the calculateEvaluation(**void**) method returns that arc length.

Finally, an example usage (main function) of the GeodesicProblem class can be found in the GeodesicProblemApplication.cpp file in the Applications folder of `Flood`.

The BrachistochroneProblem class in Flood

The BrachistochroneProblem class in `Flood` implements the objective functional for the brachistochrone problem. The whole class is very similar to the GeodesicProblem class. To construct a brachistochrone problem object the following sentence is used,

```
BrachistochroneProblem
brachistochroneProblem(&multilayerPerceptron);
```

where &multilayerPerceptron is a reference to a MultilayerPerceptron object. This neural network must have one input and one output variable, otherwise `Flood` will throw an error message.

Dealing with boundary conditions within this class is performed in the same manner as for the geodesic problem, that is, the
calculateParticularSolution(Vector<**double**>),
calculateHomogeneousSolution(Vector<**double**>),
calculateParticularSolutionDerivative(Vector<**double**>) and
calculateHomogeneousSolutionDerivative(Vector<**double**>)
methods need to be implemented and any output from the neural network must be calculated using
calculateOutput(Vector<**double**>input,BoundaryConditionsType&boundaryConditions).

Also, the calculateEvaluation(**void**) method is written in a similar way as for the geodesic problem. The descent time for some function represented by the multilayer perceptron is calculated with the calculateDescentTime(**void**) method, which integrates the calculateDescentTimeIntegrand(**double**, **double**) method using the OrdinaryDifferentialEquations class by using the method

```
int calculateRungeKuttaFehlbergIntegral(T, Vector<double>&, Vector<double>&,
double (T::*f)(double, double),double, double, double, double, int).
```

The BrachistochroneProblemApplication.cpp file in the Applications folder of `Flood` contains the main function used to solve the problem stated in this section.

The CatenaryProblem class in Flood

The class representing the objective functional for the catenary problem in `Flood` is called CatenaryProblem. This class is also similar to the two former, but in this it

includes a constraint. An object of the catenary problem class can be constructed as follows,

```
CatenaryProblem
catenaryProblem(&multilayerPerceptron);
```

where &multilayerPerceptron is a reference to a MultilayerPerceptron object. This neural network must have one input and one output neuron.

The homogeneous and particular solution terms are written in the
calculateParticularSolution(Vector<**double**>),
calculateHomogeneousSolution(Vector<**double**>),
calculateParticularSolutionDerivative(Vector<**double**>) and
calculateHomogeneousSolutionDerivative(Vector<**double**>)
methods. The output from the neural network satisfying the boundary conditions is calculated using the
calculateOutput(Vector<**double**>input,BoundaryConditionsType&boundaryConditions) method.

As it happens for all the ObjectiveFunctional classes, the calculateEvaluation(**void**) method needs to be implemented. This is given by Equation (8.36), and consists of two terms, the length error term and the potential energy term.

In order to calculate the error in the constraint made by some function represented by the neural network the following methods are used:
calculateLengthIntegrand(**double**, **double**),
calculateLength(**void**) and
calculateLengthError(**void**).

To calculate the objective functional itself this class uses the
calculatePotentialEnergyIntegrand(**double**, **double**) and
calculatePotentialEnergy(**void**) methods.

The calculateLength(**void**) and calculatePotentialEnergy(**void**) methods use the OrdinaryDifferentialEquations class to integrate the respective integrands. To do that, an ordinary differential equations object is included as a member in the CatenaryProblem class. When that functions need to be integrated the following method is used

```
int calculateRungeKuttaFehlbergIntegral(T, Vector<double>&, Vector<double>&,
double (T::*f)(double, double),double, double, double, double, int).
```

Evaluation of the objective functional in (8.36) is made by a weighted sum of the length error term and the potential energy. The
setLengthErrorWeight(**double**) and
setPotentialEnergyWeight(**double**)
methods set the weights of that two terms, respectively.

To conclude, the CatenaryProblemApplication.cpp file in the Applications folder contains an example of the use of the CatenaryProblem class in `Flood`.

The IsoperimetricProblem class in Flood

The IsoperimetricProblem class in `Flood` represents the concept of objective functional for the isoperimetric problem. This class is very similar to the CatenaryProblem class. An object here is constructed in the following manner,

```
IsoperimetricProblem isoperimetricProblem(&multilayerPerceptron);
```

where &multilayerPerceptron is a reference to a MultilayerPerceptron object. In this case that neural network must have one input and two output neurons.

As it happens always when some boundary conditions must be satisfied, the homogeneous and particular solution terms are written in the
calculateParticularSolution (Vector<**double**>),
calculateHomogeneousSolution(Vector<**double**>),
calculateParticularSolutionDerivative (Vector<**double**>) and
calculateHomogeneousSolutionDerivative(Vector<**double**>) methods.
Also, the output from the neural network satisfying the boundary conditions is calculated using the
calculateOutput(Vector<**double**>input,BoundaryConditionsType&boundaryConditions) method.
Remember that the homogeneous and particular solution terms are implemented in the concrete ObjectiveFunctional class.

On the other hand, in order to implement the getEvaluation(**void**) method in Equation (8.53), two terms need to be evaluated, the error in the constraint, and the objective functional itself.

To calculate the perimeter error the
calculatePerimeterIntegrand(**double**, **double**),
calculatePerimeter(**void**), and
calculatePerimeterError(**void**) methods are implemented.

To calculate the area the
calculateAreaIntegrand(**double**, **double**) and
calculateArea(**void**) methods are used.

In order to evaluate both terms in the objective functional expression integration of functions is needed. This is performed here with an ordinary differential equations approach and making use of the OrdinaryDifferentialEquations class. The following method is used here,

```
int calculateRungeKuttaFehlbergIntegral(T, Vector<double>&, Vector<double>&,
double (T::*f)(double, double),double, double, double, double, int).
```

The weights of the constraint error and the objective itself can be modified with the setPerimeterErrorWeight(**double**) and setAreaWeight(**double**) methods.

An example of the use of the IsoperimentricProblem class can be found in the IsoperimetricProblemApplication.cpp file. This is a main function contained in the Applications folder.

Chapter 9

Optimal control problems

This section gives a brief introduction to optimal control theory, in which the most important aspects of mathematical formulation are reviewed. The problem of optimal control is intimately related to the calculus of variations, and therefore it can be formulated as a novel learning task for the multilayer perceptron.

In order to validate this numerical method, an optimal control problem with analytical solution is solved by means of a multilayer perceptron. Two practical applications of chemical and aeronautical engineering interest are also approached with encouraging results.

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 [32].

The formulation of an optimal control problem requires:

(i) A mathematical model of the system.

(ii) A statement of the physical constraints.

(iii) A specification of the performance criterion.

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 $\mathbf{y}(\mathbf{x})$ the control variables to the system and $\mathbf{u}(\mathbf{x})$ the state variables of the system. Then the mathematical model (or state equation) can be written as

$$\mathcal{L}(\mathbf{y}(\mathbf{x}), \mathbf{u}(\mathbf{x}), \mathbf{x}) = \mathbf{f}, \tag{9.1}$$

where $\mathcal{L}$ is some algebraic or differential operator and $\mathbf{f}$ is some forcing term.

Physical constraints

An optimal control problem might be specified by a set of constraints on the controls and the states of the system.

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.

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 [32].

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.*

Similarly, a state which satisfies the state constraints is called an admissible state [32].

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.*

Performance criterion

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

$$\begin{array}{rccc} F: & Y & \rightarrow & \mathbb{R} \\ & \mathbf{y}(\mathbf{x}) & \mapsto & F[\mathbf{y}(\mathbf{x})]. \end{array}$$

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 [32].

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 5 (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 performance criterion*

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

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 [32].

Solution approach

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. Following the nomenclature introduced in this section, the optimal control problem for the multilayer perceptron translates as follows:

Problem 6 (Optimal control problem for the multilayer perceptron) *Let* Y *and* U *be the function spaces of admissible controls and states, respectively. Let also* V *be the space consisting of all controls* $\mathbf{y}(\mathbf{x}; \underline{\alpha})$ *that a given multilayer perceptron can define, with dimension* s*. Find an admissible control* $\mathbf{y}^*(\mathbf{x}; \underline{\alpha}^*) \in Y$ *which causes the system*

$$\mathcal{L}(\mathbf{y}(\mathbf{x}; \underline{\alpha}), \mathbf{u}(\mathbf{x}), \mathbf{x}) = \mathbf{f}$$

to be in an admissible state $\mathbf{u}^*(\mathbf{x}) \in U$*, and for which the performance criterion*

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

defined on V*, takes on a minimum or a maximum value.*

9.2 Validation examples

In this section a variational formulation for the multilayer perceptron is applied for validation to an optimal control problem. The selected example can be solved analytically, which enables a fair comparison with the neural network results.

9.3 The car problem

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. This problem is included with the Flood library [37], and it has been published in [39]. Another variant of the car problem not considered here can also be found in [44].

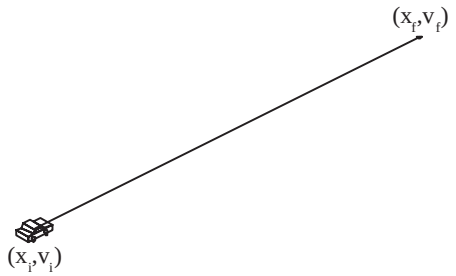

Figure 9.1: The car problem statement.

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,

$$\begin{aligned} \dot{x}(t) &= v(t), && (9.2)\\ \dot{v}(t) &= a(t) + d(t), && (9.3)\\ x(0) &= x_i, && (9.4)\\ v(0) &= v_i, && (9.5) \end{aligned}$$

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

$$\begin{aligned} 0 \leq \quad a(t) \quad &\leq \sup(a), && (9.6)\\ -\sup(d) \leq \quad d(t) \quad &\leq 0. && (9.7) \end{aligned}$$

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

$$F[(a, d)(t)] = t_f. \tag{9.8}$$

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,

$$\begin{aligned} E_x[(a,d)(t)] &= x(t_f) - x_f \\ &= 0, \end{aligned} \tag{9.9}$$

$$\begin{aligned} E_v[(a,d)(t)] &= v(t_f) - v_f \\ &= 0. \end{aligned} \tag{9.10}$$

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 [32]

$$a^*(t) = \begin{cases} 1, & 0 \leq t < 1, \\ 0, & 1 \leq t \leq 2, \end{cases} \tag{9.11}$$

$$d^*(t) = \begin{cases} 0, & 0 \leq t < 1, \\ -1, & 1 \leq t \leq 2, \end{cases} \tag{9.12}$$

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-posed, 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 $(a, b)(t; \underline{\alpha})$ 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

$$\begin{aligned} (a,d): \quad \mathbb{R} &\to \mathbb{R}^2 \\ t &\mapsto (a,d)(t; \underline{\alpha}, t_f), \end{aligned}$$

where

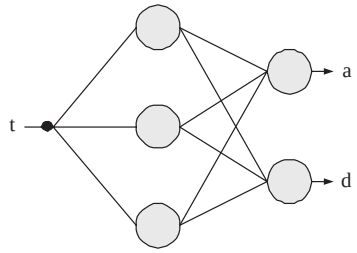

Figure 9.2: Network architecture for the car problem.

$$a(t;\underline{\alpha},t_f) = b_1^{(2)} + \sum_{j=1}^{3} w_{1j}^{(2)} \tanh\left(b_j^{(1)} + w_{j1}^{(1)} t\right), \tag{9.13}$$

$$d(t;\underline{\alpha},t_f) = b_2^{(2)} + \sum_{j=1}^{3} w_{2j}^{(2)} \tanh\left(b_j^{(1)} + w_{j1}^{(1)} t\right). \tag{9.14}$$

Equation (9.13) 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

$$a(t;\underline{\alpha},t_f) = \begin{cases} 0, & a(t;\underline{\alpha},t_f) < 0. \\ a(t;\underline{\alpha},t_f), & 0 \le a(t;\underline{\alpha},t_f) \le 1. \\ 1, & a(t;\underline{\alpha},t_f) > 1. \end{cases} \tag{9.15}$$

$$d(t;\underline{\alpha},t_f) = \begin{cases} -1, & d(t;\underline{\alpha},t_f) < -1. \\ d(t;\underline{\alpha},t_f), & -1 \le d(t;\underline{\alpha},t_f) \le 0. \\ 0, & d(t;\underline{\alpha},t_f) > 0. \end{cases} \tag{9.16}$$

All the parameters in the neural network are initialized at random.

Formulation of variational problem

From Equations (9.9), (9.10) and (9.8), the car problem formulated in this Section can be stated so as to find a control $(a,d)^*(t;\underline{\alpha}^*,t_f^*) \in V$ such that

$$E_x[(a,d)^*(t;\underline{\alpha}^*,t_f^*)] = 0, \tag{9.17}$$

$$E_v[(a,d)^*(t;\underline{\alpha}^*,t_f^*)] = 0, \tag{9.18}$$

and for which the functional

$$T[(a,d)(t;\underline{\alpha},t_f)],$$

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 $(a,d)^*(t;\underline{\alpha}^*,t_f^*)$ for which the objective functional

$$F[(a,d)(t;\underline{\alpha},t_f)] = \rho_X E_x^2 + \rho_V E_v^2 + \rho_T T, \tag{9.19}$$

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 (9.19) requires a numerical method for integration of ordinary differential equations. Here we choose the Runge-Kutta-Fehlberg method with tolerance 10^{-12} [59].

The objective function gradient vector, $\nabla f(\underline{\alpha})$, 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 [56]. 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 $1.999 \cdot 10^{-3}$.

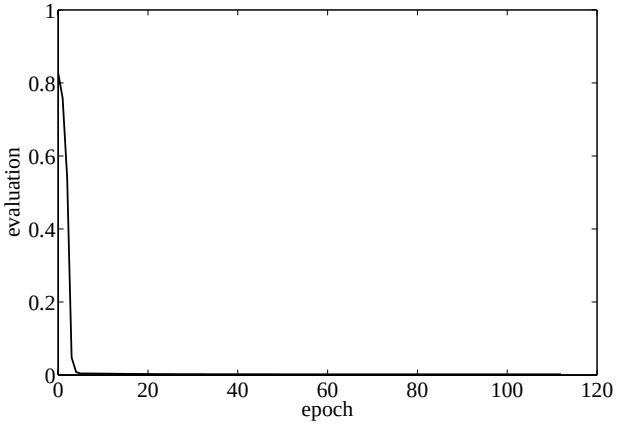

Figure 9.3: Evaluation history for the car problem.

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 (9.11). 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%.

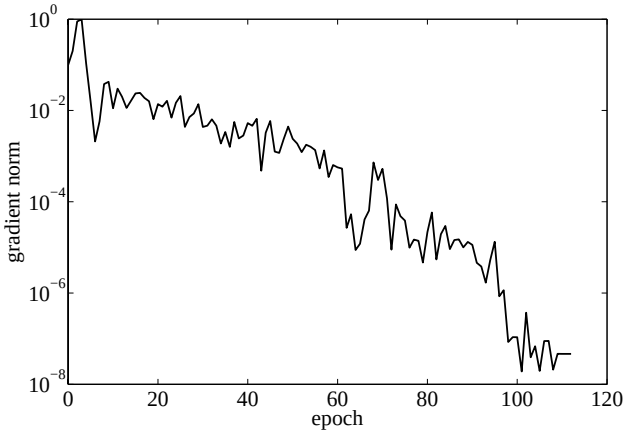

Figure 9.4: Gradient norm history for the car problem.

N	=	113
M	=	7565
CPU	=	$324s$
$\|\underline{\alpha}^*\|$	=	6.84336
$f(\underline{\alpha}^*, t_f^*)$	=	0.00199951
$e_x(\underline{\alpha}^*, t_f^*)$	=	$5.00358 \cdot 10^{-4}$
$e_v(\underline{\alpha}^*, t_f^*)$	=	$4.99823 \cdot 10^{-4}$
$\|\nabla f(\underline{\alpha}^*, t_f^*)\|/s$	=	$4.63842 \cdot 10^{-8}$
t_f^*	=	1.99901

Table 9.1: Training results for the car problem.

The analytical form of the optimal control addressed by the neural network is as follows

$$\begin{aligned} a^*(\underline{\alpha}^*, t_f^*) &= -1.31175 \\ &+ 6.85555 \tanh(-1.1448 + 1.48771t) \\ &- 0.387495 \tanh(2.52653 - 1.5223t) \\ &+ 16.1508 \tanh(12.2927 - 12.3053t), \qquad (9.20) \\ d^*(\underline{\alpha}^*, t_f^*) &= 1.82681 \\ &- 4.91867 \tanh(-1.1448 + 1.48771t) \\ &- 0.839186 \tanh(2.52653 - 1.5223t) \\ &+ 6.76623 \tanh(12.2927 - 12.3053t), \qquad (9.21) \end{aligned}$$

subject to the lower and upper bounds

$$a^*(t;\underline{\alpha}^*,t_f^*) = \begin{cases} 0, & a^*(t;\underline{\alpha}^*,t_f^*) < 0. \\ a^*(t;\underline{\alpha}^*,t_f^*), & 0 \leq a^*(t;\underline{\alpha}^*,t_f^*) \leq 1. \\ 1, & a^*(t;\underline{\alpha}^*,t_f^*) > 1. \end{cases} \quad (9.22)$$

$$d^*(t;\underline{\alpha}^*,t_f) = \begin{cases} -1, & d^*(t;\underline{\alpha}^*,t_f^*) < -1. \\ d^*(t;\underline{\alpha}^*,t_f^*), & -1 \leq d^*(t;\underline{\alpha}^*,t_f^*) \leq 0. \\ 0, & d^*(t;\underline{\alpha}^*,t_f^*) > 0. \end{cases} \quad (9.23)$$

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.

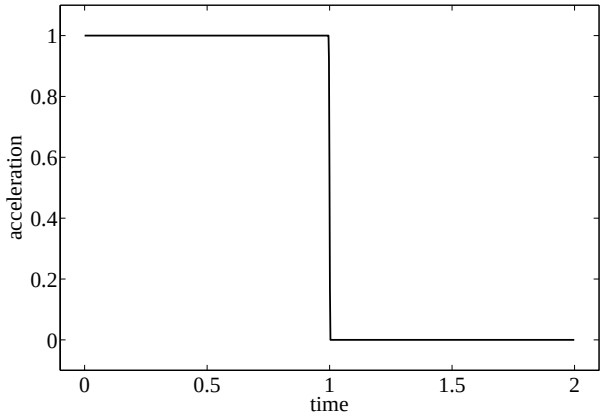

Figure 9.5: Neural network solution for the optimal acceleration in the car problem.

9.4 Related classes in Flood

The CarProblem class in Flood

The CarProblem class in `Flood` represents the concept of objective functional for the car optimal control problem. An object of this class is constructed in the following manner,

```
CarProblem carProblem(&multilayerPerceptron);
```

where &multilayerPerceptron is a reference to a MultilayerPerceptron object. In this case that neural network must have one input (the time) and two output neurons (the acceleration and the deceleration). The multilayer perceptron object must also include one independent parameter (the final time).

Lower and upper bounds must be set to the neural network for the maxinTd 9Timal

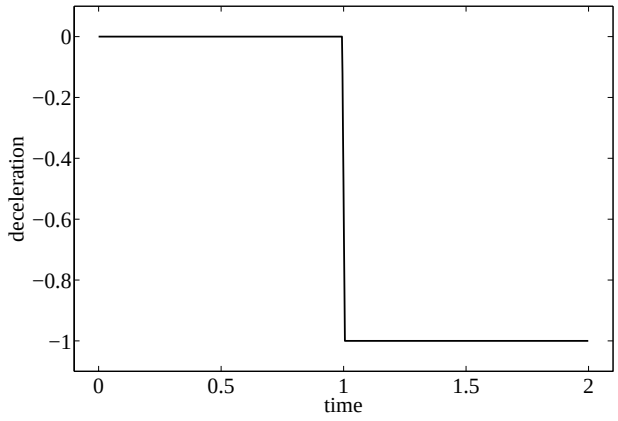

Figure 9.6: Neural network solution for the optimal deceleration in the car problem.

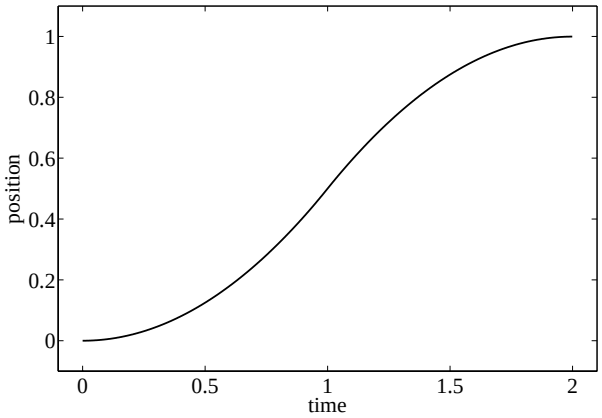

Figure 9.7: Corresponding optimal trajectory for the position in the car problem.

In order to evaluate the error terms in the objective functional expression integration of ordinary differential equations is required. This is performed making use of the OrdinaryDifferentialEquations class.

The weights of the constraints error and the objective itself can be modified with the setFinalPositionErrorWeight(**double**), setFinalVelocityErrorWeight(**double**) and setFinalTimeWeight(**double**) methods.

An example of the use of the CarProblem class can be found in the CarProblemApplication.cpp file. This is a main function contained in the Applications folder.

The FedBatchFermenterProblem class in Flood

The FedBatchFermeterProblem class is very similar to the one described above. To construct a fed batch fermenter problem the following sentence is used

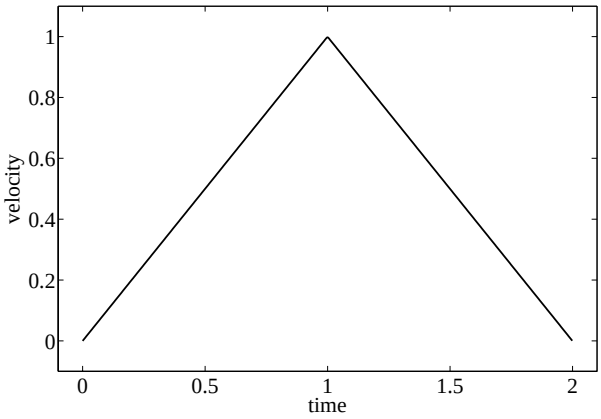

Figure 9.8: Corresponding optimal trajectory for the position in the car problem.

```
FedBatchFermenterProblem fedBatchFermeterProblem(&multilayerPerceptron);
```

where in this case the MultilayerPerceptron object must have one input and one output neuron, and should not include any independent parameter.

The FedBatchFermeterProblemApplication.cpp file shows how to build an application in order to solve this problem.

The AircrafLandingProblem class in Flood

The AircraflLandingProblem class is also included with `Flood` to represents the concept of objective functional for this optimal control problem. This class is very similar to the two other ones described in this section.

Also, the AircraftLandingProblemApplication.cpp file is an example application of this problem.

Chapter 10

Inverse problems

Here the main issues concerning the theory of inverse problems are covered. These concern variational nature and ill-posseness, so that neural networks can be trained to learn the solution of this type of problems.

Two artificially generated inverse problems are here attempted for validation, achieving good results. As they make use of the finite element method, neural networks can be seen as a valuable tool for real problems. A real application in the metallurgical industry is also satisfactorily resolved in this section.

10.1 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 [33]. 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 [33]. Inverse problems are found in many areas of science and engineering.

An inverse problem is specified by:

(i) A mathematical model of the system.

(ii) A set of actual observations to that system.

(iii) A statement of the physical constraints.

(iv) A specification of the error measure.

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

$$\mathcal{L}(\mathbf{y}(\mathbf{x}), \mathbf{u}(\mathbf{x}), \mathbf{x}) = \mathbf{f}, \tag{10.1}$$

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.

Physical constraints

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 [32].

Definition 3 (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 [32].

Definition 4 (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 $\widehat{\mathbf{u}}(\mathbf{x})$:

$$\begin{array}{rccc} E: & Y & \to & \mathbb{R} \\ & \mathbf{y}(\mathbf{x}) & \mapsto & E[\mathbf{y}(\mathbf{x})]. \end{array}$$

The error functional $E[\mathbf{y}(\mathbf{x})]$ is of the form

$$E[\mathbf{y}(\mathbf{x})] = \|\mathbf{u}(\mathbf{x}) - \widehat{\mathbf{u}}(\mathbf{x})\|, \tag{10.2}$$

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 [11].

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

Problem 7 (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

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

defined on Y, takes on a minimum value.

On the other hand, inverse problems might be ill-posed [61]. 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 [18].

Solution approach

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.

A variational formulation for neural networks provides a direct method for the solution of variational problems, and therefore inverse problems. Following the nomenclature here introduced, the inverse problem for the multilayer perceptron can be formulated as follows:

Problem 8 (Inverse problem for the multilayer perceptron) *Let Y and U be the function spaces of admissible unknowns and states, respectively. Let also V be the space consisting of all unknowns $\mathbf{y}(\mathbf{x}; \underline{\alpha})$ that a given multilayer perceptron can define, with dimension s. Find an admissible unknown $\mathbf{y}^*(\mathbf{x}; \underline{\alpha}^*) \in Y$ which causes the system*

$$\mathcal{L}(\mathbf{y}(\mathbf{x}; \underline{\alpha}), \mathbf{u}(\mathbf{x}), \mathbf{x}) = \mathbf{f}$$

to be in an admissible state $\mathbf{u}^*(\mathbf{x}) \in U$, and for which the error functional

$$E[\mathbf{y}(\mathbf{x}; \underline{\alpha})],$$

defined on V, takes on a minimum value.

10.2 Microstructural modeling of aluminium alloys

This work presents a neural networks approach for modeling the dissolution rate of hardening precipitates in aluminium alloys using inverse analysis. The effective activation energy is also in unison determined as that providing the best results.

As way of illustration, a class of multilayer perceptron extended with independent parameters is trained to find the optimal dissolution curve of hardening precipitates and the effective activation energy of aluminium alloys 2014-T6 and 7449-T79. The results from this Section are published in [41].

Introduction

In heat-treatable aluminium alloys, the dominant microstructural effects are due to changes in the precipitation state of the material, which provide the principal strengthening mechanism [50]. Modeling the dissolution rate of hardening precipitates is therefore of great importance for predicting the hardness distribution after reheating and subsequent natural ageing of the base metal. Subsequent applications of that dissolution model involve a wide range of operational conditions, such as multi-step isothermal heat treatments and continuous thermal cycles [50].

In this regard, various semi-empirical approaches to model the dissolution rate of hardening precipitates in aluminium alloys have proposed in the literature.

The dissolution model of Myhr and Grong [50] is composed from elements derived from thermodynamics, kinetic theory and simple dislocation mechanics, and provides the most important ideas from which this work is developed. It contains a single independent variable, the time, and a single state variable, the volumetric fraction of hardening precipitates. The Myhr and Grong dissolution model provides a basis for obtaining quantitative information of the reaction kinetics through simple hardness measurements, rather than through microscopic observations of the precipitates fraction.

However, Shercliff et al. [58] showed that the dissolution model of Myhr and Grong overestimates the amount of relative volumetric fraction of precipitates at the later stages of this process. They also proposed a new form for the curve of dissolution.

In this work a novel approach using neural networks is proposed for the identification of microstructural parameters for precipitates dissolution in precipitation hardenable aluminium alloys. The microstructural modeling of aluminium alloys is formulated as a variational problem including independent parameters. This makes it possible to be approached with an extended class of multilayer perceptron to construct the model of different aluminium alloys [44].

The novel methodology is here applied to two different aluminium alloys, 2014-T6 and 7449-T79. The experimental hardness data needed to construct the model is taken from both the literature and through independent measurements.

First we deal with the physical basis of the dissolution model, which are due to Myhr and Grong. The modeling problem is then formulated as a variational problem with associated independent parameters.

Then the two materials studied here, aluminium 2014-T6 and aluminium 7449-T79, are described. The experimental data actually employed in the construction of the model is also depicted.

To finish we provide the numerical results for the two case studies. The dissolution rate of hardening precipitates in these aluminium alloys is modeled, and the effective activation energy is also in unison determined as that providing the best results.

Both the source code and the experimental data used in this work can be found in the open source neural networks C++ library Flood [37].

Dissolution models for aluminium alloys

First, and assuming that the nucleation of precipitates is negligible compared to the dissolution of precipitates, the following linear relationship between the Vickers hardness and the volumetric fraction of precipitates can be established [51]

$$\frac{f}{f_0}(HV) = \frac{HV - \min(HV)}{\max(HV) - \min(HV)}, \tag{10.3}$$

where HV is the actual hardness, $\max(HV)$ is the initial hardness in the hardened state, and $\min(HV)$ is the final hardness after complete dissolution of the precipitates, which can be obtained directly from a Vickers hardness test. Equation (10.3) is extremely useful since the hardness is much easier to measure than the relative volume fraction of precipitates.

On the other hand, as it has been said above, this work is based on the model developed by Myrh and Grong [50] for cylindrical precipitates. This describes the kinetic of transformation of hardening precipitates in aluminium alloys at constant temperature by the following expression,

$$\frac{f}{f_0}(t) = 1 - \sqrt{\frac{t}{t^*(T)}}, \tag{10.4}$$

where f/f_0 is the ratio of the current volume fraction of hardening precipitates f to the initial volume fraction of hardening precipitates f_0, t is the time and $t^*(T)$ is the time for complete dissolution at temperature T. The full dissolution time $t^*(T)$ can be calculated as [50]

$$t^*(T) = t_R \exp\left[\frac{Q}{R}\left(\frac{1}{T} - \frac{1}{T_R}\right)\right], \tag{10.5}$$

where $R = 8.314 \cdot 10^{-3} kJ/molK$ is the universal gas constant, t_R is the reference time for complete dissolution at the reference temperature T_R selected for calibration, and Q is the effective activation energy for precipitates dissolution.

It is easy to see that taking $\log(1 - f/f_0)$ as a function of $\log(t/t^*)$ in Equation (10.4), the resulting plot results in a straight line of slope 0.5.

However, as it was shown by Shercliff et al. [58], the dissolution model of Myrh and Grong overestimates the dissolution rate at the later stages. Shercliff et al. proposed a new model of the form

$$\log(1 - \frac{f}{f_0}) = g(\log(\frac{t}{t^*})). \tag{10.6}$$

where g is a function whose form is given by a "look-up table". Here the early stages of dissolution follow a straight line of gradient 0.5, but this slope steadily decreases at the later stages.

The results from this work will show that although Equation (10.6) seems to give a good estimation at medium and later stages of dissolution, it underestimates the dissolution rate at the lower stages.

The formulation of Shercliff et al. also presents some other drawbacks. First, the modeling process occurs in two steps, estimation of the effective activation energy and generation of the dissolution model. This results in poor accuracy. Second, a "look-up table" is an unsuitable result here. Third, the logarithmic representation in the y-axis produces spurious effects. Indeed, separates the data at the lower stages of dissolution and joins it at the later stages.

The representation of the dissolution model proposed in this work is logarithmic in the x-axis but not in the y-axis. In particular it is of the form

$$1 - \frac{f}{f_0} = g(\log(\frac{t}{t^*})). \tag{10.7}$$

where the function g is to be found from a suitable function space.

The representation for the dissolution model of Equation (10.7) is chosen instead that of Equation (10.6) for a number of reasons. First, it makes the y-axis varying from 0 to 1 and gives all dissolution values the same importance.

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

Experimental data

Experimental tests have been performed in order to get the isothermal time evolution of Vickers hardness at different temperatures and for various aluminium alloys. In particular, two materials have been used for the isothermal heat treatments, 2014-T6 and 7449-T79 aluminium alloys.

The Vickers hardness data for aluminium alloy 2014-T6 is taken from [58], while that for aluminium alloy 7449-T79 is obtained from an independent test performed within the DEEPWELD Specific Targeted Research Project (STREP) co-funded by the 6th Framework Programme of the European Community (AST4-CT-2005-516134).

For both types of aluminium, each sample was immersed in a salt bath at different temperatures ranging from $200°C$ to $400°C$ during periods ranging from 2 seconds to about 1 day. The samples were then water quenched. Immediately after, the surface was polished and about five Vickers hardness measurements were performed at room temperature on each sample. The hardness measurements were repeated one week later to observe ageing. The total number of samples for aluminium 2014-T6 is 45, and the total number of samples aluminium 7449-T79 is 70.

Figures 10.1 and 10.2 depict these Vickers hardness test for aluminium alloys 2014-T6 and AA7449-T6, respectively. Note that, in both figures the Vickers hardness decreases with the time, due to dissolution of hardness precipitates.

Table 10.1 shows the parameters used here for both aluminium alloys needed to transform from Vickers hardness to volumetric fraction of hardening precipitates in Equation (10.3). Here $\min(HV)$ is the minimum Vickers hardness, $\max(HV)$ is the maximum Vickers hardness, $t_R[s]$, is the reference time in seconds and $T_R[K]$ is the reference temperature in Kelvin.

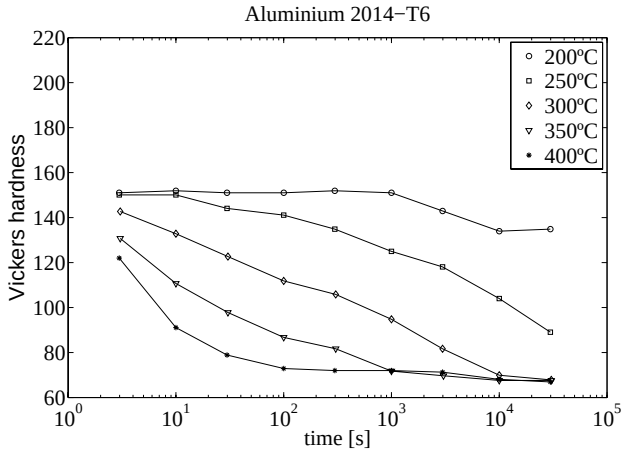

Figure 10.1: Vickers hardness test for aluminium alloy 2014-T6.

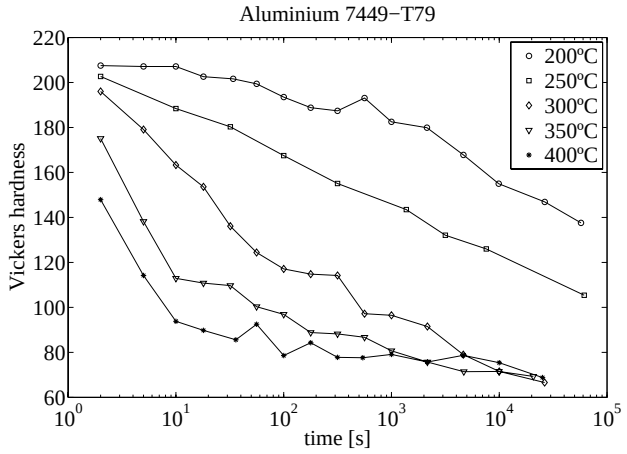

Figure 10.2: Vickers hardness test for aluminium alloy 7449-T79.

Aluminium	$\min(HV)$	$\max(HV)$	$t_R[s]$	$T_R[K]$
2014-T6	65	155	10000	573.16
7449-T79	137.6	207.5	16	623.16

Table 10.1: Parameters for aluminium alloys 2014-T6 and 7449-T79.

Numerical results

A variational formulation for neural networks is in this section applied to find the optimal dissolution model and effective activation energy of aluminium alloys 2014-T6 and 7449-T79.

Selection of function space

The first step is to choose a function space to represent the dissolution model $(1 - f/f_0)(\log(t/t^*))$. Here a multilayer perceptron with a sigmoid hidden layer and a linear output layer is used. This neural network is very useful in inverse problems, since it is a class of universal approximator [29].

The number of inputs and output neurons are constrained by the problem. Here the multilayer perceptron must have one input, $\log(t/t^*)$, and one output neuron, $1 - f/f_0$. However, the number of hidden neurons is a design variable in the problem.

In order to draw a correct network architecture for the dissolution model, different sizes for the hidden layer are compared and that providing best validation performance is chosen. In particular, the Vickers hardness tests are divided into training and validation subsets. A multilayer perceptron with 1, 2 and 3 hidden neurons are then trained with the training data and their performance against the validation data are compared. It is found that the optimal network architecture for the two case studied considered here is that with 1 hidden neuron.

Overall, the network architecture used to represent the dissolution model for aluminium 2014-T6 and aluminium 7449-T79 has one input, one sigmoid hidden neuron and one linear output neuron. This is the simplest multilayer perceptron possible. Figure 10.3 is a graphical representation of this network architecture.

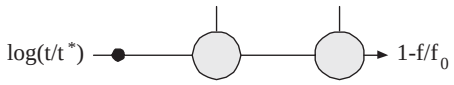

Figure 10.3: Network architecture for aluminium alloys 2014-T6 and 7449-T79.

On the other hand, and in order to obtain a correct representation for the solution, information about the effective activation energy is required. Thus, an independent parameter Q must be associated to the multilayer perceptron. Note that a standard software tool which implements the multilayer perceptron will not be capable of dealing with this problem, since it needs to include independent parameters.

This extended class of multilayer perceptron spans a family V of functions $(1 - f/f_0)(\log(t/t^*; \underline{\alpha}, Q)$ of dimension $d = 4 + 1$, where 4 is the number of biases and synaptic weights and 1 is the number of independent parameters. Elements of V are of the form

$$\begin{aligned} 1 - \frac{f}{f_0} : \quad \mathbb{R} &\rightarrow \mathbb{R} \\ \log\left(\frac{t}{t^*}\right) &\mapsto (1 - \frac{f}{f_0})(\log\left(\frac{t}{t^*}\right); \underline{\alpha}, Q). \end{aligned}$$

Here the biases and synaptic weights, $\underline{\alpha}$, are initialized at random in the interval $[-1, 1]$, and the activation energy, Q, is also initialized at random in the interval $[100, 200]$. In this way a random model with a random value for the activation energy is used as an initial guess.

Formulation of variational problem

The second step is to select an appropriate objective functional, in order to formulate the variational problem. A Minkowski error could be used here to perform robust mod-

eling. Also, a regularization term could be included to prevent overfitting. However, the Vickers hardness tests for both alloys do not show the presence of significant outliers. Also, the network architecture chosen is that providing the minimum validation error. So instead, the simpler mean squared error between the dissolution model and the volumetric fraction data is used. The mathematical expression of this objective functional is

$$F\left[(1-\frac{f}{f_0})(\log(\frac{t}{t^*});\underline{\alpha},Q)\right]=\frac{1}{S}\sum_{\mathsf{s}=1}^{\mathsf{S}}\left((1-\frac{f}{f_0})(\log(\frac{t}{t^*});\underline{\alpha},Q)^{(\mathsf{s})}-(1-\frac{f}{f_0})^{(\mathsf{s})}\right)^2 \quad (10.8)$$

where S is the number of samples available, and $(1-f/f_0)^{(\mathsf{s})}$ is calculated by means of Equation 10.3.

Note that, in Equation (10.8), the pairs of values $(\log(t/t^*), 1-f/f_0)$ are not fixed, since they depend on the particular value used for the effective activation energy Q. In this way, the optimal value for this independent parameter will provide the minimum dispersion of the data. As a consequence, a standard software tool implementing the mean squared error will not be applicable to this problem. Instead, programming the objective functional for this particular application is required.

The variational problem can then be formulated so as to find a function $(1-f/f_0)^*(\log(t/t^*);\underline{\alpha}^*,Q^*)$ and an independent parameter Q^* for which the objective functional in Equation (10.8) takes on a minimum value.

On the other hand, evaluation of the objective function gradient vector, $\nabla f(\underline{\alpha},Q)$, is performed via numerical differentiation. In particular, the central differences method is used with an epsilon value of 10^{-6}.

Solution of reduced function optimization problem

The third step is to choose a suitable training algorithm to solve the reduced function optimization problem. To check for presence of local minima a quasi-Newton method with BFGS train direction and Brent optimal train rate methods has been used from different initial guesses. The results provided are in all cases the same, and no local minima seem to appear for these particular case studies.

For both aluminium alloys, the training algorithm is set to stop when the quasi-Newton method cannot perform any better. At this point the algorithm gives zero optimal train rate in the Brent method for some gradient descent train direction. Figures 10.4 and 10.5 depict the training history for aluminium alloys 2014-T6 and 7449-T79, respectively. Note that a base 10 logarithmic scale is used for the y-axis in both plots. They depict the mean squared error value and its gradient norm as a function of the training epoch. In both figures the evaluation decreases very fast during the first few epochs to reach a stationary value. Also, the gradient norm approaches a very small value. These two signs indicate that the algorithm has fully converged to a minimum.

Table 10.2 shows the training results for these case studies. Here N denotes the number of epochs, M the number of evaluations, CPU the CPU time in a laptop AMD 3000, $f(\underline{\alpha}^*,Q^*)$ the final objective function value, $\nabla f(\underline{\alpha}^*,Q^*)$ its gradient norm and Q^* the optimum effective activation energy.

The mathematical expression of the dissolution model provided by the neural network for aluminium 2014-T6 is given by

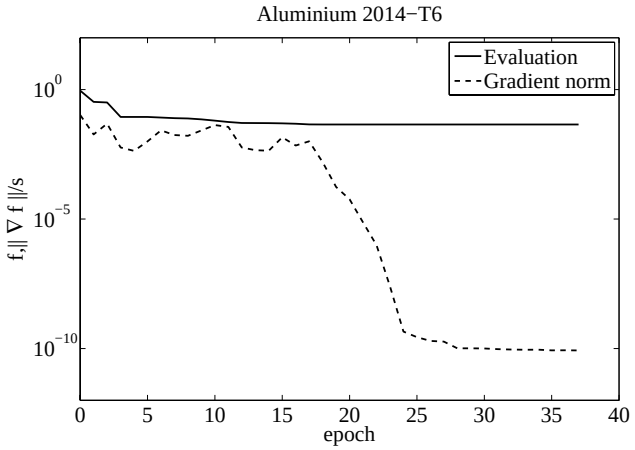

Figure 10.4: Training history for aluminium alloy 2014-T6.

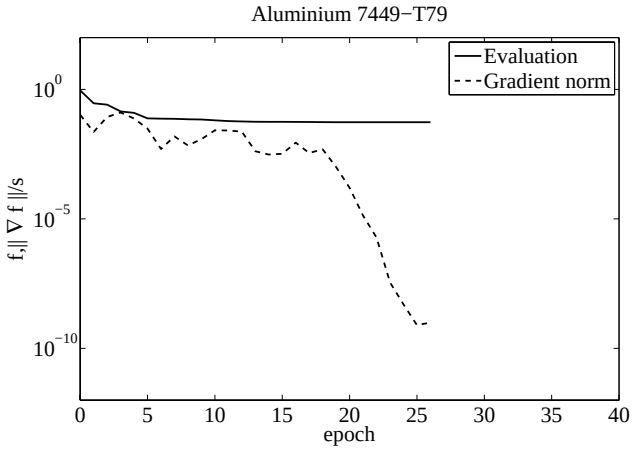

Figure 10.5: Training history for aluminium alloy 7449-T79.

Aluminium	N	M	CPU	$f(\underline{\alpha}^*, Q^*)$	$\nabla f(\underline{\alpha}^*, Q^*)$	Q^*
2014-T6	38	1614	$1\,s$	$4.465 \cdot 10^{-2}$	$8.453 \cdot 10^{-11}$	151.636
7449-T79	27	1282	$1\,s$	$5.398 \cdot 10^{-2}$	$9.854 \cdot 10^{-10}$	145.458

Table 10.2: Training results for aluminium alloys 2014-T6 and 7449-T79.

$$(1 - \frac{f}{f_0})(\log(\frac{t}{t^*}); \underline{\alpha}^*, Q^*) = \frac{1}{2}\Bigg\{0.0324727$$
$$+ \quad 0.959812 \tanh\left[1.0037 + 3.67865\left(2\frac{\log(t/t^*) + 6}{12} - 1\right)\right] + 1\Bigg\}. \quad (10.9)$$

with $Q^* = 151.636$. The mathematical expression for aluminium 7449-T79 is

$$(1 - \frac{f}{f_0})(\log(\frac{t}{t^*}); \underline{\alpha}^*, Q^*) = \frac{1}{2}\Bigg\{ - 0.0159086$$
$$+ \quad 0.974911 \tanh\left[0.259412 + 3.15452\left(2\frac{\log(t/t^*) + 6}{12} - 1\right)\right] + 1\Bigg\}. \quad (10.10)$$

with $Q^* = 145.458$.

Figures 10.6 and 10.7 show the dissolution models for aluminium 2014-T6 and aluminium 7449-T79, respectively. Comparing the shape of these curves to those reported in the literature there are similarities and differences. For medium and high stages of dissolution they are very similar tho those reported by Shercliff et al. [58]. However, these results show that the dissolution model Shercliff et al. underestimates the dissolution rate at the lower stages, while the neural networks method does not.

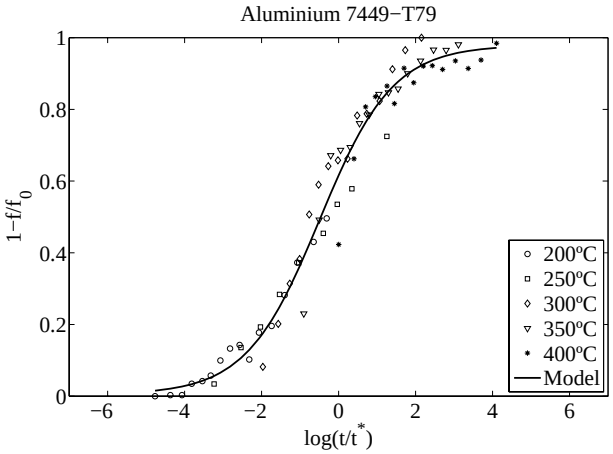

Figure 10.7: Dissolution model for aluminium alloy 7449-T79.

```
PrecipitateDissolutionModeling precipitateDissolutionModeling(&multilayerPerceptron);
```

where &multilayerPerceptron is a reference to a MultilayerPerceptron object. This neural network must have one input and one output neurons for the normalized time and the dissolution rate, respectively. It must also include one independent parameter, for the activation energy value.

The method loadVickersHardnessTest(**char***) loads the experimental data into memory. Look at the PrecipitateDissolutionModeling folder inside Data for the format of this file.

On the other hand the getEvaluation(**void**) method returns the error between the outputs from the neural network and the dissolution values provided by the algebraic model, and for the actual activation energy in the independent parameter.

On the other hand, a quite general regularized Minkowski error is implemente in this class. The Minkowski parameter, the Minkowski error weight and the regularization weight modify the objective functional expression.

An example of the use of the PrecipitateDissolutionModeling class can be found in the PrecipitateDissolutionModelingApplication.cpp file. This is a main function contained in the Applications folder.

Chapter 11

Optimal shape design

This section includes a short survey to optimal shape design, a theory which is developed from the abstract field of calculus of variations. In this way, optimal shape design is here stated as a new learning task for the multilayer perceptron.

The application of this approach is validated through an optimal shape design problem whose exact solution is known. More practical implementation issues are studied by using a neural network to design an airfoil with given thickness and providing maximum aerodynamic efficiency for transonic flight conditions.

11.1 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 [47]. 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:

(i) A mathematical model of the system.

(ii) A statement of the physical constraints.

(iii) A specification of the performance criterion.

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

$$\mathcal{L}(\mathbf{y}(\mathbf{x}), \mathbf{u}(\mathbf{x}), \mathbf{x}) = \mathbf{f}, \tag{11.1}$$

where $\mathcal{L}$ is some algebraic or differential operator and $\mathbf{f}$ some forcing term.

Physical constraints

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

Two important types of shape 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 shape variables are restricted to fall in some interval, then the problem is said to have lower and upper bounds.

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 5 (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 6 (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.*

Performance criterion

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

$$\begin{array}{rccc} F: & Y & \to & \mathbb{R} \\ & \mathbf{y}(\mathbf{x}) & \mapsto & F[\mathbf{y}(\mathbf{x})]. \end{array}$$

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

To sum up, the optimal shape design problem can be formulated as

Problem 9 (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 performance criterion*

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

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.*

Solution methods

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. From the nomenclature introduced in this section, the optimal shape design problem for the multilayer perceptron can be written as follows:

Problem 10 (Optimal shape design problem for the multilayer perceptron) *Let Y and U be the function spaces of admissible shapes and states, respectively. Let also V be the space consisting of all shapes $\mathbf{y}(\mathbf{x}; \underline{\alpha})$ that a given multilayer perceptron can define, with dimension s. Find an admissible shape $\mathbf{y}^*(\mathbf{x}; \underline{\alpha}^*) \in Y$ which causes the system*

$$\mathcal{L}(\mathbf{y}(\mathbf{x}; \underline{\alpha}), \mathbf{u}(\mathbf{x}), \mathbf{x}) = \mathbf{f}$$

to be in an admissible state $\mathbf{u}^*(\mathbf{x}) \in U$, and for which the performance criterion

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

defined on V, takes on a minimum or a maximum value.

11.2 The minimum drag problem

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 [37], and it has been published in [40] and [39].

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 11.1.

The drag of such a body $y(x)$ can be expressed as

$$D[y(x)] = 2\pi q \int_0^l y(x)[C_p y'(x)]dx, \tag{11.2}$$

where q is the free-stream dynamic pressure and C_p the pressure coefficient [10]. For a slender body $y'(x) \ll 1$, the pressure coefficient can be approximated by the Newtonian flow relation

$$C_p = 2\left[y'(x)\right]^2, \tag{11.3}$$

which is valid provided that the inequality $y'(x) \geq 0$ is satisfied.

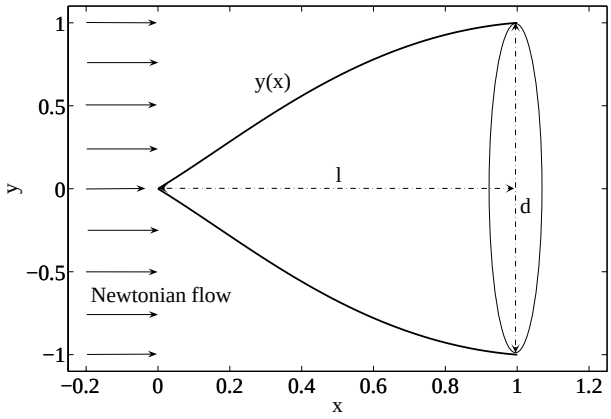

Figure 11.1: The minimum drag problem statement.

From Equations (11.2) and (11.3) we obtain the following approximation for the drag,

$$D[y(x)] = 4\pi q \int_0^l y(x)[y'(x)]^3 dx. \tag{11.4}$$

It is convenient to introduce the following dimensionless variables associated with the axial coordinate and the radial coordinate

$$\xi = \frac{x}{l}, \tag{11.5}$$

$$\eta = \frac{2y}{d}. \tag{11.6}$$

In that way, both ξ and η vary from 0 to 1.

Also, a dimensionless coefficient associated with the drag can be defined as

$$C_D[\eta(\xi)] = \tau^2 \int_0^1 \eta(\xi)[\eta'(\xi)]^3 d\xi, \tag{11.7}$$

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

$$\eta^*(\xi) = \xi^{3=4}, \tag{11.8}$$

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 11.2.

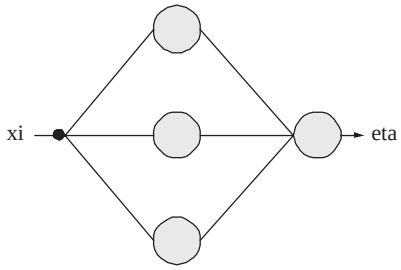

Figure 11.2: Network architecture for the minimum drag problem.

Such a multilayer perceptron spans a family V of parameterized functions $\eta(\xi;\underline{\alpha})$ of dimension $s = 10$, which is the number of parameters in the network. The elements of this function space are of the form

$$\begin{array}{rccc} \eta: & \mathbb{R} & \rightarrow & \mathbb{R} \\ & \xi & \mapsto & \eta(\xi;\underline{\alpha}), \end{array}$$

where

$$\eta(\xi;\underline{\alpha}) = b_1^{(2)} + \sum_{j=1}^{3} w_{1j}^{(2)} \cdot \tanh\left(b_j^{(1)} + w_{j1}^{(1)}\xi\right). \tag{11.9}$$

The outputs from the neural network in Figure 11.2 must hold the boundary conditions $\eta(0) = 0$ and $\eta(1) = 1$. A suitable set of particular and homogeneous solution terms here is

$$\varphi_0(\xi) = \xi, \tag{11.10}$$
$$\varphi_1(\xi) = \xi(\xi - 1), \tag{11.11}$$

respectively. This gives

$$\eta(\xi;\underline{\alpha}) = \xi + \xi(\xi - 1)\eta(\xi;\underline{\alpha}). \tag{11.12}$$

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

$$\eta(\xi;\underline{\alpha}) = \begin{cases} 0, & \eta(\xi;\underline{\alpha}) < 0. \\ \eta(\xi;\underline{\alpha}), & 0 \leq \eta(\xi;\underline{\alpha}) \leq 1. \\ 1, & \eta(\xi;\underline{\alpha}) > 1. \end{cases} \tag{11.13}$$

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 11.3 depicts the starting random shape for this problem.

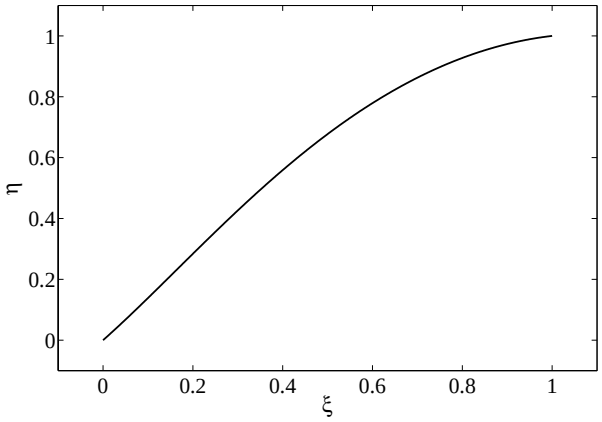

Figure 11.3: Initial guess for the minimum drag problem.

Formulation of variational problem

From Equation (11.7), the variational statement of this problem is to find a function $\eta^*(\xi;\underline{\alpha}^*) \in V$ for which the functional

$$C_D[\eta(\xi;\underline{\alpha})]/\tau^2 = \int_0^1 \eta(\xi;\underline{\alpha})[\eta'(\xi;\underline{\alpha})]^3 d\xi, \tag{11.14}$$

defined on V, takes on a minimum value.

In order to evaluate the objective functional in Equation (11.14) the integration of a function is needed. Here we apply the Runge-Kutta-Fehlberg method [59] with tolerance 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 for training [8]. The tolerance in the Brent's method is set to 10^{-6}.

The objective function gradient vector $\nabla f(\underline{\alpha})$ 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 11.4 and 11.5 illustrate this training process.

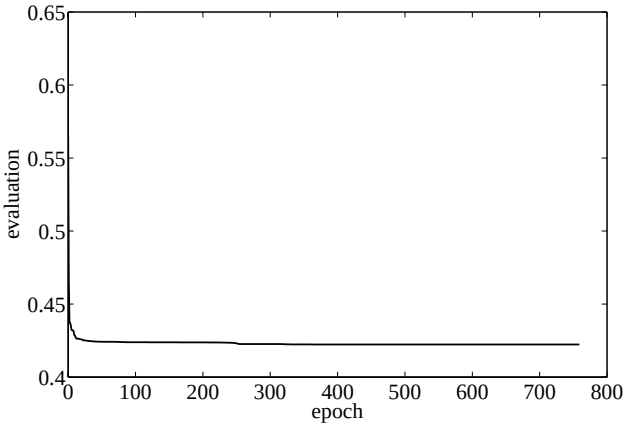

Figure 11.4: Evaluation history for the minimum drag problem.

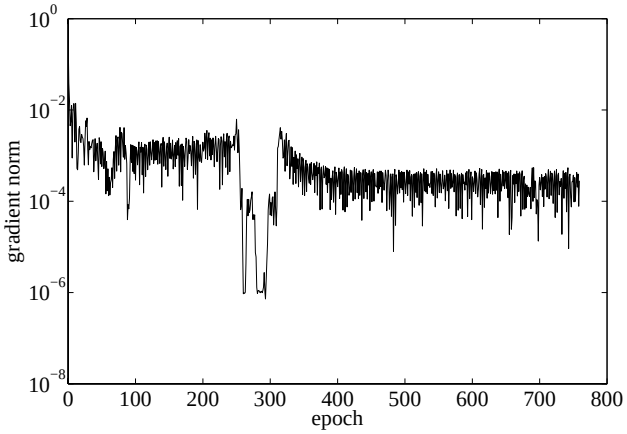

Figure 11.5: Gradient norm history for the minimum drag problem.

Table 11.1 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, $\|\underline{\alpha}^*\|$ the final parameters norm, $f(\underline{\alpha}^*)$ the final value for the

objective function and $\|\nabla f(\underline{\alpha}^*)\|$ the final gradient norm.

N	=	759
M	=	38426
CPU	=	354
$\|\underline{\alpha}^*\|$	=	122.752
$f(\underline{\alpha}^*)$	=	0.422325
$\|\nabla f(\underline{\alpha}^*)\|$	=	$2.80939 \cdot 10^{-4}$

Table 11.1: Training results for the minimum drag problem.

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 11.6.

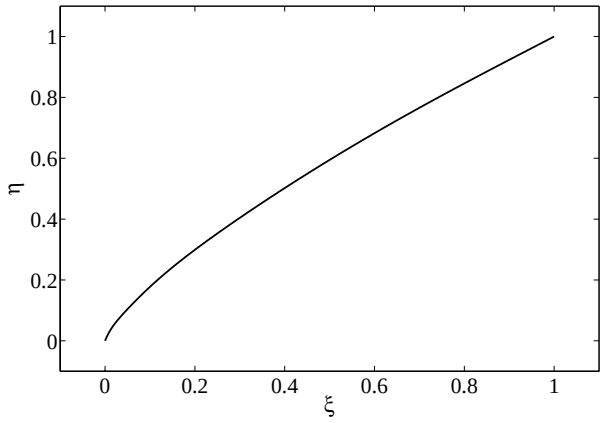

Figure 11.6: Neural network results to the minimum drag problem.

Finally, an explicit expression of the shape of such an axisymmetric body is given by

$$\begin{aligned}
\eta^*(\xi;\underline{\alpha}^*) &= \xi + \xi(\xi-1)[-164.639 \\
&- 275.014\tanh(-2.97601 - 27.2435\xi) \\
&- 79.9614\tanh(-2.62125 - 3.7741\xi) \\
&+ 201.922\tanh(-1.78294 + 0.0113036\xi)]. \qquad (11.15)
\end{aligned}$$

11.3 Related classes in Flood

Chapter 12

Function optimization problems

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 member.

12.1 Problem formulation

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.

12.2 The unconstrained function optimization problem

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 11 (Unconstrained function optimization problem) *Let $X \subseteq \mathbb{R}^n$ be a real vector space. Find a vector $\mathbf{x}^* \in X$ for which the function*

$$\begin{aligned} f : X &\rightarrow \mathbb{R} \\ \mathbf{x} &\mapsto f(\mathbf{x}) \end{aligned}$$

takes on a minimum or a maximum value.

The function $f(\mathbf{x})$ is called the objective function. The domain of the objective function for a function optimization problem is a subset X of $\mathbb{R}^n$, and the image of

that function is the set $\mathbb{R}$. The integer n 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 $f(\mathbf{x})$ is equivalent to minimization of $-f(\mathbf{x})$, 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.

12.3 The constrained function optimization problem

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 12 (Constrained function optimization problem) *Let $X \subseteq \mathbb{R}^n$ be a real vector space. Find a vector $\mathbf{x}^* \in X$ such that the functions*

$$\begin{aligned} c_i : X &\rightarrow \mathbb{R} \\ \mathbf{x} &\mapsto c_i(\mathbf{x}) \end{aligned}$$

hold $c_i(\mathbf{x}^*) = 0$, for $i = 1, \ldots, l$, and for which the function

$$\begin{aligned} f : X &\rightarrow \mathbb{R} \\ \mathbf{x} &\mapsto f(\mathbf{x}) \end{aligned}$$

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 13 (Reduced constrained function optimization problem) *Let $X \subseteq \mathbb{R}^n$ be a real vector space, and let $\rho_i > 0$, for $i = 1, \ldots, l$, be real numbers. Find a vector $\mathbf{x}^* \in X$ for which the function*

$$\begin{aligned} \bar{f} : X &\rightarrow \mathbb{R} \\ \mathbf{x} &\mapsto \bar{f}(\mathbf{x}), \end{aligned}$$

defined by

$$\bar{f}(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^{l} \rho_i \left(c_i(\mathbf{x})\right)^2,$$

takes on a minimum value.

The parameters ρ_i, $i = 1, \ldots, l$, are called the penalty term ratios, 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 the ratios ρ_i, $i = 1, \ldots, l$, it is clear that the solution $\mathbf{x}^*$ of Problem 13 will be in in a region where $c_i(\mathbf{x})$, $i = 1, \ldots, l$, are small. Thus, for increasing values of ρ_i, $i = 1, \ldots, l$, it is expected that the the solution $\mathbf{x}^*$ of Problem 13 will approach the constraints and, subject to being close, will minimize the objective function $f(\mathbf{x})$. Ideally then, as $\rho_i \rightarrow \infty$, $i = 1, \ldots, l$, the solution of Problem 13 will converge to the solution of Problem 12 [45].

12.4 The objective function gradient vector

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:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right). \tag{12.1}$$

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 central differences method

$$\frac{\partial f}{\partial x_i} = \frac{f(x_i + h) - f(x_i - h)}{2h} + \mathcal{O}(h^2), \tag{12.2}$$

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

12.5 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:

$$Hf = \begin{pmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \cdots & \dfrac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{pmatrix} \tag{12.3}$$

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 central differences method

$$\begin{aligned} \frac{\partial^2 f}{\partial x_i \partial x_j} &= \frac{f(x_i + \epsilon, x_j + \epsilon)}{4\epsilon^2} - \frac{f(x_i + \epsilon, x_j - \epsilon)}{4\epsilon^2} \\ &- \frac{f(x_i - \epsilon, x_j + \epsilon)}{4\epsilon^2} + \frac{f(x_i - \epsilon, x_j - \epsilon)}{4\epsilon^2} + \mathcal{O}(h^2). \end{aligned} \tag{12.4}$$

12.6 Sample applications

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

12.6.1 The De Jong's function optimization problem

One of the simplest test functions for optimization is the De Jong's function, which is an unconstrained and unimodal function. The De Jong's function optimization problem in n variables can be stated as:

Problem 14 (De Jong's function optimization problem) *Let $X = [-5.12, 5.12]^n$ be a real vector space. Find a vector $\mathbf{x}^* \in X$ for which the function*

$$\begin{aligned} f : X &\rightarrow \mathbb{R} \\ \mathbf{x} &\mapsto f(\mathbf{x}), \end{aligned}$$

defined by

$$f(\mathbf{x}) = \sum_{i=1}^{n} x_i^2, \tag{12.5}$$

takes on a minimum value.

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

The gradient vector for the De Jong's function is given by

$$\nabla f = (2x_1, \ldots, 2x_n), \tag{12.6}$$

and the Hessian matrix by

$$Hf = \begin{pmatrix} 2 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 2 \end{pmatrix} \tag{12.7}$$

12.6.2 The Rosenbrock's function optimization problem

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

Problem 15 (Rosenbrock's function optimization problem) *Let $X = [-2.048, 2.048]^n$ be a real vector space. Find a vector $\mathbf{x}^* \in X$ for which the function*

$$\begin{aligned} f : X &\rightarrow \mathbb{R} \\ \mathbf{x} &\mapsto f(\mathbf{x}), \end{aligned}$$

defined by

$$f(\mathbf{x}) = \sum_{i=1}^{n-1} 100 \left(x_{i+1} - x_i^2\right)^2 + (1 - x_i)^2 \tag{12.8}$$

takes on a minimum value.

The minimal argument of the Rosenbrock's function is found at $\mathbf{x}^* = (1, \ldots, 1)$. The minimum value of that function is $f(\mathbf{x}^*) = 0$.

12.6.3 The Rastrigin's function optimization problem

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

Problem 16 (Rastrigin's function optimization problem) *Let $X = [-5.12, 5.12]^n$ be a real vector space. Find a vector $\mathbf{x}^* \in X$ for which the function*

$$\begin{aligned} f : X &\rightarrow \mathbb{R} \\ \mathbf{x} &\mapsto f(\mathbf{x}), \end{aligned}$$

defined by

$$f(\mathbf{x}) = 10n + \sum_{i=1}^{n} (x_i^2 - 10\cos(2\pi x_i)) \tag{12.9}$$

takes on a minimum value.

The global minimum of the Rastrigin's Function is at $x_i^* = 0$. At this minimal argument the value of the function is $f(\mathbf{x}) = 0$.

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

$$\nabla f = (2x_1 + 10\sin(2\pi x_1)2\pi, \ldots, 2x_n + 10\sin(2\pi x_n)2\pi), \tag{12.10}$$

and the Hessian matrix by

$$Hf = \begin{pmatrix} 2 + 10\cos(2\pi x_1)4\pi^2 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 2 + 10\cos(2\pi x_n)4\pi^2 \end{pmatrix} \tag{12.11}$$

12.6.4 The Plane-Cylinder function optimization problem

The problem in this example is to find the minimum point on the plane $x_1 + x_2 = 1$ which also lies in the cylinder $x_1^2 + x_2^2 = 1$.

This constrained function optimization problem can be stated as:

Problem 17 (Plane-cylinder function optimization problem) *Let $X = [-1, 1]^2$ be a real vector space. Find a vector $\mathbf{x}^* \in X$ such that the function*

$$\begin{aligned} c : X &\to \mathbb{R} \\ \mathbf{x} &\mapsto c(\mathbf{x}), \end{aligned}$$

defined by

$$c(\mathbf{x}) = x_1^2 + x_2^2 - 1, \tag{12.12}$$

holds $c(\mathbf{x}^*) \leq 0$ and for which the function

$$\begin{aligned} f : X &\to \mathbb{R} \\ \mathbf{x} &\mapsto c(\mathbf{x}), \end{aligned}$$

defined by

$$f(\mathbf{x}) = x_1 + x_2 - 1, \tag{12.13}$$

takes on a minimum value.

The solution to the plane-cylinder function optimization problem is $(x_1^*, x_2^*) = (0, 0)$.

This constrained problem can be reduced to an unconstrained problem by the use of a penalty function:

Problem 18 (Reduced plane-cylinder function optimization problem) *Let $X \subseteq \mathbb{R}^2$ be a real vector space, and let $\rho\mathbb{R}^+$ be a positive real number. Find a vector $\mathbf{x}^* \in X$ for which the function*

$$\begin{aligned} \bar{f} : X &\rightarrow \mathbb{R} \\ \mathbf{x} &\mapsto \bar{f}(\mathbf{x}), \end{aligned}$$

defined by

$$\bar{f}(\mathbf{x}) = x_1 + x_2 - 1 + \rho \left(x_1^2 + x_2^2 - 1 \right)^2,$$

takes on a minimum value.

12.7 Related classes in Flood

`Flood` includes the classes `DeJongFunction`, `RosenbrockFunction`, `RastriginFunction`, and `PlaneCylinder` to represent the concepts of the De Jong's, Rosenbrock's, Rastrigin's and Plane-Cylinder objective functions, respectively.

```
MultilayerPerceptron multilayerPerceptron;

multilayerPerceptron.setNumberOfIndependentParameters(2);
```

To construct the default De Jong's function object with 2 variables we can use the following sentence:

```
DeJongFunction deJongFunction;
```

The `calculateEvaluation(void)` method returns the evaluation of the objective function for a given argument,

```
Vector<double> argument(2, 0.0);

argument[0] = 1.2;
argument[1] = -3.6;

multilayerPerceptron.setIndependentParameters(argument);

double evaluation = deJongFunction.calculateEvaluation();
```

On the other hand, the method `calculateGradient(void)` returns the objective function gradient vector for a given argument,

```
Vector<double> gradient = deJongFunction.calculateGradient();
```

Similarly, the `calculateHessian(void)` method returns the objective function Hessian matrix for a given argument,

```
Matrix<double> hessian = deJongFunction.calculateHessian();
```

The source code of a sample application with the `DeJongFunction` class is listed in the file `DeJongFunctionApplication.cpp`.

12.7.1 The RosenbrockFunction class

To construct the default Rosenbrock's function object with 2 variables object we use

```
RosenbrockFunction rosenbrockFunction(2);
```

The `setNumberOfVariables(int)` method sets a new number of variables in an objective function. For example, to set 3 variables in the Rosenbrock's function we can write

```
int numberOfVariables = 3;

deJongFunction.setNumberOfVariables(numberOfVariables);
```

The `setLowerBound(Vector<double>)` and `setUpperBound(Vector<double>)` methods set new lower and upper bounds in the the domain of the objective function,

```
Vector<double> lowerBound(numberOfVariables, -5.12);

Vector<double> upperBound(numberOfVariables, 5.12);

rosenbrockFunction.setLowerBound(lowerBound);

rosenbrockFunction.setUpperBound(upperBound);
```

The `calculateEvaluation(void)` method returns the evaluation of the objective function for a given argument,

```
Vector<double> argument(2, 0.0);

argument[0] = 1.2; argument[1] = -3.6;

double evaluation = deJongFunction.calculateEvaluation();
```

On the other hand, the method `calculateGradient(void)` returns the objective function gradient vector for a given argument,

```
Vector<double> gradient =
deJongFunction.calculateGradient(argument);
```

Similarly, the `calculateHessian(void)` method returns the objective function Hessian matrix for a given argument,

```
double gradientNorm =
deJongFunction.calculateGradientNorm(gradient);
```

We list below the source code of a sample application with the `DeJongFunction` class.

12.7.2 The RastriginFunction class

To construct the default Rastrigin's function object with 2 variables we can use the following sentence:

```
RastriginFunction rastriginFunction;
```

12.7.3 The PlaneCylinder class

To construct the default plane-cylinder objective function object we can use the following sentence:

```
PlaneCylinder planeCylinder;
```

The `calculateError(Vector<double>)` method returns the evaluation of the objective function for a given argument,

```
Vector<double> argument(2, 1.5);

argument[0] = 1.2; argument[1] = -3.6;

double error = planeCylinder.calculateError();
```

Appendix A

The software model of Flood

Neural networks have been described as a practical tool for the solution of variational problems. This includes data modeling, optimal control, inverse analysis and optimal shape design.

In this Appendix we model a software implementation for the multilayer perceptron. 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 small part of this work has been published in [43].

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 [57].

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 [60]. 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 [57].

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.

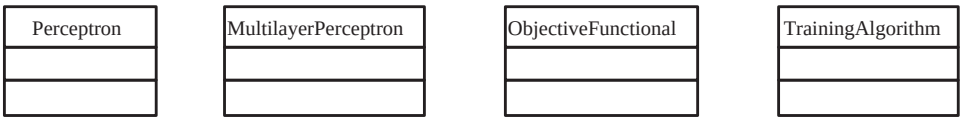

Figure A.1: A conceptual diagram for the multilayer perceptron.

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 [60]. 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 [57].

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.

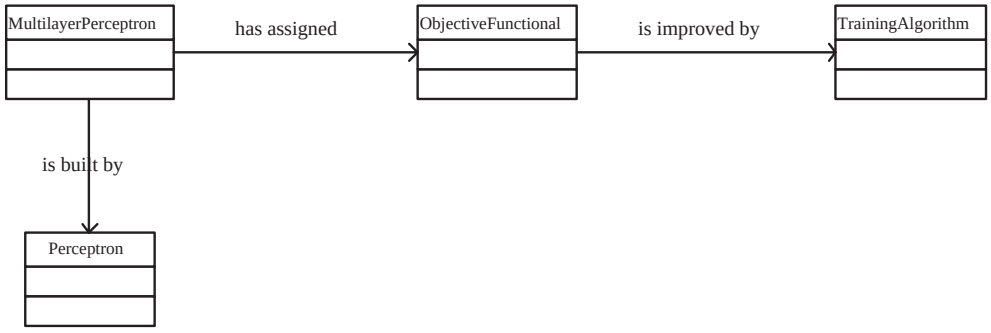

Figure A.2: Aggregation of associations to the conceptual diagram.

A.4 Derived classes

In object-oriented programming, some classes are designed only as a parent from which subclasses 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 [60]. Associations between a base class an a derived class are of the kind *is a* [57].

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, the Minkowski error or the regularized Minkowski error. Therefore the `SumSquaredError`, `MeanSquaredError`, `RootMeanSquaredError`, `NormalizedSquaredError`, `MinkowskiError` and `RegularizedMinkowskiError` 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 [15].

Training algorithm The class `TrainingAlgorithm` is abstract, because it does not represent a training algorithm for an objective function of a multilayer perceptron.

The simplest training algorithm for the multilayer perceptron, which is applicable to any differentiable objective function, is gradient descent, but it has some limitations. A faster training algorithm, with the same properties as gradient descent, is conjugate gradient. Thus, classes representing the gradient descent and the conjugate gradient training algorithms are derived. These are called `GradientDescent` and `ConjugateGradient`, respectively.

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 [60]. In UML class diagrams, classes are

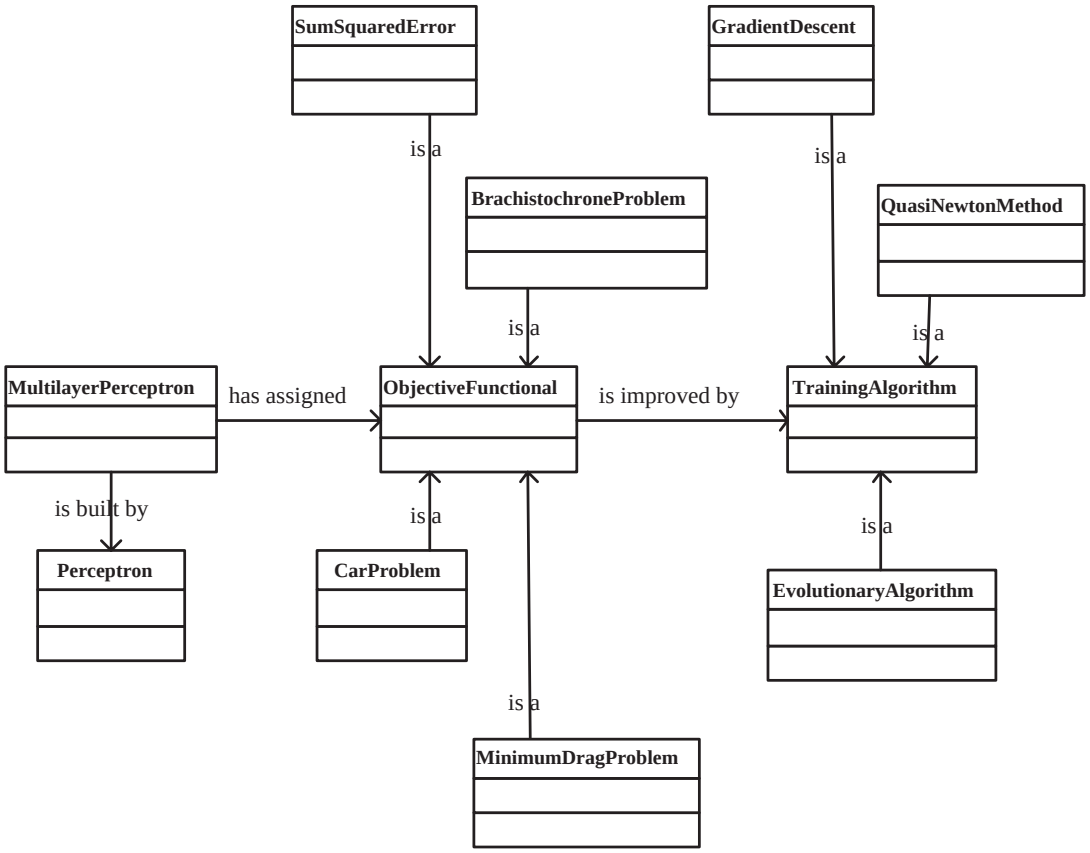

Figure A.3: Aggregation of derived classes to the association diagram.

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 [57].

A.5.1 Perceptron

A perceptron neuron model has the following attributes:

1. A number of neuron inputs.
2. A set of synaptic weights.
3. A bias.

and performs the following operations:

1. Calculate the output signal for a given set of input signals.
2. Calculate the output signal derivative for a given set of input signals.

Figure A.4 shows the class `Perceptron`, including the attributes and the operations of the base class and the derived classes.

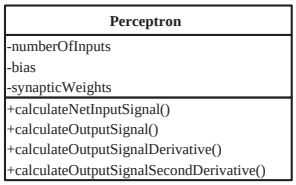

Figure A.4: Attributes and operations of the Perceptron classes.

A.5.2 Multilayer perceptron

A multilayer perceptron has the following attributes:

1. A number of network inputs.
2. A hidden layer of a given number of sigmoid neurons.
3. An output layer of a given number of linear neurons.

and performs the following operations:

1. Calculate the set of output signals for a given set of input signals.
1. Calculate the Jacobian matrix for a given set of input signals.

Figure A.5 shows the attributes and operations of the class MultilayerPerceptron.

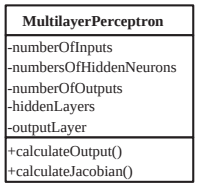

Figure A.5: Attributes and operations of the class MultilayerPerceptron.

A.5.3 Objective functional

An objective functional for a multilayer perceptron has the following attributes:

1. A relationship to a multilayer perceptron. In C++ this is implemented as a pointer to a multilayer perceptron object.

and performs the following operations:

1. Calculate the evaluation of a multilayer perceptron.
2. Calculate the objective function gradient vector of a multilayer perceptron.

On the other hand, a training data set has the following attributes:

1. A number of samples.
2. A number of input variables.
3. A number of target variables.
4. A set of input data.

5. A set of target data.

and performs the following operations:

1. Load the training data set from a data file.

Last, the utility class for numerical integration has no attributes, and performs a unique operation:

1. Calculate the definite integral of a function.

Figure A.6 shows the class `ObjectiveFunctional` together with its derived classes. It includes the attributes and the operations of the base class and the derived classes.

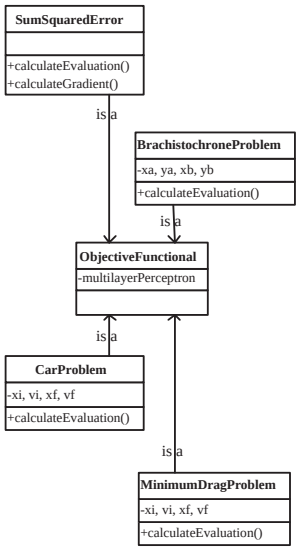

Figure A.6: Attributes and operation of the ObjectiveFunctional classes.

A.5.4 Training algorithm

A training algorithm for a multilayer perceptron has the following attributes:

1. A relationship to an objective functional for a multilayer perceptron. In C++ this is implemented as a pointer to an objective functional object.
2. A set of training parameters.

and performs the following operations:

1. Train a multilayer perceptron.

Figure A.7 shows the class `TrainingAlgorithm` together with its derived classes. It includes the attributes and the operations of the base class and the derived classes.

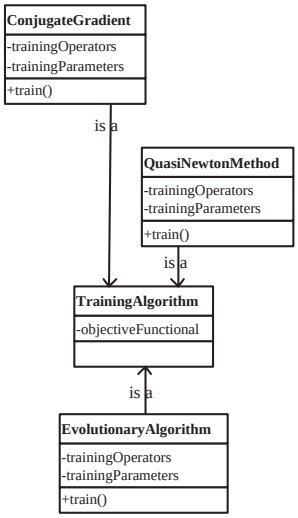

Figure A.7: Attributes and operations of the TrainingAlgorithm classes.

Appendix B

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.

B.1 Integration of functions

B.1.1 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 $[a, b]$,

$$I[y(x)] = \int_a^b f(x)dx. \tag{B.1}$$

There are a wide range of methods available for numerical integration [56]. The numerical computation of an integral is sometimes called quadrature.

B.1.2 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

$$\int_a^b f(x) \sim \int_a^b P(x). \tag{B.2}$$

To find the fitting polynomials, the Newton-Cotes formulas use Lagrange interpolating polynomials [59]. 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 [65]

$$\begin{aligned} \int_{x_1}^{x_2} f(x)dx &= h\left[\frac{1}{2}f_1 + \frac{1}{2}f_2\right] \\ &+ \mathcal{O}(h^3 f''), \end{aligned} \tag{B.3}$$

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 [65]

$$\begin{aligned} \int_{x_1}^{x_3} f(x)dx &= h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{1}{3}f_3\right] \\ &+ \mathcal{O}(h^5 f^{(4)}). \end{aligned} \tag{B.4}$$

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.

B.1.3 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 [59]. 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 [65]

$$\begin{aligned} \int_{x_1}^{x_n} f(x)dx &= h\left[\frac{1}{2}f_1 + f_2 + f_{n-1} + \frac{1}{2}f_n\right] \\ &+ \mathcal{O}\left(\frac{(b-a)^3 f''}{N^2}\right). \end{aligned} \tag{B.5}$$

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 [65]

$$\begin{aligned}\int_{x_1}^{x_n} f(x)dx &= h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{2}{3}f_3 + \frac{4}{3}f_4 + \ldots + \frac{2}{3}f_{n-2} + \frac{4}{3}f_{n-1} + \frac{1}{3}f_n\right] \\ &+ \mathcal{O}\left(\frac{1}{N^4}\right). \end{aligned} \tag{B.6}$$

B.1.4 Ordinary differential equation approach

The evaluation of the integral (B.1) is equivalent to solving for the value $I \equiv y(b)$ the ordinary differential equation

$$\frac{dy}{dx} = f(x), \tag{B.7}$$

$$y(a) = 0. \tag{B.8}$$

Section B.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.

B.2 Ordinary differential equations

B.2.1 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

$$F(x, y(x), y'(x), \ldots, y^{(n)}(x)) = 0, \tag{B.9}$$

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

$$\frac{dy_i(x)}{dx} = f_i(x, y_1, \ldots, y_N), \tag{B.10}$$

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).

B.2.2 The Euler method

The formula for the Euler method is

$$y_{n+1} = y_n + hf(x_n, y_n) + \mathcal{O}(h^2), \tag{B.11}$$

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 [56].

B.2.3 The Runge-Kutta method

Consider the use of a step like (B.11) 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

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1) \end{aligned}$$

y

The embedded fourth order formula is

$$y^*_{n+1} = y_n + c^*_1 k_1 + c^*_2 k_2 + c^*_3 k_3 + c^*_4 k_4 + c^*_5 k_5 + c^*_6 k_6 + \mathcal{O}(h^5) \tag{B.15}$$

And so the error estimate is

$$\begin{aligned} \Delta &\equiv y_{n+1} - y^*_{n+1} \\ &= \sum_{i=1}^{6} (c_i - c^*_i) k_i. \end{aligned} \tag{B.16}$$

The particular values of the various constants that we favor are those found by Cash and Karp [12], and given in Table B.1. These give a more efficient method than Fehlberg's original values [56].

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 B.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 (B.14) and (B.15), 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

$$h_0 = h_1 \left| \frac{\Delta_0}{\Delta_1} \right|^{0.2} \tag{B.17}$$

Henceforth we will let Δ_0 denote the desired accuracy. Then, Equation (B.17) 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.

$$h_0 = \begin{cases} S h_1 \left| \frac{\Delta_0}{\Delta_1} \right|^{0.20} & \Delta_0 \geq \Delta_1 \\ S h_1 \left| \frac{\Delta_0}{\Delta_1} \right|^{0.25} & \Delta_0 < \Delta_1 \end{cases} \tag{B.18}$$

B.3 Partial differential equations

B.3.1 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 [56].

The prototypical example of a hyperbolic equation is the wave equation

$$\frac{\partial^2 u(\mathbf{x}, t)}{\partial t^2} = v^2 \nabla^2 u(\mathbf{x}, t), \tag{B.19}$$

where v is the velocity of wave propagation.

The prototypical parabolic equation is the diffusion equation

$$\frac{\partial u(\mathbf{x}, t)}{\partial t} = \nabla(D \nabla u(\mathbf{x}, t)), \tag{B.20}$$

where D is the diffusion coefficient.

The prototypical elliptic equation is the Poisson equation

$$\nabla^2 u(\mathbf{x}) = \rho(\mathbf{x}), \tag{B.21}$$

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 [56]. Equations (B.19) and (B.20) both define initial value (or Cauchy) problems. By contrast, Equation (B.21) 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 [56].

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 [56].

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.

B.3.2 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 [28].

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 [49]. 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.

B.3.3 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 [28].

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 [68].

Bibliography

[1] R.K. Amiet. Effect of the incident surface pressure field on noise due to a turbulent flow past a trailing edge. *Journal of Sound and Vibration*, 57(2):305–306, 1978.

[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] R. Battiti. First and second order methods for learning: Between steepest descent and newton's method. *Neural Computation*, 4(2):141–166, 1992.

[5] E.B. Baum and F. Wilczek. What size net gives valid generalization? *Neural Computation*, 1(1):151–160, 1989.

[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] T.F. Brooks, D.S. Pope, and A.M. Marcolini. Airfoil self-noise and prediction. Technical report, NASA RP-1218, July 1989.

[10] 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.

[11] D. Bucur and G. Buttazzo. *Variational Methods in Shape Optimization Problems*. Birkhauser, 2005.

[12] 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.

[13] Z. Chen and S. Haykin. On different facets of regularization theory. *Neural Computation*, 14(12):2791–2846, 2002.

[14] P. Dadvand, R. Lopez, and E. Oñate. Artificial neural networks for the solution of inverse problems. In *Proceedings of the International Conference on Design Optimisation Methods and Applications ERCOFTAC 2006*, 2006.

[15] P. Davand. Kratos: An object-oriented environment for development of multi-physics analysis software. www.cimne.upc.edu/kratos, 2007.

[16] H. Demuth and M. Beale. *Neural Network Toolbox for Use with MATLAB. User's Gide*. The MathWorks, 2002.

[17] B. Eckel. *Thinking in C++. Second Edition*. Prentice Hall, 2000.

[18] H.W. Engl, M. Hanke, and A. Neubauer. *Regularization of Inverse Problems*. Springer, 2000.

[19] W.E. Failer and S.J. Schrec. Neural networks: Applications and opportunities in aeronautics. *Progress in Aerospace Sciences*, 32:433–456, 1996.

[20] R. Fletcher and C.M. Reeves. Function minimization by conjugate gradients. *Computer Journal*, 7:149–154, 1964.

[21] D.B. Fogel. An introduction to simulated evolutionary optimization. *IEEE Transactions on Neural Networks*, 5(1):3–14, 1994.

[22] I.M. Gelfand and S.V. Fomin. *Calculus of Variations*. Prentice Hall, 1963.

[23] 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.

[24] F. Girosi, M. Jones, and T. Poggio. Regularization theory and neural network architectures. *Neural Computation*, 7(2):219–269, 1995.

[25] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley, 1988.

[26] M.T. Hagan and M. Menhaj. Training feedforward networks with the marquardt algorithm. *IEEE Transactions on Neural Networks*, 5(6):989–993, 1994.

[27] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 1994.

[28] J.D. Hoffman. *Numerical Methods for Engineers and Scientists*. CRC Press, second edition, 2001.

[29] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[30] M.S. Howe. A review of the theory of trailing edge noise. *Journal of Sound and Vibration*, 61:437–465, 1978.

[31] J. Kennedy and R.C. Eberhart. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.

[32] D.E. Kirk. *Optimal Control Theory. An Introduction*. Prentice Hall, 1970.

[33] A. Kirsch. *An Introduction to the Mathematical Theory of Inverse Problems*. Springer, 1996.

[34] Kevin Lau. A neural networks approach for aerofoil noise prediction. Master's thesis, Department of Aeronautics. Imperial College of Science, Technology and Medicine (London, United Kingdom), 2006.

[35] M.J. Lighthill. On sound generated aerodynamically: I general theory. In *Proceedings of the Royal Society of London A 211*, pages 564–587, 1952.

[36] D.P. Lockard and G.M. Lilley. The airframe noise reduction challenge. Technical report, NASA/TM-2004-213013, 2004.

[37] R. Lopez. Flood: An open source neural networks c++ library. www.cimne.com/flood, 2008.

[38] R. Lopez, E. Balsa-Canto, and E. Oñate. Neural networks for the solution of optimal control problems. *EUROGEN 2005 Evolutionary and Deterministic Methods for Design, Optimisation and Control with Applications to Industrial and Societal Problems*, 2005.

[39] 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.

[40] R. Lopez, X. Diego, R. Flores, M. Chiumenti, and E. Oñate. Artificial neural networks for the solution of optimal shape design problems. In *accepted for the 8th World Congress on Computational Mechanics WCCM8*, 2008.

[41] R. Lopez, B. Ducoeur, M. Chiumenti, B. de Meester, and C. Agelet de Saracibar. Modeling precipitation dissolution in hardened aluminium alloys using neural networks. In *accepted for the 11th International Conference on Material Forming ESAFORM 2008*, 2008.

[42] 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.

[43] R. Lopez and E. Oñate. A software model for the multilayer perceptron. In *Proceedings of the International Conference on Applied Computing IADIS 2007*, 2007.

[44] R. Lopez and E. Oñate. An extended class of multilayer perceptron. *Neurocomputing*, In press, 2008.

[45] D.G. Luenberger. *Linear and Nonlinear Programming*. Addison Wesley, 1984.

[46] D.J.C. MacKay. Bayesian interpolation. *Neural Computation*, 4(3):415–447, 1992.

[47] B. Mohammadi and O. Pironneau. Shape optimization in fluid mechanics. *Annual Review of Fluid Mechanics*, 36:255–279, 2004.

[48] M.F. Moller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*, 6:525–533, 1993.

[49] K.W. Morton and D.F. Mayers. *Numerical Solution of Partial Differential Equations, an Introduction*. Cambridge University Press, 2005.

[50] O.R. Myrh and Ø. Grong. Process modelling applied to 6082-t6 aluminium weldments - i. reaction kinetics. *Acta Metallurgica et Materialia*, 39(11):2693–2702, 1991.

[51] O.R. Myrh and Ø. Grong. Process modelling applied to 6082-t6 aluminium weldments - ii. applications of model. *Acta Metallurgica et Materialia*, 39(11):2703–2708, 1991.

[52] 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.

[53] 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.

[54] H. Pohlheim. Geatbx - genetic and evolutionary algorithm toolbox for use with matlab. http://www.geatbx.com, 2007.

[55] M.J.D. Powell. Restart procedures for the conjugate gradient method. *Mathematical Programming*, 12:241–254, 1977.

[56] 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.

[57] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.

[58] H.R. Shercliff, M.J. Russel, A. Taylor, and T.L. Dickerson. Microstructural modelling in friction stir welding of 2000 series aluminium alloys. *Mecanique & Industries*, 6:25–35, 2005.

[59] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer-Verlag, 1980.

[60] B. Stroustrup. *The C++ Programing Language*. Addison Wesley, 2000.

[61] M. Tanaka, editor. *Inverse Problems in Engineering Mechanics IV*. Elsevier, 2003.

[62] A.N. Tikhonov and V.Y. Arsenin. *Solution of ill-posed problems*. Wiley, 1977.

[63] V. Vapnik. *Statistical Learning Theory*. John Wiley and Sons, 1998.

[64] J. Šíma and P. Orponen. General-purpose computation with neural networks: A survey of complexity theoretic results. *Neural Computation*, 15:2727–2778, 2003.

[65] E. W. Weisstein. Mathworld - a wolfram web resource. http://mathworld.wolfram.com, 2007.

[66] 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.

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

[68] O. Zienkiewicz and R. Taylor. *The Finite Element Method, Volumes 1 and 2*. Mc Graw Hill, 1988.

Index